PHYLOGENETIC SUPERTREE CONSTRUCTION USING CONSTRAINT PROGRAMMING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ÇANKAYA UNIVERSITY

BY

ALKIM ÖZAYGEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

APRIL 2006

Title of the Thesis: **Phylogenetic Supertree Construction Using Constraint Programming**

Submitted by **Alkım Özaygen**

Approval of the Graduate School of Natural and Applied Sciences, Çankaya University.

Prof. Dr. Yurdahan Güler
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Mehmet Reşit Tolun
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Mehmet Reşit Tolun
Head of Department

Examination Date : 17-04-2006

Examining Committee Members

Assoc. Prof. Dr. Veysi İşler	(METU)	
Prof. Dr. Mehmet Reşit Tolun	(Çankaya Univ.)	
Asst. Prof. Dr. Reza Hassanpour	(Çankaya Univ.)	
Dr. Abdülkadir Görür	(Çankaya Univ.)	
Dr. Ali Rıza Aşkun	(Çankaya Üniv.)	

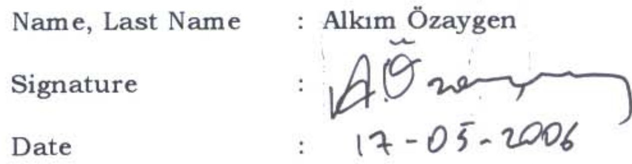
STATEMENT OF NON-PLAGIARISM

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name : Alkım Özaygen

Signature :

Date :



ABSTRACT

PHYLOGENETIC SUPERTREE CONSTRUCTION
USING CONSTRAINT PROGRAMMING

Özaygen, Alkım
M.S.c., Department of Computer Engineering
Supervisor : Prof. Dr. Mehmet Reşit Tolun

April 2006, 50 pages

In biology, a phylogenetic tree, or phylogeny, is used to show the genealogic relationships of living things. It is a codification of data about evolutionary history. The tree of life shows the path evolution took to get to the current diversity of life and can help us also to search for the genealogy of disparate living organisms.

In this thesis our aim is to provide a different approach for the construction of The Tree of Life. That is, we will propose a constraint programming solution to the decision problem of constructing a supertree that is compatible with a collection of given phylogenetic trees that share some species, which we will encode as constraint satisfaction problems.
Keywords: Phylogeny, Supertree, Constraint Programming

ÖZ

KISIT PROGRAMLAMA KULLANARAK
SÜPERAĞAÇ OLUŞTURULMASI

Özaygen, Alkım
Yükseklisans, Bilgisayar Mühendisliği Anabilim Dalı
Tez Yöneticisi : Prof. Dr. Mehmet Reşit Tolun

Nisan 2006, 50 sayfa

Biyolojide filogenetik ağaç, canlılar arası bağlantıları göstermek için kullanılır. Evrim tarihi hakkında veri kodlamasıdır. Hayat Ağacı günümüzdeki çeşitliliğe ulaşmadaki evrimin izlediği süreci gösterir ve birbirinden tamamen farklı yaşayan organizma soylarının araştırılmasında yardımcı olur.

Bu tezde amaç Hayat Ağacının oluşturulmasında farklı bir yaklaşım sunmak. Karar verme problemleri ve optimizasyon problemlerine kısıt koşul programlama çözümü öneriyoruz, ki bunu da kısıt koşul sağlama problemleri şeklinde kodlayacağız.

Anahtar Kelimeler: Filogeni, Süperağaç, Kısıt Programlama

ACKNOWLEDGMENTS

I wish to express my deepest gratitude to Assoc. Prof. Dr. Pierre Flener for his guidance, advice, criticism, encouragements and insight throughout the research. Without his enormous support and guidance this study would not be possible. Also I thank the Computing Science Division (CSD) of the Department of Information Technology (IT) at Uppsala University, Sweden, for hosting me as a visiting researcher from 3 February to 3 April 2005. The experiments in this thesis were conducted using the ILOG OPL 3.7 academic license of the ASTRA research group at Uppsala University.

I also wish to thank Prof. Dr. Mehmet Reşit Tolun and Dr. Ali Rıza Aşkun for their suggestions and comments and their encouragement during the writing of this thesis.

Many of my friends suported me during my worst time in my master program. Among all of these people, it is impossible to forget Aslı Baysuğ, Ceren Uzel, Çiğdem Aydemir, Esra Flener, Hakan Ertürk, Hakan Şapçı, Memduh Haldun Ülkenli, Nesrin Güner, Yıldız and Eftal Kurtuluş. Besides them I also wish to thank Altay Özaygen, Güzden Varinlioğlu, Hande Gözükan, Mustafa Dilaver and Tayfun Asker for their support during the writing of the thesis.

I am especially grateful to Meral and Tuna for nothing in particular but everything in general.

TABLE OF CONTENTS

STATEMENT OF NON PLAGIARISM iii

ABSTRACT iv

ÖZ v

ACKNOWLEDGMENTS vi

TABLE OF CONTENTS vii

LIST OF FIGURES ix

CHAPTERS:

1. INTRODUCTION 1
   - 1.1 Basic Phylogenetic Concepts and Definitions 2
     - 1.1.1 Trees 2
     - 1.1.2 Subtrees 4
     - 1.1.3 Rooted Triples and fans 4
   - 1.2 Constraint Programming 5
     - 1.2.1 A Short Description 5
     - 1.2.2 How it Works 6
     - 1.2.3 OPL Studio (by ILOG) 11
2. A BASIC MODEL 12
   - 2.1 Introduction 12
     - 2.1.1 Triple representation of a binary tree 13
     - 2.1.2 Supertrees 14

2.1.3 Algorithm Break Up 15

2.1.4 Algorithm OneTree 16

2.1.5 Ultrametric Trees 19

2.1.6 Min-Ultrametric 21

2.2 The Constraint Part 21

2.2.1 A Simple Example 24

3. SUPERTREE CONSTRUCTION FOR ANCESTRAL DIVERGENCE DATES AND NESTED TAXA 29

3.1 Introduction 29

3.2 Ancestral Divergence Dates 30

3.2.1 Phylogenetic Ranking 30

3.2.2 Ancestral Divergence Dates 30

3.2.3 The Constraint Part 32

3.2.4 Absolute Divergence Time 35

3.3 Nested Taxa 40

4. CONCLUSIONS 49

REFERENCES R1

LIST OF FIGURES

FIGURES

1.1 Four examples of *phylogenetic trees*. (1) and (2) are *unrooted*. (3) and (4) are *rooted*. (2) and (4) are *binary*. 2

1.2 In a binary tree each internal node has degree three with the exception of the root which has degree two. 3

1.3 An example of dendogram 4

1.4 Example of a subtree 4

1.5 A binary rooted triple $ab|c$ 5

1.6 A Search Tree for the Send More Money Problem 8

1.7 A solution to the 8Queens problem 9

1.8 The search space shrunk after making one choice (a), and making the second choice (b) 10

2.1 A binary phylogenetic tree, where species *a* and *b* are more closely related to each other than they are to species *c*. This small tree can also be represented as the rooted triple $(ab|c)$ 13

2.2 Two input trees T_1 and T_2 and a supertree 14

2.3 The new tree *T'* after applying the BreakUp algorithm. The new label set $A' = A \setminus a_1 = \{a_2, a_3, a_4, a_5\}$ 15

2.4 Applying the OneTree algorithm to two input trees ((a2, a7), a6) and ((((a1, a2),a3),(a4, a5)), a6) gives us the supertree (((((a1, a2),a3), a7),(a4, a5)), a6) 18

2.5 a) A symmetric matrix *D*. b) An ultrametric tree for matrix D 20

2.6 Construction of an ultametric tree from the matrix (shown in Figure 2.5) 20

2.7 A min-ultrametric tree and it's matrix, where the internal nodes are labelled according to their depth. 21

2.8 Two input trees ((*a2*, *a7*), *a6*) and ((((*a1*, *a2*),*a3*),(*a4*, *a5*)), *a6*) 24

2.9 13 solutions found after combining the trees in Figure 2.8 27

2.10 The four extra solutions...28

2.11 The matrix of Solution 10 and it's min-ultrametric tree. At internal node 2 we see that the tree is not bifurcating. 28

3.1 A ranked phylogenetic tree 30

3.2 An application of RANKEDTREE 32

3.3 Two rooted phylogenetic trees T_1 and T_2 and the relative divergence dates..32

3.4 Virtual branch forming without losing min-ultrametric properties, where div(*a, e*) predates div(*c, f*) 35

3.5 One of the three ranked phylogenetic trees obtained combining the trees in Figure 3.3 35

3.6 The assignment of intervals as given by the boundry value...................38

3.7 After assigning 3.0 to the internal vertice (*c*, *f*) and 3.1 to the internal vertices (*a*, *d*) and (*a*, *e*) 38

3.8 After assigning 4.0 to the internal vertice (*a*, *b*) and 4.1 to the internal vertice (*a*, *c*), (*a*, *f*), (*b*, *c*) and (*b*, *f*) 38

3.9 After assigning 4.1 to the internal vertice (*b*, *d*) and (*b*, *e*) and 0 to the internal vertice (*d*, *e*)..39

3.10 After assigning 4.1 to the internal vertice (*c*, *d*) and (*c*, *e*) 39

3.11 Finding the first solution 39

3.12 Two semi-labelled trees 40

3.13 An application of nested taxa where the input trees describe the evolution of spiders 41

3.14 One of the four possible supertree solution generated after combining the input trees in Figure 3.13 47

CHAPTER 1

INTRODUCTION

According to Pennisi today biologists have catalogued about 1.7 million of species and they estimates of the total number of species ranges from 4 to 100 million [1]. With this explosion in the amount of data in taxonomy it is no longer possible to analyze and build trees by hand. Consequently there is a growing need for new techniques to speed up this process accurately.

In recent years researchers, especially mathematicians, have shown much interest in phlogenetic tree construction and proposed new approaches in building supertrees. Until now some polynomial time algorithms have been proposed.

In this thesis we are trying to approach this problem using constraint programming, which is first proposed by Gent et al. [2].

In this chapter we discuss the basic phylogenetic concepts and five a short description of constraint programming.

In Chapter 2 we describe some algorithms used for breaking up the tree into triples and forming a tree from these triples. Then we model a simple example using OPL (*Optimization Programming Language*).

In the third chapter we extend the model proposed by Gent et al. [2] by implementing new constraints for divergence date and nested taxa informations.

1.1 Basic Phylogenetic Concepts and Definitions

1.1.1 Trees

Since this thesis is about phylogenetic trees, it is therefore appropriate to start by defining a tree.

Trees can be classified as unrooted or rooted phylogenetic trees. An *unrooted phylogenetic tree* or just *unrooted tree* is an acyclic connected graph having no internal vertices of degree two and every leaf having different label. The leaves are vertices of degree one (Figure 1.1).

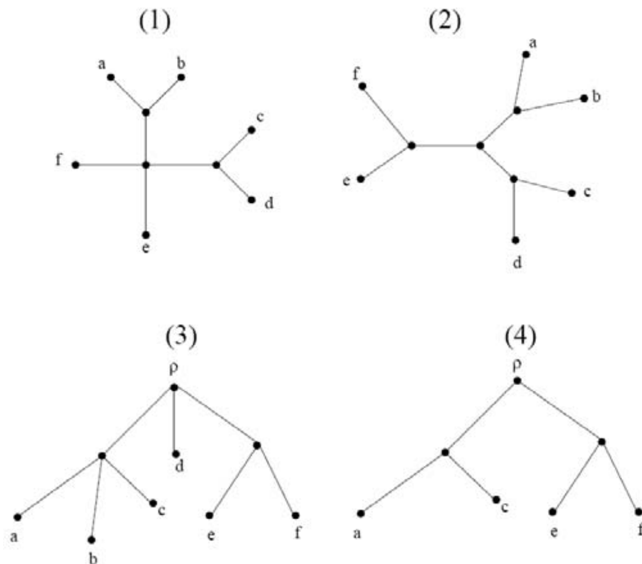


Figure 1.1: Four examples of *phylogenetic trees*. (1) and (2) are *unrooted*. (3) and (4) are *rooted*. (2) and (4) are *binary*.

A *rooted tree* on tree, on the other hand, is similar to an unrooted tree, except it has one internal vertex of degree two, which is called the root. The internal vertices of unrooted/rooted (except the root) trees can

have degree three or greater. For example a *binary phylogenetic tree*, is a tree having all internal vertices of degree three. Again the only exception is the root, which has degree two (Figure 1.2). In a fully resolved binary phylogenetic tree with n leaf nodes there are *n-1* internal nodes.

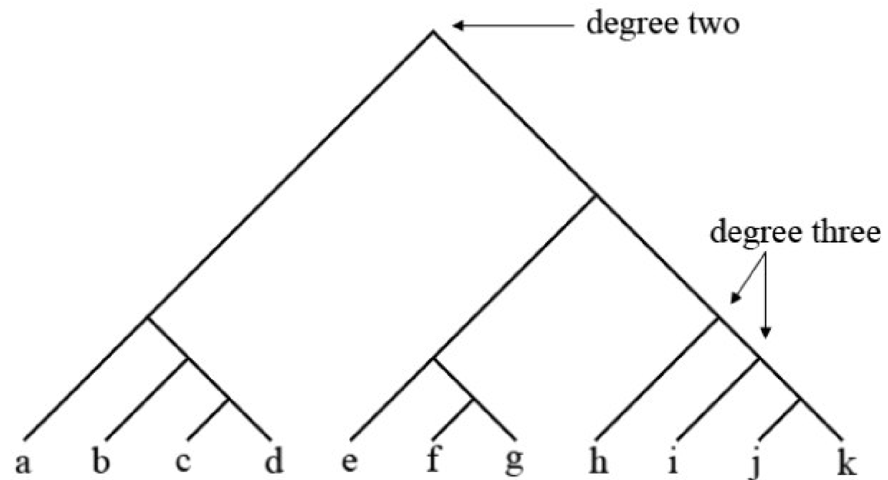


Figure 1.2: In a binary tree each internal node has degree three with the exception of the root which has degree two.

