ON THE PATTERN RECOGNITION OF NOISY SUBSEQUENCE TREES[*]

B. J. Oommen[1] and R. K. S. Loke

ABSTRACT

In this paper we consider the problem of recognizing ordered labeled trees by processing their noisy subsequence-trees which are "patched-up" noisy portions of their fragments. We assume that we are given H, a finite dictionary of ordered labeled trees. X^* is an unknown element of H, and U is any arbitrary *subsequence-tree* of X^*. We consider the problem of estimating X^* by processing Y, which is a noisy version of U. The solution which we present is, to our knowledge, the first reported solution to the problem.

We solve the problem by sequentially comparing Y with every element X of H, the basis of comparison being the constrained edit distance between two trees [OL94]. Although the actual constraint used in evaluating the constrained distance can be any arbitrary edit constraint involving the number and type of edit operations to be performed, in this scenario we use a specific constraint which implicitly captures the properties of the corrupting mechanism ("channel") which noisily garbles U into Y. The algorithm which incorporates this constraint has been used to test our pattern recognition system yielding a remarkable accuracy. Experimental results which involve manually constructed trees of sizes between 25 and 35 nodes and which contain an average of 21.8 errors per tree demonstrate that the scheme has about 92.8% accuracy. Similar experiments for randomly generated trees yielded an accuracy of 86.4%.

I. INTRODUCTION

In this paper, we consider the following problem : Suppose we have a finite dictionary of labeled ordered trees, H. Let X^* be any tree from H. U is an *arbitrary* Subsequence-Tree (SuT) of X^* obtained by randomly deleting nodes from it. The resultant tree (called a subsequence-tree or SuT of X^*) is further subjected to substitution, insertion and deletion errors yielding the *Noisy Subsequence-Tree* (NSuT), Y. Our aim is then to identify the original tree, X^*, by processing Y.

To clarify the situation, consider the following example in which X^* are U are the following trees shown in Figure I. Note that X^* has 30 nodes and that U (with 12 nodes) is one of its SuTs obtained by randomly deleting nodes from X^*.

[*] Partially supported by the Natural Sciences and Engineering Research Council of Canada. A preliminary version of this paper can be found in the Proceedings of SSPR-98, the 1998 International Symposium on Syntactic and Statistical Pattern Recognition, Sydney, Australia, August 1998. As the first author, I am very grateful to Prof. Kai Zhang from the University of Western Ontario, London, Ontario, for his input into this paper. Kai and I have written a few papers together and in my reckoning, I believe that his is probably the most authoritative voice on tree and sub-tree editing.

[1]Senior Member IEEE. Address of both authors : School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6. e-mail address : oommen@scs.carleton.ca

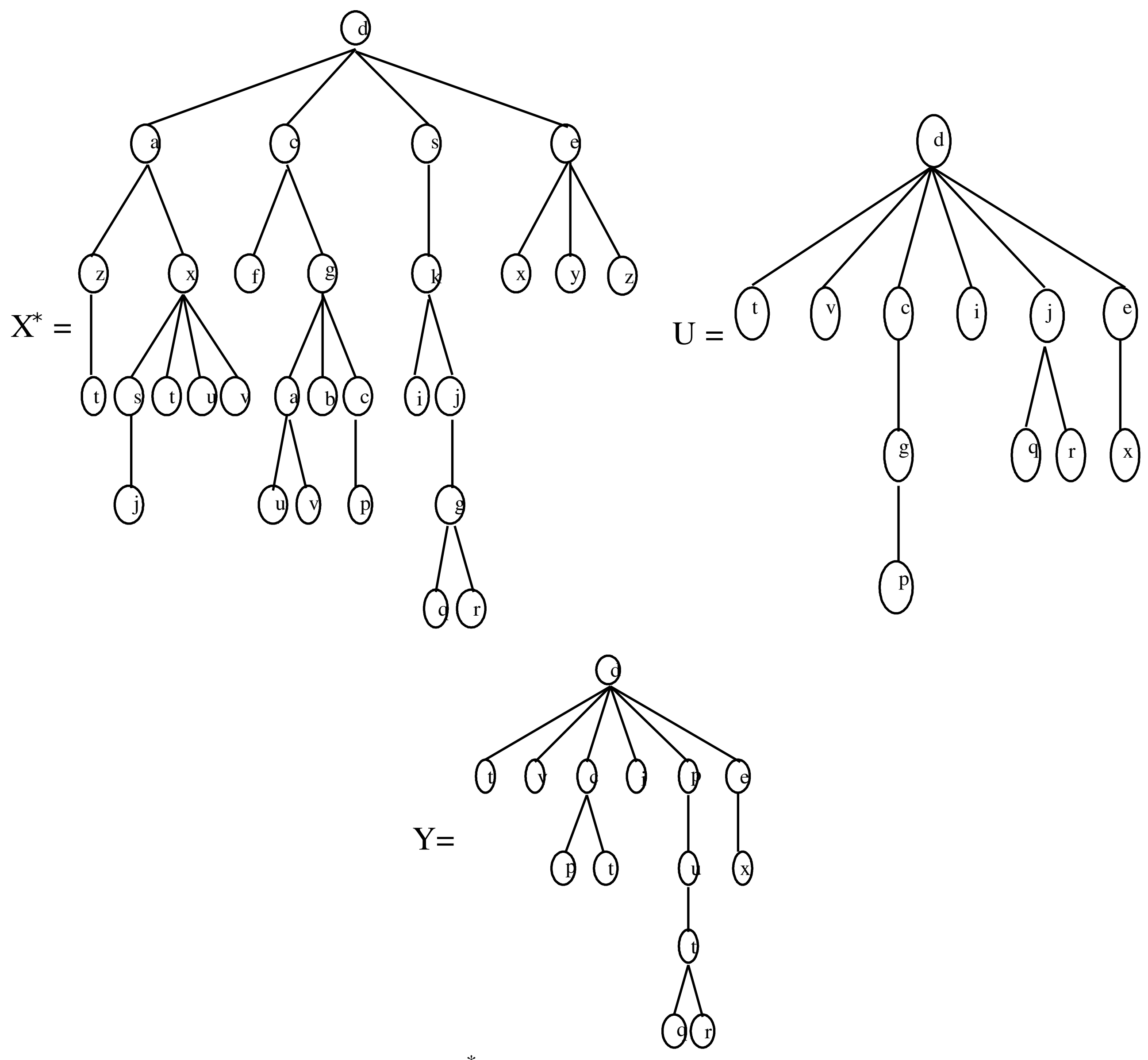


Figure I : An example of a tree X^*, U, one of its Subsequence Trees, and Y which is a noisy version of U. The problem we study involves recognizing X^* from Y.

U, is now noisily garbled and further subjected to additional substitution, insertion and deletion errors on the nodes to produce the NSuT, Y, also given in Figure I. Our aim is to recognize X^* by processing Y.

In this case, between X^* and Y, there is an overall number of 17 substitution, deletion and insertion errors. Although this is a fairly grossly mutated subsequence tree we would like to add that the algorithm presented in this paper has been able to perform Pattern Recognition (PR) in this case, and indeed, in scenarios in which the signal-to-noise ratio is much smaller.

I.1 Tree Editing and the Salient Contributions of this Paper

Unlike the string-editing[2] problem, only few results have been published concerning the tree-editing problem. In 1977 Selkow [Se77, SK83] presented a tree editing algorithm in which insertions and deletions were only restricted to the leaves. Tai [Ta79] in 1979 presented another algorithm in which insertions and deletions could take place at any node within the tree except the root. The algorithm of Lu [Lu79], on the other hand, did not solve this problem for trees of more than two levels. The best known algorithm for solving the general tree-editing problem is the one due to Zhang and Shasha [ZS89]. Also, to the best of our knowledge, in all the papers published till the mid-90's, the literature primarily contains only one numeric inter-tree dissimilarity measure - their pairwise “distance” measured by the minimum cost edit sequence.

The literature on the comparison of trees is otherwise scanty : Zhang [SZ90] has suggested how tree comparison can be done for ordered and unordered labeled trees using tree alignment as opposed to the edit distance utilized elsewhere [ZS89]. The question of comparing trees with variable length don't care edit operations was also recently solved by Zhang *et. al.* [ZSW92]. Otherwise, the results concerning unordered trees are primarily complexity results [ZSS92] - editing unordered trees with bounded degrees is shown to be NP-hard in [ZSS92] and even MAX SNP-hard in [ZJ94].

The most recent results concerning tree comparisons are probably the ones due to Oommen, Zhang and Lee [OZL96]. In [OZL96] the authors defined and formulated an abstract measure of comparison, $\Omega(T_1, T_2)$, between two trees T_1 and T_2 presented in terms of a set of elementary inter-symbol measures $\omega(.,.)$ and two abstract operators. By appropriately choosing the concrete values for these two operators and for $\omega(.,.)$, the measure Ω was used to define various numeric quantities between T_1 and T_2 including (i) the edit distance between two trees, (ii) the size of their largest common sub-tree, (iii) $\text{Prob}(T_2|T_1)$, the probability of receiving T_2 given that T_1 was transmitted across a channel causing independent substitution and deletion errors, and, (iv) the *a posteriori* probability of T_1 being the transmitted tree given that T_2 is the received tree containing independent substitution, insertion and deletion errors.

Unlike the generalized tree editing problem, the problem of comparing a tree with one of its possible subtrees or SuTs has almost not been studied in the literature at all. The only reported results for comparing trees in this setting have involved constrained tree distances[3] [OL94] and indeed, this we will be foundational basis for the PR of noisy SuTs.

The primary contribution of the paper is the application of the constrained tree distance for the NSuT recognition problem. Since Y is a noisy version of a *subsequence tree* of X^*, (and not a noisy version of X^* itself), clearly, just as in the case of recognizing noisy subsequences from strings [Oo87], it

[2]The literature on string editing is extensive. We refer the readers to the book written by Sankoff and Kruskal [SK83, WF74] and the proceedings of the recent symposia on Combinatorial Pattern Matching (CPM) for the state of the art techniques in sequence processing.

[3]The computation of the constrained edit distance was independently developed in [Zh90]. As will be clear later, we shall be using the formulation of [OL94] since we utilize "substitution-based constraints" to recognize of NSuTs.

is meaningless to compare Y with all the trees in the dictionary *themselves* even though they were the potential sources of Y. The fundamental drawback in such a comparison strategy is the fact that significant information was deleted from X^* even before Y was generated, and so Y should rather be compared with every possible subsequence tree of every tree in the dictionary. Clearly, this is intractable, since the number of SuTs of a tree is exponentially large and so an alternative way of comparing Y with every X in H has to be devised. This is achieved by taking into consideration the information about the noise characteristics of the channel which garbles U. Indeed, these characteristics are translated into edit constraints whence a constrained tree editing algorithm can be invoked to perform the classification. This is the fundamental contribution of this paper.

