OPTIMAL DECOMPOSITION STRATEGY FOR TREE EDIT DISTANCE

(Spine title: Plib)

(Thesis format: Monograph)

by

Shaofeng Jiang

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Examiners:

Supervisor:

.....
Dr. Kaizhong Zhang

.....
Dr. Lucian Ilie

.....
Dr. Marc Moreno Maza

.....
Dr. Xingfu Zou

The thesis by

Shaofeng Jiang

entitled:

Optimal Decomposition Strategy For Tree Edit Distance

is accepted in partial fulfillment of the
requirements for the degree of
Masters of Science

.....
Date

.....
Chair of the Thesis Examination Board

Abstract

An ordered labeled tree is a tree where the left-to-right order among siblings is significant. Given two ordered labeled trees, the tree edit distance is the minimum cost to convert one tree to another.

In this thesis, we present an algorithm for the tree edit distance problem by using the optimal tree decomposition strategy. Besides, the vertical compression applied to trees can significantly reduce the time complexity. We compare our method with other methods. The test result shows that our strategies on compression tree are by far the best decomposition strategy, creating the least number of relevant sub-problems.

Keywords: ordered labeled tree, tree edit distance, dynamic programming, sequence comparison, tree decomposition, heavy path, RNA secondary structure comparison.

Contents

Certificate of Examination ii

Abstract iii

List of Figures vii

List of Tables ix

1 Introduction 1

2 Background 4

2.1 String Edit Distance Problem 4

2.2 Tree Edit Distance Problem 5

2.3 Preliminaries 6

2.4 Recursive Decomposition Solution 8

2.5 Relevant SubForests and SubTrees 10

2.6 Bottom-up Enumeration 12

2.7 A Simple Algorithm 15

2.8 Improved Algorithmic Path Strategies 17

2.9 Zhang and Shasha's Algorithm 18

2.10 Klein's Algorithm 20

2.11 Demaine's Algorithm 24

2.12 Conclusion 27

3 A new algorithm **29**

3.1 Finding the Optimal Root-leaf Decomposition Path 29

3.1.1 Main Idea 29

3.1.2 Relevant Leftmost Forest and Relevant Rightmost Forest 29

3.1.3 Number of Relevant Forests For a Tree 30

3.1.4 Number of Relevant forests for a Pair of Trees 34

3.1.5 Dynamic Programming Implementation 40

3.2 Distance Computation in Bottom-up Fashion 40

3.2.1 Double Roots Encoding 41

3.2.2 Bottom-up Enumeration 41

3.2.3 A Quadratic Space Complexity Implementation 45

4 Another algorithmic improvement **49**

4.1 Main Idea 49

4.2 Vertical Reduction on Trees 49

4.3 Computation 51

4.4 Implementation 53

5 Experiment **56**

5.1 RNA and its Secondary Structure 56

5.2 String Representation of the RNA Secondary Structure 57

5.3 RNA Secondary Structure Graphs 58

5.4 Tree Representation of the RNA Secondary Structure 58

5.5 Datasets 59

5.6 Some Experimental Result 62

5.7 Evaluation 62

6 Conclusion **65**

Bibliography **67**

List of Figures

1.1 RNA structures and forest representation. From [8] (a) A segment of the RNA GI: 2347024 primary structure, (b) its secondary structure, (c) its forest representation 2

2.1 Basic tree edit operations From [2]. (a) substitution, (b) deletion, (c) insertion. 6

2.2 Tree editing constraints on sibling orders and ancestor orders preservation From [2]. (a) sibling order preservation. (b) ancestor order preservation. 6

2.3 Left and Right decomposition From [3]. (a-1) leftmost deletion. (b-1) rightmost insertion. (c-1) leftmost substitution. (a-2) rightmost deletion. (b-2) rightmost insertion. (c-2) rightmost substitution. 9

2.4 Relevant sub-forests resulting from the full decomposition. From [9]. 10

2.5 Relevant sub-forests. From [9]. (a) Relevant sub-forests from the leftmost path decomposition. (b) Relevant sub-forests from the rightmost path decomposition. 11

2.6 Relevant sub-forests. From [4]. 12

2.7 Relevant sub-forests that result from the leftmost path decomposition. From [3]. 14

2.8 An example of enumerating subforests in prefix-suffix postorder. From [3]. . . 15

2.9 An example of enumerating subforests in suffix-prefix postorder. From [3]. . . 15

2.10 Relevant sub-forests that result from the leftmost path decomposition. From [5]. 17

2.11 Relevant sub-forests that result from the rightmost path decomposition. From [5]. 17

2.12 Leftmost paths and rightmost paths in a tree and its resulting sub-trees From [3]. 18

2.13 Heavy Paths to a tree and its relevant sub-trees From [7]. 21

2.14 Example of decomposition for Klein's algorithm From [7]. 22
2.15 Example of enumerating sub-forests in H-postorder From [2]. 22

3.1 Example subforests of tree G and their root encoding representation [9]. 42
3.2 The order of processing nodes in loops A [9]. 42
3.3 The order of processing nodes in loop C and C'. [9]. 44
3.4 The order of processing nodes in loop B. [9]. 44
3.5 The order of processing nodes in loop D. [9]. 45

4.1 Maximal non-branching path [3]. 50
4.2 An example of the mapping of nodes between the original tree and its vertical reduced tree [3]. 50

5.1 RNA Tertiary Structure. 57
5.2 String representation of the RNA secondary structure. 58
5.3 Graph representation of the RNA secondary structure. 58
5.4 Tree representation of the RNA secondary structure. 60
5.5 An RNA Sequence in XML Format. 61
5.6 Alcaligenes eutrophus Sequence from the RNase P database. 62
5.7 Streptomyces bikiniensis Sequence from the RNase P database. 62
5.8 The structures of RNase P RNA of Cupriavidus metallidurans (Alcaligenes eutrophus) and Streptomyces bikiniensis. 63
5.9 Alignment Between the Secondary Structure of Alcaligenes eutrophus and Streptomyces bikiniensis. 63

List of Tables

2.1 State-of-the-Art Algorithms in Tree Edit Distance Problem 28

5.1 The Label of Base Pair 59

5.2 The Relevant Sub-problem and its Actual Run Time of Each Algorithms 64

Chapter 1

Introduction

An ordered labeled tree is a tree in which the nodes are labeled and the left-to-right order among siblings is significant. One way of comparing two ordered labeled tree is by measuring their edit distance.

Tree edit distance was first introduced by Tai [10] as a generalization of the string editing problem. The computation of the string edit distance counts the minimum number of operations(insert, delete and replace) required to transform one string into the other, which quantifies the similarity between two strings. Similarly, given two ordered labeled tree T_1 and T_2, the tree edit distance between T_1 and T_2 is the minimum cost to transform one tree into another using three elementary operations: insert, delete and institution. Tai gave an algorithm with a time complexity of $O(|T_1|^3 * |T_2|^2)$. Later on, a number of improved algorithms were developed[[12], [7], [5], [4], [3], [9]].

An ordered labeled tree can represent a scene description, an XML document, a natural language parse, and other phenomena. Given a pattern data and match data, we can formulate these data into two order labeled trees and we may want to match the pattern tree to the data tree.

RNA secondary structure comparison is an application of the tree edit distance. RNA is a single strand of nucleotides. The nucleotides in the strand have selectively sticky ends. Because

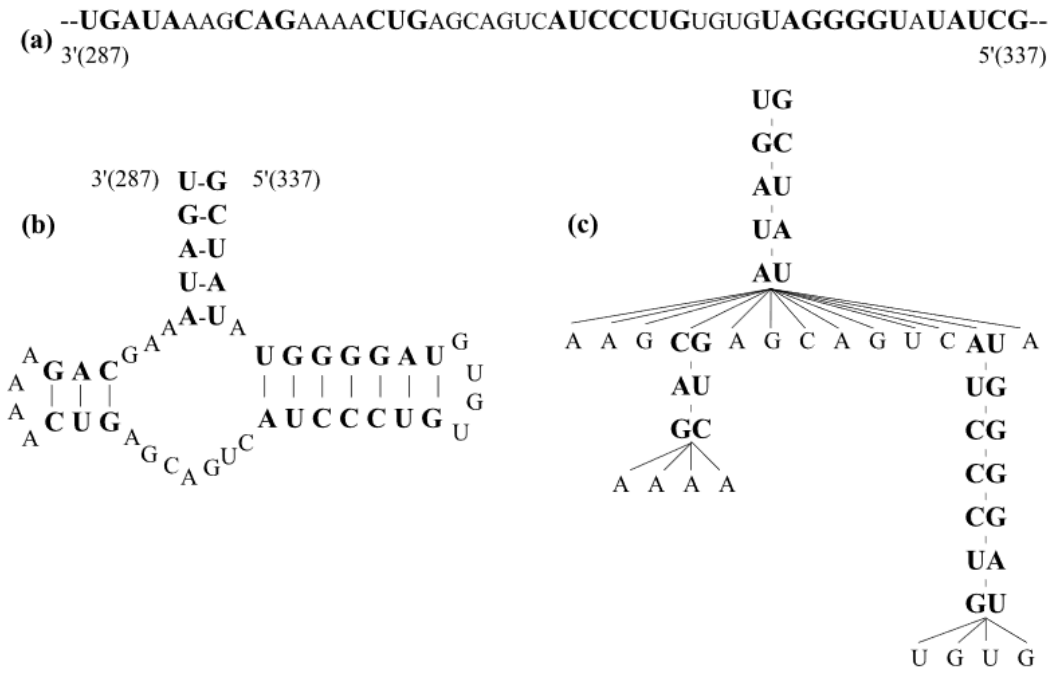


Figure 1.1: RNA structures and forest representation. From [8] (a) A segment of the RNA GI: 2347024 primary structure, (b) its secondary structure, (c) its forest representation

.

of the stickiness, the strand bends around and sticks to itself, which is topologically a tree. This tree-like structure of RNA is called the secondary structure of RNA and is depicted in Figure 1.1.

The secondary structure of RNA plays an important role in its functions preforming. In biology, it is presumed that a preserved biological function corresponds to a preserved molecular structure. Therefore, the comparison secondary structures of RNA is required in many biology problems with RNA involving, which helps reveal information regarding RNA functions.

We designed and implemented a new space and time efficient algorithm to compute the tree edit distance. Compared to other currently existing algorithm, the new algorithm use the best decomposition strategy for trees, which creates the least number of the relevant sub-problems.

The thesis is organized as follows. Chapter 2 introduces the tree edit distance problem and the related work of tree edit distance algorithm. A number of related path decomposition algorithms are described in this chapter. A detailed explanation of the design and implemen-

tation of our improved algorithm follows in Chapter 3. The evaluation of the new method is performed in Chapter 4 by comparing it with other leading methods. We conclude in Chapter 5.

Chapter 2

Background

This chapter introduces the tree editing problem and the related algorithms. The underlying concepts of the tree editing problem are first provided, followed by a naive algorithm. Then a number of improved algorithms based on closely related dynamic programming approaches are introduced.

2.1 String Edit Distance Problem

Before we introduce the tree edit distance problem, we first describe the string edit distance problem because the string edit distance problem can be seen as a special case of the tree edit distance problem.

First introduced by Wagner and Fischer [11], the string edit distance problem is to find the minimum cost to change one stringS_1 into the otherS_2 by a sequence of edit steps. The string edit distance problem can be solved by dynamic programming. The string edit distance $d(S_1, S_2)$ can be computed by Equation 1 where u and v are both the last elements or the first element of string S_1 and S_2. The cost of the three basic edit operations(substitution, deletion and insertion) are $\delta(u, v)$, $\delta(u, \varnothing)$ and $\delta(\varnothing, v)$.

Definition (String Edit Distance) The edit distance between strings S_1 and S_2 is the minimum

cost to change S_1 to S_2 via a sequence of basic edit steps.

2.2 Tree Edit Distance Problem

The tree edit distance problem was first introduced by Tai as a generalization of the string edit distance problem [11].

Definition (Tree Edit Distance) The edit distance between two trees T_1 and T_2 is the minimum cost to change T_1 to T_2 via a sequence of basic edit steps.

Analogous to the string edit distance problem, the basic operations are substitution with the cost $\delta(t_1, t_2)$, insertion with the cost $\delta(\varnothing, t_2)$ and deletion with the cost $\delta(t_1, \varnothing)$, where t_1 and t_2 is a node in tree T_1 and T_2 respectively. The concept of basic operations in the tree edit distance problem is similar to that in the string edit distance problem. Substitution means changing a tree node into another. Insertion first insert a node into a tree. If the inserted node is the children of a node in the tree, the children of this node become the children of the inserted node. In contrast to insertion, deletion first delete a node from a tree then the children of the deleted node become the children of the parent of the deleted node. Figure 2.1 illustrates these editing operations.

In this thesis, we focus on the general editing problem, which means no additional constraints are added on the order of insertions and deletions. In other words, insertions and deletions can take place in any order at any node within the tree. However, the substitution operation should satisfy the following constraints:

1. One-to-one relationship: A node in one tree can be replaced at most one node in another tree.

2. Sibling order is preserved: For any two substitution steps($t_1[i] \rightarrow t_2[j]$) and ($t_1[i'] \rightarrow t_2[j']$), $t_1[i]$ is to the left of $t_1[i']$ if and only if $t_2[j]$ is an ancestor of $t_2[j']$.

3. Ancestor order is preserved: For any two substitution step($t_1[i] \rightarrow t_2[j]$) and ($t_1[i'] \rightarrow t_2[j']$), $t_1[i]$ is an ancestor of $t_1[i']$ if and only if $t_2[j]$ is an ancestor of $t_2[j']$.

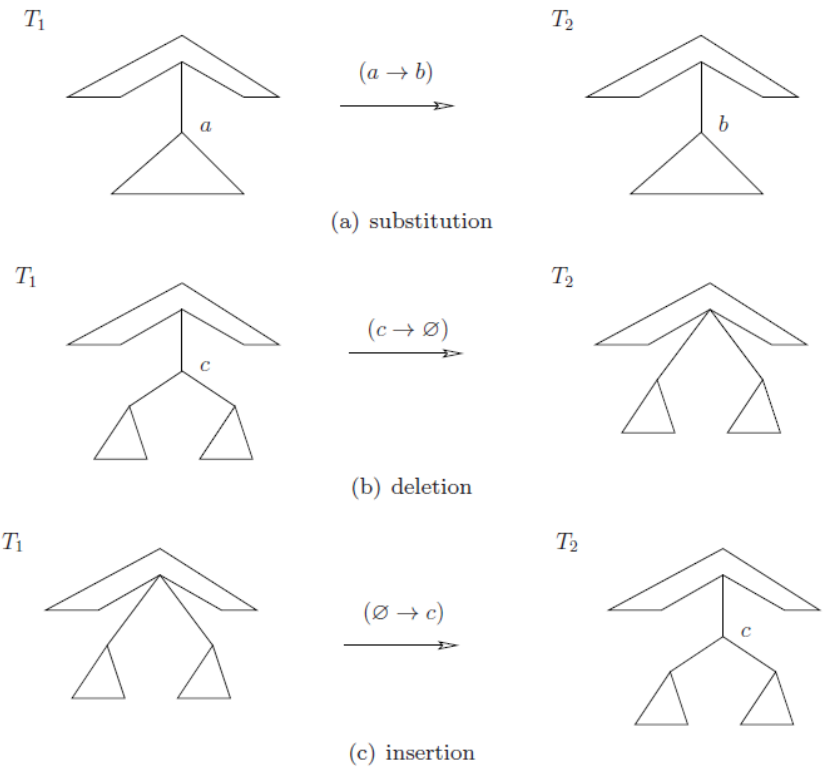


Figure 2.1: Basic tree edit operations From [2]. (a) substitution, (b) deletion, (c) insertion.

These constraints of the preservation of sibling and ancestor order are shown in Figure 2.2.

2.3 Preliminaries

Before we study the tree edit distance problem, it would be beneficial to define some notations, which can help analyze the algorithm clearly.

Firstly, trees and forests should be clearly defined as they are objects in algorithms.

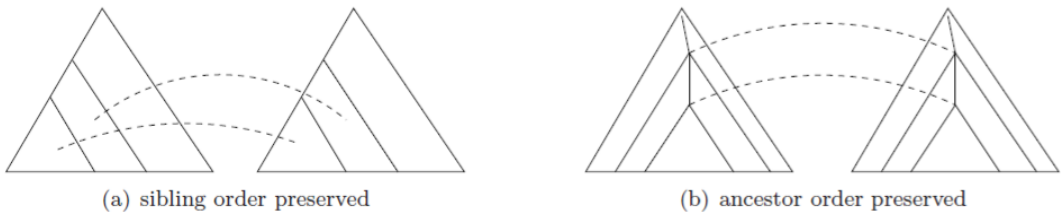


Figure 2.2: Tree editing constraints on sibling orders and ancestor orders preservation From [2]. (a) sibling order preservation. (b) ancestor order preservation.

Definition (Tree and forests) A tree is a node connected to an ordered sequence of disjoint trees. Such a sequence of tree is called a forest [5].

