ABSTRACT

PATTERN DISCOVERY IN STRUCTURAL DATABASES WITH APPLICATIONS TO BIOINFORMATICS

by
Sen Zhang

Frequent structure mining (FSM) aims to discover and extract patterns frequently occurring in structural data such as trees and graphs. FSM finds many applications in bioinformatics, XML processing, Web log analysis, and so on. In this thesis, two new FSM techniques are proposed for finding patterns in unordered labeled trees. Such trees can be used to model evolutionary histories of different species, among others.

The first FSM technique finds cousin pairs in the trees. A cousin pair is a pair of nodes sharing the same parent, the same grandparent, or the same great-grandparent, etc. Given a tree T, our algorithm finds all interesting cousin pairs of T in $O(|T|^2)$ time where $|T|$ is the number of nodes in T. Experimental results on synthetic data and phylogenies show the scalability and effectiveness of the proposed technique. This technique has been applied to locating co-occurring patterns in multiple evolutionary trees, evaluating the consensus of equally parsimonious trees, and finding kernel trees of groups of phylogenies. The technique is also extended to undirected acyclic graphs (or free trees).

The second FSM technique extends traditional MAST (maximum agreement subtree) algorithms by employing the Apriori data mining technique to find frequent agreement subtrees in multiple phylogenies. The correctness and completeness of the new mining algorithm are presented. The method is also extended to unrooted phylogenetic trees.

Both FSM techniques studied in the thesis have been implemented into a toolkit, which is fully operational and accessible on the World Wide Web.

PATTERN DISCOVERY IN STRUCTURAL DATABASES WITH APPLICATIONS TO BIOINFORMATICS

by
Sen Zhang

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

Department of Computer Science

January 2005

APPROVAL PAGE

PATTERN DISCOVERY IN STRUCTURAL DATABASES WITH APPLICATIONS TO BIOINFORMATICS

Sen Zhang

Dr. Jason T.L. Wang, Dissertation Advisor — Date
Professor of Computer Science Department, New Jersey Institute of Technology

Dr. James A. McHugh, Committee Member — Date
Professor of Computer Science Department, New Jersey Institute of Technology

Dr. Frank Shih, Committee Member — Date
Professor of Computer Science Department, New Jersey Institute of Technology

Dr. Qun Ma, Committee Member — Date
Assistant Professor of Computer Science Department, New Jersey Institute of Technology

Dr. Katherine Grace Herbert, Committee Member — Date
Assistant Professor of Computer Science Department, Montclair State University

BIOGRAPHICAL SKETCH

Author:	Sen Zhang
Degree:	Doctor of Philosophy
Date:	January 2005

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2005
- Master of Science in Computer Science,
  South China University of Technology and Science, Guangzhou, China, 1995
- Bachelor of Science in Computer Science,
  Tianjin University, Tianjin, China, 1992

Major: Computer Science

Presentations and Publications:

S. Zhang and J. T.L. Wang, "Knowledge Discovery from Evolutionary Trees," *Advanced Methods for Knowledge Discovery from Complex Data*, Eds. S. Bandyopadhyay, U. Maulik, L. Holder and D. Cook, Springer Verlag, London, to appear.

D. Shasha, J. T.L. Wang and S. Zhang, "Unordered Tree Mining with Applications to Phylogeny," *IEEE International Conference of Data Engineering*, pp. 766-778, April 2004.

S. Zhang, L. Liao, J.F. Tomb and J. T.L. Wang, "Clustering Enzymes in Metabolic Pathways: Some Preliminary Results," In the *Proceedings of the the 2002 ACM BIOKDD Workshop on Data Mining in Bioinformatics*, Edmonton, Alberta, Canada, July 2002.

S. Zhang, J. T.L. Wang and K. G. Herbert, "XML Query by Example," in the *International Journal of Computational Intelligence and Applications*, World Scientific Publishing, Vol. 2, No. 3, pp. 329-338, September 2002.

To my beloved parents, parents-in-law, Min and Daniel!

ACKNOWLEDGMENT

The author wishes to express his genuine gratitude to his thesis advisor, Dr. Jason T.L. Wang, for his advice, guidance, encouragement, patience and constant support throughout this research. Without him, this work would have not been possible.

The author likes to acknowledge and thank the members of his thesis proposal and dissertation committees Dr. James A. MucHgh, Dr. Frank Shih, Dr. Qun Ma, Dr. Katherine G. Herbert, Dr. Wynne Hsu, Dr. Bin Tian and Dr. Michael M. Yin. The author is indebted to them for their precious time, helpful comments, insightful critics and constructive advices.

The author also wants to thank Dr. Alexandros Gerbessiotis for his precious help during the past three years. Being a teaching assistant for him has been a pleasant experience and an effective way to solidify the author's knowledge in the assisting field.

Studying in the New Jersey Institute of Technology for the past five years as an international graduate student, the author had received tremendous help from Dr. Ronald Kane, Ms. Clarisa González-Lenahan and Mr. Jeff Grundy and their offices. Their caring and timely help had always made the life in the New Jersey Institute of Technology enjoyable and unforgettable. The author wants to express his sincere thanks to all of them, especially to Ms. Clarisa for her helpful advices.

The author would like to extend his heartful thanks to his colleagues in Data and Knowledge Engineering Lab Dr. Katherine G. Herbert and Dr. Huiyuan Shan, for their constant help and substantial collaboration. The author also greatly appreciates Dr. Katherine G. Herbert and Dr. Michael Kerley for their proofreading the manuscript of this work and their constructive comments. The former lab members Dr. Michael M. Yin, Dr. Qichen Ma, Dr. Xiong Wang and Dr. Xiaoming Wu all deserve special thanks for their useful discussions at different stages of the author's Ph.D. study. The author would like to offer his special thanks to all wonderful friends at NJIT, New Jersey. They are Binghu Zhang, Jingxuan Liu, Wenxin Mao, Hairong Zhao, Chunqi Han, Li Zhang, Zhifen Liu, Yue Li, Pengxiang Wang and Xiangying Yin, and many others.

And last, but not least, the author is deeply indebted to his parents, his parents-in-law, his brother, his wife and his son for their love, faith, encouragement and many sacrifices during the past five years.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

1.1 Motivation

Frequent structure mining aims to discover and extract patterns frequently occurring in structural data such as trees and graphs. It finds applications in many scientific database domains such as XML data, semi-structured data, Web log analysis, linguistic and bioinformatics. Significant developments of various frequent pattern discovery algorithms have been witnessed during the past two decades. Each of them differs from the others in how to answer the following questions: what kind of domain knowledges can be carried by a certain pattern? how to efficiently mine those patterns? are the interesting patterns embedded or induced substructures? and whether the interesting patterns will be discovered exactly or approximately?

This work studies how to find frequent patterns from unordered trees and investigates the usefulness of discovered patterns in phylogenetic tree applications - an important bioinformatics research field.

A rooted unordered labeled tree is a tree in which there is a root for the tree, each node may have a label, and the left-to-right order among siblings is unimportant.[1] A rooted unordered tree can be used by biologists to model the evolutionary history of a set of Taxa (organisms or species) that have a common ancestor. Such trees are also known as phylogenetic trees (phylogenies) or evolutionary trees. To be more specific, a phylogenetic tree is a leaf-labeled tree structure depicting the evolutionary history of a set of organisms. The internal nodes within a particular tree represent older organisms from which their child nodes descend, and the children represent divergences in the evolution of the genetic composition in the parent organism. Since

[1]Rooted unordered labeled trees shall be simply referred as trees when the context is clear.

a phylogenetic tree conveys to biologists evolutionary history of different Taxa mainly through its hierarchical divergences, which makes the order of siblings immaterial in such a tree, a phylogeny is deemed an unordered tree.

For a variety of reasons, biologists usually deal with many different trees concerning with the same set of Taxa. First, for the same input data, different theories about the evolutionary history of the same set of species often result in different evolutionary trees. Second, some tree reconstruction algorithms such as Most Parsimonious method can produce many equally parsimonious trees for the same input data. Third, different biological sequence data of the same set of Taxa usually result in different trees, even being processed by the same tree reconstruction method. As a consequence, biologists need to conduct further analysis on these trees in order to extract more useful information.

Traditionally, biologists analyze these trees through pairwise comparisons. For this purpose, various tree measures such as partition metric, quartet distance, triplet distance and nearest neighbor interchange distance have long been researched. However, numerical values of such distance measures are usually too abstract to be informative, especially when the number of data trees under investigation is large. Furthermore, with more and more phylogenetic trees being produced by supertree algorithms, the above tree distance measures are not applicable to such trees any more. This is because that all these classical measures are established for trees built upon the exactly same set of Taxa, while trees generated by supertree algorithms are allowed to share partially overlapped leaf label sets.

Thus, there is a clear need for advanced pattern discovery algorithms that can assist biologists to find more useful information from multiple phylogenetic trees generated by diverse resources.

The main goal of this dissertation is to develop novel frequent structure mining algorithms for phylogenetic tree analysis. To be specific, cousin pair mining algorithm and frequent agreement subtree mining algorithm will be discussed. Cousin pair mining algorithm aims to discover patterns formed by two nodes measured by varying cousin distances. This work first formalizes cousin pair mining problem under a generic unordered tree model, then the solution to the problem is shown to be useful in a variety of phylogenetic tree applications.

In contrast to cousin mining problem, frequent agreement subtree mining algorithm will find interesting patterns with varying tree sizes. It is a data mining alternative to the traditional maximum agreement subtree problem. Since phylogenetic trees are unordered, a canonical form for phylogenetic trees is proposed to solve unordered tree isomorphism problem. Furthermore, an agreement subtree could be embedded in a data tree, and this kind of embedding subtree mining problem has not been fully researched in literature. This work proposes a novel candidate subtree generation method, which has also considered various optimization techniques.

Depending on if biologists have sufficient evidences to suggest a distinct root [2] for the tree, a phylogenetic tree can be reconstructed as either a rooted tree or an unrooted tree. To meet this challenge, both algorithms have also been extended to unrooted tree scope.

Finally, both cousin mining algorithm and frequent agreement tree mining algorithm have been successfully implemented and further integrated into an online mining engine toolkit.

1.2 Thesis Organization

The subsequent chapters are outlined as following. Chapter 2 presents cousin pair mining algorithm. In this chapter, the cousin pair mining problem in rooted tree scope will be presented first, then the algorithm is extended to unrooted tree. Chapter 3

[2] A distinct root indicates a clear common ancestor of all Taxa under investigation.

focuses on the frequent agreement subtree mining problem and a novel solution to attack the problem. In this chapter, the canonical form of rooted phylogenetic trees will be introduced first, then a unique pattern expansion scheme will be presented to solve the problem. The solution also successfully extends to unrooted trees. Chapter 4 describes two online mining engines built upon the proposed mining algorithms. Chapter 5 concludes the thesis and points out future work.

CHAPTER 2

COUSIN PATTERN MINING

2.1 Introduction

This chapter presents a new FSM technique - cousin pair algorithm for finding patterns in rooted unordered labeled trees. Section 2.1 discusses the motivation and background of the work. Section 2.2 introduces notation and terminology. Section 2.3 presents algorithms for finding frequent cousin pairs in trees. Section 2.4 reports experimental results on both synthetic data and phylogenies, showing the scalability and effectiveness of the proposed approach. Section 2.5 discusses applications of this new approach to locating co-occurring patterns in multiple phylogenies, evaluating the consensus of equally parsimonious trees, and finding kernel trees from groups of phylogenies. Section 2.6 describes extensions of the algorithms for undirected acyclic graphs. Section 2.7 summarizes the chapter.

The patterns this chapter wants to find from these trees contain "cousin pairs." For example, consider the three trees in Figure 2.1. In the figure, a and y are cousins with distance 0 in T_1; e and f are cousins with distance 0.5 in T_2; b and f are cousins with distance 1 in all the three trees.

The measure "distance" represents kinship of two nodes. Cousins with distance 0 are siblings, sharing the same parent node. Cousins of distance 1 share the same grandparent. Cousins of distance 0.5 represent aunt-niece relationships. the algorithms described in this chapter can find cousins of varying distances in a single tree or multiple trees.

Finding the cousin pairs helps to better understand the evolutionary history of the species, because cousin pairs in these trees represent evolutionary relationships between species that share a common ancestor.

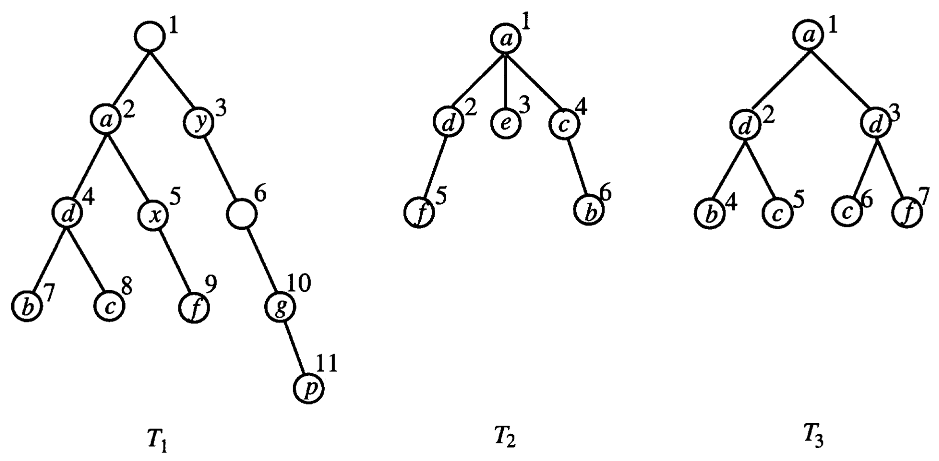


Figure 2.1 Three trees T_1, T_2 and T_3. Each node in a tree may or may not have a label, and is associated with a unique identification number (represented by the integer outside the node).

The cousins (patterns) discovered from trees can be used in several ways. As shown later, they can be used to evaluate the quality of a consensus tree [45] obtained from multiple phylogenies or can be used to compute the distance between two phylogenies. The last part of this chapter also discusses extensions of the techniques for undirected acyclic graphs (or free trees).

In the past, much work on frequent structure mining was conducted with applications to XML data, documentation and semi-structured data processing [5, 40, 41, 59, 60, 61]. The major difference between these works lies in the different patterns they discover, which range from XML DTD, tags, schemas, to structural associations in documents. More recently Zaki [69] developed algorithms capable of finding frequent embedded tree patterns in a forest where he models a document by an ordered labeled tree. Chen *et al.* [9] studied techniques for selectivity estimation in the context of XML querying. Other related work on general tree matching, inclusion and isomorphism detection can be found in [11, 30, 50].

There has also been work in graph mining. For example, the authors of [28, 32] extended the Apriori technique [2], originally designed for association rule mining, to find frequent subgraphs in graph data. Yan and Han [65] found closed frequent subgraphs in graph data. Dehaspe *et al.* [14] applied inductive logic programming to find frequent substructures (subgraphs) describing the carcinogenesis of chemical compounds. Cook and Holder [12] found repeated patterns in graphs using the minimum description length principle. Yoshida and Motoda [67] used beam search for mining subgraphs. Wang *et al.* [62] applied geometric hashing techniques to find frequent substructures in 3-D graphs and used the substructures to classify proteins and chemical compounds.

Th cousin-based distance measure presented in this chapter joins the many others already developed [24, 39, 46, 51]. This work differs from the above approaches in three ways. First, in contrast to the other tree mining methods (e.g. [56, 69]), which focused on trees where the order of siblings matters, the cousin mining algorithms concern with unordered trees, which are appropriate for the phylogenetic application. Second, this algorithm finds cousin pairs with varying distances from the trees. These frequent cousin pairs differ from the patterns found in all the previous work, entailing new methods in the discovery process. When applied to phylogeny, the proposed cousin-based distance measure can be used to compare evolutionary trees for which existing methods are not suitable. Third, in contrast to the other graph mining methods (e.g. [12, 67]) which are based on heuristic search and hence may miss some interesting patterns, this algorithm performs a complete search without missing any patterns satisfying the user-specified requirement.

2.2 Preliminaries

Let Σ be a finite set of labels. A rooted unordered labeled tree of size $k > 0$ on Σ is a quadruple $T = (V, N, L, E)$, where

- V is the set of nodes of T in which a node $r(T) \in V$ is designated as the root of T and $|V| = k$.

- $N : V \mapsto \{1 \ldots, k\}$ is a numbering function that assigns a unique identification number $N(v)$ to each node $v \in V$.
- $L : V' \mapsto \Sigma$, $V' \subseteq V$, is a labeling function that assigns a label $L(v)$ to each node $v \in V'$; the nodes in $V - V'$ do not have labels. Obviously, this labeling function allows multiple nodes to have the same label.
- $E \subset N(V) \times N(V)$ contains all parent-child pairs in T.

For example, refer to the trees in Figure 1. The node numbered 6 in T_1 does not have a label. The nodes numbered 2, 3 in T_3 have the same label d and the nodes numbered 5, 6 in T_3 have the same label c. The following paragraphs introduce a series of definitions that will be used in the new algorithms.

Cousin distance. Given two labeled nodes u, v of tree T where neither node is the parent of the other, the least common ancestor w of u and v is denoted as $lca(u, v)$, and the heights of u, v in the subtree rooted at w are represented as $H(u, w)$, $H(v, w)$ respectively. The definition of *cousin distance* of u and v, denoted as $c\_dist(u, v)$, is shown in Figure 2.2. The cousin distance $c\_dist(u, v)$ is undefined

$$c\_dist(u,v) = \begin{cases} H(u,w) - 1 & \text{if } H(u,w) = H(v,w) \\ \max\{H(u,w), H(v,w)\} - 1.5 & \text{if } |H(u,w) - H(v,w)| = 1 \end{cases}$$

Figure 2.2 Definition of the cousin distance between two nodes u and v.

if $|H(u, w) - H(v, w)|$ is greater than 1, or one of the nodes u, v is unlabeled. (The cutoff of 1 is a heuristic choice that works well for phylogeny. In general there could be no cutoff or the cutoff could be much greater.)

The cousin distance definition is inspired by genealogy. Node u is a first cousin of v, or $c\_dist(u, v) = 1$, if u and v share the same grandparent. In other words, v is a child of u's aunts or vice versa. Node u is a second cousin of v, or $c\_dist(u, v) = 2$, if u and v have the same great-grandparent, but not the same grandparent. For two nodes u, v that are siblings, i.e. they share the same parent, $c\_dist(u, v) = 0$.

The number "0.5" is used to represent the "once removed" relationship. When the word "removed" is used to describe a relationship between two nodes, it indicates that the two nodes are from different generations. The words "once removed" mean that there is a one-generation difference. For any two labeled nodes u and v, if u is v's parent's first cousin, then u is v's first cousin, once removed and $c\_dist(u, v) = 1.5$. "Twice removed" means that there is a two-generation difference. The cousin distance definition requires $|H(u, w) - H(v, w)| \leq 1$ and excludes the twice removed relationship. As mentioned above, this is a heuristic rather than a fundamental restriction.

For example, consider again T_1 in Figure 1. There is a one-generation difference between the aunt-niece pair y, x and $c\_dist(y, x) = 0.5$. Node b is node f's first cousin and $c\_dist(b, f) = 1$. Node d is node g's first cousin, once removed, and $c\_dist(d, g) = 1.5$. Node f is node g's second cousin, and $c\_dist(f, g) = 2$. Node f is node p's second cousin, once removed, and $c\_dist(f, p) = 2.5$.

Notice that parent-child relationships are not included in this work because the internal nodes of phylogenetic trees usually have no labels. So, parent-child pairs are not considered at all. This heuristic works well in phylogenetic applications, but could be generalized. One such generalization proposed by Wang *et al.*uses the UpDown distance [55]. Another approach, suggested by a reviewer, is to use one upper limit parameter for inter-generational (vertical) distance and another upper limit parameter for horizontal distance.

Table 2.1 Cousin Pair Items of T_3 in Figure 2.1

Distance	Cousin Pair Items
0	$(b, c, 0, 1), (c, f, 0, 1), (d, d, 0, 1)$
0.5	$(d, b, 0.5, 1), (d, c, 0.5, 2), (d, f, 0.5, 1)$
1	$(b, f, 1, 1), (b, c, 1, 1), (c, c, 1, 1), (c, f, 1, 1)$

Cousin pair item. Let u, v be cousins in tree T. A *cousin pair item* of T is a quadruple $(L(u), L(v), c\_dist(u, v), occur(u, v))$, where $L(u)$ and $L(v)$ are labels of u, v, respectively, $c\_dist(u, v)$ is the cousin distance of u, v and $occur(u, v) > 0$ is the number of occurrences of the cousin pair in T with the specified cousin distance. Table 2.1 lists all the cousin pair items of tree T_3 in Figure 1. Consider, for example, the cousin pair item (d, c, 0.5, 2) in the second row of Table 1. Node 2 and node 6, node 3 and node 5 respectively, is an aunt-niece pair with cousin distance 0.5. When taking into account labels of these nodes, it is clear that the cousin pair (d, c) with distance 0.5 occurs 2 times totally in tree T_3, and hence (d, c, 0.5, 2) is a valid cousin pair item in T_3.

A cousin pair item may also consider the total number of occurrences of the cousins u and v regardless of their distance, for which case λ replaces $c\_dist(u, v)$ in the cousin pair item. For example, in Table 1, T_3 has $(b, c, 0, 1)$ and $(b, c, 1, 1)$, and hence the cousin pair item $(b, c, \lambda, 2)$ can be obtained. Here, the cousin pair (b, c) occurs once with distance 0 and occurs once with distance 1. Therefore, when ignoring the distance, the total number of occurrences of (b, c) is 2. Likewise it can ignore the number of occurrences of a cousin pair (u, v) by using λ in place of $occur(u, v)$ in the cousin pair item. For example, in Table 1, T_3 has $(b, c, 0, \lambda)$ and $(b, c, 1, \lambda)$. Furthermore, both the cousin distance and the number of occurrences could be ignored and the cousin labels are the only concerns. For example, T_3 has (b, c, λ, λ), which simply indicates that b, c are cousins in T_3.

Frequent cousin pair. Let $\mathcal{S} = \{T_1, T_2, \ldots, T_n\}$ be a set of n trees and let d be a given distance value. Let $\delta_{u,v,i}$ be 1 if T_i has the cousin pair item $(L(u), L(v), d,$ $occur(u, v))$, $occur(u, v) > 0$; otherwise $\delta_{u,v,i}$ is 0. The *support* of the cousin pair (u, v) with respect to the distance value d is defined as $\Sigma_{1 \leq i \leq n} \delta_{u,v,i}$. Thus the support value represents the number of trees in the set $\mathcal{S}$ that contain *at least* one occurrence of the cousin pair (u, v) having the specified distance value d. A cousin pair is *frequent* if its support value is greater than or equal to a user-specified threshold, *minsup*.

For example, consider Figure 1 again. T_1 has the cousin pair item $(c, f, 1, 1)$, T_2 has the cousin pair item $(c, f, 0.5, 1)$ and T_3 has the cousin pair item $(c, f, 1, 1)$ and $(c, f, 0, 1)$. The support of (c, f) w.r.t. distance 1 is 2 because both T_1 and T_3 have this cousin pair with the specified distance. Cousin distances can also be ignored when finding frequent cousin pairs. For example, the support of (c, f) is 3 when the cousin distances are ignored.

Given a set $\mathcal{S}$ of trees, this approach offers the user several alternative kinds of frequent cousin pairs in these trees. For example, the algorithm can find, in a tree T of $\mathcal{S}$, all cousin pairs in T whose distances are less than or equal to *maxdist* and whose occurrence numbers are greater than or equal to *minoccur*, where *maxdist* and *minoccur* are user-specified parameters. The algorithm can also find all frequent cousin pairs in $\mathcal{S}$ whose distance values are at most *maxdist* and whose support values are at least *minsup* for a user-specified *minsup* value. The following section will describe the techniques used in finding these frequent cousin pairs in a single tree or multiple trees.

2.3 Tree Mining Algorithms

Given a tree T and a node u of T, let *children_set*(u) contain all children of u. the algorithm preprocesses T to obtain *children_set*(u) for every node u in T. The algorithm also preprocesses T to be able to locate a list of all ancestors of any node u in $O(1)$ time using a conventional hash table.

Now, given a user-specified value *maxdist*, all valid distance values 0, 0.5, 1, 1.5, ..., *maxdist* are considered. For each valid distance value d, *my_level(d)* and *mycousin_level(d)* are defined as follows:

$$my\_level(d) = 1 + \lfloor d \rfloor \tag{2.1}$$

$$mycousin\_level(d) = my\_level(d) + R \tag{2.2}$$

where

$$R = 2 \times (d - \lfloor d \rfloor) \tag{2.3}$$

Let $m = my\_level(d)$ and $n = mycousin\_level(d)$. Intuitively, given a node u and the distance value d, beginning with u, the algorithm can go m levels up to reach an ancestor w of u. Then, from w, the algorithm can go n levels down to reach a descendant v of w. Referring to the cousin distance definition in Figure 2, $c\_dist(u, v)$ must be equal to the distance value d. Furthermore, all the siblings of u must also be cousins of the siblings of v with the same distance value d. These nodes are identified by their unique identification numbers. To obtain cousin pair items having the form $(L(u), L(v), c\_dist(u, v), occur(u, v))$, the node labels of u and v are checked and add up the occurrence numbers for cousin pairs whose corresponding node labels are the same and whose cousin distances are the same. Figure 2.3 summarizes the algorithm.

Notice that within the loop (line 3 - line 10) of the algorithm in Figure 3, it finds cousin pairs with cousin distance d where d is incremented from 0 to *maxdist*. In line 8 where a cousin pair with the current distance value d is formed, the algorithm checks, through node identification numbers, to make sure this cousin pair is not identical to any cousin pair with less distance found in a previous iteration in the loop. This guarantees that only cousin pairs with exact distance d are formed in the

```
Procedure: Single_Tree_Mining
Input: A tree $T$ and a maximum degree value allowed, $maxdeg$, and a minimum
    occurrence number allowed, $minoccur$.
Output: All cousin pair items of $T$ where the cousin pairs have a degree less than
    or equal to $maxdeg$ and an occurrence number greater than or equal to
    $minoccur$.
1. for each node $p$ where $children\_set(p) \neq \emptyset$ do
2.   begin
3.     for each valid degree value $d \leq maxdeg$ do
4.       begin
5.         let $u$ be a node in $children\_set(p)$;
6.         calculate $m = my\_level(d)$ and $n = mycousin\_level(d)$ as defined
           in Eq.(2.3.2.1), (2.3.2.2);
7.         beginning with $u$, go $m$ levels up to reach an ancestor $w$ and
           then from $w$, go $n$ levels down to reach a descendant $v$ of $w$;
8.         combine all siblings of $u$ and all siblings of $v$ to form cousin pairs
           with the degree value $d$;
9.         if a specific pair of nodes with the degree $d$ has been found
           previously, don't double-count them;
10.      end;
11.  end;
12. add up the occurrence numbers of cousin pairs whose corresponding node
    labels are the same and whose cousin degrees are the same to get
    qualified cousin pair items of $T$.
```

Figure 2.3 Algorithm for finding frequent cousin pair items in a single tree.

current iteration in the loop.

Lemma 1. Algorithm Single_Tree_Mining correctly finds all cousin pair items of T where the cousin pairs have a distance less than or equal to *maxdist* and an occurrence number greater than or equal to *minoccur*.

Proof. The correctness of the algorithm follows directly from two observations: (i) every cousin pair with distance d where $0 \leq d \leq maxdist$ is found by the algorithm; (ii) because of Step 9 that eliminates duplicate cousin pairs from consideration, no cousin pair with the same identification numbers is counted twice.

Lemma 2. The time complexity of algorithm Single_Tree_Mining is $O(|T|^2)$.

Proof. The algorithm visits each children set of T. For each visited node, it takes at most $O(|T|)$ time to go up and down to locate its cousins. Thus, the time spent in finding all cousin pairs identified by their unique identification numbers is $O(|T|^2)$. There are at most $O(|T|^2)$ such cousin pairs. Through the table lookup, this algorithm gets their node labels and adds up the occurrence numbers of cousin pairs whose distances and corresponding node labels are the same in $O(|T|^2)$ time.

To find all frequent cousin pairs in a set of trees $\{T_1, \ldots, T_k\}$ whose distance is at most *maxdist* and whose support is at least *minsup* for a user-specified *minsup* value, first all cousin pair items in each of the trees that satisfy the distance requirement are found. Then all frequent cousin pairs can be located by counting the number of trees in which a qualified cousin pair item occurs. This procedure will be referred to as Multiple_Tree_Mining and its time complexity is clearly $O(kn^2)$ where $n = \max\{|T_1|, \ldots, |T_k|\}$.

Table 2.2 Parameters and Their Default Values Used in the Algorithms

Name	Meaning	Value
minoccur	minimum occurrence number of an interesting cousin pair in a tree	1
maxdist	maximum distance allowed for an interesting cousin pair	1.5
minsup	minimum number of trees in the database that contain an interesting cousin pair	4

2.4 Experiments and Results

A series of experiments was conducted to evaluate the performance of the proposed tree mining algorithms, on both synthetic data and phylogenies, run under the Solaris operating system on a SUN Ultra 60 workstation. The tree mining algorithms were implemented using the K programming language (`www.kx.com`). The synthetic data was produced by a C++ program based on the algorithm developed in [26]. This program is able to generate a large number of random trees from the whole tree space. The phylogenies were obtained from TreeBASE, available at `www.treebase.org` [48].

Table 2.2 summarizes the parameters of these algorithms and their default values used in the experiments. The value of 2 was used for minimum support because the phylogenies in TreeBASE substantially differ from each other and using this support value allowed users to find interesting patterns in the trees. Table 2.3 lists the parameters and their default values related to the synthetic trees. The *fanout* of a tree is the number of children of each node in the tree. The *alphabet_size* is the total number of distinct node labels these synthetic trees have.

Figure 2.4 shows how changing the *fanout* of synthetic trees affects the running time of the algorithm Single_Tree_Mining. 1000 trees were tested and the average was plotted. The other parameter values are as shown in Table 2.2 and Table 2.3. Given

Table 2.3 Parameters and Their Default Values Related to Synthetic Trees

Name	Meaning	Value
tree_size	number of nodes in a tree	200
database_size	number of trees in the database	1000
fanout	number of children of each node in a tree	5
alphabet_size	size of the node label alphabet	200

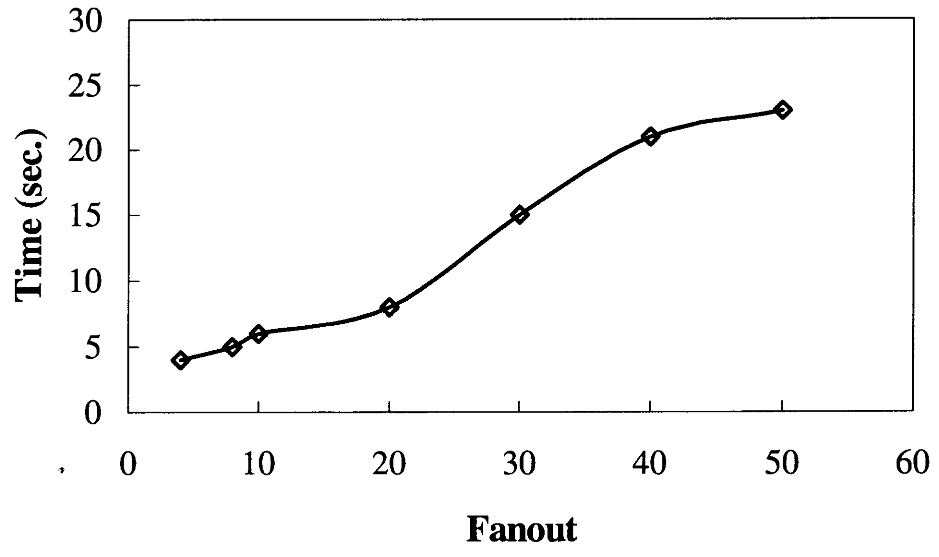


Figure 2.4 Effect of *fanout*.

a fixed *tree_size* value, a large fanout value will result in a small number of children sets, which will consequently reduce the times of executing the outer for-loop of the algorithm, c.f. Step 1 in Figure 3. Therefore, one may expect that the running time of Single_Tree_Mining drops as *fanout* increases. Surprisingly enough, however, Figure 2.4 shows that the running time of Single_Tree_Mining increases as a tree becomes bushy, i.e. its *fanout* becomes large. This happens mainly because for bushy trees, each node has many siblings and hence more qualified cousin pairs could be generated, cf. Step 8 in Figure 3. As a result, it takes more time in the post-processing stage to aggregate those cousin pairs, cf. Step 12 in Figure 3.

Figure 2.5 shows the running times of Single_Tree_Mining with different *maxdist* values for varying node numbers of trees. 1000 synthetic trees were tested and the average was plotted. The other parameter values are as shown in Table 2.2 and Table 2.3. It can be seen from the figure that as *maxdist* increases, the running time becomes large, because more time will be spent in the inner for-loop of the algorithm for generating cousin pairs, cf. Steps 3 - 10 in Figure 3. It is also observed that a lot of time needs to be spent in aggregating qualified cousin pairs in the post-processing stage of the algorithm, cf. Step 12 in Figure 3. This extra time, though not explicitly described by the asymptotic time complexity $O(|T|^2)$ in Lemma 2, is reflected by the graphs in Figure 2.5.

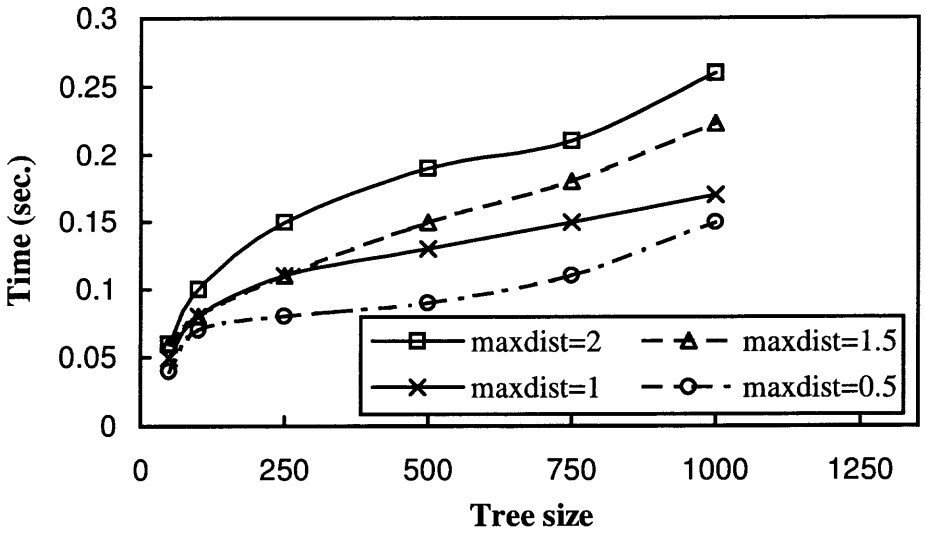


Figure 2.5 Effect of *maxdist* and *tree_size*.