The leaves of the tree represent species. For example let $L(T)$ be the set of leaves for tree T. If T the set of trees, then we can say that $L(T)$ is the union of the leaf sets of the trees in T.

In a rooted tree we say that a vertex a is an *ancestor* of a vertex b, if the path from b to the root passes through a. We can also say that b is the *descendant* of a.

The vertices adjacent to a vertex that are descendants of the vertex are called the *children* of the vertex, and the adjacent vertex that is an ancestor is called the *parent* of that vertex. Sometimes the internal vertices of a phylogenetic tree are labelled (section *Nested Taxa*).

Rooted phylogenetic trees can be displayed with a vertical axis representing the time each branching point occurred. These diagrams are called *dendograms* (Figure 1.3).

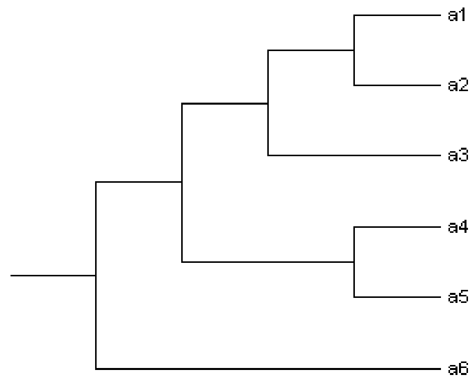


Figure 1.3: An example of dendogram.

1.1.2 Subtrees

Let T be a rooted tree and choose a vertex v in T. If we remove the edge between v and the parent of v, say p, we get two connected subgraphs. Then let v be the root of the subgraphs containing v, then this is called the subtree of T rooted at v. Briefly a subtree T' is a tree whose vertices and edges form the subsets of the vertices and edges of a given tree T. An example of a subtree is shown in Figure 1.4.

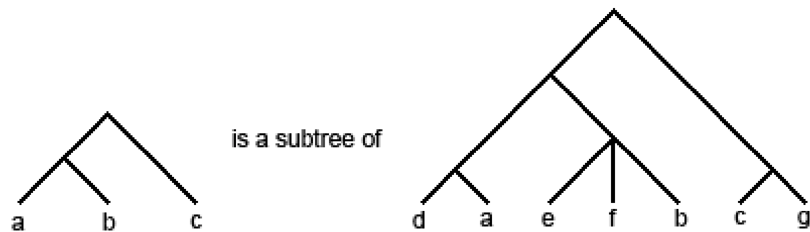


Figure 1.4: Example of a subtree.

1.1.3 Rooted Triples and Fans

For every three leaves a, b, c there are four possible rooted trees with leaf set a, b, c.

The binary rooted trees on three leaves are called rooted triples and ((ab)c) (or $ab|c$) denotes the rooted triple with a pair of leaves *a*, *b* connected to a third leaf c via the root (Figure 1.5). For a rooted triple ((ab)c) to fit a rooted tree *T*, the path from *a* to *b* does not share any vertices with the path from *c* to the root. Briefly a rooted triple is a tree with three leaves and two internal vertices.

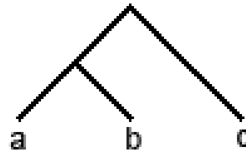


Figure 1.5: A binary rooted triple ((*ab*)*c*).

Non-binary rooted trees with three leaves are called *fan triples*. We call a fan with k leaves a *k-fan*.

1.2 Constraint Programming

1.2.1 A Short Description

Constraint programming [3][4][5] is an alternative approach to programming developed since the mid 1980s. Based on a combination of techniques dealing with reasoning and computing, it is now becoming the method of choice for modelling many types of optimization problem, in particular, those involving heterogeneous constraints and combinatorial search. It has been successfully applied in a number of fields including molecular biology, electrical engineering, operations research and numerical analysis. It has recently been identified by the *ACM* (*Association for Computing Machinery*) as a strategic directions in computing research (*http://www.acm.org/pubs/surveys/sdcr/*).

The reason for this interest in constraint programming is simple. Early programming languages, such as FORTRAN-66, closely reflected the underlying physical architecture of the computer. Since then, the major direction of programming language design has been to give the

programmer freedom to define objects and procedures which correspond to entities and operations in the application domain. Object oriented languages, in particular, provide good mechanisms for declaring program components which capture the behaviour of entities in a particular problem domain. However, traditional programming languages, including object oriented languages, provide little support for specifying *relationships* or *constraints* among programmer-defined entities. It is the role of the programmer to explicitly maintain these relationships, and to find objects which satisfy them.

However, for many applications, the important point of the problem is to model the relationships and find objects that satisfy them. For this reason, since the late 1960's, there has been interest in programming languages which allow the programmer simply to state relationships between objects. It is the role of the underlying implementation to ensure that these relationships or '*constraints*' are maintained. Such languages are called *constraint programming* languages.

1.2.2 How it Works

Let us begin by defining what a constraint is. We encounter constraints in our everyday life and these help us to take decisions. Some examples of constraints are:

- I must feed the cat before going to school.
- I must take an appointment before going to the dentist.
- I must have 1 YTL to take the bus in Ankara.

We can define a combinatorial problem as to search for a certain combination of values assigned to variables such that they satisfy a set of constraints. In some cases we need an assignment that minimises or maximises a certain entity. Such problems are called *combinatorial optimisation problems*.

Briefly Constraint Programming is a framework for solving combinatorial (optimisation) problems based on constraints.

A given problem can be specified in many different ways. It is up to the user to do this in a correct and effective way.

In a constraint program at least the following must be considered:

1. Domains (sets of values), e.g. **Z**, **R**, {1, 2, 3, 4, 5}, [1, 10), {"red", "green", "yellow"}.

2. Variables that range over domains, e.g., $x \in \{1, 2, 3, 4, 5\}$

3. Constraints that define a set of valid combinations of values for a set of variables. E.g. the set of valid combinations for $x < y$, where $x, y \in \{1, 2, 3\}$, is $\{(1, 2), (1, 3), (2, 3)\}$.

Example 1: Send More Money

Assign distinct values to the variables s, e, n, d, m, o, r, y such that the equation

$$\begin{array}{rr} & send \\ + & more \\ \hline = & money \end{array}$$

holds.

Domains: {0, ..., 9}

Variables: $s, e, n, d, m, o, r, y \in \{0, \ldots, 9\}$

Constraints:

$s \neq e$, $s \neq n$, $s \neq d$, ..., $y \neq m$, $y \neq o$, $y \neq r$,

$$\begin{aligned} & 1000 \cdot s + 100 \cdot e + 10 \cdot n + d \\ + \quad & 1000 \cdot m + 100 \cdot o + 10 \cdot r + e \\ = \quad & 1000 \cdot m + 1000 \cdot o + 100 \cdot n + 10 \cdot e + y, \end{aligned}$$

$s \neq 0$, $m \neq 0$

The above is then a constraint-based model of the problem. Here we need to find a solution, i.e., an assignment to the variables *s, e, n, d, m, o, r, y* from the domain {0, ..., 9} such that all the constraints are satisfied.

A simple and naïve way of doing this is ordinary search: Try all combinations of assignments in some systematic way until a satisfying one is found. But this is very inefficient. There are 10^8 such combinations which makes the search tree huge (Figure 1.6).

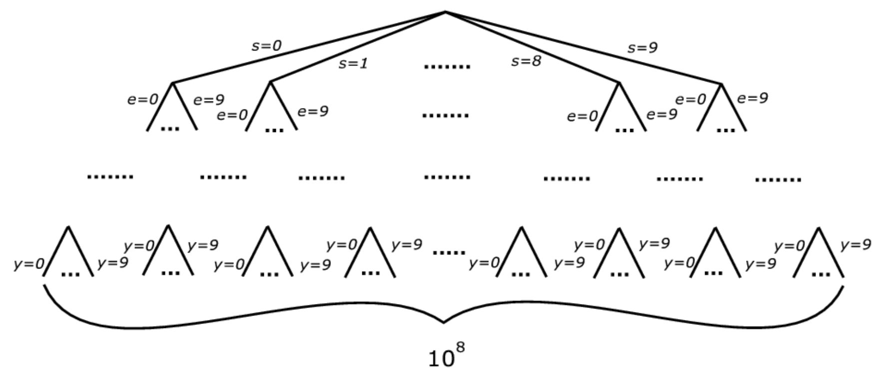


Figure 1.6: A Search Tree for the Send More Money Problem.

But in a constraint programming framework, the constraints are active entities that try to remove values from the domains of the variables and thus exclude certain combinations automatically, which shrinks the search space by pruning branches in the search tree.

Example 2: *n*Queens

Given is a chess board and 8 queens. We are trying to find a way to place the 8 queens on the chess board such that no two queens attack each other following the rules of chess. One of the possible solutions is presented in Figure 7.

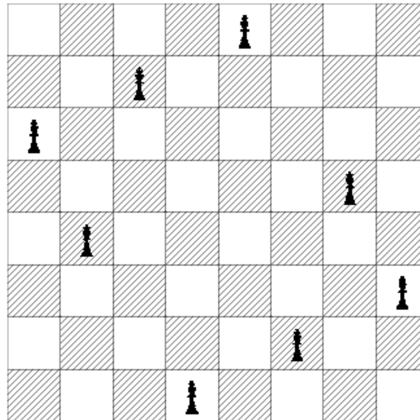


Figure 1.7: A solution to the 8Queens problem.

The problem can be generalized to find a way to place *n* queens on an *nxn* chess board with the conditions:

1. No two queens on the same row.
2. No two queens on the same column.
3. No two queens on the same NorthWest-SouthEast diagonal.
4. No two queens on the same NorthEast-SouthWest diagonal.

We can model the problem as:

Let q_i denote the row of the queen placed in column *i*. Then the constraints can be stated as follows:

Variables: $q_1, \ldots, q_n \in \{1, \ldots, n\}$

Constraints:

1. For every $i \neq j \in \{1, \ldots, n\}$: $q_i \neq q_j$
2. For every $i < j \in \{1, \ldots, n\}$: $q_i - i \neq q_j - j$
3. For every $i < j \in \{1, \ldots, n\}$: $q_i + i \neq q_j + j$

Solving the *n*Queens Problem

A complete search tree for the generalised *n*Queens problem has n^n leaves. (Since we have n variables in our model and each variable can take any out of n values.) Again, using only naïve search is not practical even for small n. With the search space shrinking ability of constraint programming this is not a very challenging problem.

Let us look to the Five-Queens Problem step by step in order to view how constraint programming shrinks the search space.

After making one choice the search space shrunk (Figure 1.8.a) and after making the second choice we'll have a few options to place the other queens (Figure 1.8.b).

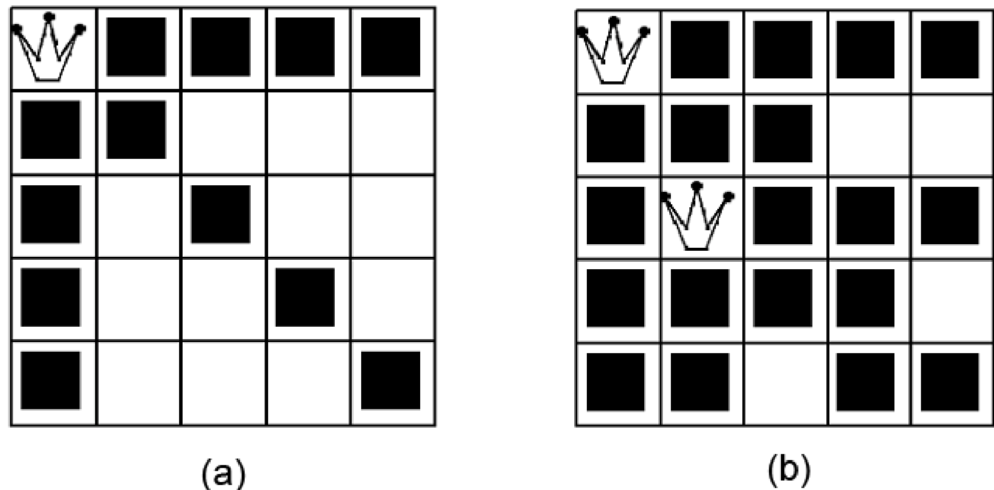


Figure 1.8: The search space shrunk after making one choice (a), and making the second choice (b).

3. OPL Studio (by ILOG)

The Optimization Programming Language (OPL)[6] developed by Pascal Van Hentenryck is a new modelling language for combinatorial optimization that simplifies the formulation and solution of combinatorial problems. The most significant dimension of OPL is the support for constraint programming, including sophisticated search specifications, logical and higher order constraints, and support for scheduling and resource allocation applications.

OPL Studio (by *ILOG*) [7] is the development environment of OPL. In addition to the traditional "edit, execute, and debug" cycle support, it provides automatic visualisations of the results, visual tools for debugging and monitoring OPL models (i.e., visualizations of the search space).

In this thesis we used ILOG OPL 3.7 for the construction of the supertrees.

CHAPTER 2

A BASIC MODEL

2.1 Introduction

Usually the evolutionary relationship of species in biological studies is represented by a rooted tree with labelled leaves where the leaves represent the species and the internal vertices represent the ancestors.

In general our aim is to combine a set of input rooted trees with labels at the leaves to get a single supertree with all the labels in the set of input trees. Here, according to Ng and Wormald [8], the set of output supertrees must "fit" the set of input trees as much as possible. Because sometimes in the input trees a leaf can be labelled with several labels. It is also possible that some input trees can contain conflicting information. In this chapter we will simply consider that each leaf or species is labelled by a single label, the labels all being different, and that conflicting information in the input trees result in no output supertree.

To obtain an output supertree, the obtained supertree must be *compatible* with the input trees. That is the topology of each input tree must be equivalent to a subtree of the obtained supertree while respecting the labelling.

