B.Sc. in Computer Science and Engineering Thesis

Construction of Phylogenetic Trees

Submitted by

Afifa Tahira
201114014

Md. Farhan Tanveer Rifat
201114026

Sharmistha Bardhan
201114062

Supervised by

Professor Dr. Md. Saidur Rahman

Department of Computer Science and Engineering

Military Institute of Science and Technology (MIST), Dhaka-1216

Department of Computer Science and Engineering
Military Institute of Science and Technology

December 2014

CERTIFICATION

This thesis paper titled **"Construction of Phylogenetic Trees"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in December 2014.

Group Members:

Afifa Tahira

Md. Farhan Tanveer Rifat

Sharmistha Bardhan

Supervisor:

Professor Dr. Md. Saidur Rahman
Department of Computer Science and Engineering
Military Institute of Science and Technology (MIST),
Dhaka-1216

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis paper, titled, "Construction of Phylogenetic Trees", is the outcome of the investigation and research carried out by the following students under the supervision of Professor Dr. Md. Saidur Rahman, Department of Computer Science and Engineering, Military Institute of Science and Technology (MIST), Dhaka-1216.

It is also declared that neither this thesis paper nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Afifa Tahira
201114014

Md. Farhan Tanveer Rifat
201114026

Sharmistha Bardhan
201114062

ACKNOWLEDGEMENT

We are thankful to Almighty Allah for his blessings for the successful completion of our thesis. Our heartiest gratitude, profound indebtedness and deep respect go to our supervisor, Professor Dr. Md. Saidur Rahman, Department of Computer Science and Engineering, Military Institute of Science and Technology (MIST), Dhaka-1216, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing thesis.

We are especially grateful to the Department of Computer Science and Engineering (CSE) of Military Institute of Science and Technology (MIST) for providing their all out support during the thesis work.

Finally, we would like to thank our families and our course mates for their appreciable assistance, patience and suggestions during the course of our thesis.

Dhaka
December 2014

Afifa Tahira

Md. Farhan Tanveer Rifat

Sharmistha Bardhan

ABSTRACT

A phylogenetic tree is an unordered, distinctly leaf-labeled tree which shows evolutionary relationship among the leaves that represent the species. Inferring phylogenies using computational methods has several important applications in biological and biochemical research. Drug discovery and conservation biology are such examples. Phylogenetic trees have some important applications in practical fields, such as forensics, gene and protein function detection and drug design. In Bioinformatics, computation of the Tree of Life considering all living beings on earth is a grand challenge, in which phylogenetic tree can play important role to derive such relations.

Phylogenetic tree construction from valid triplet set is one of the most common approaches of construction till date. A valid triplet set consists of a number of rooted triplets which are individually a phylogenetic tree of three leaves. Aho $et\ al.$ first investigate the problem and later other researchers gradually meliorate and enrich his idea. These great research works solve many problems as well as introduce some open problems related to phylogenetic tree construction. Such a problem is to construct a phylogenetic tree consistent with all triplets in the given triplet set. Experimentally obtained data may contain incorrect information which results in erroneous triplet set, and thus maximum rooted triplet consistency $(MaxRTC)$ is maintained in this case. During phylogenetic tree construction, maintenance of $MaxRTC$, sometimes ignores some important triplets where loss of information occurs. Moreover, the tree constructed from the triplet set contains unevenly distributed species.

In this thesis, we study different aspects of the problem and introduce a heuristic algorithm to construct phylogenetic tree on all the triplets in the triplet set by making a small number of corrections on the triplets, where approximately even distribution of species is ensured. For a given set of m triplets with n species, this algorithm runs in $O(m \times n)$ time. Though the algorithm is less efficient, the loss of information is less here than previous algorithms. On the other hand, checking the consistency of the triplet in a phylogenetic tree is another important problem in the field of phylogenetics. It is useful to compare two phylogenetic trees and measure the dissimilarity between them to check the accuracy. So it is very important to check this consistency in minimum amount of time. In this thesis, we give an algorithm to check the consistency of the triplet in constant time. This algorithm requires an initialization stage of time complexity $O(n^2)$ that preprocess the phylogenetic tree. Then we check if the triplet satisfies the phylogenetic tree or not in $O(1)$ time. Though the initial complexity is high, the algorithm is helpful because it can check the consistency of a triplet in constant time.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATION

RTC	: Rooted Triplet Consistency
$MaxRTC$	: Maximum Rooted Triplet Consistency
PCG	: Pairwise Compatibility Graph
LCA	: Lowest Common Ancestor
$AllRTC$	: All Rooted Triplet Consistency

LIST OF SYMBOLS

T	: Phylogenetic Tree
P	: Triplet
$d_{rt}(T_1, T_2)$	: Dissimilarity between T_1 and T_2
R	: Rooted triplet set
π_c	: Partition of constraints

CHAPTER 1
INTRODUCTION

For almost 150 years, the theory of evolution represents the model of the evolution of species in the world. The discipline that deals with the modeling of evolution is called $phylogenetics$. The word $phylogenetics$ is originated from Greek words $phyle$ meaning tribe or race and $genesis$ meaning birth or beginnings. The data structure that has been used (and is still used) by the scientists and scholars to describe the evolutionary history is called phylogenetic tree. A $phylogenetic\ tree$ corresponds to a weighted tree-graph where the leaves represent the biological objects of interest.

The reconstruction of these trees are very interesting from both a computer scientific and biological point of view. For various reasons, inferring an accurate phylogenetic tree from experimental data is problematic. In this thesis we concentrate on two phylogenetic tree problems. The first problem that we consider is a phylogenetic tree construction problem from given erroneous data inputs. Another problem that we consider, involves finding a constant time algorithm that checks a triplet's consistency in a given phylogenetic tree.

Determining the evolutionary history of mankind and all other living and extinct species on earth is one of the fundamental goals of evolutionary biology. The "Tree of Life" is name of the ongoing project where many biologists are working to form one-encompassing tree, joining different parts of the tree together [2]. Phylogenetic trees are very useful to biologists; they represent a hypothesis about the geneological relationships among organisms derived from data such as DNA sequences. To perform phylogenetic analysis, first problem is to construct an accurate phylogenetic tree.

1.1 Phylogenetic Trees and Related Problems

In this section, we enlight the problem of phylogenetic tree construction. At first, we define the phylogenetic tree in a formal way. Later we discuss about the method of construction of a phylogenetic tree and our study related to pairwise compatibility graph. And then we define the problem of triplet consistency checking with a phylogenetic tree.

1.1.1 Phylogenetic Tree

A *phylogenetic tree* is an unordered, distinctly leaf-labeled tree which represents a binary evolutionary relationship between the species. A phylogenetic tree is also known as *cladogram* or *dendogram*. In this tree the leaves refer to the existing species and the internal nodes refer to the hypothetical ancestors. Two similar species are represented as neighbors and will be joined to a common parent branch. Phylogenetic tree is divided into two categories. [3] One is the unrooted tree, where the common ancestor is unknown. Other one is the rooted tree where there is a common ancestor. Figure 1.1 illustrates the two categories of tree. In the rooted phylogentic tree, the root is the most ancient ancestor, common to all species. In this thesis, we concentrate our analysis to only rooted phylogenetic trees.

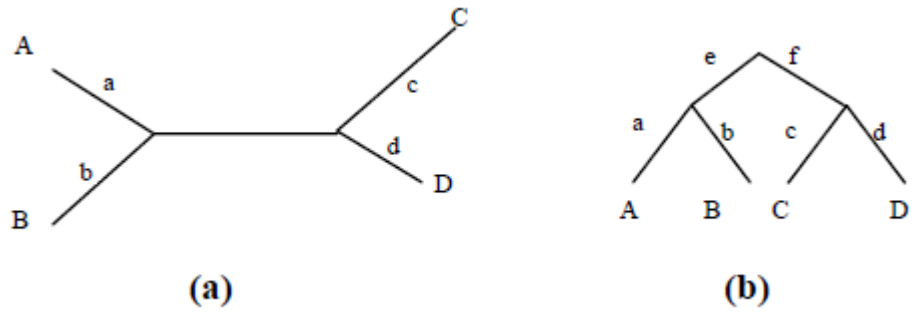

Figure 1.1: (a) An unrooted tree and (b) a rooted tree.

As an example of phylogenetic tree, we illustrate the phylogeny of lizards in Figure 1.2. In the figure, five current species are shown in the leaves of the tree. Here, time is the vertical dimension with the current time at the lowest branch and earlier times above it. The lines above the extant species represent the same species, just in the past. When two lines converge to a point, it means two taxa (species) diverge from a common ancestral taxon. The top most point is the taxon of the past, from where all other taxa is evaluate.

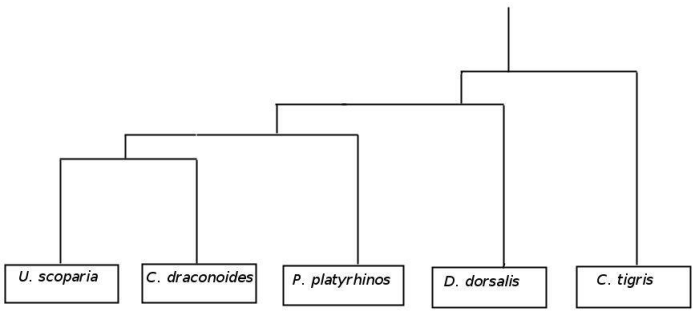

Figure 1.2: Phylogeny of lizards.

1.1.2 Construction of a Phylogenetic Tree

There are different methods to construct a phylogenetic tree. The method is dependent on the type of the given data. We can divide these methods into two main categories. One is distance based, another is character based. Both of these two categories offer a vast variety of options when constructing trees. Here we are using Supertree method. A $supertree$ method is a method for merging an input collection of phylogenetic trees on overlapping sets of $rooted\ triplets$ into a single phylogenetic tree called $supertree$ [4]. A $rooted\ triplet$ is a binary phylogenetic tree with exactly three leaves. If a unique rooted triplet with a leaf set x, y, z where the lowest common ancestor (lca) of x and y is a proper descendant of the lca of y and z is $xy|z$.

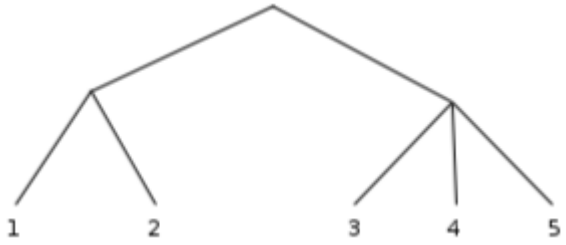

Figure 1.3: A solution to given triplet constraints.

If we represent $xy|z$ with index 1, 2, 3 respectively to x, y, z then the constraint of the triplet is:

$$(1,2) < (1,3)$$

Suppose we are given a set of constraints,

$$(1,2) < (1,3),$$

$$(3,4) < (1,5),$$

$$(3,5) < (2,4).$$

We can construct one possible tree T with these constraints, shown in Figure 1.3. Now, if we add another constraint $(4,5) < (1,2)$, then it is not possible to construct a tree satisfying all these constraints.

The input collection of given rooted triplets to construct a supertree might contain erroneous data because of error in experimental data. So, a supertree method should construct phylogenetic trees keeping as much of branching information as possible. Supertree methods are used because of two main reasons:

- Supertrees can be used to deduce hypothetical evolutionary relationships between taxa which do not occur together in any one of the input trees. For example, "the tree of life" project's goal is to construct a tree that represents the evolution of more than one and a half million species [2] that requires data from so many different sources to be combined.

- In the context of building a phylogenetic tree from sequence or character data, supertree methods may be convenient in case where the taxa set is too large for computationally expensive phylogenetic tree reconstruction methods such as likelihood or maximum parsimony.

In the supertree method, we follow a divide-and-conquer approach. First we apply an expensive method to infer a collection of highly accurate trees for small, overlapping subsets of the taxa. Then supertree method is applied, which is computationally cheaper. Many interesting algorithms are developed for this [5, 6].
Rooted supertree methods are preferable over unrooted supertree method on the point of computational complexity [7]. On the other hand, RTC is solvable in polynomial time, but others; such as, quartet consistency is NP-hard [8]. Reliable rooted triplets can be inferred from [6] or Sibley-Ahlquist-style DNA-DNA hybridization experiments [9]. RTC-based supertree methods is applicable to marsupial species as well as rbcL gene data [2] and $Cryptococcus\ gattii\ yeast$ data [10] and it performs well. In this thesis we construct phylogenetic tree using supertree method.

1.1.3 Pairwise Compatibility Graphs

In the field of different graph construction problem and different graph study problem, Pairwise Compatibility Problem (PCG) has become an interesting sector of it. PCG is used in many important graph studies and to determine evolutionary relationship between various species.

Let T be an edge weighted tree and G = (V, E) be a graph. We call graph G a Pairwise compatibility graph, if each vertex of G corresponds to a leaf of T and an edge $(u, v) \in E$ if and only if the distance between the two leaves of T corresponding to u and v is within a given range [11]. In Figure 1.4 we see an example of PCG. In Figure 1.4, there is an edge weighted tree T and the pairwise compatibility graph G of T. The PCG is constructed considering the distance between a pair of leaves in between four to seven.

During our thesis we also study some properties of pairwise compatibility graphs. It is because; Pairwise compatibility graphs have application in reconstruction of evolutionary relationships of a set of species from Phylogeny. Phylogeny represents biological data or

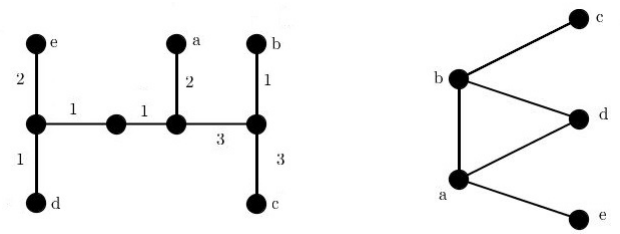

Figure 1.4: An edge weighted tree T and a pairwise compatibility graph G of T.

various species [11]. Some study relating Pairwise Compatibility Graph and Phylogenetic Tree construction is performed till date. Kearney $et\ al.$ introduce the pairwise compatibility graph concept while dealing with a sample problem in a phylogenetic tree [12]. Therefore building a proper relationship between a Pairwise Compatibility Graph and Phylogenetic Tree a new gateway in graph study and evaluating evolutionary relationship between species.

At the preliminary stage of our thesis, we study thoroughly on Pairwise Compatibility Graph. Since both the PCG and Phylogenetic tree is considered for deriving evolutionary relationship among set of species. Initially understanding the properties of Phylogenetic tree becomes easier since we know about PCGs. In our future development consideration, we have Pairwise Compatibility Graph and Phylogenetic Tree relationship and derive some effective relationship between these two concepts.

1.1.4 Consistency of a Triplet

Checking the consistency of a triplet with a phylogenetic tree is often an essential measure to compare two phylogenetic trees. Given a triplet $P(xy|z)$ and a phylogenetic tree T, the triplet P will be consistent with T, if the lowest common ancestor of x and y is a proper descendant of the lowest common ancestor of x and z, in the tree T. So we can divide this problem in three parts:

- Find the lowest common ancestor of x and y in tree T, say u.
- Find the lowest common ancestor of x and z in tree T, say v.
- Check if u is a proper descendant of v in the tree T.

In our thesis, we find a constant time algorithm which gives a solution to this problem. We pre-process the tree T in such a way so that each of the three sub problems can be solved in constant time. And finally after the processing, we can check the consistency of the triplet in constant time.

1.2 Motivation of the Thesis

In this section, we describe our motivation of the thesis briefly. In Section 1.2.1, we describe some practical applications and research works on phylogenetics which motivated us to do our thesis on this topic. In latter sections, we describe some previous works related to constructing phylogenetic trees and triplet consistency checking which inspired us for the thesis.

1.2.1 Motivation of Studying Phylogenetics

In 1990 a young woman in Florida contracted AIDS without having previously been exposed to the established risks of HIV infection. Scientists at the Centers for Disease Control in Atlanta launched an extensive investigation to find the cause of the infection. They discovered that a dentist suffering from AIDS had infected several of his patient with HIV [13]. This Finding is the result of application of two scientific methodologies, the experimental method of DNA sequencing and the mathematical method of Phylogenetic analysis.

Earlier phylogenetics just focused on the evolution of species based on morphological characteristics, but nowadays the explosive advancement in molecular biology now requires the investigation of proteins as well. During the last ten years, a lot of research has been carried out on the development of computational techniques for molecular biology research. This research effort, which includes research into algorithmic and combinatorial problems as well as software development, is called computational molecular biology. A lot of the motivation for computational biology research has come from the Human Genome Project, which aims to sequence the DNA of humans and to use this information to understand genes and their functions.

We always assume that, we can represent the evolutionary history of a group of related biological species as a rooted tree. Although disagreement about the true nature of evolution generate much controversy, this is a general consideration. The root of the tree represents the ancient species from which all the other species evolves. Any internal node of the tree represents a speciation event which splits the original species at that node into two or more new species, depending on the number of outgoing edges from the internal node. We bijectively label the leaves of the tree, by the group of species for available data.

1.2.2 Motivation of Studying Phylogenetic Tree Construction with Erroneous Data

Phylogenetic tree construction is a complex yet important problem in the field of bioinformatics. Once constructed, a phylogenetic or evolutionary tree can lend insight into the evolution of different species. The issue is that for a large number of species the problem

grows to a computational complexity that is not easily solved. For this reason, new methods are being researched and applied to phylogenetic tree construction and have provided some promising results.

The research on construction of phylogenetic tree has been triggered on 1981, by A. V. Aho [14]. Aho gave an of $O(kn)$ to find $Rooted\ Triplet\ Consistency\ (RTC)$ for a given triplet set in the form of constraints. Here, k is the number of triplets and n is the number of leaves in the phylogenetic tree. This algorithm returns the resultant phylogenetic tree consisting all the triplets, if tree construction is possible; otherwise it does not construct any tree. Aho's approach of RTC is studied for years to optimize the running time and to modify to return a tree even when the data set is not completely adaptable. Later, Wu [15] gave an algorithm of $O((k + n^2)3^n)$ which follows $Maximum\ Rooted\ Triplet\ Consistency$ $MaxRTC$, which is an dynamic-programming algorithm to produce a phylogenetic tree with maximum adaptable triplets of input data set. Following $MaxRTC$, Gasieniec [16] has given two polynomial algorithms that introduce two new techniques of $One\text{-}Leaf\text{-}Split$ and $Min\text{-}Cut\text{-}Split$. Vast research work had been done in this field to produce $MaxRTC$ phylogenetic tree with minimum loss.

But all these research works to produce $MaxRTC$ use the method of edge deletion, causing the deletion of one or more triplets. It causes a great loss of data in the resultant phylogenetic tree. At this point we found our inspiration of research to construct phylogenetic tree with erroneous data with minimum loss. Instead of edge deletion, we replace a vertex of the erroneous triplet with the most suitable option, as an attempt to correct the error, which saves a lot more data than edge deletion.

1.2.3 Motivation of Studying Triplet Consistency Check

Checking the consistency of a triplet in constant time has an important significance in the field of research in phylogeny. It has many applications in genetics, bio-informatics and evaluating the evolutionary relationship among species. It is an effective way to check the accuracy of an algorithm that constructs phylogenetic tree from triplets. This method is useful to compute the $rooted\ triplet\ distance$ between two phylogenetic trees.

The term rooted triplet distance was first introduced in 1975, by Dobson [17]. Given two phylogenetic trees T_1 and T_2 with same leaf label set L, the rooted triplet distance between T_1 and T_2 is $d_{rt}(T_1,T_2)$ is the number of rooted triplets over L that are consistent with exactly one of T_1 and T_2. Rooted triplet distance d_{rt} is a measure of dissimilarity between two phylogenetic trees. The value of d_{rt} will be smaller if two trees are similar and the number of consistent triplet sets is larger. This measure of dissimilarity between two phylogenetic trees is useful to compare two phylogenetic trees. This comparison helps in the sector of

evaluating methods for phylogenetic reconstruction [18] or querying phylogenetic database [19]. There are various methods developed in recent years to compare two phylogenetic trees. Some are $Robinson\text{-}Foulds\ distance$ [20], $tripartitions\ distance$, the $\mu\text{-}distance$, the $nodal\ distance$ are some of these. Some special structure of phylogenetic networks like $galled\ trees$ have been studied by Cardona $et\ al.$ [21] and later the complexity has been reduced to $O(n^{2.687})$ by Jansson and Lingas [22]. These vast scope of research on this field motivated us to study the consistency of a triplet in a phylogenetic tree.