In this thesis, we only consider ordered and labeled tree. The word forest may be used for denoting forests, trees as well as a node which are reduced from a tree after a series of deletions.

Some classical notations for trees and forests are introduced.

Notation Let T be a tree, which is composed of a node l connected to the sequence of trees $T_1, \cdots, T_n$, T can be written as $l(T_1 \circ \cdots \circ T_n)$.

- $r(T)$ denotes the root of T, which is l in the tree representation form of $l(T_1 \circ \cdots \circ T_n)$.
- T° denotes the forest F after deleting r(T), that is $T_1 \circ \cdots \circ T_n$.
- $lr(T)$ denotes the root of the leftmost tree in the forest T°, that is the root of T_1.
- $rr(T)$ denotes the root of the rightmost tree in the forest T°, that is the root of T_n.

Notation Let F be a forest of the form of $T_1 \circ \cdots \circ T_n$, where $T_1, \cdots, T_n$ are trees in the forest.

- $|F|$ denotes the size of F, which is the number of nodes in the forest.
- $\#leaves(F)$ denotes the number of leaves of F.
- $depth(F)$ denotes the depth of F, that is the maximal depth of the trees in F.
- $F(i)$, where i is a node of F, denotes the sub-tree of F rooted at i.
- $F - i$, where i is a node of F, denotes the forest after deleting node i.
- $lr(F)$ denotes the root of the leftmost tree in the forest $T_1 \circ \cdots \circ T_n$, that is the root of T_1.
- $rr(F)$ denotes the root of the rightmost tree in the forest $T_1 \circ \cdots \circ T_n$, that is the root of T_n.

2.4 Recursive Decomposition Solution

Before we study the recursive decomposition in the tree edit distance problem, we recall the the recursive decomposition in the string edit distance problem. The string edit distance problem can be solved by measuring the distance for all pairs of prefixes or suffixes of two strings.

The edit distance between two strings S_1 and S_2 can be computed by equation as follows:

$$d(S_1, S_2) = min \begin{cases} \delta(S_1 - u, S_2) + \delta(u, \varnothing) \\ \delta(S_1, S_2 - v) + \delta(\varnothing, v) \\ \delta(S_1 - u, S_2 - v) + \delta(u, v) \end{cases} \tag{2.1}$$

It is a right decomposition if and only if u and v are both the last element of the string S_1 and S_2. Similarly, when u and v are both the first element of the string S_1 and S_2, it is a left decomposition. Implementation of the string edit distance is completed with a two-dimensional table, which gives a $O(n^2)$ solution.

Analogous to the string edit distance problem, the tree edit distance can be solved in a recursive decomposition way. To compute the tree edit distance, the roots of the trees are the first elements to decompose. The Equation 2.2 computes the tree-to-tree distance.

Let T_1 and T_2 are both trees,

$$d(T_1, T_2) = min \begin{cases} \delta(T_1 - r(T_1), T_2) + \delta(r(T_1), \varnothing) \\ \delta(T_1, T_2 - r(T_2)) + \delta(\varnothing, r(T_2)) \\ \delta(T_1 - r(T_1), T_2 - r(T_2)) + \delta(r(T_1), r(T_2)) \end{cases} \tag{2.2}$$

The decomposition to trees create sub-problems for forests$(T_1 - r(T_1), T_2 - r(T_2))$. Similar to the Equation 2.1 for the calculation of the string edit distance, let F_1 and F_2 are both forests and u and v are nodes in forest F_1 and F_2 respectively, the forest-to-forest distance can be

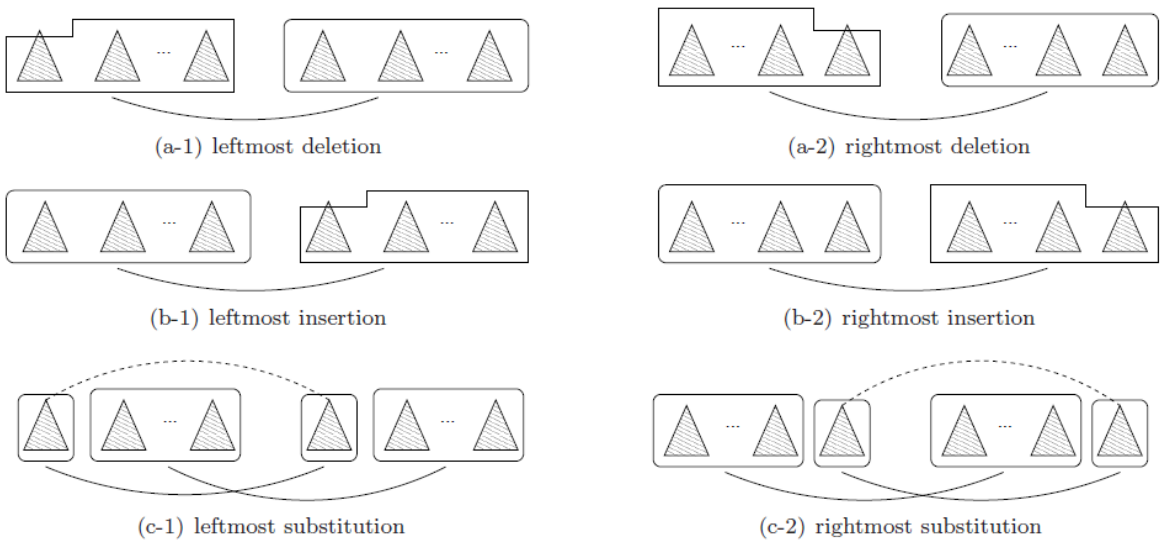


Figure 2.3: Left and Right decomposition From [3]. (a-1) leftmost deletion. (b-1) rightmost insertion. (c-1) leftmost substitution. (a-2) rightmost deletion. (b-2) rightmost insertion. (c-2) rightmost substitution.

computed by equation as follows:

$$d(F_1, F_2) = min \begin{cases} \delta(F_1 - u, F_2) + \delta(u, \varnothing) \\ \delta(F_1, F_2 - v) + \delta(\varnothing, v) \\ \delta(F_1 - F(u), F_2 - F(v)) + \delta(F(u), F(v)) \end{cases} \tag{2.3}$$

Analogous to the string edit distance problem, the computation of forest-to-forest takes on two possible directions: left and right decomposition. Figure 2.3 illustrates the left and right decomposition.

- left decomposition where u and v are $lr(F_1)$ and $lr(F_2)$ respectively.(Figure 2.3 (a-1), (b-1), (c-1)).

- right decomposition where u and v are $rr(F_1)$ and $rr(F_2)$ respectively.(Figure 2.3 (a-2), (b-2), (c-2)).

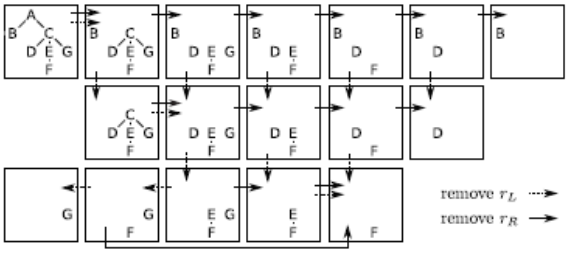


Figure 2.4: Relevant sub-forests resulting from the full decomposition. From [9].

2.5 Relevant SubForests and SubTrees

The recursive decomposition creates relevant sub-forests recursively.

Definition (Relevant sub-forest) The relevant sub-forests are the forests that appear in the recursive calls in the Equation 2.2 and 2.3.

The set of all sub-forest resulting from any decomposition from the forest F is called the full decomposition, denoted by $\mathcal{A}(F)$.

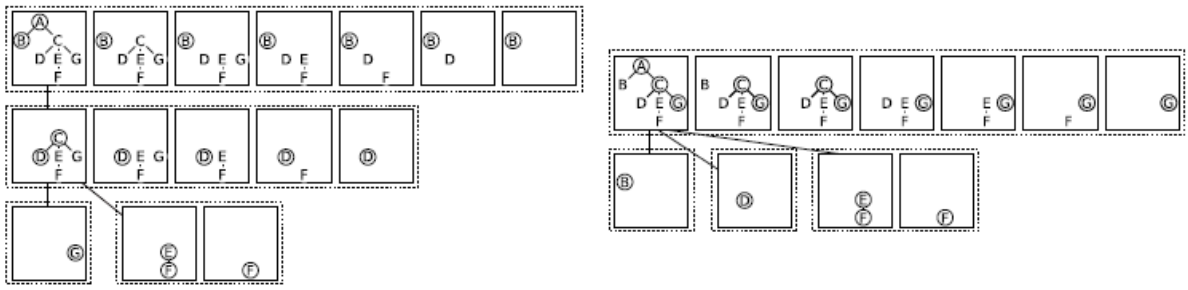
Definition (Full decomposition) The full decomposition of a tree is the set of all sub-forests of F obtained by recursively removing the leftmost and rightmost root nodes, $lr(F)$ and $rr(F)$, from F and the resulting sub-forests.

$$\mathcal{A}(\varnothing) = \varnothing$$

$$\mathcal{A}(F) = \{F\} \cup \mathcal{A}(F - rr(F)) \cup \mathcal{A}(F - lr(F))$$

Figure 2.4 illustrates the relevant sub-forests of a tree resulting from the full decomposition of a tree F.

To decompose a tree, leftmost or rightmost root node can be chosen at each recursive step, resulting in the path decomposition, which is a subset of the full decomposition. Each choice of direction in each step is called a decomposition strategy. The set of decomposition strategies can be indicated by a root-leaf path.



(a) left path decomposition (15 relevant subforests) (b) right path decomposition (11 relevant subforests)

Figure 2.5: Relevant sub-forests. From [9]. (a) Relevant sub-forests from the leftmost path decomposition. (b) Relevant sub-forests from the rightmost path decomposition.

Definition (Root-leaf path) The root-leaf path indicates the choice of decomposition strategy at each step in the recursive call. If the rightmost root of forest F_1 or F_2 is on the path, then $u = lr(F_1)$ and $v = lr(F_2)$ in the Equation 2.3. On the contrary, if the leftmost root of forest F_1 or F_2 is on the path, then $u = rr(F_1)$ and $v = rr(F_2)$

The set of sub-forests decomposed by a root-leaf path from the forest F is called the path decomposition, denoted by $(F)(F)$.

Definition (Path decomposition) Path decomposition is a set of sub-forests of F obtained by recursively removing the leftmost or rightmost root nodes, $lr(F)$ and $rr(F)$, from F and the resulting sub-forests.

$$\mathcal{F}(\varnothing) = \varnothing$$

$$\mathcal{F}(F) = \{F\} \cup \begin{cases} \mathcal{F}(F - rr(F)) \\ \mathcal{F}(F - lr(F)) \end{cases}$$

Figure 2.5(a) is an example of relevant sub-forests that result from the leftmost path decomposition, which successively delete the rightmost root of the resulting forest. This creates 15 relevant sub-forests.Symmetrically, the rightmost path decomposition creates 11 relevant sub-forests(Figure 2.5(b)).

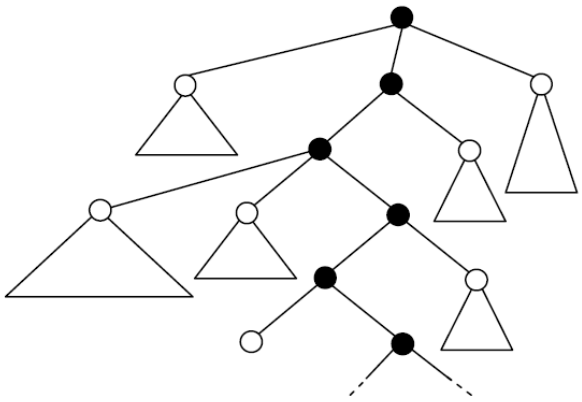


Figure 2.6: Relevant sub-forests. From [4].

Definition (Relevant sub-trees) The relevant sub-trees of tree T for a root-leaf path are all sub-trees that result from removing the path from T, that is, all sub-trees of T that are connected to a node of the path.

As shown in Figure 2.6, the black nodes belong to the root-leaf path and the white nodes are connected to one of the node on the path. The sub-trees rooted at the white nodes are sub-tree result from the path decomposition.

Notation Let T be a tree, γ be a root-leaf path.

- $\Gamma(T)$ denotes the set of roots of relevant sub-trees partitioned by the path γ.

So far, we have considered relevant sub-trees and sub-forests with respect to a single root-leaf path. If a root-leaf path can be defined for each of the resulting sub-trees of tree T, the path decomposition can be recursively applied to all resulting sub-trees. This procedure is called the recursive path decomposition.

2.6 Bottom-up Enumeration

The space-efficient implementations of the tree edit distance use a bottom-up approach, which computes distance between smaller pairs of sub-trees first. In contrast to the top-down decom-

position, the bottom-up enumeration order the sub-trees computation carefully to make sure all the sub problems from the Equation 2.2 and 2.3 have already been computed beforehand.

The order of bottom-up enumeration is closely related to the top-down recursive decomposition. The top-down recursive path decomposition can be categorized into two approaches.

- the recursion direction is fixed to be either leftmost or rightmost,
- the recursion direction may vary between leftmost and rightmost.

For either approach, we need an approach to enumerate the sub-problems. For the fixed direction recursion, one way to enumerate the sub problems is enumerate sub-trees as well as the sub-forests contained in each sub-tree in left-to-right or right-to-left postorder.

- LR-postorder: The sub-trees as well as the sub-forests contained in each sub-trees are enumerated in left-to-right postorder.
- RL-postorder: The sub-trees as well as the sub-forests contained in each sub-trees are enumerated in right-to-left postorder.

Figure 2.7 is an example of relevant sub-forests that result from tree T and each of its relevant sub-trees with respect to the leftmost path. The top-down view gives the recursive right decomposition while the bottom-up gives the left-to-right postorder enumeration.

To enumerate the full decomposition, two approaches can be used, which is suffix-prefix and prefix-suffix postorder enumeration. In prefix-suffix postorder enumeration leftmost root is fixed then enumerate each nodes in the sub-tree, whereas rightmost root is fixed then enumerate each nodes in the sub-tree in suffix-prefix postorder.

- prefix-suffix postorder: Enumerate the rightmost root in left-to-right postorder then enumerate the leftmost root in the tree.
- suffix-prefix postorder: Enumerate the leftmost root in right-to-left postorder then enumerate the rightmost root in the tree.

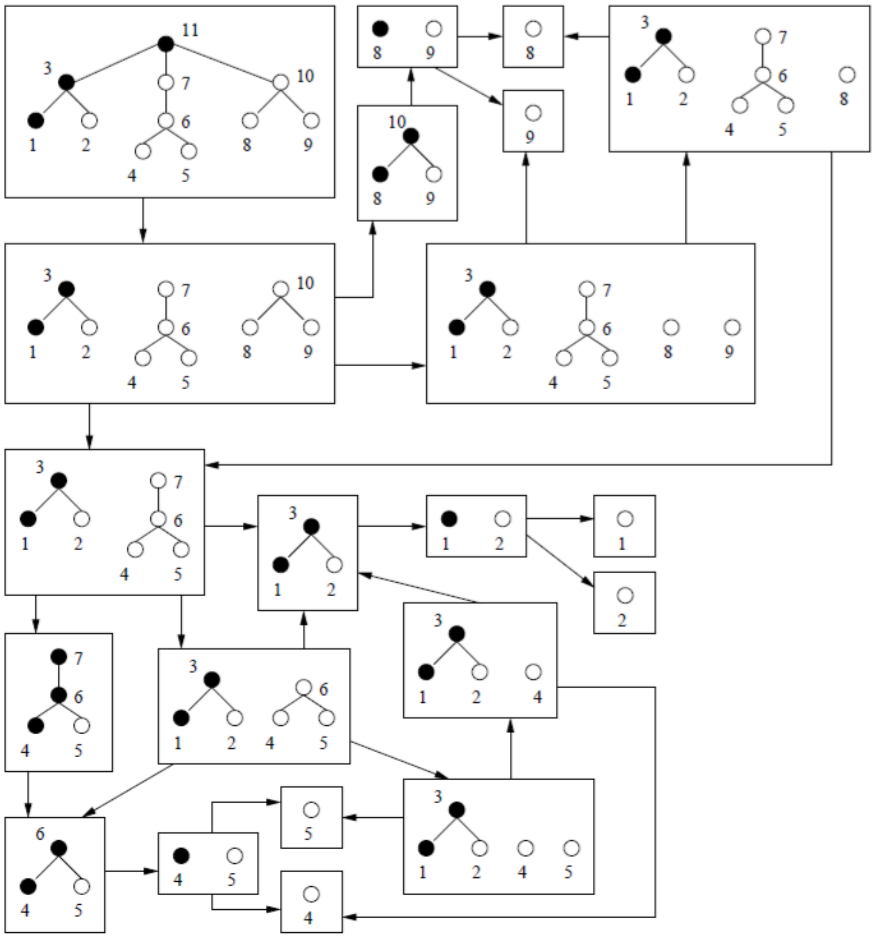


Figure 2.7: Relevant sub-forests that result from the leftmost path decomposition. From [3].