In brief, we are given a set of rooted trees with labelled leaves and need to find a rooted tree *T* such that *T* contains subtrees homomorphic to all the given trees.

To obtain such supertrees we begin by breaking up each input tree into a set of triples and fans (using the *BreakUp* algorithm [8]). We then combine all the triples to produce supertrees that are compatible with the given set of input trees, of course, if the set is consistent (using the *OneTree* algorithm [8][9]).

2.1.1 Triple representation of a binary tree

In Figure 2.1 we have three species: *a*, *b* and *c*. Here *a* and *b* are more closely related to each other than they are to *c*. In a more specific way, we say that *the most common ancestor of a and b is greater than the most recent common ancestor of a and c (equally b and c).* We can notice also from the figure that the most recent common ancestor of *a* and *b* is the furthest internal node from the root, which is the most recent ancestor of *a* and *c* (equally *b* and *c*). We compare most recent common ancestors by measuring their distance from the root. That is,

$$mrca(a, b) > mrca(a, c) \quad (2.1)$$

$$\text{equally } mrca(a, b) > mrca(b, c) \quad (2.2)$$

$$mrca(a, c) = mrca(b, c) \quad (2.3)$$

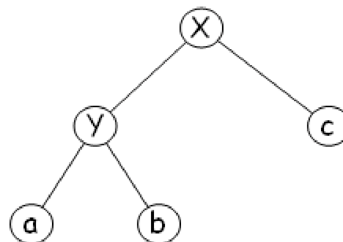


Figure 2.1: A binary phylogenetic tree, where species *a* and *b* are more closely related to each other than they are to species *c*. This small tree can also be represented as the rooted triple ((*ab*)*c*).

In Figure 2.1 we see that the most recent common ancestor of a and b is interior node Y. We can note that *mrca(a, b) = Y*. All the relations can be written as:

$$mrca(a, b) = mrca(b, a) = Y$$

$$mrca(a, c) = mrca(b, c) = X$$

We can say that

$$mrca(a, b) > mrca(a, c) \quad => \quad Y > X$$

$$mrca(a, b) > mrca(b, c) \quad => \quad Y > X$$

In triple notation the binary tree in Figure 2.1 is shown as ((*a*, *b*), *c*). This means that

$$mrca(a, b) > mrca(a, c)$$

$$mrca(a, b) > mrca(b, c)$$

$$mrca(a, c) = mrca(b, c)$$

2.1.2 Supertrees

Suppose we have a set of *k input* trees T with different, overlapping leaf sets. Let $S = \bigcup_{i=1}^{k} L(Ti)$ (i.e., the set of all species which are in at least one of the trees in *T*). So with a supertree method we take *T* as input and return a supertree with the leaf set *S* (Figure 2.2). And we will construct the possible supertrees using the triples.

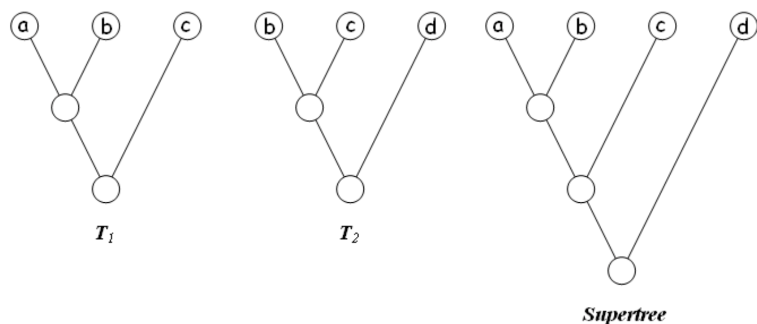
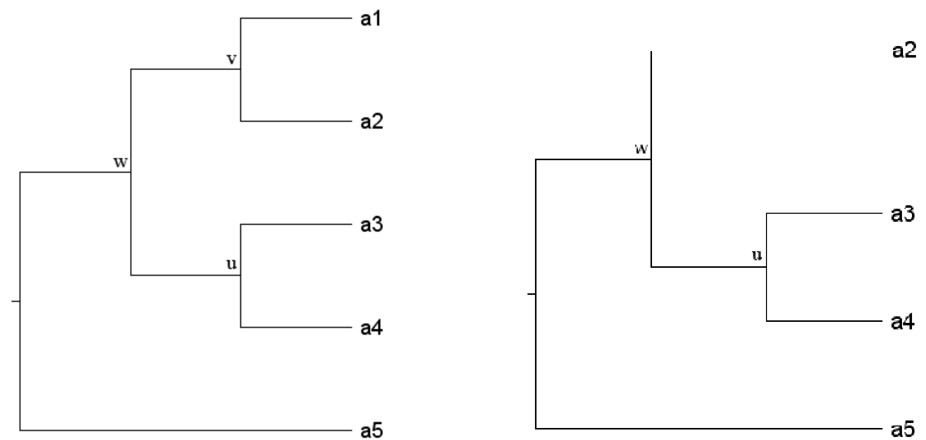


Figure 2.2: Two input trees T_1 and T_2 and a supertree.

In Figure 2.2 we can represent the two trees T_1 and T_2 as triples, that is, $((a,b),c)$ and $((b,c),d)$.

2.1.3 Algorithm Break Up

The *BreakUp* algorithm takes as input a tree T and outputs a set of rooted triples and fans, lets say G, that define that tree. The algorithm finds the deepest interior node v in the tree. Interior node $v_{\{a1,\ a2)}$ has two leaf nodes, a_1 and a_2. *BreakUp* finds the parent of $v_{\{a1,\ a2)}$, call it node w. From w we find the sibling of v, call it node u. From u find any leaf node, let's say a_3. *BreakUp* algorithms then writes out the triple $((a_1,\ a_2),\ a_3)$, deletes leaf nodes a_1 and a_2, and renames interior node v to become the new leaf node labeled as a_2. The algorithm is then applied to the reduced tree and terminates when T is reduced to a triple or less (Figure 2.3). So the idea is to start from the top of a branch and trim off a triple or fan as appropriate, and then repeat the process.



Tree: $(((a2),(a3,a4)),a5)$

Triples: $\{\ ((a1,\ a2),a3),\ ((a3,\ a4),a2),\ ((a2,\ a4),a5)\ \}$

Figure 2.3: The new tree T' after applying the BreakUp algorithm. The new label set $A' = A \setminus a_1 = \{a_2,\ a_3,\ a_4,\ a_5\}$.

In a formal way we can represent the *BreakUp* algorithm according to Ng and Wormald as:

BreakUp(A, T, G)

Input: non-empty set $A = \{a_1, a_2, \ldots, a_n\}$ of labels, tree T with label set A.

Output: set G of triples and fans.

1. If T is a triple or fan then add T to G and return.
2. Identify among the set of all internal vertices of T a vertex $v = v_{\{a1, a2)}$, say, that is minimal under the partial ordering ≤.
3. **If** $deg(v) > 3$ **then**

   add fan $a_{i1} \ldots a_{ik}$ to G, *where* $a_{i1} \ldots a_{ik}$ are the labels of the leaves attached to v.

   Let $A' = A \setminus \{a_{i3} \ldots a_{ik}\}$ and T' = tree obtained from T by deleting the leaves with labels $a_{i3} \ldots a_{ik}$.

   else

   identify a label a_{i3} where $v_{\{ai1,a13\}}$ is the immediate successor to v and add the triple $(a_1, a_2), a_3$ to G. Let $A' = A \setminus \{a_1\}$ and T' = tree obtained from T by deleting the leaves with labels a_{i1} and a_{i2}, and label the vertex v by a_{i2}.
4. BreakUp(A', T, G).

2.1.4 Algorithm OneTree

The *OneTree* algorithm takes as input a set of rooted triples G, produced by processing a number of trees with the *BreakUp* algorithm, and a set of species A and outputs a supertree that contains the species in A respecting the triples in G.

The *OneTree* algorithm constructs the tree if the input set is consistent. Inconsistent input will result in the output of the empty tree. Thus, the algorithm can be used as a test for consistency.

The algorithm starts constructing at the root level and uses the input triples and fans to divide the labels into disjoint subsets, where labels in the same subset must lie in the same branch attached to the root. For example for a triple $(ab)c$, the labels a and b must be on the same branch, whilst for a fan $(ab...e)$, the labels must all be on one branch or else each one on a different branch. If there is only one triple in R then the tree is defined by that triple. Otherwise the algorithm constructs a graph G using R as follows. Construct in G the edges $\{(a,b) \mid ((a,b),c) \in R \wedge \{a, b, c\} \subseteq S\}$. If G is a single component then the *OneTree* algorithm delivers the empty tree. If not the algorithm creates an internal node, lets say v. For each component S_i in G we collect in R_i the set of rooted triples in R with leaves in S_i. *OneTree* is then called recursively on each pair (S_i, R_i), and the resultant subtrees are then attached to v. Note that this can result in trees having interior nodes with degree greater than three.

In a formal way we can represent the *OneTree* algorithm according to Ng and Wormald as:

OneTree(G, A, v, T)

Input: set G of triples and fans, nonempty set $A = \{a_1,\dots,a_n\}$ of labels containing all labels in G, vertex v.

Output: tree T with root vertex v.

1. If n = 1, set $T = v$ with label a_1 and return. If $n = 2$, create T by attaching two new vertices to v, label them a_1 and a_2 and return.
2. Create sets $S_i = \{a_i\}$, $i = 1,\dots, n$.
3. For each triple $(a,b),c$, merge the two sets S_i *and* S_j containing a and b (if $i \neq j$).
4. **repeat**

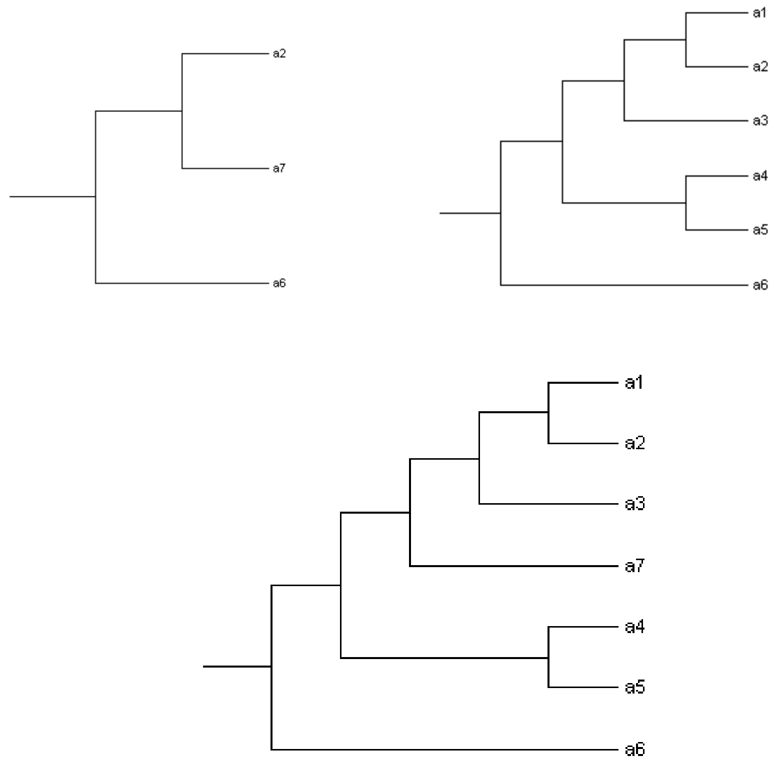
   for any fan F with at least two labels in the same set S_i, merge S_i with all sets S_j containing any label in F.

   until

   each fan with at least two labels in the same set S_i has every label in S_i.
5. If there is now only one set S_i, set $T = 0$ and return.

6. For each set S_i, create a vertex v_i, set G' := the set of triples and fans containing only those labels in S_i, and call OneTree(G', S_i, v_i, T'). If $T' = 0$, then set $T = 0$ and return. Otherwise, add T' and the edge v_i to T.

In Figure 2.4 two trees are combined to produce a supertree. First the two input trees, with the BreakUp algorithm, are broken up into rooted triples. *OneTree* algorithm takes these rooted triples, transforms them into a supertree. In Figure 2.4 we show only one of the nine possible supertrees that respect the rooted triples produced.



Supertree: (((((a1, a2), a3), a7), (a4, a5)), a6)

Triples: {((a2, a7), a6), ((a1, a2), a3), ((a2, a3), a4), ((a4, a5), a3), ((a2, a5), a6)}

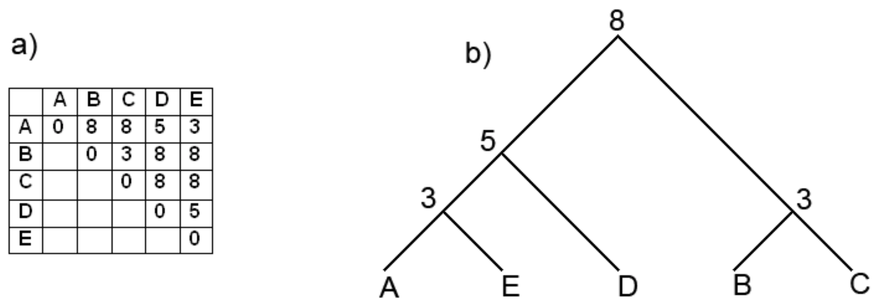
Figure 2.4: Applying the OneTree algorithm to two input trees ((a2, a7), a6) and ((((a1, a2),a3),(a4, a5)), a6) gives us the supertree (((((a1, a2),a3), a7),(a4, a5)), a6).

After having two algorithms, *BreakUp* algorithm for breaking up the input trees to rooted triples and OneTree algorithm for deciding if a supertree can be constructed from these rooted triples generated, we need an algorithm to find an optimized supertree solution to the selected problem. For the optimized supertree solution Semple and Steel [10] and more recently Page [11] have proposed some algorithms. For the decision and optimisation problem Gent et al. [2] proposed a constraint satisfaction programming approach, where they encode the decision and optimisation problem as a constraint satisfaction problems [12]. In this thesis we will use as base this constraint programming approach. So first we need to use the ultrametric matrix method to represent the trees in matrix form to operate on. Then we will extract the constraints for generating binary tree(s).