Besides these, our paper suggests a new perspective for generalized computation models. In [Oo87] we had devised an algorithm for the recognition of noisy subsequence from strings and this was achieved by evaluating the inter-string constrained edit distance. The reader will soon observe that the results of this paper are not mere extensions of the string editing problem. This is because, unlike in the case of strings, the topological structure of the underlying graph *prohibits* the two-dimensional generalizations of the corresponding computations. Indeed, inter-tree computations require the simultaneous maintenance of *meta-tree* considerations represented as the parent and sibling properties of the respective trees, which are completely ignored in the case of linear structures such as strings. This further justifies the intuition that not all "string properties" generalize naturally to their corresponding "tree properties", as will be clarified later.

I.2 Applications of the Results

Apart from its academic "pioneering" significance, the problem studied has also applications in a variety of fields. First of all, the significance of the results in structural/syntactic pattern recognition cannot be overstated. Typically, when the pattern to be recognized is inherently a "two-dimensional" structure, it cannot be adequately represented using a one-dimensional (string or circular string) approximation. By representing the pattern as a tree and by utilizing tree comparison algorithms one can, generally speaking, achieve excellent recognition strategies. Indeed, such schemes have been utilized in PR in areas such as clustering [Lu79, Lu84] and waveform correlation [CL85]. However, when the pattern to be recognized is occluded and only noisy information of a *fragment* of the pattern is available, the problem encountered can be perceived as one of recognizing a tree by processing the information in one of its noisy subtrees or subsequence trees. We thus believe that the results presented here will have potential applications in **all** the areas of pattern recognition where either the modeling or the knowledge representation involves trees. Application domains therefore are numerous, and include those which involve classification trees, modeling using tree structures, scene analysis and the processing of search trees.

The other major application of these principles would be in the processing of ribonucleic acid (RNA) secondary structures and their tree representations [KM91, LON89, LNM89, SZ90, Sh88, Ta79, TSSS87, ZJ94, ZSS92, Zh90, ZS89]. A molecule of RNA is made up of a long sequence of subunits (the ribonucleotides (RN)) which are linked together. Each ribonucleotide contains one of the four possible bases, abbreviated by A, C, G, and U. This base sequence is called the primary structure of the RNA molecule. Under natural conditions, an RN sequence twists and bends and the bases form bonds with one another to yield complicated patterns. The latter bonding pattern is called its secondary structure. Research in this field has shown that similar *structures* have similar functionality and the use of sequence comparison by itself is inadequate for determining the structural homology [SZ90]. Thus, the problem reduces to comparing RNA secondary structures.

If one examines a typical secondary structure (see [SZ90]) it becomes apparent that such a structure may be represented as a tree. This representation was proposed by Shapiro and Zhang [SZ90, Sh88] using node values such as M, H, I, B, R and N (for Multiple loop, Hairpin loop, Internal loop, Bulge loop, helical stem Region, and exterNal single-stranded region respectively). Since this representation only considers the topology of the loops and stem regions, a more dissected representation would have to also consider the sizes of the loops and the helical stems.

The comparison of RNA secondary structure trees can help identify conserved structural motifs in an RNA folding process [LON89, LNM89] and construct taxonomy trees [SZ90]. In all such molecular biological domains the algorithm proposed here can be used to recognize (classify) RNA secondary structure trees by merely processing noisy (garbled) versions of their subsequence trees. We believe that this could assist the biologist trace proteins when only their fragments are available for examination[4].

We conclude this sub-section by observing that studies in compiler construction have also used tree-comparison algorithms in automatic error recovery and correction of programming languages [Ta79]. Indeed, with no loss of generality we believe that the results of this paper can be used in any problem domain involving the comparison of tree-patterns with other tree-patterns representing a noisy sub-pattern which has been "occluded" at multiple junctures.

The proofs of the major theoretical results claimed here are given in [OL94]. In the interest of brevity they will be omitted here except when the proofs can be useful for the *implementation* of the respective algorithms.

II. NOTATIONS AND DEFINITIONS

II.1 Notation

Let N be an alphabet and N^* be the set of trees whose nodes are elements of N. Let μ be the null tree, which is distinct from λ, the null label not in N. $N = N \cup \{\lambda\}$. A tree $T \in N^*$ with M nodes is said to

[4] Since we do not have access to such data, we welcome help and input from researchers who are willing to collaborate with us. Hopefully, this could lead to joint research endeavours/projects and results.

be of size $|T|=M$, and will be represented in terms of the postorder numbering of its nodes. The advantages of this ordering are catalogued in [ZS89].

Let T[i] be the i^{th} node in the tree according to the left-to-right postorder numbering, and let $\delta(i)$ represent the postorder number of the leftmost leaf descendant of the subtree rooted at T[i]. Note that when T[i] is a leaf, $\delta(i) = i$. 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 Tree(i). Size(i) is the number of nodes in Tree(i). The father of i is denoted as f(i). If $f^0(i) = i$, the node $f^k(i)$ can be recursively defined as $f^k(i) = f(f^{k-1}(i))$. The set of ancestors of i is : $\text{Anc}(i) = \{f^k(i) \mid 0 \le k \le \text{Depth}(i)\}$.

II.2 Elementary Edit Operations and Sub-Trees

An edit operation on a tree is either an insertion, a deletion 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 label or λ, the null label. $x = \lambda$ 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 of $x = \lambda$ and $y = \lambda$ has not been defined -- it is not needed. The formal definitions of these follow.

Insertion of node x into tree T : Node x will be inserted as a son of some node u of T. It may either be inserted with no sons or take as sons any subsequence of the sons of u. If u has sons $u_1,u_2,..,u_k$, then for some $0 \le i \le j \le k$, node u in the resulting tree will have sons $u_1,...,u_i$, x, $u_j,...,u_k$, and node x will have no sons if $j = i+1$, or else have sons $u_{i+1},...,u_{j-1}$. This is shown in Figure II.

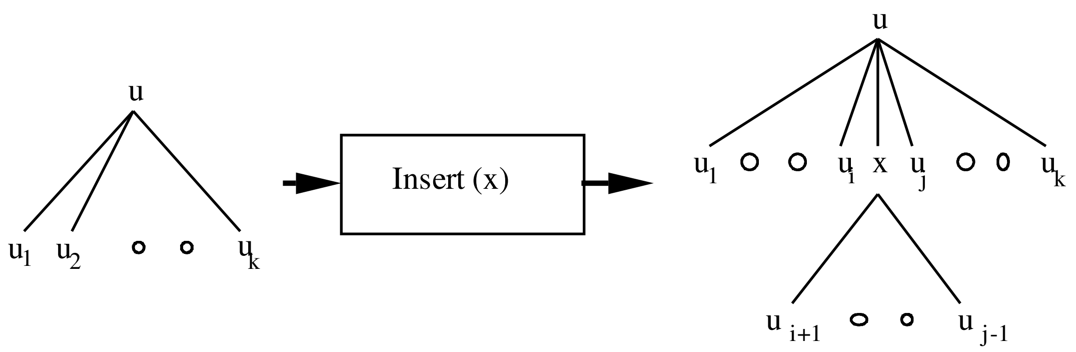


Figure II : An example of the insertion of a node.

Deletion of node y from a tree T : If node y has sons $y_1,y_2,...,y_k$ and node u, the father of y, has sons $u_1,u_2,...,u_j$ with $u_i=y$, then node u in the resulting tree obtained by the deletion will have sons $u_1,u_2,...,u_{i-1},y_1,y_2,...,y_k,u_{i+1},...,u_j$. See Figure III.

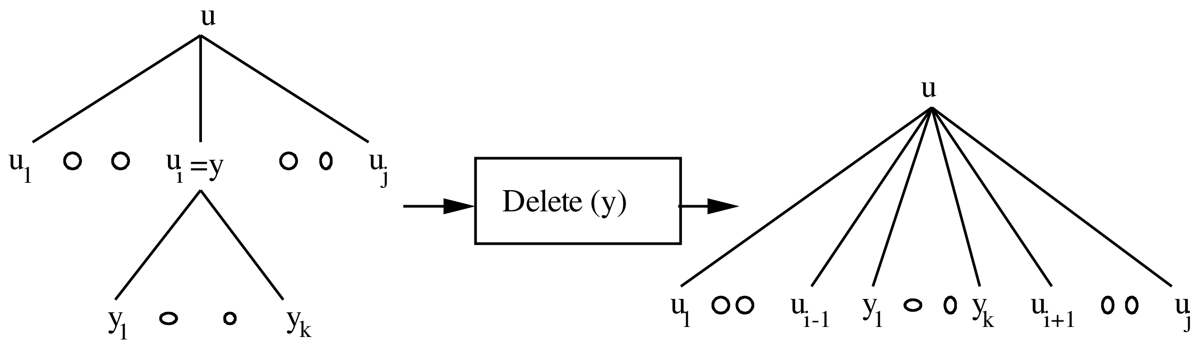

u
u_1 o o u_i=y o o u_j
y_1 o o y_k
Delete (y)
u
u_1 oo u_{i-1} y_1 o o y_k u_{i+1} o o u_j

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

Substitution of node x by node y in T : Node y in the resulting tree will have the same father and sons as node x in the original tree. This is shown in Figure IV.

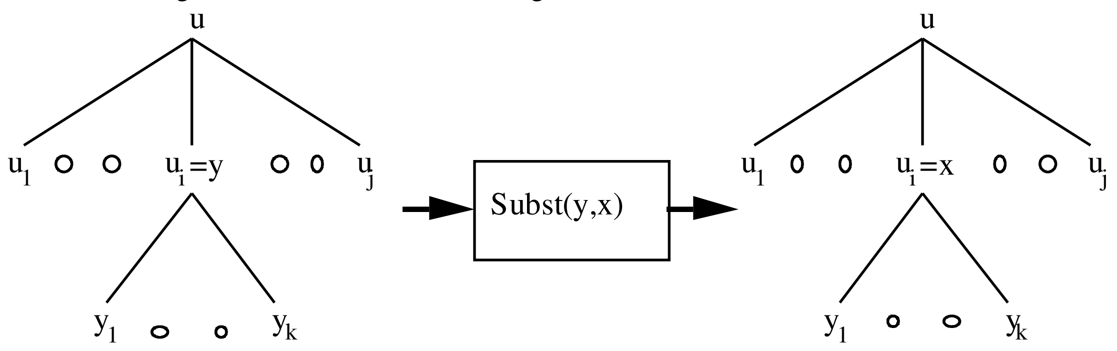


Figure IV : An example of the substitution of a node by another.