Figures 2.6 and 2.7 show the running times of Multiple_Tree_Mining when applied to 1 million synthetic trees and 1,500 phylogenies obtained from TreeBASE, respectively. Each phylogeny has between 50 and 200 nodes and each node has between 2 and 9 children (most internal nodes have 2 children). The size of the node label alphabet is 18,870. The other parameter values are as shown in Table 2.2 and Table 2.3. Figure 2.7 shows that Multiple_Tree_Mining can find all frequent cousin pair items in the 1,500 phylogenetic trees in less than 150 seconds. The algorithm scales up well—its running time increases linearly with increasing number of trees (Figure 2.6).

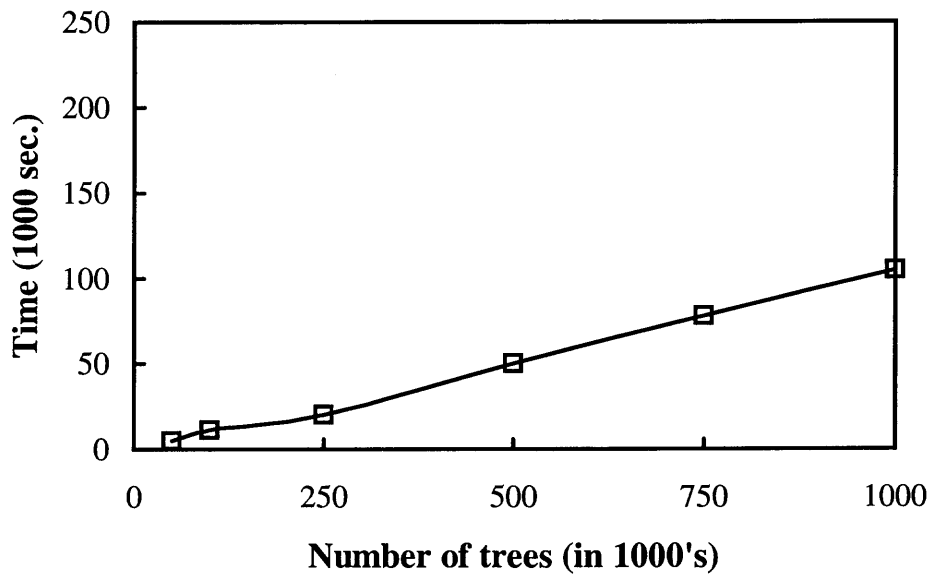


Figure 2.6 Effect of *database_size* for synthetic trees.

2.5 Applications

This section describes several applications of our techniques, showing how they can be used (i) to discover co-occurring patterns in multiple phylogenies; (ii) to evaluate the quality of a consensus tree obtained from equally parsimonious trees with the same Taxa (leaf nodes); and (iii) to find kernel trees from groups of phylogenies with different Taxa.

2.5.1 Discovering Co-occurring Patterns in Multiple Phylogenies

Multiple_Tree_Mining is applied to the phylogenies associated with each study in TreeBASE to discover co-occurring patterns in these phylogenies. The parameter values used in the algorithm are as shown in Table 2.2. Figure 2.8 shows some results for the phylogenies reported in the study [15] maintained in TreeBASE.

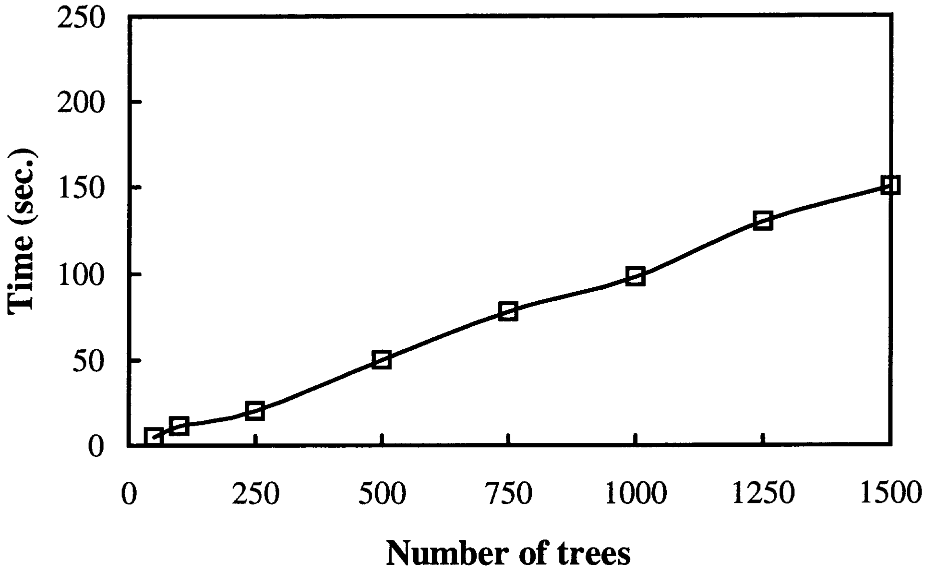


Figure 2.7 Effect of *database_size* for phylogenies.

These phylogenies were constructed for 8 seed plants (Taxa): Cycadales, Ginkgoales, Coniferales, Ephedra, Welwitschia, Gnetum, Angiosperms and Outgroup_Seed_Plants.

There are several interesting cousin pairs in Figure 2.8. For example, (Ginkgoales, Ephedra) is a frequent cousin pair with distance 1.5, which is highlighted by an underscore and occurs in the two trees in the two right windows in the figure. (Gnetum, Welwitschia) is another frequent cousin pair with distance 0, which is highlighted by bullets and occurs in all four trees in the figure. These frequent cousin pairs show evolutionary associations between the Taxa studied in [15].

2.5.2 Evaluating the Quality of Consensus Trees

One important subject in phylogeny is to automatically reconstruct phylogenetic trees from a set of molecular sequences or species. The most commonly used method is based on the maximum parsimony principle [22]. This method often generates multiple trees rather than a single tree for the input sequences or species. When the number of equally parsimonious trees is too large to suggest an informative evolution hypothesis, a consensus tree is sought to summarize the set of parsimonious trees.

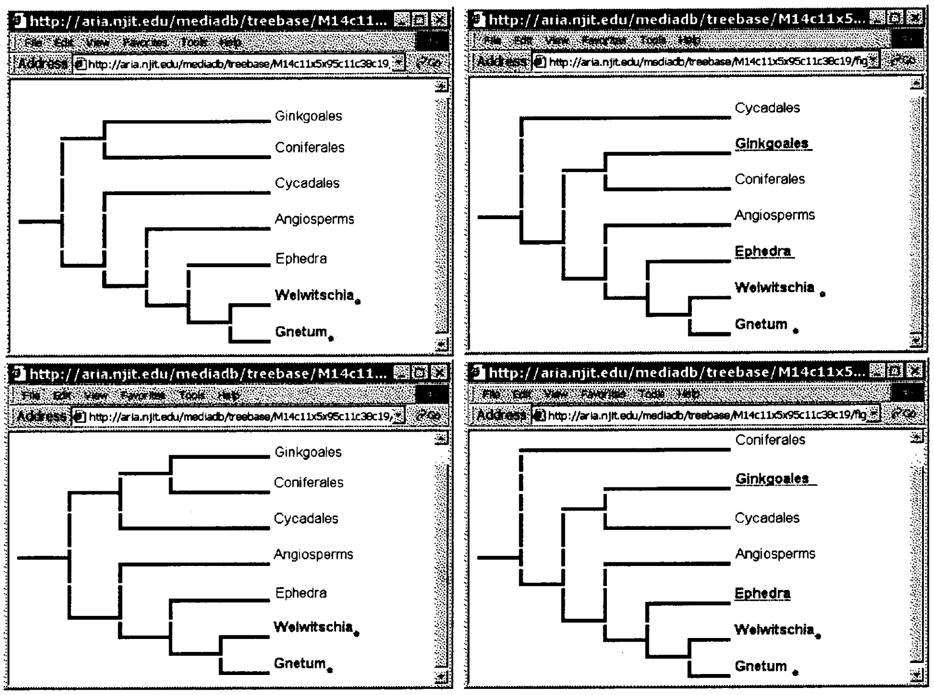


Figure 2.8 Discovering co-occurring patterns in multiple phylogenies.

Sometimes the set is divided into several clusters and for each cluster a consensus tree is derived. [54]

There are five most popular methods for generating consensus trees: Adams [1], strict [13], majority [38], semi-strict [8], and Nelson [42]. Here, the quality of the consensus tree generated by each of the above five methods are evaluated. The quality is measured by considering the cousin pairs shared between the consensus tree and the original parsimonious trees from which the consensus tree is generated.

Specifically, let C be a consensus tree and let T be an original parsimonious tree. the similarity score between C and T, denoted $\delta(C, T)$, is defined as

$$\delta(C,T) = \sum_{i=1}^{k} \frac{1}{1 + |c\_dist_C(cp_i) - c\_dist_T(cp_i)|} \tag{2.4}$$

where $|.|$ is the absolute value of the indicated number. Each cp_i, $1 \leq i \leq k$, is a cousin pair whose node labels occur in both C and T; $|c\_dist_C(cp_i) - c\_dist_T(cp_i)|$ is the difference of the cousin distances of the cp_i shared by C and T. Thus, if the shared cousin pair cp_i has the same distance in C and T, it will contribute 1 to $\delta(C,T)$. If its distance is different in C and T, the value it contributes to $\delta(C,T)$ will be less than 1.

Let S be the set of original parsimonious trees from which the consensus tree C is generated. The average similarity score of the consensus tree C with respect to the set S is

$$\Delta(C,S) = \frac{\sum_{T \in S} \delta(C,T)}{|S|} \tag{2.5}$$

where $|S|$ is the total number of trees in the set S. The higher the average similarity score C has, the better consensus tree C is.

Figure 2.9 compares average similarity scores of the consensus trees generated by the five methods mentioned above for varying number of equally parsimonious trees. The parameter values used by the algorithm for finding the cousin pairs are as shown in Table 2.2. The parsimonious trees were generated by the PHYLIP tool [20] using the first 500 nucleotides extracted from six genes representing paternally, maternally, and biparentally inherited regions of the genome among 16 species of Mus [36]. It can be seen from Figure 2.9 that the majority consensus method is the best, yielding consensus trees with the highest scores.

2.5.3 Finding Kernel Trees from Groups of Phylogenies

Existing phylogenetic distance measures, such as those implemented in the widely used COMPONENT tool [44], are designed for comparing evolutionary trees with the same Taxa (leaf nodes). However, some applications in phylogeny, such as supertree construction, are concerned with assembling information from smaller phylogenies that share some but not necessarily all Taxa in common. The COMPONENT tool doesn't work for these phylogenies.

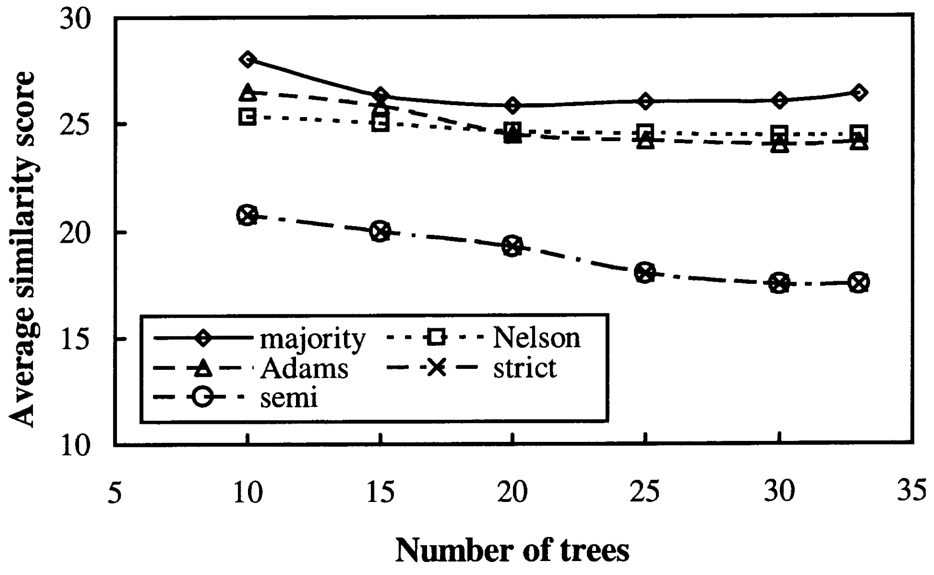


Figure 2.9 Comparing the quality of consensus trees using frequent cousin pairs.

A distance measure is proposed for comparing phylogenetic trees based on the frequent cousin pairs found in the trees. Specifically, let T_1 and T_2 be two trees. Let $cpi(T_1)$ contain all the cousin pair items of T_1 and let $cpi(T_2)$ contain all the cousin pair items generated from T_2. The *tree distance* of T_1 and T_2, denoted $t\_dist(T_1, T_2)$, is defined as

$$t\_dist(T_1, T_2) = \frac{|cpi(T_1) \cap cpi(T_2)|}{|cpi(T_1) \cup cpi(T_2)|} \tag{2.6}$$

Depending on whether the cousin distance and the number of occurrences of a cousin pair in a tree are considered, there are four different types of cousin pair items in the tree. Consequently four different tree distance measures are obtained. These measurements are represented by $t\_dist_{null}(T_1, T_2)$ (considering neither the cousin distance nor the occurrence number in each tree), $t\_dist_{cdist}(T_1, T_2)$ (considering the cousin distance only in each tree), $t\_dist_{occ}(T_1, T_2)$ (considering the occurrence number only in each tree), and $t\_dist_{occ\_cdist}(T_1, T_2)$ (considering both the cousin distance and the occurrence number in each tree), respectively.

For example, referring to the trees T_2 and T_3 in Figure 1, we have $t\_dist_{null}(T_2, T_3) = \frac{4}{12} = 0.33$, $t\_dist_{cdist}(T_2, T_3) = \frac{2}{16} = 0.125$, $t\_dist_{occ}(T_2, T_3) = \frac{4}{12} = 0.33$, $t\_dist_{occ\_cdist}(T_2, T_3) = \frac{2}{16} = 0.125$.[1]

Using the proposed tree distance measure, kernel trees from groups of phylogenies can be found. Specifically, considering k, $2 \leq k \leq 5$, groups of phylogenies, referred to as $jset_1, \ldots, jset_k$, where the phylogenies were generated by PHYLIP [20] using the LSU rDNA sequences representing 32 ascomycetes [35]. Data in the same group are parsimonious trees for the same Taxa while different groups share some but not all Taxa in common. The method finds the kernel trees $best_1, \ldots, best_k$ such that the average pairwise distance between the kernel trees is minimized and $best_i$ comes from $jset_i$. The distance measure used is the tree distance $t\_dist_{occ\_cdist}$ described above and the parameter values are as shown in Table 2.2. Figure 2.10 shows the time spent in finding the kernel trees as a function of the group number k. The found kernel trees could constitute a good starting point in building a supertree for the phylogenies in the k groups.

2.6 Extensions to Graphs

Some phylogenetic tree reconstruction methods such as MP [22] and ML [19] may produce unrooted unordered labeled trees. These trees are also known as free trees or undirected acyclic graphs (UAGs) [58, 70]. This section discusses an extension of our single tree mining algorithm to find frequent cousin pairs in one such graph.

In UAGs, the cousin distance between two nodes u, v is modified as follows:

$$c\_dist(u, v) = \frac{n}{2} - 1 \tag{2.7}$$

[1]The intersection and union of two sets of cousin pair items take into account the occurrence numbers in them. For example, suppose $cpi(T_1) = \{(a, b, m, occur1)\}$ and $cpi(T_2) = \{(a, b, m, occur2)\}$. Then $cpi(T_1) \cap cpi(T_2) = \{(a, b, m, \min(occur1, occur2))\}$ and $cpi(T_1) \cup cpi(T_2) = \{(a, b, m, \max(occur1, occur2))\}$.

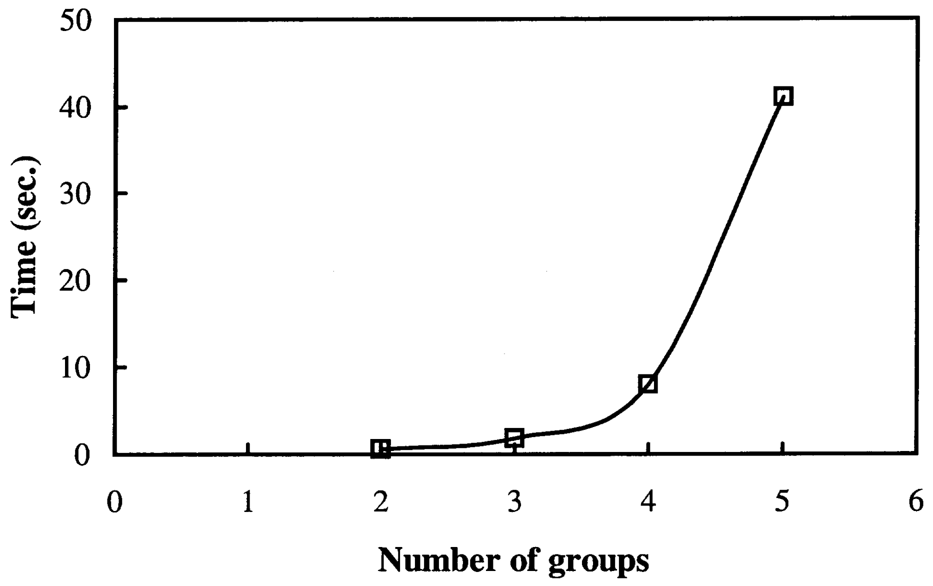


Figure 2.10 Time spent in finding kernel trees.

where n is the number of edges between u and v. Thus, given a cousin distance value d, the number of edges, n, between u and v can be calculated as follows:

$$n = (d+1) \times 2 \tag{2.8}$$

Given an undirected acyclic graph G, together with the maximum distance allowed ($maxdist$) and the minimum occurrence number allowed ($minoccur$), frequent cousin pairs can be found in G by arbitrarily choosing an edge e in G and putting an artificially created node r on e so that G becomes a rooted tree T_r with r being the root (see Figure 2.11). This tree consists of two subtrees, one being on the left side of r, denoted T_r^l, and the other being on the right side of r, denoted T_r^r. The Single_Tree_Mining algorithm in Figure 3 can be modified and applied to T_r as follows.

Consider each valid distance value $d \leq maxdist$. Let u be a node in a children set of T_r. The algorithm goes m levels up to reach an ancestor w, and then from w, it goes n levels down to get a cousin v of u where

$$m + n = (d+1) \times 2 \tag{2.9}$$

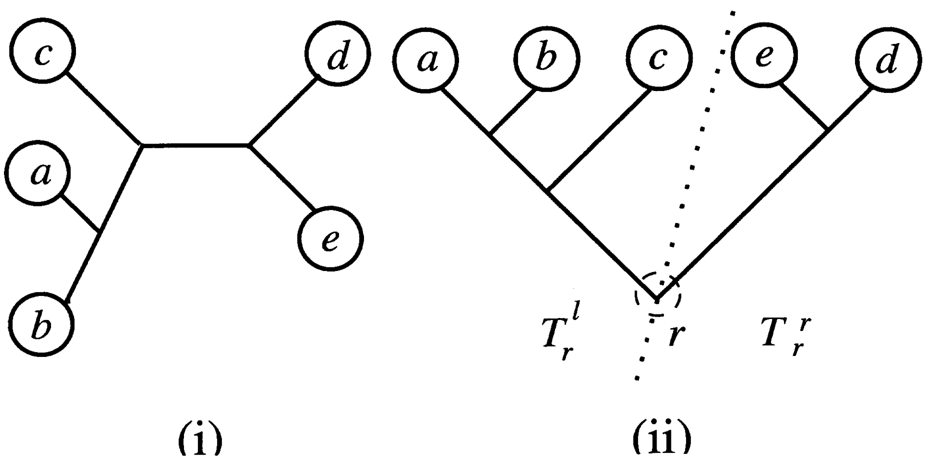


Figure 2.11 Illustration of converting an undirected acyclic graph (shown in (i)) to a rooted tree with node r being the root (shown in (ii)).

and $1 \leq m, n \leq [(d+1) \times 2] - 1$. Referring to Eq. (8), the $[(d+1) \times 2]$ represents the number of edges between u and v. Thus, instead of only considering *my_level*(d) and *mycousin_level*(d) as defined in Eq. (2), (3), all combinations of m, n satisfying Eq. (2.9) are considered. For example, suppose d is 2. Possible combinations of (m, n) include (1, 5), (2, 4), (3, 3), (4, 2), and (5, 1). The above calculation is correct when both u and v are in T_r^l, or both u and v are in T_r^r. Otherwise the additional edge created due to the insertion of the root r has to be considered. Specifically, suppose u is in T_r^l and v is in T_r^r. Then the m, n used in traversing the tree should be modified as follows:

$$m + n = [(d+1) \times 2] + 1 \tag{2.10}$$

and $1 \leq m, n \leq [(d+1) \times 2]$, to take into account the additional edge inserted in T_r. Clearly the time complexity of this algorithm is $O(|G|^2)$. One can easily extend this algorithm to find frequent cousin pairs in multiple graphs.

2.7 Summarization

This chapter presented new algorithms for finding and extracting frequent cousin pairs with varying distances from a single tree or multiple trees. The software for these algorithms can be downloaded from `http://cs.nyu.edu/cs/faculty/shasha/paper/cousins.k`. The algorithm for the single tree mining method, described in Section 3, is a quadratic-time algorithm. It is suspected that the best-case time complexity for finding all frequent cousin pairs in a tree is also quadratic. In the future, alternative approaches (e.g. dynamic programming) will be investigated to find these patterns in phylogenetic trees.

Notice that this approach differs from the work on computing least common ancestors of two nodes in a tree (e.g. [7, 25]) in that the definition of cousin distance is used to guide the search and mining process. Specifically given a cousin distance value d, beginning with a node u, the algorithm moves up to find an ancestor *anc* of u, and then from *anc* the algorithm moves down to reach a cousin v of u. The number of steps to move up and down is completely determined by the given distance value d. Thus, this method can systematically enumerate the cousins rather than taking random pairs of nodes and finding out what kind of cousins they are.

A similarity measure based on frequent cousin pairs is also introduced and used to compare five popular methods for consensus tree generation. This is the first attempt to evaluate the quality of consensus trees through a quantitative measure. Other possible measures could be based on the various distances for phylogenetic trees as described in [44].

CHAPTER 3

FREQUENT AGREEMENT SUBTREE MINING

3.1 Introduction

This chapter presents a new frequent substructure mining technique for finding frequent agreement subtrees from multiple phylogenetic trees. The agreement between a pattern p and a data tree t is measured by subtree isomorphism. The pattern p is said to be an agreement subtree of t, intrinsically if p is isomorphic to a subtree st of t (The concept of "agreement subtree" will be formally introduced in Section 3.2). In the context of data mining research, it is also said that p is supported by t, if p is an agreement subtree of t.

Agreement subtrees are traditionally regarded as significant patterns in phylogeny research. Different theories about the evolutionary history of the same set of species often result in different evolutionary trees, thus a fundamental problem challenging biologists is to determine how much the two theories have in common. To a certain extent, this problem can be answered by computing a maximum agreement subtree (MAST) of **two** given evolutionary trees[21]. When **multiple** evolutionary trees are of concern, finding out a MAST has the same importance; however, finding frequent agreement subtrees from **multiple** trees is expected to be a more flexible and more feasible alternative to finding only a MAST.

Consider a set of *five* phylogenetic trees built on *six* Hamamelis-related species and their subtrees obtained from the Study $S497$ [34] in TreeBASE [47]. The *five* trees and *three* subtrees are shown in Figure 3.1, and the *six* species are shown in Table 3.1. In the figure, the top row shows *five* phylogenetic trees, with each of them depicting a hypothesis about the evolutionary history of the *six* species. The bottom row shows *three* subtree patterns: st_1, st_2 and st_3. Obviously, if MASTs are

Table 3.1 Six Species

Species name	Symbol
Hamamelis_virginiana	1
Hamamelis_vernalis	2
Hamamelis_mexicana	3
Hamamelis_japonica	4
Fothergilla_major	5
Hamamelis_mollis	6

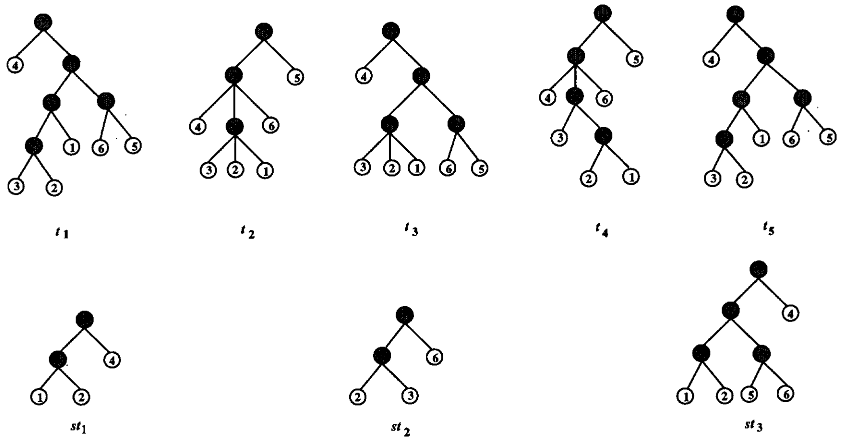


Figure 3.1 *Five* data trees and their subtrees.

the only interesting patterns in phylogenetic tree analysis, then the targeted patterns will include st_1 and st_2, but excluding st_3. This is simply because that st_1 and st_2 are supported by all *five* data trees; while st_3 is supported only by *three* trees - t_1, t_3 and t_5. However, still being supported by the majority of the data set, st_3 could be even more interesting than both st_1 and st_2, because the number of leaves of st_3 is prominently larger than the leaf numbers of the other two subtrees. In this sense, it is highly desirable that a new method can discover all such interesting patterns, including not only st_1 and st_2, but also st_3 in this case, as long as they are supported by a significant portion of the data trees.

Informally, the problem addressed by this chapter can be described as follows. Given a database of phylogenetic trees and a user specified *minsup* value, the goal of the proposed problem is to discover all frequent agreement subtrees which find supports higher than *minsup*. Although various tree mining algorithms have been developed for discovering different tree patterns in the scope of generic tree structured data, none of these algorithms is immediately effective in finding frequent agreement subtrees in a database of leaf-labeled trees. To the best of the author's knowledge, Phylominer, which will be introduced in this chapter, is the first mining algorithm in this field.

3.1.1 Related Work

This work focuses on algorithmic issues. The nature of the problem, however, inevitably puts this work at a fusing point of the traditional phylogenetic tree research and the state of the art of tree mining techniques.

Related Work in Agreement Subtree. As mentioned previously, the *maximum agreement subtree* approach is an important consensus method in phylogeny research. It can be used to reconcile different evolutionary trees for the same set of species, or it can be used to infer species trees by observing congruence in gene trees from multiple (unlinked)loci. Moreover, agreement tree distance [21] can be naturally defined between two trees by calculating the number of leaves of both trees not in their MAST.

Finden and Gordon [21] introduced a heuristic algorithm for the MAST problem on **two binary rooted** trees which has an $O(n^5)$ running time and does not guarantee an optimal solution. Kubicka *et al.* developed an $O(n^{(0.5+e)logn})$ algorithm for the same problem [31]. Steel and Warnow provided a polynomial time algorithm that takes about the $O(n^2)$ time complexity for finding a maximum agreement subtree of two trees [53]. In a more general sense, Kao reported an $O(nlog^2n)$ algorithm to compute MAST for trees with constant degrees (opposed to binary trees), and Farach *et al.* proposed an $O(n^{1.5}logn)$ algorithm [18] to calculate MAST between two rooted trees with unbounded degrees. When MAST problem is generalized from **two** trees to a set of **multiple** trees, the problem has been shown polynomial time solvable for trees with bounded degrees[4, 17, 18]. Both rooted and unrooted agreement tree problems have been well studied in articles[29, 33].

Amir *et al.*[4] proved that finding a MAST from an instance of multiple trees of unbounded degrees is NP-complete. In addition to this result, it has long been noticed that a maximum agreement subtree could usually be of small size, when a large number of data trees are concerned [23]. Unfortunately, it turns out in reality that one is usually presented with more than two trees, sometimes as many as thousands of trees [4]. In such case, a small MAST could be uninformative. For good measure, being able to find out only one single maximum agreement subtree will suffice to qualify most MAST algorithms, even for those datasets having more than one maximum agreement subtree. All these unfavorable aspects of the MAST problem motivate computer scientists to seek for a frequent agreement subtree mining solution - a more feasible alternative to MAST.

Related Work in Structure Mining. Graph mining and tree mining are two interrelated subfields in structural data mining research. In the recent years, various tree mining algorithms have been zealously researched. Asai *et al.* [5] proposed a rightmost expansion scheme to mine all subtrees in rooted ordered trees. Independently, Zaki [69] developed algorithms capable of finding frequent embedded tree patterns in a forest where he models documents by ordered rooted trees. Yang *et al.* [66] applied a customized rightmost expansion scheme to solving a frequent

XML query pattern discovery problem. More recently, the interests in tree mining algorithms have shifted to unordered tree mining area. Shasha *et al.* [57] developed a cousin mining algorithm to find patterns in unordered trees for both rooted and unrooted versions. Xiao *et al.* [64] proposed an efficient maximal frequent subtree mining solution through path joining operation. Asai *et al.* [6] and Nijssen *et al.* [43] independently discussed an essentially identical unordered tree enumeration technique which was for the first time introduced to tree mining problems. Shortly afterwards, Chi *et al.* [10] reported their unordered unrooted tree mining work by transforming an unrooted tree a rooted tree. Other related work on general tree matching, inclusion and isomorphism detection can be found in several literatures [11, 30, 50].

In parallel with tree mining, graph mining has also been deeply researched. Yan and Han [65] proposed a novel canonical graph form to find closed frequent subgraphs in graph data. Huan *et al.*. devised a different canonical form to efficiently discover frequent subgraphs in the presence of isomorphism [27]. For the readers who are interested in the state of the art of graph mining, they will be referred to a survey paper [63] by Washio and Motoda.

Different from previous researches, the Phylominer algorithm can efficiently and completely find exact agreement subtrees which are embedded in a given set of phylogenetic trees, it thus joins the many others already developed [24, 39, 46, 51].

3.1.2 Novel Contribution of Phylominer

The main contributions of this work can be highlighted as follows:

- Proposes and formalizes a unique frequent agreement subtree mining problem in rooted phylogenetic tree field.

- Adopts an effective phylogeny-aware canonical form, which is for the first time to be used in phylogenetic tree mining problems to mitigate the chore of dealing with isomorphism problem.
- Introduces a phylogeny-aware subtree pattern expansion scheme.
- Designs a novel tree mining algorithm, Phylominer, which is immediately useful for phylogeny research.
- Proves the correctness and analyzes the algorithmic complexity of Phylominer.
- Extends the mining algorithm to unrooted trees.

In addition to the above analytical contributions, the algorithm has also been fully implemented and the correctness of the implementation has been strictly verified. In the implementation, partition metric is used to verify the agreement of a subtree in a data tree. The Phylominer algorithm is experimentally evaluated on a large number of synthetic trees, and the algorithm has also been applied to the real datasets obtained from COMPONENT [44] package and TreeBASE website [47] to produce informative mining results.

The rest of this chapter is organized as follows. Section 3.2 introduces relevant terminologies of the problem. Section 3.3 presents Phylominer algorithm for finding frequent agreement subtrees from multiple trees. Section 3.4 analyzes the correctness and time complexity of the algorithm. Section 3.5 extends the Phylominer algorithm to unrooted labeled trees. Section 3.6 reports the experimental results of the algorithm on both synthetic data and real phylogenetic trees, showing the scalability and effectiveness of the proposed approach. Section 3.7 reports an online engine. Finally, section 3.8 summarizes the chapter and points out some future work.

3.2 Preliminaries

This work considers tree mining problems for both rooted trees and unrooted trees; however, a solid solution for the rooted tree problem will be studied first, then it will be shown that the unrooted tree mining problem can be similarly solved through a sandwiched **URRU** transformation.

Let L denote a set of labels, corresponding to a set of Taxa that could be species, proteins or genes under investigation. Let cardinality of L, denoted as $|L|$, to be k. Without loss of generosity, L can be instantiated as a set of k natural numbers $\{n_1, n_2, \dots, n_{k-1}, n_k\}$, where $n_i \in N$.

Phylogenetic tree. A phylogenetic tree t on L is a rooted tree where all leaves are labeled bijectively from the label set L; all internal nodes have no labels; and a special node, denoted as r(t), is distinguished as the root. Furthermore, the fanout of each internal node is at least two. The **size** of a phylogeny t is measured by the cardinality of L, i.e. the number of leaves of t.

Despite the insignificance of the left-to-right order among sibling nodes of its every internal node, an unordered tree is always represented in one specific order or another. As a matter of fact, by arbitrarily exchanging positions of sibling nodes, one particular unordered tree can be represented by an exponential number of different ordered trees. Obviously, any two such ordered tree representations are isomorphic[1] to each other. The **isomorphism** between two trees t and t' is denoted as $t \equiv t'$.

To prune a leaf. When a leaf l is removed from a tree t, it is said that the leaf l is pruned. In case that l has only one sibling, denoted as *l.sibling*, the parent of l, denoted as *l.parent*, will be suppressed as well. As a result, *l.parent.parent*

[1]This can be seen by establishing between nodes of two trees a bijective mapping which preserves the parent-child relationships in both trees.

will connect to *l.sibling* directly. Formally such operation is called a **forced edge contraction** or simply an **edge contraction**.

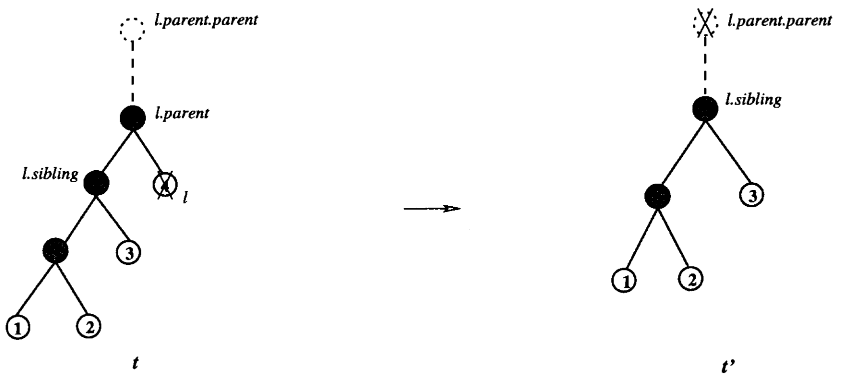


Figure 3.2 An example of pruning a leaf among two children of the tree root.