2.1.5 Ultrametric Trees

An ultrametric tree is a rooted tree where each internal node is labeled by a number and has at least two children. In the tree along the path from the root to the leaf the labels strictly decrease.

The ultrametric tree can be represented by a symmetric $n \times n$ matrix D of real numbers. So each of the leaves of the tree T is labeled by a unique row of D and each internal node of T is labeled by one entry from D. In the matrix, $D_{i,j}$ is the distance data measure. It can be viewed as the length of time since species i and j diverged. We can also say that $D_{i,j}$ represents the label of the most recent common ancestor of i and j. In Figure 2.5 we present a symmetric matrix D and its ultrametric tree.



a)

	A	B	C	D	E
A	0	8	8	5	3
B		0	3	8	8
C			0	8	8
D				0	5
E					0

Figure 2.5: a) A symmetric matrix *D.* b) An ultrametric tree for matrix D.

From Figure 2.5.a) we see that the diagonal values are all 0, since an existing species doesn't diverge from itself. If we look at the symmetric matrix and ultrametric tree we can see that the most common ancestor of *A* and *E* is labeled 3 and the most common ancestor of *A* and *C* is labeled 8. As we have said the most recent common ancestor label can be viewed as the distance data measure. We can also consider the leaves as species. So we can say that the species *A* and *E* have diverged say 3 million years ago and *A* and *C* have diverged 8 million years ago.

The tree is constructed from the matrix in the following manner:

In Figure 2.5 consider the row for A. It has distances 0 8 8 5 3 to the nodes *A, B, C, D, E.* Since 8 is the largest label we can say that it is the least common ancestor of '*A*' and there are nodes 5 and 3 on the same path as shown in the Figure 2.6.

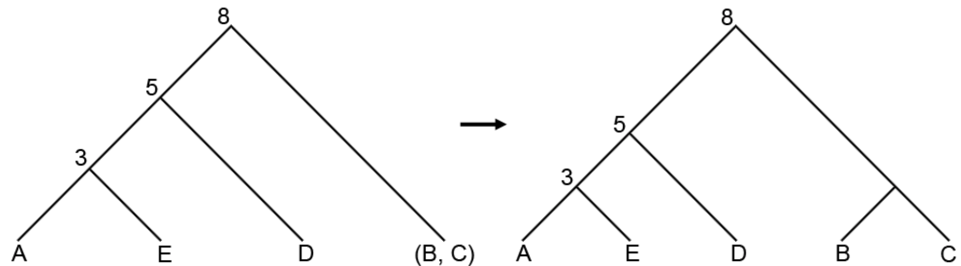
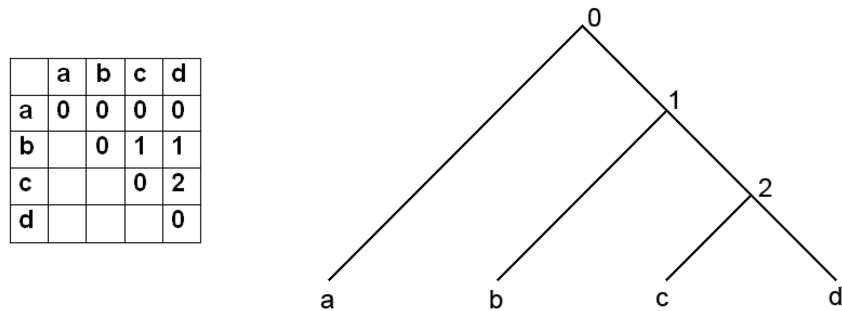


Figure 2.6: Construction of an ultametric tree from the matrix (shown in Figure 2.5).

2.1.6 Min-Ultrametric Trees

The min-ultrametric tree is a rooted tree where we label internal nodes according to their depth. Here the root has depth zero, and the depth of an internal node is one plus the label of its parent node. In Figure 2.7 we present a min-ultrametric tree, where the internal nodes are labelled with their depth.



	a	b	c	d
a	0	0	0	0
b		0	1	1
c			0	2
d				0

Figure 2.7: A min-ultrametric tree and its matrix, where the internal nodes are labelled according to their depth.

2.2 The Constraint Part

Having defined some basic tree concept and explained some algorithms to form a supertree we can now pass to the constraint part of the job. Here we will present a constraint encoding which provides a unique representation of trees. Here the basic idea is to encode the *depth* of the most recent common ancestors in the tree. We know that in a fully resolved binary phylogenetic tree with n leaf nodes (species) there are $n-1$ internal nodes. So we have a symmetric $n \times n$ two dimensional array D where each variable $D_{i,j}$ takes a value in the range 0 to $n-2$ and the value assigned represent the depth of the most recent common ancestor of leaf nodes i and j. Since the matrix is symmetric, $D_{i,j} = D_{j,i}$ and we set arbitrarily the diagonal values to 0 ($D_{i,i} = 0$).

To form the matrix from the triples of the form $((a, b), c)$ we will encode the constraint as:

$$\text{triple}(a, b, c) \equiv [(D_{a,c} = D_{b,c}) \wedge (D_{a,b} > D_{b,c}) \wedge (D_{a,b} > D_{a,c})] \quad (2.4)$$

This constraint tells us that

1. The most recent common ancestor of a and c ($D_{a,c}$) must be equal to the most recent common ancestor of b and c ($D_{b,c}$).

2. The most recent common ancestor of a and b ($D_{a,b}$) must be greater to the most recent common ancestor of b and c ($D_{b,c}$).

3. The most recent common ancestor of a and b ($D_{a,b}$) must be greater to the most recent common ancestor of a and c ($D_{a,c}$).

To guarantee that D is a min-ultrametric matrix, we have to encode the constraint as:

$$\forall a \in \{1..\, n-1\}.\ \forall b \in \{a+1..n-1\}.\ \forall c \in \{b+1..n\}$$

$$(triple(a, b, c) \vee triple(b, c, a) \vee triple(c, a, b)) \quad (2.5)$$

This constraint tells us that for every a, b and c, where a, b and c are different, there must be a triple $((a, b), c)$ or $((b, c), a)$ or $((c, a), b)$.

This also tells us that in the variables $D_{a,b}$, $D_{b,c}$, and $D_{a,c}$ the minimum value must be shared by two of them and not the third one.

But these two constraints are not enough. We know that the internal nodes along the path from the root to the leaf have to increase and do not contain any gaps. For example a path 0, 1, 3, 4 is an unwanted solution. Because we have a gap at depth 2. However a solution like 0, 1, 2, 3 would be a legal sequence. So we have to insert another constraint that will get rid of the solutions which have gaps. For example if there is some d where $D_{c,d} = 2$, then there have to be some values like a and b where $D_{c,b} = 1$ and $D_{c,a} = 0$. That is the depths let's say from the root to a leaf have to be simply 0, 1, 2, ... So our new constraint is as:

$$\forall a\ \forall b\ [(D_{a,b} = i \wedge i > 0) \rightarrow \exists\, c\ (D_{a,c} = i - 1) \quad (2.6)$$

Using this encoding the number of consistent possible instantiations of variables in D is $(2n-2)! / (2^{n-1}(n-1)!)$ [13][14].

Using the set of triples R and instantiating D we can then start constructing the tree. With this encoding we obtain the same results as the algorithm *OneTree*. In order to find all solutions we allow our solving procedure to backtrack in the search space whenever it finds a solution, and to continue on for the next solution.

With the constraint written as

$$\text{triple}(a, b, c) \equiv [(D_{a,c} = D_{b,c}) \wedge (D_{a,b} > D_{b,c}) \wedge (D_{a,b} > D_{a,c})] \quad (2.7)$$

we can only find solutions for a binary tree. To generate trees with internal nodes having more than two children we can change this constraint to

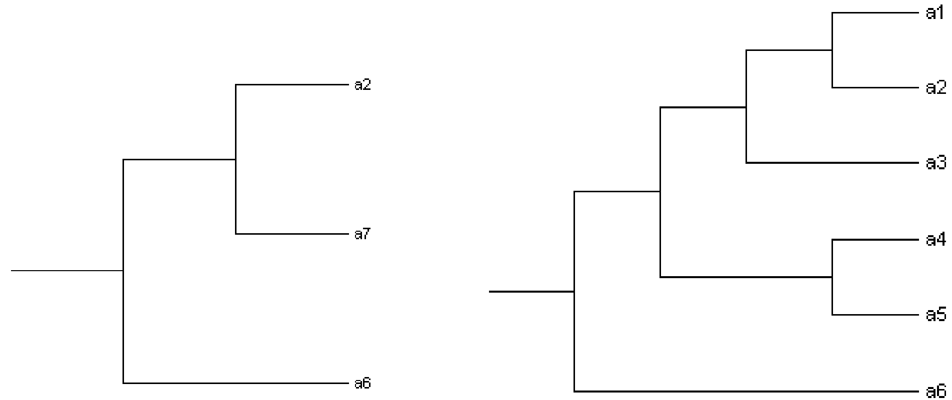
$$\text{fantriple}(a, b, c) \equiv [(D_{a,c} = D_{b,c}) \wedge (D_{a,b} \geq D_{b,c}) \wedge (D_{a,b} \geq D_{a,c})] \quad (2.8)$$

to allow the fans.

So far, our constraint model is the same as the one of Gent et al. [2].

2.2.1. A Simple Example

Let's solve the simple example of Figure 2.8.



Trees: ((*a2*, *a7*), *a6*)
((((*a1*, *a2*),*a3*),(*a4*, *a5*)), *a6*)
Triples: {((*a2*, *a7*), *a6*),
((*a1*, *a2*), *a3*), ((*a2*, *a3*), *a4*), ((*a4*, *a5*), *a3*), ((*a2*, *a5*), *a6*)}

Figure 2.8: Two input trees ((*a2*, *a7*), *a6*) and ((((*a1*, *a2*),*a3*),(*a4*, *a5*)), *a6*).

Here we have two input trees ((*a2*, *a7*), *a6*) and ((((*a1*, *a2*),*a3*), (*a4*, *a5*)), *a6*).

In OPL our encoding is as follows:

```
enum Species ...;

range intNodes 0..card(Species)-2;

var intNodes D[Species, Species];

struct Triple {Species i; Species j; Species k;};
{Triple} Triples = ...;

solve {
  forall(ordered i, j in Species)
    D[i, j] = D[j, i];

  forall(i in Species)
    D[i,i]=0;

  forall(ordered i, j in Species)
    D[i, j] > 0   => sum(k in Species: k<>i) (D[i, k] =
D[i, j] - 1) > 0;

  forall (triple in Triples)
    D[triple.i,triple.j] > D[triple.i,triple.k] =
   D[triple.j,triple.k];

  forall (ordered i, j, k in Species)
```

```
    D[i, j] = D[i, k] < D[j, k] \/ D[i, j] = D[j, k] < D[i, k] \/
D[i, k] = D[j, k] < D[i, j];
};

display(ordered i,j in Species) D[i,j];
```

and the data file as:

```
Species = {a, b, c, d, e, f, g};

Triples = { <a, b, c> <d, e, c> <c, b, e> <e, b, f> <a, g, f>

};
```

In the model, we first declare the enumeration, called `Species`, of leaf species of the given trees; as indicated by the "..." annotation, it is to be imported from a data file. Then, we declare the range, called `intNodes`, of the depths of the internal nodes to be 0,...,n-2 where n is the size of `Species`. Indeed, the most unbalanced phylogenetic tree, namely the caterpillar tree, has all its n-1 internal nodes on a path, hence the depths 0,...,n-2 suffice to label them.

The constraint

```
forall(ordered i, j  in Species)
      D[i, j] = D[j, i];
```

is used in order to enforce the symmetry, where the most recent common ancestor of species i and j must be the same recent common ancestor of species j and i. Also we have to fix $D_{i,i}$ to an abitrary value between 0 and n-1. Here we chose 0 by stating

```
forall(i in Species)
  D[i,i]=0;
```

The constraint

```
forall(ordered i,j in Species)
      D[i, j] > 1   => sum(k in Species: k<>i) (D[i, k] =
      D[i, j] - 1) > 0;
```

tells that if the most recent common ancestor of species i and j is greater than one, then it must have a child one minus of its value. This constraint prevent a gap forming along side the branches.

The constraint

```
forall (triple in Triples)
      D[triple.i,triple.j] > D[triple.i,triple.k] =
D[triple.j,triple.k];
```

says that the matrix must be formed according to every triple listed. Where the most recent common ancestor of i, j ($D_{i,j}$) must be equal to the most recent common ancestor of i, k ($D_{i,k}$) which are less than the most common ancestor of j, k ($D_{j,k}$).

The constraint

```
forall (ordered i, j, k in Species)
      D[i, j] = D[i, k] < D[j, k] \/ D[i, j] =
D[j, k] < D[i, k] \/ D[i, k] = D[j, k] < D[i, j];
```

is a representation of, for the species i, j and k, all being different

1. The most recent common ancestor of i, j ($D_{i,j}$) must be equal to the most recent common ancestor of i, k ($D_{i,k}$) which are less than the most common ancestor of j, k ($D_{j,k}$).
   or
2. The most recent common ancestor of i, j ($D_{i,j}$) must be equal to the most recent common ancestor of j, k ($D_{j,k}$) which are less than the most common ancestor of i, k ($D_{i,k}$).
   or
3. The most recent common ancestor of i, k ($D_{i,k}$) must be equal to the most recent common ancestor of j, k ($D_{j,k}$) which are less than the most common ancestor of i, j ($D_{i,j}$).

This constraint tells us that for every species i, j and k, where i, j and k are different, there must be a triple $((i, j), k)$ or $((j, k), i)$ or $((k, i), j)$.

This also tells us that in the variables $D_{i,j}$, $D_{j,k}$, and $D_{i,k}$ the minimum value must be shared by two of them and not the third one.