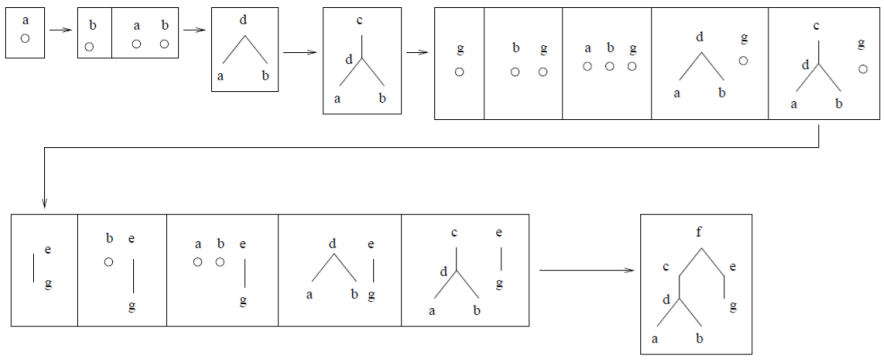


Figure 2.8: An example of enumerating subforests in prefix-suffix postorder. From [3].

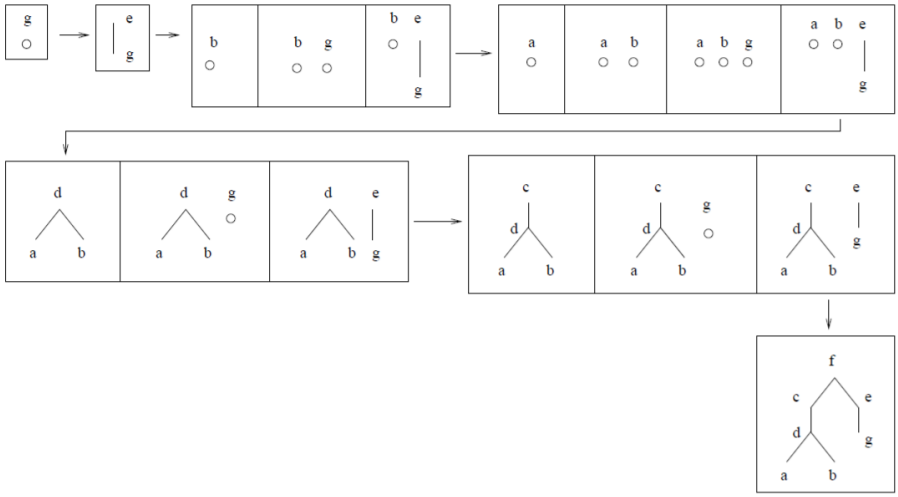


Figure 2.9: An example of enumerating subforests in suffix-prefix postorder. From [3].

Figure 2.8 and Figure 2.9 is the example of prefix-suffix order enumeration and suffix-prefix order enumeration respectively. In Figure 2.10, sub-forests having the same rightmost root are in contiguous boxes while sub-forests having the same leftmost root are in contiguous boxes.

2.7 A Simple Algorithm

In this section, we provide a simple algorithm to compute the tree edit distance, which runs in $O(m^2n^2)$, where m and n are the size of tree. To compute the tree edit distance between two pairs, we need to calculate the distance of all sub-forests pairs, which are enumerate in prefix-suffix or suffix-prefix postorder.

Lemma 2.7.1 *The full decomposition takes $O(|T|^2)$ steps, where $|T|$ is the size of the tree.*

Proof We consider the prefix-suffix postorder only as the suffix-prefix postorder is symmetrical. Let f_i be the number of sub-forests with distinct leftmost roots which contain t_i as the rightmost root. As menetioned in section 2.6, to enumerate the full decomposition in prefix-suffix postorder, first fix the rightmost root then enumerate the leftmost root in the tree in right-to-left postorder. Summing over all nodes to enumerate, we have $\sum_{i=1}^{|T|} \leq \sum_{i=1}^{|T|} |T| = O(|T|^2)$

By enumerating all pairs of sub-forests between two trees, the tree edit distance can be calculated. Here is a simple algorithm runs in $O(m^2n^2)$ time using $O(m^2n^2)$ space.

Algorithm 1: Compute tree edit distance by enumerating all pairs in $O(m^2n^2)$ time.

```
   inputs : (T1, T2), with |T1| = m and |T2| = n
   output: d(T1[i], T2[j]) for 1 ≤ i ≤ m and 1 ≤ j ≤ n
1  L1 ← POSTORDER(T1);
2  L2 ← POSTORDER(T2);
3  for i = 1 to |L1| do
4  |    for j = 1 to |L2| do
5  |    |    compute d(L1[i], L2[j]) as in Equation 2.3
6  |    end
7  end
8  d(T1[i], T2[j]) ← d(L1[|L1|][|L2|]);
9  return d(T1[i], T2[j]);
```

Lemma 2.7.2 *The upper bound of the post-order enumeration method for tree edit distance problem is $O(m^2n^2)$, where m and n are the size of two trees.*

Proof From lemma 2.7.1, the upper bound(full decomposition with no assumption on the strategy) of the number of relevant sub-forests for one tree of size n is $O(n^2)$. Therefore, the upper bound of the relevant sub-forests pairs of trees of sizes m and n respectively is $O(m^2n^2)$. This concludes the proof.

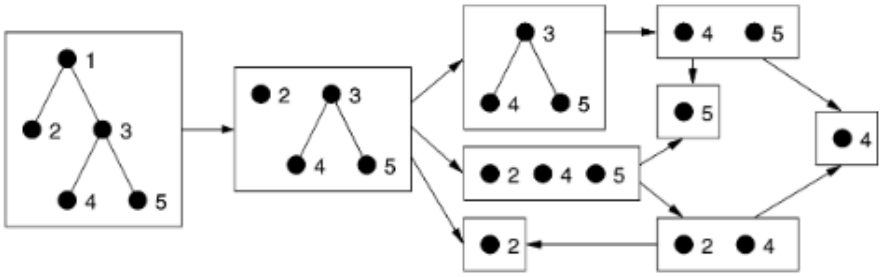


Figure 2.10: Relevant sub-forests that result from the leftmost path decomposition. From [5].

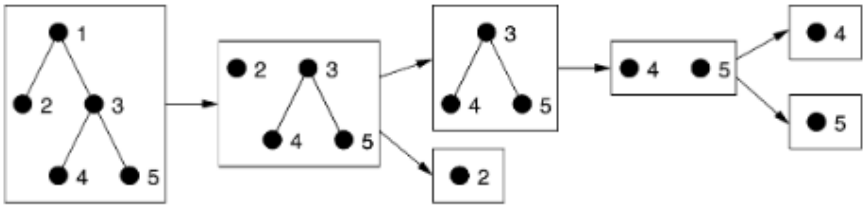


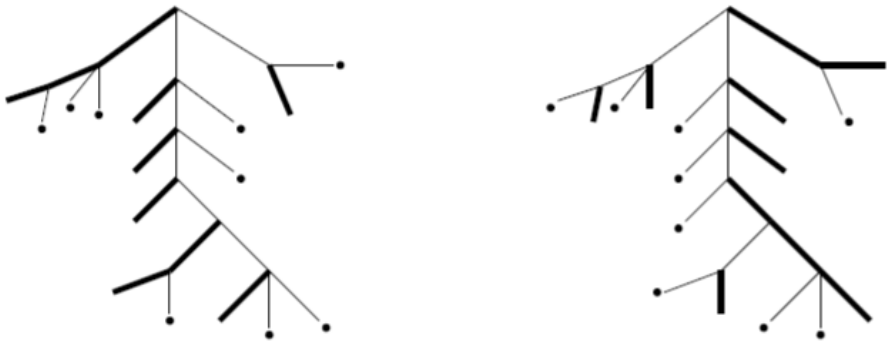
Figure 2.11: Relevant sub-forests that result from the rightmost path decomposition. From [5].

2.8 Improved Algorithmic Path Strategies

Leftmost or rightmost root node is chosen at each recursive step, resulting in the path decomposition, which is the subset of the full decomposition. We use a root-leaf path to indicate the set of decomposition strategies.

Different path decomposition creates different number of relevant sub-forests. Figure 2.10 and Figure 2.11 is an example of different sub-forests that result from different path decomposition strategies. In Figure 2.10, the path is rightmost path, and in Figure 2.11, the path is leftmost path. This gives respectively 7 and 9 sub-forests. Therefore, to make further improvement we look for ways to take advantage of the overlap among sub-forests that are contained in the same sub-tree, and the overlap of sub-trees as well.

The current state-of-the art path strategies are leftmost(rightmost) paths on both tree, heavy paths on one tree as well as heavy paths on both trees.



(a) leftmost paths (b) rightmost paths

Figure 2.12: Leftmost paths and rightmost paths in a tree and its resulting sub-trees From [3].

2.9 Zhang and Shasha's Algorithm

Zhang and Shasha's algorithm [12] fixes the direction in each recursion, say right decomposition. The situation for the recursive left decomposition is symmetrical to that for the recursive right decomposition.

Full decomposition is not needed in the Zhang and Shasha's algorithm since the direction to decompose is fixed. This means that only the path decomposition to the whole tree and its resulting sub-trees is needed instead of the full decomposition. This means that the enumeration of the Zhang and Shasha's algorithm will be in LR-postorder while the symmetrical rightmost paths algorithm used RL-postorder enumeration. See Figure 2.12(a) as an example of the leftmost paths applied the whole tree and its resulting sub-trees and Figure 2.12(b) is the example of rightmost paths.

Taking the advantage of the overlap among sub-trees, all sub-trees sharing the same leftmost leaf can be handled together. In other words, a set of sub-tress along the leftmost path can be handled together with the LR-postorder enumeration to avoid redundant computation.

The root of each relevant sub-trees resulted from the leftmost paths is refereed to as an "LR-keyroot", which is defined as follows.

Definition (LR-keyroots). An LR-keyroot is either the root of T or has a left sibling.

The procedure of the Zhang and Shasha's algorithm as follows. First we identify all LR-

keyroots and sort them in LR-postorder. Then enumerate each pairs of relevant sub-trees in order and calculate the distance between them.

Algorithm 2: Zhang and Shasha's Algorithm

```
inputs : (T1, T2), with |T1| = m and |T2| = n
output: d(T1[i], T2[j]) for 1 ≤ i ≤ m and 1 ≤ j ≤ n
L1 ← LR_KEYROOT_POSTORDER(T1);
L2 ← LR_KEYROOT_POSTORDER(T2);
for i = 1 to |L1| do
    for j = 1 to |L2| do
        TREE_TREE_DISTANCE(T1(L1[i]), T2(L2[j]))
    end
end
d(T1[i], T2[j]) ← d(L1[|L1|][|L2|]);
return d(T1[i], T2[j]);
```

In function TREE_TREE_DISTANCE in Algorithm 2, the leftmost root is fixed and the rightmost roots are enumerated to construct relevant sub-forests. The enumeration algorithm is shown in algorithm 3.

Algorithm 3: TREE_TREE_DISTANCE

```
inputs : (T1(a), T2(b))
output: d(T1(a), T2(b))
L1 ← LR_POSTORDER(T1(a));
L2 ← LR_POSTORDER(T2(b));
for i = 1 to |L1| do
    for j = 1 to |L2| do
        rr_F1 ← L1[i];
        rr_F2 ← L2[j];
        if rr_F1 == lr(T1(a)) ∧ rr_F2 == lr(T2(b)) then
            compute d(T1(rr_F1), T2(rr_F2)) as in Equation 2.2;
        else
            compute d((lr(T1(a), rr_F1), (lr(T2(b)), rr_F2)) as in Equation 2.3;
    end
end
return d(T1(a), T2(b));
```

Lemma 2.9.1 *The time complexity of Zhang and Shasha's algorithm is* $O(mn \times min\{depth(T_1), \#leaves(T_1)\} \times min\{depth(T_2), \#leaves(T_2)\})$

Proof Each sub problems can be solved in constant time. Thus, the time complexity can be counted by the number of sub problems. According to the enumeration scheme from algorithm 3, the number can be counted by the occurring of each node representing as the rightmost root in relevant sub-forests. The maximal times a node representing the rightmost root of relevant sub-forests can be estimated by the maximal number of non-leaf LR-keyroots that a path in the tree may contain. Firstly, since the number of LR-keyroots on any path is bounded by the depth of the path, the number of sub problems is bounded by $\#depth(T)$. Secondly, the number of sub-trees can not exceed the number of leaves as each relevant sub-trees in the Zhang and Shasha's algorithm have distinct leftmost leaves. Therefore, the number of relevant sub-trees decomposed by a leftmost path is bounded by $depth(T)$ or $\#leaves(T)$, whichever is smaller. Therefore the number of sub-forests in either tree is $|T| \times min\{depth(T), \#leaves(T)\}$

Lemma 2.9.2 *The tree edit distance problem can be solved in $O(mn)$ space, where $m = |T_1|$ and $n = |T_2|$*

Proof Dynamic programming method is used to implement, which fills out two $m \times n$ tables. One permanent matrix stores the tree-tree distance and the other temporary matrix stores the forest-forest distance. The temporary forest-forest distance matrix can be overwritten when the computation moves from one pair of relevant sub-trees to another pair. Besides, the permanent tree-tree distance matrix are fetched for use in computing forest-forest distances.

2.10 Klein's Algorithm

Zhang and Shasha's algorithm improves the time complexity by taking the advantage of the overlap of sub-forests with the same leftmost root. However, the running time can be improved in some trees as it is dependent on the shapes of the trees. Kelvin [7] explored and designed a new decomposition strategy based on a type of path called "heavy path". Zhang and Shasha's algorithm ignores shapes of trees and fixes the direction in each decomposition steps while the direction may change in each steps according to different tree shapes.

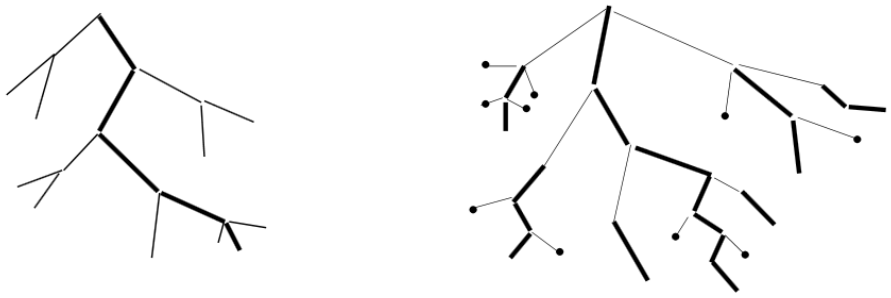


Figure 2.13: Heavy Paths to a tree and its relevant sub-trees From [7].

Heavy child and heavy paths was first introduced by Harel and Tarjan[6]. The definition is as follows.

Definition (Heavy Child) For any node t in Tree T, the heavy child is the root of the largest sub-trees among the sibling sub-trees, denoted by $heavy(t)$.

Definition (Heavy Path) The descending path of Tree T from root to leaf consists of the sequence of nodes $r, heavy(r), heavy(heavy(r)) \cdots$ is called the heavy path, denoted by $P(T)$

Figure 2.13 is an example of heavy paths. In the left picture, a tree's heavy path is indicated in bold. The figure on the right depicts heavy paths applied to the whole tree and its relevant sub-trees.

Let F be a forest denoted as $l(f) \circ t$, where $l(f)$ is the leftmost tree in the forest, t be the rest of the forest and F' be another forest to compare. In Klein's algorithm, the procedure of the decomposition using the heavy path of the tree is as follows:

1. if l belongs to the heavy path, apply right decomposition, otherwise apply left decomposition

2. apply this scheme recursively to all relevant sub-forests of $l(f) \circ t$

Figure 2.14 illustrates the this procedure. For each step, the nodes on the heavy path are indicated by circles.

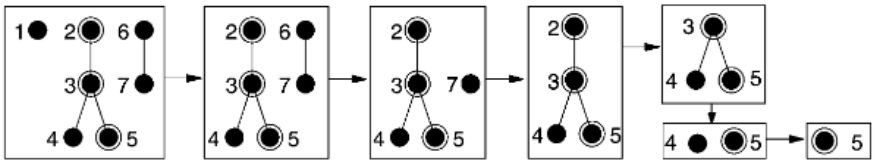


Figure 2.14: Example of decomposition for Klein's algorithm From [7].

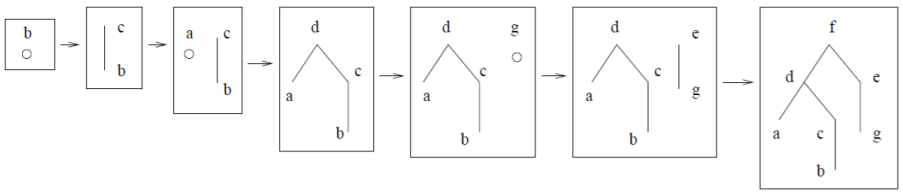


Figure 2.15: Example of enumerating sub-forests in H-postorder From [2].