Remark 1. When a leaf *l* has only one sibling *l.sibling*, and *l.parent* is already the root of a tree, the above definition of **edge contraction** can not explain the case literally when *l* is pruned. This is because the node *l.parent*, being the root of the tree, can not have another node as its parent. To make the above definition of **edge contraction** applicable to such a special case, an existence of a virtual node, denoted as *vr*, can be imaged to be the dangling parent of the root, such that, *vr* can be used as the *l.parent.parent*. This way, when *l* is pruned, the **edge contraction** can still apply as connecting *l.sibling* to *l.parent.parent*, i.e. the *vr*. After that, the *vr* should be trimmed off, since the purpose of its imaginary existence is purely to make the **edge contraction** logically consistent with the above definition. Figure 3.2 shows one example of such situation. When leaf 4 is deleted, the edge connecting the root and the sibling of 4 is also contracted.

Observation 1. Pruning a leaf *l* from a phylogenetic tree *t* may trigger at most one edge contraction, which, if happens, consequently eliminates the internal node which is immediately adjacent to it, i.e. the parent of *l*.

Observation 2. Adding a new leaf to a phylogenetic tree may create a new internal node. This is because adding a new leaf to a tree t is the reversed operation of pruning a leaf from t.

Subtree. A tree st on SL is a subtree of t on L, if $SL \subset L$ and st is obtainable by restricting t to the leaf set SL through pruning all leaves $l \in L - SL$. This subtree relationship is denoted as $st \equiv t|_{SL}$.

For example, in Figure 3.3, t is a data tree reconstructed on a species set $L = \{1, 2, 3, 4, 5, 6\}$, and tree st_2 is a tree on a set $SL = \{1, 2, 4, 5\}$. st_2 can be obtained from t by pruning both the leaf 6 and the leaf 3. Therefore, st_2 is a subtree of t. The subtree is also called a restricted subtree, because it is obtained by restricting a data tree to a subset of the leaf set of the data tree.

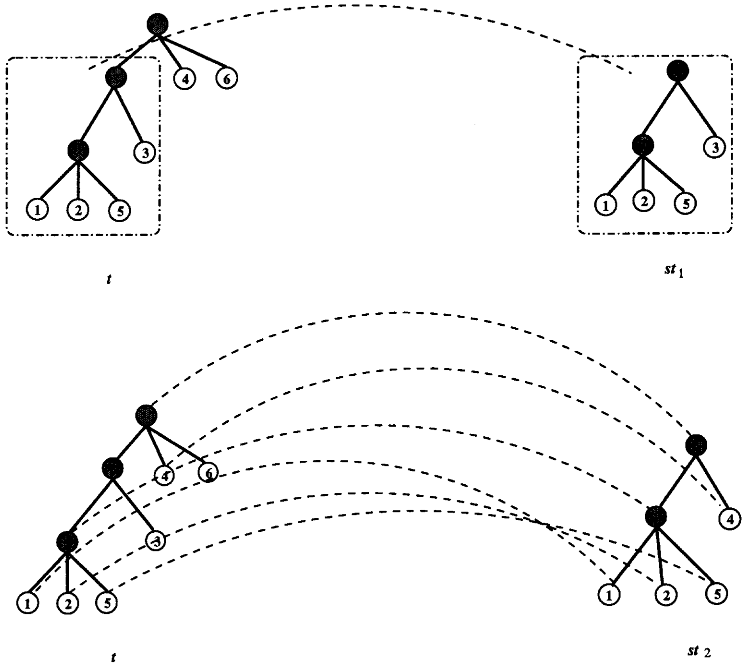


Figure 3.3 A subtree could be induced or embedded with respect to the data tree.

Apparently, due to possible edge contractions, a subtree is not necessarily an induced subtree of a data tree, where connectivities have to be strictly preserved.

Figure 3.3 shows *two* different cases: (i) st_1 happens to be an induced subtree of t; (ii) st_2 is not an induced subtree of t. In fact, by establishing a mapping relationship between all nodes of a subtree and the corresponding nodes of a data tree, it is clear that st_2 is mathematically embedded in the data tree t.

Formally, consider a tree t and a subtree st of the tree t, and let N_t and N_{st} be the sets of their nodes respectively. The subtree mapping is subject to the following injective function f: $N_{st} \rightarrow N_t$, if for all nodes $u, v \in N_{st}$

- f preserves labels, i.e. $label(f(u)) = label(u)$,
- f preserves ancestors, i.e. $f(u) \in desc(f(v))$ if and only if $u \in desc(v)$ and,
- f preserves least common ancestors (LCA), i.e. $LCA(f(u), f(v)) = f(LCA(u, v))$.

It will be shown in the next section that a proper consideration of this embedding feature between a subtree and a data tree is instrumental in developing a correct tree expansion scheme.

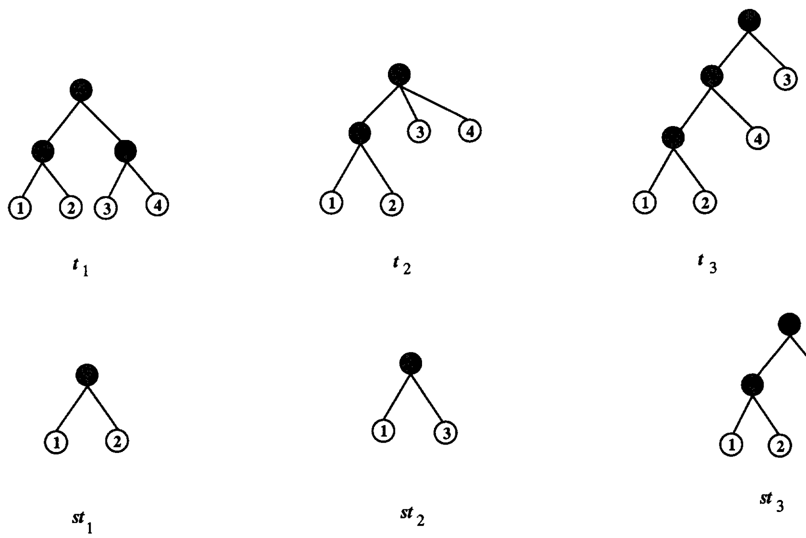


Figure 3.4 *Three* agreement subtrees and *one* maximum agreement subtree.

Agreement subtree. Let $DT = \{t_1, t_2, \ldots, t_m\}$ be a set of multiple phylogenies on the leaf set L and let SL be a subset of L. Then a leaf-labeled tree st on SL is an agreement subtree (or AST) for all $t_i \in DT$, if $t_1|_{SL} \equiv t_2|_{SL} \cdots \equiv t_m|_{SL} \equiv st$. The *size* of the AST is the number of leaves in the AST. As a special case, when m is 1, a subtree st of a single tree t can also be called an agreement subtree of t.

Maximum agreement subtree. If in addition, st has the maximum number of leaves among all agreement subtrees for DT, st is a *maximum agreement subtree (or MAST)* for DT.

In Figure 3.4, st_1, st_2 and st_3 are all agreement subtrees of 3 data trees; while only st_3 is also a MAST.

Frequent agreement subtree. Let $DT = \{t_1, t_2, \ldots, t_3\}$ denote a profile of trees on L where each $t \in DT$ is a leaf-labeled tree. For a given pattern p, p is said to be supported by a t if p is a subtree of t. Formally, $supp_{p,i}$ is defined to be 1 if t_i supports p; otherwise $supp_{p,i}$ is 0. The *support* of the pattern p with respect to DT is defined as $\sum_{1 \leq i \leq m} supp_{p,i}$. Thus the support value represents the number of trees in the profile of DT that support subtree p. A subtree is frequent if its support is more than or equal to a user-specified minimum support (*minsup*) value. It is typical that the *minsup* is given as a percentage of the total number of trees in DT. Given a user specified *minsup* value, the goal of the frequent agreement subtree mining algorithm is to efficiently discover all frequent agreement subtrees in a given DT. The frequent agreement subtree mining problem is considered as a data mining extension of the traditional MAST problem. Given a *minsup* of 50%, Figure 3.1 can be formally reinterpreted as follows, st_3 is a frequent subtree with a support value of 60%, because the subtree st_3 finds agreement from *three* out the *five* data trees; on the other hand, with support values of 100%, st_1 and st_2 are frequent subtrees having only 3 leaves though.

Table 3.2 Notations and Their Descriptions

Notation	Description
DT	A dataset of multiple phylogenetic trees
k-leaf (sub)tree	A (sub)tree with k leaves
C^k	A set of k-leaf candidate subtrees
F^k	A set of frequent k-leaf subtrees
t	A data tree of a phylogeny in DT
f^k	A k-leaf frequent subtree
t^k_{hlp}	A subtree after the heaviest leaf being pruned from a t^k
t^k_{shlp}	A subtree after the second heaviest leaf being pruned from a t^k
c^k_{hlp}	A subtree after the heaviest leaf being pruned from a candidate tree c
t_{hst}	The heaviest subtree of t
t_{chst}	The complementary subtree of t_{hst} with respect to t
j^k	A k-leaf subtree produced by the joining procedure
c^k	A k-leaf candidate subtree
$cns(t^k)$	A canonical newick string of a k-leaf subtree t^k

3.3 Frequent Agreement Subtree Discovery

Figure 3.5 shows a relatively more complicated example of 10 data trees and their 25 frequent subtrees meeting a *minsup* value of 30%. All the frequent subtrees, if correctly discovered, are listed in the figure according to the order of their leaf numbers. The first row shows all 2-leaf subtrees; the second row lists *eight* 3-leaf subtrees; the third row lists *three* 4-leaf subtrees; and no subtrees of 5 leaves or subtrees of larger sizes are frequent. In the figure, the support value of each subtree is also shown inside the square box under the subtree. In addition to these

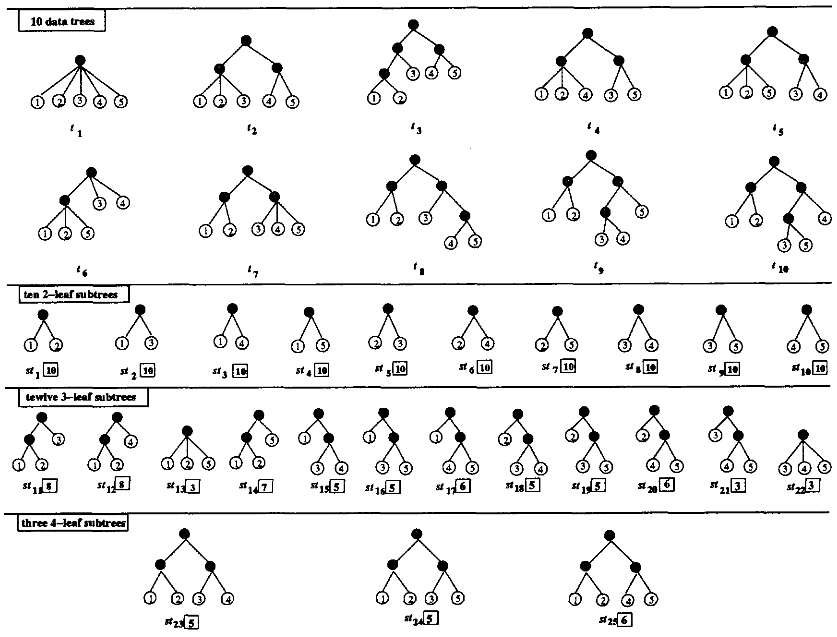


Figure 3.5 A running example of Phylominer on 10 data trees of *five* leaves under the *minsup* value of 30%.

information, supporting tree identifier lists for subtrees are also important and can be easily obtained through pattern-against-data verification. This section will present a novel algorithm which can discover all these information efficiently, correctly and completely.

Table 3.2 shows the notation that will be used in the rest of this chapter. For convenience, a tree t with k leaves is denoted as a k-leaf tree or simply a k tree, regardless how many internal nodes the tree t may have. The algorithm developed in this work is named Phylominer, which adopts the progressive Apriori approach to discover all subtrees level by level [3, 32, 69]. The high level pseudo code of the algorithm is shown in Figure 3.6.

Phylominer initially enumerates all $\frac{|L|*|L-1|}{2}$ 2-leaf subtrees, which are obtained by combinatorially assigning 2 different labels from L to a 2-leaf tree skeleton. All these 2-leaf trees are automatically frequent, because each of them appears in all data trees that are built on the same set of Taxa. Starting from this initial set, in the

```
Procedure: Phylominer($DT$, $\delta$, $\theta$)
Input: $DT$, a set of phylogenetic trees.
          $\delta$, a global variable for minsup.
          $\theta$, a global variable for scutoff, cutoff value of tree size.
Output: $FST$, a global variable for a set of frequent subtrees.
1.  $FST \leftarrow \emptyset$;
2.  $F_2$ $\leftarrow$frequent 2-leaf subtrees;
3.  $FST \leftarrow FST \cup F_2$;
4.  $EquivClasses_2$ $\leftarrow$equivalence classes of $F_2$;
5.  k = 2;
6.  while ($k < \theta$ and $|F_k| \geq k + 1$)
7.     begin
8.        $F_{k+1}$ = Grow_Frequent_Subtrees($EquivClasses_k$, $k$);
9.        $FST = FST \cup F_{k+1}$;
10.       $EquivClasses_{k+1} \leftarrow$ equivalence classes of $F_{k+1}$;
11.       $k = k + 1$;
12.    end;
13. return $FST$;
```

Figure 3.6 Algorithm for finding frequent subtrees in a database of trees.

subsequent passes, Phylominer iterates through the main computational loop. During each iteration, the algorithm will call the subroutine **Grow_Frequent_Subtrees** (Line 8 in Figure 3.6) to find frequent subtrees whose sizes are greater than the previous frequent ones by one leaf. Leaves in the algorithm correspond to items in traditional frequent itemset discovery. Namely, as these algorithms increase the size of frequent itemsets by adding a single item at a time; Phylominer algorithm increases the size of frequent subtrees by adding a leaf node once a time, regardless whether a new internal node will also be introduced or not. Obviously, the high level framework is typically an Apriori algorithm; however, what remains challenging is how the basic Apriori framework is going to be materialized in Phylominer algorithm.

Subsection 3.3.1 briefly introduces the tree format used in the algorithm. Subsection 3.3.2 focuses on a canonical form of phylogenetic trees. Subsection 3.3.3 discusses the usage of the canonical form in indexing subtrees. Subsection 3.3.4 formalizes the concept of equivalence class. Subsection 3.3.5 dedicates to a novel tree expansion method, which is the core of the algorithm. Subsection 3.3.6 discusses subtree verification using partition metric algorithm.

3.3.1 Tree Input Representation

The Phylominer algorithm uses Newick[2] notation to express input trees, intermediate candidate trees and final output trees. Newick notation represents a tree by a compact parenthesized string form, where a pair of parenthesis, i.e. '(' and ')', delimits the sibling relationship of the nodes immediately enclosed inside them, a comma ',' separates two sibling nodes, and a ';' terminates the string. For example, the Newick strings for trees t_1 and t_2 in Figure 3.1 are "*(4,(((3,2),1),(6,5));*" and "*((4,(3,2,1),6),5);*" respectively.

[2]A phylogeny standard data format.

Obviously, Newick string notation for phylogenetic trees is equivalent to, but more succinct than, the in-memory link list based general tree structure. Moreover, it requires only $O(N)$ time to convert an in-memory phylogenetic tree structure into its Newick string, and vice versa. Therefore, most tree manipulating operations used in this algorithm will be performed directly on Newick string notation of trees to achieve high efficiency. The only operation requiring the in-memory link-list-based general tree structure is to obtain a restricted subtree from a data tree; however, it happens only once for each verification of the presence of a candidate subtree in a potential data tree. Another operation which may require the in-memory link-list-based tree structure is the canonical labeling scheme described in Subsection 3.3.2. However, such operations are actually never performed, because the canonical form of any candidate subtree is automatically observed throughout the joining procedure(c.f. Lemma 7 in Section 3.4), which also greatly contributes to the efficiency of the mining algorithm. Finally, whenever appropriate, the Newick notation is used in this work to illustrate the details of joining operations.

3.3.2 Canonical Form of Phylogenetic Trees

Isomorphism is an important problem to solve in any unordered structural data mining task such as subgraph mining and generic unordered subtree mining, so is it in phylogenetic tree mining. The importance lies in that lack of consideration of isomorphism in candidate generation stage will produce a huge set of candidates with uncontrollable redundancy.

In this work, a canonical form of phylogenetic trees is formalized to solve tree isomorphism problem, such that each phylogenetic tree has only one valid form to distinctly represent it. The foundation of the canonical form is a total ordering among leaf labels in L, which simply conforms to the integer comparison property, i.e. the ordering of L is $1 < 2 < \ldots < n < n + 1 < \ldots$. Based on this leaf label ordering scheme, originally unlabeled internal nodes of the t can be assigned to virtual labels as following.

Label assignments of internal nodes. Each internal node will be assigned to a virtual label, which is exactly the label with the lowest rank among all the labels of its immediate children.

Here shows a bottom-up procedure to assign labels to all internal nodes including the root. The procedure is essentially a depth first traversal (DFT) process, which is going to be materialized with the following operation. When DFT backtracks to an internal node, the label of the lowest rank among all its children is picked to be the label of the internal node. This operation is performed at every internal node following the bottom up backtracking, until the root of the whole tree is reached. Finally, every internal node will be assigned to proper labels.

Hereafter, $l(v)$ is used to denote the label of a node v, regardless that v is an internal node with a virtual label or a leaf node with an original label. Then the canonical form of a phylogenetic tree can be defined as following.

Canonical form. Every unordered leaf-labeled tree can be uniquely represented by a canonical form, in which all nodes (including both leaf nodes and internal nodes) follow a normalized order, such that for every sibling pair (v, u), node v always appears before node u in the depth first traversal order if $l(v) < l(u)$.

The procedure to obtain a canonical form of a given tree is called the **normalization procedure**, which can be done by further enriching the above mentioned **label assignment procedure** with a sorting operation and a reordering operation at each internal node backtracking. To be concrete, when DFT backtracks to an internal node, the key operation to be performed is to sort the children nodes according to the total ascending order of their labels. Based on this order, all

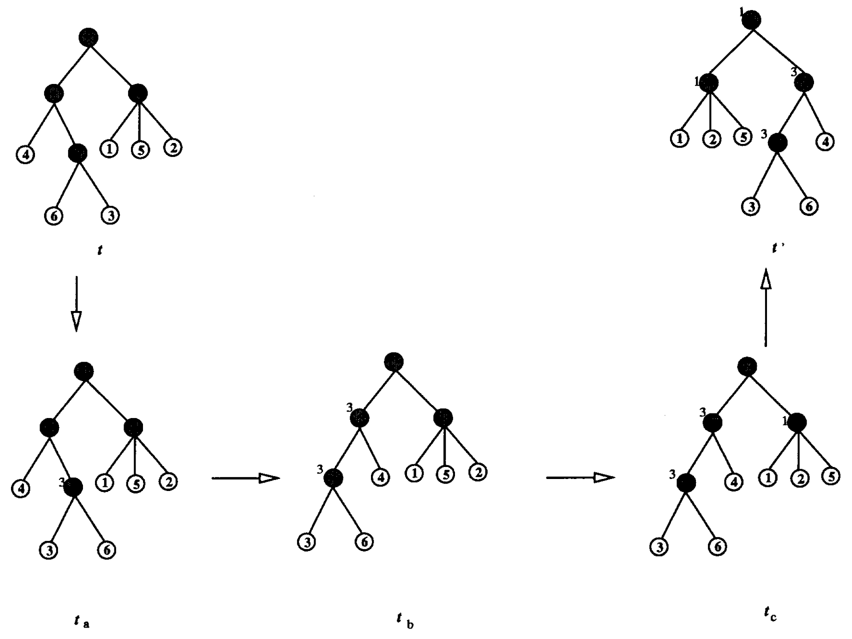


Figure 3.7 A running example to normalize a phylogenetic tree.

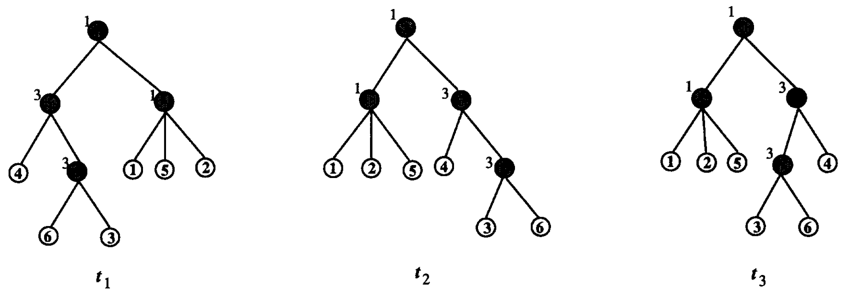


Figure 3.8 Examples of *three* rooted views of the same unrooted tree and the virtual labels assigned to their internal nodes.

children of the current internal node are then reordered from left to right to observe the canonical ordering among them. At meantime, the label with the lowest rank among its children is still picked to be the virtual label of the internal node. By performing these operations bottom-up starting from leaves of the tree, the subtree rooted at each internal node will be normalized recursively, and finally the whole tree rooted at the root will be normalized. Figure 3.7 shows a running example to normalize a tree.

Lemma 1. The above normalization procedure has time complexity $O(N)$, where N is the number of leaf nodes of the tree t.

Proof. Assuming there are I internal nodes in the tree t, and each $i \in I$ has $f(i)$ children nodes. To normalize a subtree at each internal node v requires sorting all its $f(i)$ children nodes, which can be done in $O(f(i))$ times by using the count sort algorithm. To normalize the t, the above sorting operation will be conducted at all I nodes, thus the total time complexity for normalizing the whole tree is $O(\sum_i f(i)) \leq O(2N) = O(N)$. Thus the lemma is proved.

As mentioned above, the canonical form is the most straightforward way to detect the isomorphism between two trees. Figure 3.8 shows an example of 3 different ordered representations of one same unordered tree. It can be seen that t_3 is in its canonical form; while t_1 and t_2 are not. However, once t_1 and t_2 are normalized to their canonical forms, the isomorphism of all these *three* trees is apparent.

Very naturally, the Newick string of a tree in its canonical form is called the **Canonical Newick String** of the tree. For example, the Canonical Newick String for the unordered tree in Figure 3.8 is "((1,2,5),((3,6),4));".

Lemma 2. It takes $O(N)$ time to obtain the Canonical Newick String of a phylogenetic tree, where N is the number of leaf nodes the tree has.

Proof. As previously shown, it takes $O(N)$ time to normalize a tree to its canonical form, and it takes also $O(N)$ time to get the Newick string from a canonical form. Therefore the total time to get a Canonical Newick String of a phylogenetic tree is $O(N) + O(N) = O(N)$.

Property 1. A direct pruning of the last leaf of a canonical form tree will result in a subtree still in its canonical form.

Property 2. A direct pruning of the last second leaf of a canonical form tree will result in a subtree still in its canonical form.

Remark 2. A direct pruning means that a simple pruning of a leaf without further normalizing the tree. In Section 3.3.4, it will be shown that these two direct pruning properties suggest a joining scheme, which emphasizes how to arrange the last leaves of both k-subtrees in getting a $(k+1)$-subtrees.

It should be noted that this canonical form is specifically defined for phylogenetic tree applications, which makes the discussed canonical form intrinsically different from other canonical forms proposed by [6, 10] for unordered tree mining and other canonical forms proposed by [32, 65, 27] for graph mining. This method, however, shares with the above mentioned canonical forms the same tenet in that they can systematically solve isomorphism problems posed by unordered relationship, thus alleviate the redundant candidates generation problem.

3.3.3 Indexing Phylogenetic Trees using Canonical Newick String

As previously shown, a phylogenetic tree can be represented by a unique Canonical Newick String. Thus two trees can be compared by directly comparing their string representations. As a consequence, the total ordering among trees' string representations can be used to index multiple trees. In developing the Phylominer algorithms, traditional database index methods such as hash table and B-trees *etc.*

have been applied to canonical tree strings to facilitate database related operations. For example, hashing technique is utilized to register a frequent tree to a proper equivalence class; B-tree structure is used to index all subtrees inside a particular equivalence class; finally, when it is time to retrieve a subtree from a set of frequent subtrees for the downward closure checking purpose, both hashing and B-tree algorithms will be used again.

3.3.4 Equivalence Class

Weight Scheme. Once all internal nodes have been labeled, every leaf $i \in n$ can be associated with a weight, denoted as $w(i)$, by concatenating the label from the root to the leaf.

For example, the weights of the leaves of t_3 in Figure 3.8 are the following, $w(1)$ is "1, 1, 1", $w(4)$ is "1, 3, 4", $w(5)$ is "1, 1, 5" and so on. The weights of leaf nodes can be compared from the most significant element down to the least significant element. For example, the weight order of the leaves in Figure 3.8 is $w(4) > w(6) > w(3) > w(5) > w(2) > w(1)$.

Heaviest leaf. The heaviest leaf, denoted as l_h, is the leaf with the heaviest weight among all the leaves.

Observation 3. If t is in its canonical form, the heaviest leaf l_h of a t is the last leaf of t according to the DFT order, i.e. the right most leaf of t. In fact, weights of all leaves in a normalized phylogenetic tree should be in ascending order from left to right.

Given a k-leaf tree t on a leaf set L, if any leaf $l \in L$ is pruned from t, a $(k-1)$-leaf subtree can always be obtained. Among these k $(k-1)$-leaf subtrees, the $(k-1)$-leaf subtree resulted from the pruning of l_h is called $(k-1)$-prefix subtree of t, denoted as t_{hlp}. For example, in Figure 3.9, the leaf of label 4 is the heaviest leaf

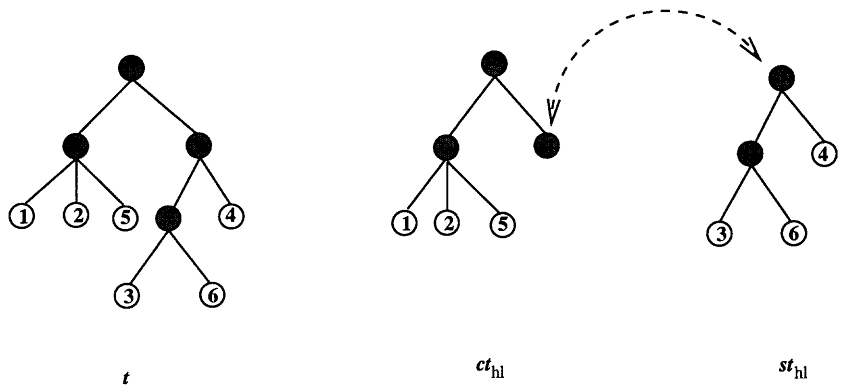


Figure 3.9 A tree can be separated into a heaviest subtree and its complementary subtree.

of t, and the rest part of t is t_{hlp}.

Equivalence class. Suppose both t and t' are in their canonical forms, t and t' are said to be in the same equivalence class if their $(k-1)$-prefix trees are isomorphic to each other, i.e. $t_{hlp} \equiv t'_{hlp}$, and the shared equivalence class is identified as the Canonical Newick String of their $(k-1)$-prefix tree.

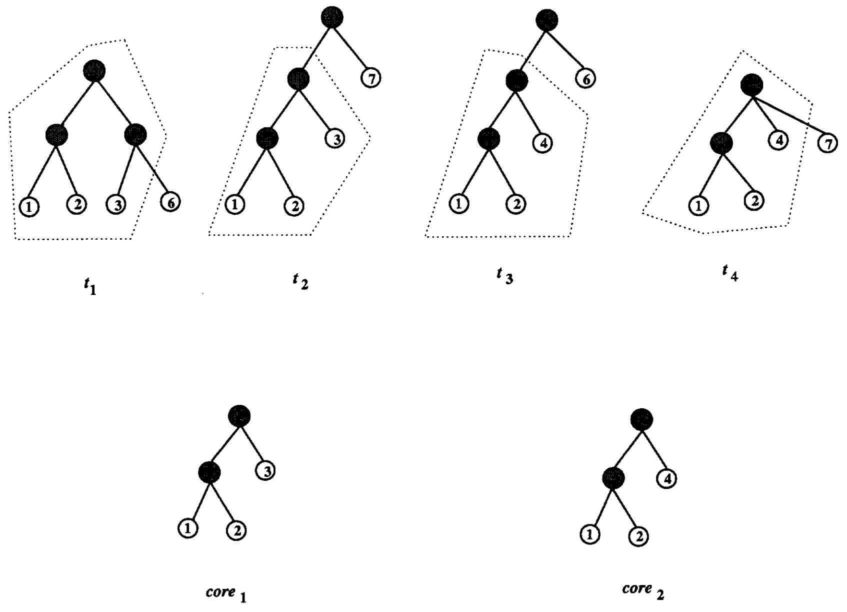


Figure 3.10 Tree 1 and tree 2 are in the same equivalence class; while tree 3 and tree 4 are in a different equivalence class.

The relation "having the same prefix tree as each other" for a set of subtrees is an equivalence relation, because the relation on these trees is reflective, symmetric and transitive. The equivalence relation partitions the set of k-subtrees into disjoint subsets called equivalence classes. Consider trees in Figure 3.10, t_1 and t_2 are in an equivalence class, because they share the same prefix tree $core_1$; while t_3 and t_4 are in another equivalence class, since they share the same prefix tree $core_2$.

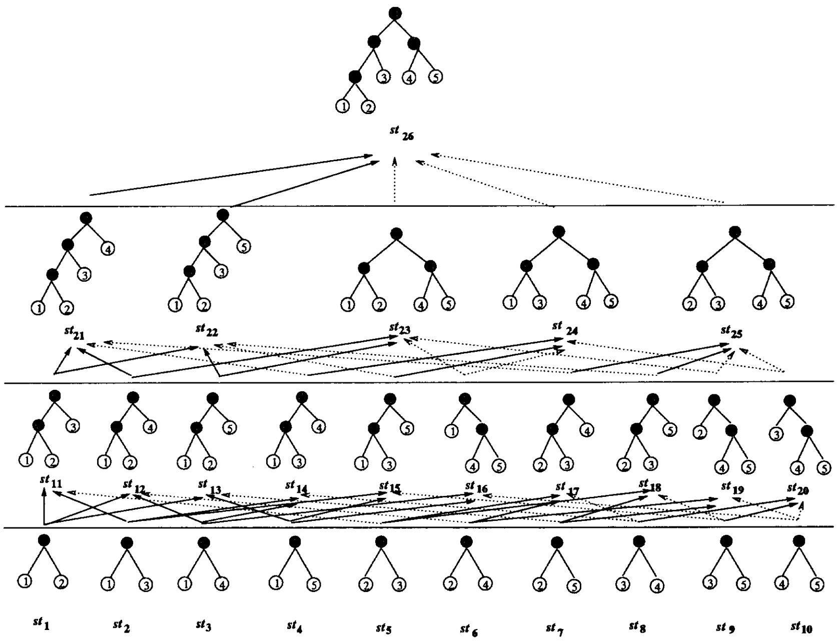

st_{26}
st_{21} st_{22} st_{23} st_{24} st_{25}
st_{11} st_{12} st_{13} st_{14} st_{15} st_{16} st_{17} st_{18} st_{19} st_{20}
st_1 st_2 st_3 st_4 st_5 st_6 st_7 st_8 st_9 st_{10}

Figure 3.11 An example of equivalence class based expansion lattice.

The heaviest local subtree. Given a tree t, the heaviest local subtree, denoted as st_{hl}, is defined as the subtree rooted at the parent of the heaviest leaf. Correspondingly, the rest part of the tree t after st_{hl} is taken off is called **the complementary tree**, denoted as ct_{hl}. For example, in Figure 3.9, leaf 4 is the l_h in t, and the st_{hl} is the heaviest local subtree of t while ct_{hl} is the complementary tree of the heaviest local subtree st_{hl}.

```
Procedure: Grow_Frequent_Subtrees($EquivClasses_k$, k)
Input: $EquivClasses_k$, the equivalence classes for all frequent k-trees.
       $k$, tree size.
Output: $F_{k+1}$, a set of frequent (k+1)-subtrees.
1.  $C_{k+1} \leftarrow \emptyset$, $EquivClasses_{k+1} \leftarrow \emptyset$;
2.  for each $aec \in EquivClasses_k$ do
3.    if $|aec| \geq 2$ then
4.      for each pair of elements $x, y \in aec$ which are not on the same leaf set, do
5.        if $(STID(x) \cap STID(y) > \delta)$ then
6.          $C \leftarrow Phylo\_Join(x, y)$;
7.          for each $c \in C$ do
8.            if downward_closure_checking($c$) = TRUE, then
9.              $freq \leftarrow Frequency\_Count(c)$;
10.             if $(freq > \delta)$ then
11.               $F_{k+1} \leftarrow F_{k+1} \cup c$;
12.               if $c_{hlp} \notin EquivClasses_{k+1}$ then
13.                 $EquivClasses_{k+1} \leftarrow EquivClasses_{k+1} \cup c_{hlp}$;
14.                 register c to $c_{hlp}$;
15. $F_{k+1} = \cup EquivClasses_{k+1}$;
16. return $F_{k+1}$;
```

Figure 3.12 Algorithm for finding all frequent (k+1)-trees based on k-trees.

3.3.5 Candidate Generation

In the candidate generation phase, candidate subtrees of size $k+1$ are generated from pairwisely joining trees over frequent k-subtrees. In order for two frequent k-subtrees to be eligible for further join, the two subtrees must be in the same equivalence class. This means except that the heaviest leaves of the both trees are different, the rest parts of the two trees are isomorphic to each other. The Figure 3.11 shows how joinings are performed without producing redundant candidate trees. This figure shows a total of 26 subtrees, with the largest one decomposable down to the smallest ones level by level. In this figure, a k-leaf subtree can be obtained through joining two $(k-1)$-leaf subtrees only when they are in the same equivalence class. In this case, solid lines have been drawn from them pointing to the expanded target trees. Otherwise, dashed lines have been drawn from $(k-1)$-leaf subtrees to proper k-trees. These dashed lines indicate that the involved $(k-1)$-leaf trees are subtrees of those k-trees; however, they are not eligible to participate any joining operations. For example, a 4-leaf subtree st_{23} actually has $four$ 3-leaf subtrees, which are st_{12},st_{13}, st_{16} and st_{19}. Here, if simply taking a combinatorial joining point of view, st_{23} should be obtainable from any of the six different pairwise joinings from the 4 trees. However, all these joinings, if conducted, will duplicate exactly same candidates. This redundancy is obviously undesirable and turns out to be avoidable by wisely joining only subtrees in the same equivalence class. To be specific, only st_{12} and st_{13} are in an equivalence class and thus will join to form st_{23}, any other pairwise subtrees won't be joined because they are not in the same equivalence class.

