Algorithms for the Calculation and Visualisation of Phylogenetic Networks

Dissertation

der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Dipl.-Math. Tobias Klöpper

aus Bremen

Tübingen
2008

Tag der mündlichen Qualifikation:	02.05.2008
Dekan:	Prof. Dr. M. Diehl
1. Berichterstatter:	Prof. Dr. D. H. Huson
2. Berichterstatter:	Prof. Dr. D. Bryant

Erklärung

Hiermit erkläre ich, dass ich diese Schrift selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die im Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben der Quellen kenntlich gemacht sind.

Tübingen, Januar 2008 *Tobias Klöpper*

Zusammenfassung

Die Evolutionstheorie beschreibt die Entwicklung der Arten als einen stetigen Prozess der Anpassung an die Umwelt. Im klassischen Modell entwickelt sich eine Spezies durch Mutationen und Speziation weiter. Diese Ereignisse lassen sich durch phylogenetische Bäume darstellen. Wird das evolutionäre Modell jedoch um Ereignisse, wie zum Beispiel Rekombination, erweitert kann dieses nicht mehr anhand eines Baumes dargestellt werden. Phylogenetische Netzwerke sind eine Klasse von Graphen, welche entwickelt wurden, um diese zusätzlichen Ereignisse zu modellieren. Diese Netzwerke können in zwei Klassen unterteilt werden: die *expliziten* Netzwerke, welche die evolutionären Abläufe direkt modellieren und die *impliziten* Netzwerke, welche nicht direkt die evolutionären Abläufe modellieren, sondern die von den Abläufen erzeugten Signale.

Gegenstand dieser Arbeit ist die Entwicklung von neuen Algorithmen zur Rekonstruktion und Visualisierung von *expliziten* phylogenetischen Netzwerken. Dabei wird eine Lösung bei der Rekonstruktion als optimal angesehen, wenn sie den evolutionären Aufwand minimiert. Ein Problem, welches bei der Rekonstruktion dieser Netzwerke auftritt, ist die hohe Anzahl an möglichen Graphen, von welchen gewählt werden kann, um eine optimale Lösung zu erhalten, und daß es keine Möglichkeit gibt, diese Wahl effizient zu gestalten. Ein Weg die Anzahl von Graphen, welche betrachtet werden müssen dennoch zu reduzieren, ist die Zerlegung des Problems in kleine voneinander unabhängige Einheiten.

Durch eine intelligente Reduzierung der betrachteten Graphenklasse, konnte gezeigt werden, daß eine eindeutige Zerlegung des Problems in unabhängige Teilprobleme möglich ist. Desweiteren wurde ein effizienter Algorithmus entwickelt, welcher die Berechnung der optimalen Lösungen für die unabhängigen Teilprobleme ermöglicht.

Außerdem wurde ein Algorithmus entwickelt, welcher es erlaubt, explizite phylogenetische Netzwerke zu zeichnen. Die Entwicklung des Algorithmus wurde so gestaltet, daß vorhandene Algorithmen zur Visualisierung von phylogenetischen Bäumen erweitert werden. Hierzu wird eine Modifizierung des phylogenetischen Netzwerks durchgeführt und eine Optimierung zur Minimierung sich überschneidener Kanten entwickelt.

Im weiteren Teil der Arbeit werden zwei Softwareprojekte vorgestellt, welche zum Ziel haben, die Erreichbarkeit von neuen Methoden und die Aussagekraft von großen phylogenetischen Graphen zu verbessern. In dem ersten Projekt wurde ein Managementsystem für Plugins entwickelt, welches erlaubt, eine installierte Software nachträglich mit neuen Methoden (Plugins)

zu erweitern. In dem zweiten Projekt wurde eine Software entwickelt, welche phylogenetische Graphen mit zusätzlichen Informationen zu annotieren erlaubt.

Abstract

Evolution describes the development of species as a steady adaption to the environment. In the classical model, a species develops by mutation and speciation events, which can be modelled using phylogenetic trees. However, if the evolutionary model is generalized by integrating events such as recombination, a tree can no longer describe the process. Phylogenetic networks are a class of graphs that have been developed to describe these more complex processes. These networks can be divided into two groups: those networks that model evolutionary events *explicitly* and those networks that do so *implicitly*.

In this thesis, we focus on the development of new algorithms for the reconstruction and visualization of explicit phylogenetic networks. A reconstruction is called optimal if it minimizes the evolutionary costs. The large number of possible graphs from which one can choose an optimal solution presents one of the hardest problems in the reconstruction, since no possibility exists to choose the right one efficiently. One possible way to reduce the number of graphs one has to choose from, is to break down the problem into smaller independent sub-problems.