1.3 Scope of the Thesis

Phylogenetic tree is vast field of interest among researchers and scientist. We focus our research on the construction of phylogenetic tree from given triplet set. There are various algorithms available to construct tree from triplet set. But most of them is not fit with erroneous data set. In this thesis, we research on how we can construct a phylogenetic tree from erroneous data set with minimum changes. We also give an algorithm to chack the consistency of a triplet in a phylogenetic tree.

1.3.1 Simulation of Aho′s Algorithm

In this thesis, we study the algorithm by Aho $et\ al.$ [14], which we consider the first initiative to construct the phylogenetic trees. We study different aspects of this algorithm through simulating different scenarios and implementation. In Chapter 3 of this thesis we describe our study about this algorithm. This study is helpful for us for further development.

1.3.2 Phylogenetic Tree with Erroneous Data

Later, we study the case of constructing phylogenetic tree with erroneous data. Here we concentrate to find out the data which contains the erroneous information. Then we correct that specific triplet in a systematic approach. We replace the species which cause the error with another species. Here we introduce a new term $AllRTC$ - All Rooted Triplet Consistency, as our attempt is to produce a phylogenetic tree which satisfies all the RTCs. We also concentrate on constructing a distributed phylogenetic tree, so that the tree is not dense. In Chapter 4, we give this method to ensure the construction of a phylogenetic tree with minimum error.

1.3.3 Triplet Consistency Check

Another topic of our research is on checking the consistency of triplet set with a phylogenetic tree. This is useful in comparing two phylogenetic trees and measuring the dissimilarity. In this thesis, we introduce an algorithm to check the consistency of a triplet in constant time. To implement this, the phylogenetic tree should be preprocessed. In the initialization stage we apply a naive approach to find the pairwise lowest common ancestors and finding the ancestors. In chapter 5, we discuss the algorithm in detail.

1.4 Organization of the Thesis

We give some preliminaries e.g. definitions of basic terms of graph theory and algorithms related to this thesis in Chapter 2. Then in Chapter 3, we discuss the existing algorithm of constructing phylogenetic tree from constraints and we describe Aho's algorithm to construct phylogenetic tree. Chapter 4 consists of one part of our main research. We give the new algorithm of construction of phylogenetic tree from erroneous triplet set here. We also discuss how to identify erroneous data and how to correct them in this chapter. In Chapter 5, we give our research on triplet consistency check in a phylogenetic tree in constant time. Finally, Chapter 6 summarizes our thesis and discusses open problems and future possibilities.

CHAPTER 2
PRELIMINARIES

In this chapter, we define some basic terms of graph theory and algorithms related to our thesis. In Section 2.1, we define some basic terminologies related to graph theory. Later, in Section 2.2, some basic terms related to algorithms and complexity is defined, which will be used throughout the thesis.

2.1 Basic Terminology

In this section we give some basic definitions used throughout the thesis. These definitions are mostly collected from the manuscript Basic Graph Theory [23] and the book Planar Graph Drawing [24].

2.1.1 Graph

A *graph* G is a tuple consisting of a finite set of *vertices* V and a finite set of *edges* E where each edge is an unordered pair of vertices. We denote the set of vertices of G by $V(G)$ and the set of edges by $E(G)$. An edge connecting vertices v_i and v_j in V is denoted by (v_i, v_j). An edge (v_i, v_j) is called a *loop* if $v_i = v_j$. A graph is called a *simple graph* if there is no loop or multiple edges between any two vertices in G. A subgraph of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

In Figure 2.1, we show a graph G with a vertex set $V(G) = \{v_1, v_2, v_3, v_4, v_5\}$ and edge set $E(G) = \{(v_1, v_2), (v_2, v_5), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5), (v_3, v_1)\}$. Since there is no

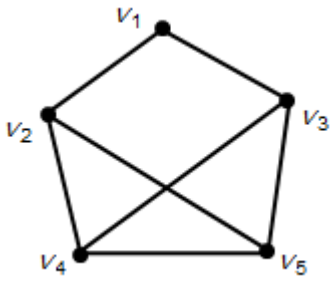

Figure 2.1: Illustration for a graph.

loop or multiple edges between two vertices, it is a simple graph.

2.1.2 Degree of a Vertex

The $degree$ of a vertex v in a graph G, denoted by $deg(v)$ or $d(v)$, is the number of edges incident to v in G, with each loop at v counted twice. In Figure 2.1, the degree of the vertex v_1 is 2 and the degree of the vertex v_2 is 3.

2.1.3 Path Graph

A $path\ graph$ is a graph G that contains a list of vertices $v_1, v_2, \cdots, v_p$ of G such that for $1 \leq i \leq p-1$, there is an edge (v_i, v_{i-1}) in G and these are the only edges in G. The two vertices v_1 and v_p are called the $end-vertices$ of G.

Figure 2.2: Illustration for a path graph.

In Figure 2.2, a path graph with four vertices is illustrated. A $path\ graph$ with n vertices is denoted by P_n. The degree of each vertex of a path graph is two except for the two end-vertices, both of which have degree one.

2.1.4 Cycle Graph

A $cycle\ graph$ is a graph that is obtained by joining the two end-vertices of a graph. Thus the degree of each vertex of a cycle is two. A cycle graph with n vertices is often denoted by C_n. In Figure 2.3, a cycle graph with four vertices is illustrated.

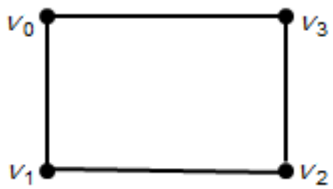

Figure 2.3: Illustration for a cycle graph.

2.1.5 Connected Graph

A graph is a *connected graph* if there is a path between every pair of vertices. Inversely, a graph is a *disconnected graph* if there no path between every pair of vertices.

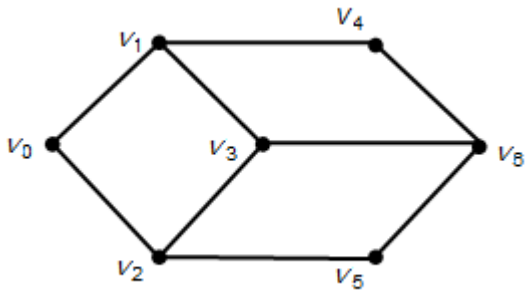

Figure 2.4: Illustration for a connected graph.

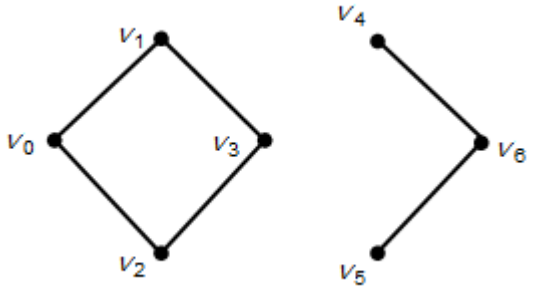

Figure 2.5: Illustration for a disconnected graph.

In the graph of Figure 2.4, there is path between every pair of vertices; thus it is a connected graph. But in the graph of Figure 2.5, there is no path between the vertices v_3 and v_4; so it is a disconnected graph.

2.1.6 Connected Component of a Graph

A connected subgraph which does not contain any other larger subgraph, is called a *maximal connected subgraph* of a graph. A *connected component* of a graph, is a maximal connected subgraph. A graph can have one or more connected components.

In Figure 2.6, the graph $G = (V, E)$ have two connected components $C_1 = (V_1, E_1)$ and $C_2 = (V_2, E_2)$, where $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$, $V_1 = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ and $V_2 = \{v_6, v_7, v_8\}$.

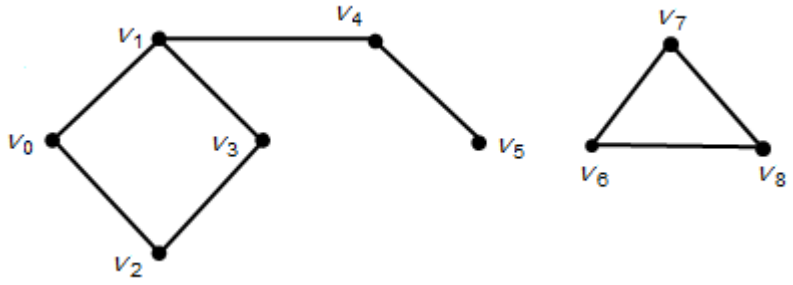

Figure 2.6: Illustration for a graph with two connected components.

2.1.7 Connectivity of a Graph

The *connectivity* $\kappa(G)$ of a connected graph G is the minimum number of vertices whose removal results in a disconnected graph or a single vertex graph K_1. A graph G is *k-connected* if $\kappa(G) \geq k$. A *separating set* or a *vertex cut* of a connected graph G is a set $S \subseteq V(G)$ such that $G - S$ has more than one component. If a vertex-cut contains exactly one vertex, then we call the vertex-cut a *cut-vertex*. If a vertex-cut contains exactly two vertices, then we call the two vertices a *separation-pair*.
The *edge-connectivity* $\kappa'(G)$ of a connected graph G is the minimum number of edges whose removal results in a disconnected graph. A graph is *k-edge-connected* if $\kappa'(G) \geq k$. A *disconnecting set* of edges in a connected graph is a set $F \subseteq E(G)$ such that $G - F$ has more than one component. If a disconnecting set contains exactly one edge, it is called a *bridge*.

In the graph of Figure 2.7, connectivity$\kappa(G) = 3$, as we must remove at least three ver-

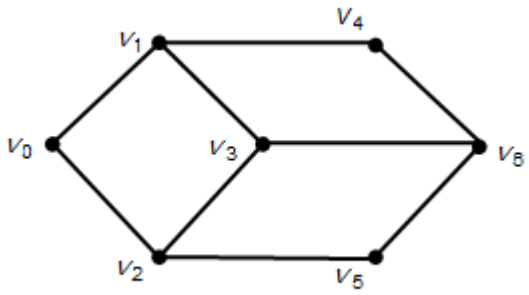

Figure 2.7: Illustration for a connected graph.

tices to disconnect the graph. We can call this graph a 3-connected graph. For the same graph, separating set or vertex-cut would be $S = \{v_1, v_2, v_3\}$. The edge connectivity of the graph is $\kappa'(G) = 3$, because we need to delete at least three edges to disconnect the graph. We can call this graph 3-edge-connected. A disconnecting set for this graph can be $F = \{(v_1, v_4), (v_3, v_6), (v_2, v_5)\}$.

2.1.8 Tree

A $tree$ is a connected graph containing no cycle. The vertices in a tree are usually called $nodes$. A $rooted\ tree$ is a tree in which one of the nodes is distinguished from the others. The distinguished node is called the $root$ of the tree. Every node u other than the root is connected by an edge to some other node p called the parent of u. We also call u a child of p. A $leaf$ is a node of a tree that has no children. An $internal\ node$ is a node that has one or more children. Thus every node of a tree is either a leaf or an internal node.
The parent child relationship can be extended naturally to ancestors and descendants. Suppose that $u_1, u_2, \cdots, u_l$ is a sequence of nodes in a tree such that u_1 is the parent of u_2, which is a parent of u_3, and so on. Then node u_1 is called an $ancestor$ of u_l and u_1 is a $descendant$ of u_1. The root is an ancestor of every node in a tree and every node is a descendant of the root.

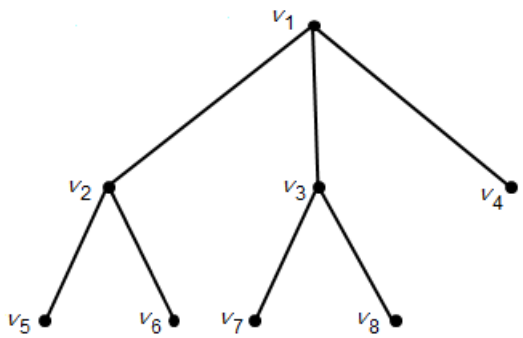

Figure 2.8: Illustration for a tree.

In Figure 2.8, a tree is with 8 nodes is illustrated. Here, v_1 is the root node; v_1, v_2, v_3, v_4 are internal nodes; v_5, v_6, v_7, v_8 are leaf nodes. Node v_2 is an ancestor of node v_5 but not of node v_7. Node v_5 is a descendant of v_2, but not of v_3.
A collection of trees is called a $forest$. In other words, a forest is a graph with no cycle. Such a graph is also called an $acyclic\ graph$. Each component of a forest is a tree.

2.1.9 Binary Tree

A $binary\ tree$ is either a single node or consists of a node and two subtrees rooted at the node, both of the subtrees are binary trees. A $complete\ binary\ tree$ is a rooted tree with each internal node having exactly two children.

Figure 2.9 illustrates a binary tree with 15 nodes.

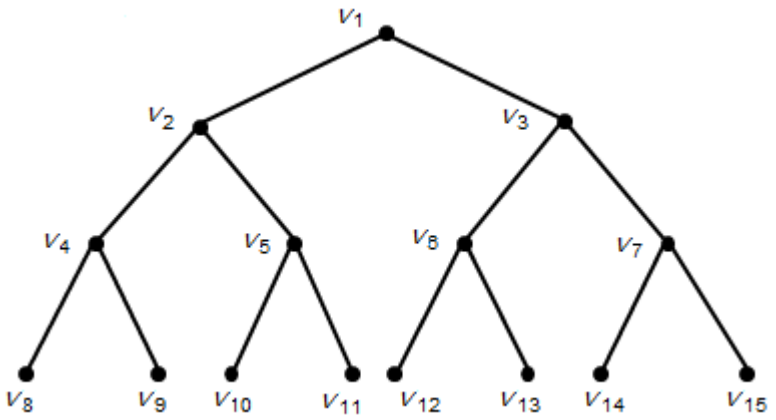

Figure 2.9: Illustration for a binary tree.

2.1.10 Set

A set is a collection of distinguishable objects, called its $members$ or $elements$. If an object x is a member of a set S, we write $x \in S$. If x is not a member of S, we write $x \notin S$. We can describe a set by explicitly listing its members as a list inside braces.
For example, we can define a set S to contain precisely the numbers 1, 2 and 3 by writing S = { 1, 2, 3}. Since 2 is a member of the set S, we can write $2 \in S$, and since 4 is not a member of the set S, we can write $4 \notin S$.
Given two sets A and B, we can also define new sets by applying $set\ operations$:

- The $intersection$ of the sets A and B is the set $A \cap B$ = { $x : x \in A$ and $x \in B$ }
- The $union$ of the sets A and B is the set $A \cup B$ = { $x : x \in A$ or $x \in B$ }
- The $difference$ between sets A and B is the set A - B = { $x : x \in A$ and $x \notin B$ }

2.1.11 Array

An $array$ is a data structure in which the location of an entry can be uniquely determined by its index and the entry can be accessed in constant time. A vector or a set of variables is usually stored as a (one-dimensional) array and a matrix is stored as a two-dimensional array.

2.1.12 List

A $list$ is a data structure which consists of homogeneous records, linked together in a linear fashion. Each record contains one or more items of data and one or more pointers. In a $singly\ linked\ list$, each record has a single forwarding pointer indicating the address of

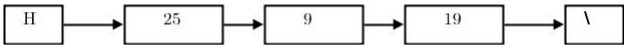

Figure 2.10: Illustration for a list.

the memory cell of the next record. In a $doubly\ linked\ list$, each record has forward and backward pointers indicating the address of the memory cell of the next and the previous records, respectively.

In the Figure 2.10, we show a singly linked list, holding three integer data 25, 9 and 19. The pointer 'H' indicates the starting of the list and $null$ indicates the end of the list.

2.2 Algorithms and Complexity

In this section, we give some basic definitions related to the algorithms and complexity of the algorithms.

2.2.1 The Notation $O(n)$

In analyzing the complexity of an algorithm, we are often interested only in the "asymptotic behavior", that is, the behavior of the algorithm when applied to very large inputs. To deal with such a property of functions we shall use the following notations for asymptotic running time. Let $f(n)$ and $g(n)$ are the functions from the positive integers to the positive reals, then we write $f(n)$ = $O(g(n))$ if there exists positive contants c_1 and c_2 such that $f(n) \leq c_1 g(n) + c_2$ for all n. Thus the running time of an algorithm may be bounded from above by phrasing like "takes time $O(n^2)$".

2.2.2 Polynomial Algorithms

An algorithm is said to be $polinomial\ bounded$ (or simply $polynomial$) if its complexity is bounded by a polynomial of the size of a problem instance. Examples of such complexities are $O(n)$, $O(nlogn)$, $O(n^100)$, etc. The remaining algorithms are usually referred as $exponential$ or $nonpolynomial$. Example of such complexity are $O(2n)$, $O(n!)$, etc.
When the running time of an algorithm is bounded by $O(n)$, we call it a $linear\ time$ indexlinear time algorithm or simply a $linear$ algorithm.

2.2.3 Constant Time

In computational complexity theory, $constant\ time$ refers to the computation time of a problem when the time needed to solve that problem doesn't depend on the size of the data that is as input. Constant time is noted as $O(1)$.
For example, accessing the elements in the array takes constant time as we can pick up an element using the index and start working with it. However finding the minimum value in an array is not a constant time operation as we need to scan each element of the array and then decide the minimum of those elements. Hence it is linear time operation and takes $O(n)$ time.

2.2.4 Recursive Algorithms

A $recursive\ algorithm$ is a problem solving technique in which it calls itself recursively one or more times to deal with closely related sub problems, to solve a specific problem. These algorithm break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem. In general, recursive algorithms require more memory and computation compared to iterative algorithms. Still, recursive algorithms are widely used in problem solving for its simplicity.

2.2.5 Graph Searching Algorithms

In graph theory, we often need a method to explore the vertices and edges of a graph. $Graph\ searching\ algorithms$ systematically follow the edges of the graph so as to visit the vertices of the graph. Depending on the order of exploring unvisited edges, there are two basic algorithms of searching a graph - $Breadth\ First\ Search\ (BFS)$ and $Depth\ First\ Search\ (DFS)$. We can use these algorithms to find number of connected components of a graph.

- **Breadth First Search (BFS)**: Given a graph $G = (V, E)$ and a distinguished $source$ vertex s, BFS systematically explores the edges of G to discover every vertex that is reachable from s, taking the ones closest to s first. Every vertex is visited at most once. It also produces a $breadth\text{-}first\ tree$ with root s that contains all reachable vertices. For any vertex v reachable from s, the simple path in the breadth-first tree from s to v in G, that is, a path containing the smallest number of edges. The running time of this algorithm is $O(V + E)$ where V is the vertex set and E is edge set.

- **Depth First Search (DFS)**: Given a graph $G = (V, E)$, DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once

all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source. The algorithm repeats this entire process until it has discovered every vertex. The predecessor subgraph of a depth-first search forms a $depth\text{-}first\ forest$ comprising several $depth\text{-}first\ trees$. The running time of this algorithm is also $O(V + E)$.

2.2.6 Tree Traversal

In a tree T, we can order the nodes based on the way the edges are chosen to be traversed. Let us consider a node v from which a new edge would be explored and another node would be reached. We mark a node u and call the label of u the $rank$ of u. The rank of the root of the tree is 0. So the rank of a vertex u is the number of vertices explored before u is reached for the first time. Such a traversal is called a $pre\text{-}order$ traversal of the vertices of the tree. If a vertex u is labeled after all vertices located in the subtree rooted at u are labeled, then the traversal is called $post\text{-}order$ traversal. In case of a binary tree, if the vertex u is labeled after all vertices located in the left subtree rooted at u are labeled, but before all vertices located in the right subtree rooted at u are labeled, then the traversal is called $in\text{-}order$ traversal.

CHAPTER 3
SIMULATION OF AHO′S ALGORITHM

In this chapter we present an overview of previous studies on Phylogenetic Tree, an elaborate demonstration of Aho′s algorithm and step by step simulation of this algorithm.