The nature of the equivalence class suggests to expand patterns through a rightmost joining (reminiscent of the rightmost extension schemes in [5, 69]). Following this inspiration, the focus of joining is thus on how to form a new $(k+1)$-tree by correctly gluing the 2 heaviest leaves of the two k-subtrees to the isomorphic part of both k-subtrees, while this isomorphic part itself is a $(k-1)$-leaf prefix tree.

Furthermore, if two trees are in the same equivalence class, the differences of the two trees will happen only within their heaviest subtrees respectively; at mean time, the complementary trees are either the same or one is the subtree of the other. Therefore, the essence of of the joining operation is actually how to perform the joining operation on the two heaviest subtrees to get the expanded heaviest candidate subtrees. Once the joining can be performed systematically on the two heaviest subtrees, each expanded heaviest candidate subtree will be glued back to the smaller complementary tree to form the final candidate subtree.

Based on the above analysis, the problem of joining two k-leaf subtrees has been transformed to how to join their heaviest subtrees; however, these heaviest subtrees are not necessarily in either the same size or in the same topology. Depending on what topologies the two trees or the two heaviest subtrees have, the joining operations can be formally classified into the following two cases, and in each case, at most 4 different candidate trees can be generated from joining the two trees.

- Case 1. When two trees are of same topology.
  - Case 1.1. When both the heaviest subtrees are binary trees, four potential candidates can be generated. Suppose (lt, hl_1) and (lt, hl_2) are the heaviest subtrees of t_1 and t_2 respectively, where hl_1 and hl_2 are heaviest leaves in t_1 and t_2 respectively, and lt denotes the left subtree in both heaviest trees since the left subtrees must be the same. Obviously, in the expanded candidate tree, hl_1 and hl_2 could be sibling, so two candidates are denoted as $j_{[1]} = (lt, lower(hl_1, hl_2), higher(hl_1, hl_2))$ and $j_{[2]} = (lt, (lower(hl_1, hl_2), higher(hl_1, hl_2)))$ respectively. Examples of $j_{[1]}$ and $j_{[2]}$ are illustrated by 4-leaf subtrees of j_{4-1} and j_{4-2} respectively in Figure 3.13. Another way to understand the joining operation on two k-leaf trees is to take one tree as the skeleton, and then to expand the skeleton by adding the heaviest leaf from the other tree to get a $k+1$ size

tree. Recall the observation 2 in Section 3.2, it is known adding a new leaf to a tree may also create a new internal node. Therefore, two additional candidates should also be considered and they are $j_{[3]} = ((lt, hl_1), hl_2)$ and $j_{[4]} = ((lt, hl_2), hl_1)$. In this sense, $j_{[2]}$ actually introduces a new internal node also. Examples of $j_{[3]}$ and $j_{[4]}$ are illustrated by 4-leaf subtrees of j_{4-3} and j_{4-4} respectively in Figure 3.13

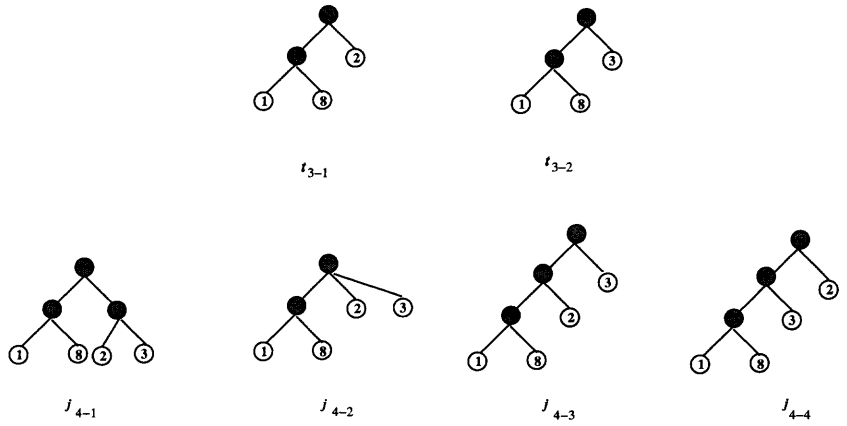


Figure 3.13 An example for case 1.1 expansion.

- Case 1.2. When both the heaviest subtrees are multi-forked trees, two potential candidates should be generated. Suppose that $(st_1, \ldots, st_m, hl_1)$ and $(st_1, \ldots, st_m, hl_2)$ are the heaviest subtrees of t_1 and t_2 respectively, where $st_1, \ldots st_m$ are the m ($m \geq 2$ due to that they are multi-forked.) sibling subtrees (or branches) of hl_1 and hl_2. The expanded candidates can be in either the form of $j_{[1]} = (st_1, \ldots, st_m, min(hl_1, hl_2), max(hl_2, hl_1))$ or the form of $j_{[2]} = (st_1, \ldots, st_m, (min(hl_1, hl_2), max(hl_1, hl_2)))$. Figure 3.14 shows examples of $j_{[1]}$ and $j_{[2]}$.

- Case 2. When the two heaviest subtrees are of different topologies, only one candidate $(k+1)$-leaf subtree can be generated. Since the two heaviest subtrees are different from each other, one of them can be further identified as the larger tree, and the other one the smaller tree.

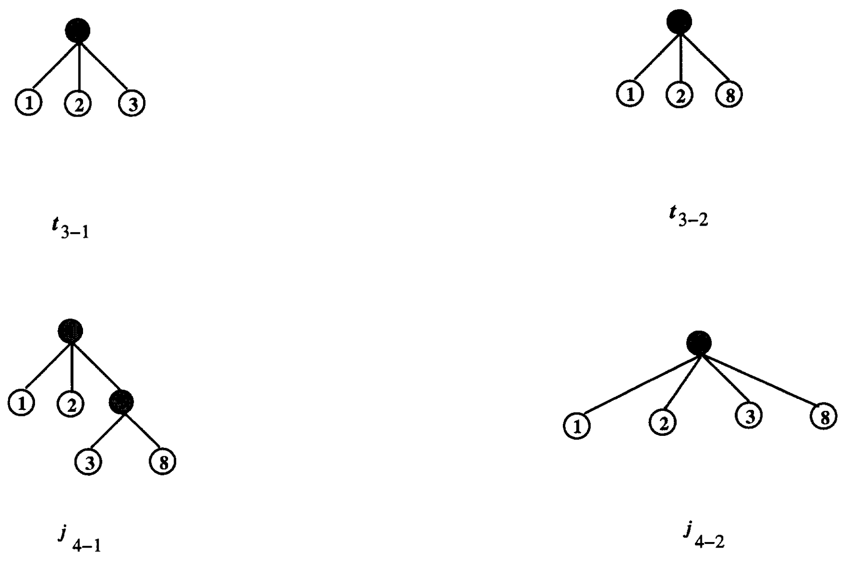


Figure 3.14 An example for case 1.2 expansion.

Formally, let $h(t)$ and $s(t)$ denote the height and the size of a tree t respectively. Given two trees t_1 and t_2, t_1 is identified to be larger than t_2, if either of the following rules holds.

- Rule 1. $h(t_1) > h(t_2)$. It means the nesting level of t_1 is larger than that of t_2,
- Rule 2. $s(t_1) > s(t_2)$. This case may happen when $h(t_1) = h(t_2)$. In this case, the fanout of the root of t_2 must be 2, and the fanout of the root of t_1 must be 3. Otherwise rule 1 will apply.

Let t_1 and t_2 to be expressed as (t_{1hlp}, hl_1) and (t_{2hlp}, hl_2) respectively. When t_1 is larger than t_2, hl_1 must be the heaviest leaf in the expanded subtree, and there must exist a subtree lst in t_{1hlp} which is isomorphic to t_{2hlp}. Let lst to be replaced by t_2, then get the $(t_{1hlp} \oplus hl_2, hl_1)$ as the final expanded tree, where $\oplus$ denotes the gluing operation. This joining operation can be easily understood if the larger tree is taken as an umbrella under which is a part of the larger tree replaced by the whole smaller tree. Figure 3.15 and Figure 3.16 show examples for rule1 and rule 2 respectively.

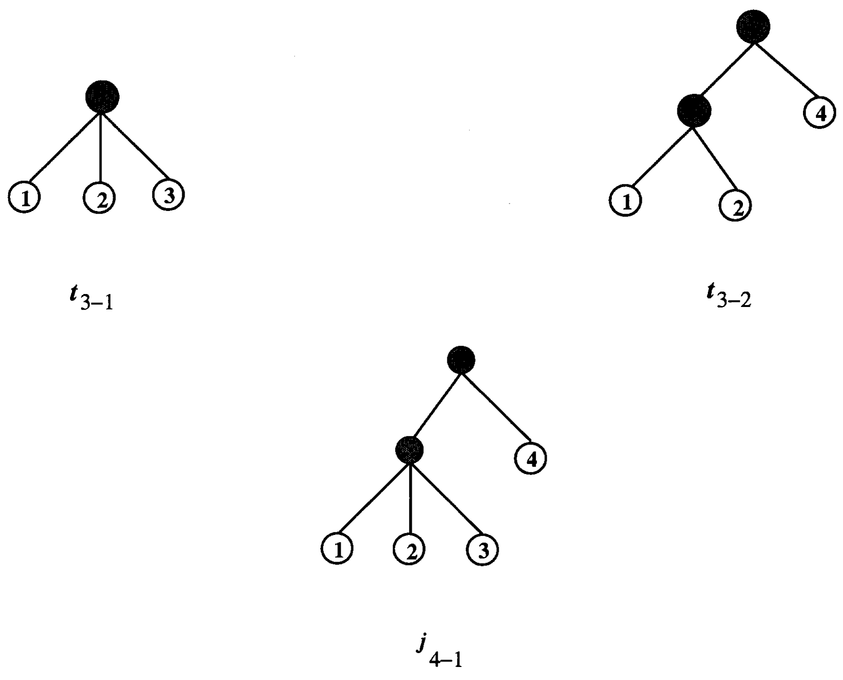


Figure 3.15 An example for rule 1 of case 2 expansion.

The overall algorithm for discovering all frequent $(k+1)$-subtrees from all k-subtrees is shown in Figure 3.12. For each pair of k-leaf frequent subtrees that are in the same equivalence class, the procedure **Phylo_Join** is called at line 6 to generate all possible candidate subtrees of size $k+1$. For each j^{k+1} produced by the above presented joining procedure to be a candidate c^{k+1}, it has to be verified to be frequent by downward closure checking[3]. If the j^{k+1} passes the downward closure checking, **Phylo_Join** then appends it to C^{k+1}, otherwise the j^{k+1} can be safely discarded. There is no need to check whether a c^{k+1} is already in C^{k+1} or not, since each particular j^{k+1} can only be generated once due to the fact that all subtrees so generated are always in their canonical forms.

[3]Please note that the downward closure checking is done by hashing on the leaf sets of those k-leaf subtrees.

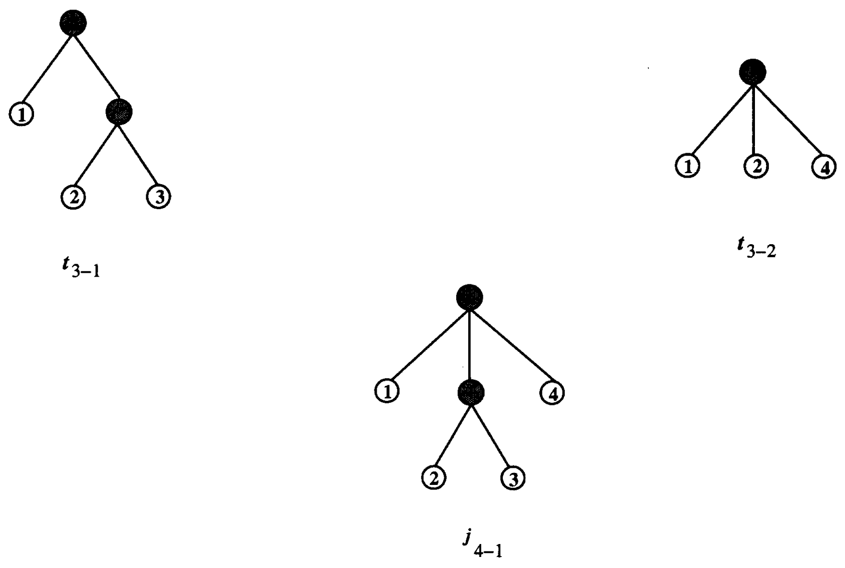


Figure 3.16 An example for rule 2 of case 2 expansion.

Compared with other tree expansion algorithms achieved through joining or enumeration, the joining scheme described here is remarkably unique, since it has been carefully designed to take the intrinsic features of phylogenetic trees into consideration. To be specific, on one side, the joining procedure is able to produce all agreement subtree candidates which are mathematically embedded in some data trees; on the other side, the self joining operation which should be considered in most other tree mining algorithms is never performed in this problem. This is so because all leaves of any phylogenetic tree are uniquely labeled. For the same reason, the algorithm never considers joining two subtrees with different topologies but sharing exactly same all k leaves.

Lemma 3. The joining can be done in $O(k)$ time, where k is the number of leaves of the two data trees.

Proof. The joining operation is performed on the newick strings of two data trees, each of which has the length of $O(k)$, where k is the size of leaf label set of both trees. The only operations used in the joining procedure are string parsing, string

Procedure: Phylo_Join(xt, yt)

Input: xt and yt are two trees in the same equivalence class.

Output: C, a set of candidate $(k+1)$ subtrees.

```
1.  $C \leftarrow \emptyset$;
2.  split($xt$) $\rightarrow$ ($xt_{hst}, xt_{chst}$) ;
3.  split($yt$) $\rightarrow$ ($yt_{hst}, yt_{chst}$) ;
4.  if Same-topology($xt_{hst}, yt_{hst}$) then
5.    if Binary-root($xt_{hst}, yt_{hst}$) then
6.      $(xt_{hst} \otimes yt_{hst}) \rightarrow Ch_{[1-4]}$; /*case 1.1*/
7.    else
8.      $(xt_{hst} \otimes yt_{hst}) \rightarrow Ch_{[1-2]}$; /*case 1.2*/
9.  else
10.   if $height(xt_{hst}) > height(yt_{hst})$ then
11.     $(xt_{hst} \otimes yt_{hst}) \rightarrow Ch_{[1]}$; /*case 2.1*/
12.   else
13.     $(xt_{hst} \otimes yt_{hst}) \rightarrow Ch_{[1]}$; /*case 2.2*/
14.  for each elements $ch_i \in Ch_{[]}$ do
15.    $c_i \leftarrow ch_i \oplus smaller(xt_{chst}, xy_{chst})$;
16.  $C \leftarrow C \cup c_i$;
17.  return $C$;
```

Figure 3.17 Algorithm for joining two k-trees to form candidate $(k+1)$-trees.

replacement and string concatenation, which can all be done within $O(k)$. The lemma is thus proved.

Lemma 4. The separating operation and back-gluing operation can be done in $O(k)$ time.

Proof. To separate the heaviest subtree from a tree t needs $O(N)$ time, because the essential operation is a consecutive string extraction action, which can be done within $O(N)$ time. At the end of the expansion, an expanded heaviest subtree needs to be glued back to a complementary tree, which will also need $O(N)$ time, because it can be accomplished by a substring replacement operation.

3.3.6 Frequency Counting

Once all candidate subtrees are generated from the current joining iteration, Phylominer will compute the support for each candidate by checking its number of occurrences in the given data trees. Given a candidate tree p on SL and data tree t on L, $t|_{SL}$ can be obtained by pruning all leaves $l \in L - SL$ from t. Obviously, the p is an agreement subtree of t if and only if $t|_{SL}$ will be isomorphic to p. The isomorphism between two trees is verified by calculating their partition metric. Two trees are isomorphic with each other, if and only if the partition metric of the two trees is *Zero*. The most efficient algorithm to calculate partition metric has a the time complexity of $O(N)$[13]. The basic idea of this most efficient algorithm is to represent each leaf with a binary number, so that a partition can be uniquely decided by summarizing all binary bits of those leaves in that partition. Once all partitions are represented by numbers, the partition metric can be efficiently obtained by simply comparing those numbers.

To further optimize the algorithm, supporting tree ID (STID) list [16, 68] is maintained to narrow down the searching scope for verifying the presence of a subtree in data trees. Being associated with each subtree, STID is actually a simple data structure which memorizes a list of identifiers of those data trees that support the subtree. Before the frequency of a c^{k+1} is to be computed, the intersection set of the STID lists of its frequent k-leaf subtrees will be computed first. If the size of the intersection list is already below the support, the c^{k+1} will be safely pruned. Otherwise, one-against-one agreement verification can be limited to the relatively shorter intersection STID list only. Actually, this optimization technique is performed immediately before the expansion stage, so that there is no need to perform further expansion for two f^k subtrees, if the cardinality of their interaction STIDs is already too small(c.f. line 5 in Figure 3.12).

3.4 Correctness and Complexity Analysis

Lemma 5. (Correctness) Any discovered subtree is a positive agreement frequent subtree.

Proof. In order to be identified as a frequent subtree, a candidate has to pass frequency counting step. Therefore, the correctness of Phylominer algorithm lies in the correctness of the frequency counting step. While the frequency counting step is actually to count the number of those data trees which support the candidate, the correctness of frequency counting again lies behind the loyalty the agreement verification achieved by the partition metric. Since the correctness of the partition metric is obvious, the lemma is thus proved.

Lemma 6. (Completeness) It is sufficient to consider only trees in the same equivalence class for joining operations, even then one still enumerates all possible candidate subtrees. In other words, the joining is complete without missing any frequent subtree.

Proof. The lemma is proved by using the mathematical induction method. It is sufficient to show that any $(k+1)$-leaf frequent subtree can always be generated from two k-leaf frequent subtrees.

Base step:

Check it is true for $k = 2$. Clearly, all frequent 2-leaf subtrees will be successfully discovered, because the frequent 2-leaf subtrees are obtained through a brute force enumeration. Thus the statement holds for $k = 2$.

Hypothesis step:

Now it is safe to assume the lemma holds when $k = n$, i.e. all n-leaf frequent subtrees are already identified to be frequent, then it will be shown that the lemma also holds when $k = n + 1$, i.e. all potential $(n + 1)$-leaf subtree candidates will be generated.

Induction step:

Suppose a nt of $n + 1$ leaves is a frequent subtree in its canonical form (A tree t can always be normalized to nt), it will be shown that nt cannot be missed by the candidate generation step.

Given a nt, it is known that nt_{hlp} and nt_{shlp} can be obtained by taking off the heaviest leaf and the second heaviest leaf from nt, respectively. Obviously, both nt_{hlp} and nt_{shlp} are in their canonical forms (Recall properties 1 and 2 in Section 3.3.2). From the downward closure theory, it is known, if nt is frequent, nt_{hlp} and nt_{shlp} must be frequent, thus must be in F^k; moreover, they must be in the same equivalence class. Therefore nt_{hlp} and nt_{shlp} will be joined by **Phylo_Join** procedure. Since the joining procedure has exhaustively considered all possible expansions, nt must be in the candidates set from joining on nt_{hlp} and nt_{shlp}.

By induction, it can be concluded that the lemma holds for all subtrees of varying number of leaves, i.e. all frequent subtrees will be generated. The Lemma is thus proved.

Theorem 1. Phylominer will correctly find all frequent agreement subtrees.

Proof. Since Phylominer is based on a candidate generation-and-verification scheme, the theorem is immediately true given lemma 5 and lemma 6 have been previously proved.

Corollary 1. (Non-Redundancy Candidate Generation) Each candidate is generated once at most.

Proof. Obviously, t_{hlp} and t_{shlp} are unique, and the results of different joinings are disjoint. Therefore, no tree will be generated more than once. Thus, the corollary is proved.

Lemma 7. (Automatic Canonicalization) The joining operation will generate candidate trees automatically in their canonical forms.

Proof. Please notice that, as the initial set, all frequent 2-leaf subtrees can be normalized directly by arranging the smaller leaf at the left branch, and the larger right on a binary tree. After that, in the subsequent joining iterations, the ways in all joining cases to arrange the two heaviest leaves in the subtree expansion scheme will guarantee that each expanded subtree will also be in its canonical form. Therefore, all discovered frequent subtrees are in their canonical forms. The lemma is thus proved.

This automatic normalization is a main factor contributing to the efficiency of the algorithm.

Theorem 2. The time complexity of Phylominer is $O(|F|^2mN)$, where $|F|$ is the cardinality of the frequent subtree set, m is the number of data trees, and N is the size of the label set.

Proof. Inside each pair of joining, it requires $O(k)$ time to generate up-to *four* (a constant) candidates, and each candidate-against-data tree verification costs $O(N)$

time. Since there are a total of m data trees to be verified against each candidate, the total time complexity of each pair of joining is $O(k+Nm) = O(N+Nm) = O(Nm)$. Considering that there are at most $|F|^2$ valid pairs of joinings, the total time complexity is $O(|F|^2 mN)$. Thus, the theorem is proved.

Remark 3. This is a very pessimistic upper bound, because with pattern size growing, the number of data trees which need to be verified against patterns drops quickly, therefore the real number of data trees involved in verification step is far less than m. Another reason is that the pairwise joining operation happens only in the same equivalence class, therefore $|F|^2$ is a very loose upper bound for the number of iterations. As a matter of fact, this upper bound can never be reached when the algorithm is applied to most real world datasets.

Remark 4. This is a pseudo polynomial time algorithm, since $|F|$ is not an input parameter.

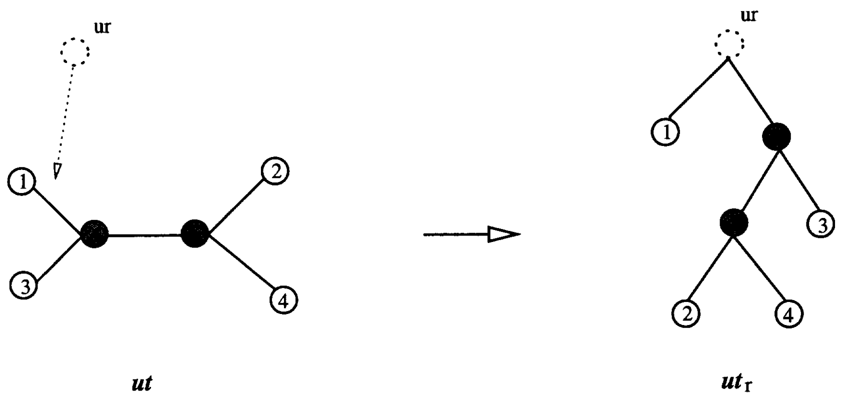


Figure 3.18 An unrooted tree and its rooted canonical form version.

3.5 Extension to Unrooted Tree Mining Problem

As mentioned in previous chapters, a few phylogeny reconstruction algorithms such as Most Parsimonious and Maximum Likelihood will produce unrooted trees. An unrooted tree has more freedom in its Newick representations. For example

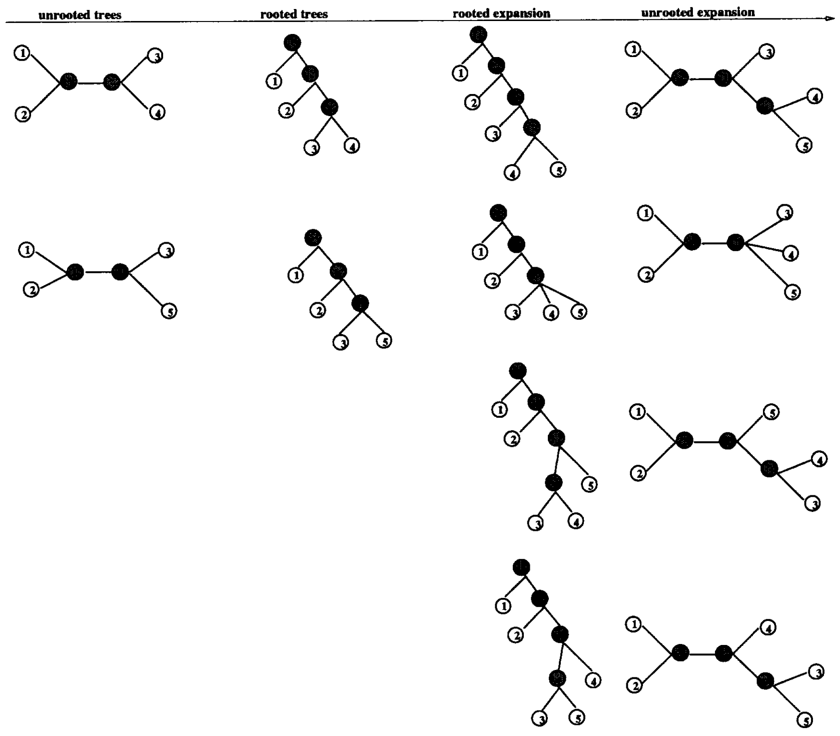


Figure 3.19 An example of case 1.1 unrooted tree joining under transformation.

"*((1,2),(3,4));*", "*(1,(2,(3,4)));*", "*(1,((3,4),2));*" and "*(2,(1,(3,4)));*" all represent the same unrooted tree. For this reason, it is generally believed that to mine frequent agreement subtrees in unrooted trees is more difficult than in their rooted counterparts. Surprisingly, a slightly modified version of Phylominer can be devised to efficiently solve the tree mining problem in unrooted trees. The modification is essentially a sandwiched **URRU** transformation, which stands for a transformation starting from k size [**U**nrooted trees] to [**R**ooted trees], after that, pairwise rooted trees joining are conducted to get $(k+1)$ size [**R**ooted trees], which are finally transformed back to [**U**nrooted trees] again.

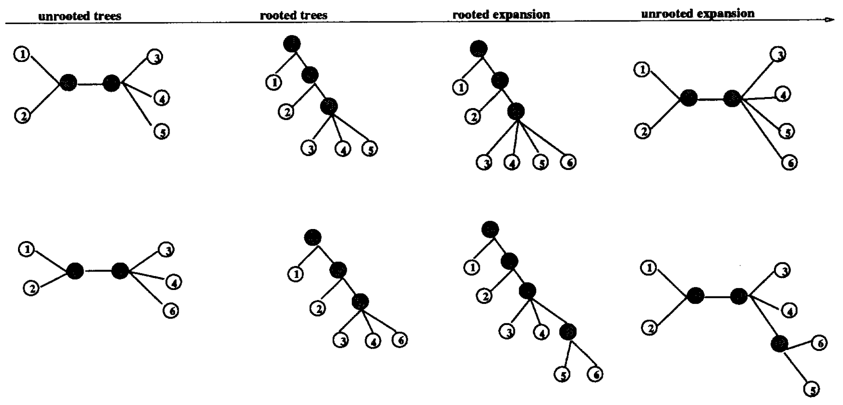


Figure 3.20 An example of case 1.2 unrooted tree joining under transformation.

Given an unrooted tree ut, a root can be inserted at the dangling edge which connects to the leaf of the smallest label of the tree to form a rooted tree ut_r of ut. Figure 3.18 shows an unrooted tree and the corresponding rooted tree in its canonical form. This transformation allows an unrooted tree to be treated as a rooted tree, therefore, two unrooted trees, after being rooted, can be processed using the joining procedure of rooted trees to produce ut_r^{k+1} candidate trees from two ut_r^k trees. Once a candidate ut_r^{k+1} is in hand, it will be reversed back to an unrooted tree ut^{k+1} by suppressing the root which is the node of valence 2. Figure 3.19 and Figure 3.20 show examples of joining two unrooted trees under the two sub-cases of case 1 joining respectively. Figure 3.21 and Figure 3.22 illustrate how to join two unrooted trees under rule 1 and rule 2 of case 2 respectively. In tandem with this modified candidate generation method, the partition metric is still employed to perform agreement subtree verification. This algorithm is called UPhylominer, namely for unrooted phylogenetic tree mining algorithm.

While in the rooted tree scenario, the initial set of frequent subtrees is composed of all 2-leaf subtrees, which are obtained through combinatorial enumeration instead of going through candidate generation-and-verification process; in unrooted tree scenario, the initial frequent subtree set consists of all 3-leaf subtrees. This is because a 3-leaf unrooted tree is a star tree, which is the only possible topology for any 3-leaf unrooted tree; thus, all 3-leaf trees can also be obtained through combinatorial enumeration without going through candidate generation-and-verification process. After all 3-leaf subtrees are rooted and normalized properly, UPhylominer will iterate mining loops progressively to get all frequent unrooted subtrees as Phylominer will do. At the final stage, all discovered trees will be converted back to unrooted trees. Figure 3.23 shows a running example of unordered subtree mining on the same set of trees used in Fig 3.5, but viewed as unrooted trees here.

The correctness of the UPhylominer can be similarly proved as that of Phylominer has been, and the time complexity remains the same.

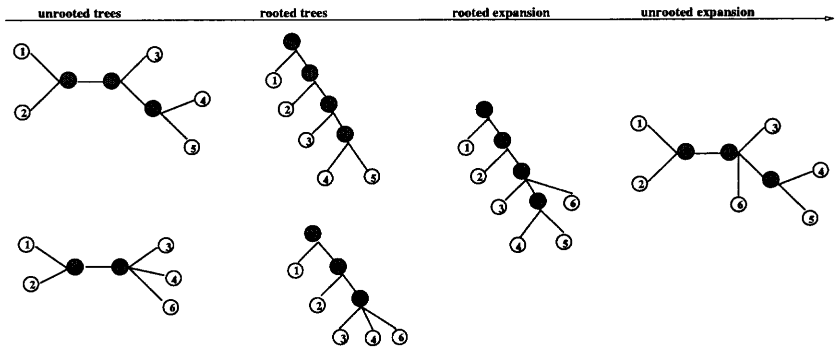


Figure 3.21 Examples of rule 1 of case 2 unrooted tree joining under transformation.

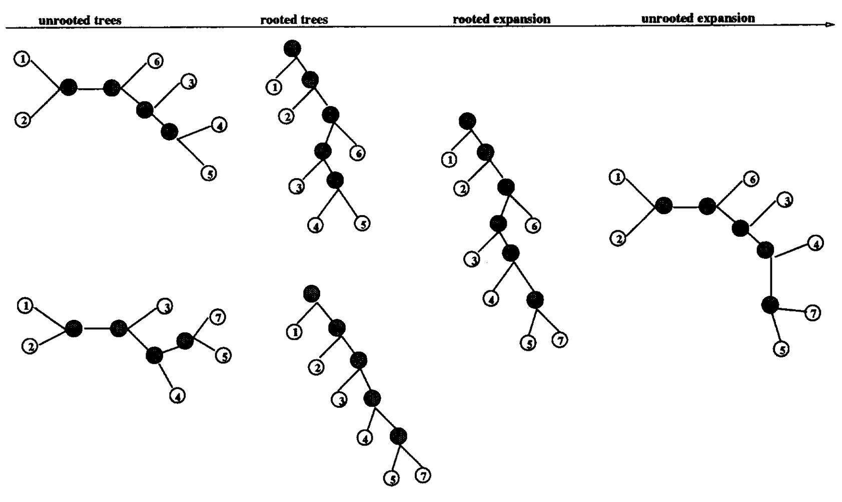


Figure 3.22 Examples of rule 2 of case 2 unrooted tree joining under transformation.

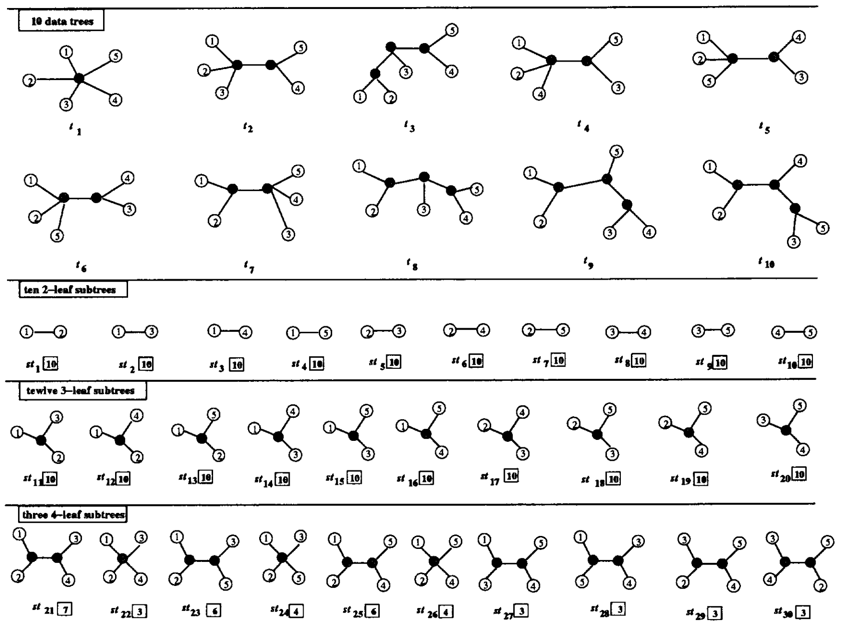


Figure 3.23 A running example of UPylominer on unrooted leaf-labeled trees.

3.6 Experiments

3.6.1 Synthetic Datasets

For correctness verification and performance evaluation purposes, a tree generator is implemented in C++ to generate synthetic datasets subject to user specified parameters. The basic idea behind the data generator is similar to, but more powerful than, the one used in COMPONENT. COMPONENT can generate binary trees only, while the generator developed in this work can generate trees of various degrees by generalizing the algorithm described in Holmes distance[26]. Table 3.3 lists the parameters and their default values, where *fanout* of a node is the number of children of a node.

Table 3.3 Synthetic Dataset Parameters

Notation	Parameter	Default setting
$\|DT\|$	The size of a dataset DT	600
$\|S\|$	The cardinality of a leaf label set	10
LF	The largest fanout	5
SF	The smallest fanout	2

3.6.2 Correctness of Implementation

The described algorithm has been fully implemented in C++. Although the correctness of the algorithm has been previously proved, it remains to be challenging to verify the correctness of the implementation of the algorithm. Given a particular input dataset, a successful implementation of the algorithm should discover correct number of frequent subtrees, correct support values for each individual subtree and correct supporting list of data trees containing the frequent subtree. To verify this implementation, the following test strategy is carefully designed. First, a tree t of n leaves is randomly generated by the tree generator. Then the tree t is duplicated for

m times to get the input dataset DT. The advantage of such test dataset is that the total number of frequent subtrees in the dataset can be easily calculated in advance even without conducting the real mining operation on the dataset. The reason is that every subtree of tree t will be a frequent subtree of DT and its support value will be 100% due to the fact that all data trees are the same. In fact, the total number of frequent subtrees is $T(n) = \sum_{1 \leq i \leq n} C_n^i$. For example, suppose the tree t is "(((1,3),4,(8,(2,5)),7),6);", where $n = 8$, then the total number of frequent subtrees should be $T(n) = C_8^1 + C_8^2 + C_8^3 + C_8^4 + C_8^5 + C_8^6 + C_8^7 + C_8^8 = 255$. To get a testing dataset of multiple trees, the t is duplicated for 100 times. The *minsup* can be set to an arbitrary value. After running on this dataset, the algorithm actually finds all 255 subtrees in their canonical forms, and also finds the support values of those subtrees to be exactly 100%. This mining result clearly confirms that the implementation is correct and the algorithm design is correct, because the results are consistent with the values obtained from the prior analysis, and these correct results can be obtained only when all underlying techniques are implemented correctly. A total of 187 different datasets have been tested to verify the correctness of the implementation, and all results obtained showed that the algorithm has been successfully implemented. It has also been conducted to manually check test result on those weakly controlled datasets. A weakly controlled dataset means that the exact mining results can not be calculated in advance solely based on theoretical analysis. In such case, each reported frequent subtree can be rigorously verified only by manually checking its presence in each data tree based on subtree isomorphism theory. When the accumulated support value for that subtree is obtained, the subtree can be easily detected to be frequent or not. Such kind of test is conducted on 231 different datasets, and all testing results are verified to be correct. Therefore, the implementation is robust and correct.