Symmetrical to Klein's decomposition, right decomposition can be first applied to relevant sub-forests then left decomposition. Let F be a forest denoted as $l(f) \circ t$, where $l(f)$ is the leftmost tree in the forest, t be the rest of the forest and F' be another forest to compare, the procedure is shown as follows:

1. if l belongs to the heavy path, apply left decomposition, otherwise apply right decomposition

2. apply this scheme recursively to all relevant sub-forests of $l(f) \circ t$

Two ways of top-down decomposition give two ways of bottom-up enumeration, referred as "H-postorder". Nodes on the right side of the heavy child can first be enumerated in left-to-right postorder then nodes on the left side are enumerated in right-to-left postorder. Alternatively, the second version is symmetrical to the first one, i.e., right-to-left postorder then left-to-right postorder intermittently. Figure 2.15 is an example of H-postorder enumeration in Klein's algorithm.

As listed in Algorithm 4, the keyroots in the larger tree are sorted in H-postorder. On the other hand, the full decomposition of the smaller tree is sorted in prefix-suffix or suffix-prefix

postorder. Then compute each relevant sub-trees of the larger tree with respect to each relevant sub-forests of the smaller tree.

Algorithm 4: Klein's Algorithm

inputs : (T_1, T_2), with $|T_1| = m$, $|T_2| = n$ and $m \leq n$
output: $d(T_1[i], T_2[j])$ for $1 \leq i \leq m$ and $1 \leq j \leq n$

```
L_1 ← PREFIX_SUFFIX_POSTORDER(T_1);
L_2 ← H_KEYROOT_POSTORDER(T_2);
for i = 1 to |L_1| do
    for j = 1 to |L_2| do
        compute d(L_1[i], L_2[j]) as in Equation 2.3
    end
end
d(T_1[i], T_2[j]) ← d(L_1[|L_1|][|L_2|]);
return d(T_1[i], T_2[j]);
```

Lemma 2.10.1 *Let $h_1, h_2, \cdots h_k$ be any nodes on the same heavy path he same path where h_i is an ancestor of h_j if $i < j$. Then, $\left|T(h_j)\right| \leq |T(h_i)|/2$ if $j = i + 1$*

Proof By definition, heavy child is the root of the largest sub-trees among the sibling sub-trees. Then, $\left|T(h_j)\right| \leq |T(h_i)|/2$ is always true. Otherwise, it is a contradiction to the fact that h_j is a heavy child.

Lemma 2.10.2 *The time complexity of Klein's algorithm is $O(m^2 n \log n)$, where $|T_1| = m$, $|T_2| = n$ and $m \leq n$.*

Proof Each sub problems can be solved in constant time. Thus, the time complexity can be counted by the number of sub problems. According to the enumeration scheme from algorithm 3, the number can be counted by the occurring of each node representing as the rightmost root in relevant sub-forests. The maximal times a node representing the rightmost root of relevant sub-forests can be estimated by the maximal number of non-leaf keyroots that a path in the tree may contain. From Lemma 2.10.1, for each nodes in the heavy path that is being visited, the corresponding subtree size is reduced by at least a factor of 2 with respect to its parent. In

other word, it takes at most $log_2(|T|)$ steps from the root to path. That is to say, the number of sub-forests of the larger tree is bounded by $n \times logn$

Since the direction in each steps is not fixed, the full decomposition rather than the path decomposition of the smaller is needed to be considered, which creates at most m^2 relevant sub-forests.

To sum up, the number of pair of relevant sub-forests is $m^2 n \log n$, which concludes the proof.

Lemma 2.10.3 *The space complexity of Klein's algorithm is $O(mn)$, where $|T_1| = m$, $|T_2| = n$ and $m \leq n$.*

Proof We use one temporary matrix of the size $m \times n$ and one permanent matrix of the size m^2 to store the intermediate result. The permanent matrix stores the tree-tree distance and the temporary one stores the distance between between a specific sub-tree in the larger tree and each relevant sub-forests in the full decomposition of the smaller tree. The temporary matrix can be rewritten when a new sub-tree is enumerated. Additionally, the results in the permanent tree-tree distance matrix are fetched for use in computing forest-forest distances.

2.11 Demaine's Algorithm

Klein algorithm reduces the upper bound on the number of relevant sub-forests required from $O(min\{depth(T), \#leaves(T)\})$ to $O(\log |T|)$ for one tree. However, the cost to this strategy is having to consider all sub-forests(full decomposition) not partial sub-forests(path decomposition) in the other tree. Demaine [4] improved this strategy by a way that applies the heavy path decomposition on both trees.

Let T_1 and T_2 be two trees, assuming that $|T_1| \leq |T_2|$, Demaine's algorithm works as follows:

1. If $|T_1| > |T_2|$, compute $d(T_2, T_1)$

2. Recursively, compute $d(T_1, T_2(k))$ with k being the root of relevant sub-trees.

3. Compute $d(T_1, T_2)$ by enumerating full decomposition of T_1 in prefix-suffix or suffix-prefix postorder, and path decomposition of T_2 in H-postorder.

Lemma 2.11.1 *Let T_1 and T_2 be two trees, $R(T_1, T_2)$ be the number of relevant sub-forests pairs encountered by the algorithm, we have*

$$R(T_1, T_2) \leq 4(|T_1|\,|T_2|)^{\frac{3}{2}}$$

Proof This can be proved by induction on $|T_1| + |T_2|$.

Basis: if $|T_1| + |T_2| = 0$, then both trees are empty and $R(T_1, T_2) = 0$ always holds.

Induction: if $|T_1| + |T_2| > 0$, for the case $|T_1| \geq |T_2|$, we have established

$$R(T_1, T_2) \leq |T_2|^2\,|T_1| + \sum_{v \in \Gamma(T_1)} R(T_1(v), T_2)$$

We consider the first case only for the other case when $|T_1| < |T_2|$ is symmetric. Hence, by the induction hypothesis,

$$R(T_1, T_2) \leq |T_2|^2\,|T_1| + \sum_{v \in \Gamma(T_1)} 4(|T_1(v)|\,|T_2|)^(\frac{3}{2}) \tag{2.4}$$

$$= |T_2|^2\,|T_1| + 4\,|T_2|^{\frac{3}{2}} \sum_{v \in \Gamma(T_1)} |T_1(v)|^{\frac{3}{2}} \tag{2.5}$$

We observed that any tree T has the following two properties:

- $\sum_{v \in \Gamma(T)} |T(v)| \leq |T|$. Because either relevant sub-trees in tree T is disjoint.

- $|T(v)| \leq \frac{|T|}{2}$ for any sub-trees in the set $\Gamma(T)$.

Applying these two inequalities to Equation 2.5, we have

$$\begin{aligned} R(T_1, T_2) &\leq |T_2|^2 |T_1| + 4 |T_2|^{\frac{3}{2}} \sum_{v \in \Gamma(T_1)} max_{v \in \Gamma(T_1)} \sqrt{|T_1(v)|} \\ &\leq |T_2|^2 |T_1| + 4 |T_2|^{\frac{3}{2}} |T_1| \sqrt{\frac{|T_1|}{2}} \\ &= |T_2|^2 |T_1| + \sqrt{8}(|T_1| |T_2|)^{\frac{3}{2}} \\ &\leq 4(|T_1| |T_2|)^{\frac{3}{2}} \end{aligned}$$

Lemma 2.11.2 *The time complexity of Demaine's algorithm is $O(m^2 n(1+\log \frac{n}{m}))$, where $|T_1| = m$, $|T_2| = n$, and $m \leq n$.*

Proof To analyze the time complexity, we count the total number of sub-problems. Step (2) produces recursive calls for each pairs of relevant sub-trees while every new relevant sub-problems is created in step (3). In step (2), the algorithm computes distance between tree pairs $(T_1(v), T_2)$ for all $v \in$. Hence, the number of sub-problems encountered in this step (2) is $\sum_{v \in \Gamma T_1} R(T_1(v), T_2)$. Therefore, we define set $A, B \subset T_1$ as follows:

- $A = \{a \in light(T_1) \cap |T_1(a)| \geq m\}$
- $B = \{b \in T_1 - A\}$

For each $v \in \Gamma(T_1)$, notice that v is either in A or B. It is in A if $|T_1(v)| \geq m$, and in B otherwise. If $v \in A$, all sub-problems arising from the computation of $(T_1(u), T_2)$ for $u \in \Gamma(T_1(v))$. If $v \in B$, the sub-problems is from the recursive call in step (2).For each nodes a in set A, the number of sub-problems produced in step(3) is $|T_2|^2 |T_1(a)|$. Therefore, the total number of sub-problems in set A is $|T_2|^2 \sum_{a \in A} |T_1(a)|$. $\sum_{a \in A} |T_1(a)|$ is bounded by the maximal time of a node in the set A representing the rightmost root of relevant sub-forests. This can be estimated by the maximal number of proper ancestor in the set A that any node in the T_1 may contain. For $v \in T_1$, define $depth_A(v)$ as the number of proper ancestor that is in the set A. We claim that for any $v \in T_1$, $depth_A(v) \leq 1 + \log(\frac{n}{m})$. To prove this, let $a_1, a_2 \cdots a_k$ be any sequences in A

where a_i is a descendant of a_{i-1}. By the definition of set A, we know that for any node a_k in the sequence, $m \leq T_1(a_i) \leq n$, where $i \in [1 \cdots k]$. By Lemma 2.10.1, $a_i \leq \frac{1}{2}a_{i-1}$, where $i \in [1 \cdots k]$. Therefore, $k \leq \log(\frac{n}{m})$. In other words, for $v \in T_1$, $depth_A(v) \leq 1 + \log(\frac{n}{m})$. Therefore, we have the number of relevant sub-forests of the set A

$$|T_1|^2 \sum_{a \in A} |T_1(a)| \leq m^2 \sum_{v \in T_1} (1 + depth_A(v)) \leq m^2 \sum_{v \in T_1} (2 + \log(\frac{n}{m})) = m^2 n(2 + \log(\frac{n}{m}) \tag{2.6}$$

By Lemma 2.11.1, the number of relevant sub-problems in set B is

$$\sum_{b \in B} R(T_1(b), T_2) \leq 4\,|T_2|^{\frac{3}{2}} \sum_{b \in B} |T_1(b)|^{\frac{3}{2}} \tag{2.7}$$

$$\leq 4\,|T_2|^{\frac{3}{2}} \sum_{b \in B} |T_1(b)|\, max_{b \in B} \sqrt{|T_1(b)|} \tag{2.8}$$

$$\leq 4\,|T_2|^{\frac{3}{2}}\,|T_1|\,\sqrt{m} \tag{2.9}$$

$$= 4m^2 n \tag{2.10}$$

Therefore, according to Equation 2.6 and 2.10, the total number of relevant sub-problems is at most

$$m^2 n(2 + \log \frac{n}{m}) + 4m^2 n = O(m^2 n(1 + \log \frac{n}{m}))$$

Remark It has been shown that there exist tree for which $\Omega(m^2 n(1 + \log(n/m)))$, no matter what strategy is used. [4]

2.12 Conclusion

We conclude the state of the art algorithms in tree edit distance problem in Table 2.1. We introduced 3 algorithms in this chapter, and made a conclusion on time and space complexity of these algorithms.

	Time	**Space**	**Comments**
Tai(1979)	$O(n^6)$	$O(n^6)$	first algorithm
Zhang&Shasha(1989)	$O(n^2 \log^2(n))$	$O(n^2)$	efficient for balanced tree
Klein(1998)	$O(n^3 \log(n))$	$O(n^2)$	with no consideration on the shape of the smaller tree
Demaine et al.(2009)	$O(n^3)$	$O(n^2)$	worse case is frequent

Table 2.1: State-of-the-Art Algorithms in Tree Edit Distance Problem

Chapter 3

A new algorithm

Our new algorithm to compute the tree edit distance is introduced in this chapter. It is implemented in C++. The new algorithm consists of two main steps: finding the root-leaf decomposition path and bottom-down enumeration for distance computation.

3.1 Finding the Optimal Root-leaf Decomposition Path

3.1.1 Main Idea

The heavy path decomposition can be seen as a greedy strategy which usually leads to a local optimum. However, a dynamic programming method can be applied to find the global optimum, which saves time complexity in each actual runs. To find the global optimal path of a tree, each possible paths is quantified and the path with the least number of sub-problems are selected to decompose the tree.

3.1.2 Relevant Leftmost Forest and Relevant Rightmost Forest

Two categories of decomposition strategies are introduced in Chapter 2. The one is path decomposition and the other is full decomposition. To quantify the cost of each paths, the actual number of sub-forests resulting from path decomposition and full decomposition is the prereq-

uisite. Relevant leftmost forests, relevant rightmost forests and relevant special forests are first introduced by Dulucq [5].

Definition (Relevant leftmost forests) Let T be a tree. The set of leftmost sub-forests of T is the least set satisfying,

- for each node i of T, T(i) is the leftmost forest,
- if $t \circ l(g)$ is a leftmost sub-forest, then $t \circ g$ is the leftmost sub-forest too.

Symmetrically, we have relevant rightmost forests.

Definition (Relevant rightmost forests) Let T be a tree. The set of rightmost sub-forests of T is the least set satisfying,

- for each node i of T, T(i) is the rightmost forest,
- if $l(g) \circ t$ is a rightmost sub-forest, then $g \circ t$ is the rightmost sub-forest too.

Definition (Relevant special forests) Let T be a tree. The set of special forest is the set of relevant sub-forest resulting from the full decomposition.

Notation Let T be a tree,

- $\#left(T)$ denotes the number relevant leftmost forests.
- $\#right(T)$ denotes the number of relevant rightmost forest.
- $\#spec(T)$ denotes the number of relevant special forest.

3.1.3 Number of Relevant Forests For a Tree

We define $R(T)$ as the set of relevant sub-forests in the tree T. Let T is the tree of the form $l(g)$, where l is the root of the tree, and g is the forest under the root. According to the Equation 2.2, we have the Lemma 3.1.1.

Lemma 3.1.1 *$R(T) = R(l(g)) = \{l(g)\} \cup R(g)$, no matter what the direction is.*

Proof Straightforward implication of the Equation 2.2.

We define $R(F)$ as the set of relevant sub-forest in the forest F. If we consider the left decomposition, F can be the form $l(g) \circ t$, where $l(g)$ be the leftmost tree, and t be the rest of the forest. On the contrary, F can also be of the form $t \circ l(g)$, where $l(g)$ be the rightmost tree, and t be the rest of the forest. Then we have the Lemma 3.1.2.

Lemma 3.1.2

$$R(F) = R(l(g) \circ t) = l(g) \circ t \cup R(g \circ t) \cup R(l(g)) \cup R(t), \ if \ the \ direction \ is \ left.$$

$$R(F) = R(t \circ l(g)) = t \circ l(g) \cup R(t \circ g) \cup R(l(g)) \cup R(t), \ if \ the \ direction \ is \ right.$$

Proof Straightforward implication of the Equation 2.3.

Given a decomposition strategy, the number of relevant sub-forests is a measure of the complexity of the associated edit distance algorithm. To qualify the sub-problems, we denote $\#rel$ the number of relevant forests.

Lemma 3.1.3 *The number of relevant sub-forests of tree F with respect to a root-leaf path is equal to the number of nodes in F.*

Proof The proof is by induction on the size of F.

Basis: by the definition of path decomposition, $|F| = 1$, then $\mathcal{F} = 1$, which is consistent with the Lemma.

Induction: We assume that $|F_k| = k$ holds for forest F_k of size k. Then for the forest F_{k+1} of

size $k+1$, by the definition of path decomposition, we have

$$\begin{aligned} |\mathcal{F}(F_{k+1})| &= |\{F_{k+1}\} \cup \mathcal{F}(F_{k+1} - v)| \\ &= 1 + |\mathcal{F}(F_{k+1} - v)| \\ &= 1 + k \end{aligned}$$

This concludes the proof.

Lemma 3.1.4 *The number of relevant sub-forests produced by a recursive path decomposition of tree T with root-leaf path partitioning is the sum of the sizes of all the relevant sub-trees in the recursive decomposition. Let* $\Gamma(T)$ *be the set of sub-trees partitioned by a root-leaf path.*

$$|\mathcal{F}(T)| = \sum_{T' \in \Gamma(T)} |T'|$$

Proof From the definition of the path decomposition and Lemma 3.1.3.

Remark Let T be a tree,

$$\#right(T) = \sum(|T(i)|, i \in T) - \sum(|T(j)|, \ j \ is \ a \ rightmost \ child).$$

$$\#left(T) = \sum(|T(i)|, i \in T) - \sum(|T(j)|, \ j \ is \ a \ leftmost \ child).$$

Lemma 3.1.5 *Let T be a forest of size n.*

$$\#spec(T) = \frac{n(n+3)}{2} - \sum_{i \in T} |T(i)|$$

Proof The proof is by induction of the size n.

Basis: if $n = 0$, then $\#spec(T) = 0$ always hold.