With this constraint we generate the solutions without fans. After searching the solutions we get nine solutions. The solutions are shown in Figure 2.9.

Solution [1]	Solution [2]	Solution [3]	Solution [4]	Solution [5]	Solution [6]
D[a1,a2] = 4	D[a1,a2] = 4	D[a1,a2] = 4	D[a1,a2] = 4	D[a1,a2] = 4	D[a1,a2] = 4
D[a1,a3] = 3	D[a1,a3] = 3	D[a1,a3] = 3	D[a1,a3] = 3	D[a1,a3] = 3	D[a1,a3] = 3
D[a1,a4] = 2	D[a1,a4] = 2	D[a1,a4] = 2	D[a1,a4] = 2	D[a1,a4] = 2	D[a1,a4] = 2
D[a1,a5] = 2	D[a1,a5] = 2	D[a1,a5] = 2	D[a1,a5] = 2	D[a1,a5] = 2	D[a1,a5] = 2
D[a1,a6] = 1	D[a1,a6] = 1	D[a1,a6] = 1	D[a1,a6] = 1	D[a1,a6] = 1	D[a1,a6] = 1
D[a1,a7] = 2	D[a1,a7] = 2	D[a1,a7] = 3	D[a1,a7] = 4	D[a1,a7] = 5	D[a1,a7] = 2
D[a2,a3] = 3	D[a2,a3] = 3	D[a2,a3] = 3	D[a2,a3] = 3	D[a2,a3] = 3	D[a2,a3] = 3
D[a2,a4] = 2	D[a2,a4] = 2	D[a2,a4] = 2	D[a2,a4] = 2	D[a2,a4] = 2	D[a2,a4] = 2
D[a2,a5] = 2	D[a2,a5] = 2	D[a2,a5] = 2	D[a2,a5] = 2	D[a2,a5] = 2	D[a2,a5] = 2
D[a2,a6] = 1	D[a2,a6] = 1	D[a2,a6] = 1	D[a2,a6] = 1	D[a2,a6] = 1	D[a2,a6] = 1
D[a2,a7] = 2	D[a2,a7] = 2	D[a2,a7] = 3	D[a2,a7] = 5	D[a2,a7] = 4	D[a2,a7] = 2
D[a3,a4] = 2	D[a3,a4] = 2	D[a3,a4] = 2	D[a3,a4] = 2	D[a3,a4] = 2	D[a3,a4] = 2
D[a3,a5] = 2	D[a3,a5] = 2	D[a3,a5] = 2	D[a3,a5] = 2	D[a3,a5] = 2	D[a3,a5] = 2
D[a3,a6] = 1	D[a3,a6] = 1	D[a3,a6] = 1	D[a3,a6] = 1	D[a3,a6] = 1	D[a3,a6] = 1
D[a3,a7] = 2	D[a3,a7] = 2	D[a3,a7] = 4	D[a3,a7] = 3	D[a3,a7] = 3	D[a3,a7] = 2
D[a4,a5] = 3	D[a4,a5] = 3	D[a4,a5] = 3	D[a4,a5] = 3	D[a4,a5] = 3	D[a4,a5] = 4
D[a4,a6] = 1	D[a4,a6] = 1	D[a4,a6] = 1	D[a4,a6] = 1	D[a4,a6] = 1	D[a4,a6] = 1
D[a4,a7] = 3	D[a4,a7] = 4	D[a4,a7] = 2	D[a4,a7] = 2	D[a4,a7] = 2	D[a4,a7] = 3
D[a5,a6] = 1	D[a5,a6] = 1	D[a5,a6] = 1	D[a5,a6] = 1	D[a5,a6] = 1	D[a5,a6] = 1
D[a5,a7] = 4	D[a5,a7] = 3	D[a5,a7] = 2	D[a5,a7] = 2	D[a5,a7] = 2	D[a5,a7] = 3
D[a6,a7] = 1	D[a6,a7] = 1	D[a6,a7] = 1	D[a6,a7] = 1	D[a6,a7] = 1	D[a6,a7] = 1

Solution [7]	Solution [8]	Solution [9]
D[a1,a2] = 5	D[a1,a2] = 5	D[a1,a2] = 5
D[a1,a3] = 3	D[a1,a3] = 4	D[a1,a3] = 4
D[a1,a4] = 2	D[a1,a4] = 2	D[a1,a4] = 3
D[a1,a5] = 2	D[a1,a5] = 2	D[a1,a5] = 3
D[a1,a6] = 1	D[a1,a6] = 1	D[a1,a6] = 1
D[a1,a7] = 4	D[a1,a7] = 3	D[a1,a7] = 2
D[a2,a3] = 3	D[a2,a3] = 4	D[a2,a3] = 4
D[a2,a4] = 2	D[a2,a4] = 2	D[a2,a4] = 3
D[a2,a5] = 2	D[a2,a5] = 2	D[a2,a5] = 3
D[a2,a6] = 1	D[a2,a6] = 1	D[a2,a6] = 1
D[a2,a7] = 4	D[a2,a7] = 3	D[a2,a7] = 2
D[a3,a4] = 2	D[a3,a4] = 2	D[a3,a4] = 3
D[a3,a5] = 2	D[a3,a5] = 2	D[a3,a5] = 3
D[a3,a6] = 1	D[a3,a6] = 1	D[a3,a6] = 1
D[a3,a7] = 3	D[a3,a7] = 3	D[a3,a7] = 2
D[a4,a5] = 3	D[a4,a5] = 3	D[a4,a5] = 4
D[a4,a6] = 1	D[a4,a6] = 1	D[a4,a6] = 1
D[a4,a7] = 2	D[a4,a7] = 2	D[a4,a7] = 2
D[a5,a6] = 1	D[a5,a6] = 1	D[a5,a6] = 1
D[a5,a7] = 2	D[a5,a7] = 2	D[a5,a7] = 2
D[a6,a7] = 1	D[a6,a7] = 1	D[a6,a7] = 1

Figure 2.9: 13 solutions found after combining the trees in Figure 2.8.

The constraint

```
forall (ordered i, j, k in Species)
      D[i, j] = D[i, k] < D[j, k] \/ D[i, j] =
D[j, k] < D[i, k] \/ D[i, k] = D[j, k] < D[i, j];
```

was for generating binary trees. If we want that this model generates solutions comprising supertrees with fans, we should change this contstraint to

```
forall (ordered i, j, k in Species)
     D[i, j] = D[i, k] <= D[j, k] \/ D[i, j] = D[j, k] <=
 D[i, k] \/ D[i, k] = D[j, k] <= D[i, j];
```

where we will get 13 solutions. The four extra solutions with fans are shown in Figure 2.10.

Solution [10]	Solution [11]	Solution [12]	Solution [13]
D[a1,a2] = 4	D[a1,a2] = 4	D[a1,a2] = 4	D[a1,a2] = 4
D[a1,a3] = 3	D[a1,a3] = 3	D[a1,a3] = 3	D[a1,a3] = 3
D[a1,a4] = 2	D[a1,a4] = 2	D[a1,a4] = 2	D[a1,a4] = 2
D[a1,a5] = 2	D[a1,a5] = 2	D[a1,a5] = 2	D[a1,a5] = 2
D[a1,a6] = 1	D[a1,a6] = 1	D[a1,a6] = 1	D[a1,a6] = 1
D[a1,a7] = 2	D[a1,a7] = 2	D[a1,a7] = 3	D[a1,a7] = 4
D[a2,a3] = 3	D[a2,a3] = 3	D[a2,a3] = 3	D[a2,a3] = 3
D[a2,a4] = 2	D[a2,a4] = 2	D[a2,a4] = 2	D[a2,a4] = 2
D[a2,a5] = 2	D[a2,a5] = 2	D[a2,a5] = 2	D[a2,a5] = 2
D[a2,a6] = 1	D[a2,a6] = 1	D[a2,a6] = 1	D[a2,a6] = 1
D[a2,a7] = 2	D[a2,a7] = 2	D[a2,a7] = 3	D[a2,a7] = 4
D[a3,a4] = 2	D[a3,a4] = 2	D[a3,a4] = 2	D[a3,a4] = 2
D[a3,a5] = 2	D[a3,a5] = 2	D[a3,a5] = 2	D[a3,a5] = 2
D[a3,a6] = 1	D[a3,a6] = 1	D[a3,a6] = 1	D[a3,a6] = 1
D[a3,a7] = 2	D[a3,a7] = 2	D[a3,a7] = 3	D[a3,a7] = 3
D[a4,a5] = 3	D[a4,a5] = 3	D[a4,a5] = 3	D[a4,a5] = 3
D[a4,a6] = 1	D[a4,a6] = 1	D[a4,a6] = 1	D[a4,a6] = 1
D[a4,a7] = 2	D[a4,a7] = 3	D[a4,a7] = 2	D[a4,a7] = 2
D[a5,a6] = 1	D[a5,a6] = 1	D[a5,a6] = 1	D[a5,a6] = 1
D[a5,a7] = 2	D[a5,a7] = 3	D[a5,a7] = 2	D[a5,a7] = 2
D[a6,a7] = 1	D[a6,a7] = 1	D[a6,a7] = 1	D[a6,a7] = 1

Figure 2.10: The four extra solutions.

The min-ultrametric tree of Solution 10 with fans is shown in Figure 2.11.

	a1	a2	A3	a4	a5	a6	a7
a1	0	4	3	2	2	1	2
a2	4	0	3	2	2	1	2
a3	3	3	0	2	2	1	2
a4	2	2	2	0	3	1	2
a5	2	2	2	3	0	1	2
a6	1	1	1	1	1	0	1
a7	2	2	2	2	2	1	0

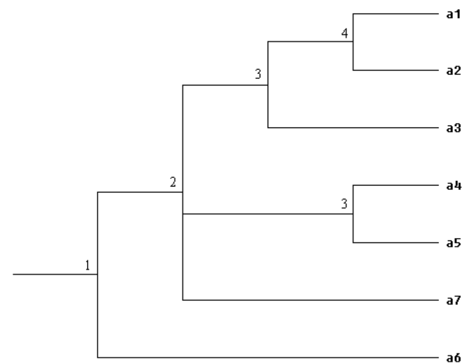


Figure 2.11: The matrix of Solution 10 and its min-ultrametric tree. At internal node 2 we see that the tree is not bifurcating.

CHAPTER 3

SUPERTREE CONSTRUCTION FOR ANCESTRAL DIVERGENCE DATES AND NESTED TAXA

3.1 Introduction

Until now we have shown a basic model with min-ultrametric supertree construction. In this chapter we will try to show some new constraints in order to extend the allowable information that can be used for phylogenetic inference. In the first part we will include in the input ancestral divergence dates which may be either relative or absolute. For example, in this group the input could include information such as whether one particular divergence event on one side of a tree occurred before or after a divergence event on the other side of the tree, or actual time estimates of certain divergence events. In the second part we will deal with nested taxa. We will use input rooted trees in which some of the interior vertices as well as all of their leaves are labelled, which will allow the inclusion of nested taxa in the input. For the description of the problems we used Bryant, Semple and Steel's works [15][16].

3.2 Ancestral Divergence Dates

3.2.1 Phylogenetic Ranking

A phylogenetic ranking helps us to arrange the tree's interior vertices according to the order of the speciation events. For example let T be a rooted phylogenetic tree. We will create a rank function for this tree to arrange the order in the set of interior vertices (all positive integers), call it V_{iv}. Here for all $v_1, v_2 \in V_{iv}$, we say that $r(v_1) < r(v_2)$ if v_2 is a proper descendant of v_1. We say that the pair (T, r) is a *ranked phylogenetic tree.* So the ranking of the interior vertices of the rooted phylogenetic tree is an ordering of the speciation events. An example of a ranked phylogenetic tree is illustrated in Figure 3.1.

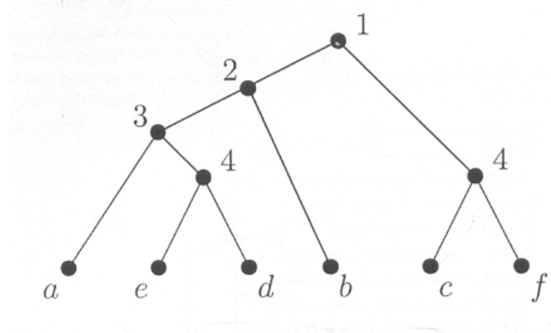


Figure 3.1: A ranked phylogenetic tree.

For example the speciation event of species c and d took place before the speciation event of species a and b. Here two different interior vertices $((e, d)$ and $(c, f))$ may be assigned the same positive integer which means that there is no particular ordering on the associated speciation events.

3.2.2 Ancestral Divergence Dates

Here we will briefly describe the RANKEDTREE algorithm that incorporates relative divergence times developed by Semple et al. [15]. Then we will proceed to our constraint programming approach.

The RANKEDTREE algorithm is an extension of the BUILD approach, developed for other purposes in 1981 [17], which outputs a tree precisely if the input collection satisfies a particular compatibility criterion. So the RANKEDTREE algorithm takes as input the rooted phylogenetic trees as well as the information detailing the order of the speciation events that occurred.

A relative divergence date "div(*c, d*) predates div(*a, b*)" tells us that if *a, b, c, d* are leaf labels of *T*, then the rank assigned to the interior vertex of *T* corresponding to the most recent common ancestor of *c* and *d* is less than the rank assigned to the interior vertex of *T* corresponding to the most recent common ancestor of *a* and *b*.

If we look at the ranked phylogenetic tree shown in Figure 3.2 we can see that it preserves the relative divergence date 'div(*e, b*) predates div(*c, f*)'. A collection *P* of rooted phylogenetic trees and a collection *D* of relative divergence dates are *compatible* if there is a ranked phylogenetic tree *T* such that the discrete topology of *T* displays each of the trees in *P* and the ranking of the interior vertices of *T* preserves all of the relative divergence dates in *D*.