3.6.3 Performance Analysis

A series of experiments have been conducted to evaluate the performance of the proposed tree mining algorithm on synthetic data, run under the Solaris operating system on a SUN Ultra 60 workstation.

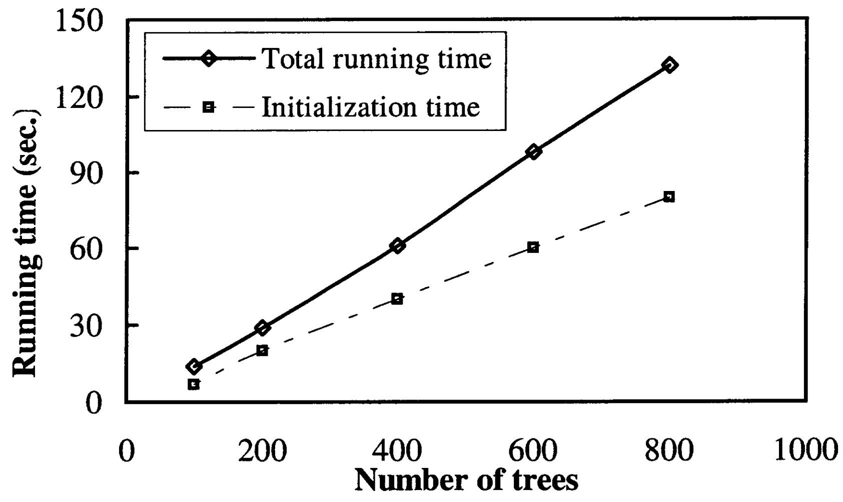


Figure 3.24 Scalability on the size of dataset.

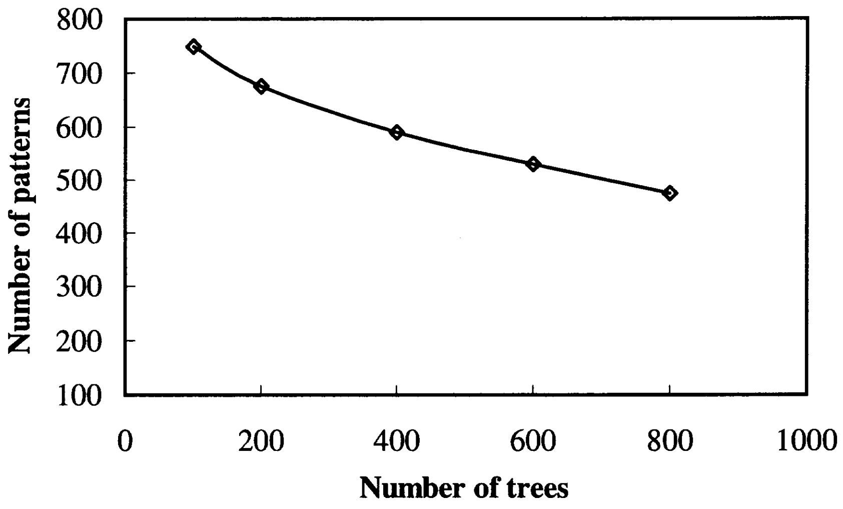


Figure 3.25 The size of dataset via Number of patterns.

Figure 3.24 shows how changing the *database size* of synthetic datasets affects the overall running time of the algorithm Phylominer. The *eight* datasets generated for this experiment contain different numbers of trees ranging from 100 to 800, while

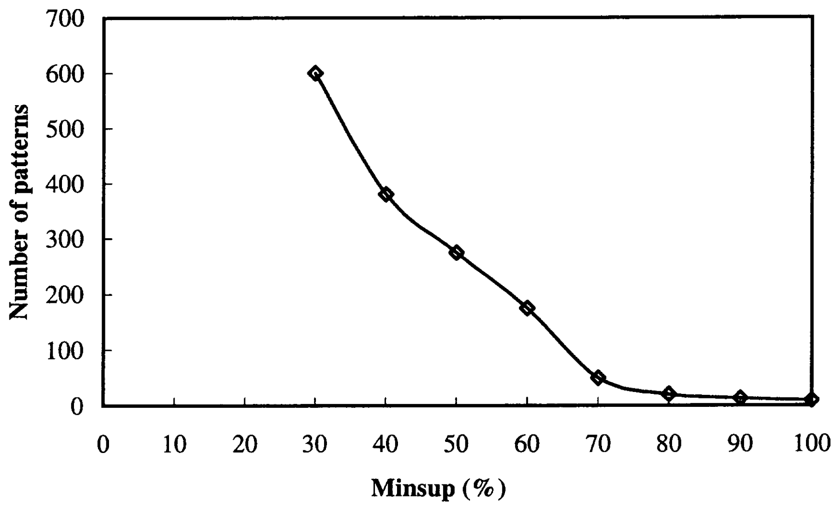


Figure 3.26 *minsup* vs. the number of discovered frequent agreement subtrees.

each tree has the same number of leaves of 15. The *minsup* value is set to 30%. Other parameters are default values shown in Table 3.3. As the figure shows, the total running time scales up linearly with respect to the sizes of datasets. This is because, the more trees a dataset contains, the more times the agreement verification will be conducted against the dataset. Another measurement indicated by the dashed line in the figure shows that the time spent on the initialization stage scales up with the growing of the dataset size as well. This is because the initialization step essentially comprises the following two operations. One is pattern enumeration, where the number of patterns is related to the tree size only, regardless how many trees a dataset contains as long as the Taxa set is the same. However, the more trees a dataset contains, the more time will be spent on preparing the supporting tree ID lists, and this is the exact reason why the initialization time still scales up linearly with the sizes of datasets.

Corresponding to the above overall running time performance, the numbers of patterns obtained from the same set of experiments are shown in Figure 3.25. The figure shows that, with the increasing numbers of trees of datasets on the same leaf label set, the numbers of patterns will decrease to a stable value. The reason is that, in general, the more randomly generated trees a dataset has, the less consensus information will be embodied in the dataset. This explains why the number of patterns declines with the increasing of dataset size. One the other hand, although the number of larger size frequent subtrees could drop dramatically to *zero* due to the increasing of *minsup* value; the initialization set will guarantee that the final mining result contains at least all 2-leaf subtrees, and the number of which is a fixed value. This explains why the numbers of qualified patterns finally reach a stable value.

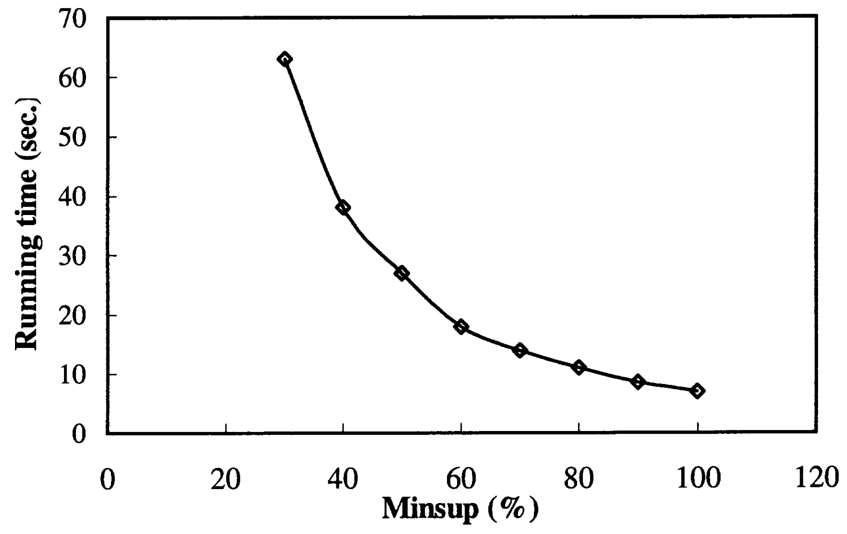


Figure 3.27 *minsup* vs. running time.

Figure 3.26 shows how changing of *minsup* affects the number of patterns discovered by the algorithm. The data used in this experiment contains 200 synthetic trees, with each tree having 15 leaves. The values of other parameters are shown in Table 3.3. It can be seen from the figure that as *minsup* increases, the number of qualified patterns drops quickly. This experimental result is well consistent with the following analysis. When *minsup* goes up, the number of qualified patterns at $k > 3$ level certainly drops. Consequently, the number of patterns in $k + 1$ size will drop in a non-linearly way. This effect will be cascadingly transferred from lower levels to higher levels. Finally, the total number of the qualified patterns will certainly drop. It can also be observed that once the *minsup* reaches a certain point, 0.8 in this case, the numbers of patterns reach a stable value. This is because the numbers of 2-leaf subtrees embedded in these data trees are always the same. As already mentioned, forming the initial set of the mining algorithm, these 2-leaf subtrees will appear in all mining results regardless what the *minsup* values will be, because their support values are always 100%.

Figure 3.27 shows how changing of *minsup* affects the running time of Phylominer on the same dataset as the previous experiment. This performance figure shows that as *minsup* increases, the running time of Phylominer drops quickly. This can be explained by the fact that the number of discovered patterns actually decreases with the increasing of *minsup*. Consequently, less valid pairwise joinings in each equivalence classes will be conducted. As a mutual result of the above two factors, the overall running time drops quickly.

Figure 3.28 shows the distribution of numbers of patterns of two datasets under the *minsup* values of 5% and 25% respectively. The first dataset used in this experiment contains 100 randomly generated trees on 13 leaves. When the *minsup* is set to 5%, there are a total of 7261 patterns to be discovered, among them, there are 78 2-leaf subtrees, 1144 3-leaf subtrees, 5939 4-leaf subtrees and 100 5-leaf subtrees. When the *minsup* is raised to 25%, the total number of patterns will quickly drop to 782, among them are only 78 2-leaf subtrees and 604 3-leaf subtrees. The second dataset is a special testing dataset composed of 100 copies of a same 13-leaf tree, which is also one of the datasets used in the verification experiment for the correctness of the implementation. The distribution of the mining result is shown by the dashed line with triangle markers in the figure. The number of patterns with respect to all different sizes of subtrees gradually increases first and then gradually drops to a small value again. Obviously, the distribution strictly follows the combinatorial mathematical calculation on the power subsets of a set. Adding up all these numbers, a total of 6634 patterns are obtained. Another interesting observation is that the mining result on the second dataset is invariant to different *minsup* values. Therefore only the result under the *minsup* value of 25% is reported here, given the same result will be obtained for all the other valid *minsup* values.

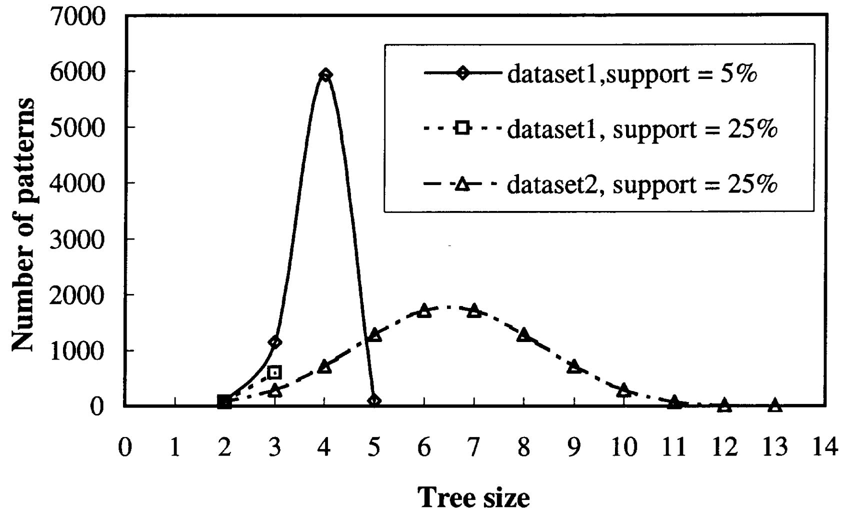


Figure 3.28 The number of frequent subtrees vs. tree size.

Table 3.4 compares the mining results of the same dataset when being viewed as rooted trees against being viewed as unrooted trees. The dataset is composed of 30 trees of 15 leaves, and the tests are conducted for *minsup* values of 30% and 50% respectively. The comparison shown in this table is interesting. Theoretically speaking, for the same set of k labels, there are approximately $k-1$ more times rooted trees than unrooted trees. Therefore, there seems to exist a higher chance for more candidate trees to be frequent when data trees are deemed rooted than are deemed unrooted. This intuition is clearly supported by the numbers of 3-leaf patterns under the *minsup* of 30%. It is, however, generally not supported by the overall comparison. The surprising fact is that in most cases, less frequent subtrees will be discovered when data trees are viewed as rooted trees than are viewed as unrooted trees. This is because when the total number of data trees is fixed, the more possible candidate trees there are, the lower possibility there is for each candidate tree to be frequent. This can be seen from 3 leaves trees. Given any 3-leaf unrooted tree, the only topology is a star tree; but if deemed as a rooted tree, there could exist 4 possible topologies, diluting the chance for each possible topology to be frequent. That is why the numbers of unrooted frequent subtrees are larger than those of rooted subtrees in general.

Table 3.4 Comparison of Mining Results between Rooted Trees and Unrooted Trees under *minsup* Value of 30%

minsup	30%		50%	
pattern size	rooted	unrooted	rooted	unrooted
1	15	15	15	15
2	105	105	105	105
3	747	455	32	455
4	4	15	0	53

3.6.4 Datasets from TreeBASE and COMPONENT

The Phylominer algorithm has been applied to several datasets shipped together with COMPONENT. Experiment results on "epi216.nex" and "peg.nex" are reported here.

File "epi216.nex" is a simple dataset consisting of *three* 10-leaf trees. Table 3.5 shows

Table 3.5 Data Mining Result on epi216.nex

Support	Time	Total Pattern Number	MAST Number	MAST Size
100%	34	380	3	7
60%	63	796	8	8

the experimental mining results under two different *minsup* settings, 60% and 100%,

Table 3.6 Data Mining Result on peg.nex

Support	Time	Total Pattern Number	MAST Number	MAST Size
100%	34	251	3	7
89%	35	251	3	7
77%	35	251	3	7
66%	63	285	1	7
55%	70	285	1	8
44%	70	285	1	8
33%	150	480	4	8

respectively. When *minsup* is set to 100%, a total of 380 frequent subtrees will be discovered. Among the discovered subtrees, there are *three* largest subtrees, and each of them has 7 leaves. Obviously, they are exactly the MASTs in the classical agreement subtree analysis problem. When *minsup* is set to 70%, there are a total of 796 subtrees frequently occurring in the dataset. Out of these subtrees, there are *eight* largest frequent subtrees, and the size of the largest frequent subtrees is also *eight*.

File "peg.nex" is a dataset consisting of 9 different trees for 11 species, the species '1' is deleted in this experiment, because position of leaf '1' is invariant to other leaves in these 9 trees. Thus the actual dataset contains *nine* trees for 10 species instead. *Nine* trees are identified as from t_1 to t_9. Different support values have been used to get a series of experiment results, which are reported in Table 3.6. Particularly when *minsup* is set to 33%, Phylominer found a total of 480 frequent subtrees, among them the *four* largest subtrees with *eight* leaves are "(((2,3),(5)),(6,((7,9),8)))", "(((2,3),(5)),(6,((7,10),8)))", "(((2,3),(5)),(6,(8,(9,10))))" and "(((2,3),(5)),((7,(9,10)),8))". Among these 4 largest frequent subtrees, the first three all appear in *three* data trees of t_1, t_2 and t_5, while the last one appears in *six* trees of t_1, t_2, t_3, t_4, t_6 and t_7 out of *nine* data trees. As a contrast, when support goes to 100%, a total of 251 subtrees will be discovered, among them *three* largest subtrees of *seven* leaves are "(((2,3),(5)),((7,9),8))", "(((2,3),(5)),((7,10),8))" and "(((2,3),(5)),(8,(9,10)))".

Table 3.7 Data Mining Result on *five* Trees of Study-497 in TreeBASE

Support	Time	Total pattern number	MAST number	Size MAST
100%	30	30	9	3
80%	30	30	9	3
60%	40	55	3	5
50%	63	91	9	5

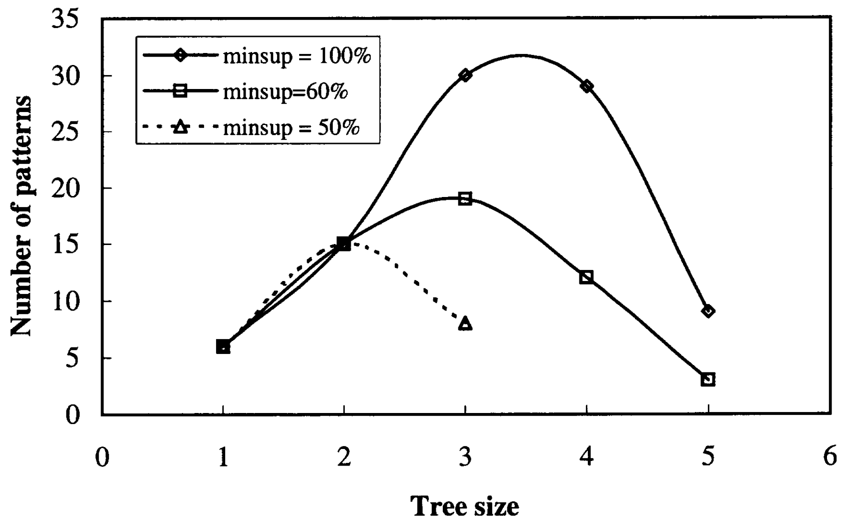


Figure 3.29 The distribution of numbers of frequent subtrees vs. sizes of subtrees.

Finally, the algorithm is applied to the dataset shown in Figure 3.1. The result is shown in Table 3.7. From this table, it can be seen that with the *minsup* decreasing, the running time goes up, and the total number of interesting patterns goes up as well. The largest frequent subtrees have 3 leaves only when *minsup* is set to 100%, while the largest frequent subtrees have 5 leaves when *minsup* decreases to 50%. The distribution of frequent subtrees are shown in Figure 3.29. In the figure, distributions associated with *minsup* values of 100% and 80% are completey overlapped, this is because out of *five* data trees, t_1 and t_5 are exactly the same. The overall distributions associated with the *three* different *minsup* values are consistent with experiment result reflected in Figure 3.28.

Experiments on these real datasets exhibit that the algorithm can systematically discover all interesting patterns in data trees. These frequent agreement trees help users to explore more consensus information that could not have been discovered by the traditional MAST algorithms. Furthermore, the algorithm can find all support values of those patterns and which data trees contain those subtrees.

3.6.5 Discussion

The Phylominer has been designed with the least restrictions. In addition to the *minsup* parameter, a *scutoff* parameter specifies a maximum size of frequent subtrees, which will stop the execution of the program when the discovered patterns in the last iteration reach the given size; users can also set a *maxpn* parameter which will stop the program when the number of discovered pattern reaches the given maximum pattern number. Furthermore, the algorithm has no bounding restriction on degree values of tree nodes.

Interesting enough, when the input parameters are restricted to some special settings, the results of Phylominer conform to that of some traditional methods. For example, when the *minsup* value is set to 100%, the subtrees of the largest size will be all MAST trees. Thus this setting allows the algorithm to be used to verify the correctness of any other MAST algorithms' implementations. When the *scutoff* is set to 4 in unrooted tree mining, the result will be all frequent quartets. When the *scutoff* value is set to 3 in rooted tree mining, the result will be all frequent triplets.

The traditional MAST problem considers a profile of trees which are built upon exactly the same label set. For example, if the label set is $\{1, 2, 3, 4\}$, the size of every tree in the profile has to be of *four*, and the leaves of those trees must be labeled as '1', '2', '3' and '4' respectively. This restriction can actually be removed in the (U)Phylominer algorithms. In the rooted version of the algorithm described above, the initial set of 2-leaf frequent subtrees is enumerated by a brute force manner, since all trees share the leaf label set; while when the trees in a dataset do not share exactly the same leaf label set, the brute force manner can be replaced by finding out both all 1-leaf subtrees and all 2-leaf subtrees through an inverted list technique. Similar logic applies to unrooted tree version. In the original unrooted version, all data trees share exactly the same set of labels, thus, all 3-leaf star trees are automatically enumerated to compose the initial set; while in a set of trees which have only overlapped leaf labels, a 3-leaf star tree is not necessarily to be frequent. However, all frequent 1-leaf subtrees, 2-leaf subtrees and 3-leaf subtrees can be easily obtained through an inverted list intersection technique. Therefore, the method discussed in this work can process a profile of trees which have more freedom in their leaf label set formations. This extension is envisioned to be useful in those emerging super-tree applications.

3.7 An Online Mining Engine

An online mining engine has been developed, which allows remote users to interact with the core mining algorithm via Internet. Figure 3.30 shows the engine architecture. The system is composed of four components: Web-based Interface, Input Processor,

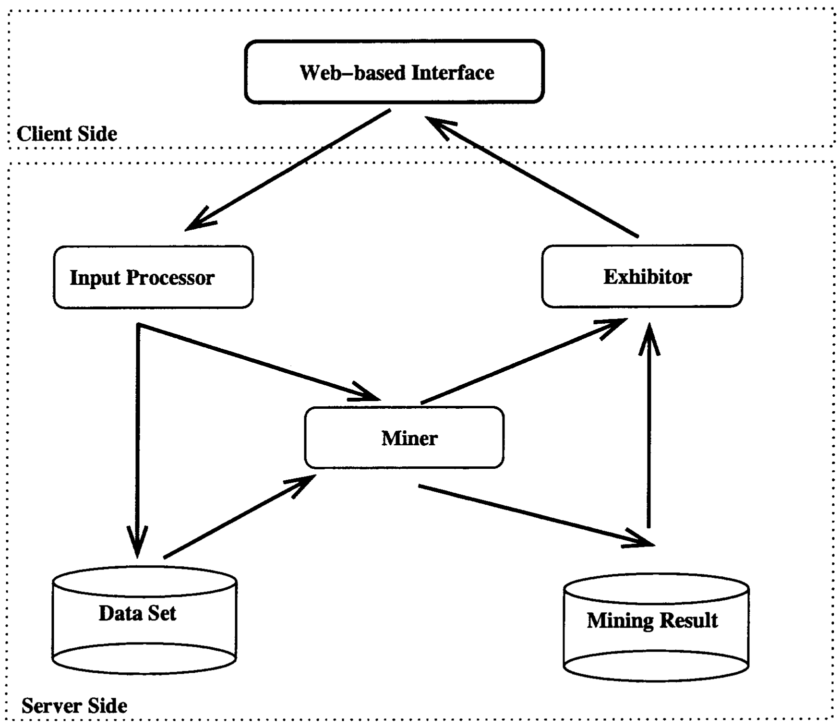
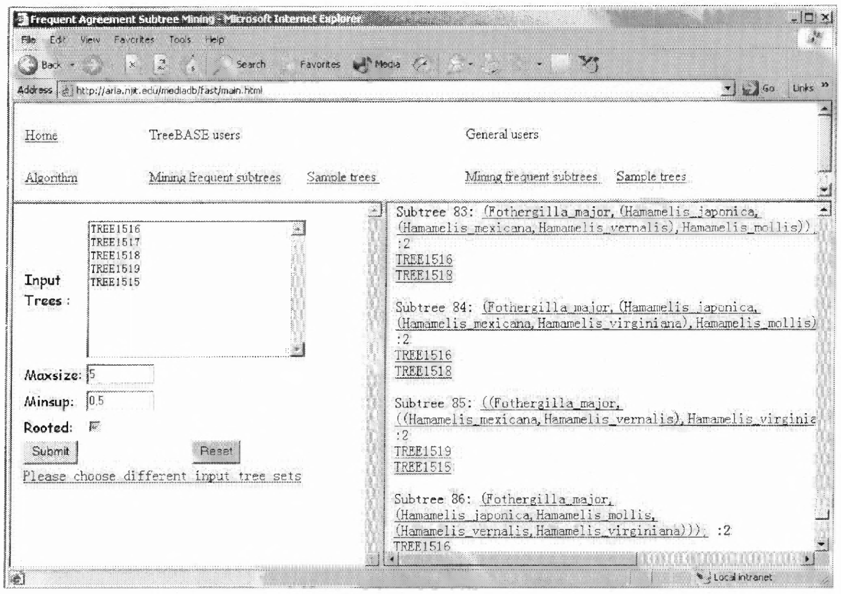


Figure 3.30 The architecture of the mining engine.

Miner and Exhibitor. Web-based Interface, implemented using HTML, allows users to input data, to set up mining parameters, and to submit the mining requests to the server. On the server side, Input Processor, a module implemented in Perl CGI script, will upload input data together with mining parameters to the web server, and then activate the core Miner module. Miner module is a C++ implementation of Phylominer algorithm, which conducts the real mining operation and outputs the mining results in XML files. Once the Miner completes its output serialization, it surrenders the control to Exhibitor module by providing an XML link and a HTML page via XSL transformation. Through the Exhibition module, users are able to browse XML formatted result using XML-aware browsers or simply to examine the HTML page transformed from XML using any traditional HTML browser.

Figure 3.31 illustrates the system's working interface. The top frame shows the main menu of the system; the lower left frame shows an user input dataset with a


Frequent Agreement Subtree Mining - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://aria.njit.edu/mediadb/fast/main.html Go Links
Local intranet

Home TreeBASE users General users

Algorithm Mining frequent subtrees Sample trees Mining frequent subtrees Sample trees

Input Trees :
TREE1516
TREE1517
TREE1518
TREE1519
TREE1515

Maxsize: 5

Minsup: 0.5

Rooted: ✓

Submit Reset

Please choose different input tree sets

```
Subtree 83: (Fothergilla_major, (Hamamelis_japonica,
(Hamamelis_mexicana, Hamamelis_vernalis), Hamamelis_mollis)),
:2
TREE1516
TREE1518

Subtree 84: (Fothergilla_major, (Hamamelis_japonica,
(Hamamelis_mexicana, Hamamelis_virginiana), Hamamelis_mollis)
:2
TREE1516
TREE1518

Subtree 85: ((Fothergilla_major,
((Hamamelis_mexicana, Hamamelis_vernalis), Hamamelis_virginia
:2
TREE1519
TREE1515

Subtree 86: (Fothergilla_major,
(Hamamelis_japonica, Hamamelis_mollis,
(Hamamelis_vernalis, Hamamelis_virginiana))), :2
TREE1516
```

Figure 3.31 The main interface of mining engine.

setting of mining parameters; and the right frame shows the mining result. When clicking on any subtree link in the right frame, the user will be redirected to a separate window where the subtree is shown in a Java applet (the upper window of Figure 3.32). When the user click on the link associated with a data tree which supports a subtree, she or he will be shown the data tree within which the leaves of the subtree are highlighted in red color and decorated by red bullets (the lower window of Figure 3.32). If the user is interested in XML data, he or she can click on the XML output link to get all mining result in a well organized nested format (the Figure 3.33). The meanings of the tags and data in the XML are much self-explanatory.

3.8 Summarization

In this chapter, a novel tree mining algorithm, Phylominer, has been presented, to efficiently discover frequently occurring agreement subtrees in a given phylogeny dataset. Phylominer algorithm is immediately useful for scientists in phylogenetic research discipline, and it can also be used by linguistics researchers and other domain

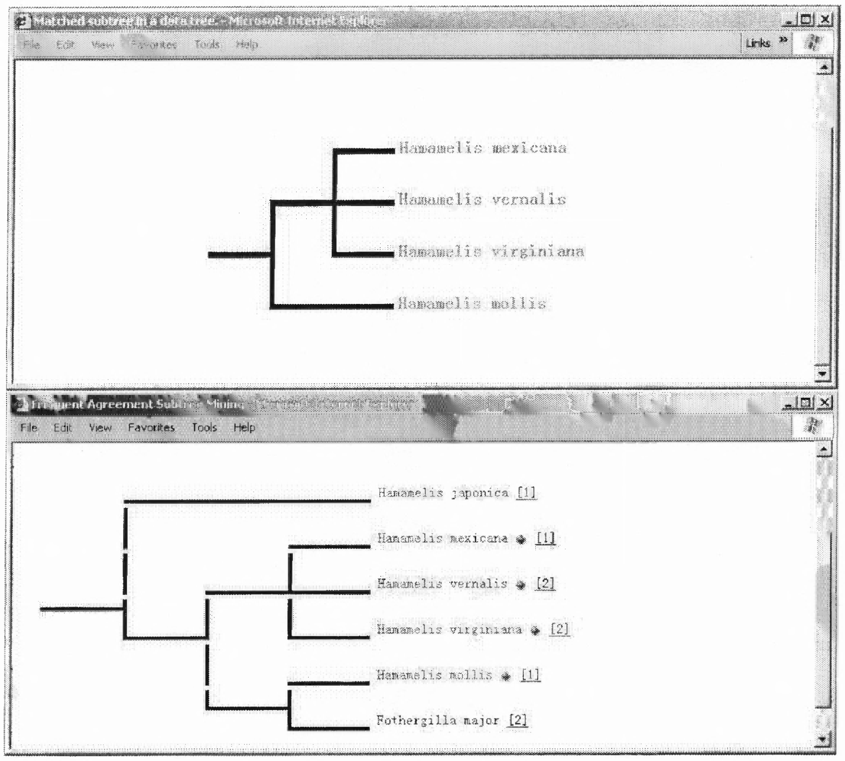
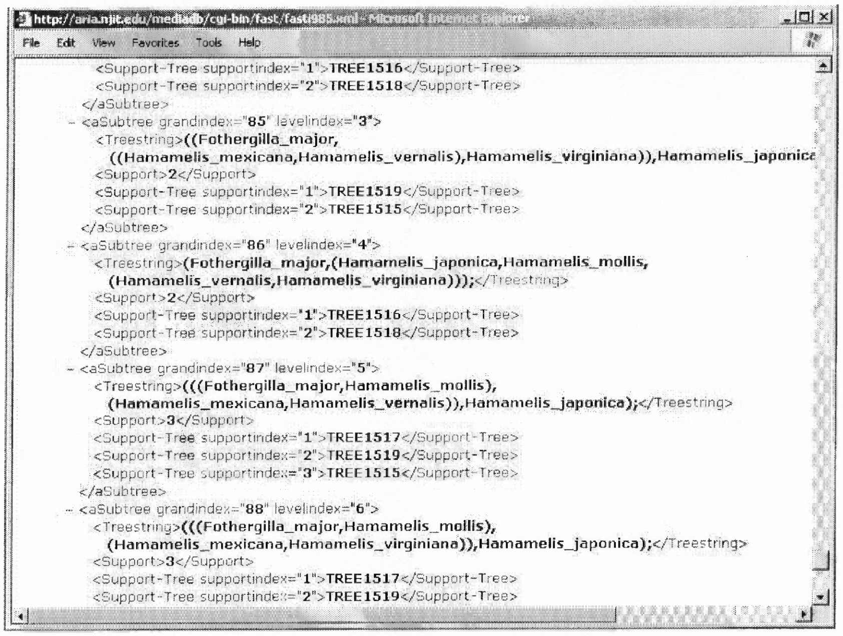


Figure 3.32 A screenshot showing a pattern tree and its presence in a data tree.

practitioners as long as the interested data in their domains can be modeled as leaf-labeled trees.

The details of the algorithm have been discussed, and the time complexity and correctness of the algorithm have been analyzed. At this stage, a rudimentary version of the algorithm has been implemented. Based on this version, Phylominer algorithm is evaluated on both synthetic datasets and real world phylogenetic trees. The experiments show that Phylominer can scale well to large amount of simulated data profile, and the mining results in the real world data are interesting and informative.

In the future, several relevant algorithms such as vertical mining algorithm, parallel algorithm and approximation algorithm will be deeply researched. The tree expansion proposed in this work is based on joining operation; however, the author realized that tree expansion through righmost growing enumeration could be another


http://aria.njit.edu/mediadb/cgi-bin/fast/fast985.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help

```
      <Support-Tree supportindex="1">TREE1516</Support-Tree>
      <Support-Tree supportindex="2">TREE1518</Support-Tree>
    </aSubtree>
  - <aSubtree grandindex="85" levelindex="3">
      <Treestring>((Fothergilla_major,
        ((Hamamelis_mexicana,Hamamelis_vernalis),Hamamelis_virginiana)),Hamamelis_japonica
      <Support>2</Support>
      <Support-Tree supportindex="1">TREE1519</Support-Tree>
      <Support-Tree supportindex="2">TREE1515</Support-Tree>
    </aSubtree>
  - <aSubtree grandindex="86" levelindex="4">
      <Treestring>(Fothergilla_major,(Hamamelis_japonica,Hamamelis_mollis,
        (Hamamelis_vernalis,Hamamelis_virginiana)));</Treestring>
      <Support>2</Support>
      <Support-Tree supportindex="1">TREE1516</Support-Tree>
      <Support-Tree supportindex="2">TREE1518</Support-Tree>
    </aSubtree>
  - <aSubtree grandindex="87" levelindex="5">
      <Treestring>(((Fothergilla_major,Hamamelis_mollis),
        (Hamamelis_mexicana,Hamamelis_vernalis)),Hamamelis_japonica);</Treestring>
      <Support>3</Support>
      <Support-Tree supportindex="1">TREE1517</Support-Tree>
      <Support-Tree supportindex="2">TREE1519</Support-Tree>
      <Support-Tree supportindex="3">TREE1515</Support-Tree>
    </aSubtree>
  - <aSubtree grandindex="88" levelindex="6">
      <Treestring>(((Fothergilla_major,Hamamelis_mollis),
        (Hamamelis_mexicana,Hamamelis_virginiana)),Hamamelis_japonica);</Treestring>
      <Support>3</Support>
      <Support-Tree supportindex="1">TREE1517</Support-Tree>
      <Support-Tree supportindex="2">TREE1519</Support-Tree>
```

Figure 3.33 A piece of XML showing the mining result.

viable approach that needs further exploration. It would also be interesting to build upon this work further researches such as performing pattern based classification[37] and developing super-tree reconstruction algorithms[49].