By carefully reducing the class of phylogenetic networks under consideration, we were able to show that indeed the main problem can be broken up into smaller parts. Furthermore, we developed an efficient algorithm for calculating all optimal solutions for each independent sub-problem.

In addition, we developed an algorithm that is capable of drawing explicit phylogenetic networks. The algorithm was designed in such a way that the algorithms available for drawing phylogenetic trees can be generalized to draw explicit phylogenetic networks. To do so, we modified the explicit phylogenetic network and extended the tree drawing algorithm by adding an optimization step, which minimizes the number of crossing edges.

Furthermore, we present two software projects which aim at extending the availability of new phylogenetic methods within SplitsTree and increasing the useablility of large phylogenetic graphs. In the first project, we implemented a management system for plugins, which allows the application to dynamically integrate new methods that are stored on a database in the Internet. In the second project, we developed software that allows for the annotation of phylogenetic graphs within SplitsTree.

Acknowledgments

I am tremendously thankful to Daniel Huson, who has given me the chance to write this PhD on the very interesting field of phylogenetic networks. He not only made this thesis possible, but also provided a perfect working environment in every respect. At all times, Daniel encouraged and supported my scientific career. Whenever I got lost in my research, he listened to my problems and provided clues to bring me closer to the solution. Furthermore, he supported my interest in scientific projects not directly related to my thesis and for this, I am extremely thankful. I am also indebted to David Bryant for the co-supervision of this thesis.

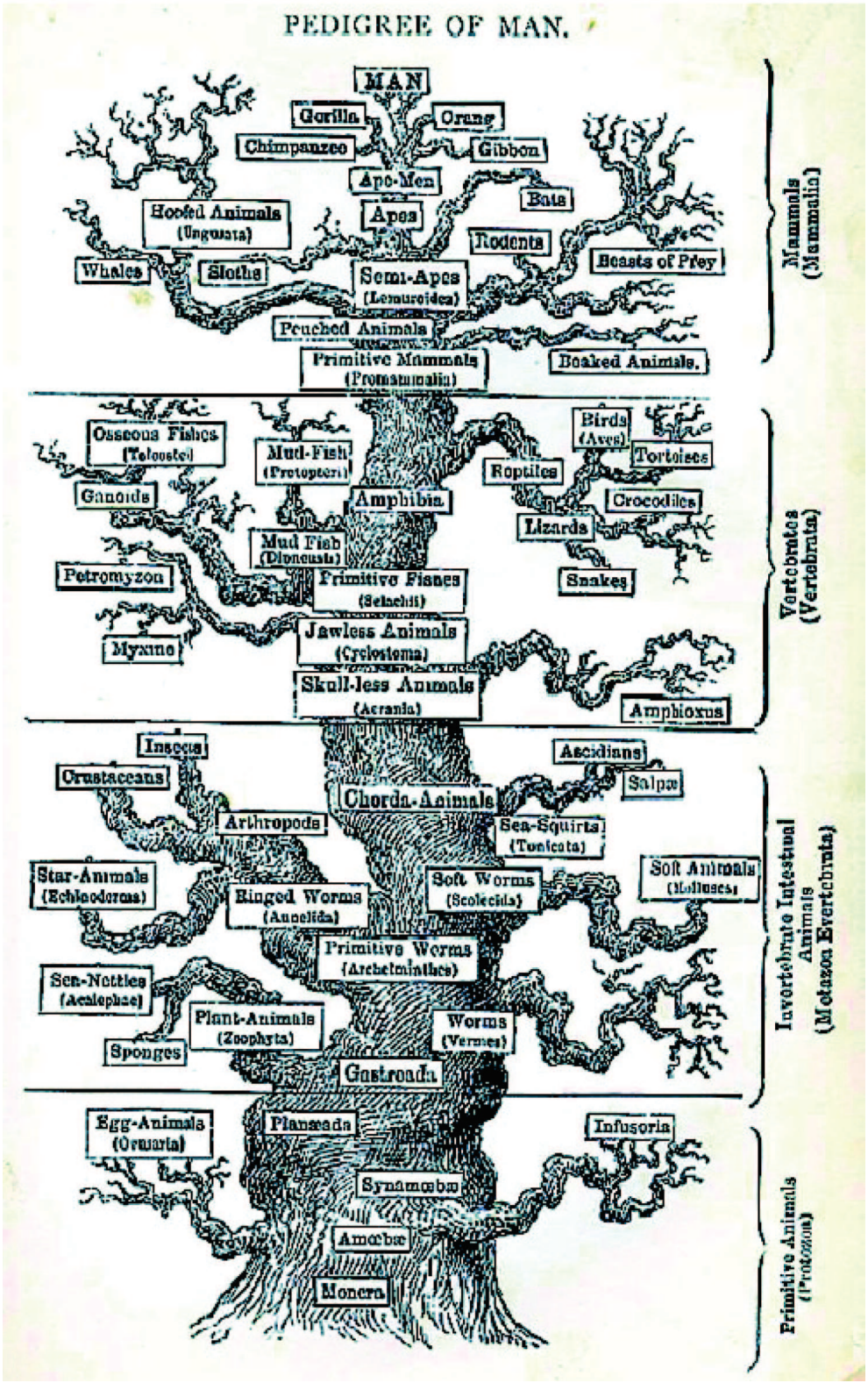
In addition to my two supervisors, I am very thankful to Kay Nieselt, who was the first person to introduce me to the research field of phylogenetics.