Induction: if $n > 0$, then $T = l(g) \circ t$. We assume that the sub-forest of T $(g \circ t)$, whose size is $n - 1$, satisfies the induction hypothesis. Therefore, we have

$$\#spec(g \circ t) = \frac{(n-1)(n+1)}{2} - \sum_{i \in g \circ t} |g \circ t(i)|$$

Since $g \circ t$ is a sub-forest of T, this implies

$$\#spec(g \circ t) = \frac{(n-1)(n+1)}{2} - \sum_{i \in g \circ t} |F(i)|$$

By definition of the full decomposition, the set of special forest of T consists of two kinds of sub-forests:

- those containing the node l, which gives $|t| + 1$ sub-forests.
- those not containing the node l, which gives $\#spec(g \circ t)$ sub-forests.

Then we have

$$\#spec(T) = |t| + 1 + \#spec(g \circ t)$$

Note that $|t| + 1$ can be written as $n - |l(g)| + 1$

It follows that

$$\begin{aligned}
\#spec(F) &= n - |l(g)| + 1 + \frac{(n-1)(n+2)}{2} - \sum_{i \in g \circ t} |F(i)| \\
&= n + 1 + \frac{(n-1)(n+2)}{2} - \sum_{i \in F} |F(i)| \\
&= \frac{n(n+3)}{2} - \sum_{i \in F} |F(i)|
\end{aligned}$$

This concludes the proof.

3.1.4 Number of Relevant forests for a Pair of Trees

With the analysis of the number of relevant forests for a tree resulting from the path decomposition and the full decomposition, we are now able to look for the total number of the relevant forest for a pair of trees. Given a pair of trees T_1 and T_2 provided with the root-leaf path for T_1, it appears that all relevant forests of T_1 fall within three categories:

- those that are compared with all rightmost forests of T_2,
- those that are compared with all leftmost forests of T_2,
- those that are compared with all special forests of T_2.

For a tree with root-leaf path, the node on the path inherit sub-forests of another tree from its parent. Therefore, Let T be a tree provided with the root-leaf path, the status of a node in the tree can be separated into four categories depending on the direction and the heritage:

- Free node: the node is the root of T, or is not on the root-leaf path.
- Left node: the node is on the root-leaf path and is the leftmost child of its parent, which inherit leftmost forests of another tree.
- Right node: the node is on the root-leaf path and is the rightmost child of its parent, which inherit rightmost forests of another tree.
- All node: the node is on the root-leaf path and is neither the leftmost child nor the rightmost child of its parent.

Lemma 3.1.6 *Given a pair of trees T_1 and T_2 provided with the root-leaf path for T_1, and i be a free node of T_1:*

1. *if the direction of i is left, then $T_1(i)$ compared with all rightmost forests of T_2.*
2. *if the direction of i is right, then $T_1(i)$ compared with all leftmost forests of T_2.*

Proof Proof by [5]

Lemma 3.1.7 *Given a pair of trees T_1 and T_2 provided with the root-leaf path for T_1, and i be a node of T_1 that is not free, and j be the parent of i:*

1. *if the direction of i is left, if i is the rightmost child of j and $T_1(j)$ is compared with all rightmost forests of T_2, then $T_1(i)$ is compared with all rightmost forests of T_2.*
2. *if the direction of i is right, if i is the leftmost child of j and $T_1(j)$ is compared with all leftmost forests of T_2, then $T_1(i)$ is compared with all leftmost forests of T_2.*
3. *otherwise $T_1(i)$ is compared with all special forests of T_2.*

Proof Proof by [5]

Notation Let A be a tree, i be a node of A, and j be the parent of i(if i is not the root). Let B be another tree, i' be a node of B, and j' be the parent of i'(if i' is not the root). The root-leaf path can on either tree.

- $Free(A(i), B(i'))$ the set of $R(A, B) \cap (A(i), B(i'))$ if i and i' is free
- $RightA(A(i), B(i'))$ the set of $R(A, B) \cap (A(i), B(i'))$ if i is on the root-leaf path and is the rightmost child of j
- $RightB(A(i), B(i'))$ the set of $R(A, B) \cap (A(i), B(i'))$ if i' is on the root-leaf path and is the rightmost child of j'
- $LeftA(A(i), B(i'))$ the set of $R(A, B) \cap (A(i), B(i'))$ if i is on the root-leaf path and is the leftmost child of j
- $LeftB(A(i), B(i'))$ the set of $R(A, B) \cap (A(i), B(i'))$ if i' is on the root-leaf path and is the leftmost child of j'
- $AllA(A(i), B(i'))$ the set of $R(A, B) \cap (A(i), B(i'))$ if i is on the root-leaf path and neither the rightmost nor leftmost child of j

- $AllB(A(i), B(i'))$ the set of $R(A, B) \cap (A(i), B(i'))$ if i' is on the root-leaf path and neither the rightmost nor leftmost child of j'

With the notation, we are now able to formulate the main result of the section, which gives the total number of relevant forests for a root-leaf path.

Theorem 3.1.8 *Let(A, B) be a pair of trees, and the root-leaf is on either tree.*

1. *if A and B is reduced to a single node*

$$\begin{aligned} Free(A, B) &= LeftA(A, B) = LeftB(A, B) \\ &= RightA(A, B) = RightB(A, B) \\ &= AllA(A, B) = AllB(A, B) = 1 \end{aligned}$$

2. *if A is reduced to a single node, and B is a tree of the form $B = l(B')$, where B' is a sub-tree.*

$$\begin{aligned} Free(A, B) &= min \begin{cases} RightB(A, B') + \#right(A) \\ LeftB(A, B') + \#left(A) \end{cases} \\ LeftA(A, B) &= \#left(B) \\ LeftB(A, B) &= LeftB(A, B') + \#left(A) \\ RightA(A, B) &= \#right(B) \\ RightB(A, B) &= RightB(A, B') + \#right(B) \\ AllA(A, B) &= \#spec(B) \\ AllB(A, B) &= AllB(A, B') + \#spec(A) \end{aligned}$$

3. *if A is reduced to a single node, and B is a tree of the form $B = l(B_1 \circ \cdots \circ B_n)$, where $B_1, B_2 \cdots B_n$ are sub-trees.*

$$Free(A,B) = min \begin{cases} \sum_{i>1} Free(A,B_i) + LeftB(A,B_1) + \#left(A) * |B - B_1| \\ \sum_{i \neq j} Free(A,B_i) + AllB(A,B_j) \\ \quad +min \begin{cases} \#right(A) * (1 + \left|B_1 \circ \cdots \circ B_{j-1}\right|) + \#spec(A) * (\left|B_{j+1} \circ \cdots \circ B_n\right|) \\ \#left(A) * (1 + \left|B_n \circ \cdots \circ B_{j+1}\right|) + \#spec(A) * (\left|B_1 \circ \cdots \circ B_{j-1}\right|) \end{cases} \\ \sum_{i<n} Free(A,B_i) + RightB(A,B_n) + \#right(A) * |B - B_n| \end{cases}$$

$$LeftA(A,B) = \#left(B)$$

$$LeftB(A,B) = \sum_{i>1} Free(A,B_i) + LeftB(A,B_1) + \#left(A) * (|B - B_1|)$$

$$RightA(A,B) = \#right(B)$$

$$RightB(A,B) = \sum_{i<n} Free(A,B_i) + RightB(A,B_n) + \#left(B) * (|B - B_n|)$$

$$AllA(A,B) = \#spec(B)$$

$$AllB(A,B) = min \sum_{i \neq j} Free(A,B_i) + AllB(A,B_j) + \#spec(A) * (\left|A - A_j\right|)$$

4. *if A is a tree of the form $A = l(A')$, and B is a tree of the form $B = l(B_1 \circ \cdots \circ B_n)$, where*

$B_1, B_2 \cdots B_n$ *are sub-trees.*

$$
\begin{aligned}
Free(A,B) = min \begin{cases}
Free(A',B) + min \begin{cases} \#left(B) \\ \#right(B) \end{cases} \\
\sum_{i>1} Free(A,B_i) + LeftB(A,B_1) + \#left(A) * (|B - B_1|) \\
\sum_{i \neq j} Free(A,B_i) + AllB(A,B_1) \\
\quad + min \begin{cases} \#right(A) * (1 + \left|B_1 \circ \cdots \circ B_{j-1}\right|) + \#spec(A) * (\left|B_{j+1} \circ \cdots \circ B_n\right|) \\ \#left(A) * (1 + \left|B_n \circ \cdots \circ B_{j+1}\right|) + \#spec(A) * (\left|B_1 \circ \cdots \circ B_{j-1}\right|) \end{cases} \\
\sum_{i<n} Free(A,B_i) + RightB(A,B_n) + \#right(A) * (|B - B_n|)
\end{cases}
\end{aligned}
$$

$$LeftA(A,B) = LeftA(A',B) + \#left(B)$$

$$LeftB(A,B) = \sum_{i>1} Free(A,B_i) + LeftB(A,B_1) + \#left(A) * (|B - B_1|)$$

$$RightA(A,B) = RightA(A',B) + \#right(B)$$

$$RightB(A,B) = \sum_{i<n} Free(A,B_i) + RightB(A,B_n) + \#right(A) * (|B - B_n|)$$

$$AllA(A,B) = AllA(A',B) + \#spec(B)$$

$$AllB(A,B) = min \sum_{i \neq j} Free(A,B_i) + AllB(A,B_j) + \#spec(A) * (\left|B - B_j\right|)$$

5. *if A is a tree of the form* $A = l(A_1 \circ \cdots \circ A_n)$*, and B is a tree of the form* $B = l'(B_1 \circ \cdots \circ$

B_n)

$$
Free(A,B) = min \begin{cases}
\sum_{i>1} Free(A_i,B) + LeftA(A_1,B) + \#left(B) * (|A - A_1|) \\
\sum_{i\neq j} Free(A_i,B) + AllA(A_j,B) \\
\quad +min \begin{cases} \#right(B) * (1 + \left|A_1 \circ \cdots \circ A_{j-1}\right|) + \#spec(B) * (\left|A_{j+1} \circ \cdots \circ A_n\right|) \\ \#left(B) * (1 + \left|A_n \circ \cdots \circ A_{j+1}\right|) + \#spec(B) * (\left|A_1 \circ \cdots \circ A_{j-1}\right|) \end{cases} \\
\sum_{i<n} Free(A_i,B) + RightA(A_n,B) + \#right(B) * (|A - A_n|) \\
\sum_{i>1} Free(A,B_i) + LeftB(A,B_i) + \#left(A) * (|B - B_1|) \\
\sum_{i\neq j} Free(A,B_i) + AllB(A,B_j) \\
\quad +min \begin{cases} \#right(A) * (1 + \left|B_1 \circ \cdots \circ B_{j-1}\right|) + \#spec(A) * (\left|B_{j+1} \circ \cdots \circ B_n\right|) \\ \#left(A) * (1 + \left|B_n \circ \cdots \circ B_{j+1}\right|) + \#spec(A) * (|B_1 \circ \cdots \circ B_n|) \end{cases} \\
\sum_{i<n} Free(A,B_i) + RightB(A,B_n) + \#right(A) * (|B - B_n|)
\end{cases}
$$

$$
LeftA(A,B) = \sum_{i>1} Free(A_i,B) + LeftA(A_1,B) + \#left(B) * (|A - A_1|)
$$

$$
LeftB(A,B) = \sum_{i>1} Free(A,B_i) + LeftB(A,B_1) + \#left(A) * (|B - B_1|)
$$

$$
RightA(A,B) = \sum_{i<n} Free(A_i,B) + RightA(A_n,B) + \#right(B) * (|A - A_n|)
$$

$$
RightB(A,B) = \sum_{i<n} Free(A,B_i) + RightA(A,B_n) + \#right(A) * (|B - B_n|)
$$

$$
AllA(A,B) = min \sum_{i\neq j} Free(A_i,B) + AllA(A_j,B) + \#spec(B) * (\left|A - A_j\right|)
$$

$$
AllB(A,B) = min \sum_{i\neq j} Free(A,B_i) + AllB(A,B_j) + \#spec(A) * (\left|B - B_j\right|)
$$

3.1.5 Dynamic Programming Implementation

To compute the cost of each possible root-leaf paths, a dynamic programming way to implement is to define seven tables of size $n*m$ to store intermediate results, where n and m is the size of tree A and tree B respectively. For any pair of trees (T_1, T_2), define dynamic programming tables $Free$, $RightA$, $RightB$, $LeftA$, $LeftB$, $AllA$ as well as $AllB$ indexed by nodes of tree A and B. The definition of these seven tables borrowed from the Theorem 3.1.8. The algorithm loops over every pair of sub-trees in post-order of the nodes $i \in A$ and $j \in B$. At each step, the favorite node on the root-leaf path is chosen to be the child that minimizes the number of relevant forests. For instance, if $A = l(A_1 \circ \cdots \circ A_n)$ and $B = l'(B_1 \circ \cdots \circ B_n)$ then, the favorite child is selected to be the root of the sub-tree in tree A or B. The optimal root-leaf of each pairs of sub-trees can be built up by tracing back from table $Free(A, B)$.

Lemma 3.1.9 *The time complexity of this pre-processing is in $O(mn)$, where m and n is the size of tree A and B respectively.*

Proof Firstly, $\#right(A)$, $\#left(A)$ and $\#spec(A)$ are needed to compute. This can be made in $O(m)$. Similarly, $\#right(B)$, $\#left(B)$ and $\#spec(B)$ can be computed in $O(n)$. Next, we need to fill up seven array of the size $m * n$. Each cells stores the number relevant sub-forests of a pair of sub-trees in tree A and B. The time to fill up the cell of each cell of node i in the tree A and node j in the tree B is proportional to the sum of the degree of i and the degree of j. In other words, the time for the computation of each tables is in $O(n \sum_{i \in A} deg(i) + m \sum_{j \in B} deg(j))$, that is in $O(mn)$.

3.2 Distance Computation in Bottom-up Fashion

Once the optimal root-leaf path is set up, it is possible to compute the distance.

3.2.1 Double Roots Encoding

We use root encoding to identify all sub-forests that can result from decomposing two trees using Equation 2.2 and 2.3.

Definition (Double roots encoding) Let F be a forest, the root of the leftmost tree $lr(F)$ and the root of the rightmost tree $rr(F)$ be two nodes of forest F, $lr(F) \leq rr(F)$. The root encoding $F_{lr(F),rr(F)}$ defines a sub-forest of F with a set of nodes and edges. The set of nodes is defined as follows. Let x be nodes in the forest and x succeeds $lr(F)$ in left-to-right pre-order and x succeeds $rr(F)$ in right-to-left pre-order.

$$N(F_{lr(F),rr(F)}) = \{lr(F), rr(F)\} \cup \{x\}$$

The edges set is defined as follows.

$$E(F_{lr(F),rr(F)}) = \{(v, w) \in E(F) | v \in F_{lr(F),rr(F)} \cap w \in F_{lr(F),rr(F)}\}$$

Figure 3.1 is an example of sub-forests of tree G and their root encoding representation. The figure on the left is a sub-forest in G(black nodes) while the right on is a sub-tree. In the figures, the leftmost root and the rightmost root are marked with arrows. The left subscript of a node is its left-to-right and the right subscript its right-to-left pre-order. Take the left figure as an example, The left-to-right pre-order of the leftmost root is 2 and the right-to-left pre-order of rightmost root is 4. As shown in the right figure, if the right-to-left pre-order of the leftmost root is the same as the left-to-right pre-order of the rightmost root, the forest becomes a tree.

3.2.2 Bottom-up Enumeration

We presented the double root encoding for indexing sub-forests in the last section. In this section we make the use of this indexing scheme and present an algorithm for the enumeration of sub-forests pairs in bottom-up fashion.

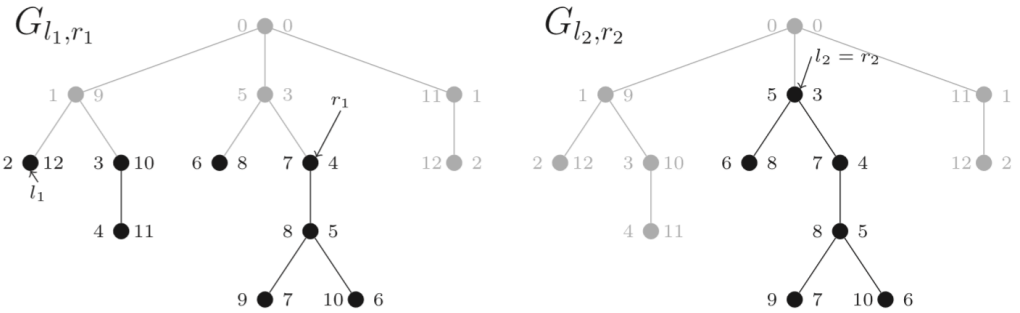


Figure 3.1: Example subforests of tree G and their root encoding representation [9].

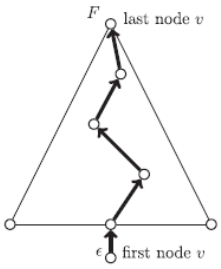


Figure 3.2: The order of processing nodes in loops A [9].

The input of the algorithm are two trees A and B and a root-leaf path γ of tree A. In algorithm 5, $p(v)$ is defined as the parent of the node v.