3.1 Introduction

Phylogenetic tree is one of the most interesting fields of data structure that is used to describe tree like evolutionary relationship among biological species or some other entities depending upon similarities and dissimilarities in their characteristics. Estimation of the tree plays a critical role in a wide variety of molecular studies and comparative genomics. In most of the cases, the tree construction is performed based on thousands of different protein alignment. Therefore construction becomes difficult in various situations. For very large set of data, constructing a correct tree representing the evolutionary relationship among the given species is futile most of the time. Moreover, though different studies shows different methods for tree construction, constructing within a small amount of time is far from us.

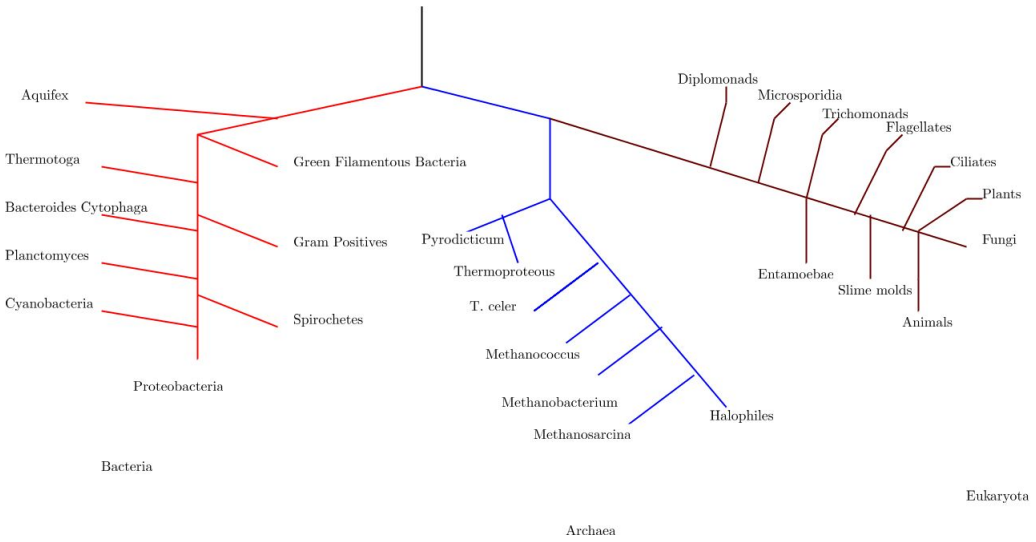

Figure 3.1: Phylogentic Tree of Life - A speculatively rooted tree for rRNA genes [1].

Figure 3.1 shows a speculatively rooted tree for rRNA genes. This tree representation showing the evolutionary relationship among the Bacteria, Archaea and Eucaryota. Thereby the major branches are shown with the most common species are shown in this figure.

For such problems we define a $Phylogenetic\ Tree$ as a rooted, unordered, distinctly leaf-labeled tree in which every internal node has at least two children [25]. For any rooted triplet $xy|z$ and Phylogenetic Tree T which also have leaves labeled x, y and z, if the lca in T of pair (x, y) is descendant of the lca of the pair (x, z), then $xy|z$ is consistent in T; otherwise they are inconsistent. Triplet consistency checking is another relevant topic of Phylogenetic Tree construction or related Phylogenetic studies. Figure 3.2 demonstrates such an example. Here the triplet is of the form $ab|c$. Next to it we have our Phylogenetic Tree. From the figure we can see that the given triplet is consistent on the tree since the lca of pair (a, b) is the descendant of the pair (a, c).

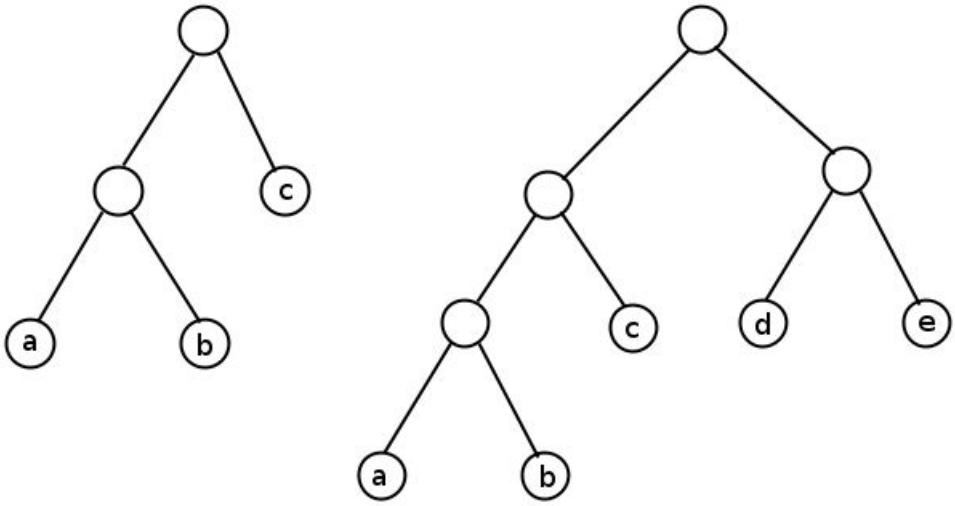

Figure 3.2: Triplet $ab|c$ is consistent in the phylogenetic tree T.

Several studies are performed till date in this topic. The subject is same but the field is different such as some research work is on Phylogenetic tree construction, some are on triplet consistency check, etc. But in 1981, Aho $etal.$ introduces an algorithm for Phylogenetic tree construction [14]. This polynomial time algorithm outputs a Phylogenetic tree that is consistent with every rooted triplet in the given triplet set, say R. If such a tree cannot be constructed from R, then it is not possible to construct such a tree with whom every triplet in the triplet set is consistent with. It is because the error in the triplets that prevents the consistency of the triplet with the constructed tree. The main focus of our thesis is to detect the erroneous triplet and correct it in the minimum elementary level so that the Phylogenetic tree can be constructed with whom all the triplets in the triplet set are consistent. So as a start we go through Aho′s procedure and simulate the algorithm with necessary examples to understand it and have a proper indication to achieve the proper goal of our thesis.

3.2 Stages of the Problem Analysis

In this section the problem solved by Aho is discussed in an elongated manner with proper example.

3.2.1 Problem Definition

In the problem definition constraints instead of triplets are considered. Therefore the tree is constructed from the given set of constraints. The constraints are of the form $(i, j) < (k, l)$ where $i \neq j$ and $k \neq l$ where the lca of the pair (i, j) is a proper descendant of pair (k, l). Here the order of i and j in (i, j) and of k and l in (k, l) are irrelevant. Therefore is it possible to construct a tree T from the set of constraints or such a tree does not exist is the main objective of this problem. Now considering the following example for proper understanding of the problem,

Example 1: For the given set of constraints,

$$(1, 2) < (1, 3)$$

$$(3, 4) < (1, 5)$$

$$(3, 5) < (2, 4)$$

The resulting tree is in Figure 3.3.

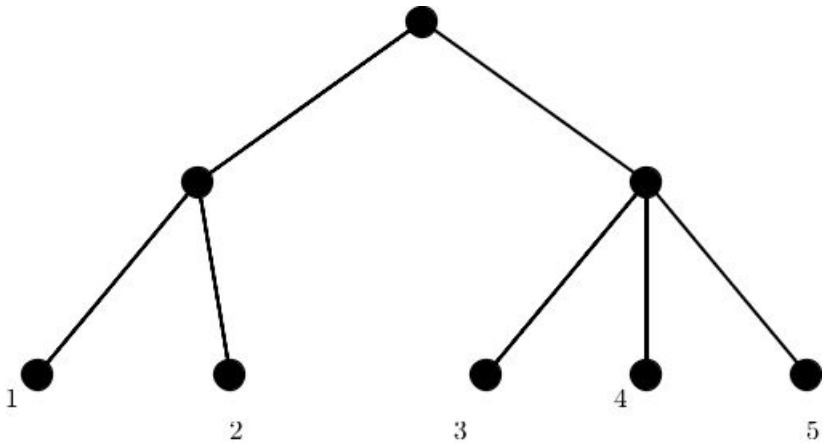

Figure 3.3: Solution for the constraints given in Example 1.

In Figure 3.3, a possible solution for the given constraint is considered where a proper Phylogenetic tree T is constructed. If we consider another constraint $(4, 5) < (1, 2)$ in the constraint set, then no tree can simultaneously satisfy all the constraints in the constraint set.

3.2.2 Analysis of the Problem and Solution Technique

In the main paper, it is stated that, the main idea behind the solution is to determine for a potential tree T the sets of leaves that are descendants of each child of the root of T [14]. Let these sets are, $S_1,\ S_2,\ \cdots, S_r$ where $r \geq 2$ this is because if a tree satisfying a set of constraints exists, then there must exist another tree satisfying the same set of constraints if we merge each node having one child with that child.

There are two conditions that these sets must satisfy for each constraint $(i, j) < (k, l)$. These conditions are,

1. In the first condition i and j must be in the same set. Otherwise pair (i, j) is the root of T where the root cannot be a proper descendant of the pair (k, l).
2. The last condition is, either k and l are in different sets, or $i,\ j,\ k\ and\ l$ are all together in one set. Otherwise the pair (i, j) cannot be a proper descendant of pair (k, l).

Therefore if we can partition the nodes into two or more sets satisfying the above two conditions and if we can recursively build trees for each set then a tree exists; otherwise one does not. If we define a partition by π_c for leaves $1,\ 2\ \cdots, n$ for the given set of constraints C following the rules,

1. If $(i, j) < (k, l)$ is the constraint, then i and j are in one block of π_c.
2. If $(i, j) < (k, l)$ is the constraint and k and l are in one block, then $i,\ j,\ k\ and\ l$ are all in the same block of π_c.
3. No two leaves are in the same block of π_c unless it follows from (1) and (2).

Therefore, we can see that the recursive algorithm presented by Aho is to build a tree T satisfying a set of constraint, say C where S is a nonempty set of nodes. If such a tree does not exist, it returns a null tree. The basic idea of the algorithm is to compute the partition c checking that if it has at last two blocks $r \geq 2$ and construct the sets of constraints C_m, $1 \leq m \leq r$, C_m is C restricted to those constraints that involve members of S_m only [14].

Up to this we discuss briefly about the main ideas of Ahos algorithm and different aspects of the algorithm. Now a step by step simulation of the algorithm is presented with proper example and demonstration of each step relating it with the algorithm to have a proper understanding on how does the algorithm works and construct a Phylogenetic Tree.

3.2.3 Aho′s Algorithm

In this section we present Aho′s algorithm that is used for simulation and provided by the author [14].

We present Aho′s algorithm in two steps. They are,

1. procedure BUILD(S, C)
2. procedure PIC()

The Procedure BUILD

procedure BUILD(S, C)
if S consists of a single node i then
  return the tree consisting of node i alone
else
  begin
    compute π_c = S_1, S_2, $\cdots$, S_r;
    if $r = 1$ **then**
      return the null tree
    else
      for $m := 1$ **to** r **do**
        begin
          C_m := $(i, j) < (k, l)$ in $C \mid i,\ j,\ k,\ l$ are in S_m;
          T_m := BUILD(S_m, C_m);
          if T_m = the null tree **then**
            return the null tree
        end;
    /* if we reach here a tree exists */
    let T be the tree with a new node for its root and whose children are the roots of T_m, $1 \leq m \leq r$;
    return T
  end
end BUILD

The General Case Partitioning Algorithm

procedure PIC()

1. **for** each leaf l mentioned in a constraint **do**
   begin
   set L_l to the empty list;
   set $S[l]$ to l;
   end;

2. **for** each constraint $(i, j) < (k, l)$ **do**
   begin
   let c be the implication $k \equiv l \rightarrow i \equiv l$;
   add c to $L_S[k]$;
   add c to $L_S[l]$;
   add the command $i \equiv j$ to Q;
   end;

3. **while** Q is not empty **do**
   begin
   remove a command $p \equiv q$ from Q;
   if $S[p] \neq S[q]$ **then**
   begin
   let L be the shorter of $L_S[p]$ and $L_S[q]$;
   for each implication $u \equiv v \rightarrow x \equiv y$ on L **do**
   if one of u and v is in $S[p]$ and the other is in $S[q]$ **then**
   add the command $x \equiv y$ to Q;
   append $L_S[p]$ to $L_S[q]$;
   merge $S[p]$ and $S[q]$;
   end;
   end;

end PIC

3.2.4 Simulation of the Algorithm

In this section we present an elaborative demonstration of Ahos algorithm with proper explication of that.

For the simulation of the algorithm we consider the following constraint set,

$$(1, 3) < (2, 5)$$

$$(1, 4) < (3, 7)$$

$$(2, 6) < (4, 8)$$

$$(3, 4) < (2, 6)$$

$$(4, 5) < (1, 9)$$

$$(7, 8) < (2, 10)$$

$$(7, 8) < (7, 10)$$

$$(8, 10) < (5, 9)$$

From the given constraint set we can see that the number of species in this case is 10, thus n = 1, 2, $\cdots$, 10. For the algorithm simulation we need a queue, set array and a list according to the algorithm. Initially the states of the queue Q, set array S and list L is as shown in Figure 3.4. From the figure, we can see that, the list L is allocated for 10 species, the queue is initially empty and in the set array each set allocated 10 species is initialized by the species itself.

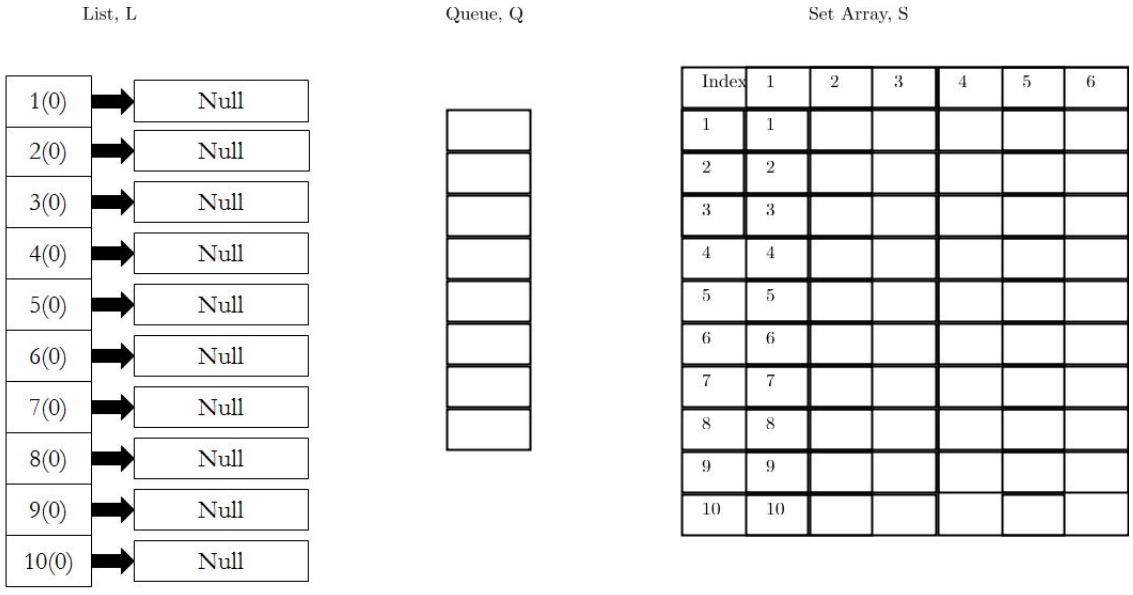

Figure 3.4: Initial stage of the list array L, queue Q and set array S.

Iteration Number 1

In Step 1 we consider each of the constraint one by one, convert it into a command and make changes to the initial state of the Q, L and S.

1. For the constraint 1 we have $(1,3) < (2,5)$, the implication for it is, $2 \equiv 5 \rightarrow 1 \equiv 5$ where $i = 1$, $j = 3$, $k = 2$, $l = 5$. Now $S[2] = 2$ and $S[5] = 5$. So this implication is added to the $L(2)$ and $L(5)$ list array and in queue $1 \equiv 3$ is added. Now the current situation of Q, L and S can be seen from Figure 3.5. From Figure 3.5 we can see that at $L(2)$ and $L(5)$ the implication is added and the command is added in queue.

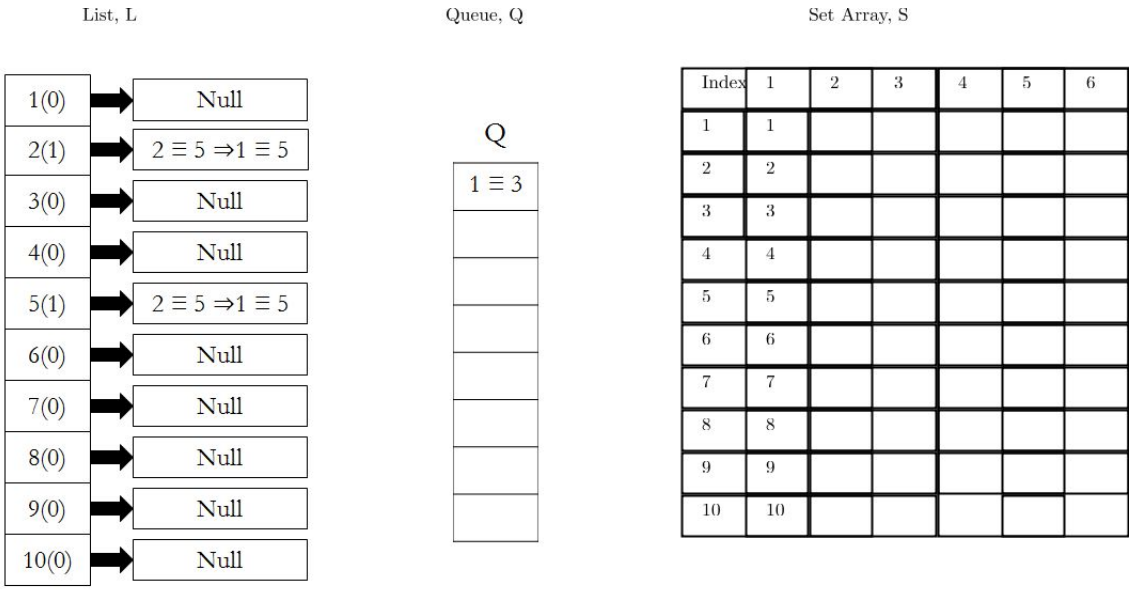

Figure 3.5: For the constraint $(1,3) < (2,5)$, the current state of the list array L, queue Q and set array S.

2. For the constraint 2 we have $(1,4) < (3,7)$, the implication for it is, $3 \equiv 7 \rightarrow 1 \equiv 7$ where $i = 1,\ j = 4,\ k = 3,\ l = 7$. Now $S[3] = 3$ and $S[7] = 7$. So this implication is added to the $L(3)$ and $L(7)$ list array and in queue $1 \equiv 4$ is added. Now the current situation of Q, L and S can be seen from Figure 3.6. From Figure 3.6 we can see that at $L(3)$ and $L(7)$ the implication is added and the command is added in queue.

3. For the constraint 3 we have $(2,6) < (4,8)$, the implication for it is, $4 \equiv 8 \rightarrow 2 \equiv 8$ where $i = 2,\ j = 6,\ k = 4,\ l = 8$. Now $S[4] = 4$ and $S[8] = 8$. So this implication is added to the $L(4)$ and $L(8)$ list array and in queue $2 \equiv 6$ is added. Now the current situation of Q, L and S can be seen from Figure 3.7. From Figure 3.7 we can see that at $L(4)$ and $L(8)$ the implication is added and the command is added in queue.