To illustrate the ranking in a tree let's take the phylogenetic cat family tree example shown in Figure 3.2(a) and (b). Here each species name is abbreviated to 3-letters labels and the branch lengths are represented by the ranking of the internal vertices (which does not reflect the real time). Here we can notice that only the three species 'LPA', 'PON', and 'CCR' are present in both trees. A resulting supertree obtained from combining these two trees is shown in Figure 3.2(c).

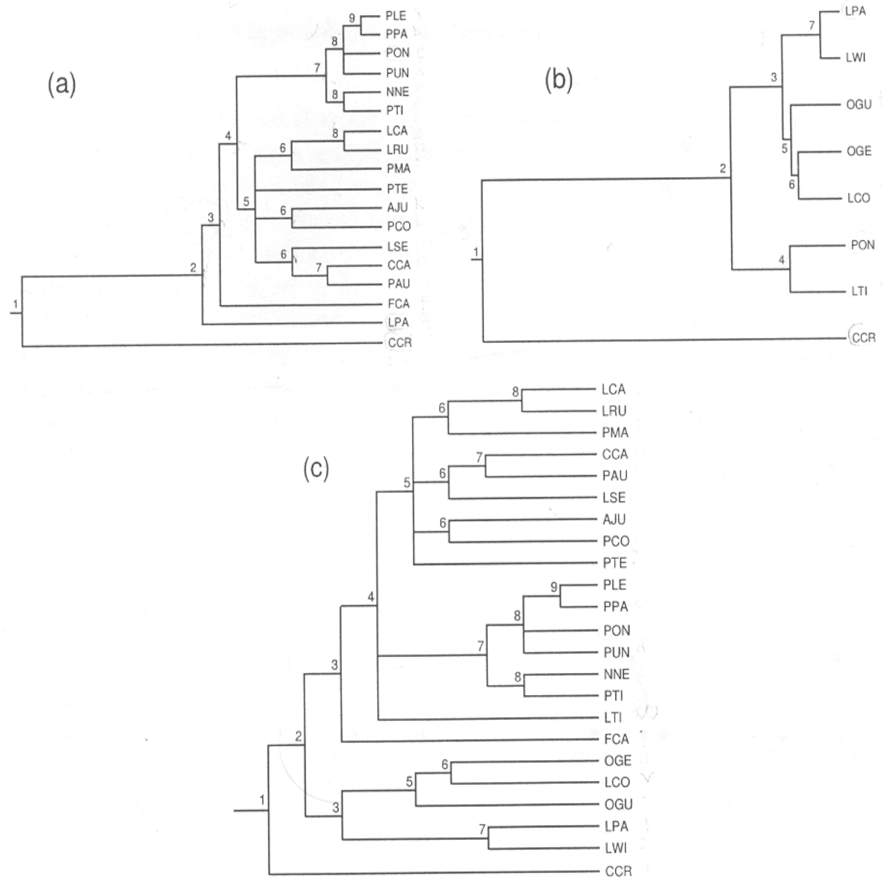


Figure 3.2: An application of RANKEDTREE.

3.2.3 The Constraint Part

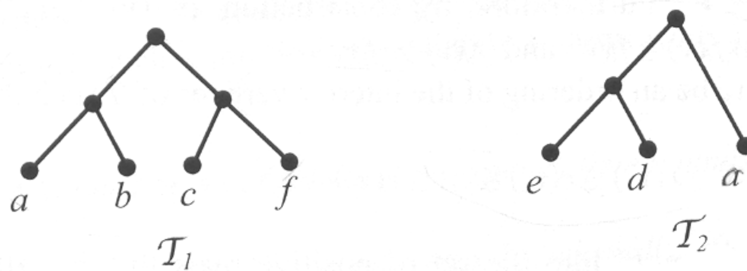


Figure 3.3: Two rooted phylogenetic trees T_1 and T_2 and the relative divergence dates.

div(a, e) predates div(c, f)
and div(a, b)predates div(a, d).

In the example of Figure 3.3, we will try to combine these trees with their relative divergence time using constraint programming.

Below is the model (.mod file) and the data (.dat) for Figure 3.3.

The model file:

```
enum Species ...;

range intNodes 0..card(Species)-2;

var intNodes D[Species, Species];

struct Triple {Species i; Species j; Species k;};
{Triple} Triples = ...;

struct Predate {Species i; Species j; Species k; Species l;};
{Predate} Predates = ...;

solve {
  forall(ordered i, j in Species)
    D[i, j] = D[j, i];

  forall(i in Species)
    D[i, i] = 0;

  forall(ordered i,j in Species
     D[i,j]>0 =>  sum(k in Species: k<>i & k<>j)
    (D[i,k]=D[i,j]-1) > 0  \/ sum(p in Predates: p.k=i \/ p.l=i)
    (D[p.i,p.j]=D[i,j]-1) > 0 ;

  forall(triple in Triples)
    D[triple.i, triple.j] > D[triple.i, triple.k] =
   D[triple.j, triple.k];

  forall(ordered i, j, k in Species)
    D[i, j] = D[i, k] <  D[j, k] \/ D[i, j] = D[j, k] <
   D[i, k] \/ D[i, k] = D[j, k] <  D[i, j];

//  D[i, j] = D[i, k] <= D[j, k] \/ D[i, j] = D[j, k] <=
// D[i, k] \/ D[i, k] = D[j, k] <= D[i, j];  // with fans

  forall(predate in Predates)
    D[predate.i, predate.j] < D[predate.k, predate.l];
};

search{
generate(D);
  forall(i in Species)
    forall(j in Species: i<j & not bound(D[i,j]))
      tryall(k in [dmin(D[i,j])..dmax(D[i,j])]) D[i,j]=k;
};

display(ordered i, j in Species) D[i,j];
```

The data file:

```
Species = {a, b, c, d, e, f};

Triples = {<a, b, c> <c, f, b> <e, d, a>};

Predates = {
   <a, e, c, f>
   <a, b, a, d>
} ;
```

We can notice that in this example we split the model and the data part to two different files. Here our aim is to find a way to represent this new problem with a minimum syntactic change. So while holding most of our constraints from the the previous example, we have to insert a new and simple constraint for this problem.

In this model we first declared a record `Predate` consisting of four fields `i`, `j`, `k` and `l` using `Predates` in the data file with

```
struct Predate {Species i; Species j; Species k; Species l;};
{Predate} Predates = ...;
```

and using a new constraint

```
   forall(predate in Predates)
    D[predate.i, predate.j] < D[predate.k, predate.l];
```

which states that the divergence of species *i* and *j* occured before the divergence of species *k* and *l*. For example from the data file on our example, we can say that the divergence of species *a* and *e* occured before the divergence of species *c* and *f*.

Also in this model we used the constraint

```
forall(ordered i,j in Species)
D[i,j]>0 =>  sum(k in Species: k<>i & k<>j)
  (D[i,k]=D[i,j]-1) > 0  \/ sum(p in Predates: p.k=i \/ p.l=i)
  (D[p.i,p.j]=D[i,j]-1) > 0 ;
```

in place of

```
forall(ordered i,j in Species)
    D[i, j] > 1   => sum(k in Species: k<>i) (D[i, k] =
D[i, j] - 1) > 0;
```

With this constraint our intention was again to prevent gap forming along the branch. But this time using the predates data. This constraint tells that if the most recent common ancestor of species i and j is greater than one, then it must have a child one minus of its value or it must have a child having one minus the predate value. For example here for the most recent common ancestor for the species c and f $D_{[c,\ f]}$ must be predated by the most recent common ancestor of species a and e $D_{[a,\ e]}$. So in this example if $D_{[b,\ e]}$ is 2 then $D_{[c,\ f]}$ cannot be 4. Where this constraint prevent any gap along the branch. In other words there are some virtual branches forming, and their internal nodes increasing by one along these virtual branches (shown in Figure 3.4).

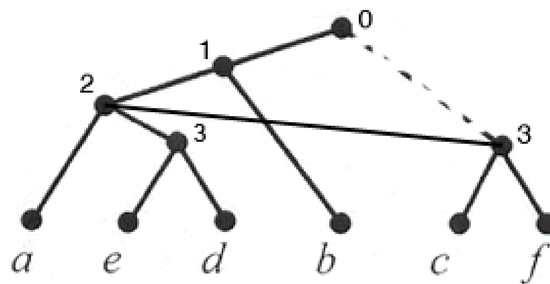


Figure 3.4: Virtual branch forming without losing min-ultrametric properties, where div(a, e) predates div(c, f).

So running this model we find the solution shown in Figure 3.5 below.

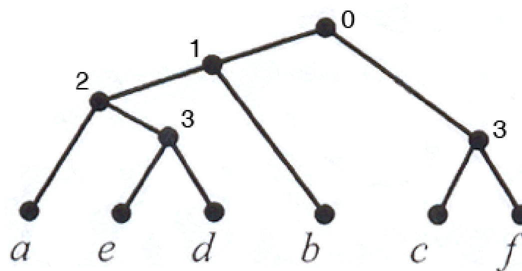


Figure 3.5: One of the three ranked phylogenetic trees obtained combining the trees in Figure 3.3.

3.2.3 Absolute Divergence Time

Now let's go further and insert some absolute divergence time bounds to our internal vertices. The divergence time bound for species a

and b is a lower or an upper bound denoted by $l(a, b)$ and $u(a, b)$, respectively on the unit of time ago that a and b diverged. These species can also have both lower and upper bounds, where obviously $l(a, b) < u(a, b)$. If there is no upper or lower bound on the divergence time of a and b, these will be taken obviously as: $l(a, b) = 0$ and $u(a, b) = \infty$.

Now let's return to our example of Figure 3.3 and improve it by adding some divergence time bounds as follows:

$$l(a, d) = 1 \text{ and } u(a, d) = 3.5$$

$$l(a, b) = 4 \text{ and } u(a, b) = 6$$

$$l(c, f) = 3 \text{ and } u(c, f) = 5$$

Below is the model (.mod file) and the data (.dat) for this example.

The model:

```
enum Species=...;

range intNodes 0..65;

var intNodes D[Species, Species];

struct Triple {Species i; Species j; Species k; };
{Triple} Triples =...;

struct Predate {Species i; Species j; Species k; Species l;};
{Predate} Predates =...;

struct Boundary {Species i; Species j; int k; int l;};
{Boundary} Boundaries =...;

solve {
  forall(ordered i, j  in Species)
    D[i, j] = D[j, i];

  forall(i in Species)
    D[i,i]=0;

  forall (triple in Triples)
    D[triple.i,triple.j] < D[triple.i,triple.k] =
   D[triple.j,triple.k];

  forall (ordered i, j, k in Species)
    D[i, j] = D[i, k] > D[j, k] \/ D[i, j] = D[j, k] > D[i, k] \/
   D[i, k] = D[j, k] > D[i, j];

  forall (predate in Predates)
    D[predate.i,predate.j] > D[predate.k,predate.l];

  forall(ordered Boundary in Boundaries)
    Boundary.k <= D[Boundary.i, Boundary.j] <= Boundary.l;
};
```

```
search{
generate(D);
  forall(i in Species)
    forall(j in Species: i<j & not bound(D[i,j]))
      tryall(k in [dmin(D[i,j])..dmax(D[i,j])]
ordered by decreasing k) D[i,j]=k;
};
display(ordered i,j in Species) D[i,j]/10.0;
```

The data file:

```
Species = {a, b, c, d, e, f};

Triples = { <a, b, c> <c, f, b> <e, d, a> };

Predates = {
   <a, e, c, f>
   <a, b, a, d>
} ;

Boundaries = {
     <a, d, 10, 35>
     <a, b, 40, 60>
     <c, f, 30, 50>
            };
```

The concept in this model, is the same as the previous example, but this time we include also a boundary information. To represent this information in our model we first declared a record `Boundary` consisting of four fields `i`, `j`, `k` and `l` using `Boundaries` in the data file with

```
struct Boundary {Species i; Species j; int k; int l;};
{Boundary} Boundaries =...;
```

and inserted a boundary constraint as:

```
forall(ordered Boundary in Boundaries)
   Boundary.k <= D[Boundary.i, Boundary.j] <= Boundary.l;
```

telling that the most recent common ancestor of species *i* and *j* must be within the boundary conditions stated in the data file.

But the tricky part is in the search procedure:

```
search{
  generate(D);
  forall(i in Species)
    forall(j in Species: i<j & not bound(D[i,j])) tryall(k in
[dmin(D[i,j])..dmax(D[i,j])] ordered by decreasing k) D[i,j]=k;
};
```

Here we determined how the search will be, trying from the minimum value to maximum value $D_{[i,\ j]}$. Using the 'stepping in model' feature in ILOG OPL studio and also displaying the value of *D* we are able to analyze the boundary values and possible solutions step by step. Below in the Figures 3.6 to 3.11 we can see the steps of the value of *D* while searching for a solution.

	a	b	c	d	e	f
a	0	[40..60]	[41..65]	[31..35]	[31..35]	[41..65]
b	[40..60]	0	[41..65]	[0..65]	[0..65]	[41..65]
c	[41..65]	[41..65]	0	[0..65]	[0..65]	[30..34]
d	[31..35]	[0..65]	[0..65]	0	[0..34]	[0..65]
e	[31..35]	[0..65]	[0..65]	[0..34]	0	[0..65]
f	[41..65]	[41..65]	[30..34]	[0..65]	[0..65]	0

Figure 3.6: The assignment of intervals as given by the boundry value.

	a	b	c	d	e	f
a	0	[40..60]	[41..65]	31	31	[41..65]
b	[40..60]	0	[41..65]	[0..65]	[0..65]	[41..65]
c	[41..65]	[41..65]	0	[0..65]	[0..65]	30
d	31	[0..65]	[0..65]	0	[0..30]	[0..65]
e	31	[0..65]	[0..65]	[0..30]	0	[0..65]
f	[41..65]	[41..65]	30	[0..65]	[0..65]	0

Figure 3.7: After assigning 3.0 to the internal vertice (*c*, *f*) and 3.1 to the internal vertices (*a*, *d*) and (*a*, *e*).

	a	b	c	d	e	f
a	0	40	41	31	31	41
b	40	0	41	[0..65]	[0..65]	41
c	41	41	0	[0..65]	[0..65]	30
d	31	[0..65]	[0..65]	0	[0..30]	[0..65]
e	31	[0..65]	[0..65]	[0..30]	0	[0..65]
f	41	41	30	[0..65]	[0..65]	0

Figure 3.8: After assigning 4.0 to the internal vertice (*a*, *b*) and 4.1 to the internal vertice (*a*, *c*), (*a*, *f*), (*b*, *c*) and (*b*, *f*).

	a	b	c	d	e	f
a	0	40	41	31	31	41
b	40	0	41	40	40	41
c	41	41	0	[1..65]	[1..65]	30
d	31	40	[1..65]	0	0	[0..65]
e	31	40	[1..65]	0	0	[0..65]
f	41	41	30	[0..65]	[0..65]	0

Figure 3.9: After assigning 4.1 to the internal vertice (*b*, *d*) and (*b*, *e*) and 0 to the internal vertice (*d*, *e*).

	a	b	c	d	e	f
a	0	40	41	31	31	41
b	40	0	41	40	40	41
c	41	41	0	41	41	30
d	31	40	41	0	0	[2..65]
e	31	40	41	0	0	[0..65]
f	41	41	30	[2..65]	[0..65]	0

Figure 3.10: After assigning 4.1 to the internal vertice (*c*, *d*) and (*c*, *e*).

	a	b	c	d	e	f
a	0	40	41	31	31	41
b	40	0	41	40	40	41
c	41	41	0	41	41	30
d	31	40	41	0	0	41
e	31	40	41	0	0	41
f	41	41	30	41	41	0

Figure 3.11: Finding the first solution.