Let $d(x,y) \geq 0$ be the cost of transforming node x to node y. If $x \neq \lambda \neq y$, d(x,y) will represent the cost of substitution of node x by node 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. We assume that :

$$d(x,y) \geq 0;\ d(x,x) = 0; \qquad (1)$$

$$d(x,y) = d(y,x); \qquad (2)$$

and

$$d(x,z) \leq d(x,y) + d(y,z) \qquad (3)$$

where (3) is essentially a "triangular" inequality constraint.

Let S be a sequence $s_1, ..., s_k$ of edit operations. An S-derivation from A to B is a sequence of trees $A_0, ..., A_k$ such that $A = A_0$, $B = A_k$, and $A_{i-1} \rightarrow A_i$ via s_i for $1 \leq i \leq k$. We extend the inter-node edit distance d(.,.) to the sequence S by assigning :

$$W(S) = .$$

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

$$D(T_1,T_2) = \text{Min}\left\{W(S) \mid S \text{ is an S-derivation transforming } T_1 \text{ to } T_2\right\}.$$

It is easy to observe that :

$$D(T_1,T_2) \leq d(T_1[|T_1|],T_2[|T_2|]) + +$$

II.3 Mappings between Trees

A Mapping is a description of how a sequence of edit operations transforms T_1 into T_2. A pictorial representation of a mapping is given in Figure V. Informally, in a mapping the following holds :

(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) $1 \le i \le |T_1|$, $1 \le j \le |T_2|$;

(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.

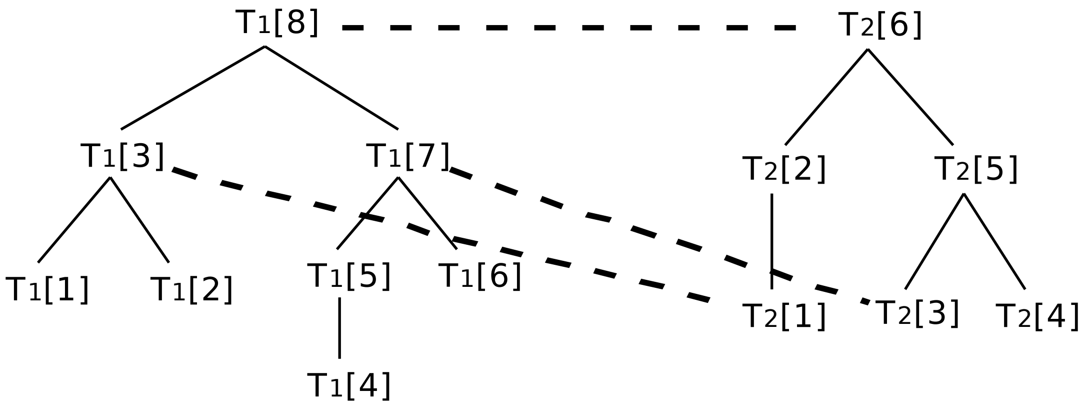


Figure V : An example of a mapping between two labeled ordered trees.

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

cost (M) = + + .

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 I.

Given S, an S-derivation $s_1, ..., s_k$ of edit operations from T_1 to T_2, there exists a mapping M from T_1 to T_2 such that cost (M) $\le$ W(S). Conversely, for any mapping M, there exists a sequence of editing operations such that W(S) = cost (M).

Proof : Same as the proof of Lemma 2 in [ZS89]. •••

Due to the above lemma, we obtain

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

Thus, to search for the minimal cost edit sequence we need to only search for the optimal mapping.

III. EDIT CONSTRAINTS

In this section we shall consider how we can specify the editing of 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 sequence. Some examples of constrained editing are presented below :

(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 ?

The method which we use for specifying arbitrary edit constraints is analogous (but not identical) to the one shown in [Oo86] and is specified by a constraint set, τ. After describing it we shall discuss the definition and computation of $D_\tau(T_1, T_2)$, the distance between T_1 and T_2 subject to this constraint.

Consider the problem of editing T_1 to T_2, where $|T_1| = N$ and $|T_2| = M$. Editing a postorder-forest of T_1 into a postorder-forest of T_2 using exactly i insertions, e deletions, and s substitutions, corresponds to editing $T_1[1..e+s]$ into $T_2[1..i+s]$. 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 :

$$\max\{0,M-N\} \le i \le q \le M,$$

$$0 \le e \le r \le N,$$

$$0 \le s \le R.$$

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

$$H_i = \{j \mid \max\{0,M-N\} \le j \le M\},$$

$$H_e = \{j \mid 0 \le j \le N\}, \text{ and,}$$

$$H_s = \{j \mid 0 \le j \le \text{Min}\{M,N\}\}.$$

H_i, H_e, and H_s are called the set of *permissible values* of i, e, and s.

Theorem I specifies the feasible triples for editing $T_1[1..r]$ to $T_2[1..q]$.

Theorem I.

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 $\{(q-s, r-s, s) \mid 0 \le s \le \text{Min}\{M,N\}\}$.

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. But, the number of substitutions can take any value from 0 to 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. The result follows. •••

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 above. Thus, to edit T_1 to T_2 performing no more than k deletions, the sets Q_s and Q_i are both ϕ, the null set, and $Q_e = \{j \mid j \in H_e, j \le k\}$. Similarly, to edit T_1 to T_2 performing k_i insertions, k_e deletions, and k_s substitutions yields $Q_i = \{k_i\} \cap H_i$, $Q_e = \{k_e\} \cap H_e$, and $Q_s = \{k_s\} \cap H_s$.

Theorem II.

Every edit constraint specified for the process of editing T_1 to T_2 is 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 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 1 requires that N-j substitutions be performed. Let

$$Q_e^* = \{N\text{-}j \mid j \in Q_e\}, \text{ and}$$
$$Q_i^* = \{M\text{-}j \mid j \in Q_i\}.$$

Thus, for any constraint, the number of substitutions permitted is $Q_s \cap Q_e^* \cap Q_i^* \subseteq H_s$. •••

To clarify matters, consider the trees T_1 and T_2 shown in Figure V. Let us suppose that we want to transform T_1 to T_2 by performing at most 5 insertions, at least 3 substitutions and the number of deletions being 3. Then

$$Q_i = \{0,1,2,3,4,5\}, Q_e = \{3\}, \quad \text{and } Q_s = \{3,4,5,6\}.$$

From these we see that

$$Q_e^* = \{5\}, \text{ and } Q_i^* = \{1,2,3,4,5,6\} \text{ yielding, } \tau = Q_s \cap Q_e^* \cap Q_i^* = \{5\}.$$

Hence, the optimal transformation must contains **exactly** 5 substitutions.

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 $\tau = \phi$, the null set. We now consider the computation of $D_\tau(T_1,T_2)$.

IV. CONSTRAINED TREE EDITING

Since edit constraints can be written as unique subsets of H_s, we 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 Const_F_Wt($T_1[i'..i]$,$T_2[j'..j]$,s) or more precisely by Const_F_Wt([i'..i],[j'..j],s). The distance between

$T_1[1..i]$ and $T_2[1..j]$ subject to this constraint is given by Const_F_Wt(i,j,s) since the starting index of both trees is unity. As opposed to this, the distance between the subtree rooted at i and the subtree rooted at j subject to the same constraint is given by Const_T_Wt(i,j,s). The difference between Const_F_Wt and Const_T_Wt is subtle. Indeed,

$$\text{Const\_T\_Wt}(i,j,s) = \text{Const\_F\_Wt}(T_1[\delta(i)..i],T_2[\delta(j)..j],s).$$

These weights obey the following properties proved in [OL94].

Lemma II

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) $\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],\mu,0) = \text{Const\_F\_Wt}(T_1[\delta(i_1)..i\text{-}1],\mu,0) + d(T_1[i],\lambda).$

(iii) $\text{Const\_F\_Wt}(\mu,T_2[\delta(j_1)..j],0) = \text{Const\_F\_Wt}(\mu,T_2[\delta(j_1)..j\text{-}1],0) + d(\lambda,T_2[j]).$

(iv) $\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],0)$

$= \text{Min}$

(v) $\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],\mu,s) = \infty$ if $s > 0$.

(vi) $\text{Const\_F\_Wt}(\mu,T_2[\delta(j_1)..j],s) = \infty$ if $s > 0$.

(vii) $\text{Const\_F\_Wt}(\mu,\mu,s) = \infty$ if $s > 0$.

Proof : The proofs are found as the proofs of Lemmas IIa and IIb of [OL94]. •••

Lemma II essentially states the properties of the constrained distance when either s is zero or when either of the trees is null. These are thus "basis" cases that can be used in any recursive computation. For the non-basis cases we consider the scenarios when the trees are non-empty and when the constraining parameter, s, is strictly positive. Theorem III gives the recursive property of Const_F_Wt in such a case.

Theorem III.

Let $i_1 \in \text{Anc}(i)$ and $j_1 \in \text{Anc}(j)$. Then

$\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s)$

$= \text{Min}$

Proof :

The proof of the theorem is in [OL94]. Since it is central to our algorithm we sketch it below.

We intend to find a minimum cost mapping M between $T_1[\delta(i_1)..i]$ and $T_2[\delta(j_1)..j]$ using exactly s substitutions. 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 Const_F_Wt(.,.,.) remains unchanged, we have :

$\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s) = \text{Const\_F\_Wt}(T_1[\delta(i_1)..i\text{-}1],T_2[\delta(j_1)..j],s) + d(T_1[i],\lambda).$

(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 Const_F_Wt(.,.,.) remains unchanged, we have :

$\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s) = \text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j-1],s) + d(\lambda,T_2[j])$.

(iii) Consider the 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 $\delta(i_1) \le h \le \delta(i)-1$, 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 $k \le j$. So h has to equal to i. By symmetry, k must equal j, so $(i,j)\in M$.