I would also like to thank all former and current staff of the department of Algorithms in Bioinformatics - namely Marine Gaudefroy-Bergmann, Jan Schulze, Regula Rupp, Alexander Auch, Christan Rausch, Juliane Damaris Klein, Suparna Mitra, Olaf Delgado Friedrichs and Tobias Dezulian for a fun and productive time together. Thereby, I am especially indebted to Daniel Richter for sharing an office with me, and his good sense of humor. Besides the many persons I have met at the Institute, I would like to thank C. Nickias Kienle, Georg Zeller and Andreas Schmidt for long talks about football, even longer nights playing poker and all the fruitful scientific discussions. Furthermore, I thank Dirk Fasshauer, Anand Radhakrishnan and Stefan Pabst for their interest in my scientific work.

I am especially attached to my family, who have supported me my whole life and who have given me the possibility and strength to walk this road. My whole love is devoted to my wife Katrin, who has supported me all this years and who fills my life with joy.

Finally, I would like to thank my good luck for this great life!

In accordance with the standard scientific protocol, I will use the personal pronoun "we" to indicate the reader and the writer, or (as explained in Appendix B) my scientific collaborators and myself.



Tree of Life by Ernst Häckel (1874) [Hae74]

Contents

Chapter 1

Introduction

The ability of the living to adapt to their environment is the focus of many scientific studies. A classic example of an animal adapting to its environment is the peppered moth in England. In general, peppered moths will spend the day resting on trees or lichen. The original predominant species in England had a light coloration similar to the coloring of the trees and lichen they where resting on, which effectively camouflaged them against their predators (mainly birds). During the Industrial Revolution, many species of lichen died out and the trees were blackened by soot. Consequently, the light colored moths became easy targets for their predators. At the same time, a second type of peppered moth with dark coloration became the new predominant type in England, since their coloration effectively camouflaged them against their predators [Ket55; Ket56; Maj02].

The ability to mutate enables organisms to specifically adapt to their environment and ensures their survival. The optimization process triggered by the adaption is generally called *evolution*. The fundamental properties of the evolutionary process were first recognized and described by C. Darwin and W. Whewell [Wal58; Dar59]. In his famous book, *The Origin of Species*, Darwin introduced the basic principles of his theory of evolution, the "Struggle for Existence" and "Natural Selection", which are still valid today. He deduced these principles from his studies about the variation within species. One consequence of these principles is the emergence of new subspecies by speciation and selection. Furthermore, new subspecies could develop far apart from the original species and thus be considered an independent new species. Darwin also recognized that this evolutionary process could be visualized using a directed tree structure, where nodes in the tree correspond to speciation events and edges to adaptation (mutation) events. The first evolutionary tree drawn by Darwin is shown in Figure 1.1. Consequently following the idea of evolution he deduced the existence of a *Tree of Life* "*which fills with its dead and broken branches the crust of the earth, and covers the surface with its ever-branching and beautiful ramifications*" [Dar59].

The science of reconstructing the evolutionary history of a group of species is called *phylogeny*. Phylogeny uses either morphological characters or genetic information to model such an evolutionary history. As there is a vast amount of genetic sequence information available today, most algorithms use these to calculate a most likely evolutionary history.