Here it is important to note that the minimum value of the internal vertice can be assigned 0. In which case we say that the divergence has just occurred for the species (in this case the internal vertice (*d*, *e*)). Also we transformed the decimal values to integers and put the interior vertex values to be at most 65.

3.3 Nested Taxa

Until now we saw how to combine collections of rooted phylogenetic with overlapping leaf sets into a single rooted phylogenetic tree. But in all these collections of rooted phylogenetic tree we didn't see any nested taxa. For example, we didn’t take into account examples like “mammals” and “domestic cat”. Because the “domestic cat” is nested inside the “mammals”, they cannot be represented by two distinct leaves in a single rooted phylogenetic tree. But in practice there is a need to insert this taxonomic information into the resulting supertree as well. In brief we need to find a way to combine rooted trees where the resulting supertree displays all nestings shared by all of the input trees. We call the rooted tree in which all the leaves as well as some of the interior vertices are labelled, semi-labelled trees. Two semi-labelled trees are shown in Figure 3.12.

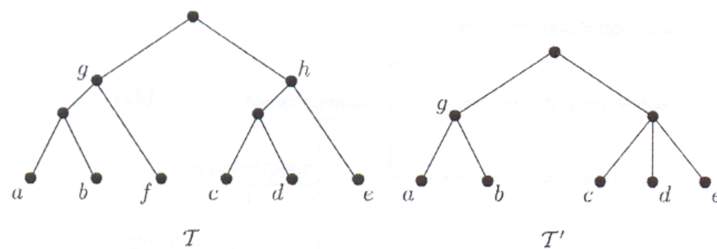


Figure 3.12: Two semi-labelled trees.

Semple and Daniel [16] [18] proposed two algorithms (SEMI-LABELLEDBUILD and ANCESTRALBUILD) for combining collections of rooted semi-labelled trees following a problem posed by Page [11]. Instead of describing these two algorithms we will use constraint programming and illustrate the resulting model on the example of Semple et al. [18] taken from study *S1x6x97c14c42c30* in TreeBASE [19] where the input trees describe the evolution of spiders, shown in Figure 3.13.

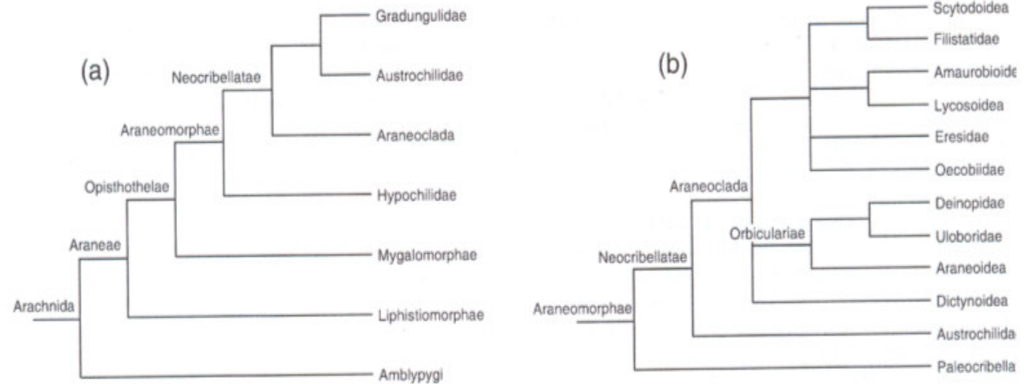


Figure 3.13: An application of nested taxa where the input trees describe the evolution of spiders.

The model file:

```
//* Example from the TreeBASE study: S1x6x97c14c42c30

{string} Species = ...;

struct Triple {
   string i;
   string j;
   string k;
   };
{Triple} Triples =...;

struct Polytomy {
   {string} s;
};

{Polytomy} Polytomies =...;
struct Ancestor {
   string anc;
   string suc;
   };

{Ancestor} Ancestors =...;

{string} Leaves = Species diff {ancestor.anc | ancestor in
Ancestors};

range intNodes 0..8;

var intNodes D[Leaves, Leaves];

{Triple} Triplets = Triples
   union {<ancestor.suc,l,m> | ancestor in Ancestors & <k,l,m> in
Triples : k = ancestor.anc}
   union {<k,ancestor.suc,m> | ancestor in Ancestors & <k,l,m> in
Triples : l = ancestor.anc}
   union {<k,l,ancestor.suc> | ancestor in Ancestors & <k,l,m> in
Triples : m = ancestor.anc} ;
```

```
{Triple} Triplettes = Triplets diff {<k,l,m> | <k,l,m> in Triplets
& <p,q> in Ancestors : k = p \/ l = p \/ m = p };

solve {
   forall(ordered i, j  in Leaves)
     D[i, j] = D[j, i];

   forall(i in Leaves)
     D[i, i] = 0;

   forall(ordered i, j in Leaves)
     D[i, j] > 0   => sum(k in Leaves: k<>i) (D[i, k] =
    D[i, j] - 1) > 0;

   forall (triplette in Triplettes)
     D[triplette.i, triplette.j] > D[triplette.i, triplette.k] =
    D[triplette.j, triplette.k];

   forall(ordered i, j, k in Leaves)
     D[i, j] = D[i, k] <= D[j, k] \/ D[i, j] = D[j, k] <= D[i, k]
\/
    D[i, k] = D[j, k] <= D[i, j];

   forall (<s> in Polytomies)
     forall (ordered i, j in s: i <> s.first() \/ j <>
s.next(s.first()))
       D[i, j] = D[s.first(), s.next(s.first())];
};

display(ordered i, j in Leaves) D[i,j];
```

The data file:

```
Species = {
        "Scytodoidea", "Filistatidae", "Amaurobioidea",
        "Lycosoidea", "Eresidae", "Oecobiidae", "Deinopidae",
        "Uloboridae", "Araneoidea", "Dictynoidea",
        "Austrochilidae", "Paleocribellatae", "Gradungulidae",
        "Araneoclada", "Hypochilidae", "Mygalomorphae",
        "Liphistiomorphae", "Amblypygi",

        "Neocribellatae", "Araneomorphae","Orbiculariae",
        "Opisthothelae", "Araneae", "Arachnida"
         };
Triples = {
   <"Scytodoidea", "Filistatidae", "Eresidae">
   <"Amaurobioidea", "Lycosoidea", "Scytodoidea">
   <"Deinopidae", "Uloboridae", "Araneoidea">
   <"Deinopidae", "Araneoidea", "Scytodoidea">
   <"Scytodoidea", "Amaurobioidea", "Araneoidea">
   <"Scytodoidea", "Dictynoidea", "Austrochilidae">
   <"Scytodoidea", "Austrochilidae", "Paleocribellatae">

   <"Gradungulidae", "Austrochilidae", "Araneoclada">
   <"Austrochilidae", "Araneoclada", "Hypochilidae">
   <"Austrochilidae", "Hypochilidae", "Mygalomorphae">
   <"Austrochilidae", "Mygalomorphae", "Liphistiomorphae">
```

```
<"Austrochilidae", "Liphistiomorphae", "Amblypygi">

<"Orbiculariae", "Dictynoidea", "Austrochilidae">
<"Araneomorphae", "Mygalomorphae", "Liphistiomorphae">
<"Araneoclada", "Austrochilidae", "Paleocribellatae">
<"Neocribellatae", "Hypochilidae", "Mygalomorphae">
<"Araneomorphae", "Mygalomorphae", "Liphistiomorphae">
<"Opisthothelae", "Liphistiomorphae", "Amblypygi">
<"Paleocribellatae", "Hypochilidae", "Mygalomorphae">
      };

Polytomies = {
  <{"Scytodoidea", "Amaurobioidea", "Eresidae",
         "Oecobiidae"}>
  <{"Scytodoidea", "Deinopidae", "Dictynoidea"}>
           };

Ancestors = {
        <"Orbiculariae", "Araneoidea">
        <"Orbiculariae", "Uloboridae">
        <"Orbiculariae", "Deinopidae">

        <"Araneoclada", "Scytodoidea">
        <"Araneoclada", "Filistatidae">
        <"Araneoclada", "Amaurobioidea">
        <"Araneoclada", "Lycosoidea">
        <"Araneoclada", "Eresidae">
        <"Araneoclada", "Oecobiidae">
        <"Araneoclada", "Orbiculariae">
        <"Araneoclada", "Dictynoidea">

        <"Neocribellatae", "Gradungulidae">
        <"Neocribellatae", "Austrochilidae">
        <"Neocribellatae", "Araneoclada">

        <"Araneomorphae", "Neocribellatae">
        <"Araneomorphae", "Hypochilidae">
        <"Araneomorphae", "Paleocribellatae">


        <"Opisthothelae", "Mygalomorphae">
        <"Opisthothelae", "Araneomorphae">

        <"Araneae", "Opisthothelae">
        <"Araneae", "Liphistiomorphae">

        <"Arachnida", "Araneae"> <"Arachnida", "Amblypygi">
          } ;
```

In this model we declared a `Polytomies` and `Ancestors` in the data file with

```
struct Polytomy {
{string} s;
};

{Polytomy} Polytomies =...;
struct Ancestor {
```

```
        string anc;
        string suc;
        };
```

and using the constraint

```
        forall (<s> in Polytomies)
             forall (ordered i, j in s: i <> s.first() \/ j <>
      s.next(s.first()))
               D[i, j] = D[s.first(), s.next(s.first())];
        };
```

we stated that the most recent common ancestor for every fan species are equal.

And using the statement

```
{string} Leaves = Species diff {ancestor.anc | ancestor in
Ancestors};
```

we defined the leaves in the supertree by replacing all the ancestors in `Species` by their `descendants`.

By stating

```
{Triple} Triplets = Triples
   union {<ancestor.suc,l,m> | ancestor in Ancestors & <k,l,m> in
Triples : k = ancestor.anc}
   union {<k,ancestor.suc,m> | ancestor in Ancestors & <k,l,m> in
Triples : l = ancestor.anc}
   union {<k,l,ancestor.suc> | ancestor in Ancestors & <k,l,m> in
Triples : m = ancestor.anc} ;
```

we define `Triplets` by adding the descendants to the ancestors in the `Triples` enumeration and by

```
        {Triple} Triplettes = Triplets diff {<k,l,m> | <k,l,m> in
      Triplets & <p,q> in Ancestors : k = p \/ l = p \/ m = p };
```

we define `Triplettes` by removing the ancestors triples from the `Triplets`.

We determined the internal nodes, from the input trees internal nodes depth (by counting the paranthesis in the newick file representations of the input trees), to be in the range 0 to 8.

Since the input trees are semi labelled we can’t directly apply the BreakUp algorithm to collect the triples. Instead we can apply a three step process:

1. for each given tree T, generate triples using BreakUp on T;

2. for each given tree T, generate triples using BreakUp on each tree obtained from T by cutting all the descendants of some 'labeled' internal node of T;

3. if the root of some given tree T is labeled, say N, and is not the root of some other given tree T', then generate triples using BreakUp on the subtree of T' rooted at the parent of N, where the subtree rooted at N has been replaced by T.

In the first step the triples

For T_1

```
<"Gradungulidae", "Austrochilidae", "Araneoclada">
<"Austrochilidae", "Araneoclada", "Hypochilidae">
<"Austrochilidae", "Hypochilidae", "Mygalomorphae">
<"Austrochilidae", "Mygalomorphae", "Liphistiomorphae">
<"Austrochilidae", "Liphistiomorphae", "Amblypygi">
```

and for T_2

```
<"Scytodoidea", "Filistatidae", "Eresidae">
<"Amaurobioidea", "Lycosoidea", "Scytodoidea">
<"Deinopidae", "Uloboridae", "Araneoidea">
<"Deinopidae", "Araneoidea", "Scytodoidea">
<"Scytodoidea", "Amaurobioidea", "Araneoidea">
<"Scytodoidea", "Dictynoidea", "Austrochilidae">
<"Scytodoidea", "Austrochilidae", "Paleocribellatae">
```

are generated.