Now, by the ancestor property of M (please refer to [OL94] for the details of this argument), any node in the subtree rooted at $T_1[i]$ can only be touched by a node in the subtree rooted at $T_2[j]$. Since exactly s substitutions must be performed in this transformation, the total number of substitutions used in the sub-transformation from $T_1[\delta(i_1)..\delta(i)-1]$ to $T_2[\delta(j_1)..\delta(j)-1]$ 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 $T_1[i] \rightarrow T_2[j]$). If s_2-1 is the number of substitutions used in the sub-transformation from $T_1[\delta(i)..i-1]$ to $T_2[\delta(j)..j-1]$, s_2 can take any value between 1 to Min{Size(i),Size(j),s}. Hence,

$\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s)$

$$=$$

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. •••

Theorem III naturally leads to a recursive algorithm, except that its time and space complexities will be prohibitively large. The main drawback with using Theorem III is that when substitutions are involved, the quantity $\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s)$ between the forests $T_1[\delta(i_1)..i]$ and $T_2[\delta(j_1)..j]$ is computed using the Const_F_Wts of the forests $T_1[\delta(i_1)..\delta(i)-1]$ and $T_2[\delta(j_1)..\delta(j)-1]$ and the Const_F_Wts of the remaining forests $T_1[\delta(i)..i-1]$ and $T_2[\delta(j)..j-1]$. If we note that, under certain conditions, the removal of a sub-forest leaves us with an entire tree, the computation is simplified. Thus, if $\delta(i)=\delta(i_1)$ and $\delta(j)=\delta(j_1)$ (i.e., i and i_1, and j and j_1 span the same subtree), the subforests from $T_1[\delta(i_1)..\delta(i)-1]$ and $T_2[\delta(j_1)..\delta(j)-1]$ do not get included in the computation. If this is not the case, the $\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s)$ can be considered as a combination of the $\text{Const\_F\_Wt}(T_1[\delta(i_1)..\delta(i)-1], T_2[\delta(j_1)..\delta(j)-1],s-s_2))$ and the tree weight between the trees rooted at i and j respectively, which is $\text{Const\_T\_Wt}(i,j,s_2)$. This is proved below.

Theorem IV.

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

$\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s)$

$= \text{Min}$

otherwise,

$\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s)$

$= \text{Min}$

Sketch of Proof :

By Theorem III, if $\delta(i) = \delta(i_1)$ and $\delta(j) = \delta(j_1)$, the forests $T_1[\delta(i_1)..\delta(i)-1]$ and $T_2[\delta(j_1)..\delta(j)-1]$ are both empty. Thus,

$\text{Const\_F\_Wt}(T_1[\delta(i_1)..\delta(i)-1], T_2[\delta(j_1)..\delta(j)-1],s-s_2) = \text{Const\_F\_Wt}(\mu,\mu,s-s_2)$

which is equal to zero if $s_2 = s$, or is equal to ∞ if $s_2 < s$. The first part of the theorem follows.

For the second part, using arguments given in [OL94] we see that:

$$\text{Const\_F\_Wt}(T_1[\delta(i_1)..i],T_2[\delta(j_1)..j],s) \leq \text{Const\_F\_Wt}(T_1[\delta(i_1)..\delta(i)-1],T_2[\delta(j_1)..\delta(j)-1],s-s_2) + \text{Const\_T\_Wt}(i,j,s_2).$$

Analogously we can show that :

$$\text{Const\_T\_Wt}(i,j,s_2) \leq \text{Const\_F\_Wt}(T_1[\delta(i)..i-1],T_2[\delta(j)..j-1],s_2-1) + d(T_1[i],T_2[j]).$$

Theorem III and these two inequalities justify replacing $\text{Const\_T\_Wt}(i,j,s_2)$ for the corresponding Const_F_Wt expressions, and the result follows. The details of the proof are found in [OL94]. •••

Theorem IV suggests that we can use a dynamic programming flavored algorithm to solve the constrained tree editing problem. The second part of Theorem IV suggests that to compute $\text{Const\_T\_Wt}(i_1,j_1,s)$ we have to have available 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 $\delta(j_1)$. The theorem also 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-product in the process of computing the Const_F_Wt between the trees rooted at i_1 and j_1. These distances are obtained as a by-product because, if the forests are trees, Const_F_Wt is retained as a Const_T_Wt. 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 is called the set of Essential_Nodes, and these are merely those nodes for which the computation would involve the second case of Theorem IV as opposed to the first. We define the set Essential_Nodes of tree T as :

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

Observe that 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[5]. Thus, the Const_T_Wt can be computed for the entire tree if Const_T_Wt of the Essential_Nodes are computed.

The nodes in Essential_Nodes will be exactly the same nodes contained in the set LR_keyroots set defined in [ZS89]. Although these sets are identical, the implication of a node being in the sets is different. In [ZS89] $i \in \text{LR\_keyroots}[T_1]$ and $j \in \text{LR\_keyroots}[T_2]$ implies that the corresponding tree

[5]For example, for the trees in Figure V, $\text{Essential\_Nodes}(T_1) = \{2,6,7,8\}$ and $\text{Essential\_Nodes}(T_2) = \{4,5,6\}$.

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 **constrained** tree weights rooted at these trees need precomputation *for all feasible values* of s which are relevant.

Using Theorem IV we now develop a bottom-up approach for computing the Const_T_Wt between all pairs of subtrees. Note that the function δ() and the set Essential_Nodes() can be computed in linear time. We assume that these are stored in arrays δ [] and Essential_Nodes [] respectively. Furthermore, we assume that the elements in Essential_Nodes [] are sorted in the postorder representation.

```
ALGORITHM T_Weights
Input :           Trees $T_1$ and $T_2$ and the set of elementary edit distances.
Output :          Const_T_Wt(i, j, s), $1 \le i \le |T_1|$, $1 \le j \le |T_2|$, and $1 \le s \le$ Min{$|T_1|$, $|T_2|$}.
Assumption :      Preprocess ($T_1$,$T_2$) yields the δ[] and Essential_Nodes [] arrays for both trees.
                  These quantities are assumed to be global.
BEGIN
        Preprocess ($T_1$,$T_2$) ;
        For i' = 1 to | Essential_Nodes$_1$[]| Do
            For j' = 1 to | Essential_Nodes$_2$[]| Do
                i = Essential_Nodes$_1$ [i'];
                j = Essential_Nodes$_2$ [j'];
                Compute_Const_T_Wt(i,j);
            EndFor
        EndFor
END ALGORITHM T_Weights.
```

We shall now compute Const_T_Wt(i, j, s) and store it in a **permanent** three-dimensional array Const_T_Wt. From Theorem IV, we observe that to compute the quantity Const_T_Wt(i, j, s) the quantities which are involved are precisely the terms Const_F_Wt([δ(i)..h], [δ(j)..k], s') 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, $\delta(i) \le h \le i$, $\delta(j) \le k \le j$, and where s' is in the set of feasible values and satisfies $0 \le s' \le s =$ Min {|Tree$_1$(i)|, |Tree$_2$(j)|}. Our intention is store **these** values using a single **temporary** three-dimensional array Const_F_Wt [.,.,.]. But in order to achieve this, it is clear that the base indices of the temporary three-dimensional array Const_F_Wt [.,.,.] will have to be adjusted each time the procedure is invoked, so as to utilize 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 Const_F_Wt [.,.,.] can be used to store the values in each phase of the computation by assigning for all $1 \le x_1 \le i-\delta(i)+1$, $1 \le y_1 \le j-\delta(j)+1$:

Const_F_Wt [x_1,y_1,s'] = Const_F_Wt([δ(i)..δ(i)+x_1-1], [δ(j)..δ(j)+y_1-1], s').

Consequently, we note that for every x_1, y_1, and s' in any intermediate step in the algorithm, the quantity Const_T_Wt() that has to be stored in the permanent array can be obtained by incorporating these base values again, and has the form Const_T_Wt [x_1+b_1, y_1+b_2, s']. The algorithm is formally given below, and its correctness is proven in detail in [OL94]. It is omitted here in the interest of brevity.

```
ALGORITHM Compute_Const_T_Wt
Input :     Indices i, j and the quantities assumed global in T_Weights.
Output :    Const_T_Wt [$i_1$, $j_1$, s],  $\delta_1(i) \le i_1 \le i$, $\delta_2(j) \le j_1 \le j$, $0 \le s \le$ Min{Size(i),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 = 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] */
   Const_F_Wt [0][0][0] = 0;         /* Initialize Const_F_Wt */
   For $x_1$ = 1 to N Do
      Const_F_Wt [$x_1$][0][0] = Const_F_Wt [$x_1$-1][0][0] + d($T_1$[$x_1$+$b_1$]$\rightarrow\lambda$);
      Const_T_Wt [$x_1$+$b_1$][0][0] = Const_F_Wt [$x_1$][0][0];
   EndFor
   For $y_1$ = 1 to M Do
      Const_F_Wt [0][$y_1$][0] = Const_F_Wt [0][$y_1$-1][0] + d($\lambda\rightarrow T_2$[$y_1$+$b_2$]);
      Const_T_Wt [0][$y_1$+$b_2$][0] = Const_F_Wt [0][$y_1$][0];
   EndFor
   For s = 1 to R Do
      Const_F_Wt [0][0][s] = $\infty$;
      Const_T_Wt [0][0][s] = Const_F_Wt [0][0][s];
   EndFor
   For  $x_1$ = 1 to N Do
      For  $y_1$ = 1 to M Do
         Const_F_Wt [$x_1$][$y_1$][0] = Min
         Const_T_Wt [$x_1$+$b_1$][$y_1$+$b_2$][0] = 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
                    $Const\_T\_Wt[x_1+b_1][y_1+b_2][s] = Const\_F\_Wt[x_1][y_1][s]$;
                Else
                    $Const\_F\_Wt[x_1][y_1][s]$
                        = Min
                        where  $a = \delta_1(x_1+b_1) - 1 - b_1$, $b = \delta_2(y_1+b_2) - 1 - b_2$,
                               $c = Size(x_1+b_1)$ and $d = Size(y_1+b_2)$.
                EndIf
            EndFor
        EndFor
    EndFor
END ALGORITHM Compute_Const_T_Wt
```

As a result of invoking Algorithm T_Weights (which, in turn, repeatedly invokes Algorithm Compute_Const_T_Wt for all pertinent values of i and j) we will have computed the constrained inter-tree edit distance between T_1 and T_2 subject to the constraint that the number of substitutions performed is s, for all feasible substitutions. Our algorithm is similar in spirit to the one independently reported in [Zh90] although the latter, just as in [Oo86], deals with the problem by utilizing the number of insertions permitted as the "free" variable. In our case, however, we have chosen to use the number of substitutions as the free variable. This makes it differ from the philosophies of [Oo86] and [Zh90], but helps us retain the underlying principles of tree editing as in [ZS89], in which the substitution operation has to be handled differently from the other operations.

The space required by the above algorithm is obviously $O(|T_1| * |T_2| * \text{Min}\{|T_1|, |T_2|\})$.