4. For the constraint 4 we have $(3,4) < (2,6)$, the implication for it is, $2 \equiv 6 \rightarrow 3 \equiv 6$ where $i = 3,\ j = 4,\ k = 2,\ l = 6$. Now $S[2] = 2$ and $S[6] = 6$. So this implication

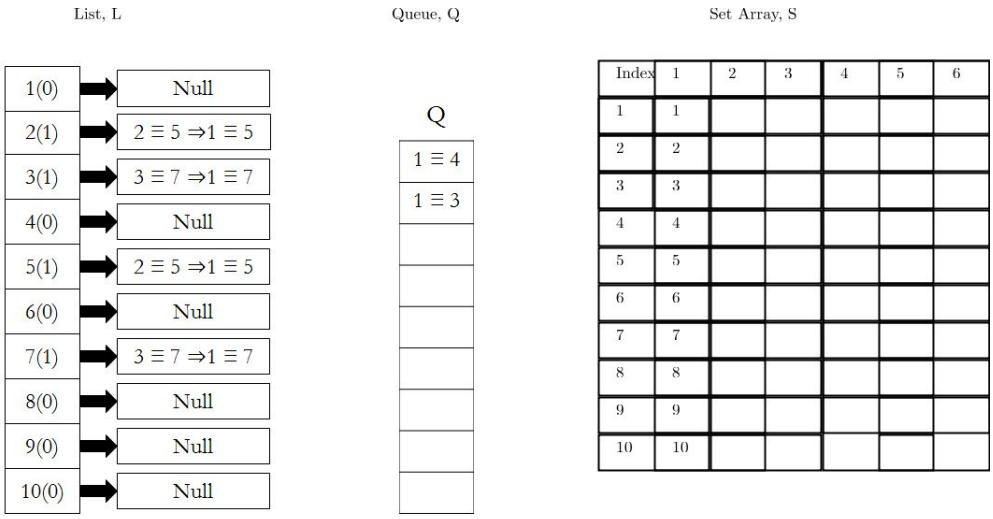

Figure 3.6: For the constraint $(1, 4) < (3, 7)$, the current state of the list array L, queue Q and set array S.

is added to the $L(2)$ and $L(6)$ list array and in queue $3 \equiv 4$ is added. Now the current situation of $Q,\ L$ and S can be seen from Figure 3.8. From Figure 3.8 we can see that at $L(2)$ and $L(6)$ the implication is added and the command is added in queue.

5. For the constraint 5 we have $(4, 5) < (1, 9)$, the implication for it is, $1 \equiv 9 \rightarrow 4 \equiv 9$ where $i = 4,\ j = 5,\ k = 1,\ l = 9$. Now $S[1] = 1$ and $S[9] = 9$. So this implication is added to the $L(1)$ and $L(9)$ list array and in queue $4 \equiv 5$ is added. Now the current situation of $Q,\ L$ and S can be seen from Figure 3.9. From Figure 3.9 we can see that at $L(1)$ and $L(9)$ the implication is added and the command is added in queue.

6. For the constraint 6 we have $(7, 8) < (2, 10)$, the implication for it is, $2 \equiv 10 \rightarrow 7 \equiv 10$ where $i = 7,\ j = 8,\ k = 2,\ l = 10$. Now $S[2] = 2$ and $S[10] = 10$. So this implication is added to the $L(2)$ and $L(10)$ list array and in queue $7 \equiv 8$ is added. Now the current situation of $Q,\ L$ and S can be seen from Figure 3.10. From Figure 3.10 we can see that at $L(2)$ and $L(10)$ the implication is added and the command is added in queue.

7. For the constraint 7 we have $(7, 8) < (7, 10)$, the implication for it is, $7 \equiv 10 \rightarrow 7 \equiv 10$ where $i = 7,\ j = 8,\ k = 7,\ l = 10$. Now $S[7] = 7$ and $S[10] = 10$. So this implication is added to the $L(7)$ and $L(10)$ list array and in queue $7 \equiv 8$ is added. Now the current situation of $Q,\ L$ and S can be seen from Figure 3.11. From Figure 3.11 we can see that at $L(7)$ and $L(10)$ the implication is added and the command is added in queue.

8. For the constraint 8 we have $(8, 10) < (5, 9)$, the implication for it is, $5 \equiv 9 \rightarrow 8 \equiv 9$ where $i = 8,\ j = 10,\ k = 5,\ l = 9$. Now $S[5] = 5$ and $S[9] = 9$. So this implication is added to the $L(5)$ and $L(9)$ list array and in queue $8 \equiv 10$ is added. Now the current

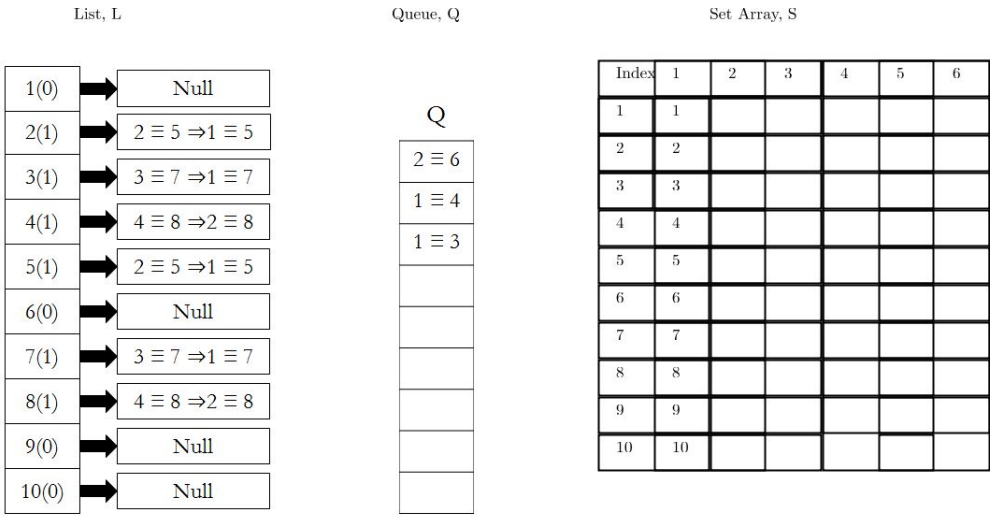

Figure 3.7: For the constraint $(2, 6) < (4, 8)$, the current state of the list array L, queue Q and set array S.

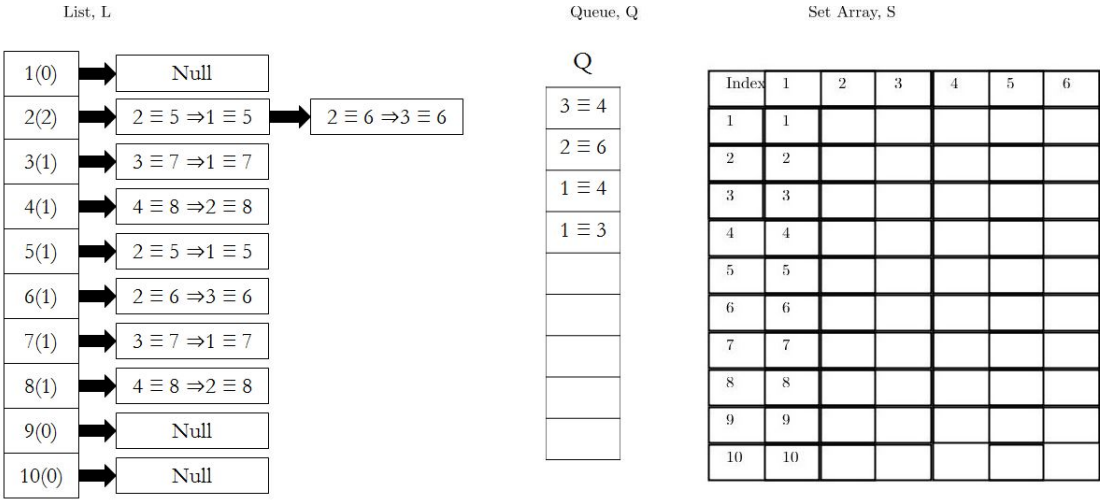

Figure 3.8: For the constraint $(3, 4) < (2, 6)$, the current state of the list array L, queue Q and set array S.

situation of Q, L and S can be seen from Figure 3.12. From Figure 3.12 we can see that at $L(5)$ and $L(9)$ the implication is added and the command is added in queue.

Now in Step 2, we consider each of the command in the queue and do the merge operation between set arrays and append lists.

1. Dequeue the command $1 \equiv 3$. For the command $1 \equiv 3$, $S[1] = 1$ and $S[3] = 3$, since $S[1] \neq S[3]$, choosing $L(1)$ be the shorter list. Therefore considering each of the implication in this list we get,

   - $1 \equiv 9 \rightarrow 4 \equiv 9$ where $u = 1$, $v = 9$, $x = 4$ and $y = 9$. Now $S[1] = u$ but $S[3] \neq v$. Therefore we proceed to the next step by merging $L(3)$ with $L(1)$ and

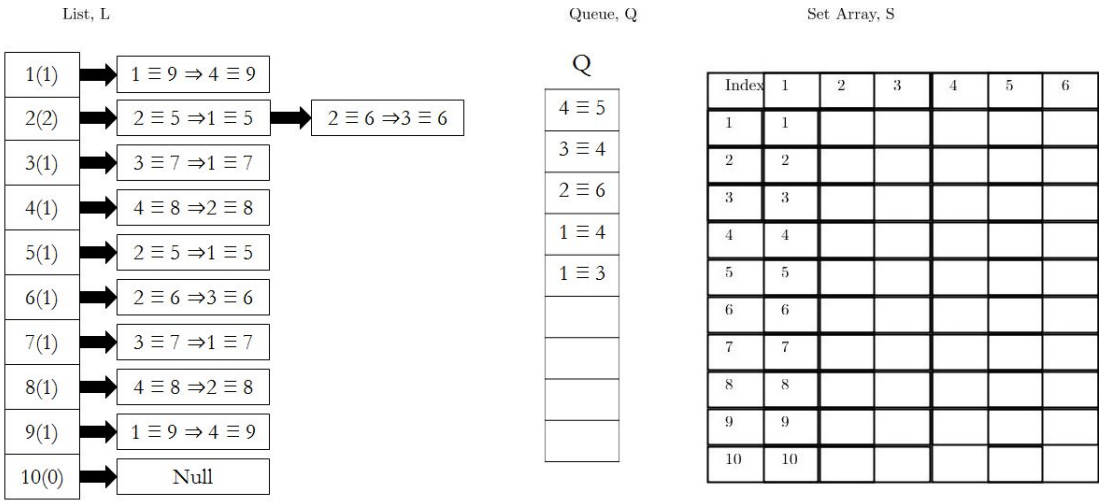

Figure 3.9: For the constraint $(4, 5) < (1, 9)$, the current state of the list array L, queue Q and set array S.

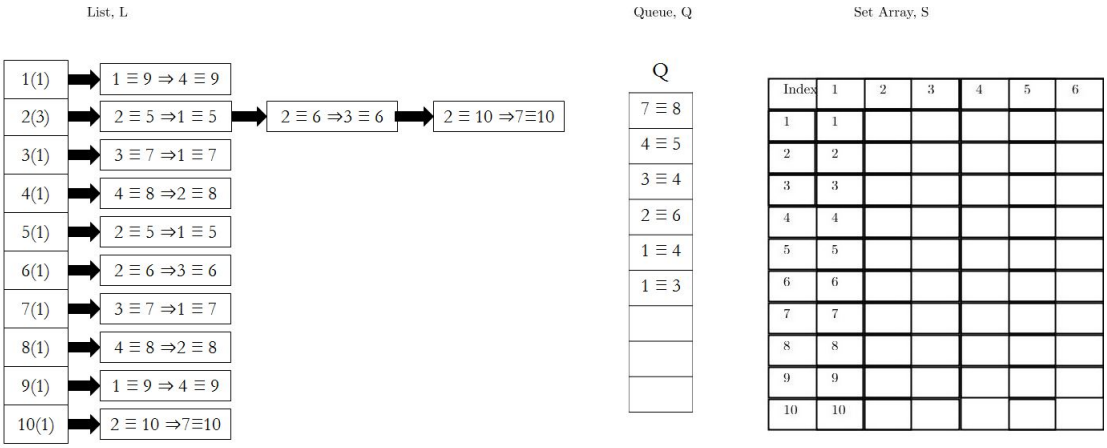

Figure 3.10: For the constraint $(7, 8) < (2, 10)$, the current state of the list array L, queue Q and set array S.

placing $S[3] = 1$.

2. Dequeue the command $1 \equiv 4$. For the command $1 \equiv 4$, $S[1] = 1$ and $S[4] = 4$, since $S[1] \neq S[4]$, choosing $L(4)$ be the shorter list. Therefore considering each of the implication in this list we get,

   - $4 \equiv 8 \rightarrow 2 \equiv 8$ where $u = 4,\ v = 8,\ x = 2$ and $y = 8$. Now $S[4] = u$ but $S[1] \neq v$. Therefore we proceed to the next step by merging $L(4)$ with $L(1)$ and placing $S[4] = 1$.

3. Dequeue the command $2 \equiv 6$. For the command $2 \equiv 6$, $S[2] = 2$ and $S[6] = 6$, since $S[2] \neq S[6]$, choosing $L(6)$ be the shorter list. Therefore considering each of the implication in this list we get,

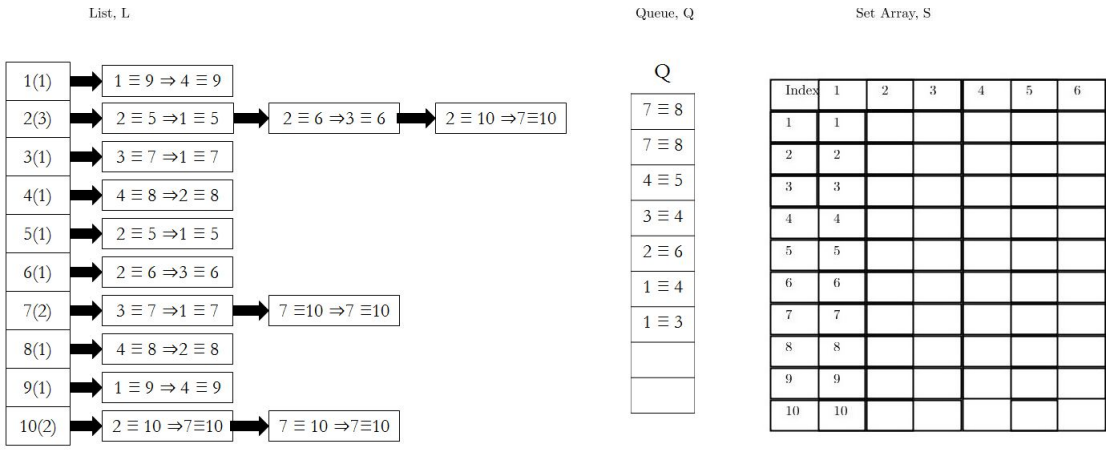

Figure 3.11: For the constraint $(7, 8) < (7, 10)$, the current state of the list array L, queue Q and set array S.

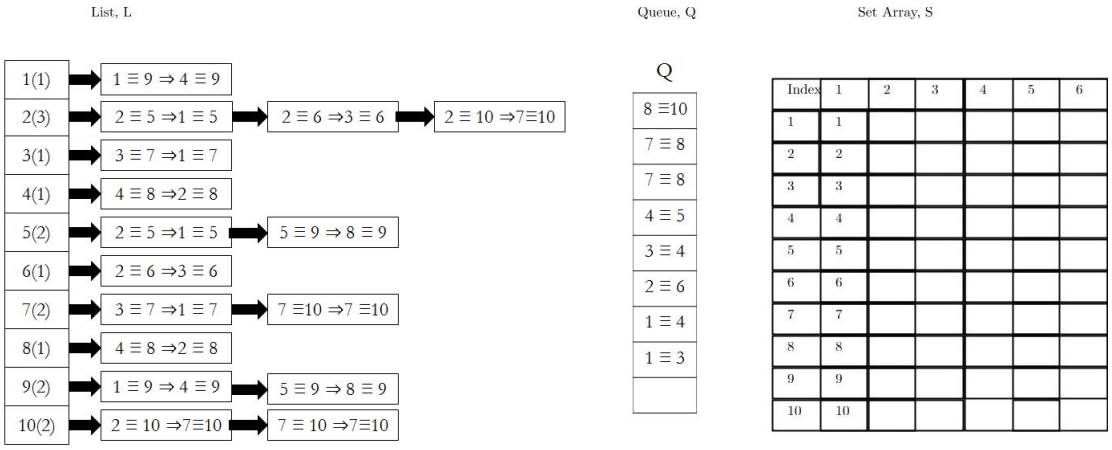

Figure 3.12: For the constraint $(8, 10) < (5, 9)$, the current state of the list array L, queue Q and set array S.

- $2 \equiv 6 \rightarrow 3 \equiv 6$ where $u = 2,\ v = 6,\ x = 3$ and $y = 6$. Now $S[2] = u$ and $S[6] = v$. Therefore add the command $3 \equiv 6$ in queue. Then we proceed to the next step by merging $L(6)$ with $L(2)$ and placing $S[6] = 2$.

4. Dequeue the command $3 \equiv 4$. For the command $3 \equiv 4,\ S[3] = 1$ and $S[4] = 1$, since $S[3] = S[4]$, we proceed to the next stage.

5. Dequeue the command $4 \equiv 5$. For the command $4 \equiv 5,\ S[4] = 1$ and $S[5] = 5$, since $S[4] \neq S[5]$, choosing $L(5)$ be the shorter list. Therefore considering each of the implication in this list we get,

   - $2 \equiv 5 \rightarrow 1 \equiv 5$ where $u = 2,\ v = 5,\ x = 1$ and $y = 5$. Now $S[5] = v$ but $S[4] \neq u$. Therefore we proceed to the next implication by merging $L(5)$ with $L(1)$ and placing $S[5] = 1$.

- $5 \equiv 9 \rightarrow 8 \equiv 9$ where $u = 5,\ v = 9,\ x = 8$ and $y = 9$. Now $S[5] = u$ but $S[4] \neq v$. Therefore we proceed to the next step.

6. Dequeue the command $7 \equiv 8$. For the command $7 \equiv 8,\ S[7] = 7$ and $S[8] = 8$, since $S[7] \neq S[8]$, choosing $L(8)$ be the shorter list. Therefore considering each of the implication in this list we get,

   - $4 \equiv 8 \rightarrow 2 \equiv 8$ where $u = 4,\ v = 8,\ x = 2$ and $y = 8$. Now $S[8] = v$ and $S[7] \neq u$. Therefore we proceed to the next step by merging $L(8)$ with $L(7)$ and placing $S[8] = 7$.

7. Dequeue the command $7 \equiv 8$. For the command $7 \equiv 8,\ S[7] = 7$ and $S[8] = 7$, since $S[7] = S[8]$, we proceed to the next step.

8. Dequeue the command $8 \equiv 10$. For the command $8 \equiv 10,\ S[8] = 7$ and $S[10] = 10$, since $S[8] \neq S[10]$, choosing $L(10)$ be the shorter list. Therefore considering each of the implication in this list we get,

   - $2 \equiv 10 \rightarrow 7 \equiv 10$ where $u = 2,\ v = 10,\ x = 7$ and $y = 10$. Now $S[10] = v$ and $S[8] \neq u$. Therefore we proceed to the next step by merging $L(10)$ with $L(7)$ and placing $S[10] = 7$.
   - $7 \equiv 10 \rightarrow 7 \equiv 10$ where $u = 7,\ v = 10,\ x = 7$ and $y = 10$. Now $S[8] = u$ and $S[10] = v$. Therefore add the command $7 \equiv 10$ in queue. Then we proceed to the next step.

9. Dequeue the command $3 \equiv 6$. For the command $3 \equiv 6,\ S[3] = 1$ and $S[6] = 2$, since $S[3] \neq S[6]$, choosing $L(2)$ be the shorter list. Therefore considering each of the implication in this list we get,

   - $2 \equiv 5 \rightarrow 1 \equiv 5$ where $u = 2,\ v = 5,\ x = 1$ and $y = 5$. Now $S[6] = u$ and $S[3] \neq v$. Therefore we proceed to the next step by merging $L(2)$ with $L(1)$ and placing $S[2] = 1$ and $S[6] = 1$.
   - $2 \equiv 6 \rightarrow 3 \equiv 6$ where $u = 2,\ v = 6,\ x = 3$ and $y = 6$. Now $S[6] = u$ and $S[3] \neq v$. Therefore we proceed to the next step.
   - $2 \equiv 10 \rightarrow 7 \equiv 10$ where $u = 2,\ v = 10,\ x = 7$ and $y = 10$. Now $S[6] = u$ and $S[3] \neq v$. Therefore we proceed to the next step.
   - $2 \equiv 6 \rightarrow 3 \equiv 6$ where $u = 2,\ v = 6,\ x = 3$ and $y = 6$. Now $S[6] = u$ and $S[3] \neq v$. Therefore we proceed to the next step.

10. Dequeue the command $7 \equiv 10$. For the command $7 \equiv 10,\ S[7] = 7$ and $S[10] = 7$, since $S[7] = S[10]$, we proceed to the next step.