On the second step

For T_1

```
<"Neocribellatae", "Hypochilidae", "Mygalomorphae">
<"Araneomorphae", "Mygalomorphae", "Liphistiomorphae">
<"Opisthothelae", "Liphistiomorphae", "Amblypygi">
```

for T_2

```
<"Orbiculariae", "Dictynoidea", "Austrochilidae">
<"Araneoclada", "Austrochilidae", "Paleocribellatae">
```

On the third step placing "Paleocribellatae" on T_2 under "Araneomorphae" on T_1 we get

```
<"Paleocribellatae", "Hypochilidae", "Mygalomorphae">
```

To generate the ancestor data structure we placed all the species under labeled internal node. For example here we didn't placed all the species under "Arachnida". We just stated

```
<"Arachnida", "Araneae"> <"Arachnida", "Amblypygi">
```

in order to prevent to state all the species under "Araneae".

After running the model we get 4 solutions. To generate a graphical representation of this 4 possible solutions we first produce ultrametric matrix solutions using an OPL script shown below.

```
ofile result("matrix.txt");
Model m("Spiders.mod","Spiders.dat") editMode;

int k := 0;
while m.nextSolution() do {
     k := k + 1;
     result << "-Solution " << k << "-" <<endl;
     result << "    " << card(m.Leaves) << endl;

   forall(i in m.Leaves) {
      result << i << "          ";

   forall(j in m.Leaves) {
      if i=j then
             result << m.D[i,j] << "         " ;
      else
             result << 9-m.D[i,j] << "         " ;
}
   result  <<  endl;
}
   result << endl;
}
  result.close();
```

In this script we substracted internal node numbers from nine (which is out of the range of internal values) in order to get an ultrametric matrix. The output of the script is shown below.

```
-Solution 1-
  17
Scytodoidea  0 2 3 3 3 3 4 4 4 4 5 6 5 6 7 8 9
Filistatidae  2 0 3 3 3 3 4 4 4 4 5 6 5 6 7 8 9
Amaurobioidea  3 3 0 2 3 3 4 4 4 4 5 6 5 6 7 8 9
Lycosoidea  3 3 2 0 3 3 4 4 4 4 5 6 5 6 7 8 9
Eresidae  3 3 3 3 0 3 4 4 4 4 5 6 5 6 7 8 9
Oecobiidae  3 3 3 3 3 0 4 4 4 4 5 6 5 6 7 8 9
Deinopidae  4 4 4 4 4 4 0 2 3 4 5 6 5 6 7 8 9
Uloboridae  4 4 4 4 4 4 2 0 3 4 5 6 5 6 7 8 9
```

Araneoidea 4 4 4 4 4 4 3 3 0 4 5 6 5 6 7 8 9
Dictynoidea 4 4 4 4 4 4 4 4 4 0 5 6 5 6 7 8 9
Austrochilidae 5 5 5 5 5 5 5 5 5 5 0 6 4 6 7 8 9
Paleocribellatae 6 6 6 6 6 6 6 6 6 6 6 0 6 6 7 8 9
Gradungulidae 5 5 5 5 5 5 5 5 5 5 4 6 0 6 7 8 9
Hypochilidae 6 6 6 6 6 6 6 6 6 6 6 6 6 0 7 8 9
Mygalomorphae 7 7 7 7 7 7 7 7 7 7 7 7 7 7 0 8 9
Liphistiomorphae 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 0 9
Amblypygi 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 0

-Solution 2-
17
Scytodoidea 0 2 3 3 3 3 4 4 4 4 5 6 5 6 7 8 9
Filistatidae 2 0 3 3 3 3 4 4 4 4 5 6 5 6 7 8 9
Amaurobioidea 3 3 0 2 3 3 4 4 4 4 5 6 5 6 7 8 9
Lycosoidea 3 3 2 0 3 3 4 4 4 4 5 6 5 6 7 8 9
Eresidae 3 3 3 3 0 3 4 4 4 4 5 6 5 6 7 8 9
Oecobiidae 3 3 3 3 3 0 4 4 4 4 5 6 5 6 7 8 9
Deinopidae 4 4 4 4 4 4 0 2 3 4 5 6 5 6 7 8 9
Uloboridae 4 4 4 4 4 4 2 0 3 4 5 6 5 6 7 8 9
Araneoidea 4 4 4 4 4 4 3 3 0 4 5 6 5 6 7 8 9
Dictynoidea 4 4 4 4 4 4 4 4 4 0 5 6 5 6 7 8 9
Austrochilidae 5 5 5 5 5 5 5 5 5 5 0 6 4 6 7 8 9
Paleocribellatae 6 6 6 6 6 6 6 6 6 6 6 0 6 5 7 8 9
Gradungulidae 5 5 5 5 5 5 5 5 5 5 4 6 0 6 7 8 9
Hypochilidae 6 6 6 6 6 6 6 6 6 6 6 5 6 0 7 8 9
Mygalomorphae 7 7 7 7 7 7 7 7 7 7 7 7 7 7 0 8 9
Liphistiomorphae 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 0 9
Amblypygi 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 0

-Solution 3-
17
Scytodoidea 0 1 2 2 2 2 3 3 3 3 4 5 4 6 7 8 9
Filistatidae 1 0 2 2 2 2 3 3 3 3 4 5 4 6 7 8 9
Amaurobioidea 2 2 0 1 2 2 3 3 3 3 4 5 4 6 7 8 9
Lycosoidea 2 2 1 0 2 2 3 3 3 3 4 5 4 6 7 8 9
Eresidae 2 2 2 2 0 2 3 3 3 3 4 5 4 6 7 8 9
Oecobiidae 2 2 2 2 2 0 3 3 3 3 4 5 4 6 7 8 9
Deinopidae 3 3 3 3 3 3 0 1 2 3 4 5 4 6 7 8 9
Uloboridae 3 3 3 3 3 3 1 0 2 3 4 5 4 6 7 8 9
Araneoidea 3 3 3 3 3 3 2 2 0 3 4 5 4 6 7 8 9
Dictynoidea 3 3 3 3 3 3 3 3 3 0 4 5 4 6 7 8 9
Austrochilidae 4 4 4 4 4 4 4 4 4 4 0 5 3 6 7 8 9
Paleocribellatae 5 5 5 5 5 5 5 5 5 5 5 0 5 6 7 8 9
Gradungulidae 4 4 4 4 4 4 4 4 4 4 3 5 0 6 7 8 9
Hypochilidae 6 6 6 6 6 6 6 6 6 6 6 6 6 0 7 8 9
Mygalomorphae 7 7 7 7 7 7 7 7 7 7 7 7 7 7 0 8 9
Liphistiomorphae 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 0 9
Amblypygi 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 0

-Solution 4-
17
Scytodoidea 0 1 2 2 2 2 3 3 3 3 4 6 4 5 7 8 9
Filistatidae 1 0 2 2 2 2 3 3 3 3 4 6 4 5 7 8 9
Amaurobioidea 2 2 0 1 2 2 3 3 3 3 4 6 4 5 7 8 9
Lycosoidea 2 2 1 0 2 2 3 3 3 3 4 6 4 5 7 8 9
Eresidae 2 2 2 2 0 2 3 3 3 3 4 6 4 5 7 8 9
Oecobiidae 2 2 2 2 2 0 3 3 3 3 4 6 4 5 7 8 9

```
Deinopidae  3 3 3 3 3 3 0 1 2 3 4 6 4 5 7 8 9
Uloboridae  3 3 3 3 3 3 1 0 2 3 4 6 4 5 7 8 9
Araneoidea  3 3 3 3 3 3 2 2 0 3 4 6 4 5 7 8 9
Dictynoidea  3 3 3 3 3 3 3 3 3 0 4 6 4 5 7 8 9
Austrochilidae  4 4 4 4 4 4 4 4 4 4 0 6 3 5 7 8 9
Paleocribellatae  6 6 6 6 6 6 6 6 6 6 6 0 6 6 7 8 9
Gradungulidae  4 4 4 4 4 4 4 4 4 4 3 6 0 5 7 8 9
Hypochilidae  5 5 5 5 5 5 5 5 5 5 5 6 5 0 7 8 9
Mygalomorphae  7 7 7 7 7 7 7 7 7 7 7 7 7 7 0 8 9
Liphistiomorphae  8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 0 9
Amblypygi  9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 0
```

The number of 17 at the head of the ultrametric matrix, computed from

```
result << "    " << card(m.Leaves) << endl;
```

gives us the number of species, which is required in the input file by the tree drawing software PHYLIP [20].

Using the first solution matrix on PHYLIP we can generate in Newick tree format [21] one of the 4 possible solutions. Finally we use TreeView software [22] to generate the final tree shown in Figure 3.14.

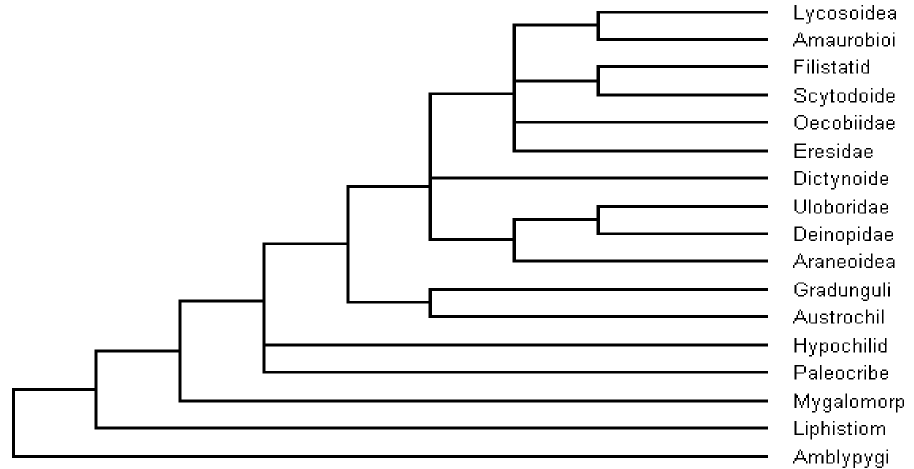


Figure 3.14: One of the four possible supertree solution generated after combining the input trees in Figure 3.13.

CHAPTER 4

CONCLUSIONS

It has been shown much interest in phylogenetic systematics recently. Because it is a method for biologists to reconstruct the pattern of events that have led to the distribution and diversity of life. Due to the "Tree of Life" initiatives [19] [23], and studies the researchers have tried to find some new methods to combine large number of trees to construct phylogenies on hundreds, or even thousands of species, where the construction of Supertree is one of these methods and in which we tried to approach in a different way, using constraint programming.

The advantages of using constraint programming was the ability to model different types of model by adding only one or two new constraints and to get every possible solutions.

It is important to note that all of the models described in this thesis provides a solution or no solution. Which means that each algorithm either returns a supertree with certain desirable properties relative to the input or returns a statement indicating that there is no such supertree, which is limiting their use. However this can be progressed to some models which will always return a supertree and

whose input includes information that goes beyond the properties shown here.

REFERENCES

1. **Pennisi, E.** (2003), *Modernizing the Tree of Life*, Science, vol. 300, pp. 1692-1697.

2. **Gent, IP.** et al. (2003), *Supertree Construction Using Constraint Programming*, in:Lecture Notes in Computer Science 2833*, Rossi, F (ed),* Proc. 9th International Conference on Principles and Practice of Constraint Programming (CP'2003), Springer.

3. **Apt., K. R.** (2003), *Principles of Constraint Programming*, Cambridge University Press.

4. **Marriott, K., Stuckey, P.** (1998), *Programming With Constraints: An Introduction*, MIT Press.

5. **Dechter, R.** (2003), *Constraint Processing*. Morgan Kaufmann Publishers.

6. **Van Hentenryck, P.** (1999), *The OPL Optimization Programming Language*, MIT Press.

7. *http://www.ilog.com,* "Ilog".

8. **Ng, M. P., Wormald, N. C.** (1995), *Reconstruction of Rooted Trees From Subtrees*, Discrete Applied Mathematics, 69:19-31.

9. **Bryant, D., Steel, M.** (1995), *Extension Operation on Sets of leaf-labeled Trees*, Advances in Applied Mathematics, 16:425-453.

10. **Semple, C., Steel, M.** (2000), *A Supertree Method For Rooted Trees*, Discrete Applied Mathematics, 105:147-158.

11. **Page, R. D. M.** (2002), *Modified Mincut Supertrees*, Proceedings of the Second International Workshop on Algorithms in Bioinformatics (WABI 2002), pp.537-552, Springer.

12. **Tsang, E.P.K. (1993)**, Foundations of Constraint Satisfaction, Academic Press, London and San Diego.

13. **Schröder, E.** (1870), *Vier Combinatorische Probleme*, *Z*eit. für Math. Phys.

14. **Semple, C. and Steel, M.** (2004). *Cyclic Permutations and Evolutionary Trees*, Advances in Applied Mathematics 32(4): 669-680.

15. **Bryant, D.** et al. (2004), *Supertree Methods for Ancestral Divergence Dates and Other Applications*. In *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life* (ed. O. Bininda-Edmonds), Computational Biology Series, Kluwer, pp.129-150.

16. **Semple, C.** et al. (2004), *Supertree Algorithms for Ancestral Divergence Dates and Nested Taxa*, Bioinformatics 20, pp. 2355-2360.

17. **Aho, A. V.** et al. (1981), *Inferring a Tree From Lowest Common Ancestors With an Application to the Optimization of Relational Expressions*, SIAM Journal on Computing 10:405-421.

18. **Daniel, P. and Semple, C.** (2004), *Supertree Algorithms for Nested Taxa.* In *Phylogenetic Supertrees: Combining Information to Reveal the Tree of Life* (ed. O. Bininda-Edmonds), Computational Biology Series, Kluwer, pp.151-171.

19. *http://www.treebase.org,* "Treebase".

20. *http://evolution.genetics.washington.edu/phylip.html,* "PHYLIP Software".

21. *http://evolution.genetics.washington.edu/phylip/newicktree.html,* "Newick Tree File".

22. *http://darwin.zoology.gla.ac.uk/~rpage/treeviewx/*, "Treeview".

23. *http://tolweb.org/tree/*, "Tree of Life".