CHAPTER 4

IMPLEMENTATION

This chapter reports two online mining engines wrapping the two algorithms presented in the previous chapters - Cousin Miner, which works on a set of sample TreeBASE trees, available at “http://aria.njit.edu/ mediadb/cousin”, and frequent agreement subtree online mining engine, which is available at “http://aria.njit.edu/ mediadb/fast”.

4.1 Cousin Miner

4.1.1 Introduction

Finding cousin pairs with various distances from a set of unordered trees is a novel approach to phylogenetic data mining. This project is conducted at New Jersey Institute of Technology, New York University and University of Western Ontario, Canada. The project aims to produce algorithms, data structures, and tools that allow approximate search and mining across general trees (with applications to phylogeny). The underlying algorithms are based on Cousins.k. A system has been built to allow you to find interesting cousin pairs in phylogenies obtained from TreeBASE. Click on the "Mining" button in the menu to try the system. Click on the "Instruction" button for guidelines of using the system.

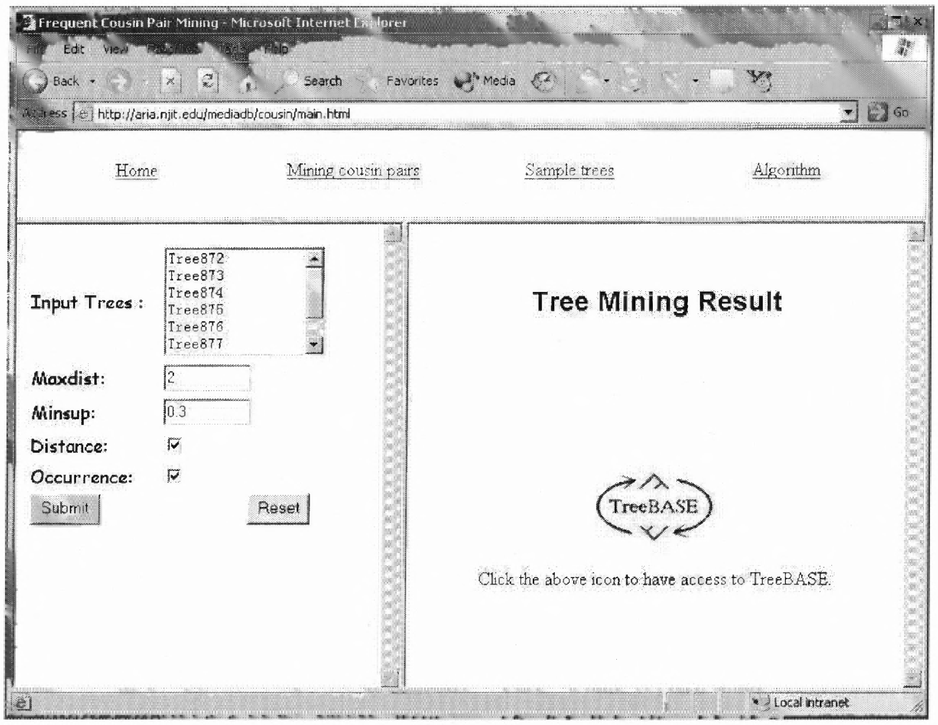
4.1.2 Instruction

This is a prototype system, run on a Solaris Sun workstation, that performs phylogenetic data mining on a set of 26 sample trees obtained from TreeBASE. Due to resource limitations, this online demo allows a user to work on at most *eight* trees each time and the system will find interesting cousin pairs from these input trees.

The user can also download the software from "http://www.cs.nyu.edu/cs/faculty/shasha/papers/cousins.html" and perform data mining locally on a larger dataset.

Input.

Two types of interfaces have been designed for users to input data trees.


Frequent Cousin Pair Mining - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://aria.njit.edu/mediadb/cousin/main.html
Go
Home
Mining cousin pairs
Sample trees
Algorithm
Submit
Reset
TreeBASE
Local intranet

Input Trees : Tree872, Tree873, Tree874, Tree875, Tree876, Tree877

Maxdist: 2

Minsup: 0.3

Distance: ☑

Occurrence: ☑

Tree Mining Result

Click the above icon to have access to TreeBASE.

Figure 4.1 The main interface of cousin pattern mining engine.

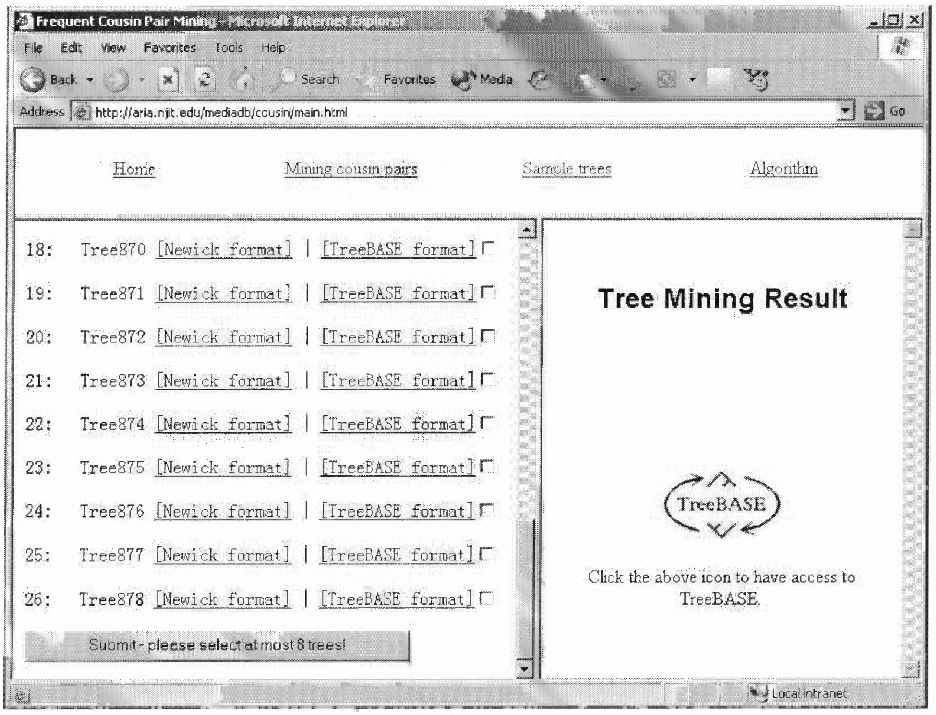
Interface 1 (main interface).

In the main interface shown in Figure 4.1, you can submit a request in *three* steps:

- Type in tree IDs as described in TreeBASE in the text window (these trees must be from the 26 sample trees).
- Provide appropriate parameter values Maxdist and Minsup, and indicate whether "Distance" and "Occurrence" will be considered in the execution.
- Press the "Submit" button.

Interface 2.

Users can also input data trees through the page of sample trees by clicking on "Sample trees" button in the main menu or click "Please choose different input tree sets" hyperlink in the previous interface 1. Figure 4.2 is a screenshot of interface 2.


Frequent Cousin Pair Mining - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://aria.njit.edu/mediadb/cousin/main.html Go
Home
Mining cousin pairs
Sample trees
Algorithm
Submit - please select at most 8 trees!
Tree Mining Result
TreeBASE
Local intranet

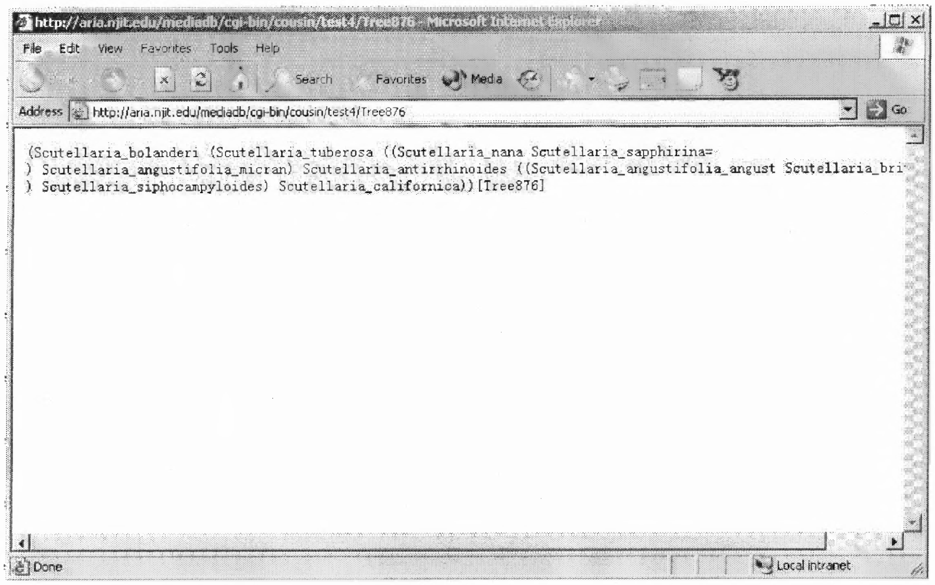
18: Tree870 [Newick format] | [TreeBASE format]
19: Tree871 [Newick format] | [TreeBASE format]
20: Tree872 [Newick format] | [TreeBASE format]
21: Tree873 [Newick format] | [TreeBASE format]
22: Tree874 [Newick format] | [TreeBASE format]
23: Tree875 [Newick format] | [TreeBASE format]
24: Tree876 [Newick format] | [TreeBASE format]
25: Tree877 [Newick format] | [TreeBASE format]
26: Tree878 [Newick format] | [TreeBASE format]

Click the above icon to have access to TreeBASE.

Figure 4.2 A screenshot of sample TreeBASE trees.

Click [Newick format] to view the parenthesized tree format of a phylogenetic tree. Figure 4.3 is a screenshot of the Newick format of tree876. Click [TreeBASE format] to view the corresponding graphic representation of the tree, shown in Figure 4.4.

You are allowed to select at most *eight* trees as input trees, which will appear in the main interface once selected. If the number of selected trees is 0, then the default tree IDs in the main interface will be the input trees. If the number of selected trees is larger than 8, then only the first 8 trees are taken as input trees.



Address http://aria.njit.edu/mediadb/cgi-bin/cousin/test4/Tree876

```
(Scutellaria_bolanderi (Scutellaria_tuberosa ((Scutellaria_nana Scutellaria_sapphirina=
) Scutellaria_angustifolia_micran) Scutellaria_antirrhinoides ((Scutellaria_angustifolia_angust Scutellaria_bri
) Scutellaria_siphocampyloides) Scutellaria_californica))[Tree876]
```

Figure 4.3 A screenshot of the Newick format of tree876.

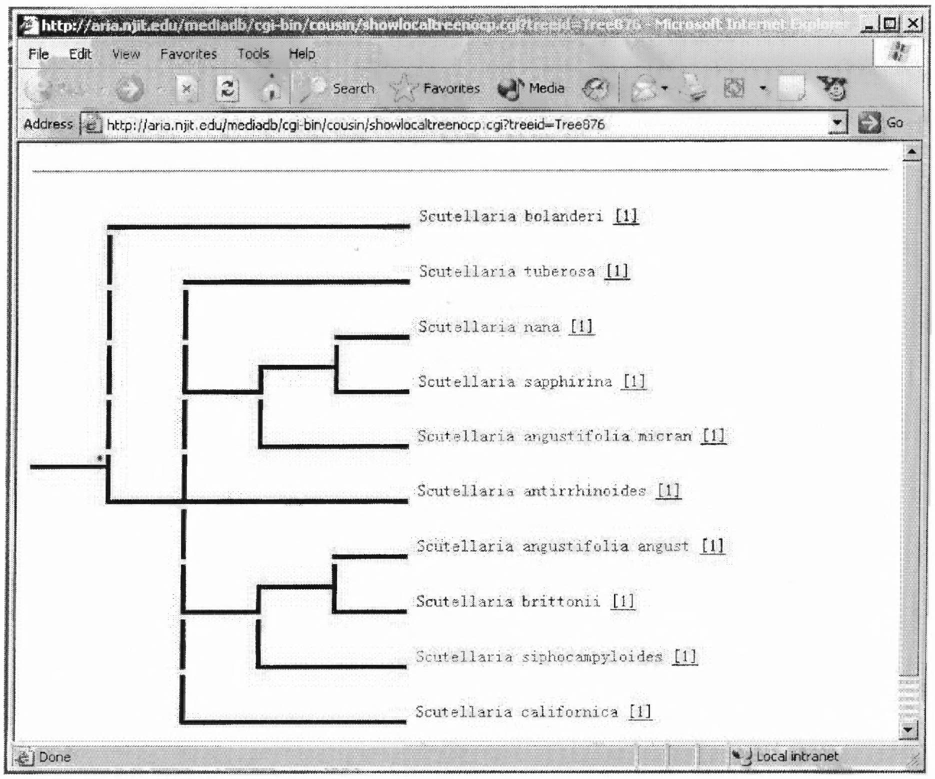
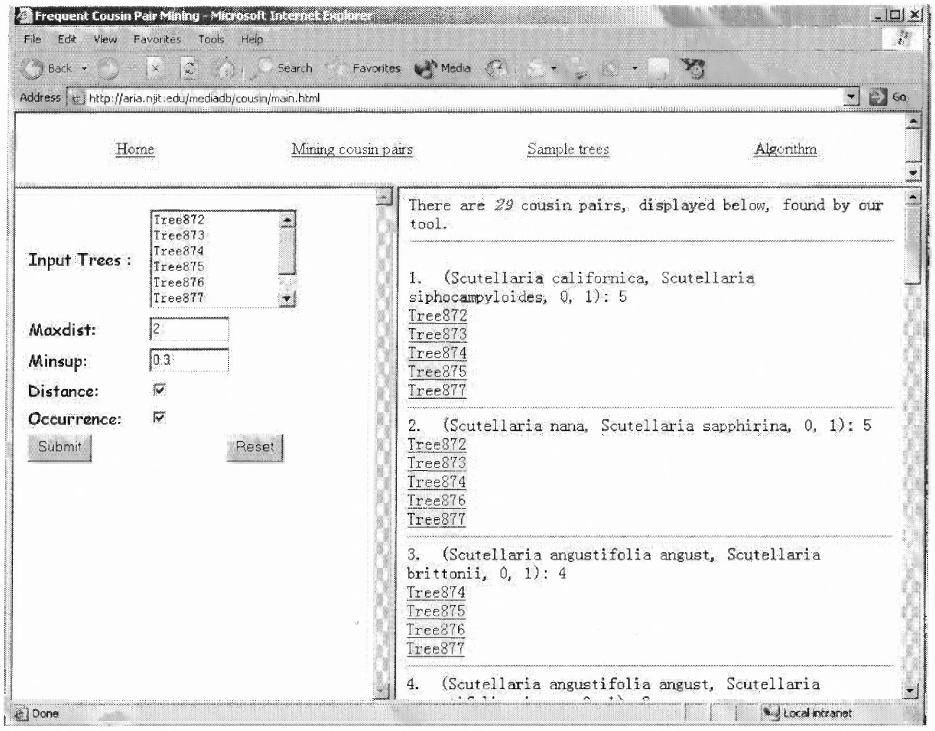

http://aria.njit.edu/mediadb/cgi-bin/cousin/showlocaltreenocp.cgi?treeid=Tree876
Scutellaria bolanderi [1]
Scutellaria tuberosa [1]
Scutellaria nana [1]
Scutellaria sapphirina [1]
Scutellaria angustifolia micran [1]
Scutellaria antirrhinoides [1]
Scutellaria angustifolia angust [1]
Scutellaria brittonii [1]
Scutellaria siphocampyloides [1]
Scutellaria californica [1]

Figure 4.4 A screenshot of TreeBASE graphic representation of tree876.

View the tree mining result.

The result consists of all cousin pairs satisfying the user-specified parameter values. Each cousin pair is followed by a number, which is the total number of input trees containing the cousin pair, and the IDs of these trees are also displayed.

When clicking on a tree id, the user will see the graphic representation of the tree. The cousin pair will be highlighted in red color, with red bullets. Figure 4.5


Frequent Cousin Pair Mining - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://aria.njit.edu/mediadb/cousin/main.html
Go
Home
Mining cousin pairs
Sample trees
Algorithm
Submit
Reset
Done
Local intranet

Input Trees : Tree872 Tree873 Tree874 Tree875 Tree876 Tree877

Maxdist: 2

Minsup: 0.3

Distance: ✓

Occurrence: ✓

There are 29 cousin pairs, displayed below, found by our tool.

1. (Scutellaria californica, Scutellaria siphocampyloides, 0, 1): 5
Tree872
Tree873
Tree874
Tree875
Tree877

2. (Scutellaria nana, Scutellaria sapphirina, 0, 1): 5
Tree872
Tree873
Tree874
Tree876
Tree877

3. (Scutellaria angustifolia angust, Scutellaria brittonii, 0, 1): 4
Tree874
Tree875
Tree876
Tree877

4. (Scutellaria angustifolia angust, Scutellaria

Figure 4.5 A screenshot of a list of mining result.

is a screenshot of a list of mining result. Figure 4.6 is a screenshot showing how a cousin pair is highlighted in a data tree.

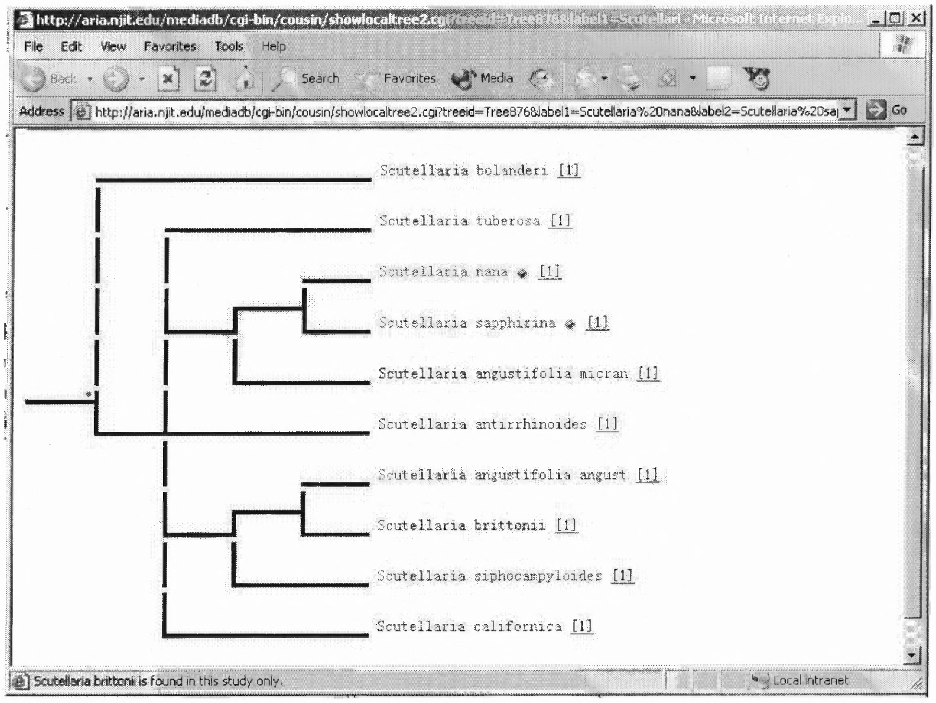

File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://aria.njit.edu/mediadb/cgi-bin/cousin/showlocaltree2.cgi?treeid=Tree876&label1=Scutellaria%20nana&label2=Scutellaria%20sa
Go
Scutellaria bolanderi [1]
Scutellaria tuberosa [1]
Scutellaria nana [1]
Scutellaria sapphirina [1]
Scutellaria angustifolia micran [1]
Scutellaria antirrhinoides [1]
Scutellaria angustifolia angust [1]
Scutellaria brittonii [1]
Scutellaria siphocampyloides [1]
Scutellaria californica [1]
Scutellaria brittonii is found in this study only.
Local intranet

Figure 4.6 A screenshot showing a highlighted cousin pair in a tree.

4.2 Agreement Subtree Miner

4.2.1 Introduction

Discovering frequent agreement subtrees of various sizes from a set of phylogentics trees is a novel approach to mine phylogenetic data. This project is conducted at New Jersey Institute of Technology, New York University and University of Western Ontario, Canada. The project aims to produce algorithms, data structures, and tools that allow data mining across multiple phylogenetic trees.

A system has been built to allow you to find interesting agreement subtrees in phylogenies obtained from TreeBASE and elsewhere. Click on the "Mining" button in the menu to try the system. Click on the "Instruction" button for guidelines of using the system.

Internet Explore 6.0 or above are recommended to browse mining results, because XSL functions used in this website are not supported by earlier versions of Internet Explore.

4.2.2 Instruction

This is a prototype system running on a Solaris Sun workstation, which performs phylogenetic data mining to find interesting frequent agreement subtrees from input trees. The system accepts two types of inputs. The first type of input data uses a subset of tree IDs from a pool of 17 sample trees obtained from TreeBASE, and users familiar with this database can test the system using this input method. Due to resource limitations, this TreeBASE online demo allows a user to work on at most *eight* trees each time. The second type of input data consists of trees with the same taxa in the newick format. The second type of input data is designed for general users. Again due to resource limitations, the maximum number of trees that can be tested is 500, and the size of trees has to be less than 12.

Input data format.

Our system accepts trees in newick format, which is a standard format for representing phylogenetic trees. Both string newick format and numeric newick format are acceptable.

1. String newick format.

A simple data in string newick format is shown below.

```
((Hamamelis_mexicana, (Hamamelis_vernalis, Hamamelis_virginiana)),
Hamamelis_mollis);
```

2. Numeric newick format.

A simple tree in numeric newick format is shown below, where taxa are represented by numbers from l_1 to l_n, where l_1, l_2, ... , l_n are n natural numbers representing n Taxa.

```
(1,2,(3,5),(7,(4,6)));
```

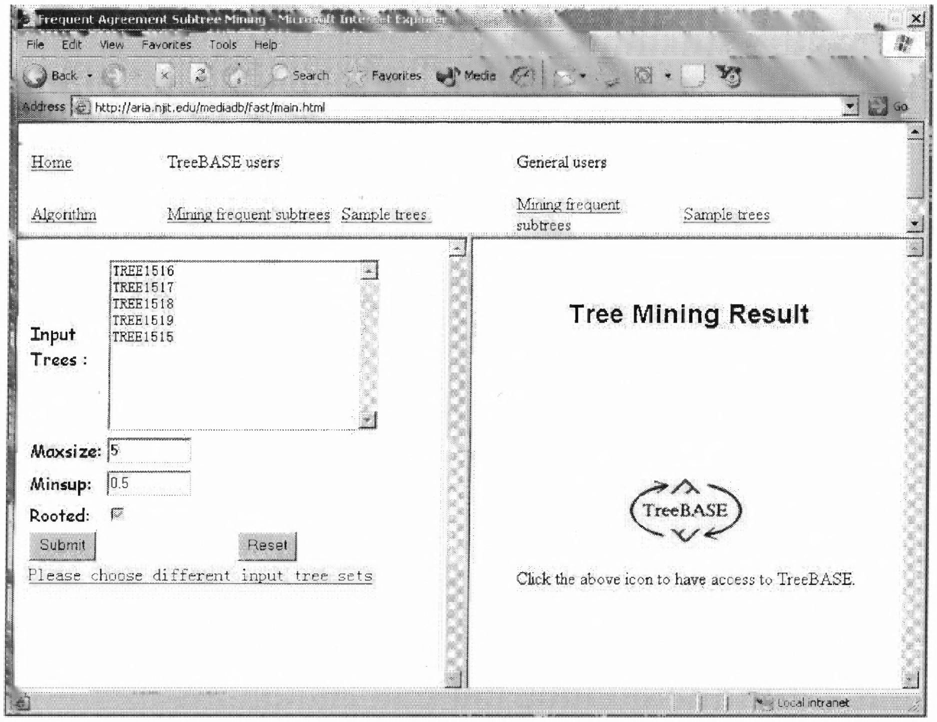
Output data format.

The output consists of an array of frequent subtrees in newick format, with each of them associated with its support value and a list of supporting tree IDs. To facilitate the post-processing of the mining result, the output data is wrapped in XML format, which can be further transformed to HTML or other formats using XSTL language. In fact, the system provides both XML data and HTML data transferred from the XML data via an XSL template. The output will be further described in the following interface section.

Mining Engine Interface Illustration.

In this section, how the mining engine works is shown through a series of screenshots.

Being mentioned earlier, frequent agreement subtree (FAST) mining engine provides two types of user interfaces: one is for TreeBASE users and the other is for general users. However, only the first group of interfaces will be introduced in the following paragraph, because the second group of interfaces differ first the first one only in different input data formats.


Frequent Agreement Subtree Mining - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://aria.njit.edu/mediadb/fast/main.html
Go
Home
TreeBASE users
General users
Algorithm
Mining frequent subtrees
Sample trees
Mining frequent subtrees
Sample trees
Submit
Reset
Please choose different input tree sets
Tree Mining Result
TreeBASE
Click the above icon to have access to TreeBASE.
Local intranet

Input Trees : TREE1516 TREE1517 TREE1518 TREE1519 TREE1515

Maxsize: 5

Minsup: 0.5

Rooted: ✓

Figure 4.7 A screenshot showing the interface for TreeBASE users.

Interface 1: Mining frequent subtrees for TreeBASE user.

This is the primary interface designed for TreeBASE users. Here you can submit a request in 3 steps:

- Type in tree IDs as described in TreeBASE in the text window (these trees must be from the 17 sample trees).

  For example, suppose the input is the following set of TreeBASE IDs,

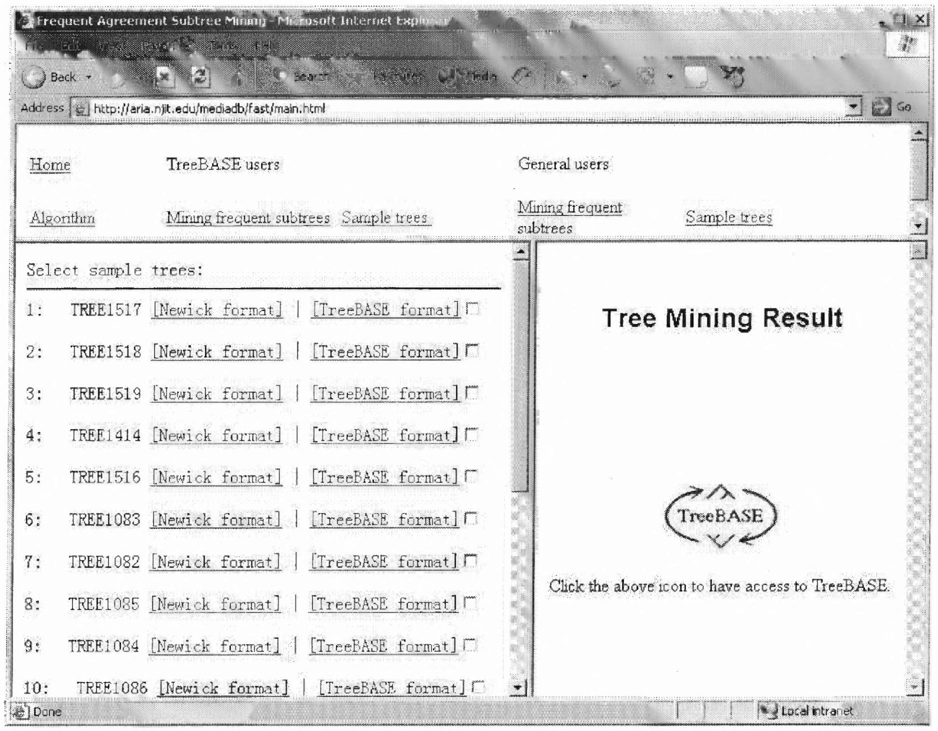
  ```
  TREE1516
  TREE1517
  TREE1518
  TREE1519
  ```

  Internally, the online engine will preprocess the input to collect the real phylogenetic trees, shown below, corresponding to these tree IDs.

  ```
  TREE TREE1516 = ((Hamamelis_japonica, (Hamamelis_mexicana,
  (Hamamelis_vernalis, Hamamelis_virginiana)), Hamamelis_mollis),
  Fothergilla_major);
  TREE TREE1517 = (Hamamelis_japonica, ((Hamamelis_mexicana,
  Hamamelis_vernalis, Hamamelis_virginiana), (Hamamelis_mollis,
  Fothergilla_major)));
  TREE TREE1518 = ((Hamamelis_japonica, (Hamamelis_mexicana,
  Hamamelis_vernalis, Hamamelis_virginiana), Hamamelis_mollis),
  Fothergilla_major);
  TREE TREE1519 = (Hamamelis_japonica, (((Hamamelis_mexicana,
  Hamamelis_vernalis), Hamamelis_virginiana), (Hamamelis_mollis,
  Fothergilla_major)));
  ```

- Provide appropriate parameter values for Maximum and Minsup. For example, the maximum size being 5 means the system will discover frequent subtrees with at most 5 leaves. The minimum support of being 0.3 means a subtree is frequent only if it occurs in at least 30 percent of input data trees.

- Click the "Submit" button.


Frequent Agreement Subtree Mining - Microsoft Internet Explorer
Back
Address http://aria.njit.edu/mediadb/fast/main.html
Go
Done
Local intranet

Home TreeBASE users General users

Algorithm Mining frequent subtrees Sample trees Mining frequent subtrees Sample trees

```
Select sample trees:
1:  TREE1517 [Newick format] | [TreeBASE format]
2:  TREE1518 [Newick format] | [TreeBASE format]
3:  TREE1519 [Newick format] | [TreeBASE format]
4:  TREE1414 [Newick format] | [TreeBASE format]
5:  TREE1516 [Newick format] | [TreeBASE format]
6:  TREE1083 [Newick format] | [TreeBASE format]
7:  TREE1082 [Newick format] | [TreeBASE format]
8:  TREE1085 [Newick format] | [TreeBASE format]
9:  TREE1084 [Newick format] | [TreeBASE format]
10: TREE1086 [Newick format] | [TreeBASE format]
```

Tree Mining Result

TreeBASE

Click the above icon to have access to TreeBASE.

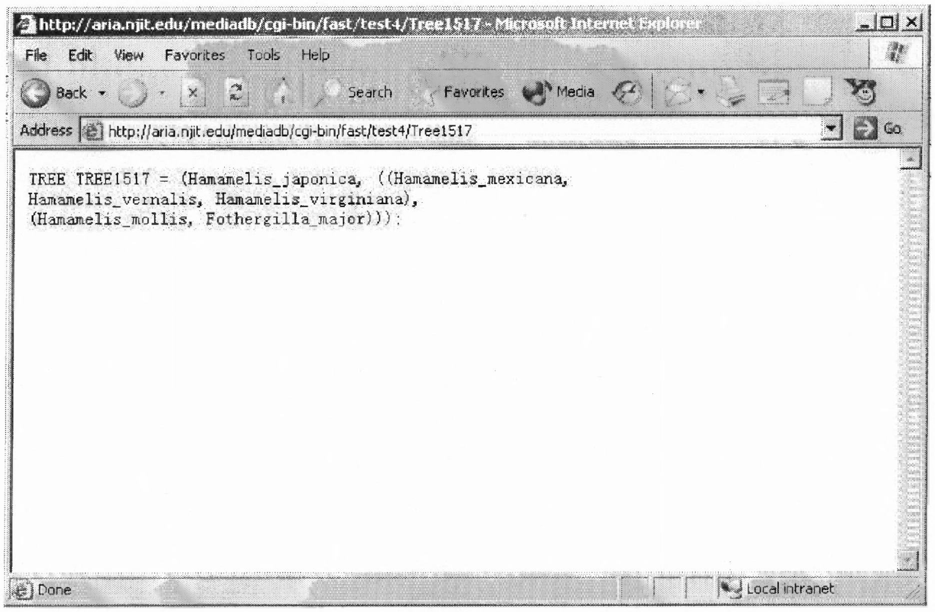
Figure 4.8 A screenshot showing a batch of sample TreeBASE trees.

Figure 4.7 is a screenshot of interface 1.

Interface 2: a database of TreeBase trees identified by TreeBASE IDs. Users can also input data trees through the page of sample trees by clicking on "Sample TreeBASE trees" button in the main menu or click "Please choose different input tree sets" hyperlink in previous interface 1.

Users are allowed to select at most 8 trees as input trees, which will appear in the main interface once selected. If the number of selected trees is 0, then the default tree IDs in the main interface will be the input trees. If the number of selected trees is larger than 8, then only the first 8 trees are taken as input trees. To select, check the boxes of interesting trees.

Click [Newick format] to view the parenthesized tree format (newick format) of a phylogenetic tree. Figure 4.9 shows the Newick format of tree1517.


http://aria.njit.edu/mediadb/cgi-bin/fast/test4/Tree1517 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://aria.njit.edu/mediadb/cgi-bin/fast/test4/Tree1517 Go
Done
Local intranet

```
TREE TREE1517 = (Hamamelis_japonica, ((Hamamelis_mexicana,
Hamamelis_vernalis, Hamamelis_virginiana),
(Hamamelis_mollis, Fothergilla_major)));
```

Figure 4.9 A screenshot showing the Newick format of tree1517.

Click [TreeBASE format] to view the corresponding graphic representation of the tree, as shown in Figure 4.10.

Mining result.

The result consists of all frequent subtrees satisfying the user-specified parameter values as well as the relevant informations pertaining to each subtree. The relevant informations refer to the total number of input trees containing the subtree and the IDs of these trees. The mining result is then wrapped in XML format and transformed to HTML through XSL.

Figure 4.11 shows the raw XML data of mining result. Figure 4.12 is a screenshot showing a list of mining results. Figure 4.13 shows the graphic representation of the subtree in a java applet when user clicks the any subtree link.