After the above steps the queue is now empty. Therefore for all these steps, the list array and the set array will be changed and they are now of the form shown in Figure 3.13 and Figure 3.14. In Figure 3.13 and Figure 3.14 we can see that after the First iteration we have three sets of species. Therefore up to now tree construction is possible from the given constraint set.

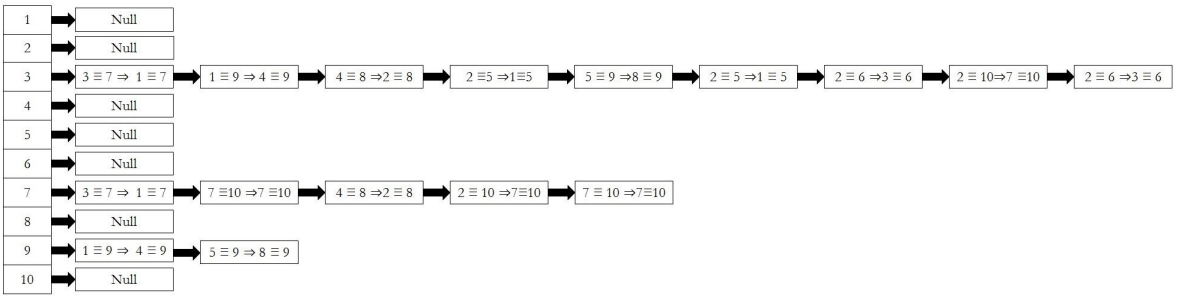

Figure 3.13: After Iteration 1 state of the list array L.

Index	1	2	3	4	5	6
1	1	2	3	4	5	6
2						
3						
4						
5						
6						
7	7	8	10			
8						
9	9					
10						

Figure 3.14: After Iteration 1 state of the set array S.

In Figure 3.15 we see the initial tree after first iteration. In the successive iterations this tree is going to have a tree like structure. According to the simulations shown above here we have three branches in the tree.

In Step 3 we consider now the newly created species sets and repeat Steps 1 and 2 recursively until we have a single element in the tree. Now considering the three sets we have,

$$S1 = 1, 2, 3, 4, 5, 6$$

$$S2 = 7, 8, 10$$

$$S3 = 9$$

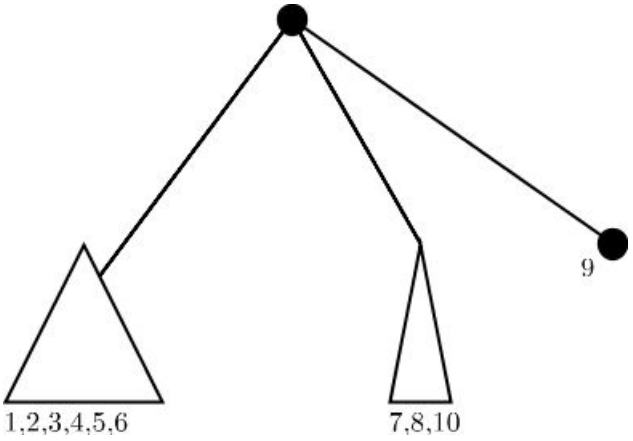

Figure 3.15: After Iteration 1 state of the tree.

Iteration Number 2

In iteration number 2 we consider these three sets. Since S_3 has a single node the recursive call is not applicable for it. Now for set S1 we consider the following set of constraints, C_1,

$$(1,3) < (2,5)$$

$$(3,4) < (2,6)$$

And again for set S_2 we consider the following set of constraints, C_2,

$$(7,8) < (7,10)$$

Since it is a recursive call, first S_1 with C_1 call is made. Until every node in this branch is explored and tree like structure is obtained, this recursion continues. After this iteration we obtain a tree that looks like the following one given in Figure 3.16. From Figure 3.16 we can see that this time we have four sets from branch one.

The four sets are,

$$S1 = 1, 3, 4$$

$$S2 = 2$$

$$S3 = 5$$

$$S4 = 6$$

Since set S_2, S_3 and S_4 contains single element. They are not candidate for the recursive call. Therefore the candidate of recursive call is set S_1.

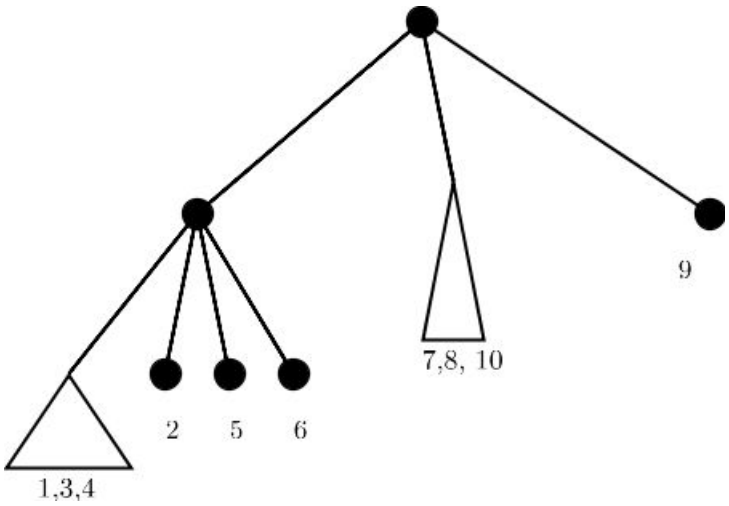

Figure 3.16: After Iteration 2 state of the tree.

Iteration Number 3

For the next iteration we consider set S_1 and for S_1 we have the following constraint set C_1 = $NULL$. Since none of the constraint is supporting this set. Therefore after this iteration the structure of the tree is something like Figure 3.17.

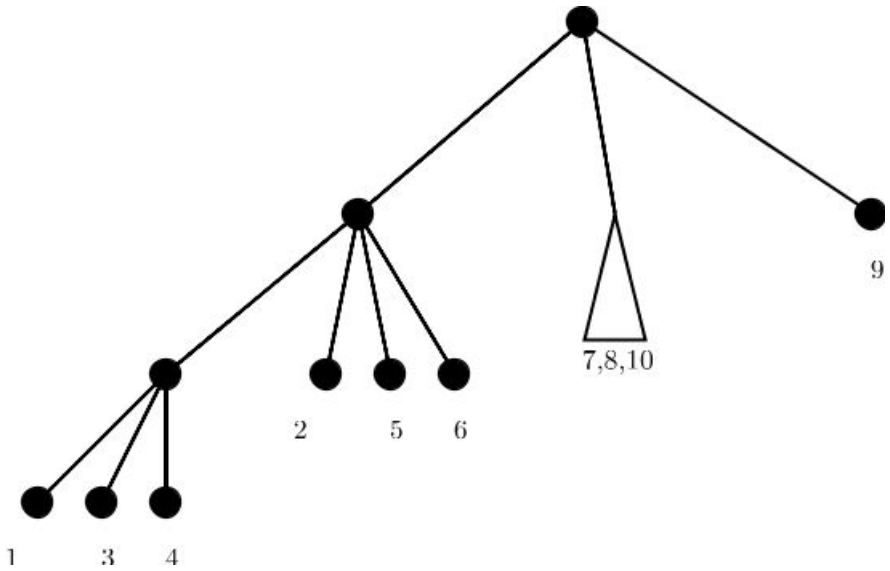

Figure 3.17: After Iteration 3 state of the tree.

From Figure 3.17 we can see that the branch 1 is completely have a tree like structure since in this iteration we get three sets each containing a single element. Therefore exploration of branch 1 is now completed. Now we have branch 2 to be explored. Therefore the recursion moves to the initial stage.

Iteration Number 4

In the initial stage we have,

$$S2 = 7, 8, 10$$

$$C2 = (7, 8) < (7, 10)$$

With S_2 and C_2 we make the call and have a tree that looks like the one in Figure 3.18.

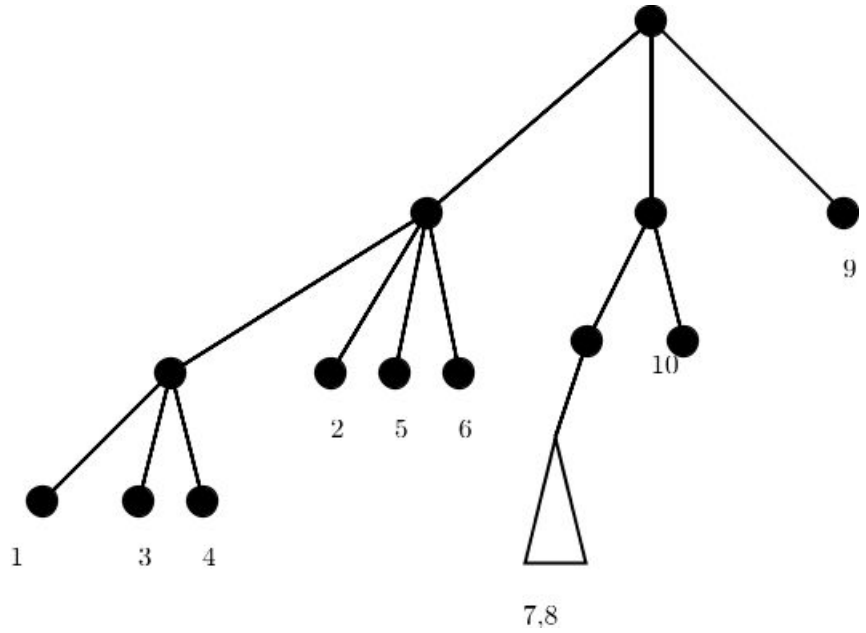

Figure 3.18: After Iteration 4 state of the tree.

From Figure 3.17 we can see that after this iteration we get two sets,

$$S1 = 7, 8$$

$$S2 = 10$$

Since S_2 has a single component the recursive call is performed with S_1.

Iteration Number 5

In this stage we have,

$$S2 = 7, 8$$

$$C2 = NULL$$

Again the constraint set is null since there is no compatible constraint set with this set of nodes. For this we have the tree that looks like the one given in Figure 3.19. Since the

constraint set is empty therefore this iteration returns two sets each containing a single node. Here the tree construction process is completed.

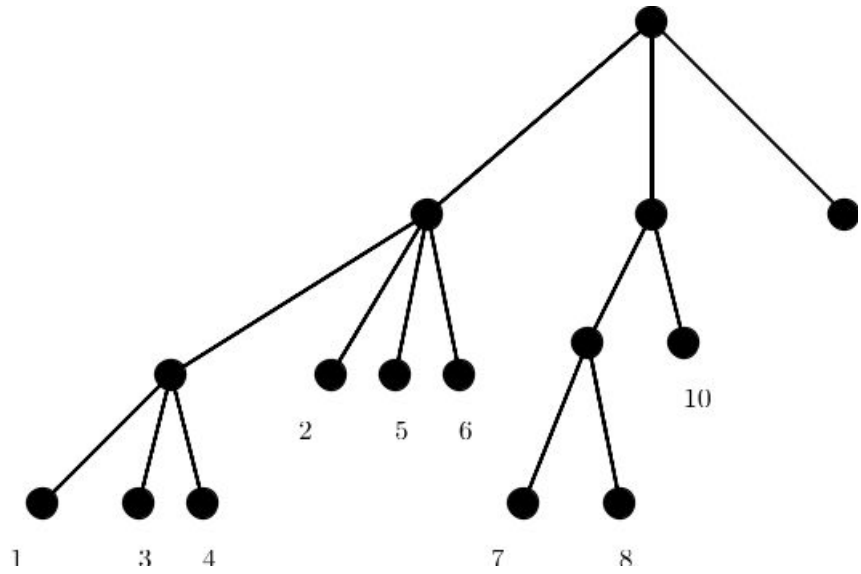

Figure 3.19: After Iteration 5 state of the tree.

3.3 Conclusion

Aho′s algorithm is one of the mostly used and well known algorithms for Phylogenetic Tree construction. Because it is easy and well understood. It recursively construct the tree that is consistent with all the constraints in the given constraint set. Using this idea of construction several research work has been done till date. We also use this idea in a different manner to do our thesis work on Phylogenetic tree construction and it is really been a helpful material for us.

CHAPTER 4
PHYLOGENETIC TREES WITH ERRONEOUS DATA

4.1 Introduction

Phylogenetic Tree construction considering the maximum number of triplets in the given triplet set is common. But while considering only those triplets that are consistent with the Phylogenetic Tree and ignoring others, results in loss of important evolutionary relationship. In this thesis, we investigate some process how we can reduce this practice.

For the triplet because of which tree cannot be constructed are considered to be erroneous. By making elementary level changes in the triplet or correcting it this error can be removed and a proper Phylogenetic Tree can be constructed. The main target is to reduce the loss of data ensuring the construction of the tree. Moreover while correcting the triplet our main concern is to do minimum number of changes in it as well as changing it in a way so that the species organization in the tree is approximately a sparse distribution.

In this section we present the overall procedure to solve the problem and a heuristic algorithm to solve it.

4.2 Problem Definition

Let P(T) denote the set of all triplets consistent with a given tree T. The tree consists of n species and the triplet set consists of m triplets. Each triplet is of the form $xy|z$ where the constraint imposed is of the form $(x, y) < (x, z)$. That is, the lowest common ancestor of the pair (x, y) is a proper descendant of the lowest common ancestor of the pair (x, z). At any point of the construction, if any triplet of the form $(i, j) < (k, l)$ acts inconsistent with the tree T, then the triplet is corrected and after that construction will proceed so that the resulting tree is consistent with all the triplets in the set. The process requires minimum number of changes to the triplets.

4.3 Stages of the Problem Analysis

In this section we present the real problem and discuss the solution of it and how it works. Identify separate stages of the problem and solution technique of the problem.

The problem is divided into some stages where different part of the problem is considered. The stages we consider here are,

1. Identification of the Erroneous Triplet.
2. Correction of the Erroneous Triplet.
3. Construction of the Tree.

4.3.1 Identification of the Erroneous Triplet

In this section we present the way to detect the erroneous triplet from the triplet set.

Suppose we have $n = 5$ species for which we are given the triplet set consists of $m = 4$ triplets of the form,

$$12|3$$

$$13|4$$

$$45|2$$

$$35|2$$

Now if we consider the step by step simulation of the problem then we will consider graph simulation because it is faster to know whether the given triplet set can build the tree or not. Moreover at any point of time if any one triplet constructs a single graph component then we need to correct that triplet and then proceed to the next triplet in the set. Thereby, before going through the triplet set the species are of the form that are shown in Figure 4.1 where there are 5 nodes initially.

Figure 4.1: Initial state of the nodes.

For the triplet $12|3$ there is an edge between the pair $(1, 2)$. Similarly for the triplet $13|4$ and $45|2$ we have edge $(1, 3)$ and $(4, 5)$. In Figure 4.2 the stages are shown for the triplets $12|3$, $13|4$ and $45|2$ respectively. In the first stage, for the triplet $12|3$ an edge between node 1 and

2 is added. Then for the triplet $13|4$ another edge between node 1 and 3 is added. Then in the ending stage, for the triplet $45|2$ an edge is added between node 4 and 5.

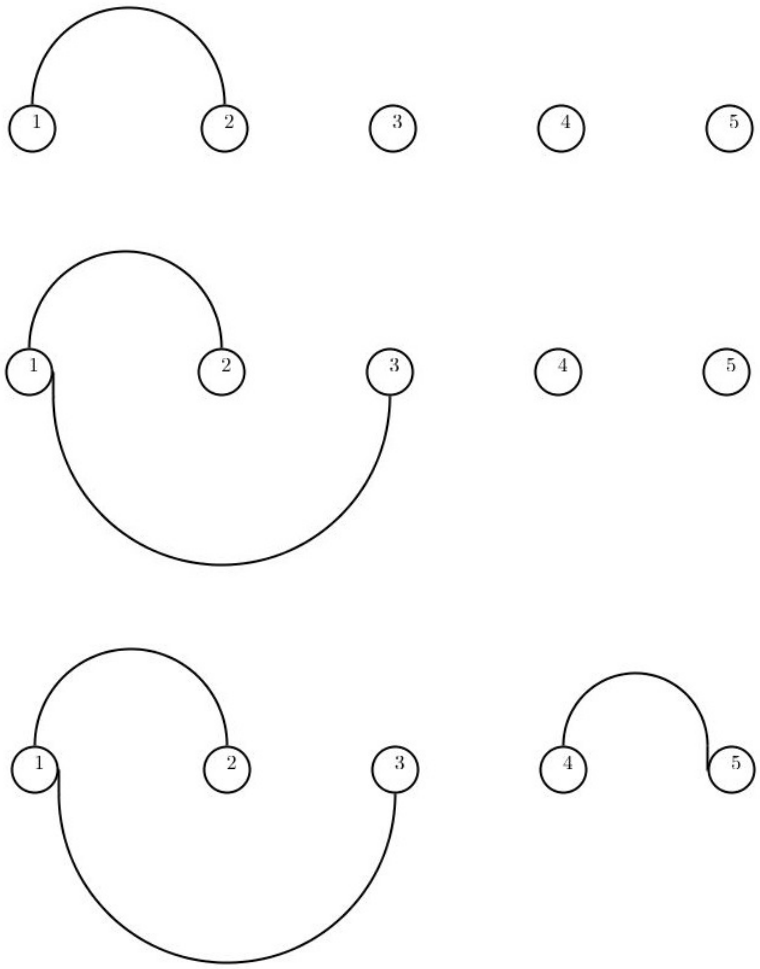

Figure 4.2: Stages of graph construction for different triplets.

But for the triplet $35|2$ it becomes a single component which means there is an error in the triplet. In Figure 4.3 the graph becomes a single component for the triplet $35|2$ after adding edge between node 3 and 5.

Now we consider this triplet as a candidate of correction and correct it.

4.3.2 Correction of the Erroneous Triplet

In this section we present the way to correct the error in a way so that the species in the tree is sparsely distributed in the tree.

Once the erroneous triplet is determined our next task is to correct the error. In order to correct the error we consider the erroneous triplet first. Then the degree of each of the node in the pair. Such as, for the i-th triplet $xy|z$ the pair is (x, y). We determine the degree of

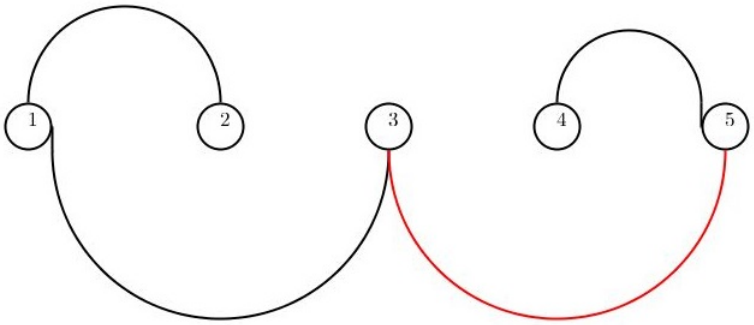

Figure 4.3: For the triplet $35|2$ error occurs (Red Edge).

node x and y in the grpah formed using upto $(i-1)$-th triplet. Here we look for the node that has the minimum degree so that if we add an edge with it, there will be an increase in its degree. Suppose x has the minimum degree. Then we check which component consists of this node. Then in this component we check for all node and find out one that has the minimum degree amongst them all. Suppose that node is w. Then we add an edge between node w and x. The triplet then becomes $xw|z$.

Suppose considering for the above example, we get that, the erroneous triplet is $35|2$. At this point of time we have two components. Those are,

$$C1 = 1, 2, 3$$

$$C2 = 4, 5$$

Therefore adding an edge between 3 and 5 converts it into one component. Thereby to avoid it we follow the above method. Here, both the degree of 3 and 5 in the graph is 1. Therefore we can consider any of them having minimum degree. Let us consider 5. Now we can check it with other nodes in the component. For them we have, 4 has the minimum degree. Therefore an edge is added between 4 and 5. The triplet becomes $45|2$. The error is removed.

After correction graph construction continues until such error occurs again. Then the above procedure will be continued. When all triplets in the triplet set is considered and the graph construction is completed with less than or equal to 2 components, the tree construction starts.

4.3.3 Construction of the Tree

In this section we present a way to construct the tree using the triplets.

Once all the triplets are corrected, the tree construction procedure takes place. For the Tree

construction process we follow any of the algorithms introduced before. Since the problem of erroneous triplet is removed all algorithms will output a proper Phylogenetic Tree.