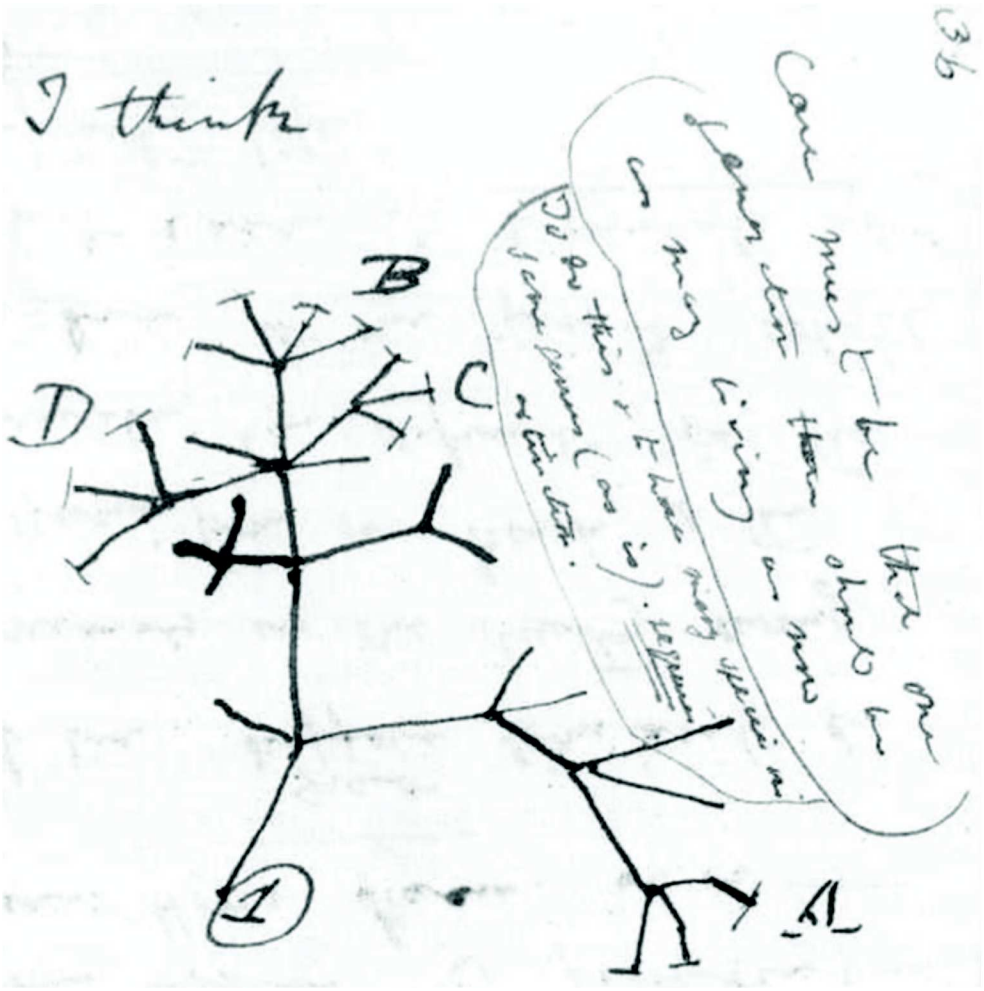


Figure 1.1: **The First Phylogenetic Tree** Darwin's first sketch of an evolutionary tree from his *First notebook on Transmutation of Species* (1837). [Dar37]

The fundamental idea of Darwin, that of reconstructing a universal *Tree of Life*, has been very actively researched ever since. For example, Ernst Häckel became famous for his work on this topic, being the first to place apes closest to humans, and also for his artistic drawing of phylogenetic trees. An example of his drawing is shown at the beginning of this thesis. With the methods available today, the rapid sequencing of large amount of genetic information and the rapid assembly of whole genomes, the reconstruction of a Tree of Life seems to become an achievable task. An important question that can be deduced from the information sampled so far is the question about the existence of such an universal tree. One argument against a Tree of Life can be derived from bacteria. Some bacteria can not only pass their genetic information to their offsprings, but also exchange genetic information via transformation (exogenous DNA is picked up from the environment), transduction (integration of foreign DNA through a bacteriophage) or conjugation (transfer of DNA via direct cell contact). Furthermore, the *endosymbiotic theory* gives strong evidence that the origin of the mitochondria and the plastids of eukaryotic cells lie in the prokaryotic kingdom and that these organelles have been

taken inside the cells by endosymbiosis [Mer05; Sch83; SDJ04; MGL99]. The mechanisms of some bacteria described above and the endosymbiotic events cannot be modeled using a tree-like graph structure. In contrast, these events point to a web-like evolutionary history and consequently, one speaks about a *Web of Life* rather than a Tree of Life. An intensively lead discourse about the validity of both theories exists; for example, see [DB07].