If Span(T) is the Min{Depth(T), Leaves(T)}, the algorithm's time complexity is [OL94] :

$$O(|T_1| * |T_2| * (\text{Min}\{|T_1|, |T_2|\})^2 * \text{Span}(T_1) * \text{Span}(T_2)).$$

V. The Noisy Subsequence Tree Recognition Problem

V.1 Principles Used in Solving the Noisy Subsequence-Tree Recognition Problem

Using the foundational concepts of constrained edit distances explained in the previous sections, we are now in a position to present our solution to the Noisy Subsequence Tree (NSuT) Recognition Problem. The assumptions made in the recognition process are quite straightforward. First of all we assume that a "Transmitter" intends to transmit a tree X^* which is an element of a finite dictionary of trees, H. However, rather than transmitting the original tree he opts to randomly delete nodes from X^* and transmit one of its subsequence trees, U. The transmission of U is across a noisy channel which is capable of introducing substitution, deletion and insertion errors at the nodes. Note that, to render the problem meaningful (and distinct from the uni-dimensional one studied in the literature) we assume that the tree **itself** is transmitted as a two dimensional entity. In other words we do not consider the serialization of this transmission process, for that would merely involve transmitting a string representation, which would, typically, be a traversal pre-defined by both the Transmitter and the Receiver. The receiver receives Y, a noisy version of U. We now show how we recognize X^* from Y.

To render the problem tractable, we assume that *some of* the properties of the channel can be observed. More specifically, we assume that L, the expected number of substitutions introduced in the process of transmitting U, can be estimated. In the simplest scenario (where the transmitted nodes are either deleted or substituted for) this quantity is obtained as the expected value for a mixture of Bernoulli trials, where each trial records the success of a node value being transmitted as an non-null symbol.

Since U can be an arbitrary subsequence tree of X^*, it is obviously meaningless to compare Y with every $X \in H$ using any known **unconstrained** tree editing algorithm. Clearly, before we compare Y to the individual tree in H, we have to use the additional information obtainable from the noisy channel. Also, since the specific number of substitutions (or insertions/deletions) introduced in any *specific* transmission is unknown, it is reasonable to compare any $X \in H$ and Y subject to the constraint that the number of substitutions that actually took place is its best estimate. Of course, in the absence of any other information, the best estimate of the number of substitutions that could have taken place is indeed its expected value, L. One could therefore use the set {L} as the constraint set to effectively compare Y with any $X \in H$. Since the latter set can be quite restrictive, we opt to use a constraint set which is a superset of {L} marginally larger than {L}. Indeed, the superset contains merely the neighbouring values, and is {L-1, L, L+1}. Since the size of the set is still a constant, there is no significant increase in the computation times. This is exactly the rationale for our recognition algorithm given below.

```
Algorithm RecognizeSubsequenceTrees
Input:   1. The finite dictionary H and L, the expected number of substitutions that can take place
            per transmission. This may or may not be tree-dependent.
         2. Y, a noisy version of a subsequence tree of an unknown $X^*$ in H.
Output: The estimate $X^+$ of $X^*$. If L is not a feasible value $L_p$ is the closest feasible integer.
BEGIN
         For every tree $X \in H$ do
         Begin
             If L is a feasible value then
                 $L_p = L$
             Else
                 $L_p$ = closest feasible integer to L
             EndIf
             $\tau = \{L_p-1, L_p, L_p+1\}$
             Compute $D_\tau(X,Y)$ using Algorithm Constrained_Tree_Distance
         End
         $X^+$ is the tree minimizing $D_\tau(X,Y)$
END Algorithm RecognizeSubsequenceTrees
```

Observe that Algorithm RecognizeSubsequenceTrees assumes that we can compute the constrained distance subject to a specified constraint set, τ. Since τ is fully defined in terms of the number of substitutions required in the comparison, all the information required for the comparison will be available in the array Const_T_Wt [.,.,.] computed using Algorithm T_Weights. Thus, after the array Const_T_Wt [.,.,.] is computed, the distance $D_\tau(T_1,T_2)$ between T_1 and T_2 subject to the constraint τ can be directly evaluated using the following ALGORITHM Constrained_Tree_Distance given below, which essentially minimizes Const_T_Wt over all the values of 's' found in the constraint set.

```
ALGORITHM Constrained_Tree_Distance
Input :      The array Const_T_Wt [.,.,.] computed using Algorithm T_Weights.
Output :     The constrained distance $D_\tau(T_1,T_2)$ with $\tau = \{L_p-1, L_p, L_p+1\}$.
BEGIN
        $D_\tau(T_1,T_2) = \infty$;
        For all $s = L_p-1, L_p, L_p+1$ 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.
```

V.2 Experimental Results

The technique developed in the previous sections was rigorously tested to verify its capability in the pattern recognition of NSuTs. The experiments conducted were for two different data sets which were artificially generated. Our primary intention was to test the schemes on real-life molecule structures. But since these structures can be thousands of characters long and could require computationally prohibitive time, we have resorted to using "relatively long" character sequences using benchmark results involving keyboard character errors. Clearly, these results are sufficient to demonstrate the power of the strategy to recognize noisy subsequence trees. Although it would be a rather trivial exercise to obtain equivalent results for biological molecules, we are unable to currently proceed with such experiments because we do not have the "confusion matrix" for amino-acids in such a setting. This work is currently being pursued, and so we have no results to report for real-life structures.

The results we have obtained for simulated trees are remarkable. As mentioned earlier, to our knowledge, these are the first reported results that demonstrate that **a tree can indeed be recognized by processing the information resident in one of its noisy random subsequence trees**. The details of the experimental set-ups and the results obtained follow.

V.2.1 Tree Representation

In the implementation of the algorithm we have opted to represent the tree structures of the patterns studied as parenthesized lists in a post-order fashion. Thus, a tree with root 'a' and children B, C and D is represented as a parenthesized list L = (B C D 'a') where B, C and D can themselves be trees in which cases the embedded lists of B, C and D are inserted in L. A specific example of a tree (taken from our dictionary) and its parenthesized list representation is given in Figure VI below.

V.3 Experiment Setup for Data Set A

In our first experimental set-up the dictionary, H, consisted of 25 *manually* constructed trees which varied in sizes from 25 to 35 nodes. An example of a tree in H is given in Figure VI above. To generate a NSuT for the testing process, a tree X^* (unknown to the classification algorithm) was chosen. Nodes from X^* were first randomly deleted producing a subsequence tree, U. In our experimental set-up the probability of deleting a node was set to be 60%. Thus although the average size of each tree in the dictionary was 29.88, the average size of the resulting subsequence trees was only 11.95.

The garbling effect of the noise was then simulated as follows. A given subsequence tree U, was subjected to additional substitution, insertion and deletion errors, where the various errors deformed the trees as described in Section II.1. This was effectively achieved by passing the string representation through a channel causing substitution, insertion and deletion errors analogous to the one used to generate the noisy subsequences in [Oo87] and which has recently been formalized in [OK98]. However, as opposed to merely mutating the string representations as in [OK98] the reader should observe that we are manipulating the underlying *list* representation of the *tree*. This involves ensuring the maintenance of the parent/sibling consistency properties of a tree - which are far from trivial.

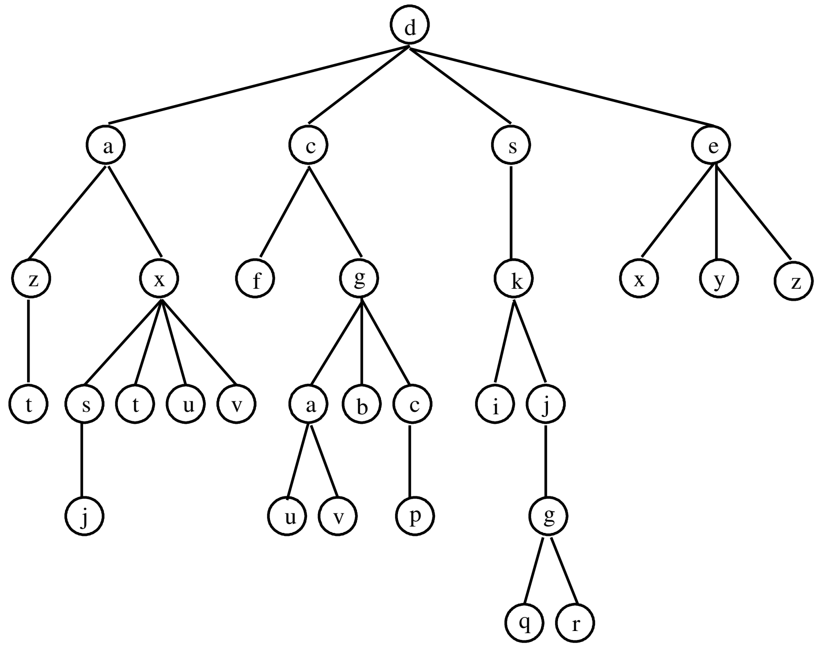


Figure VI: A tree from the finite dictionary H. Its associated list representation is as follows: ((((t)z)(((j)s)(t)(u)(v)x)a)((f)(((u)(v)a)(b)((p)c)g)c)(((i)(((q)(r)g)j)k)s)((x)(y)(z)e)d)

In our specific scenario, the alphabet involved was the English alphabet, and the conditional probability of inserting any character $a \in A$ given that an insertion occurred was assigned the value 1/26. Similarly, the probability of a character being deleted was set to be 1/20. The table of probabilities for substitution (the confusion matrix) was based on the proximity of the character keys on a standard

QWERTY keyboard and is given in Table I. The channel essentially mutated the nodes (characters, in this case) in the list ignoring the parenthesis, and whenever an insertion or a deletion was introduced special case scenarios were considered so as to insert the additional required parenthesis or remove the superfluous parenthesis respectively. Furthermore, we also ensured that the maintenance of the parenthesis was done in such a way that the underlying expression of parenthesis was well-matched.

*****	**Insert Table I Here** **(The QWERTY Confusion Matrix)**	*****

In our experiments ten NSuTs were generated for each tree in H yielding a test set of 250 NSuTs. The average number of tree deforming operations done per tree was 3.84. Table II gives a list of 5 of the NSuTs generated, their associated subsequence trees and the trees in the dictionary which they originated from. A larger subset of the trees used for these experiments and their noisy subsequence trees (both represented as parenthesized lists) are included in Table III. Table IV gives the average number of errors involved in the mutation of a subsequence tree, U. Indeed, after considering the noise effect of deleting nodes from X^* to yield U, the overall average number of errors associated with each noisy subsequence tree is 21.76.

*****	**Insert Table II Here** **(Subset of Trees used)**	*****

*****	**Insert Table III Here** **(A subset of trees and NSuTs used in Set-up A)**	*****

Type of errors	Number of errors	Average error per tree
Insertion	493	1.972
Deletion	313	1.252
Substitution	153	0.612
Total average error		3.836

Table IV : The noise statistics associated with the set of noisy subsequence trees used in testing.