When clicking on a supporting tree ID under a subtree, the user will see the graphic representation of the data tree, where the subtree is highlighted by the labels

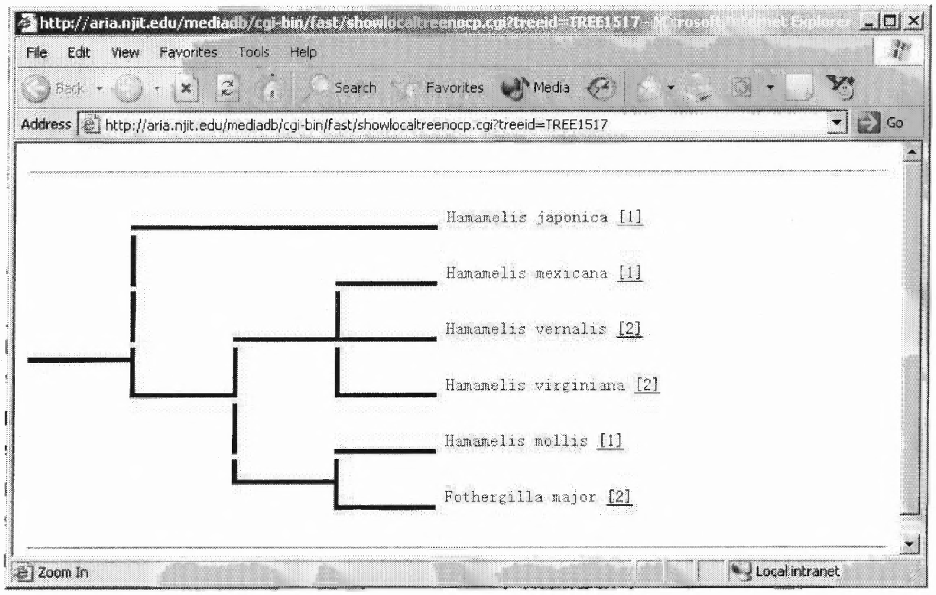
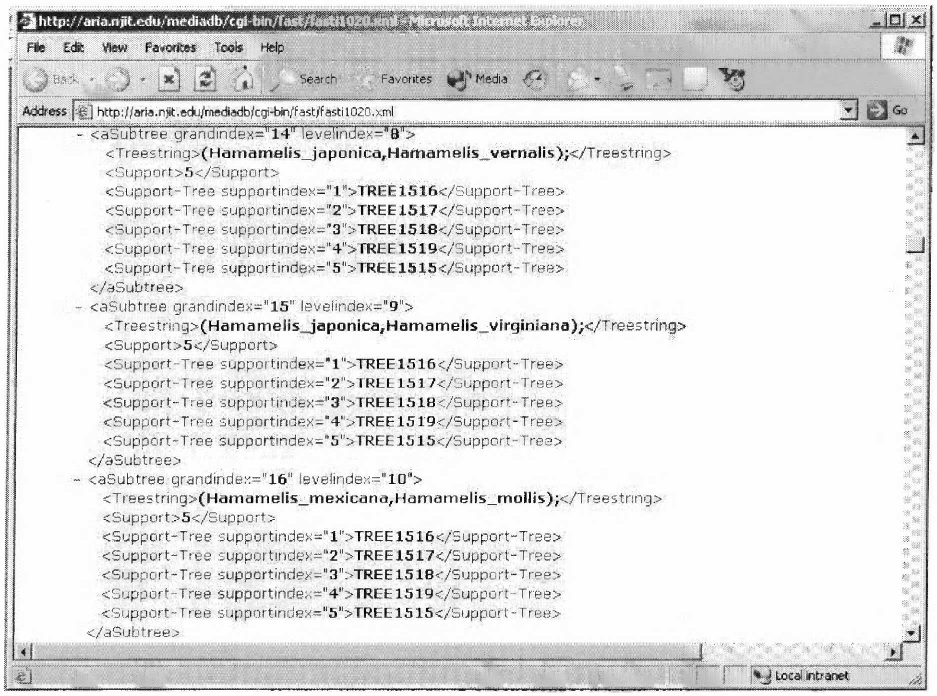
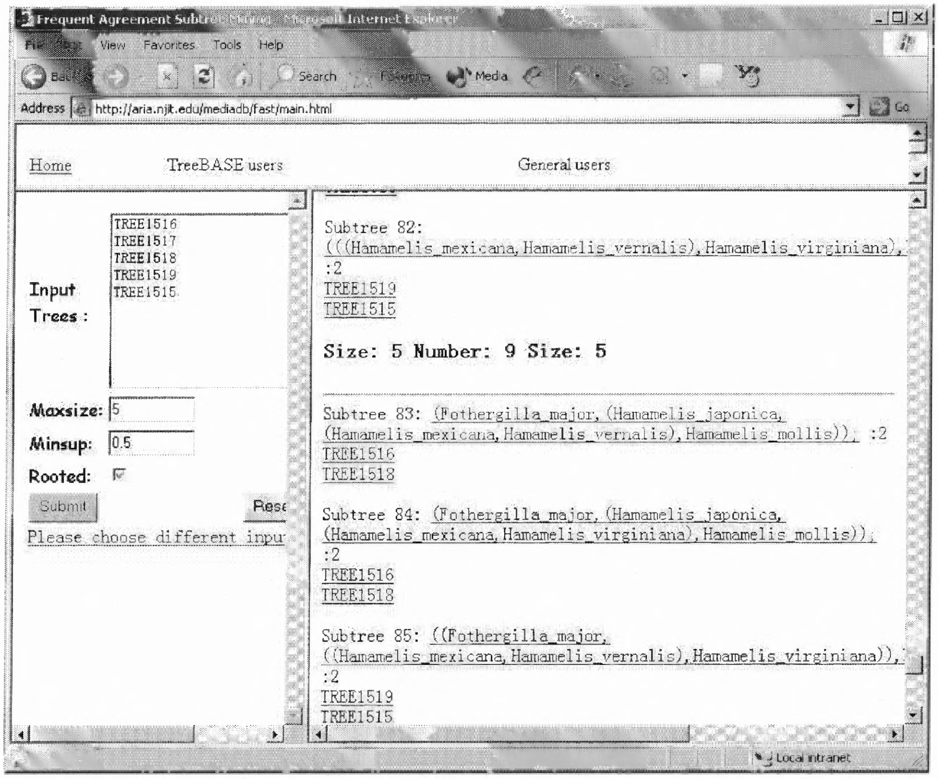

http://aria.njit.edu/mediadb/cgi-bin/fast/showlocaltreenocp.cgi?treeid=TREE1517 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://aria.njit.edu/mediadb/cgi-bin/fast/showlocaltreenocp.cgi?treeid=TREE1517
Hamamelis japonica [1]
Hamamelis mexicana [1]
Hamamelis vernalis [2]
Hamamelis virginiana [2]
Hamamelis mollis [1]
Fothergilla major [2]
Zoom In
Local intranet

Figure 4.10 A screenshot showing the graphic representation of tree1517.


http://aria.njit.edu/mediadb/cgi-bin/fast/fasti1020.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://aria.njit.edu/mediadb/cgi-bin/fast/fasti1020.xml
Local intranet

```
- <aSubtree grandindex="14" levelindex="8">
    <Treestring>(Hamamelis_japonica,Hamamelis_vernalis);</Treestring>
    <Support>5</Support>
    <Support-Tree supportindex="1">TREE1516</Support-Tree>
    <Support-Tree supportindex="2">TREE1517</Support-Tree>
    <Support-Tree supportindex="3">TREE1518</Support-Tree>
    <Support-Tree supportindex="4">TREE1519</Support-Tree>
    <Support-Tree supportindex="5">TREE1515</Support-Tree>
  </aSubtree>
- <aSubtree grandindex="15" levelindex="9">
    <Treestring>(Hamamelis_japonica,Hamamelis_virginiana);</Treestring>
    <Support>5</Support>
    <Support-Tree supportindex="1">TREE1516</Support-Tree>
    <Support-Tree supportindex="2">TREE1517</Support-Tree>
    <Support-Tree supportindex="3">TREE1518</Support-Tree>
    <Support-Tree supportindex="4">TREE1519</Support-Tree>
    <Support-Tree supportindex="5">TREE1515</Support-Tree>
  </aSubtree>
- <aSubtree grandindex="16" levelindex="10">
    <Treestring>(Hamamelis_mexicana,Hamamelis_mollis);</Treestring>
    <Support>5</Support>
    <Support-Tree supportindex="1">TREE1516</Support-Tree>
    <Support-Tree supportindex="2">TREE1517</Support-Tree>
    <Support-Tree supportindex="3">TREE1518</Support-Tree>
    <Support-Tree supportindex="4">TREE1519</Support-Tree>
    <Support-Tree supportindex="5">TREE1515</Support-Tree>
  </aSubtree>
```

Figure 4.11 A piece of XML file expressing the mining result of Phylominer algorithm.



Frequent Agreement Subtree Mining - Microsoft Internet Explorer

Address http://aria.njit.edu/mediadb/fast/main.html

Home TreeBASE users General users

Input Trees : TREE1516 TREE1517 TREE1518 TREE1519 TREE1515

Maxsize: 5

Minsup: 0.5

Rooted: ☑

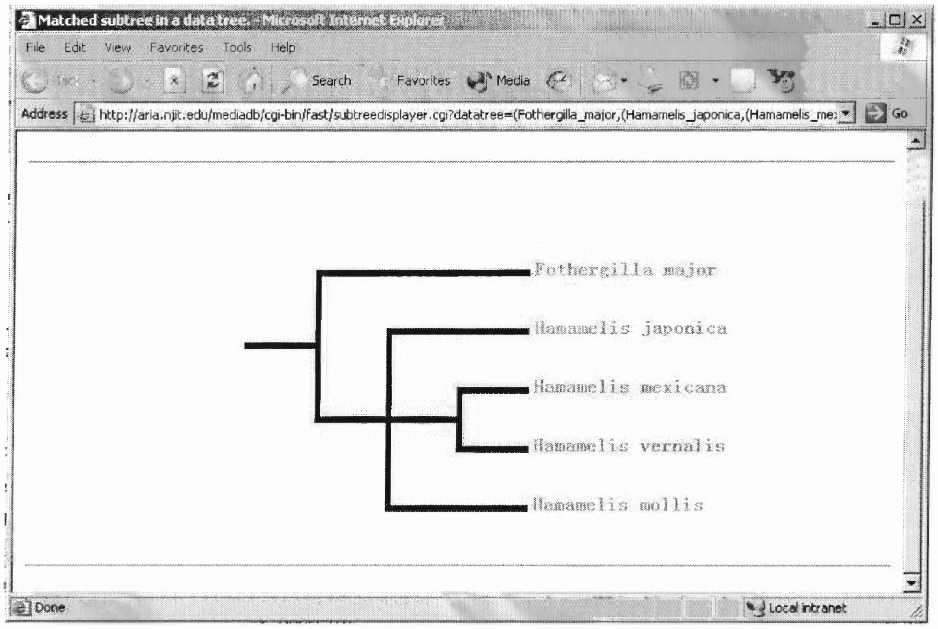
Submit Rese

Please choose different inpu

```
Subtree 82:
(((Hamamelis_mexicana,Hamamelis_vernalis),Hamamelis_virginiana),
:2
TREE1519
TREE1515

Size: 5 Number: 9 Size: 5

Subtree 83: (Fothergilla_major,(Hamamelis_japonica,
(Hamamelis_mexicana,Hamamelis_vernalis),Hamamelis_mollis)); :2
TREE1516
TREE1518

Subtree 84: (Fothergilla_major,(Hamamelis_japonica,
(Hamamelis_mexicana,Hamamelis_virginiana),Hamamelis_mollis));
:2
TREE1516
TREE1518

Subtree 85: ((Fothergilla_major,
((Hamamelis_mexicana,Hamamelis_vernalis),Hamamelis_virginiana)),
:2
TREE1519
TREE1515
```

Local intranet

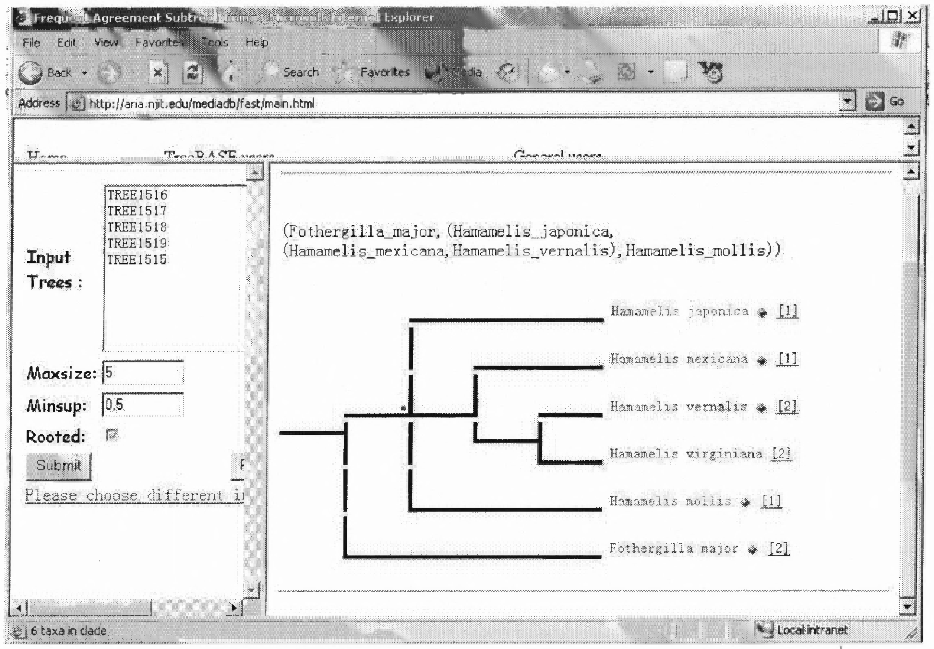
Figure 4.12 A piece of HTML via XSTL showing mining result.



Matched subtree in a data tree - Microsoft Internet Explorer

Address http://aria.njit.edu/mediadb/cgi-bin/fast/subtreedisplayer.cgi?datatree=(Fothergilla_major,(Hamamelis_japonica,(Hamamelis_me:

Figure 4.13 A subtree is shown in a Java applet.



Address http://aria.njit.edu/mediadb/fast/main.html

Input Trees : TREE1516 TREE1517 TREE1518 TREE1519 TREE1515

Maxsize: 5

Minsup: 0.5

Rooted: ✓

Please choose different i

(Fothergilla_major, (Hamamelis_japonica, (Hamamelis_mexicana, Hamamelis_vernalis), Hamamelis_mollis))

6 taxa in clade

Figure 4.14 A subtree is highlighted in a data tree.

of the subtree being rendered red color and with bullets. An example is shown in Figure 4.14.

The interfaces for general users are omitted here, because they are similar to those for TreeBASE users. Interested readers are pointed to the *FAST* website "http://aria.njit.edu/mediadb/fast" for the rest of the instruction.

CHAPTER 5

CONCLUSIONS

This dissertation has presented two frequent structural pattern discovery techniques, which can be applied to phylogeny research as well as other application domains, as long as the data of those applications can be modeled as unordered trees. In cousin mining problem, a novel concept of cousin pair is formally defined. Based on this concept, an unordered tree can be deemed as a set of cousin pair patterns of various kinship distances. Given the cousin pair representations of m multiple unordered trees, then the mining activities can be performed on those cousin pair patterns. An efficient algorithm with a $m|T|^2$ running time complexity has been developed to discover frequent cousin pair patterns from multiple trees. This algorithm is also extended to the unrooted tree scope. The algorithm is implemented in K language and an online mining engine has been setup for the Internet users. The algorithm is experimented on both synthetic data and real phylogeny to achieve good performance. Cousin pair mining has been experimented in several phylogenetic tree applications including consensus tree comparison, co-evolution discovery and supertree analysis.

The frequent agreement subtree mining problem is an extension of the traditional maximum agreement subtree problem. The algorithm proposed in this work adopts an Apriori framework. The foundation of the algorithm is a phylogeny-aware canonical form for leaf labeled trees. Based on this canonical form, a unique tree expansion scheme is devised to grow candidate subtree patterns through joining two lower level subtrees. This technique is also extended to unrooted trees. The algorithm has been implemented in C++ and integrated into an online mining engine. The intensive test has been conducted to verify the correctness and completeness of the algorithm. The experiment of the algorithm on real phylogenetic trees achieved informative results.

The research of this work in structural database mining is continuing [52]. Related work in previous stage concentrated on tree comparisons such as XML query by example [72] and Pathway-involved gene tree distance [71] development. In the near future, the tree mining algorithm will be extended to more complicated phylogenetic tree models, where both tree topology and edge length information are equally important.

The long term goal of this research is to develop a rich set of data mining techniques for a broad range of biological data that can be modeled as tree structure or graph structure. For instance, how to mine frequent embedded tree structures from RNA trees has been drawing the author's research attention for a long time and is going to be attacked in the near future.

Another important research direction is to identify more domain-specific applications which can take advantage of the discovered patterns produced by various structural mining algorithms. In particular, the future work will consider how to utilize those frequent structures to develop novel applications such as outliers detection, classification, clustering and supertree analysis algorithms.

APPENDIX A

SOURCE CODE OF PHYLOMINER

```
/**********************************************************************/
/*                                                                    */
/*      (c) Copyright 2002-2004                                       */
/*      All rights reserved                                           */
/*      Programs written by Sen Zhang                                 */
/*      (the New Jersey Institute of Technology)                      */
/*      RA in the group of Jason T. L. Wang (New Jersey Institute */
/*      of Technology) and Dennis Shasha (New York University)    */
/*                                                                    */
/*      This is the main excerpt of Phylominer program C++ code.  */
/*      Permission to use, copy, modify, and distribute this      */
/*      software and its documentation for any purpose and without*/
/*      fee is hereby granted, provided that this copyright       */
/*      notice appears in all copies.  Programmer(s) makes no     */
/*      representations about the suitability of this            */
/*      software for any purpose.  It is provided "as is" without */
/*      express or implied warranty.                              */
/*                                                                    */
/**********************************************************************/
#include "treearray.h"
#include "frequentsubtreearray.h"
#include "expander.h"
#include "sharetester.h"
#include "detector.h"
#include "common.h"
#include <sys/types.h>
#include <time.h>

#define MINSIZE 4
#define CANDIDATESIZE 4
#define COMMA ','
#define SEMIC ';'
#define RP ')'
#define LP '('
/*****************************************************************/
/*                                                               */
/* Main function                                                 */
/*                                                               */
/*****************************************************************/
int main(int argc, char ** argv)
{
TreeArray * atreearray;
FrequentSubtreeArray * afsa;
Expander *anexpander;
ShareTester *asharetester;
Detector *adetector;
int i, j,k,l;
int treenumber;
int leafnumber;
int expandtreenumber;
char tempstring[STRLEN];
char treestring[2][STRLEN];

int ksizenumber;
int frequentnum;
double frequentnumf;
int currentlevel;
int previouslevel;
int testtreenumber;
int candidatenum;
int maximumsize;

if (argc!=5)
{
cout<<"Usage: fsd filename threshhold fileresult"<<endl;
cout<<" filename is datafile"<<endl;
cout<<" threshhold, usually larger than 0.5"<<endl;
cout<<" maximum leaf size, integer, 0 for no limit "<<endl;
cout<<" fileresult is the filename for output"<<endl;

return -1;
}

// retrieve information from data file.
afsa=new FrequentSubtreeArray();
atreearray=new TreeArray();
anexpander=new Expander();
asharetester=new ShareTester();
adetector=new Detector();
```

```
atreearray->setdatafilename(argv[1]);
frequentnumf=atof(argv[2]);
maximumsize = atoi(argv[3]);
if (maximumsize<MINSIZE&&maximumsize>0)
{
maximumsize=MINSIZE;
}

afsa->setoutputfilename(argv[3]);
atreearray->populatetree();
if (checkformat(atreearray, argv[4])==-1)
{
delete afsa;
delete atreearray;
delete anexpander;
delete asharetester;
delete adetector;

return -1;
}

   int *leafarray;
   int *treearray;
   treenumber=atreearray->gettreenumber();
   leafnumber=atreearray->getleafnumber();
   treearray=new int[treenumber];
   frequentnum=int(frequentnumf*treenumber);
if (frequentnum<=1) frequentnum =1;
if(frequentnum>treenumber) frequentnum = treenumber;

if (maximumsize>=leafnumber||maximumsize==0)
{
maximumsize=leafnumber;
}
   for(i=0;i<treenumber;i++)
treearray[i]=i+1;
   leafarray=new int[atreearray->getleafnumber()];

   time_t curr_start=time(0);
   currentlevel=2;
   afsa->setcurrentlevel(currentlevel,1);

   for(i=1;i<leafnumber;i++)
   {
 leafarray[0]=i;
 for(j=i+1;j<=leafnumber;j++)
{
sprintf(tempstring, "(%d,%d);\0", i, j);
leafarray[1]=j;
afsa->addsubtree(tempstring, currentlevel,leafarray,
treenumber, treearray);
}
}
   ksizenumber=afsa->getpreviouslevelnumber(currentlevel);

   while(ksizenumber>1&&currentlevel<maximumsize)
   {

previouslevel=currentlevel;
// start from
currentlevel++;

afsa->setcurrentlevel(currentlevel,1);

for (i=0;i<ksizenumber-1;i++)
{
for(j=i+1;j<ksizenumber;j++)
{
strcpy(treestring[0],afsa->getsubtree(previouslevel,i));
strcpy(treestring[1],afsa->getsubtree(previouslevel,j));
asharetester->settree(0,treestring[0]);
asharetester->settree(1,treestring[1]);
asharetester->sharetest();

if (asharetester->confirm()!=1)
{
continue;
}
testtreenumber=afsa->testcommon(previouslevel,i,j);
if(testtreenumber<frequentnum)
{
continue;
}

afsa->expandleafarray(previouslevel,i,j);
anexpander->settree(0,treestring[0]);
anexpander->settree(1,treestring[1]);
anexpander->expand();
expandtreenumber=anexpander->getcandidatenumber();
for (k=0;k<expandtreenumber;k++)
{
// only one of 4 candiates can be frequent
// a cut off is also implemented here.
candidatenum=0;
for (l=0;l<testtreenumber;l++)
{
adetector->setdatatree(atreearray->getsubtree(afsa->gettreeid(l)-1));
adetector->setsubtree(anexpander->getsubtree(k));
adetector->setleafarray(currentlevel,afsa->getnewleavesarray());
adetector->parsetree();

if (adetector->confirm()==1)
{
treearray[candidatenum]=afsa->gettreeid(l);
candidatenum++;
}
}
if (candidatenum>=frequentnum)
{
// prepare leaflist, ready in newleafarray

if(afsa->addsubtree(anexpander->getsubtree(k),currentlevel,
afsa->getnewleavesarray() ,candidatenum, treearray)==-1)
{
goto finish;
}
}
}
}
}

ksizenumber=afsa->getpreviouslevelnumber(currentlevel);

if (ksizenumber==0)
afsa->setcurrentlevel(currentlevel-1,0);
};

finish:
time_t curr_end=time(0);
time_t elapsed_time;
elapsed_time=curr_end-curr_start;
writetoxml(atreearray, afsa, argv[4], frequentnumf,
maximumsize, elapsed_time);
delete afsa;
delete atreearray;
delete anexpander;
delete asharetester;
delete adetector;
delete []treearray;
delete []leafarray;

return 1;
}


/************************************************************/
/*                                                          */
/* XML Mining Result Writer                                 */
/*                                                          */
/************************************************************/

int writetoxml(TreeArray * atreearray,
   FrequentSubtreeArray * afsa,
   char* outputdatafile,
   double minsup,
   int maxsize,
   time_t elapsed_time)
{
int i,j,k,l;
int lindex;
int currentlevelnumber;
int support, toplevel;
ofstream ResultFile(outputdatafile);
//Creates an ofstream object named outputfile

if (! ResultFile) // Always test file open
{
return -1;
}

ResultFile <<"<?xml version=\"1.0\"?>" << endl;
ResultFile <<"<Frequent-Agreement-Subtree>"<<endl;
ResultFile <<"<Software-Version>"<<VERSION<<"</Software-Version>"
<<endl;
ResultFile <<"<COPYWRIGHT>"<<COPYWRIGHT<<"</COPYWRIGHT>"<<endl;
ResultFile<<"<Maximum-Number-Allowed-Subtrees>"
<<MAXSUBTREENUM<<"</Maximum-Number-Allowed-Subtrees>"<<endl;
ResultFile <<"\t<Input>"<<endl;
ResultFile <<"\t\t<DataSet>"<<endl;
```

```
for(i=0;i<atreearray->gettreenumber();i++)
ResultFile<<"\t\t\t<Tree index=\""<<i+1<<"\">"
<<atreearray->getsubtree(i)<<"</Tree>"<<endl;
ResultFile<<"\t\t</DataSet>"<<endl;
ResultFile<<"\t\t<Tree-Size>"<<atreearray->getleafnumber()
<<"</Tree-Size>"<<endl;
ResultFile<<"\t\t<Tree-Num>"<<atreearray->gettreenumber()
<<"</Tree-Num>"<<endl;
ResultFile<<"\t\t<Min-Sup>"<<minsup<<"</Min-Sup>"<<endl;
ResultFile<<"\t\t<Max-Size>"<<maxsize<<"</Max-Size>"<<endl;
ResultFile<<"\t</Input>"<<endl;

//Below is the XML format of output of result.
lindex=1;
toplevel=afsa->getcurrentlevel();
ResultFile<<"<Result Total=\""<<afsa->getcounter()
<<"\" Elapsed-Time-in-Secs=\""<<elapsed_time<<"\">"<<endl;
//unrooted, starting from size 3
//rooted, starting from size 2
for(j=2;j<afsa->getcurrentlevel()+1;j++)
{
currentlevelnumber=afsa->getpreviouslevelnumber(j);
ResultFile<<"<Subtree-Set size=\""<<j<<"\" "
<<"number=\""<<currentlevelnumber
<<"\">"<<endl;
for(k=0;k<currentlevelnumber;k++)
{
ResultFile<<"<aSubtree grandindex=\""
<<lindex++<<"\" levelindex=\""
<<k+1<<"\">"<<endl;
ResultFile<<"<Treestring>"
<<afsa->getsubtree(j,k)<<"</Treestring>"<<endl;

support=afsa->getsupport(lindex-2);
ResultFile<<"<Support>"<<support<<"</Support>"<<endl;
for(l=0;l<support; l++)
{
ResultFile<<"<Support-Tree supportindex=\""<<l+1
                                    <<"\" dataindex=\""
                                    <<afsa->getsupporttree(lindex-2,l)
                                    <<"\">"
<<atreearray->getsubtree(
                                afsa->getsupporttree(lindex-2,l)-1)
                                 <<"</Support-Tree>"<<endl;
}
ResultFile<<"</aSubtree>"<<endl;

}
ResultFile<<"</Subtree-Set>"<<endl;
}
ResultFile<<"</Result>"<<endl;
ResultFile <<"</Frequent-Agreement-Subtree>"<<endl;
ResultFile.close();
return 0;
}


/*******************************************************/
/*                                                     */
/* Expander.cpp: implementation of the Expander class. */
/*                                                     */
/*******************************************************/

#include "expander.h"

Expander::Expander()
{

}

Expander::~Expander()
{

}

void Expander::settree(int pindex, char *ptree)
{
strcpy(treestring[pindex],ptree);
}

void Expander::expand()
{
int result;

lastsubtree(0);
lastsubtree(1);
changelastleaf(0);
changelastleaf(1);
```

```
int sametopology;
result=strcmp(secondsubstring[0],secondsubstring[1]);
if (result==0)
{
sametopology=0;
}
else
{
sametopology=result;
}

if (sametopology==0)
{

if (siblingnumber(0)==1)
{
candidatenumber=4;
//same topology, we take the last leaves from both trees,
//way 1: (a,b)
way1(1);
//way 2: a,b
way2(1);

// way 3: (tree1,b)
way3();
// way 4: (tree2,b)
way4();
}
else
{
// same topology, but siblings more than 3,
// with no contraction.
// way 1: (a,b)
way1(2);
// way 2: a, b
way2(2);
candidatenumber=2;
//prune operation is embedded by not calling way3 and way4.
}

}
else
{
// different topology
int larger;
int result1;
candidatenumber=1;
result1=nestnumber();
if (result1 ==-1)
{
larger=0;
}
else
{
larger=1;
}
secondsubtree(larger,abs(larger-1));
// this result is free from pruning,
char *pos;
pos=strstr(treestring[larger],lastsubstring[larger]);
if (pos==treestring[larger])
{
// the lastsubtree of larger tree is larger tree itself
// then simply copy the merged result to expandedtree
strcpy(candidatesubtree[0],mergedsubtree);
mergedtreelen=strlen(mergedsubtree);
if (mergedsubtree[mergedtreelen-1]==')')
strcat(candidatesubtree[0],";");
else
strcat(candidatesubtree[0],");");

}
else
{
// the lastsubtree of larger tree is a perfect subtree
// of larger tree, then replaced the lastsubtree
// with the mergered tree.
ll=0;
kk=0;
ii=0;
largertreelen=strlen(treestring[larger]);
mergedtreelen=strlen(mergedsubtree);
lastsublen[larger]=strlen(lastsubstring[larger]);
do{
candidatesubtree[0][kk++]=treestring[larger][ii++];
}while(kk<secondposition[larger]);

mergedtreelen=strlen(mergedsubtree);
do{
candidatesubtree[0][kk++]=mergedsubtree[ll++];
```

```
}while(ll<mergedtreelen);
ii=ii+lastsublen[larger];
do
{
candidatesubtree[0][kk++]=treestring[larger][ii++];
}while(kk<largertreelen+mergedtreelen-lastsublen[larger]-1);
candidatesubtree[0][kk++]=';';
candidatesubtree[0][kk]='\0';

}
}
}


int Expander::getsubtreenumber()
{
return 4;
}

void Expander::testdata()
{
strcpy(treestring[0],"((1,2),((3,4),51));");
strcpy(treestring[1],"((1,2),((3,4),41));");
}

void Expander::lastsubtree(int pindex)
{
int len;
int i,j,k;
int left;
char thischar;

len=strlen(treestring[pindex]);
i=len-1-1;
//first 1 for 0 based index, second 1 for ';'

while (treestring[pindex][i]==')')
{
i--;
}

firstposition[pindex]=i+1;
left=0;
do{
thischar=treestring[pindex][i--];
```

```
switch (thischar)
{
case ')':
left++;
break;
case '(':
left--;
break;
default:
break;
};

} while (left>=0);
secondposition[pindex]=i+1;

k=0;
j=secondposition[pindex];

do{
lastsubstring[pindex][k++]=treestring[pindex][j++];
}while(j<=firstposition[pindex]);

lastsubstring[pindex][k]='\0';

}

void Expander::secondsubtree(int pbindex, int psindex)
{

int len;
int i,j,k,f,l;
int subtreelen[2];
int smallsubtreelen;
int differentlevel;

len=strlen(lastsubstring[pbindex]);
i=len-1;
while(lastsubstring[pbindex][i--]!=',');

if(lastsubstring[pbindex][i]==')')
differentlevel=1;
else
differentlevel=0;

len=strlen(lastsubstring[psindex]);
```

```
i=len-1;
while(lastsubstring[psindex][i--]!=',');

if (siblingnumber(psindex)==1)
{

j=0;
smallsubtree[j++]=',';
do{
smallsubtree[j]=lastsubstring[psindex][j];
j++;
}while (j+1<=i+1); // same as while(j<i);
/*modified sept 5, 2003 */

int temp;
char *tempresult;
temp=j;


smallsubtree[j++]=')';
smallsubtree[j]='\0';
differentlevel=1;

if ((tempresult=strstr(lastsubstring[pbindex], smallsubtree))
==NULL)
{
j=temp;
smallsubtree[j++]=',';
smallsubtree[j]='\0';
differentlevel=0;
tempresult=strstr(lastsubstring[pbindex], smallsubtree);
if (tempresult==NULL)
cout<<"impossible in two sibling cases"<<endl;
}


}
else
{
j=0;
do{
smallsubtree[j]=lastsubstring[psindex][j];
j++;
}while (j<=i);
```

```
smallsubtree[j++]=')';
smallsubtree[j]='\0';
}

char *pos;
k=0;
f=0;

if (smallsubtree[0]==',')
{
do
{
mergedsubtree[k++]=lastsubstring[pbindex][f++];

}while(k<pos-lastsubstring[pbindex]+1);
l=0;

subtreelen[psindex]=strlen(lastsubstring[psindex]);
subtreelen[pbindex]=strlen(lastsubstring[pbindex]);
smallsubtreelen=strlen(smallsubtree);
do{
mergedsubtree[k++]=lastsubstring[psindex][l++];
}while(l<subtreelen[psindex]);
if(smallsubtree[1]=='(')
{
if (differentlevel==0)
{
f=f+smallsubtreelen-2;
do
{
mergedsubtree[k++]=lastsubstring[pbindex][f++];
}while(k<subtreelen[psindex]+subtreelen[pbindex]
-smallsubtreelen+2);
}
else
{
f=f+smallsubtreelen-2;
do
{
mergedsubtree[k++]=lastsubstring[pbindex][f++];
}while(k<subtreelen[psindex]+subtreelen[pbindex]
-smallsubtreelen+1);

}
}
```

```
else
{
f=f+smallsubtreelen-2;
do
{
mergedsubtree[k++]=lastsubstring[pbindex][f++];
}while(k<subtreelen[psindex]+subtreelen[pbindex]
-smallsubtreelen+2);
}
}
else if (smallsubtree[0]=='(')
{
do
{
mergedsubtree[k++]=lastsubstring[pbindex][f++];

}while(k<pos-lastsubstring[pbindex]);
l=0;
subtreelen[psindex]=strlen(lastsubstring[psindex]);
subtreelen[pbindex]=strlen(lastsubstring[pbindex]);
smallsubtreelen=strlen(smallsubtree);
do{
mergedsubtree[k++]=lastsubstring[psindex][l++];

}while(l<subtreelen[psindex]);
f=f+smallsubtreelen;
do
{
mergedsubtree[k++]=lastsubstring[pbindex][f++];
}while(k<subtreelen[psindex]+subtreelen[pbindex]
-smallsubtreelen);

}
else
{
cout<<"If impossible."<<endl;
}
if (mergedsubtree[k-1]!=')')
mergedsubtree[k++]=')';
mergedsubtree[k]='\0';

  }

int Expander::getcandidatenumber()
{
```

```
return candidatenumber;
}

char * Expander::getcandidatetree(int pindex)
{
return candidatesubtree[pindex];
}

void Expander::changelastleaf(int pindex)
{
int len;
int i,j,k,l;
i=j=k=l=0;
len=strlen(lastsubstring[pindex]);
i=len-1;
while(lastsubstring[pindex][i--]!=',');
i++;
i++;
k=i;
do{
secondsubstring[pindex][j]=lastsubstring[pindex][j];
j++;
}while (j!=i);



secondsubstring[pindex][j++]='0';
secondsubstring[pindex][j++]=')';
secondsubstring[pindex][j]='\0';
// take the last label at the same time.

do{
lastlabel[pindex][l++]=lastsubstring[pindex][k++];
}while(lastsubstring[pindex][k]!=')');
lastlabel[pindex][l]='\0';
}

int Expander::siblingnumber(int pindex)
{
int len;
int i;
char thischar;
int left=0;
int commanumber=0;
```

```
len=strlen(lastsubstring[pindex]);
i=len-1;

do{
thischar=lastsubstring[pindex][i--];
switch (thischar)
{
case ')':
left++;
break;
case '(':
left--;
break;
case ',':
if (left==1)
commanumber++;
break;
default:
break;
}

}while(left>0);

return commanumber;
}

void Expander::way1(int kind)
{
if(atoi(lastlabel[0])<atoi(lastlabel[1]))
sprintf(way[0],"(%s,%s)\0",lastlabel[0],lastlabel[1]);
else
sprintf(way[0],"(%s,%s)\0",lastlabel[1],lastlabel[0]);
memset(mergedsubtree,0,TREELEN);
subtreelen[0]=strlen(lastsubstring[0]);
smallsubtreelen=strlen(way[0]);
largertreelen=strlen(treestring[0]);
ii=subtreelen[0]-1;
while(lastsubstring[0][ii--]!=',');
ii++;
ii++;
jj=ll=kk=0;

do
{
mergedsubtree[kk++]=lastsubstring[0][jj++];
}while(kk<ii);
do
{
mergedsubtree[kk++]=way[0][ll++];
}while(ll<smallsubtreelen);
int skimlen=0;
while(lastsubstring[0][jj++]!=')')
{
skimlen++;
}
// to skim the label just replaced.
jj--;
do
{
mergedsubtree[kk++]=lastsubstring[0][jj++];
}while(kk<subtreelen[0]+smallsubtreelen-skimlen);
mergedsubtree[kk]='\0';
//second step is to put the mergedtree in treestring[0] by
//replacing the lastsubtree with mergedtree
//two cases again
mergedtreelen=strlen(mergedsubtree);

sprintf(templast,"%s%c\0",lastsubstring[0],';');
if(strcmp(treestring[0],templast)==0)
{
strcpy(candidatesubtree[0],mergedsubtree);
strcat(candidatesubtree[0],";");
}
else
{
ii=jj=kk=ll=0;
ii=largertreelen;
pos=strstr(treestring[0], lastsubstring[0]);
kk=ll=ii=jj=0;
do
{
candidatesubtree[0][kk++]=treestring[0][ll++];
}while(kk<pos-treestring[0]);
do
{
candidatesubtree[0][kk++]=mergedsubtree[ii++];
}while(ii<mergedtreelen);
if (kind==2)
ll=ll+subtreelen[0];
else
```

```
ll=ll+subtreelen[0];
do
{
candidatesubtree[0][kk++]=treestring[0][ll++];
}while(kk<largertreelen+mergedtreelen+2-subtreelen[0]);
candidatesubtree[0][kk++]=';';
        candidatesubtree[0][kk]='\0';
}
}

void Expander::way2(int kind)
{
lastlabel[0],lastlabel[1])<1)
if(atoi(lastlabel[0])<atoi(lastlabel[1]))
sprintf(way[0],"%s,%s\0",lastlabel[0],lastlabel[1]);
else
sprintf(way[0],"%s,%s\0",lastlabel[1],lastlabel[0]);

memset(mergedsubtree,0,TREELEN);
subtreelen[0]=strlen(lastsubstring[0]);
smallsubtreelen=strlen(way[0]);
largertreelen=strlen(treestring[0]);
ii=subtreelen[0]-1;
while(lastsubstring[0][ii--]!=',');
ii++;
ii++;
jj=ll=kk=0;

do
{
mergedsubtree[kk++]=lastsubstring[0][jj++];
}while(kk<ii);
do
{
mergedsubtree[kk++]=way[0][ll++];
}while(ll<smallsubtreelen);
int skimlen=0;
while(lastsubstring[0][jj++]!=')')
{
skimlen++;
}
// to skim the label just replaced.
jj--;
do
{
```

```
mergedsubtree[kk++]=lastsubstring[0][jj++];
}while(kk<subtreelen[0]+smallsubtreelen+skimlen);
mergedsubtree[kk]='\0';
mergedtreelen=strlen(mergedsubtree);
sprintf(templast,"%s%c\0",lastsubstring[0],';');
if(strcmp(treestring[0],templast)==0)
{
strcpy(candidatesubtree[1],mergedsubtree);
strcat(candidatesubtree[1],";");
}
else
{
ii=jj=kk=ll=0;
ii=largertreelen;
pos=strstr(treestring[0], lastsubstring[0]);
//put the mergedsubtree inside the largtree
kk=ll=ii=jj=0;
do
{
candidatesubtree[1][kk++]=treestring[0][ll++];
}while(kk<pos-treestring[0]);
do
{
candidatesubtree[1][kk++]=mergedsubtree[ii++];
}while(ii<mergedtreelen);
ll=ll+subtreelen[0];
// this is the difference from way1;
do
{
candidatesubtree[1][kk++]=treestring[0][ll++];
}while(kk<largertreelen+mergedtreelen+1-subtreelen[0]);
candidatesubtree[1][kk++]=';';
candidatesubtree[1][kk]='\0';
}



}

void Expander::way3()
{
    kk=0;
    ll=0;
jj=0;
lastsublen[0]=strlen(lastsubstring[0]);
```

```
mergedsubtree[kk++]='(';
do
{
mergedsubtree[kk++]=lastsubstring[0][ll++];
}while(ll<lastsublen[0]);
mergedsubtree[kk++]=',';
do{
mergedsubtree[kk++]=lastlabel[1][jj++];
}while(jj<strlen(lastlabel[1]));
mergedsubtree[kk++]=')';
mergedsubtree[kk]='\0';
sprintf(templast,"%s%c\0",lastsubstring[0],';');
if(strcmp(treestring[0],templast)==0)
{
strcpy(candidatesubtree[2],mergedsubtree);
strcat(candidatesubtree[2],";");
}
else
{

kk=ll=ii=jj=0;
largertreelen=strlen(treestring[0]);
ii=largertreelen;
pos=strstr(treestring[0], lastsubstring[0]);
//put the mergedsubtree inside the largtree

do
{
candidatesubtree[2][kk++]=treestring[0][ll++];
}while(kk<pos-treestring[0]);
do
{
candidatesubtree[2][kk++]=mergedsubtree[jj++];
}while(jj<mergedtreelen+2);
ll=ll+lastsublen[0];
// this is the difference from way1;
do
{
candidatesubtree[2][kk++]=treestring[0][ll++];
}while(kk<largertreelen+mergedtreelen+2-lastsublen[0]);
candidatesubtree[2][kk++]='\0';
candidatesubtree[2][kk]=';';
}
}
```

```
void Expander::way4()
{
kk=0;
ll=0;
jj=0;
lastsublen[1]=strlen(lastsubstring[1]);
mergedsubtree[kk++]='(';
do
{
mergedsubtree[kk++]=lastsubstring[1][ll++];
}while(ll<lastsublen[1]);
mergedsubtree[kk++]=',';
do{
mergedsubtree[kk++]=lastlabel[0][jj++];
}while(jj<strlen(lastlabel[0]));
mergedsubtree[kk++]=')';
mergedsubtree[kk]='\0';
sprintf(templast,"%s%c\0",lastsubstring[1],';');
if(strcmp(treestring[1],templast)==0)
{
strcpy(candidatesubtree[3],mergedsubtree);
strcat(candidatesubtree[3],";");
}
else
{

kk=ll=ii=jj=0;
largertreelen=strlen(treestring[1]);
ii=largertreelen;
pos=strstr(treestring[1], lastsubstring[1]);
//put the mergedsubtree inside the largtree

do
{
candidatesubtree[3][kk++]=treestring[1][ll++];
}while(kk<pos-treestring[1]);
do
{
candidatesubtree[3][kk++]=mergedsubtree[jj++];
}while(jj<mergedtreelen+2);

ll=ll+lastsublen[1];
// this is the difference from way1;
do
{
```

```
candidatesubtree[3][kk++]=treestring[1][ll++];
}while(kk<largertreelen+mergedtreelen+2-lastsublen[1]);

candidatesubtree[3][kk++]='\0';
}

}

char * Expander::getlastlabel(int index)
{
return lastlabel[index];
}

char * Expander::getsubtree(int pindex)
{
return candidatesubtree[pindex];
}
int Expander::nestnumber()
{
int i=0;
int j=0;
int length;
int left[2];
int flag=0;

for (i=0;i<2;i++)
{
length=strlen(secondsubstring[i]);
left[i]=0;
j=0;
do {
if (secondsubstring[i][j++]=='(')
left[i]++;
}while (j<length);
}

if (left[0]>left[1])
return -1;
else if (left[0]<left[1])
return 1;
else
{
flag=commanumber();
if (flag==-1)
return -1;
```

```
else if (flag==1)
return 1;
else return 0;
}
}

int Expander::commanumber()
{
int i=0;
int j=0;
int length;
int comma[2];

for (i=0;i<2;i++)
{
length=strlen(secondsubstring[i]);
comma[i]=0;
j=0;
do {
if (secondsubstring[i][j++]==',')
comma[i]++;
}while (j<length);

}

if (comma[0]>comma[1])
return -1;
else if (comma[0]<comma[1])
return 1;
else
{
cout<<"strange!"<<endl;
return 0;
}
}
```