The above procedure can now construct a Phylogenetic Tree that is consistent with all the triplets in the triplet set. Therefore we better explain the overall procedure in terms of algorithm. The $AllRTC$ algorithm present the above procedure.

4.4 Simulation of the Algorithm

In this section we simulate the overall procedure of **Algorithm** $AllRTC$ to construct a phylogenetic tree considering all triplets in the triplet set doing minimum changes to the erroneous triplets in the elementary level.

For our simulation we consider the following triplet set,

$$12|3$$

$$34|1$$

$$25|4$$

$$12|4$$

$$14|2$$

In the Identification of the Erroneous Triplet stage, we consider the triplets one by one. From the given triplet set we can see that we have total 5 species. Therefore the initial state of the graph is shown in Figure 4.4.

Figure 4.4: Initial state of the graph.

Now for the triplet $12|3$ we add an edge between the pair (1, 2). Figure 4.5 shows the change in the graph. Since after adding the edge we have total four components in the graph. Therefore, this triplet is not erroneous.

For the triplet $34|1$ we add an edge between the pair (3, 4). Figure 4.6 shows the change in the graph. Since after adding the edge we have total three components in the graph. Therefore, this triplet is not erroneous.

For the triplet $25|4$ we add an edge between the pair (2, 5). Figure 4.7 shows the change in the graph. Since after adding the edge we have total two components in the graph. Therefore,

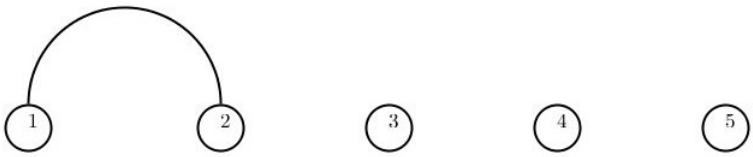

Figure 4.5: State of the graph considering the triplet $12|3$.

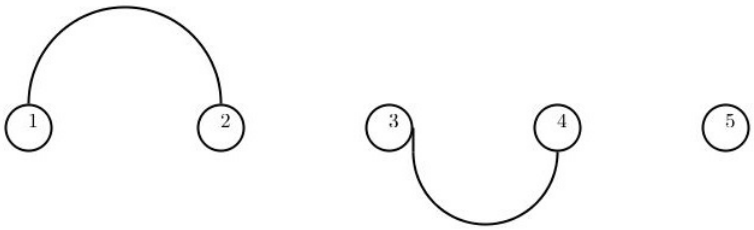

Figure 4.6: State of the graph considering the triplet $34|1$.

this triplet is not erroneous.

For the triplet $12|4$ we add an edge between the pair (1, 2). Since after adding the edge we have total two components in the graph. Therefore, this triplet is not erroneous.

For the triplet $14|2$ we add an edge between the pair (1, 4). Figure 4.8 shows the change in the graph. Since after adding the edge we have total one components in the graph. Therefore, this triplet is erroneous and a proper candidate for correction.

In the Correction of the Erroneous Triplet step we consider the triplet $14|2$. Here the degree of node 1 and node 4 is 1. Now considering node 4 has the lowest degree. In its component it has only node 3. So the correct triplet is $34|2$. Therefore the correct triplet set is,

$$12|3$$

$$34|1$$

$$25|4$$

$$12|4$$

$$34|2$$

Since there is no error in the triplet set, therefore if we apply Aho′s algorithm to construct the tree that looks like the one given in Figure 4.9

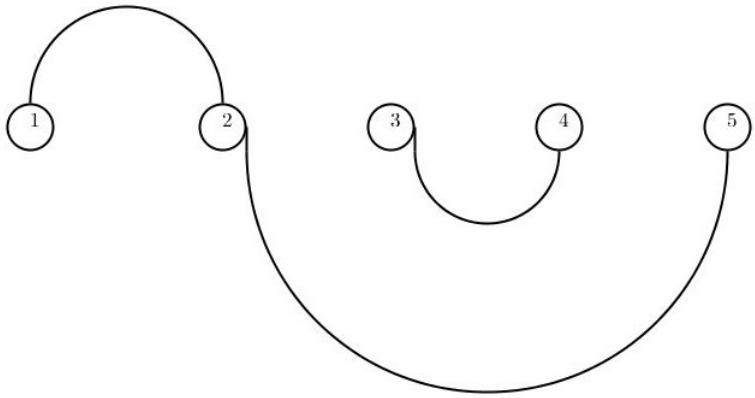

Figure 4.7: State of the graph considering the triplet $25|4$.

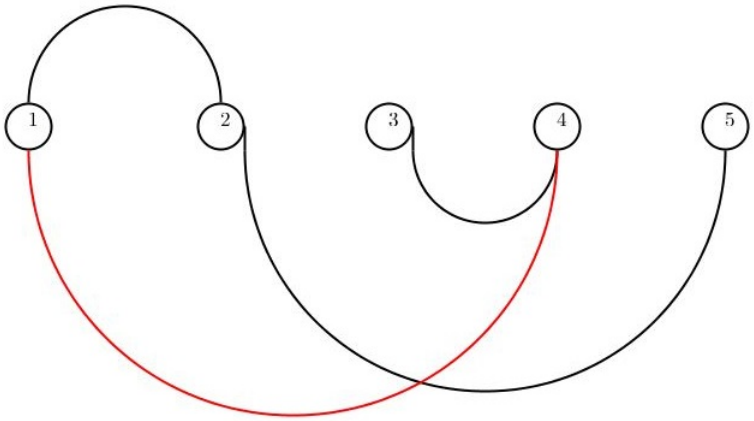

Figure 4.8: For the triplet $14|2$ error occurs (Red Edge).

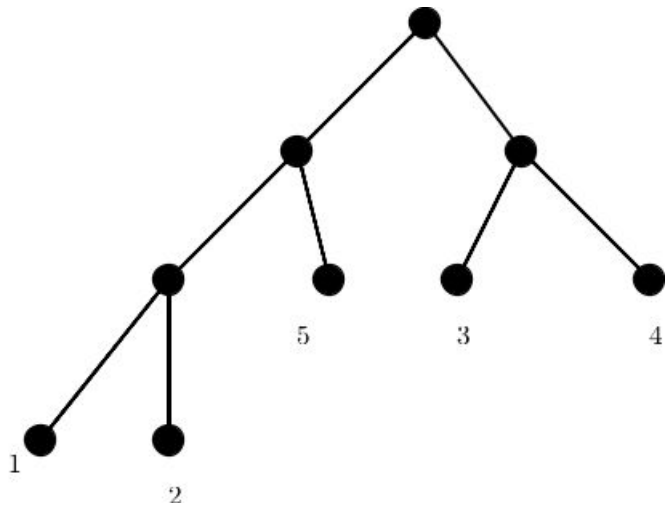

Figure 4.9: Tree for the given triplet set.

4.5 Procedure of Algorithm AllRTC

Here we give an Algorithm 1 to determine error in triplets and correct it to construct a phylogenetic tree considering all triplets in the triplet set, which we call the $AllRTC$ Algorithm.

Algorithm 1 Algorithm AllRTC

Require: $n > 0 \ \wedge m > 0 \wedge$ Triplet Set $P \wedge$ Set array S
Ensure: Phylogenetic Tree Construction
  $Q \leftarrow \phi$
  Initialize S with n sets with the species itself
  for each Triplet of the form $xy|z$ **do**
    $Q \leftarrow (x, y)$
  end for
  while $Q \neq \phi$ **do**
    $(u, v) \leftarrow Q.dequeue()$
    Add (u, v) to G
    Merge $S(u)$ with $S(v)$
    if G contains single component **then**
      Delete edge (u, v)
      if $degree(u) < degree(v)$ **then**
        $j \leftarrow u$
      else
        $j \leftarrow v$
      end if
      for each species in $S(j)$ **do**
        Calculate n = Minimum Degree Species
      end for
      $Q \leftarrow (j, n)$
    end if
  end while
  for each Triplet in P **do**
    Construct Tree T
  end for

4.6 Experiment with Real Data Set

The correctness and efficiency of the given algorithm is tested using some real data set. Those information is based on some real species. Construction of a Phylogenetic tree from this algorithm is tested to measure the correctness of it with this real triplet set. After the construction the tree is tested with the real one and measure the percentage of error.

4.6.1 Phylogeny of Lizards

For this example we consider the Phylogeny of Lizards [3]. The phylogeny of lizards include some species namely $C.tigris$, $D.dorsalis$, $C.draconoides$, $U.scoparia$ and $P.platyrhinos$. For the ease of our experiment we assign numbers to each of these species. The numbering is done as following,

$$C.tigris = 1$$

$$D.dorsalis = 2$$

$$C.draconoides = 3$$

$$U.scoparia = 4$$

$$P.platyrhinos = 5$$

For the construction of the phylogenetic tree construction we consider the following triplet set,

$$34|5$$

$$25|1$$

$$53|2$$

$$14|2$$

Now In the Identification of the Erroneous Triplet stage we consider each triplets one by one. Initially the graph looks like the one given in Figure 4.10. Each of the node represents any of the species. Therefore the total number of component is 5.

Figure 4.10: Initial Stage of the Graph.

For the triplet $34|5$, we add an edge between node 3 and node 4. This edge addition re-

sults in a graph that looks like the one given in Figure 4.11. After this the total number of components are 4.

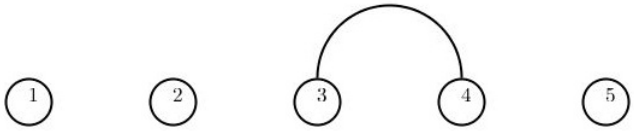

Figure 4.11: State of the graph considering the triplet $34|5$.

For the triplet $25|1$, we add an edge between node 2 and node 5. This edge addition results in a graph that looks like the one given in Figure 4.12. After this the total number of components are 3.

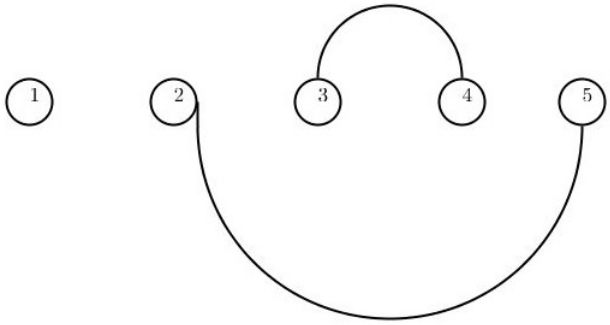

Figure 4.12: State of the graph considering the triplet $25|1$.

For the triplet $53|2$, we add an edge between node 5 and node 3. This edge addition results in a graph that looks like the one given in Figure 4.13. After this the total number of components are 2.

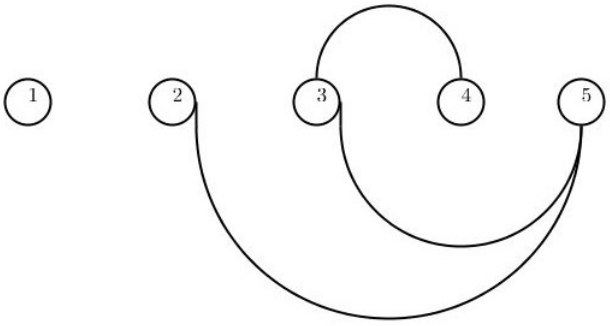

Figure 4.13: State of the graph considering the triplet $53|2$.

For the triplet $14|2$, we add an edge between node 1 and node 4. This edge addition results in a graph that looks like the one given in Figure 4.14. After this the total number of components are 1. Therefore this triplet is erroneous and a candidate for correction.

In the Correction of the Erroneous Triplet step we consider the triplet $14|2$. Here the degree of node 1 is 0 and node 4 is 1. Now node 1 has the lowest degree. Since its degree is 0 we

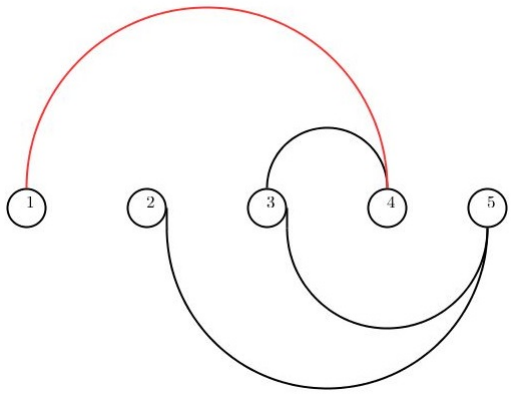

Figure 4.14: State of the graph considering the triplet $14|2$.

cannot consider it. Therefore we consider node 4 in this case. In its component it has node $2,\ 3$ and 5. The degrees of these nodes are $1,\ 2$ and 2. Though node 2 has the minimum degree, we cannot consider it since it is already present in the triplet. So, we add an edge between node 4 and 3. Therefore the correction in the triplet forms $34|2$. Therefore the correct triplet set is,

$$34|5$$

$$25|1$$

$$53|2$$

$$34|2$$

Since there is no error in the triplet set, therefore if we apply Aho′s algorithm to construct the tree that looks like the one given in Figure 4.15

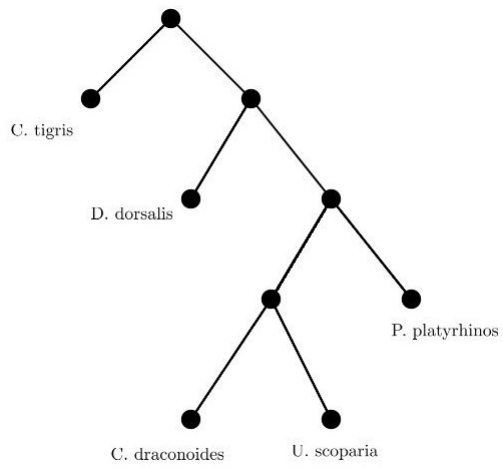

Figure 4.15: Phylogenetic Tree showing Evolutionary Relationship of Lizards.

The constructed tree is correct. But while correcting the choice of node 5 instead of node 3 is also correct. So confusion may arise. Moreover node 2 is not considered, since it is already in the triplet. So additional check is required.

4.6.2 Phylogeny of Yeast

The Fungal Biodiversity Center in Utrecht (Netherlands) provides some real yeast data. From this data set some triplets are produced [26]. For this simulation we consider only 20 triplets for proper understanding of the working procedure of our algorithm. For this example the triplet set is,

10 11|1

10 12|1

10 13|11

10 14|15

12 19|14

13 17|1

14 18|15

15 16|1

16 18|4

16 20|1

17 21|4

2 6|8

2 3|15

3 16|20

4 11|19

4 5|1

8 12|11

7 19|6

8 9|1

1 5|6

Now In the Identification of the Erroneous Triplet stage we consider each triplets one by one. Initially the graph looks like the one given in Figure 4.16. Each of the node represents any of the species. Therefore the total number of component is 21.

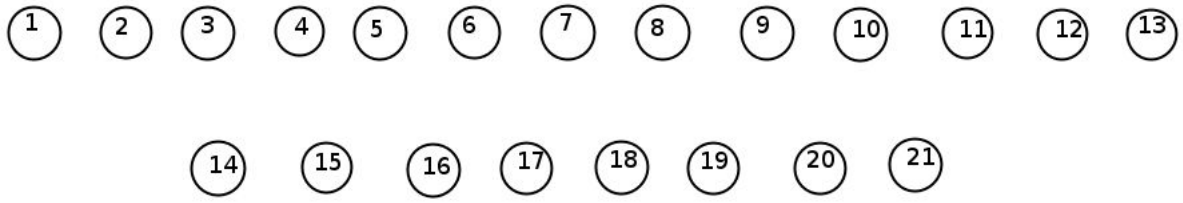

Figure 4.16: Initial Stage of the Graph.

For the triplet 10 11|1, we add an edge between node 10 and node 11. This edge addition results in a graph that looks like the one given in Figure 4.17.

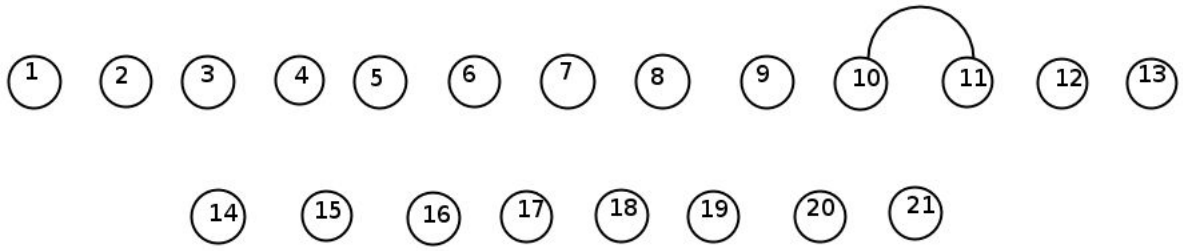

Figure 4.17: State of the graph considering the triplet 10 11|1.

For the triplet 10 12|1, we add an edge between node 10 and node 12. This edge addition results in a graph that looks like the one given in Figure 4.18.

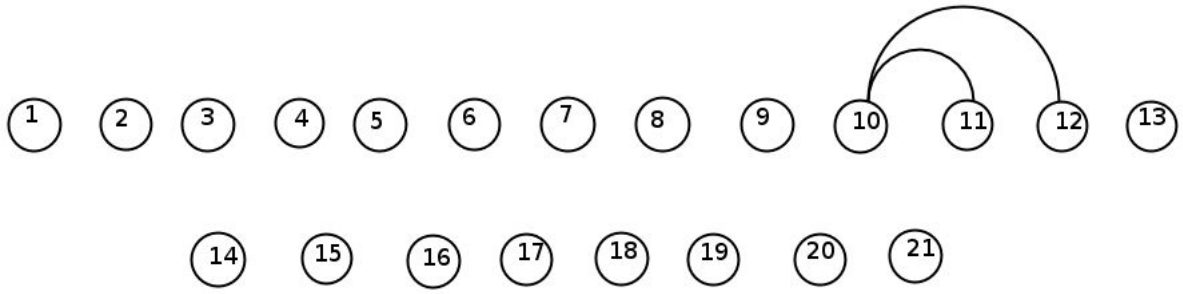

Figure 4.18: State of the graph considering the triplet 10 12|1.

For the triplet 10 13|11, we add an edge between node 10 and node 13. This edge addition results in a graph that looks like the one given in Figure 4.19.

For the triplet 10 14|15, we add an edge between node 10 and node 14. This edge addition results in a graph that looks like the one given in Figure 4.20.

For the triplet 12 19|14, we add an edge between node 12 and node 19. This edge addition results in a graph that looks like the one given in Figure 4.21.

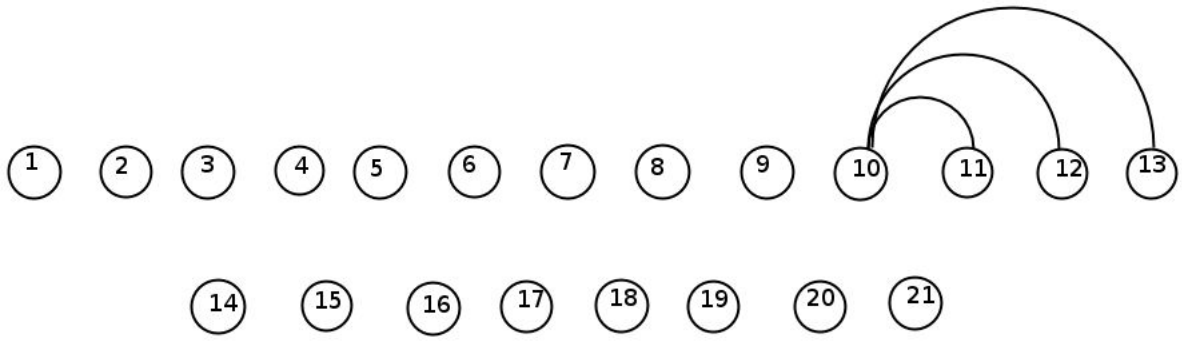

Figure 4.19: State of the graph considering the triplet $10\ 13|11$.