Every element, Y, in the set of noisy subsequence trees, was compared against the trees in H using the techniques described earlier. The results that were obtained were remarkable. Out of the 250 noisy subsequence trees tested, 232 were correctly recognized, which implies an accuracy of 92.80%. We believe that this is quite overwhelming considering the fact that we are dealing with 2-dimensional objects with an unusually high (about 73%) error rate at the node and structural level.

V.4 Experiment Setup for Data Set B

In the second experimental set-up, the dictionary, H, consisted of 100 trees which were generated randomly. Unlike in the above set (in which the tree-structure and the node values were *manually* assigned), in this case the tree structure for an element in H was obtained by randomly generating a parenthesized expression using the following stochastic context-free grammar G, where,

G = <N, A, G, P>, where,

N = {T, S, $} is the set of non-terminals,

A is the set of terminals - the English alphabet,

G is the stochastic grammar with associated probabilities, P, given below :

$T \rightarrow (S\$)$	with probability 1,
$S \rightarrow (SS)$	with probability p_1,
$S \rightarrow (S\$)$	with probability $1-p_1$,
$S \rightarrow (\$)$	with probability p_2,
$S \rightarrow \lambda$	with probability $1-p_2$, where λ is the null symbol,
$\$ \rightarrow a$	with probability 1, where $a \in A$ is a letter of the underlying alphabet.

Note that whereas a smaller value of p_1 yields a more tree-like representation, a larger value of p_1 yields a more string-like representation. In our experiments the values of p_1 and p_2 were set to be 0.3 and 0.6 respectively. The sizes of the trees varied from 27 to 35 nodes.

Once the tree structure was generated, the actual substitution of '$' with the terminal symbols was achieved by using the benchmark textual data set used in recognizing noisy subsequences [Oo87]. These textual strings consisted of a hundred strings taken from the classical book on pattern recognition by Duda and Hart [DH73]. Each string was the first line of a section or sub-section of the book, starting from Section 1.1 and ending with Section 6.4.3. Further, to mimic a UNIX/TEX file, all the Greek symbols were typed in as English strings. Subsequently, to make the problem more difficult, the spaces between words were eliminated, thus discarding the contextual information obtainable by using the blanks as delimiters. Finally, these strings were randomly truncated so that the length of the words in the dictionary was uniformly distributed in the interval [40, 80]. Thus, the first line of [DH73, Sec. 3.4.1], which reads

"In this section we calculate the *a posteriori* density $p(\theta/X)$ and the desired probability"

yielded the following string :

"inthissectionwecalculatetheaposterioridensitypthetaxandthedesiredpro".

We now consider how the above strings are transformed into parenthesized list representations for trees. The trees generated using the grammar, and the strings considered were both traversed from left to right, and each '$' symbol in the parenthesized list was replaced by the next character in the string. Thus, for example, the parenthesized expression for the tree for the above string was :

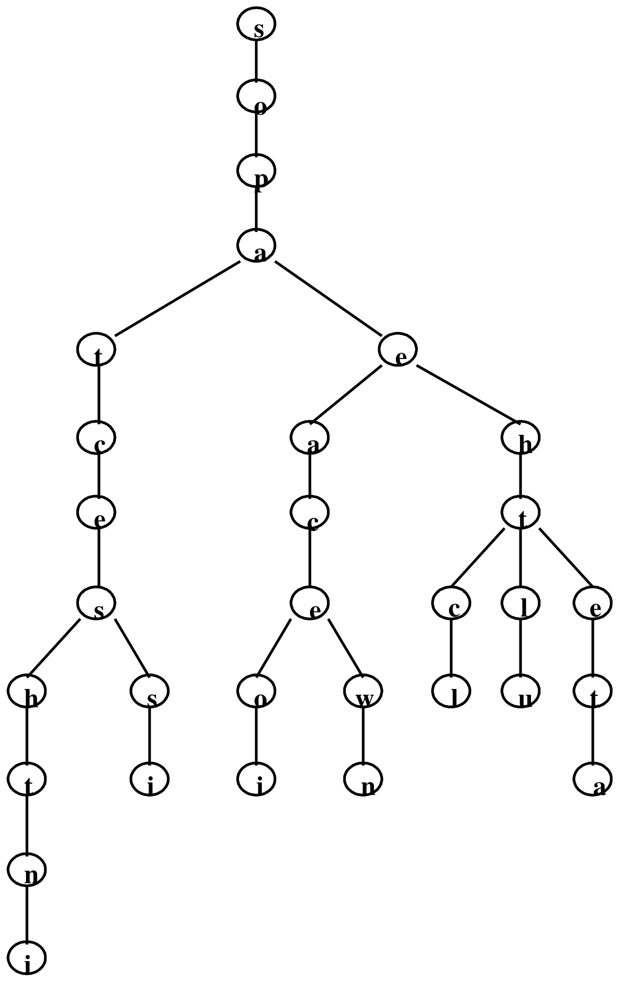
((((((((((($)$)$)(($)$)$)$)$)$)$)(((((($)($)$)$)$)((($)($)(($)$)$)$)$)$)$)$)$)

The '$"s in the string are now replaced by terminal symbols to yield the following list :

(((((((((((i)n)t)h)((i)s)s)e)c)t)((((((i)o)((n)w)e)c)a)((((l)c)((u)l)(((a)t)e)t)h)e)a)p)o)s)

The actual underlying tree for this string is given in Figure VII.

To generate a NSuT for the testing process, as in the above experimental set-up, a tree X^* (unknown to the classification algorithm) was chosen. Nodes from X^* were first randomly deleted producing a subsequence tree, U. In the present case the probability of deleting a node was set to be 60%. Thus although the average size of each tree in the dictionary was 31.45, the average size of the resulting subsequence trees was only 13.42.



String Represented : inthissectionwecalculatetheapos

Tree Represented : (((((((((((i)n)t)h)((i)s)s)e)c)t)((((((i)o)((n)w)e)c)a)((((l)c)((u)l) (((a)t)e)t)h)e)a)p)o)s)

Figure VII : The tree representation of a list obtained from a string

The garbling effect of the noise was then simulated as in the earlier set-up. Thus the subsequence tree U, was subjected to additional substitution, insertion and deletion errors by passing the string representation through a channel causing substitution, insertion and deletion errors as described earlier while simultaneously maintaining the underlying *list* representation of the *tree*. Here too the alphabet being the English alphabet, the probabilities of insertion, deletion and the various confusion substitutions were as described earlier and were based on the QWERTY keyboard.

In our experiments five NSuTs were generated for each tree in H yielding a test set of 500 NSuTs. The average number of tree deforming operations done per tree was 3.77. Table V gives the average number of errors involved in the mutation of a subsequence tree, U. Indeed, after considering the noise effect of deleting nodes from X^* to yield U, the overall average number of errors associated with each noisy subsequence tree is 21.8. The list representation of a subset of the hundred patterns used in the dictionary and their NSuTs is given in Table VI.

Type of errors	Number of errors	Average error per tree
Insertion	978	1.956
Deletion	601	1.202
Substitution	306	0.612
Total average error		3.770

Table V : The noise statistics associated with the set of noisy subsequence trees used in testing.

*****	**Insert Table VI Here** **(A subset of lists in H and their NSuTs)**	*****

Again, each noisy subsequence tree, Y, was compared against the trees in H using the constrained tree distance with the constraint $\tau = \{L_p-1, L_p, L_p+1\}$. The results that were obtained are very impressive. Out of the 500 noisy subsequence trees tested, 432 were correctly recognized, which implies an accuracy of 86.4%. The power of the scheme is obvious considering the fact we are dealing with 2-dimensional objects with an unusually high (about 69.32%) error rate. Also, the corresponding uni-dimensional problem (which only garbled the strings and not the *structure*) gave an accuracy of 95.4% [Oo87].

VIII. CONCLUSIONS

In this paper, we have considered the problem of recognizing trees by processing their noisy subsequence trees. The solution we propose (which, to our knowledge, is the first reported solution) is based on the theoretical work done by Oommen and Lee [OL94] involving constrained tree editing. Given a noisy subsequence tree Y of an unknown tree X^* in H, the technique which we propose estimates X^* by computing the constrained edit distance between every X in H and Y, and this constraint, in turn, is based on the actual garbling properties of the error generating mechanism.

We have rigorously tested our algorithm experimentally for data sets which are manually and randomly generated. The experimental results obtained are remarkable considering the level of noise introduced and the fact that the garbling mechanism mutates the string *and the underlying structure* which stores the trees. We intend to apply these techniques to recognizing biological molecules by processing tree representations of their *fragments*.

REFERENCES

[DH73] R. O Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley and Sons, New York, (1973).

[KM91] P. Kilpelainen and H. Mannila, "Ordered and unordered tree inclusion", Report A-1991-4, Dept. of Comp. Science, University of Helsinki, August 1991; to appear in *SIAM Journal on Computing.*

[LON89] S.-Y. Le, J. Owens, R. Nussinov, J.-H. Chen B. Shapiro and J.V. Maizel, "RNA secondary structures: comparison and determination of frequently recurring substructures by consensus", *Comp. Appl. Biosci.* 5, 205-210 (1989),

[LNM89] S.-Y Le, R. Nussinov, and J.V. Maizel, "Tree graphs of RNA secondary structures and comparisons", *Computers and Biomedical Research*, 22, 461-473 (1989).

[Lu79] S. Y. Lu, "A tree-to-tree distance and its application to cluster analysis", *IEEE Trans. Pattern Anal. and Mach. Intell.*, Vol. PAMI 1, No. 2: pp. 219-224 (1979).

[Lu84] S. Y. Lu, "A tree-matching algorithm based on node splitting and merging", *IEEE Trans. Pattern Anal. and Mach. Intell.*, Vol. PAMI 6, No. 2: pp. 249-256 (1984).

[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", *IEEE Trans. Pattern Anal. and Mach. Intell.*, Vol. PAMI 9, No. 5 : pp. 676-685 (1987).

[OK98] B. J. Oommen and R. L. Kashyap, "A formal theory for optimal and information theoretic syntactic pattern recognition", *Pattern Recognition*, Vol. 31, 1998, pp. 1159-1177.

[OL94] B. J. Oommen, and W. Lee, "Constrained Tree Editing", *Information Sciences*, Vol. 77 No. 3,4: pp. 253-273 (1994).