APPENDIX B

SOURCE CODE OF MAIN PERL MODULE

```
/*********************************************************************/
/*                                                                   */
/*       (c) Copyright 2002-2004                                     */
/*       All rights reserved                                         */
/*       Programs written by Sen Zhang                               */
/*       (the New Jersey Institute of Technology)                    */
/*       RA in the group of Jason T. L. Wang (New Jersey Institute   */
/*       of Technology) and Dennis Shasha (New York University)      */
/*                                                                   */
/*                                                                   */
/* This is the main excerpt of perl source code to activate the      */
/* Core Mining Algorithm.                                            */
/* Permission to use, copy, modify, and distribute this software     */
/* and its documentation for academia purpose is hereby granted,     */
/* provided that this copyright notice appears in all copies.        */
/* The programmer makes no representations about the suitability     */
/* of this software for any purpose. It is provided "as is"          */
/* without explicit or implied warranty.                             */
/*********************************************************************/

#!/usr/local/bin/perl5 -w
use CGI
$ENV{'LD_LIBRARY_PATH'}='/usr/lib:/usr/local/lib';

#Initialzation to default value
$qthreshold = 0.3;
$xq=new CGI;
$maxsize = $xq->param('maxsize');
$qthreshold  =  $xq->param('threshold');
$rooted  =  $xq->param('rooted');
$qtreeidset  =  $xq->param('treeidset');

$perl = "/usr/local/bin/perl5";
$notify ="visitnotify.pl";
&notify;

$seq=getsequence();
$prefix = "fasti",
$userinput=$prefix."$seq";
$userinputu=$userinput.".ux";
$dataout=$userinput.".xml";

open (writef, ">$userinput")|| die "Couldn't write $userinput!";
print writef "$qtreeidset";
close(writef) || die "can't close $userinput";
$kr = -111;
$someword = "";

$toufile="/afs/cad/research/ccs/wangj/mediadb/public_html/cgi-bin/
fast/umain";
$torfile="/afs/cad/research/ccs/wangj/mediadb/public_html/cgi-bin/
fast/rmain";
if($rooted!=1)
{
`$toufile $userinput $qthreshold $maxsize $dataout`;
}
else
{
`$torfile $userinput $qthreshold $maxsize $dataout`;
}
print "Content-type:text/html\n\n";
print "<HTML>\n";
print "<BODY>\n";
print "<a target='_blank' href='$dataout'>Mining Result</a>";
if($rooted!=1)
{
uproducehtml();
}
else
{

rproducehtml();
}

print "</body>";
print "</html>";
exit 1;

#############BEGIN SUB#################
sub rproducehtml
{
print<<EOF4;
<script language="javascript">
```

```
var xml = new ActiveXObject("Microsoft.XMLDOM")
xml.async = false
EOF4
print 'xml.load("';
print $dataout;
print '")';
print "\n";

print<<EOF5;
var xsl = new ActiveXObject("Microsoft.XMLDOM")
xsl.async = false
xsl.load("resultr.xsl")

document.write(xml.transformNode(xsl))
</script>
EOF5
return 1;
}
##############BEGIN SUB##################
sub uproducehtml
{
print<<EOF4;
<script language="javascript">
var xml = new ActiveXObject("Microsoft.XMLDOM")
xml.async = false
EOF4
print 'xml.load("';
print $dataout;
print '")';
print "\n";

print<<EOF5;
var xsl = new ActiveXObject("Microsoft.XMLDOM")
xsl.async = false
xsl.load("resultu.xsl")

document.write(xml.transformNode(xsl))
</script>
EOF5
return 1;
}

#######################################################
sub notify
{
```

```
use MIME::Lite;
$pre ="A visit to FAST mining website is made.\n";
@interestfields=("REMOTE_ADDR","REMOTE_PORT","REQUEST_METHOD",
"REQUEST_URI", "UNIQUE_ID");
$whoareyou="";
foreach my $key (@interestfields) {
    $whoareyou=$whoareyou."$key = $ENV{$key}\n";
}
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
$timestamp="";
$year=$year+1900;
$mon=$mon+1;
$timestamp="(Y)".$year;
$timestamp=$timestamp."-(M)".$mon;
$timestamp=$timestamp."-(D)".$mday;
$timestamp=$timestamp."-(H)".$hour;
$timestamp=$timestamp."-(M)".$min;
$timestamp=$timestamp."-(S)".$sec;

#$qtreeidset  =  $xq->param('treeidset');

$pre=$pre.$whoareyou.$timestamp."\n";
$pre=$pre."Maxsize is : ".$maxsize."\n";
$pre=$pre."Minsupp is : ".$qthreshold."\n";
$pre=$pre."Rooted is : ".$rooted."\n";
$pre=$pre."Input Tree Set is : \n". $qtreeidset. "\n";

$pre1=$pre."You can visit the following link for history \n";
$pre1=$pre1."http://aria.njit.edu/cgi-bin/sequence/fastlog.dat\n";

$msg = MIME::Lite->new(
From=>'',
To => '',
Cc => '',
Subject=>'This is a message from Fast Website',
Type=>'TEXT',
Data =>$pre1);
#Data => 'This is a sample Message from Cousin website');

MIME::Lite->send('smtp', 'MAILHOST.NJIT.EDU', Timeout=>60);

$msg->send;
&putinlog($pre);
return 1;
}
```

```
sub putinlog
{
my $logfile;
local($firstp) = @_;
$datain=$firstp."\n------------------------------------------\n";

$logfile="../sequence/fastlog.dat";
        open(ALOG,">> $logfile");
        flock(ALOG,2); # exlusive access
        #seek(ALOG,0,0);
        print ALOG $datain;
        flock(ALOG,8); # release the file
        close ALOG;

}


sub getsequence
{
my $lsequence;

$seqfile="../sequence/cousincounter.dat";
        open(SEQUENCE,"+< $seqfile");
        flock(SEQUENCE,2); # exlusive access
        $lsequence=<SEQUENCE>;
        $llsequence=$lsequence+1;

        seek(SEQUENCE,0,0);
        print SEQUENCE $llsequence;
        flock(SEQUENCE,8); # release the file
        close SEQUENCE;
        return $lsequence;
        exit 1;

}
```

APPENDIX C

PHYLOMINER MINING RESULT IN XML FORMAT

C.1 Schema for XML Output Format of Mining Result

```
/*****************************************************************/
/*                                                               */
/*      (c) Copyright 2002-2004                                  */
/*      All rights reserved                                      */
/*      Programs written by Sen Zhang                            */
/*      (the New Jersey Institute of Technology)                 */
/*      RA in the group of Jason T. L. Wang (New Jersey Institute */
/*      of Technology) and Dennis Shasha (New York University)   */
/*                                                               */
/*      This is the main excerpt of XML related code for         */
/*      displaying Frequent agreement subtree Mining Result.     */
/*      Permission to use, copy, modify, and distribute this     */
/*      software and its documentation for any purpose and without */
/*      fee is hereby granted, provided that this copyright      */
/*      notice appears in all copies.  Programmer(s) makes no    */
/*      representations about the suitability of this           */
/*      software for any purpose.  It is provided "as is" without */
/*      express or implied warranty.                             */
/*                                                               */
/*****************************************************************/

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://aria.njit.edu/meidadb/fast"
xmlns="http://aria.njit.edu/mediadb/fast"
elementFormDefault="qualified">

<xs:element name="Frequent-Agreement-Subtree">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Input">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="DataSet">
       <xs:complexType>
        <xs:element name="Tree" maxOccurs="unbounded">
         <xs:complexType>
```

```
          <xs:attribute name="orderid" type="xs:string"
          use="required"/>
         </xs:complexType>
        </xs:element>
       </xs:complexType>
      </xs:element>
      <xs:element name="Tree-Size" type="xs:positiveInteger"/>
      <xs:element name="Tree-Num" type="xs:positiveInteger"/>
      <xs:element name="Min-Sup" type="xs:decimal"/>
      <xs:element name="Max-Size" type="xs:positiveInteger"/>
     <xs:sequence>
    </xs:complexType>
    <xs:element name="Result">
     <xs:complexType>
      <xs:element name="Subtree-Set" maxOccurs="unbounded">
       <xs:complexType>
        <xs:element name="aSubtree" maxOccurs="unbounded">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="Treestring"/>
           <xs:element name="Support" type="xs:positiveInteger"/>
           <xs:element name="Support-Tree" maxOccurs="unbounded"/>
          </xs:seuqence>
         </xs:complexType>
        </xs:element>
       </xs:complexType>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

C.2 A Sample XML Format of Mining Result

```
<?xml version="1.0"?>
<Frequent-Agreement-Subtree>
<Software-Version>Customized PHYLOMINER for TreeBASE(June 27, 2004)
</Software-Version>
<COPYWRIGHT>NJIT, Data Engineering Lab</COPYWRIGHT>
<Maximum-Number-of-Subtrees>9000</Maximum-Number-of-Subtrees>
<Input>
<DataSet>
<Tree index="1" treeid="TREE1516">
((Hamamelis_japonica,(Hamamelis_mexicana,(Hamamelis_vernalis,
Hamamelis_virginiana)), Hamamelis_mollis),Fothergilla_major);
</Tree>
<Tree index="2" treeid="TREE1517">
(Hamamelis_japonica,((Hamamelis_mexicana, Hamamelis_vernalis,
Hamamelis_virginiana),(Hamamelis_mollis,Fothergilla_major)));
</Tree>
<Tree index="3" treeid="TREE1518">
((Hamamelis_japonica,(Hamamelis_mexicana,Hamamelis_vernalis,
Hamamelis_virginiana),Hamamelis_mollis),Fothergilla_major);
</Tree>
<Tree index="4" treeid="TREE1519">
(Hamamelis_japonica,(((Hamamelis_mexicana,Hamamelis_vernalis),
Hamamelis_virginiana),(Hamamelis_mollis,Fothergilla_major)));
</Tree>
<Tree index="5" treeid="TREE1515">
(Hamamelis_japonica,(((Hamamelis_mexicana,Hamamelis_vernalis),
Hamamelis_virginiana),(Hamamelis_mollis,Fothergilla_major)));
</Tree>
</DataSet>
<Tree-Size>6</Tree-Size>
<Tree-Num>5</Tree-Num>
<Min-Sup>0.8</Min-Sup>
<Max-Size>5</Max-Size>
</Input>
<Result Total="30" Elapsed-Time-in-Secs="0">
<Subtree-Set size="1" number="6">
<aSubtree grandindex="1" levelindex="1">
<Treestring>(Fothergilla_major);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
```

```
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="2" levelindex="2">
<Treestring>(Hamamelis_japonica);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="3" levelindex="3">
<Treestring>(Hamamelis_mexicana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="4" levelindex="4">
<Treestring>(Hamamelis_mollis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="5" levelindex="5">
<Treestring>(Hamamelis_vernalis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="6" levelindex="6">
<Treestring>(Hamamelis_virginiana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
```

```
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
</Subtree-Set>
<Subtree-Set size="2" number="15">
<aSubtree grandindex="7" levelindex="1">
<Treestring>(Fothergilla_major,Hamamelis_japonica);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="8" levelindex="2">
<Treestring>(Fothergilla_major,Hamamelis_mexicana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="9" levelindex="3">
<Treestring>(Fothergilla_major,Hamamelis_mollis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="10" levelindex="4">
<Treestring>(Fothergilla_major,Hamamelis_vernalis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="11" levelindex="5">
<Treestring>(Fothergilla_major,Hamamelis_virginiana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
```

```
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="12" levelindex="6">
<Treestring>(Hamamelis_japonica,Hamamelis_mexicana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="13" levelindex="7">
<Treestring>(Hamamelis_japonica,Hamamelis_mollis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="14" levelindex="8">
<Treestring>(Hamamelis_japonica,Hamamelis_vernalis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="15" levelindex="9">
<Treestring>(Hamamelis_japonica,Hamamelis_virginiana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="16" levelindex="10">
<Treestring>(Hamamelis_mexicana,Hamamelis_mollis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
```

```
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="17" levelindex="11">
<Treestring>(Hamamelis_mexicana,Hamamelis_vernalis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="18" levelindex="12">
<Treestring>(Hamamelis_mexicana,Hamamelis_virginiana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="19" levelindex="13">
<Treestring>(Hamamelis_mollis,Hamamelis_vernalis);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="20" levelindex="14">
<Treestring>(Hamamelis_mollis,Hamamelis_virginiana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="21" levelindex="15">
<Treestring>(Hamamelis_vernalis,Hamamelis_virginiana);</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
```

```
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
</Subtree-Set>
<Subtree-Set size="3" number="9">
<aSubtree grandindex="22" levelindex="1">
<Treestring>
(Fothergilla_major,(Hamamelis_mexicana,Hamamelis_vernalis));
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="23" levelindex="2">
<Treestring>
(Fothergilla_major,(Hamamelis_mexicana,Hamamelis_virginiana));
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="24" levelindex="3">
<Treestring>
(Fothergilla_major,(Hamamelis_vernalis,Hamamelis_virginiana));
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="25" levelindex="4">
<Treestring>
(Hamamelis_japonica,(Hamamelis_mexicana,Hamamelis_vernalis));
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
```

```
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="26" levelindex="5">
<Treestring>
(Hamamelis_japonica,(Hamamelis_mexicana,Hamamelis_virginiana));
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="27" levelindex="6">
<Treestring>
(Hamamelis_japonica,(Hamamelis_vernalis,Hamamelis_virginiana));
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="28" levelindex="7">
<Treestring>
((Hamamelis_mexicana,Hamamelis_vernalis),Hamamelis_mollis);
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="29" levelindex="8">
<Treestring>
((Hamamelis_mexicana,Hamamelis_virginiana),Hamamelis_mollis);
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
```

```
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
<aSubtree grandindex="30" levelindex="9">
<Treestring>
(Hamamelis_mollis,(Hamamelis_vernalis,Hamamelis_virginiana));
</Treestring>
<Support>5</Support>
<Support-Tree supportindex="1">TREE1516</Support-Tree>
<Support-Tree supportindex="2">TREE1517</Support-Tree>
<Support-Tree supportindex="3">TREE1518</Support-Tree>
<Support-Tree supportindex="4">TREE1519</Support-Tree>
<Support-Tree supportindex="5">TREE1515</Support-Tree>
</aSubtree>
</Subtree-Set>
</Result>
</Frequent-Agreement-Subtree>
```

C.3 XSL Source Code

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="html" version="4.0"/>
<xsl:variable name="cgiprogram">showfast.cgi</xsl:variable>
<xsl:variable name="showapplet">subtreedisplayer.cgi</xsl:variable>

<xsl:template match="/">
<html>
<body>
<H2>Software Version:
<xsl:value-of select="Frequent-Agreement-Subtree/Software-Version"/>
</H2>
<H2>COPYWRIGHT:
<xsl:value-of select="Frequent-Agreement-Subtree/COPYWRIGHT"/></H2>
<H2>Maximum number of subtrees allowed to be mined:
<xsl:value-of
select="Frequent-Agreement-Subtree/Maximum-Number-Allowed-Subtrees"/>
</H2>

<hr/>
<H2>Input</H2>

<H3>Data Set: </H3>
<xsl:for-each select="Frequent-Agreement-Subtree/Input/DataSet/Tree">
<xsl:value-of select="./@treeid"/>:

    <xsl:variable name="treeid" select="./@treeid"/>
    <a target="_blank" href="{$cgiprogram}?treeid={$treeid}">
    <xsl:value-of select="."/><br/>
    </a>
</xsl:for-each>

<H3>Tree Size: <xsl:value-of select="Frequent-Agreement-Subtree/
Input/Tree-Size"/></H3>
<H3>Tree Number: <xsl:value-of select="Frequent-Agreement-Subtree/
Input/Tree-Num"/></H3>
<H3>Minsupp:
<xsl:value-of select="Frequent-Agreement-Subtree/Input/Min-Sup"/>
</H3>
<H3>Maximum Tree Size:
<xsl:value-of select="Frequent-Agreement-Subtree/Input/Max-Size"/>
</H3>
```

```
<hr/>
<H2>Result:</H2>
<H3>The total number of frequent agreement subtree is:
<xsl:value-of select="Frequent-Agreement-Subtree/Result/@Total"/>
</H3>
<H3>The total time used:
<xsl:value-of
select="Frequent-Agreement-Subtree/Result/@Elapsed-Time-in-Secs"/>
</H3>
<xsl:for-each select="Frequent-Agreement-Subtree/Result/Subtree-Set">
Size: <xsl:value-of select="./@size"/>
Number: <xsl:value-of select="./@number"/>
<br/>
<xsl:for-each select="./aSubtree">
<xsl:variable name="treestring" select="./Treestring"/>
<a target="_blank" href="{$showapplet}?datatree={$treestring}&amp;
root=1&amp; underscoretospace=1">
<xsl:value-of select="$treestring"/>
</a>
:<xsl:value-of select="./Support"/>
<br/>
<xsl:for-each select="./Support-Tree">
<xsl:variable name="treeid" select="."/>
<a href="{$cgiprogram}?treeid={$treeid}&amp;subtree={$treestring}">
<xsl:value-of select="$treeid"/></a>

<br/>
</xsl:for-each>
<br/>
</xsl:for-each>
</xsl:for-each>
</body>
</html>
<!-- close the xsl:template element -->
</xsl:template>
</xsl:stylesheet>
```

REFERENCES

[1] E. N. Adams. Consensus techniques and the comparison of taxonomic trees. *Systematic Zoology*, 21:390–397, 1972.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th International Conference on Data Engineering*, March 1995.

[4] A. Amir. Maximum agreement subtree in a set of evolutionary trees: metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1996.

[5] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa. Efficiently mining frequent substructures from semi-structured data. In *The Proceedings of International Workshop on Informations & Electrical Engineering*, pages 59–64, Suwon, Korea, May 2002.

[6] T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. In *The 6th International Conference on Discovery Science*, 2003.

[7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000.

[8] K. Bremer. Combinable component consensus. *Cladistics*, 6:369–372, 1990.

[9] Z. Chen, H.V. Jagadish, F. Korn, N. Koudas, and D. Srivastava R. Ng S. Muthukrishnan. Counting twig matches in a tree. In *17th International Conference on Data Engineering*, Heidelberg, Germany, April 2001.

[10] Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *IEEE International Conference on Data Mining*, 2003.

[11] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $o(nlog^3m)$ time. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 245–254, 1999.

[12] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[13] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 1:7–28, 1985.

[14] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press., 1998.

[15] J. A. Doyle and M. J. Donoghue. Fossils and seed plant phylogeny reanalyzed. *Brittonia*, 33:89–106, 1992.

[16] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *20th International Conference on Data Engineering*, pages 522–529, 1999.

[17] M. Farach, T. Przytycka, and M. Thorup. Maximum agreement subtree in a set of evolutionary trees: metrics and efficient algorithms. *Information Processing Letters*, 55:297–301, 1995.

[18] M. Farach and M. Thorup. Optimal evolutionary tree comparison by sparse dynamic programming (extended abstract). In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 770–779, 1994.

[19] J. Felsenstein. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolutionary*, 17(6):368–376, 1981.

[20] J. Felsenstein. PHYLIP – phylogeny inference package (version 3.2). *Cladistics*, 5:164 – 166, 1989.

[21] C. R. Finden and A. D. Gordon. Obtaining common pruned trees. *Journal of Classification*, 2:255–276, 1985.

[22] W. Fitch. Toward the defining the course of evolution: minimum change for a specific tree topology. *Systematic Zoology*, 20, 1971.

[23] G. Ganeshkumar and T. Warnow. Finding a maximum compatible tree for a bounded number of trees with bounded degree is solvable in polynomial time. In O. Gascuel and B.M.E. Moret, editors, *First International Workshop on Algorithms in Bioinformatics (WABI 2001)*, pages 156–163, 2001.

[24] M. Garofalakis and A. Kumar. Correlating xml data streams using tree-edit distance embeddings. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2003.

[25] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[26] S. Holmes and P. Diaconis. Random walks on trees and matchings. *Electronic Journal of Probability*, 7, 2002.

[27] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. 2003.

[28] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Practice of Knowledge Discovery in Databases*, pages 13–23, 2000.

[29] M. Y. Kao, T. W. Lam, T. M. Przytycka, W. K. Sung, and H. F. Ting. General techniques for comparing unrooted evolutionary trees. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 54–65, El Paso, Texas, United States, 1997. ACM Press.

[30] P. Kilpelainen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995.

[31] E. Kubicka, G. Kubicka, and F. R. McMorris. An algorithm to find agreement subtrees. *Journal of Classification*, pages 91–99.

[32] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *1st IEEE International Conference on Data Mining*, pages 313–320, 2001.

[33] T. W. Lam, W. K. Sung, and H. F. Ting. Computing the unrooted maximum agreement subtree in sub-quadratic time. *Nordic journal of Computing*, 3(4):295–322, 1996.

[34] J. Li, A. L. Bogle, A. S. Klein, and M. J. Donoghue. Phylogeny and biogeography of hamamelis(hamamelidaceae). *Harvard Papers in Botany*, 5:171–178, August 2000.

[35] H.T. Lumbsch, R. Lindemuth, and I. Schmitt. Evolution of filamentous ascomycetes inferred from LSU rDNA sequence data. *Plant Biology*, 2:525–529, 2000.

[36] B. L. Lundrigan, S. Jansa, and P. K. Tucker. Phylogenetic relationships in the genus mus, based on paternally, maternally, and biparentally inherited characters. *Systematic Biology*, 51:23–53, 2002.

[37] M. Kuramochi M. Deshpande and G. Karypis. Frequent sub-structure-based approaches for classifying chemical compounds. In *3rd IEEE International Conference on Data Mining*, pages 35–42, 2003.

[38] T. Margush and F. R. McMorris. Consensus n-trees. *Bull. Math. Biol.*, 43:239–244, 1981.

[39] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm. In *Proceedings of the International Conference on Data Engineering*, 2002.

[40] C. Moh, E. Lim, and W. Ng. DTD-miner: A tool for mining DTD from xml documents. In *Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems*, Milpitas, California, June 2000.

[41] R. Nayak, R. Witt, and A. Tonev. Data mining and XML documents. In *International Conference on Internet Computing*, Nevada, USA, 2002.

[42] G. Nelson. Cladistic analysis and synthesis: Principles and definitions, with a historical note on adanson's famille des plantes (1763-1764). *Systematic Zoology*, 28:1–21, 1979.

[43] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees: Proofs. Technical Report, Leiden Institute of Advanced Computer Science, Netherlands, Jan. 2003.

[44] R. D. M. Page. COMPONENT User's Manual(Release 1.5), 1989. University of Auckland, Auckland.

[45] R. D. M. Page and E. C. Holmes. *Molecular Evolution: A Phylogenetic Approach*. Blackwell Science, 1998.

[46] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: A fast and scalable tool for data mining in massive graphs. In *Proceedings of the 8th ACM* SIGKDD *International Conference on Knowledge Discovery and Data Mining*, pages 81–90, July 2002.

[47] W. H. Piel, M. J. Donoghue, and M. J. Sanderson. Treebase: A database of phylogenetic information. In *Proceedings of the 2nd International Workshop of Species 2000*, 2000.

[48] M. J. Sanderson, M. J. Donoghue, W. H. Piel, and T. Erikson. Treebase:a prototype database of phylogenetic analyses and an interactive tool for browsing the phylogeny of life. *American Journal of Botany*, 81(6):183, 1994.

[49] C. Semple and M. Steel. A supertree method for rooted trees. *Discrete Applied Mathematics*, 105:147–158, 2000.

[50] R. Shamir and D. Tsur. Faster subtree isomorphism. *Journal of Algorithms*, 33(2):267–280, 1999.

[51] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, pages 39–52, 2002.

[52] D. Shasha, J. T. L. Wang, and S. Zhang. Unordered tree mining with applications to phylogeny. In *IEEE International Conference of Data Engineering*, Boston, U.S., April 2004.

[53] M. Steel and D. Penny. Distributions of tree comparison metrics-some new results. *Systematic biology*, 42(2):126–141, 1993.

[54] C. Stockham, L. Wang, and T. Warnow. Statistically based postprocessing of phylogenetic analysis by clustering. In *Proceedings of 10th Int'l Conference on Intelligent Systems for Molecular Biology*, pages 285–293, Edmonton, Canada, August 2002.

[55] J. T. L. Wang, H. Shan, D. Shasha, and W. H. Piel. TreeRank: A similarity measure for nearest neighbor searching in phylogenetic databases. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, Cambridge, Massachusetts, July 2003.

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

[57] J. T. L. Wang, D. Shasha, and S. Zhang. Unordered tree mining with applications to phylogeny. In *IEEE International Conference on Data Engineering*, 2004.

[58] J. T. L. Wang, K. Zhang, G. Chang, and D. Shasha. Finding approximate patterns in undirected acyclic graphs. *Pattern Recognition*, 35(2):473–483, 2002.

[59] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 146–154, 1998.

[60] K. Wang and H. Liu. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):353–371, 2000.

[61] Q. Y. Wang, J. X. Yu, and K. Wong. Approximate graph schema extraction for semi-structured data. In C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, editors, *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology*, volume 1777 of *Lecture Notes in Computer Science*, pages 302–316. Springer, March 2000.

[62] X. Wang, J. T. L. Wang, D. Shasha, B. A. Shapiro, I. Rigoutsos, and K. Zhang. Finding patterns in three-dimensional graphs: Algorithms and applications to scientific data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):731–749, July/August 2002.

[63] T. Washio and H. Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1), July 2003.

[64] Y. Xiao, J. Yao, Z. Li, and M. Dunham. Efficient data mining for maximal frequent subtrees. In *IEEE International Conference on Data Mining*, 2003.

[65] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[66] L. Yang, M. L. Lee, and W. Hsu. Efficient mining of xml query patterns for caching. In *29th International Conference on Very Large Databases*, Berlin, Germany, 2003.

[67] K. Yoshida and H. Motoda. Clip: concept learning from inference patterns. *Artificial Intelligence*, 75(1):63–92, May 1995.

[68] M. Zaki and C. Hsiao. Charm: an efficient algorithm for closed association rule mining. Technical Report, Department of Computer Science, Rensselaer Polytechnic Institute, Oct. 1999.

[69] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM* SIGKDD *International Conference on Knowledge Discovery and Data Mining*, pages 71–80, July 2002.

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

[71] S. Zhang, L. Liao, J. Tomb, and J. T. L. Wang. Clustering and classifying enzymes in metabolic pathways: Some preliminary results. In M. J. Zaki, J. T. L. Wang, and H. T.T. Toivonen, editors, *Proceedings of the 2nd ACM SIGKDD Workshop on Data Mining in Bioinformatics*, pages 19–24, 2002.

[72] S. Zhang, J. T. L. Wang, and K. Herbert. Xml query by example. *International Journal of Computational Intelligence and Applications*, 2(3):329–338, 2002.