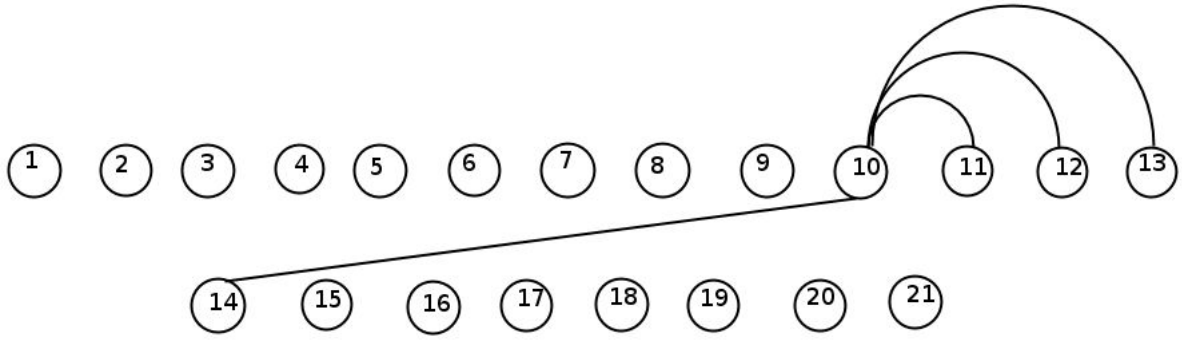

Figure 4.20: State of the graph considering the triplet $10\ 14|15$.

Then for rest of the triplets in the triplet set, $13\ 17|1$, $14\ 18|15$, $15\ 16|1$, $16\ 18|4$, $16\ 20|1$, $17\ 21|4$, $2\ 6|8$, $2\ 3|15$, $3\ 16|20$, $4\ 11|19$, $4\ 5|1$, $8\ 12|11$, $7\ 19|6$, $8\ 9|1$, we add edges in the graph that results in a graph that looks like the one given in Figure 4.22.

But for the triplet $1\ 5|6$, we add an edge between node 1 and node 5. This edge addition results in a graph that looks like the one given in Figure 4.23. Since addition of this edge results in a graph that has a single component, therefore this triplet is erroneous and a candidate of correction.

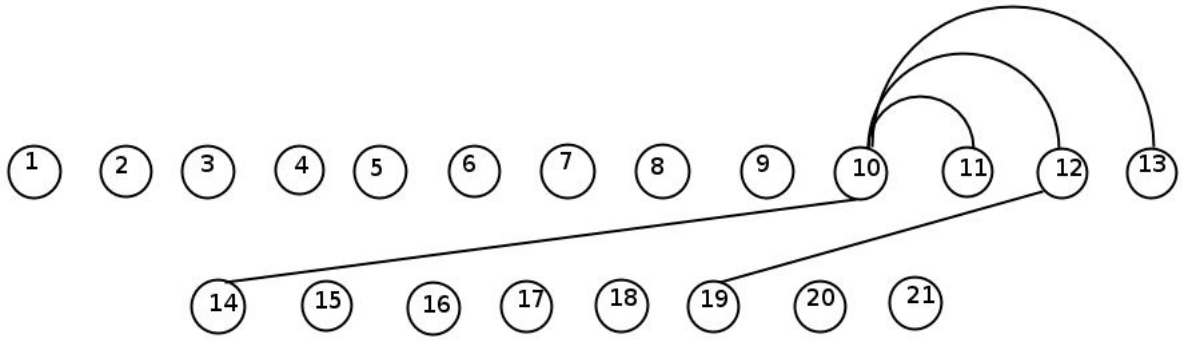

Figure 4.21: State of the graph considering the triplet $12\ 19|14$.

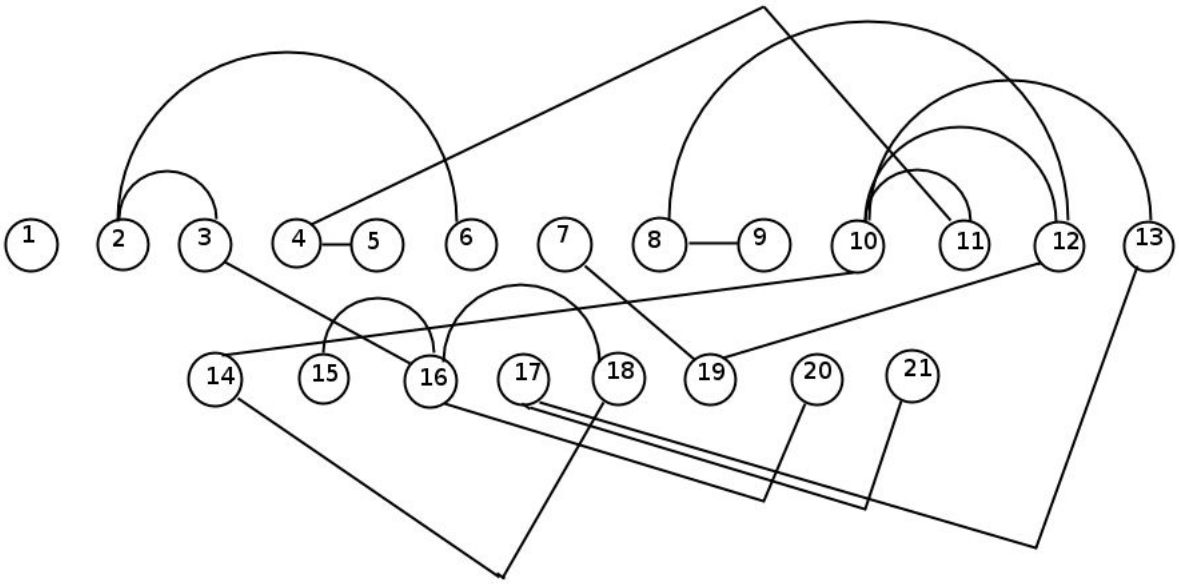

Figure 4.22: State of the graph considering the triplets in the triplet set.

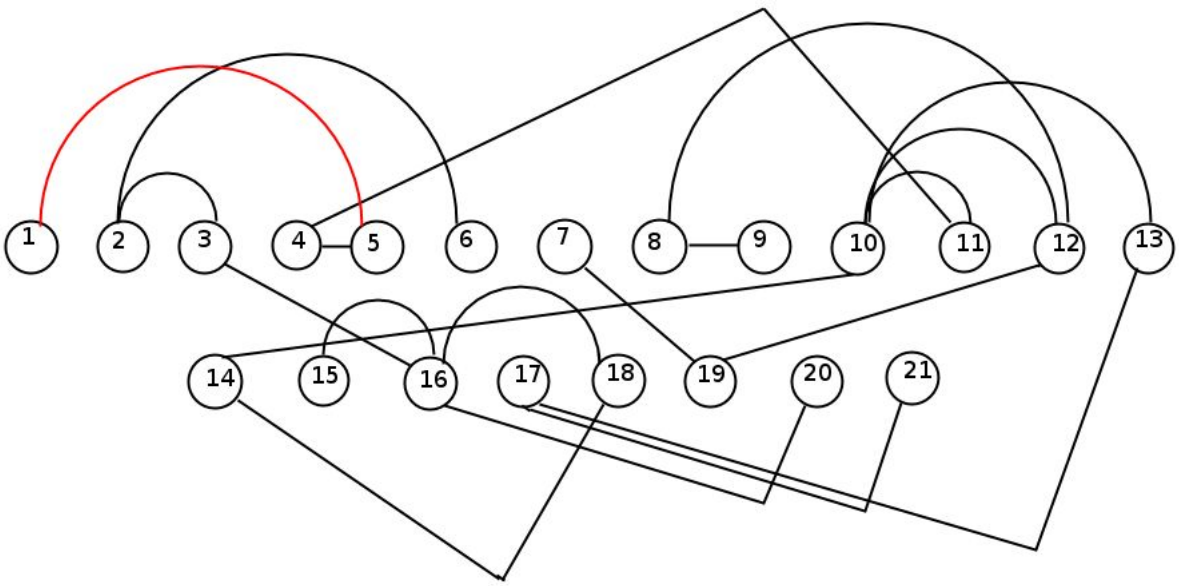

Figure 4.23: For the triplet $1\ 5|6$ error occurs (Red Edge).

In the Correction of the Erroneous Triplet step we consider the triplet $1\ 5|6$. Here the degree of node 1 is 0 and node 5 is 1. Now node 1 has the lowest degree. Since its degree is 0 we cannot consider it. Therefore we consider node 5 in this case. So, we add an edge between node 5 and 10 according to the algorithm. Therefore the correction in the triplet forms $10\ 5|6$. Therefore the correct triplet set is,

$$10\ 11|1$$

$$10\ 12|1$$

$$10\ 13|11$$

$$10\ 14|15$$

$$12\ 19|14$$

$$13\ 17|1$$

$$14\ 18|15$$

$$15\ 16|1$$

$$16\ 18|4$$

$$16\ 20|1$$

$$17\ 21|4$$

$$2\ 6|8$$

$$2\ 3|15$$

$$3\ 16|20$$

$$4\ 11|19$$

$$4\ 5|1$$

$$8\ 12|11$$

$$7\ 19|6$$

$$8\ 9|1$$

$$10\ 5|6$$

Therefore the resulting graph after correcting all the triplets in the triplet set becomes the one in Figure 4.24.

Since there is no error in the triplet set, therefore if we apply Aho′s algorithm to construct

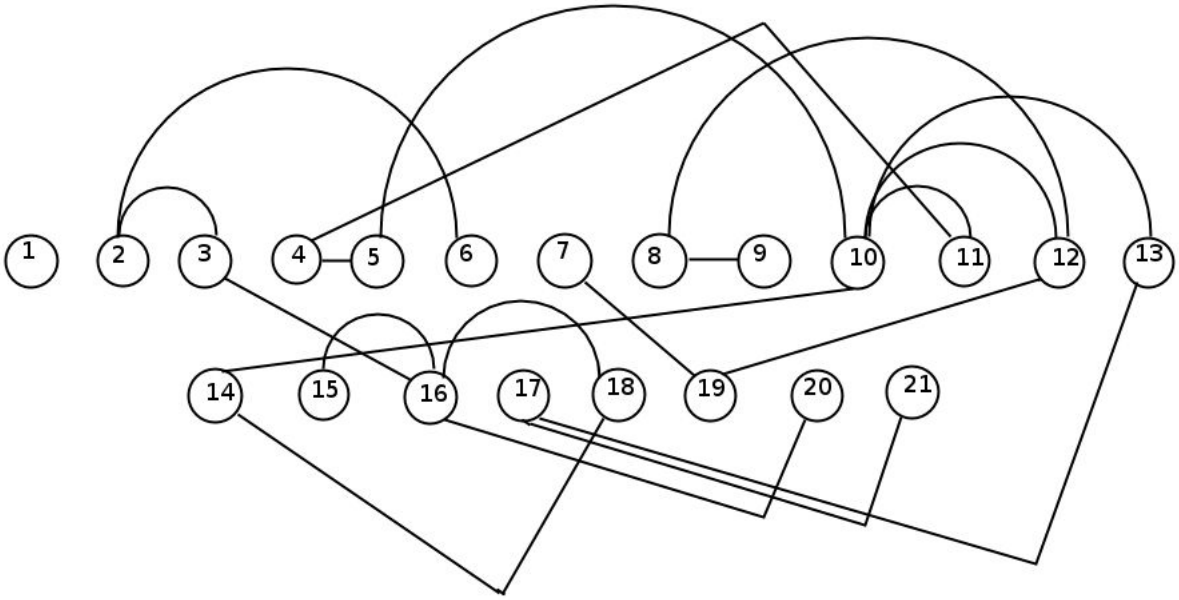

Figure 4.24: State of the graph considering all triplets in the triplet set.

the tree then it generates a tree that looks like the one given in Figure 4.25.

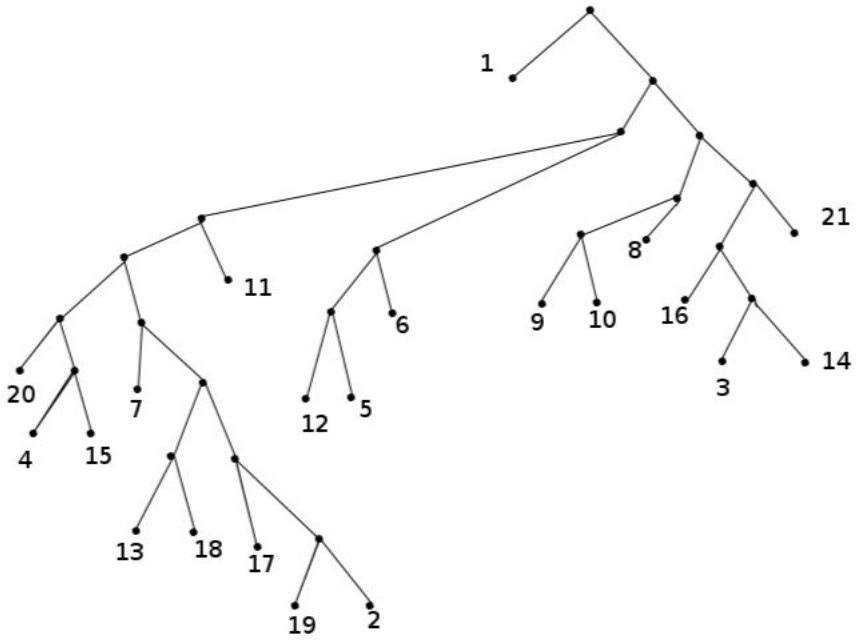

Figure 4.25: Phylogenetic Tree showing Evolutionary Relationship of Yeast.

4.7 Conclusion

In this chapter we present a method to determine the erroneous triplet from the triplet set and take corrective measures in elementary level to construct a Phylogenetic Tree that represents an evolutionary relationship among these triplets and loss of information is minimum. Here the time complexity is $O(m \times n)$ and two memory space, one of size n and another of size $m \times m$ is used where m is the number of triplet and n is the number of species. Though the process introduced here is time consuming and consume huge memory space, such an experiment is really informative in Phylogenetic Tree related studies.

CHAPTER 5
TRIPLET CONSISTENCY CHECK

5.1 Introduction

In the previous chapter, we have discussed an algorithm to construct phylogenetic tree with erroneous data. In this chapter, we are going to discuss an algorithm to check the consistency of a triplet in a phylogenetic tree in constant time. Though the algorithm requires an initialization stage which may be time and memory consuming, it can check the consistency of a triplet in O(1) time.

This process of triplet consistency check is especially useful to test an algorithm that constructs phylogenetic tree from a set of rooted triplets. In genetics, a particular evolutionary relationship may be derived experimentally and needs to be checked if it is correct with a previously established phylogenetic tree. This algorithm can be helpful in this type of situation.

Let, T be a phylogenetic tree and $P(ab|c)$ be a rooted triplet. We now have to check if the triplet P is consistent with the tree T. Here we assume that the phylogenetic tree is static.

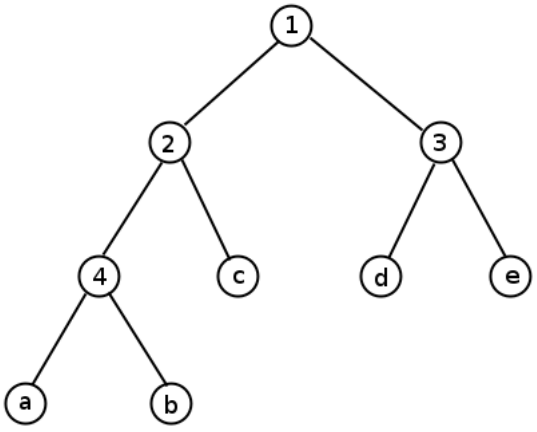

Figure 5.1: A phylogenetic tree T.

Now, triplet P will be consistent with tree T if the lowest common ancestor of a and b in tree T, is a descendant of the lowest common ancestor (LCA) of a and b in tree T.

In Figure 5.1, a phylogenetic tree T is illustrated. In Figure 5.2, a triplet $P(ab|c)$ is given which is consistent with the phylogenetic tree T. Though phylogenetic trees are leaf-labeled trees, we use a dummy labeling for the internal nodes for the phylogenetic trees and triplets

in this chapter. This temporary labeling does not change any property of the tree. This is to uniquely define each node in the tree and is essential to perform the algorithm correctly. The algorithm has two successive stages. The first stage is the initialization stage where we process the tree T to construct two matrices - Matrix A and Matrix B. Matrix A contains the pairwise LCA of all nodes of T and Matrix B keeps the information of the ancestors of each node of T. In the second stage, the consistency of P with T is checked using the matrices derived in initialization stage. The rest of the chapter is organized as follows. Section 5.2 discusses about the initialization stage of the algorithm. In Sections 5.3, we describe the method to check the consistency of a triplet. And finally Section 5.4 is the conclusion.

5.2 Initialization Stage

In this section, we describe the initialization stage of the algorithm. In this stage the phylogenetic tree T is processed to find the pairwise lowest common ancestors and the ancestor relationship with every node. In Section 5.3.1, we describe the procedure to find a 2D matrix to contain lowest common ancestors for every pair of node of tree T. In Section 5.3.2, we explore the child-parent relationship of every node of the tree.

5.2.1 Finding Pairwise Lowest Common Ancestors

In this section we will describe a method to find the lowest common ancestors for every pair of nodes in the tree T. Every node starting from the root of the tree T should be indexed from 1,2, $\cdots$, n. Suppose, we have to find the lowest common ancestors of the nodes m and n. For node m, we will make a recursive call to the parent of m until the root is reached and we will save the found nodes in an array, say pm. We will repeat the process for node n and find the array of parent pn of node n. The lowest common node of pm and pn will be the lowest common ancestor of node m and n.

Applying the same method, we will find the lowest common ancestors for all pair of nodes

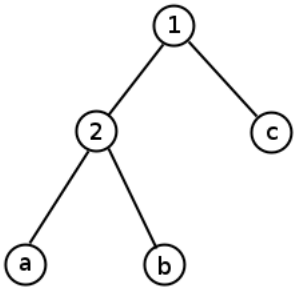

Figure 5.2: A rooted triplet P.

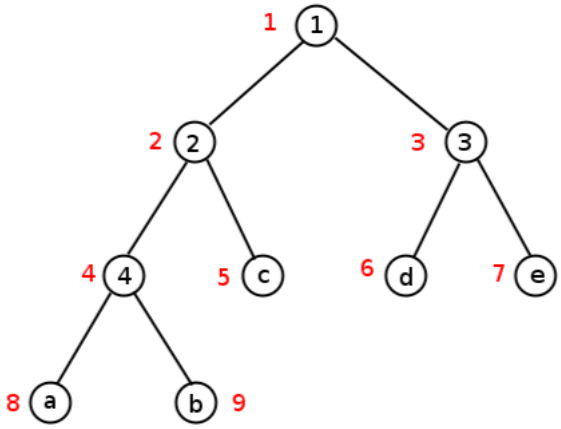

Figure 5.3: Phylogenetic tree T after indexing.

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	1	1	1	1
3	0	1	0	1	1	1	1	1	1
4	0	1	1	0	2	1	1	2	1
5	0	1	1	2	0	1	1	2	2
6	0	1	1	1	1	0	3	1	1
7	0	1	2	1	1	3	0	1	1
8	0	1	1	2	2	1	1	0	4
9	0	1	1	2	2	1	1	4	0

Figure 5.4: Pairwise lowest common ancestors of every node stored in Matrix A.

(m, n) that belongs to tree T. We will store these pairwise lowest common ancestors of every node, in a 2D matrix A. That means $A(m, n)$ will denote the lowest common ancestor of the nodes m and n.

In Figure 5.3, a phylogenetic tree T with 9 nodes are shown. We index the nodes of the tree sequentially with an increasing integer sequence of 1,2,$\cdots$,9 starting from the root node. Now for the tree T we shall derive Matrix A of pairwise lowest common ancestors by following the method described above. The matrix is shown in Figure 5.4. The first row and column denotes the indices of the nodes of the tree. All other cells of the matrix denotes the lowest common ancestors of two nodes of the tree. The value of a cell, holds the index of the lowest common ancestor of the two specific nodes. For the cells where lowest common ancestor is not possible, the value is zero. As example, The cell $A(8, 9)$ holds 4; which means the lowest common ancestor of a and b is 4.

5.2.2 Finding Ancestors

To check the consistency of a triplet with a phylogenetic tree in constant time, we should find a method to check the ancestor relation in a constant time. In this section, we will find an effective method for this.