[OZL96] B. J. Oommen, K. Zhang, and W. Lee, "Numeric similarity and dissimilarity measures between two trees", *IEEE Transactions on Computers*, Vol.TC-45, December 1996, pp.1426-1434.

[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", *Inform. Process. Letters*, Vol. 6, No. 6: pp. 184-186 (1977).

[Sh88] B. Shapiro, "An algorithm for comparing multiple RNA secondary structures", *Comput. Appl. Biosci.*, 387-393 (1988).

[SZ90] B. Shapiro and K. Zhang, "Comparing multiple RNA secondary structures using tree comparisons", *Comput. Appl. Biosci.* vol. 6, no. 4, 309-318 (1990).

[Ta79] K. C. Tai, "The tree-to-tree correction problem", *J. Assoc. Comput. Mach.*, Vol. 26 : pp. 422-433 (1979).

[TSSS87] Y. Takahashi, Y. Satoh, H. Suzuki and S. Sasaki, "Recognition of largest common structural fragment among a variety of chemical structures", *Analytical Science* Vol. 3, 23-28 (1987).

[WF74] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem", *J. Assoc. Comput. Mach.*, Vol. 21 : pp. 168-173 (1974).

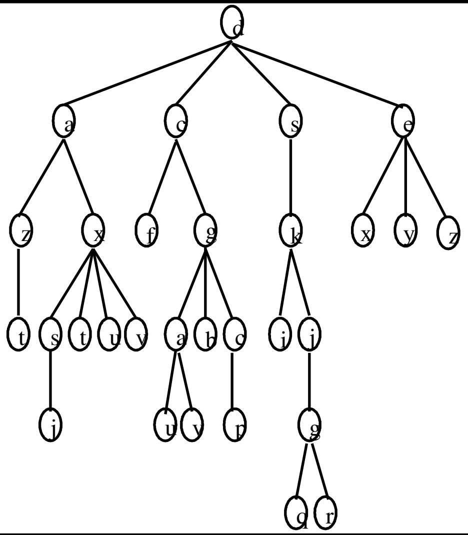
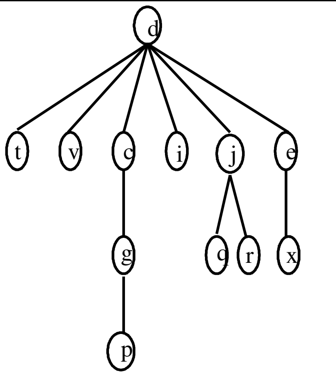
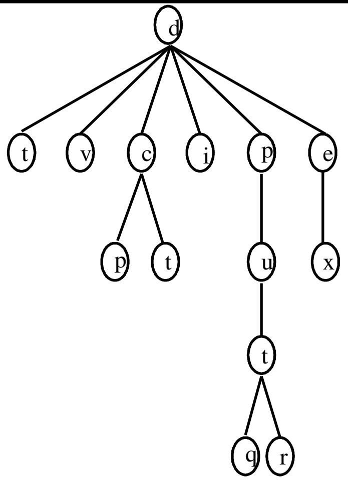
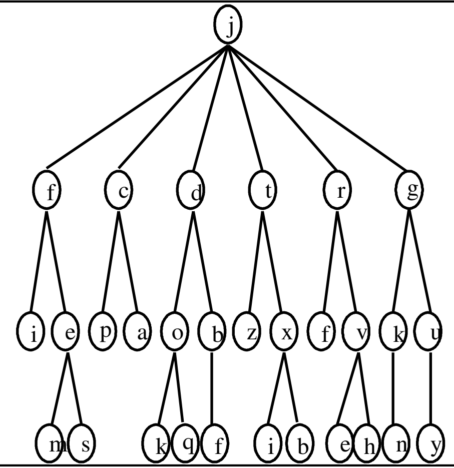
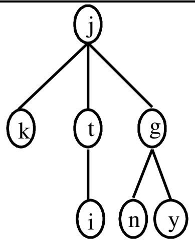
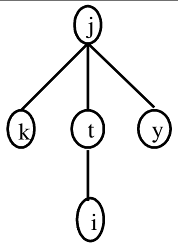
[Zh90] K. Zhang, "Constrained string and tree editing distance", Proceeding of the IASTED International Symposium , New York, pp. 92-95 (1990).

[ZJ94] K. Zhang and T. Jiang, "Some MAX SNP-hard results concerning unordered labeled trees", *Information Processing Letters,* 49, 249-254 (1994).

[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 (1989).

[ZSS92] K. Zhang, R. Statman, and D. Shasha, "On the editing distance between unordered labeled trees", *Information Processing Letters,* 42, 133-139 (1992).

[ZSW92] K. Zhang, D. Shasha and J. T. L. Wang, "Fast serial and parallel approximate tree matching with VLDC's", *Proceedings of the 1992 Symposium on Combinatorial Pattern Matching,* CPM92, 148-161 (1992).

Original tree	Subsequence tree	Noisy subsequence tree
		
		

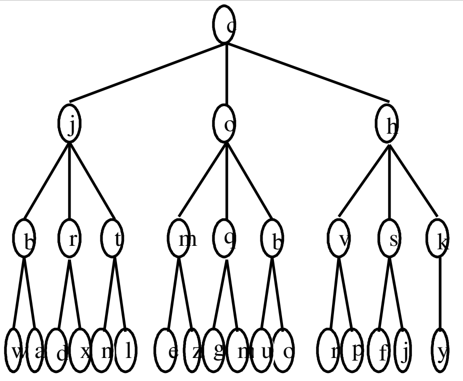
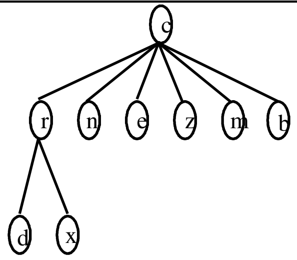
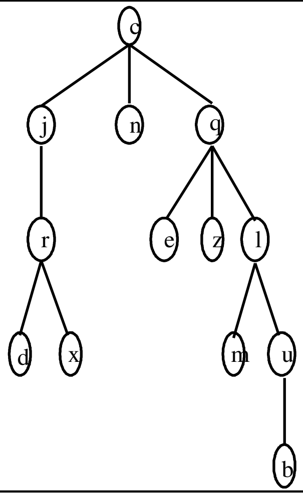
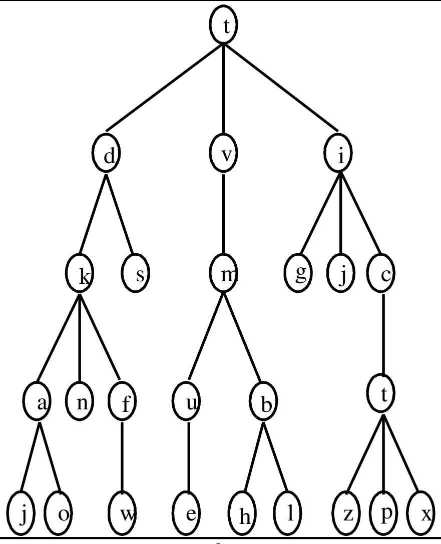
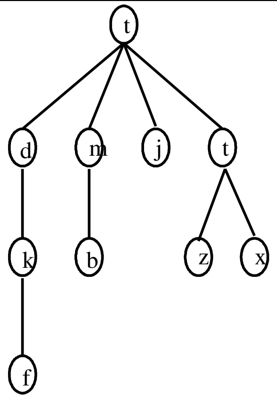
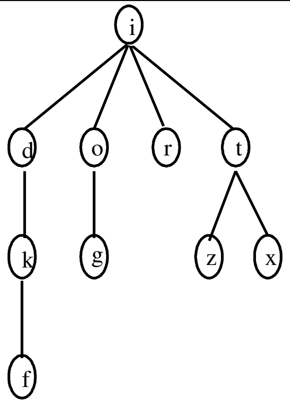
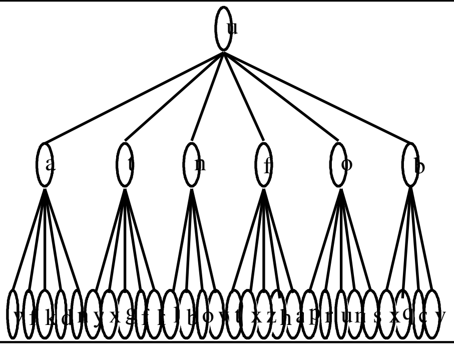
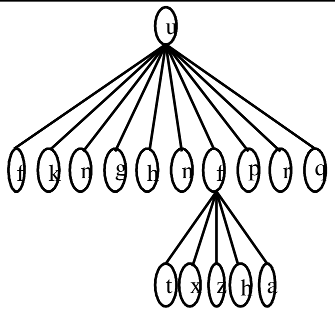
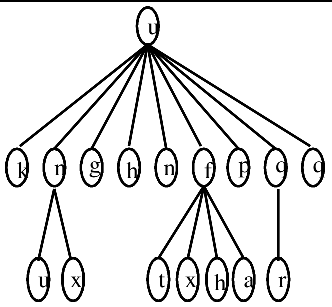
		
		
		

Table II : Examples of the original trees, the associated subsequence trees and their noisy versions.