The sub-problems are produced in nested loops and are nested as follows:$A(B(C(D)), B'(C'(D')))$. A relevant sub-forests pair is defined of the form (l_A, r_A, l_B, r_B) using double roots encoding. Each loops enumerate one of these nodes. In the innermost loop, the distance for the relevant sub-forests pair is computed via Equation 2.2 and 2.3.

Loop A iterates bottom-up over the nodes of path γ staring with a dummy leaf node ϵ which is appended to the leaf node of the path γ. The dummy is used for prevent the leaf node being treated as a special case. Figure 3.2 illustrates the order of processing nodes in loops A.

Loop C enumerates the nodes to the left of path γ. The nodes set $left(A(p(v)), v)$ for a node enumerated in loop A is illustrated in Figure 3.3. Loop C(Figure 3.3(a)) defines the leftmost root node l_A while the rightmost root node $r_A = v$ is defined by loop A. Symmetrically, loop C'(Figure 3.3(b)) iterates over the nodes from the set $right(A(p(v), v) \cup p(v)$ and defines the

Algorithm 5: SUB_FOREST_PAIRS_ENUMERATION

inputs : A, B, γ

foreach *node* $v \in A$ *on the path from* ϵ *to root* **do**
  $B' \leftarrow B$;
  $l_A^{last} \leftarrow \emptyset$;
  $l'_A \leftarrow v$;
  if γ *is right or inner* **then**
    foreach *node* $r_B \in B$ *in reverse right − to − left preorder* **do**
      if γ *is right* **then**
        $l_A^{last} \leftarrow p(v)$;
        if r_B *is the rightmost child of its parent* **then** $B' \leftarrow B(p(r_B))$;
        else $B' \leftarrow \emptyset$;
        $r_A \leftarrow v$;
      end
      foreach *node* $l_A \in left(A(p(v)), v) \cup l_A^{last}$ *in reverse left − to − right preorder* **do**
        if $l_A = p(v)$ **then** $r_A \leftarrow p(v)$;
        foreach *node* $l_B \in \{r_B\} \cup left(G, r_B)$ *in reverse left − to − right preorder* **do**
          Compute forest-to-forest distance $d(l_A, r_A, l_B, r_B)$ as in Equation 2.3;
        end
        $l'_A \leftarrow l_A$;
      end
    end
  end
  if γ *is left or inner* **then**
    foreach *node* $l_B \in B$ *in reverse right − to − left preorder* **do**
      if γ *is left* **then**
        if l_B *is the leftmost child of its parent* **then** $B' \leftarrow B(p(l_B))$;
        else $B' \leftarrow \emptyset$;
      end
      $l_A \leftarrow l'_A$;
      foreach *node* $r_A \in right(A(p(v)), v) \cup p(v)$ *in reverse right − to − left preorder* **do**
        if $r_A = p(v)$ **then** $l_A \leftarrow p(v)$;
        foreach *node* $r_B \in l_B \cup right(B', l_B)$ *in reverse right − to − left preorder* **do**
          Compute forest-to-forest distance $d(l_A, r_A, l_B, r_B)$ as in Equation 2.3;
        end
      end
    end
  end
end

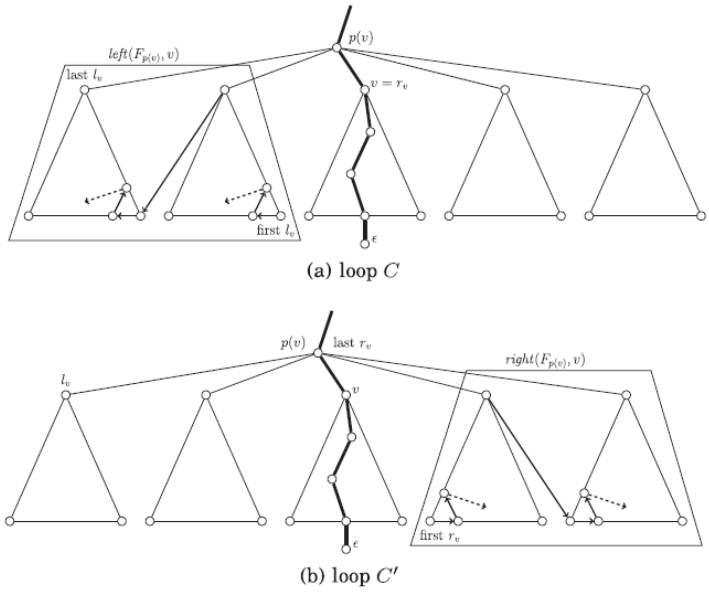


Figure 3.3: The order of processing nodes in loop C and C'. [9].

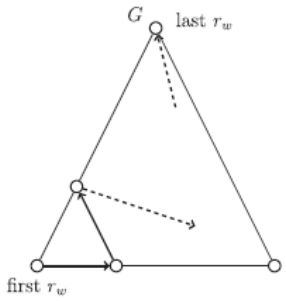


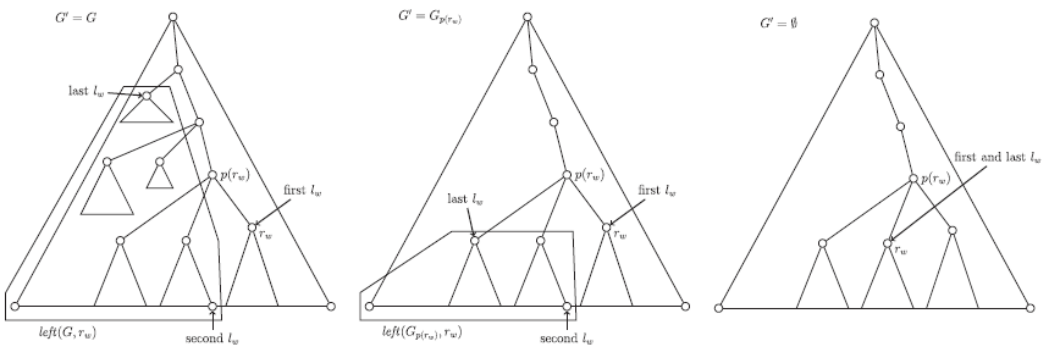
Figure 3.4: The order of processing nodes in loop B. [9].

rightmost root node of the sub-forest, while the leftmost root node is the leftmost child of $p(v)$.

Loop B and loop D defines sub-forests of the tree B. The rightmost root node of sub-forest is iterates over all nodes of tree B in reverse right-to-left preorder(Figure 3.4). After fixing the rightmost root node in loop B, the leftmost root node is iterated over all nodes to the left of leftmost root node in a specific sub-tree B'.

The specific sub-tree B' defines the set of nodes to be enumerated in loop D. The definition of B' depends on the type of path γ and the position of r_B in B.

- Inner path $B' = B$(Figure 3.5(a))



(a) $G' = G$ (b) $G' = G_{p(r_w)}$ (c) $G' = \varnothing$

Figure 3.5: The order of processing nodes in loop D. [9].

- Right path. If the rightmost root node v is the rightmost child of its parent, then $B' = B(p(v))$(Figure 3.5(b)). Otherwise, $B' = \varnothing$(Figure 3.5(c)).

Left paths are treated in loop B' and D', which are symmetric to loop B and D.

Algorithm 5 first enumerates the nodes to the left of the path then nodes to the right of the path in A. However, a symmetric version of Algorithm 5 goes on the other way: first iterates nodes to the right of the path then nodes to the right of the path. Since the directions of each nodes is not always the same, the symmetric version of Algorithm 5 is included in our project as well.

3.2.3 A Quadratic Space Complexity Implementation

We have presented an algorithm for the enumeration of sub-forests pairs. It gives rise to two data structures:

- a permanent array D that stores the distance between a pair of sub-trees in two trees.
- a temporary array F that stores the distance between a pair of sub-forests that are attached to the computation of a pair of sub-trees.

Let A and B be two trees, and γ be the root-leaf path on tree A. At the first glance, the array F requires cubic space, when the node on the root-leaf path is neither the leftmost child, nor

the rightmost child, as the full decomposition of another tree is required in these cases.But it can be made quadratic with the introduction of two memorization tables: T and Q of sizes $|B|^2$ and $|A|$, where A and B are two trees.

- T stores the distance between a specific sub-forest in A and each relevant sub-forests in B.

- Q stores the distance between each sub-forests of A defined in loop C and a specific sub-forest in G, that is G°

The tables T and Q are maintained in each loops C and C', which are needed in different calls of loops A and loops B(B'). In each loops A and B(B'), the intermediate results computed in the last loops are retrieved and is used for computation. Then, the new results are updated in T and Q for the next loop.

We now show how to retrieve intermediate result with the help of memorization tables T and Q. Before we introduce rules for obtaining the required distances when computing each pairs of relevant sub-forest, it is beneficial to recall the formula used to compute the distance of relevant sub-forest pairs. If double roots encoding scheme is applied to index forests, the Equation 2.2 will change to Equation 3.1 an 3.2.

Let A and B be two trees.

$$d(A_{lA,rA}, B_{lB,rB}) = min \begin{cases} d(A_{lA,rA} - lA, B_{lB,rB}) + \delta(lA, \varnothing) \\ d(A_{lA,rA}, B_{lB,rB} - lB) + \delta(\varnothing, lB) \\ d(A_{lA,rA} - A(lA), B_{lB,rB} - B(lB)) + d(A(lA), B(lB)) \end{cases} \tag{3.1}$$

Symmetrically, if right decomposition applied to the forests, the equation changes to

$$d(A_{lA,rA}, B_{lB,rB}) = min \begin{cases} d(A_{lA,rA} - rA, B_{lB,rB}) + \delta(rA, \varnothing) \\ d(A_{lA,rA}, B_{lB,rB} - rB) + \delta(\varnothing, rB) \\ d(A_{lA,rA} - A(rA), B_{lB,rB} - B(rB)) + d(A(rA), B(rB)) \end{cases} \tag{3.2}$$

As can be seen in the Equation 3.1, four intermediate results are needed to retrieve in each recursive calls, $d(A_{lA,rA} - lA, B_{lB,rB})$, $d(A_{lA,rA}, B_{lB,rB} - lB)$, $d(A(lA) - lA, B(lB) - lB)$ and $d(A_{lA,rA} - A(lA), B_{lB,rB} - B(lB))$ respectively. Please note that we only consider the left decomposition for the right decomposition case is symmetric.

$$d(A_{lA,rA} - lA, B_{lB,rB}) = \begin{cases} \sum_{v \in B_{lB,rB}} \delta(\varnothing, v) \; if \; A_{lA,rA} - lA = \varnothing \\ T[lB, rB] \; if \; A_{lA,rA} - lA \; is \; a \; tree \\ F[a, lB] \; otherwise \end{cases} \tag{3.3}$$

Note: $a \in A$ such that $A_{a,rA} = A_{lA,rA} - lA$

$$d(A_{lA,rA}, B_{lB,rB} - lB) = \begin{cases} \sum_{v \in A_{lA,rA}} \delta(v, \varnothing) \; if \; B_{lB,rB} - lB = \varnothing \\ Q[lA] \; if \; B_{lB,rB} \; is \; a \; tree \\ F[l_A, b] \; otherwise \end{cases} \tag{3.4}$$

Note: $b \in B$ such that $B_{b,rB} = B_{lB,rB} - lB$

$$d(A(lA), B(lB)) = \begin{cases} \sum_{v \in B(lB)} \delta(\varnothing, v) \; if \; A_{lA,rA} - lA = \varnothing \\ \sum_{v \in A(lA)} \delta(v, \varnothing) \; if \; B_{lB,rB} - lB = \varnothing \\ D[lA, lB] \; otherwise \end{cases} \tag{3.5}$$

$$d(A_{lA,rA} - A(lA), B_{lB,rB} - B(lB)) = \begin{cases} 0 \; if A_{lA,rA} - A(lA) \cap B_{lB,rB} - B(lB) = \varnothing \\ \sum_{v \in B_{lB,rB} - B(lB)} \delta(\varnothing, v) \; if A_{lA,rA} - A(lA) = \varnothing \\ \sum_{v \in A_{lA,rA} - A(lA)} \delta(v, \varnothing) \; if B_{lB,rB} - B(lB) = \varnothing \\ T[y, rB] \; if A_{lA,rA} - A(lA) \; is \; a \; tree \\ F[x, y] \; otherwise \end{cases} \tag{3.6}$$

Note: $x \in A$, such that $A_{x,rA} = A_{lA,rA} - A(lA)$, $y \in B$, such that $B_{y,rB} = B_{lB,rB} - B(lB)$.

Chapter 4

Another algorithmic improvement

An algorithmic improvement can be applied to our algorithm.

4.1 Main Idea

We notice that a large number of non-branching nodes exist in the tree representation of the RNA secondary structure. Therefore, we can construct compact representation for trees, which can aid in improving the running time for computing the tree edit distance.

4.2 Vertical Reduction on Trees

Before we introduce the reduction on trees, it is benefit to define the compressible path on trees.

Definition (Maximal Non-branching Path)A path in a tree is a non-branching path if both the post-order tree traversal and pre-order tree traversal visit the nodes on the path in consecutive order. A non-branching path is maximal if no other non-branching path contains it [3].

Figure 4.1 is an example of maximal non-branching path. Each maximal non-branching paths is enclosed by dashed line.

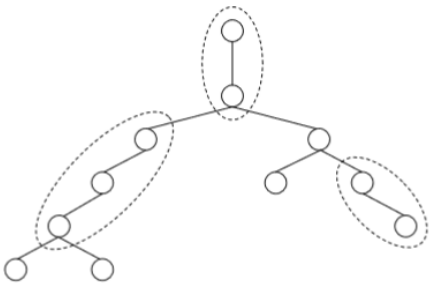


Figure 4.1: Maximal non-branching path [3].

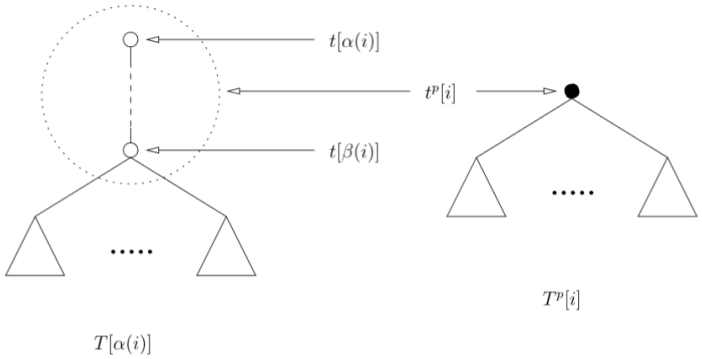


Figure 4.2: An example of the mapping of nodes between the original tree and its vertical reduced tree [3].

Each maximal non-branching paths is compressible, which gives rise to the vertical reduction.

Definition (Vertical Reduction) The vertical reduction on a tree is to replace every maximal non-branching path in the tree by a single node [3].

In Figure 4.2, we give an example of the mapping of nodes between the original tree(n the left) and its tree(on the right). Given a vertical reduced tree $\widetilde{T}$, two functions are respectively defined to map a node in compressed tree $\widetilde{t}[i]$ to the highest indexed node in the original tree T $t[\alpha(i)]$ and the lowest indexed node $t[\beta(i)]$ of in the original tree T. When $\widetilde{(t)}[i]$ corresponds to a single node in T, $t[\alpha(i)] = t[\beta(i)]$.

Please note that the compressed tree is just a compact representation of the original. Therefore, the strategy-based strategy in chapter 4 applied to the compressed tree can get the result same as that applied to the original tree. In other words, given trees (T_1, T_2) and their reduced

tree($\widetilde{T_1}, \widetilde{T_2}$), the relation $d(\widetilde{T_1}, \widetilde{T_2}) = d(T_1, T_2)$ is implied.

4.3 Computation

Following the Equation 2.3, let $\widetilde{F_1}$ and $\widetilde{F_2}$ be two relevant sub-forests in compressed trees, u and v are nodes in forests $\widetilde{F_1}$ and $\widetilde{F_2}$ respectively, the forest-to-forest distance can be computed as follows:

$$d(\widetilde{F_1}, \widetilde{F_2}) = min \begin{cases} d(\widetilde{F_1} - u, \widetilde{F_2}) + \delta(u, \varnothing) \\ d(\widetilde{F_1}, \widetilde{F_2} - v) + \delta(\varnothing, v) \\ d(\widetilde{F_1} - \widetilde{F_1(u)}, \widetilde{F_2} - \widetilde{F_2(v)}) + d(\widetilde{F_1(u)}, \widetilde{F_2(v)}) \end{cases} \tag{4.1}$$

Each pairs of forest-forest distance $d(\widetilde{F_1}, \widetilde{F_2})$ can be computed correctly since each sub-problems has already been computed beforehand if implemented in post-order. The tree-tree distance, $d(\widetilde{F_1(u)}, \widetilde{F_2(v)})$, however, have never been computed before and must compute its value.

The tree-tree distance in compressed tree can be computed in the following lemmas.