Here we will again take a 2D array B of size $n \times n$, initialized with 0's. Every row of the matrix will be dedicated to a node of the tree T. For every node of tree T, we will traverse the ancestors of the node in a recursive fashion and change the value of the array B to 1 in the specific position. That means, if the value of $B(p, q)$ is 1, it means that q is an ancestor of p.

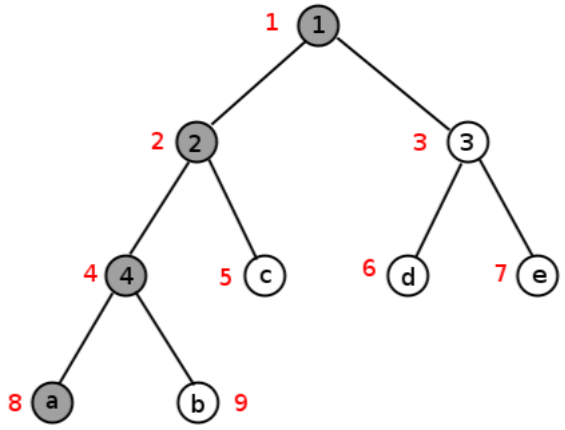

Figure 5.5: Recursive call from leaf a to find the ancestors.

8	1	2	3	4	5	6	7	8	9
	1	1	0	1	0	0	0	0	0

Figure 5.6: An array denoting ancestor relationship of 8-th node with every other node of the tree.

	1	2	3	4	5	6	7	8	9
1	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0
4	1	1	0	0	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0
6	1	0	1	0	0	0	0	0	0
7	1	0	1	0	0	0	0	0	0
8	1	1	0	1	0	0	0	0	0
9	1	1	0	1	0	0	0	0	0

Figure 5.7: Ancestor relationship of nodes in T.

In Figure 5.5, a phylogenetic tree T is shown. For the leaf node a, which is indexed as 8, we make a recursive call to its parents. This function call would return the 4th, 2nd and 1st indices of the tree, which are the ancestors of the node a. So, we would change the value of B(8,4), B(8,2), B(8,1) into 1. This change is shown in Figure 5.6. Following the similar method for every node, we shall derive the matrix B, which will denote the ancestor relation of the tree T for every pair of nodes. Matrix B for the tree T is shown in Figure 5.7, which denotes the ancestor relation of every pair of nodes of the tree.

5.3 Triplet Consistency Check

In Section 5.3, we have discussed the initialization procedure of our algorithm. Now in this section, we will describe the second stage of the algorithm to find the consistency of the triplet in constant time.

As assumed earlier, we have to check the consistency of the triplet $P(ab|c)$ with the phylogenetic tree T. We also assume that, the information of the pairwise lowest common ancestors and the ancestors of every node is available to us from the initialization step, described in Section 5.3. We will get the lowest common ancestor of the pairs a, b and a, c from $A(a, b)$ and $A(a, c)$ respectively. Let $A(a, b)$ be m and $A(a, c)$ be n. Now we have to check if m is a descendant of n or not in the tree T. We can easily check this from the array B. If the value of $B(m, n)$ is 1, then it means n is an ancestor of m in the tree T and the triplet P is consistent with the tree T. But if the value of $B(m, n)$ is 0, it means that the triplet P is not consistent with the tree T.

The cost of finding lowest common ancestor of the pairs a, b and a, c is O(1). Checking the ancestor of m and n is also done in O(1) timing. So, the whole task of triplet consistency checking is done in constant timing.

5.4 Procedure of Algorithm checkConsistency

Here we give an algorithm to check consistency of a triplet with a phylogenetic tree in constant time in algorithm 2, which we call $checkConsistency$ algorithm.

Algorithm 2 checkConsistency

Require: $n > 0 \ \wedge \ m > 0 \wedge$ Tree $T \wedge$ Triplet $xy|z$
Ensure: Triplet Consistency Check within Constant time.
  Initialize 2D array A
  Initialize 2D array B
  for each species i in T **do**
    for each species j in T **do**
      $K \leftarrow$ Ancestors of i
      $T \leftarrow$ Ancestors of j
    end for
    $A \leftarrow$ LCA of (i, j) from K and T
  end for
  for each species i in T **do**
    if $i \neq root$ **then**
      B$(i, \ root \ of \ T) \leftarrow 1$
    end if
    $r \leftarrow$ Closest ancestor of i
    while $r \neq root$ **do**
      B$(i, r) \leftarrow 1$
      $r \leftarrow$ Closest ancestor of r
    end while
  end for
  $u \leftarrow A(x, y)$
  $v \leftarrow A(x, z)$
  if B$(u, v) = 1$ **then**
    $xy|z$ Consistent with T
  else
    $xy|z$ Not consistent with T
  end if

5.5 Conclusion

In this chapter, we give a simple algorithm to check the consistency of a triplet with a static phylogenetic tree. It describes the procedure of initialization which takes n^2 space and $O(n^2)$ time. Though the initial time and memory requirement is high, this algorithm is beneficial when the number of triplets is huge. The main concern of this algorithm is constant time solution which is very important in this case.

CHAPTER 6
CONCLUSION

Phylogeny and related studies has brought revolutionary changes in the Biology and Biochemical investigation related studies in recent years. Inferring an accurate Phylogenetic tree from experimental data is a difficult task. Since constructed Phylogenetic trees are tested on real life species that presents an evolutionary relationship. So it needs to be correct and similar to the given data set. Moreover sometime it becomes necessary to check whether a triplet set is consistent with the given Phylogenetic Tree or not. In such case getting the answer in constant amount of time is important because it shows the class of the species in the triplet and ensures instant application of those in some other experiment. Therefore several studies have been performed on this topic in various aspect to make it correct. Here we summarize each chapter and its contributions. In Chapter 1, we have discussed about phylogenetic trees, their reconstruction, pairwise compatibility graph and other problems. This introductory discussion is important for proper understanding of latter chapters.

In Chapter 2, we have given some basic definition of graph theory, algorithms and data structure. These basic terminologies are used through the thesis.

In Chapter 3, we have thoroughly discussed about the Aho′s algorithm for phylogenetic tree construction. We have simulated the algorithm with an example and discussed every step for proper understanding of the process. This algorithm is the basic of the phylogenetic tree construction and proper understanding of this algorithm would give us a strong base in the research of phylogenetic trees.

In Chapter 4, we develop a new method to construct phylogenetic tree with erroneous data. The motive of this approach is to minimize the error by correcting the erroneous triplet and produce a phylogenetic tree consistent with all given triplets. We have developed this algorithm with the concept of vertex deletion, which is a new approach in this field. Moreover, we have concentrated on generating a distributed tree so that the phylogenetic tree is not dense.

In Chapter 5, we have given a simple algorithm to check the consistency of a triplet with a phylogenetic tree. This algorithm checks the consistency in constant time, which is very important to compare two phylogenetic trees. Though the initial complexity of the algorithm is high, it is essential to perform the fast checking. Where the triplet set is huge, this constant time algorithm is a very suitable one.

In this thesis we introduce two new algorithms for phylogenetic tree related problems.

Though these algorithms are not that efficient but the $AllRTC$ reduces the information loss during tree construction, do minimum corrections to erroneous triplets and construct an approximately sparse Phylogenetic Tree that shows an evolutionary relationship among a set of triplets. At the same time the $checkConsistency$ algorithm reduces the time to determine consistency of a triplet with a Phylogenetic Tree. In both the algorithms, there are some scope where some improvement is necessary that can reduce the overall time complexity of the process. Some ideas of improvement can be,

a) In $AllRTC$, the current time complexity is too high. Since we search each node to find out which one has the minimum degree. This can be optimized by keeping a data structure with each of the node where degrees of the nodes will be stored. It can be used in an interesting way to determine the minimum degree node at any point of time.

b) In $checkConsistency$, for n number of species we need $n \times n$ sized two memories. An interesting future development can reduce this memory requirement. Moreover, time complexity currently is $O(n^2)$ for the initialization stage. This will surely reduce then.

Here we conclude our thesis with some open problems:

- Would the algorithm $AllRTC$ work for a phylogenetic tree which does not have a binary representation?
- Give an algorithm to generate all possible triplet set that are consistent with a specific phylogenetic tree.
- Is the algorithm $checkConsistency$ applicable to phylogenetic networks too? If not suggest a suitable modification to solve this problem.

REFERENCES

[1] E. Gaba, "Phylogenetic Tree of Life." http://en.wikipedia.org/wiki/Phylogenetic_tree, 2006. [Online; accessed 18-December-2014].

[2] S. Snir and S. Rao, "Using max cut to enhance rooted trees consistency," in *Transactions on Computational Biology and Bioinformatics 3*, pp. 323–333, IEEE/ACM, 2006.

[3] W. K. Sung, *Algorithms in Bioinformatics: A Practical Introduction*. Taylor and Francis, 2010.

[4] J. Byrka, S. Guillemot, and J. Jansson, "New results on optimizing rooted triplets consistency," *Discrete Applied Mathematics*, vol. 158, no. 11, pp. 1136–1147, 2010.

[5] M. R. Henzinger, V. King, and T. Warnow, "Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology," *Algorithmica*, vol. 24, no. 1, pp. 1–13, 1999.

[6] B. Chor, M. D. Hendy, and D. Penny, "Analytic solutions for three taxon ML trees with variable rates across sites.," *Discrete Applied Mathematics*, pp. 750–758, 2007.

[7] M. Steel, A. W. Dress, and S. Böcker, "Simple but fundamental limitations on supertree and consensus tree methods," *Systematic Biology*, vol. 49, pp. 363–368, 2000.

[8] M. Steel, "The complexity of reconstructing trees from qualitative charactes and subtrees," *Journal of Classification*, vol. 9, pp. 91–116, 1992.

[9] S. Kannan, E. L. Lawler, and T. Warnow, "Determining the evolutionary tree using experiments.," *Journal of Algorithms*, pp. 26–50, 1996.

[10] L. van Iersel, J. Keijsper, S. Kelk, L. Stougie, F. Hagen, and T. Boekhout, "Constructing level-2 phylogenetic networks from triplets.," *Transactions on Computational Biology and Bioinformatics*, pp. 667–681, 2009.

[11] M. N. Yanhaona, K. T. Hossain, and M. S. Rahman, "Pairwise compatibility graphs," *Journal of Applied Mathematics and Computing*, pp. 479–503, 2008.

[12] M. N. Yanhaona, M. S. Bayzid, and M. S. Rahman, "Discovering pairwise compatibility graphs.," *Discrete Mathematics, Algorithms and Applications*, pp. 607–624, 2010.

[13] D. Husmeier, R. Dybowski, and S. Roberts, *Probabilistic Modeling in Bioinformatics and Medical Informatics*. Advanced Information and Knowledge Processing, Springer, 2006.

[14] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman, "Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions," in *Journal on Computing 10*, pp. 405–421, SIAM, 1981.

[15] B. Y. Wu, "Constructing the maximum consensus tree from rooted triplets," *Journal of Combinatorial Optimization*, vol. 8, no. 1, pp. 29–39, 2004.

[16] L. Gasieniec, J. Jansson, A. Lingas, and A. Östlin, "On the complexity of constructing evolutionary trees.," *Journal of Combinatorial Optimization*, pp. 183–197, 1999.

[17] A. Dobson, "Comparing the shapes of trees," in *Combinatorial Mathematics III*, vol. 452 of *Lecture Notes in mathematics*, pp. 95–100, Springer, 1975.

[18] M. Kuhner and J. Felsenstein, "A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates," *Molecular Biology and Evolution*, vol. 11, no. 3, pp. 459–468, 1994.

[19] M. S. Bansal, J. Dong, and Fernández-Baca, "Comparing and aggregating partially resolved trees," *Theoretical Computer Science*, vol. 412, no. 48, pp. 6634–6652, 2011.

[20] D. Robinson and L. Foulds, "Comparison of phylogenetic trees," *Mathematical Biosciences*, vol. 53, no. 12, pp. 131 – 147, 1981.

[21] G. Cardona, M. Llabrés, F. Rosselló, and G. Valiente, "Comparison of galled trees," *Transactions on Computational Biology and Bioinformatics*, pp. 410–427, 2011.

[22] J. Jansson and A. Lingas, "Computing the rooted triplet distance between galled trees by counting triangles," *Journal of Discrete Algorithms*, vol. 25, pp. 66–78, 2014.

[23] M. S. Rahman, "Basic Graph Theory." Manuscript, 2014.

[24] T. Nishizeki and M. S. Rahman, *Planar Graph Drawing*. World Scientific, 2004.

[25] J. Jansson, "Maximum rooted triplets consistency." http://www.iwoca.org/problems/Jansson.pdf, 2013. [Online; accessed 18-December-2014].

[26] L. v. Iersel, J. Keijsper, S. Kelk, and L. Stougie, "LEVEL 2: A fast algorithm for constructing level-2 phylogenetic networks from dense sets of rooted triplets." http://skelk.sdf-eu.org/level2triplets.html, 2008. [Online; accessed 18-December-2014].

APPENDIX A
CODES

A.1 Implemented Code

We use this code to find out $AllRTC$

```
#include<stdio.h>
#include<iostream>
#include<stdlib.h>
#include<string.h>
#include<string>
#include<sstream>
#include<vector>
#include<algorithm>
#include<math.h>
#include<queue>
#include<deque>
#include<stack>
#include<set>
#include<list>
using namespace std;

int n,m;
vector<int>S;
vector<int>C[100];
int Parent[100];
int subscript[100];

struct listStructure{
        int k;
        int l0;
        int i;
        int l1;
```

```
28 };
29
30 struct queueStructure{
31         int u;
32         int v;
33 };
34
35 struct node{
36         int value;
37         int parent;
38         string section;
39 };
40
41 vector<node>Tree;
42
43 void treeInput()
44 {
45         //Triplet Set Input
46
47     cin>>n>>m;
48
49     S.clear();
50     memset(subscript,0,sizeof(subscript));
51
52     for(int k=0;k<m;k++)
53     {
54         C[k].clear();
55     }
56
57     cout<<"Enter␣the␣set␣elements:␣\n\n";
58
59     S.push_back(0);
60
61     for(int i=1;i<=n;i++)
62     {
63         int x;
64         cin>>x;
65         S.push_back(x);
66         Parent[x] = x;
```

```
67      }
68
69      cout<<"Enter␣the␣constraints:␣\n";
70
71      for(int i=0;i<m;i++)
72      {
73          int x,y,z,w;
74
75          cin>>x;
76          cin>>y;
77          cin>>z;
78          cin>>w;
79
80          cout<<"(␣"<<x<<"␣,␣"<<y<<"␣)␣<␣(␣"<<z<<"␣,␣"<<w<<"␣)\n";
81
82          C[i].push_back(x);
83          C[i].push_back(y);
84          C[i].push_back(z);
85          C[i].push_back(w);
86      }
87      cout<<endl;
88  }
89
90  vector<int> computePIc(vector<int>s, vector<int>c[100])
91  {
92      list<listStructure>L[100];
93      int x,y,z,w;
94      queue<queueStructure>Q;
95      set<int>track;
96
97      Q.empty();
98
99      //Initialization of List array, Set array & Queue
100
101     for(int j=0;c[j].size() == 4;j++)
102     {
103         x=c[j][0];
104         y=c[j][1];
105         z=c[j][2];
```

```
106         w=c[j][3];
107
108         s[x] = x;
109         s[y] = y;
110         s[z] = z;
111         s[w] = w;
112
113         track.insert(x);
114         track.insert(y);
115         track.insert(z);
116         track.insert(w);
117     }
118
119     for(int p=0;c[p].size() == 4;p++)
120     {
121         listStructure a;
122         a.k = c[p][2];
123         a.l0 = c[p][3];
124         a.i = c[p][0];
125         a.l1 = c[p][3];
126
127         L[s[c[p][2]]].push_back(a);
128         L[s[c[p][3]]].push_back(a);
129
130         queueStructure q;
131         q.u = c[p][0];
132         q.v = c[p][1];
133         Q.push(q);
134     }
135
136     //Identification and Correction of the Erroneous Triplet
137
138     while(!Q.empty())
139     {
140         queueStructure qu;
141         list<listStructure>smallList;
142         list<listStructure>largeList;
143         int small, large;
144
```

```
145         qu = Q.front();
146         Q.pop();
147
148         if(s[qu.u] != s[qu.v])
149         {
150             int p = qu.u;
151             int q = qu.v;
152
153             if(L[s[p]].size() < L[s[q]].size())
154             {
155                 small = s[p];
156                 large = s[q];
157             }
158             else{
159                 small = s[q];
160                 large = s[p];
161             }
162
163             list<listStructure>l;
164             list<listStructure>::iterator it;
165             l = L[small];
166
167             for(it = l.begin();it != l.end();++it)
168             {
169                 listStructure a;
170                 a = *it;
171                 if((s[p] == a.k && s[q] == a.l0)
172                                 || (s[p] == a.l0 && s[q] == a.k))
173                 {
174                     queueStructure t;
175                     t.u = a.i;
176                     t.v = a.l1;
177                     Q.push(t);
178                 }
179             }
180
181             list<listStructure>::iterator itn;
182
183             for(itn = L[small].begin();itn != L[small].end();++itn)
```

```
184             {
185                 listStructure an;
186                 an = *itn;
187                 L[large].push_back(an);
188             }
189 
190             L[small].clear();
191 
192             s[p] = large;
193             s[q] = large;
194 
195             track.insert(large);
196             track.erase(small);
197 
198             int r = 1;
199 
200             while(r<=s.size())
201             {
202                 if(s[r] == small) s[r] = large;
203                 r++;
204             }
205         }
206     }
207     return s;
208 }
209 
210 vector<int> treeBuild(vector<int>s, vector<int>c[100], char p)
211 {
212         //Tree Construction
213 
214     if(s.size() == 1) return s;
215     else{
216          vector<int>PIc;
217          set<int>track;
218          set<int>::iterator m;
219 
220          PIc.clear();
221          PIc = computePIc(s,c);
222          track.clear();
```

```
223
224         for(int j=0;j<=n;j++)
225         {
226             track.insert(PIc[j]);
227             if(subscript[j] == 0) subscript[j] = PIc[j];
228         }
229
230         if(track.size() == 1){
231            vector<int>m;
232            return m;
233         }
234         else{
235            int h=0;
236
237            for(m = track.begin(); m != track.end(); ++m)
238            {
239                int t = 0;
240                int new_p = p;
241                vector<int>cm[100];
242                vector<int>sm(n+1,0);
243                stringstream sec;
244
245                if((*m) == -1 || (*m) == 0) continue;
246                else{
247                    h++;
248
249                    for(int k=0;c[k].size() == 4;k++)
250                    {
251                        int x,y,z,w;
252
253                        x = c[k][0];
254                        y = c[k][1];
255                        z = c[k][2];
256                        w = c[k][3];
257
258                        if(PIc[x] == PIc[y] && PIc[y] == PIc[z]
259                                && PIc[z] == PIc[w] && PIc[w] == *m)
260                        {
261                            cm[t].push_back(x);
```

```
                    cm[t].push_back(y);
                    cm[t].push_back(z);
                    cm[t].push_back(w);

                    sm[x] = x;
                    sm[y] = y;
                    sm[z] = z;
                    sm[w] = w;

                    t++;
                }
            }
        }

        new_p++;
        int countt = 0;
        vector<int>Tm = treeBuild(sm,cm,new_p);

        for(int i=0;i<PIc.size();i++)
        {
            if(PIc[i] == *m)
            {
                countt++;
            }
        }

        if(Tm.size() == 0)
        {
            char tt;

            if(countt > 1)
            {
                tt = new_p + 1;
            }
            else tt = new_p;

            sec << tt;

            for(int i=0;i<PIc.size();i++)
```

```
301                         {
302                             if(PIc[i] == *m)
303                             {
304                                 node a;
305 
306                                 a.value = i;
307                                 a.parent = PIc[i];
308                                 a.section = sec.str();
309 
310                                 Tree.push_back(a);
311                             }
312                         }
313                     }
314                 }
315             }
316     }
317 }
318 
319 int main()
320 {
321 
322     char P = 96;
323     treeInput();
324     treeBuild(S,C,P);
325 
326     for(int j=0;j<Tree.size();j++)
327         cout<<Tree[j].value<<"␣"<<Tree[j].parent<<"␣"
328                 <<Tree[j].section<<subscript[Tree[j].value]<<endl;
329 
330     return 0;
331 }
```

Index