Original tree:	((((t)z)(((j)s)(t)(u)(v)x)a)((f)(((u)(v)a)(b)((p)c)g)c)(((i)(((q)(r)g)j)k)s)((x)(y)(z)e)d)
Noisy tree:	((y)(v)((((t)p)g)c)(i)((q)(r)j)(x)d)
Original tree:	((((x)(((e)j)(((p)c)(v)k)a)s)((g)t)(u)p)((j)((((b)f)(g)c)(a)(((s)((l)(n)k)(u)z)d)v)(e)q)r)
Noisy tree:	(((x)((e)j)((p)c)(g)p)(f)(g)(a)(s)(l)(u)(e)r)
Original tree:	(((i)((m)(s)e)f)((p)(a)c)(((k)(q)o)((f)b)d)((z)((i)(b)x)t)((f)((e)(h)v)r)(((n)k)((y)u)g)j)
Noisy tree:	((k)((k)t)(((n)v)(y)g)j)
Original tree:	(((((t)(s)(b)c)((x)(p)(q)f)((b)(e)(d)j)r)((((h)m)(l)(e)k)((u)(w)((f)x)d)((v)(y)(z)u)q)e)p)
Noisy tree:	((((((t)y)(b)c)(((t)x)(q)r)(j)m)e)((e)(s)(x)((((c)(s)(r)(y)c)e)(z)u)q)p)
Original tree:	(((((n)d)e)j)((((w)t)b)(((v)(f)s)((j)(d)(r)p)c)f)((((p)l)r)(((x)(k)h)(z)y)((a)(m)(s)u)k)o)
Noisy tree:	((d)(w)(((j)(s)p)c)(y)(m)o)
Original tree:	(((((r)(m)(h)t)((l)(n)c)e)s)((f)((w)(((x)p)(d)q)v)(g)b)(((o)r)((a)i)j)((((s)z)((b)p)u)c)k)
Noisy tree:	(((x)(m)(h)e)(q)((r)j)k)
Original tree:	((((w)(a)b)((d)(x)r)((n)(l)t)j)(((e)(z)m)((g)(m)q)((u)(o)b)o)(((r)(p)v)((f)(j)s)((y)k)h)c)
Noisy tree:	((((w)d)(x)r)(n)(e)(z)((k)m)(b)c)
Original tree:	(((s)(v)(t)(u)(a)b)((d)(b)(e)(r)n)((l)(k)(s)h)(((t)(p)(q)(j)p)(a)i)(((c)(t)(o)(h)(j)e)r)g)
Noisy tree:	((s)(u)((d)(d)(r)n)((k)(s)h)(p)(h)g)
Original tree:	(((((a)(w)m)((v)(l)y)k)(((j)(n)f)((u)(g)t)c)u)((((b)(m)p)((d)(i)h)m)(((s)(c)y)((p)q)s)f)s)
Noisy tree:	((((v)k)((x)n)u)(b)(m)(d)(q)a)
Original tree:	((c)((((d)((l)(f)t)(o)a)((v)((k)e)k)s)((u)(s)(t)q)((m)n)(((i)(r)(h)l)(c)(w)((j)y)p)b)(l)i)
Noisy tree:	((d)(l)(k)(y)(s)(l)(c)(w)(j)(l)i)
Original tree:	(((((j)(o)a)(n)((w)f)k)(s)d)((((e)u)((h)(l)b)m)v)((g)(j)(((z)(p)(x)t)c)i)t)
Noisy tree:	((((f)m)d)((b)m)(j)((u)(z)(x)t)t)
Original tree:	(((m)((d)(y)(q)j)t)((((i)o)e)s)(((x)p)((f)u)(v)n)(c)((((w)b)((e)z)r)((g)k)p)a)
Noisy tree:	((m)((y)(c)j)((o)s)((b)(p)r)((g)k)a)
Original tree:	((((g)((d)((m)o)n)(u)p)(i)(((x)b)(t)f)(s)(((l)(a)e)c)k)((d)((((w)(y)z)(n)r)p)m)w)
Noisy tree:	((((o)p)g)(((((e)z)h)c)f)((((y)l)(a)g)c)((y)p)w)
Original tree:	(((s)(((t)(k)x)(e)((v)(y)r)i)(m)a)(((w)((j)(u)k)(g)p)(d)(o)r)(((c)(l)((i)(z)q)n)t)b)
Noisy tree:	(((t)x)(s)((v)r)(u)(n)b)
Original tree:	(((t)((((e)(j)(d)c)w)(y)((g)((t)(v)(b)n)u)h)((s)((f)((g)(n)(a)x)(h)i)r)((p)(a)(m)o)b)z)
Noisy tree:	((y)(y)(g)(g)(a)(h)((p)(a)(m)o)z)
Original tree:	((((k)(d)q)((j)s)(h)p)(((t)r)(i)o)(((g)(b)c)((w)(x)n)k)((a)((v)d)e)(((p)m)((s)n)((k)(r)f)y)l)
Noisy tree:	((x)(((s)p)(t)(i)(((g)c)((w)n)k)(a)((m)((r)y)y)i)l)
Original tree:	(((n)(((y)((w)i)(p)e)(u)((z)d)(o)(((i)((z)(l)c)f)(k)r)h)(l)v)((((j)(r)k)c)(a)((b)(t)((s)g)d)p)m)
Noisy tree:	((n)((o)(p)e)(d)(((s)z)(o)c)(((j)(r)k)c)(a)(b)m)
Original tree:	(((((o)(g)b)(u)((y)r)(s)(q)v)(p)(x)(((j)c)(f)(h)(y)t)(m)l)((a)(((k)(w)x)(r)(e)o)(z)((l) ((i)v)n)d)x)
Noisy tree:	(((((g)(y)y)(p)(x)(((h)d)(y)t)(m)l)((g)((e)y)o)z)x)

Table III : A subset of the trees used for Data Set A and their noisy subsequence trees. The trees and subsequence trees are represented as parenthesized lists.

Original tree:	(((s)((i)n)c)(e)(t)(h)(e)((a)d)(((v)(e)n)t)((o)(f)(t)(h)e)(((d)(i)g)i)(t)((a)l)((c)o)((((m)p)((u)t)e)r)t)
Noisy tree:	((n)(a)((f)t)(t)(i)(t)(a)(o)((t)r)t)
Original tree:	(((t)(o)(i)(l)(((l)(u)s)t)r)((a)(t)(e)((s)o)((m)(e)o)(((f)t)h)e)(((t)((h)e)(t)y)(p)((e)s)o)f)
Noisy tree:	(((o)(i)(u)r)((a)(t)((e)o)(f)e)(g)(p)f)
Original tree:	((((t)(h)(e)((p)r)e)c)(((e)d)((i)n)((g)(e)(x)(a)m)p)((l)(e)c)(((o)n)t)(((a)i)(n)s)(((m)(a)n)y)o)
Noisy tree:	(((((e)((p)r)e)v)d)(e)(e)(x)(a)((((q)k)n)t)((i)(b)s)(((a)n)y)o)
Original tree:	(((((t)h)e)r)((e)a)(r)((e)m)(a)(n)(y)((p)(r)(o)(b)(l)e)((m)(s)((i)n)(p)(a)(t)(t)((e)r)(n)((r)e)((c)o)g)n)
Noisy tree:	(((t)e)(((k)y)r)(((n)o)a)(b)((m)(s)(a)(t)(r)g)n)
Original tree:	((((t)(h)((e)o)(r)(i)((g)(i)(n)(a)t)i)((o)(n)o)(f)(p)(a)(r)(t)(i)((o)f)((t)h)((i)s)(b)(o)(o)((k)(i)(s)p)r)i)
Noisy tree:	(((r)((a)(o)p)(n)(f)(r)((b)t)(((p)q)h)(i)(b)(o)(o)(l)r)i)
Original tree:	((((t)h)(e)(p)((u)b)l)(i)((s)h)(((e)d)l)(((i)t)e)((r)((a)t)u)((r)((e)(o)n)((p)a)t)t)
Noisy tree:	(((n)(e)(p)((e)l)(t)((a)u)r)((a)t)t)
Original tree:	(((((b)((a)y)e)s)((d)(e)(c)(i)(s)(i)o)(((n)t)h)e)((((o)r)y)(i)((s)a)(((f)u)n)(((d)a)m)e)n)
Noisy tree:	(((e)((w)(i)o)e)(((s)a)(j)(d)e)n)
Original tree:	((w)(e)(s)(h)(((a)l)(l)n)(o)(w)((f)o)(r)(m)(a)(l)(((i)z)e)((t)(h)e)(i)(d)(e)((a)s)((j)u)(s)t)
Noisy tree:	((s)(h)(l)(w)((g)r)(m)(a)(l)(i)(e)(e)(u)t)
Original tree:	(((l)(e)((t)u)(s)((s)p)(e)(c)((i)a)l)(((i)(z)e)(t)(h)(e)((s)(e)(r)e)s)(((u)(l)(t)(s)((b)(y)c)(o)(n)(s)i)d)e)
Noisy tree:	(((e)(t)(s)l)(i)((y)s)(e)((l)(b)(e)d)e)
Original tree:	((((i)n)c)(l)(((a)s)s)((i)(f)(i)c)(a)(t)((i)o)(n)(p)(r)(o)((b)(l)(e)((m)(s)e)((a)((c)h)s)t)a)
Noisy tree:	(((i)c)(l)((k)c)(n)(r)(o)((l)(e)(w)((s)(h)s)t)a)
Original tree:	((((((t)(h)(e)(r)((e)a)r)(e)((m)a)(n)(y)(d)(i)f)((((f)(e)r)e)n)(t)(w)(a)((y)(s)t)(o)(r)(e)(p)(r)e)s)e)
Noisy tree:	((((((a)r)(y)c)((f)(e)r)((t)(w)v)s)(a)(y)(s)(o)(r)(e)(r)s)e)
Original tree:	((((((w)(h)i)((l)e)((t)h)(e)t)w)((o)c)(a)(t)(a)(g)((o)(r)y)(c)(a)(s)(e)(i)((s)h)((u)s)((((t)a)s)p)e)c)
Noisy tree:	(((((h)i)z)(t)w)(o)(t)(c)(s)((i)g)(h)(s)c)
Original tree:	((b)(y)(t)(h)(i)(n)(k)(i)((n)g)(((o)((f)a)c)l)((a)(s)(s)i)(f)((i)e)(r)(a)((s)(a)d)((e)v)(i)(c)e)
Noisy tree:	((b)(n)(c)(a)(f)(a)(s)((e)v)(i)e)
Original tree:	((((t)(h)(e)(s)(t)(r)(u)((c)((t)u)r)(((e)o)(f)(a)(b)((a)y)e)(s)(c)(l)((a)s)((s)i)((f)i)((e)r)i)s)d)
Noisy tree:	(((h)((s)(r)((r)(t)u)(o)w)(a)(l)(i)(f)s)d)
Original tree:	((((w)(e)b)(e)(g)(i)(n)((w)i)(t)(h)(t)(h)(e)(((u)n)i)(((v)a)r)(i)(((a)t)e)(n)((o)r)((m)a)(l)(d)(e)n)s)
Noisy tree:	(((w)n)(h)((a)r)(e)(n)(o)(a)(l)s)
Original tree:	((((t)((h)e)(g)(e)(n)(e)(r)(((((a)l)m)u)l)t)((i)(v)(a)((r)i)(a)((t)(e)n)((o)(r)(m)a)l)d)e)
Noisy tree:	(((n)(g)(n)(r)(((a)l)l)t)(i)(v)(a)(r)((t)(e)n)((o)(r)(m)a)e)

Original tree:	((((t)h)(e)(s)(i)(m)((p)((l)e)s)(((t)c)a)((s)(e)o)((c)c)((u)(r)(e)(s)(w)(((h)e)n)t)h)e)
Noisy tree:	((((((t)a)h)((p)(l)e)f)(s)(h)h)e)

Table VI : A subset of the trees used for Data Set B and their noisy subsequence trees. The trees and subsequence trees are represented as parenthesized lists. The original unparenthesized strings are the same as those used in [Oo87] and were obtained from [DH73].