Lemma 4.3.1 *$d(\widetilde{F_1(i)}, \widetilde{F_2(j)}) = d(F_1(\alpha(i), F_2(\alpha(j))))$, where i is a node in $\widetilde{F_1}$ and j is a node in $\widetilde{F_2}$*

Proof The result follows from the compressed tree definition and the post-order implementation.

To compute tree-to-tree distance of each sub-trees pairs in original trees, following Equation 2.2, we have Equation 4.2.

Lemma 4.3.2 *$\forall u \in \{\beta(i), \cdots, \alpha(i)\}$ where i is a node in $\widetilde{F_1}$, and $\forall v \in \{\beta(j), \cdots, \alpha(j)\}$ where j*

is a node in $\widetilde{F_2}$.

$$d(F_1(u), F_2(v)) = min \begin{cases} d(F_1^\circ(u), F_2(v)) + \delta(u, \varnothing) \\ d(F_1(u), F_2^\circ(v)) + \delta(\varnothing, v) \\ d(F_1^\circ(u), F_2^\circ(v)) + \delta(u, v) \end{cases} \tag{4.2}$$

Proof The known recursive solution for the tree-to-tree edit distance.

The computation of $d(F_1(u), F_2(v))$ involves sub-problems $d(F_1(u), F_2^\circ(\beta(j))), \forall u \in \{\beta(i), \cdots, \alpha(i)\}$ and $d(F_1^\circ(\beta(i)), F_2(v)), \forall v \in \{\beta(i), \cdots, \alpha(i)\}$ for the first time and therefore must compute their values beforehand. Therefore, initialization should be done to compute these values.

Lemma 4.3.3 *Let* $F_2^\circ(\beta(j))$ *be the form* $F_2(j_1) \circ F_2(j_2) \cdots F_2(j_l)$*, where* $j \in \widetilde{F_2}$ *and* $j_1, j_2, \cdots, j_l$ *be children of node* j*,* $\forall u \in \{\beta(i), \cdots \alpha(i)\}$*, where* $i \in \widetilde{F_1}$*,*

$$d(F_1(u), F_2^\circ(\beta(j))) = min \begin{cases} d(F_1^\circ(u), F_2^\circ(\beta(j))) + \delta(u, \varnothing) \\ min_{j_1' \le q \le j_l} \{d(F_1(u), F_2(\alpha(q))) - d(\varnothing, F_2(\alpha(q)))\} + d(\varnothing, F_2^\circ(\beta(j))) \end{cases} \tag{4.3}$$

Proof The edit distance between tree $F_1(u)$ and the forest $F_2^\circ(\beta(j))$ consists of two possible cases. In the first case, u is constrained to be deleted and the remaining substructure $F_1^\circ(u)$ is matched to $F_2^\circ(\beta(j))$. In the second case, u is constrained to match a node somewhere in $F_2^\circ(\beta(j))$. In other words, tree $F_1(u)$ is matched to a sub-tree in $F_2^\circ(\beta(j))$. Therefore, the second case is finding a sub-tree in $F_2^\circ(\beta(j))$ to be matched to $F_1(u)$ so as to minimize $d(F_1(u), F_2^\circ(\beta(j)))$ under such constraint.

Lemma 4.3.4 *Let* $F_1^\circ(\beta(i))$ *be the form* $F_1(i_1) \circ F_1(i_2) \cdots F_1(i_k)$*, where* $i \in \widetilde{F_1}$ *and* $i_1, i_2, \cdots, i_l$ *be children of node* i*,* $\forall v \in \{\beta(j), \cdots \alpha(j)\}$*, where* $j \in \widetilde{F_2}$*,*

$$d(F_1^\circ(\beta(i)), F_2(v)) = min \begin{cases} d(F_1^\circ(\beta(i)), F_2^\circ(v)) + \delta(\varnothing, v) \\ min_{i_1 \le p \le i_k} \{d(F_1(\alpha(p)), F_2(v)) - d(F_1(\alpha(p)), \varnothing)\} + d(F_1^\circ(\beta(i)), \varnothing) \end{cases} \tag{4.4}$$

Proof The proof of Lemma 4.3.4 is symmetric to that of Lemma 4.3.3. The edit distance is the minimum value under two categories of constraints.

4.4 Implementation

The compressed tree is just a compact representation of the original. Thus, the algorithm is the same as in the Chapter 3:first find the optimal root-leaf path in compressed trees then compute distance in the bottom-up fashion. By Equation 4.1 and Lemma 4.3.1, 4.3.2, 4.3.3, the tree edit distance computation on compressed trees makes no differences to that on original trees when computing forest-to-forest distance but requires extra initialization steps when computing tree-to-tree distance.

The tree-to-tree edit distance gives rise to five data structures. Let T_1 and T_2 be two original trees, while $\widetilde{T_1}$ and $\widetilde{T_2}$ be two compressed trees after vertical reductions on tree T_1 and T_2 respectively.

- D_t: a two dimensional permanent array of size $(|T_1|+1)*(|T_2|+1)$, which is used to store distance with respect to the (T_1, T_2) representation.

- $\widetilde{D_t}$: a two dimensional permanent array of size $(\left|\widetilde{T_1}\right| + 1) * (\left|\widetilde{T_2}\right| + 1)$, which is used to store distances with respect to the $(\widetilde{T_1}, \widetilde{T_2})$ representation.

- $\widetilde{D_f}$: a two dimensional temporary array of size $(\left|\widetilde{T_1}\right| + 1) * (\left|\widetilde{T_2}\right| + 1)$, which is used to store intermediate results for forest-to-forest distances.

- A_1, A_2: temporary one dimensional arrays of lengths $(|T_1|+1)$ and $(|T_2|+1)$ respectively, which is used handle boundary initialization.

We now present the algorithms to compute tree-to-tree edit distance on compressed trees. The algorithm details are shown in Algorithm 6. Let $\widetilde{A}$ and $\widetilde{B}$ be two compressed trees, $i \in \widetilde{A}$ and $j \in \widetilde{B}$.

Algorithm 6: TREE_TO_TREE_DISTANCE

for $u \leftarrow \beta(i) - 1$ **to** $\alpha(i)$ **do**
  $A_1[u] \leftarrow D_t[u, \beta(j) - 1]$;
end
for $v \leftarrow \beta(j)$ **to** $\alpha(j)$ **do**
  $A_2[v] \leftarrow D_t[\beta(i) - 1, v]$;
end
$D_t[\beta(i) - 1, \beta(j) - 1] \leftarrow \widetilde{D_f}[i - 1, j - 1]$;
for $u \leftarrow \beta(i)$ **to** $\alpha(i)$ **do**
  $D_t[u, \beta(j) - 1] \leftarrow$
  $$min \begin{cases} D_t[u - 1, \beta(j) - 1] + \delta(A[u], \varnothing) \\ min_{j_1 \le q \le j_l} \{D_t[u, \alpha(q)] - \sum_{k \in B(\alpha(q))} \delta(\varnothing, k)\} + \sum_{k \in B(\beta(j))} \delta(\varnothing, k) \end{cases}$$
end
for $v \leftarrow \beta(j)$ **to** $\alpha(j)$ **do**
  $D_t[\beta(i) - 1, v] \leftarrow$
  $$min \begin{cases} D_t[\beta(i) - 1, v - 1] + \delta(\varnothing, B[v]) \\ min_{i_1 \le p \le i_k} \{D_t[\alpha(p), v] - \sum_{k \in A(\alpha(p))} \delta(k, \varnothing) + \sum_{k \in A(\beta(i))} \delta(k, \varnothing)\} \end{cases}$$
end
for $u \leftarrow \beta(i)$ **to** $\alpha(i)$ **do**
  for $v \leftarrow \beta(j)$ **to** $\alpha(j)$ **do**
    $$D_t[u, v] \leftarrow min \begin{cases} D_t[u - 1, v] + \delta(u, \varnothing) \\ D_t[u, v - 1] + \delta(\varnothing, v) \\ D_t[u - 1, v - 1] + \delta(u, v) \end{cases}$$
  end
end
$\widetilde{D_t}[i, j] \leftarrow D_t[\alpha(i), \alpha(j)]$;
$\widetilde{D_f}[i, j] \leftarrow D_t[\alpha(i), \alpha(j)]$;
for $u \leftarrow \beta(i) - 1$ **to** $\alpha(i)$ **do**
  $D_t[u, \beta(j) - 1] \leftarrow A_1[u]$;
end
for $v \leftarrow \beta(j)$ **to** $\alpha(j)$ **do**
  $D_t[\beta(i) - 1, v] \leftarrow A_2[v]$;
end

The algorithm of forest-to-forest edit distance on compressed trees is the exactly the same as that on original trees, using Equation 3.1 or 3.2 depending on the decomposition directions. The details of the forest-to-forest edit distance computation is shown in Algorithm 7 and 8.

Algorithm 7: FOREST_TO_FOREST_DISTANCE

1 $sizelA \leftarrow$ the size of A(lA);
2 $sizelB \leftarrow$ the size of B(lB);
3 $\widetilde{D_f}[lA, lB] \leftarrow min \begin{cases} \widetilde{D_f}[lA-1, lB] + \delta(lA, \varnothing) \\ \widetilde{D_f}[lA, lB-1] + \delta(\varnothing, lB) \\ \widetilde{D_f}[lA - sizelA, lB - sizelB] + \widetilde{D_t}[lA, lB] \end{cases}$

Algorithm 8: FOREST_TO_FOREST_DISTANCE

1 $sizerA \leftarrow$ the size of A(rA);
2 $sizerB \leftarrow$ the size of B(rB);
3 $\widetilde{D_f}[rA, rB] \leftarrow min \begin{cases} \widetilde{D_f}[rA-1, rB] + \delta(rA, \varnothing) \\ \widetilde{D_f}[rA, rB-1] + \delta(\varnothing, rB) \\ \widetilde{D_f}[rA - sizerA, lB - sizerB] + \widetilde{D_t}[rA, rB] \end{cases}$

Chapter 5

Experiment

We describe an application which would benefit from our algorithm, namely RNA secondary structure comparison.

5.1 RNA and its Secondary Structure

RNA is an essential molecule in organisms which has a wide range of functions in biological systems.Cellular organisms use messenger RNA (mRNA) to convey genetic information (using the letters G, U, A, and C to denote the nitrogenous bases guanine, uracil, adenine, and cytosine) that directs synthesis of specific proteins. Besides, many viruses encode their genetic information using an RNA genome.

RNA is assembled as a chain of nucleotides, but unlike DNA it is more often found in nature as a single-strand folded onto itself, rather than a paired double-strand. However, it can fold back onto itself by means of hydrogen bonding between distant complementary nucleotides(A=U, G≡C), resulting in the secondary structure. We define the secondary structure of RNA as follows.

Definition (RNA secondary structure) A secondary structure is primarily a list of base pair ω. A valid secondary structure should satisfy the following constraints:

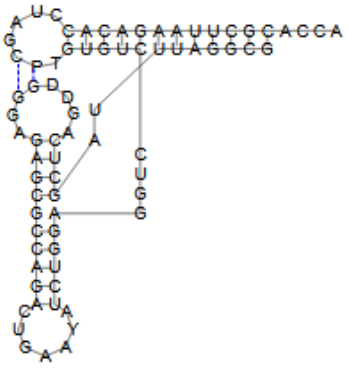


Figure 5.1: RNA Tertiary Structure.

- A base cannot participate in more than one base pair, i.e., ω is a matching on the set of sequence positions.
- No two base pairs (i, j) and $(k, l) \in \omega$ "cross" in the sense that $i < k < j < l$.

The first condition excludes tertiary structure motifs such as base triplets while the second condition avoid the pseudo knot in RNA structures. However, the fact is that pseudo knots do occur in RNA structures, namely RNA tertiary structure. To simplify, we don't consider the tertiary structure of RNA in this experiment and assume that any nucleotide participates in at most one such pair and the bonded pairs are non-crossing.

Figure 5.1 is an example of invalid RNA structure in our experiment, which has tertiary structures(marked in dotted blue lines) and base triplets.

5.2 String Representation of the RNA Secondary Structure

Secondary structure can also been stored compactly in strings consisting of dots and matching brackets:For any pair between positions i and j ($i < j$) we place an open bracket "(" at position i and a closed bracket ")" at j, while unpaired positions in the molecule are represented by a dot ".".Figure 5.2 illustrates the string presentation with dots and brackets of the RNA secondary structure.

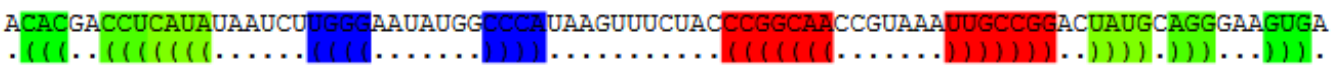


Figure 5.2: String representation of the RNA secondary structure.

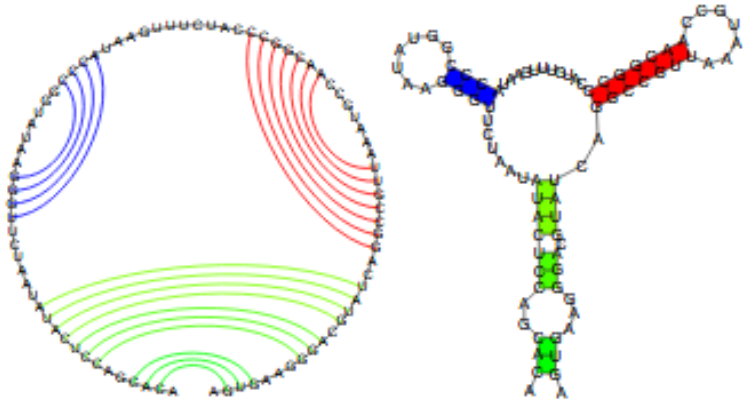


Figure 5.3: Graph representation of the RNA secondary structure.

5.3 RNA Secondary Structure Graphs

Secondary structures can be represented by ”secondary structure graphs”. According the base pair information in Figure 5.2, the string can be fold into a circle(Figure 5.3 on the left) or a secondary structure graph(Figure 5.3 on the right).

5.4 Tree Representation of the RNA Secondary Structure

In the computer science point of view, the secondary structure of an RNA molecule can be topologically represented by a tree. As shown in Figure 5.2, the RNA secondary structure graphs consists of two distinct category of nucleotides: those that are interacting via hydrogen bonding (so called stems), and those that are not(so call loops). Stems and loops in RNA can be represented as nodes in a tree. Leaves may be labeled with the corresponding unpaired base, while interior nodes are labeled with the corresponding base pair. To make things simplifier, we use different characters apart from alphabet$\{A, U, G, C\}$ to labeled each base pairs. Table

Base Pairs	Label
$A = U$	F
$G \equiv C$	J
$U = A$	P
$C \equiv G$	M

Table 5.1: The Label of Base Pair

5.1 illustrates the label of each base pairs. In this way,the RNA secondary structure can be converted into a tree. Figure 5.4 is the corresponding tree representation of the secondary structure of RNA in Figure 5.3.

5.5 Datasets

We grab RNA data from the Ribonuclease P Database [1]. The database consists of a compilation of RNase P sequences, sequence alignments, secondary structures, three-dimensional models and accessory information as well. The data can be downloaded from the website `http://www.mbio.ncsu.edu/RNaseP/home.html`.

The RNA files in the database are stored in XML format. The RNA files consist of RNA names, description, sequences as well as secondary structure base pairs. Figure 5.4 is an example of RNA files in the database. The RNA name is "A.tumefaciens RNase P RNA" with the length 402 nt. These two information are included between tags. The following information is the RNA sequence. The RNA second structure information is stored in base pairs indexed by the position in the RNA sequence. Each lines has two base indexes, meaning that two nucleotides are binding to each other.

We use the information from RNA files to construct trees. Then the problem of comparing the similarity between two RNA secondary structures becomes comparing the edit distance between two trees. Our algorithms in Chapter 3 and 4 can help solve the problem.

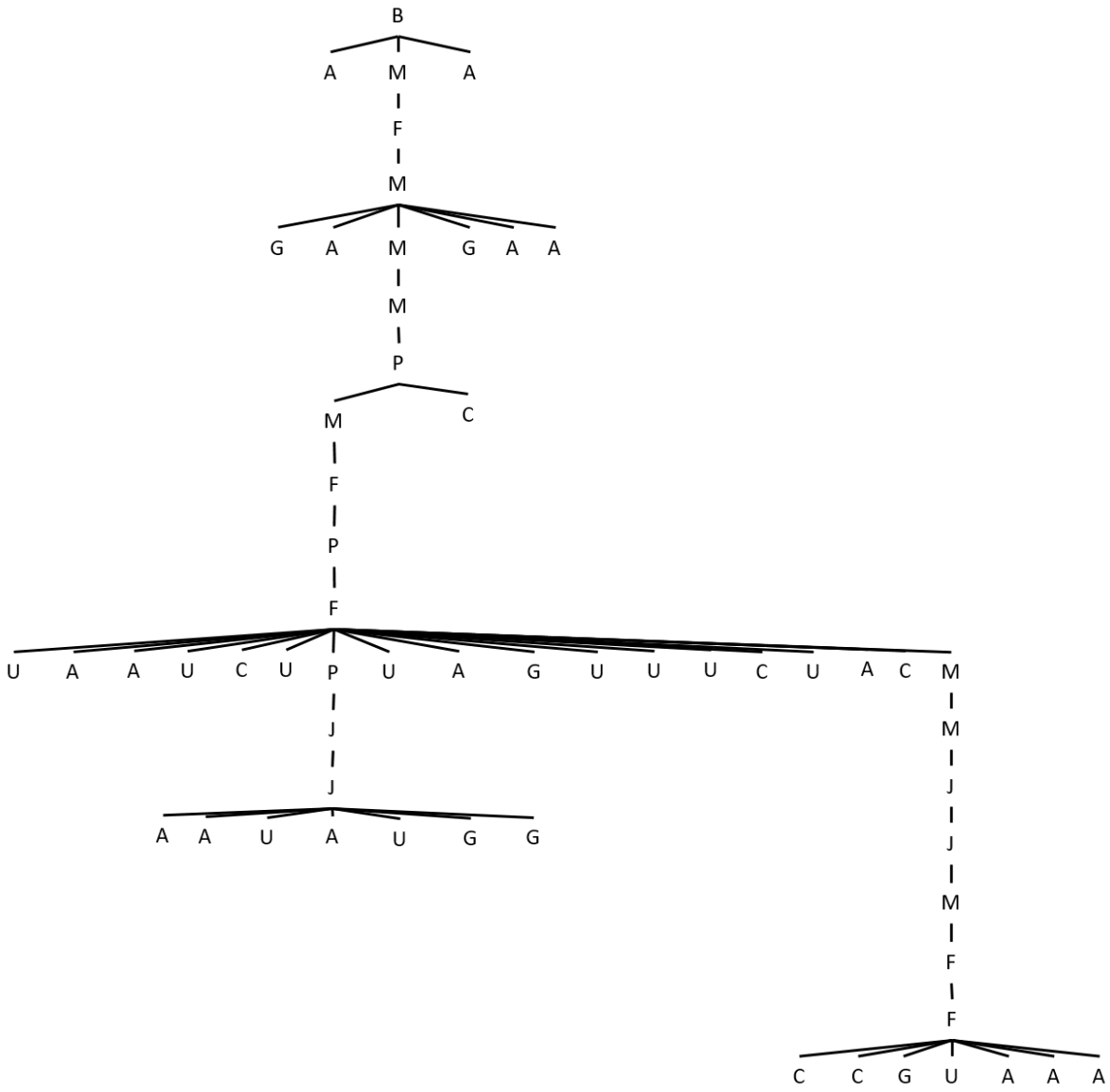


Figure 5.4: Tree representation of the RNA secondary structure.

```
<rnaml>

<rna name="A.tumefaciens RNase P RNA">

<translation-table scheme-length="402"/>

<sequence><data>

     CCAGUUGGCCGGGCAGCCGCGCCUUACCAAUGUCGAAAGACGGUAAGGUG
     AGGAAAGUCCGGGCUCCACGGAAAUACGGUGCCGGAUAACGUCCGGCGGG
     GGCGACCCCAGGGAAAGUGCCACAGAGAGCAAACCGCCAUGCCUUGCAUG
     GUAAGGGUGAAAGGGUGGGGUAAGAGCCCACCGCGCCGCUGGUGACAGUG
     GUGGCAAGGUAAACCCCACCGGGAGCAAGACCGAAUAGGGAUGACACGGG
     GCGGGCGAAAGCCUGUUCACAGCCGGUUUCCGGGCCCGUCAUCCGGGUGG
     GUUGCGAGAGGCGGCAUGCAAAUGCCGUCCCAGAUGAAUGGCUGCCACGU
     UCCGGGUCAAACCGGGGCCAUACAGAACCCGGCUUACAGGCCAACUGGCG
     AA

</data></sequence>

<structure>

     <base-pair _5p-base-num="1"  _3p-base-num="398"/>
     <base-pair _5p-base-num="2"  _3p-base-num="397"/>
     <base-pair _5p-base-num="3"  _3p-base-num="396"/>
     <base-pair _5p-base-num="4"  _3p-base-num="395"/>
     <base-pair _5p-base-num="5"  _3p-base-num="394"/>
     <base-pair _5p-base-num="6"  _3p-base-num="393"/>
     <base-pair _5p-base-num="7"  _3p-base-num="392"/>
     <base-pair _5p-base-num="8"  _3p-base-num="391"/>
     <base-pair _5p-base-num="9"  _3p-base-num="390"/>
     <base-pair _5p-base-num="10"  _3p-base-num="389"/>
     <base-pair _5p-base-num="11"  _3p-base-num="387"/>
     <base-pair _5p-base-num="12"  _3p-base-num="346"/>
     <base-pair _5p-base-num="13"  _3p-base-num="345"/>
     <base-pair _5p-base-num="14"  _3p-base-num="344"/>
     <base-pair _5p-base-num="15"  _3p-base-num="343"/>
     <base-pair _5p-base-num="16"  _3p-base-num="342"/>
     <base-pair _5p-base-num="17"  _3p-base-num="341"/>
     <base-pair _5p-base-num="18"  _3p-base-num="340"/>
     <base-pair _5p-base-num="20"  _3p-base-num="50"/>
     <base-pair _5p-base-num="21"  _3p-base-num="49"/>
     <base-pair  5p-base-num="22"   3p-base-num="48"/>
```

Figure 5.5: An RNA Sequence in XML Format.

```
RNAName: Alcaligenes-eutrophus-pb-b
RNASize: 343
(((((((((((((((((-((-(((((((----)))))))-))-------------((((--------(((((-((((-----)
AAAGCAGGCCAGGCAACCGCUGCCUGCACCGCAAGGUGCAGGGGGAGGAAAGUCCGGACUCCACAGGGCAGGGUGUUGGCUAACAG
)))-(((((----))))-)((---((---------------((((-(((((((----)))))))-))))----------------
CCAUCCACGGCAACGUGCGGAAUAGGGCCACAGAGACGAGUCUUGCCGCCGGGUUCGCCCGGCGGGAAGGGUGAAA----------
)))))))-----))))------(((((((----)))))))----------)))))))--((-(((((((((----)))))))))
GAGUCUGCGGGUAGGGAGCUGGAGCCGGCUGGUAACAGCCGGCCUAGAGGAAUGGUUGUCACGCACCGUUUGCCGCAAGGCGGGCG
)-))-----------------)-)))))))))----
GGGCGCACAGAAUCCGGCUUAUCGGCCUGCUUUGCUU
```

Figure 5.6: Alcaligenes eutrophus Sequence from the RNase P database.

```
RNAName: Streptomyces-bikiniensis-gpb-h
RNASize: 400
((((((((((((((((((-((-(((((((----)))))))-))-------------((((--------((((((((((-----))))))((((
CGAGCCGGGCGGGCGGCCGCGUGGGGGGUCUUCGGACCUCCCCGAGGAACGUCCGGGCUCCACAGAGCAGGGUGGUGGCUAACGGCCACCCGG
(----))))-)((---((---------------((((((((((----))))))))--))-------((((((-------))))))--(
GGUGACCCGCGGGACAGUGCCACAGAAAACAGACCGCCGGGGACCUCGGUCCUCGGUAAGGGUGAAACGGUGGUGUAAGAGACCACCAGC
)))))))))-----))))------(((((((----)))))))----------)))))))-(((((--((((----))))-)))))
CGAGUCCGCGGGUAGACCGCACGAGGCCGGCGGCAACGCCGGCCCUAGAUGGAUGGCCGUCGCCCCGACGACCGCGAGGUCCCGGGG
---------------)-)))))))))----
ACAGAACCCGGCGUACAGCCCGACUCGUCUG
```

Figure 5.7: Streptomyces bikiniensis Sequence from the RNase P database.

5.6 Some Experimental Result

In our experiments, we compute alignments between RNA secondary structure from Ribonuclease P Database. Figures 5.6 and 5.7 show two RNA structures in string representation. On the other hand, the corresponding graph representations are shown in Figures 5.8 and 5.9. We notice that the raw data contains tertiary structures. Therefore, we first remove the tertiary structures before constructing the corresponding tree data structure. Figure 5.10 shows the alignment results where the cost of each valid operation(deletion, insertion and substitution) is 1.

5.7 Evaluation

We empirically evaluate our algorithm with other four state-of-the-art algorithms on Ribonuclease P Database and compare the relevant sub-problems and actual run time to the algorithms proposed in Chapter 2: the algorithm by Zhang and Shasha and its symmetric version always

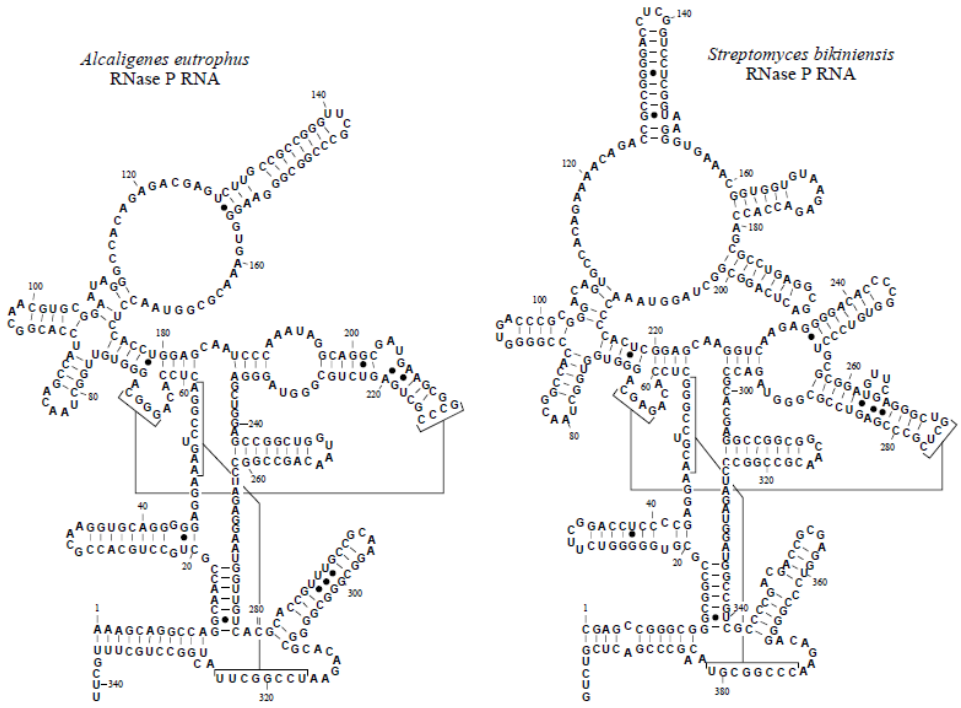


Figure 5.8: The structures of RNase P RNA of Cupriavidus metallidurans (Alcaligenes eutrophus) and Streptomyces bikiniensis.

```
Alcaligenes eutrophus ~ Streptomyces bikiniens
deletion: 1 insertion: 1 substitution: 1 distance: 134
((((-(((((((((((((-((-(((((((-----)))))))-))-------------((((--------((-(((--((((-----))))--((((
((((((-(((((((((((-((-(((((-(((-----)))-))))-))-------------((((---------(((((-(((((-----))))-)((((
AAAG-CAGGCCAGGCAACCGCUGCCUGCACCG-CAAGGUGCAGGGGGAGGAAAGUCCGGACUCCACAGGGCAGG-GUGU-UGGCUAACAGCCAU-CCAC
CGAGCC-GGGCGGGCGGCCGCGUGGGG-GUCUUCG-GAC-CUCCCCGAGGAACGUCCGGGCUCCACAGAGCA-GGGUG-GUGGCUAACGGCCA-CCCGG

(----))))-)((---((---------------((((-((((((-((--------))-))))--))-))))----------------------------
(----))))-)((---((--------------------((((((((((----))))))))))--))-----------(((((-------))))))--(
GGCAACGUGCGGAAUAGGGCCACAGAGA-CGAGUCUUGCCGCCG-GG--UUCG--CC-CGGC--GGGAAGG-GU---------G-AA-A----------
GGUGACCCGCGGGACAGUGCCACAGAAAAC-AG----ACCGCCGGGGACCUCGGUCCUCGGUAAGGG----UGAAACGGUGGUGUAAGAGACCACCAGC

----------------------------)))))))-))))))---((((-------------------------((((((----((-(((-----)))
(((((((----)))))))---------)))))))-))))---((((----(((((((----)))))))--((((((----((((((-----)))
--------CG----------C--GGUAACCUCCAC-CUGGAGCAAUCCCAA-A--------UA-G----------GCAGGCGAU-GA-AGCGGCCCGCU
GCCUGAGGCGACUCAGGCGGCUAGGUAAACCCCACUC-GGAGCAAGGUCAAGAGGGGACACCCCGGUGUCCCUGCGCGGAUGUUCGAGGGCUGCUCGCC

-)))))))-----))))-------(((((((----)))))))------------))))))--((-((((((--(-((----))-))))-))))-))
)))))))))-----))))------(((((((-(----)-)))))))------------))))))--((-(((-----(((((----))))----)))-))
-GAGUCUGCGGGUAGGGAGCUGGA-GCCGGCUGGUAACAGCCGGC-CUAGA-GGAAUGGUUGUCACGCACCGUUU--G-CCGCAAGG-CGGG-CGGGGC
CGAGUCCGCGGGUAGACCGCACGAGGCCGGC-GGCAAC-GCCGGCCCUAGAUGGA-UGGCCGUCG-CC-CCG---ACGACCGCGAGGUC---CCGG-GG

-----------------)-))))))-))))----
-----------------)-))))-))))))----
GCACAGAAUCCGGCUUAUCGGCCUG-CUUUGCUU
--ACAGAACCCGGCGUACAGCCC-GACUCGUCUG
```

Figure 5.9: Alignment Between the Secondary Structure of Alcaligenes eutrophus and Streptomyces bikiniensis.

Algorithm	*#Rel.sub*	**Time[sec]**
$Zhang-L$	849282	0.08
$Zhang-R$	2039089	0.13
$Our\ Algorithm(Before\ Compression)$	553526	0.04
$Zhang-L(Compressed)$	428766	0.05
$Zhang-R(Compressed)$	686562	0.05
$Our\ Algorithm(After\ Compression)$	291329	0.03

Table 5.2: The Relevant Sub-problem and its Actual Run Time of Each Algorithms

using right paths [12]. All algorithms are implemented as single-thread applications in C++ and run on a 64-core 3.4GHZ Ubuntu. The source code is available online.

Firstly we compare the number of relevant sub-problem computed by each of the algorithms for a pair of specific RNA secondary structures. Each relevant sub-problems are the constant-time operations that make up the complexity of the algorithm. Then, we mark the actual run time of each algorithms. The evaluation results are shown in Table 5.2.

Chapter 6

Conclusion

In this thesis, we have studied the topic of tree edit distance computation. After the definition of tree edit distance, we have given an overview of existing approaches for tree edit distance and their decomposition strategies, including leftmost paths decomposition, rightmost paths decomposition, heavy path decomposition on one tree and that on both trees as well. These methods take advantage of the overlap among sub-forests that are contained in the same sub-tree, and the overlap of sub-trees in different ways.

We proposed a new algorithm to find the optimal root-leaf path decomposition that avoid redundant computation. The algorithm uses dynamic programming and is implemented using C++. An overview description and some detailed implementations have been illustrated in Chapter 2.

Another algorithmic improvement can be applied to our algorithm to reduce time complexity. We compressed the non-branching nodes to a single node to compress tree in the vertical direction. After the vertical reduction, the compressed trees then are used for computation. The detailed implementations are shown in Chapter 3.

RNA secondary structure similarity comparison is an application of our algorithm. We test our algorithm on the Ribonuclease P Database and evaluation the time complexity by counting the relevant sub-problem and mark the actual run time. According the test result, our algorithm

is by far the best decomposition strategy for trees.

A lot of research remains to be done. Our results are good but we hope to improve them.

Bibliography

[1] James W Brown. The ribonuclease p database. *Nucleic acids research*, 26(1):351–352, 1998.

[2] Shihyen Chen. A review on the tree edit distance problem and related path-decomposition algorithms. *arXiv preprint arXiv:1501.00611*, 2015.

[3] Shihyen Chen and Kaizhong Zhang. An improved algorithm for tree edit distance with applications for rna secondary structure comparison. *Journal of Combinatorial Optimization*, 27(4):778–797, May 2014.

[4] Erik D Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms (TALG)*, 6(1):2, 2009.

[5] Serge Dulucq and Hélene Touzet. Decomposition algorithms for the tree edit distance problem. *Journal of Discrete Algorithms*, 3(2):448–471, 2005.

[6] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

[7] Philip N Klein. Computing the edit-distance between unrooted ordered trees. In *ESA*, volume 98, pages 91–102. Springer, 1998.

[8] Zhewei Liang and Kaizhong Zhang. *Algorithms for Forest Local Similarity*, pages 163–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, 40(1):3, 2015.

[10] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979.

[11] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[12] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.