As mentioned above, phylogenetic relationships have traditionally been modeled using trees. The internal vertices of a phylogenetic tree correspond to speciation events and the edges correspond to mutation events. In this thesis we will concentrate on a natural generalization of these graphs called phylogenetic networks, which can model not only mutation and speciation events but also recombination events. The work related to this thesis started in 2004. At this point in time, the term *Phylogenetic Network* was used to describe many different concepts. For example, in [GB05], Gusfield and Bansal define a phylogenetic network as containing only nodes of indegree zero, one or two. Splits networks [BD92; HB06] are also graphs that model phylogenetic information in a non-treelike way, but their internal nodes do not necessarily fulfill the requirements of a phylogenetic network as given above.

The ambivalent use of technical terms leads to an unclear definition of phylogenetic networks. Maybe this is one of the reasons why methods of phylogenetic network reconstruction are not well established. One fundamental finding that we published very early was an overview of the different classes of phylogenetic networks [HKLS05]. In fact, it is useful to distinguish between two main classes: *implicit* phylogenetic networks that provide tools to visualize and analyze phylogenetic signals that are incompatible with a tree model, such as split networks [BD92; HB06], and *explicit* phylogenetic networks that provide explicit scenarios of reticulate evolution, such as hybridization networks [SZ00; LR04; NWL04; HKLS05; BS06], HGT networks [HLT04] and recombination networks [Hud83; SH05; LSH05; HK07; DGS07]. An overview of different phylogenetic networks can be seen in Figure 1.2.

The overview of different types of phylogenetic networks in Figure 1.2, reflects the structural affinity between split networks and reticulate networks first introduced in [HKLS05]. Determining more general assertions about the mathematical relationship of these two types of networks is an interesting challenge. Because of the advantages both types of networks have on their own, a deeper and more complete understanding of the relationship between these two types of phylogenetic networks can prove beneficial and may help these methods to become more evolved in standard phylogenetic analysis.

Structural Overview of this Thesis

In the following chapter, we introduce the foundations necessary for the scientific part of this thesis. In Chapter 3, we introduce a new theorem that proves a one-to-one correspondence between structural properties of split networks and reticulate networks. This correspondence gives us the necessary insights to introduce some computational methods, which are able to

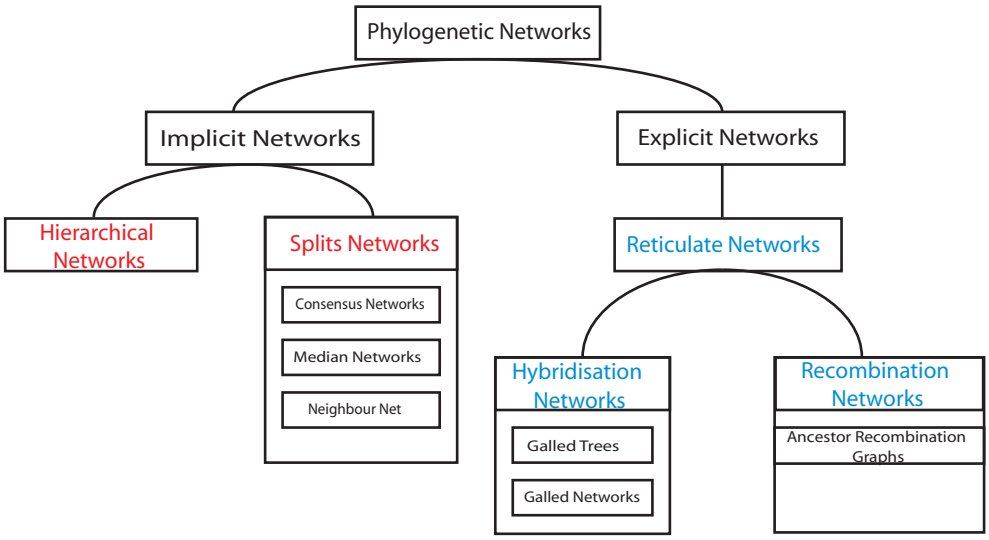


Figure 1.2: **Overview of Different Types of Phylogenetic Networks** The term *phylogenetic network* encompasses a number of different concepts, including *split networks*, and *reticulate networks*. Phylogenetic networks can be divided into two types, those that model the evolutionary history of the data *implicitly* and those that do so *explicitly*. The term "phylogenetic network" as it is used by Gusfield and Bansal in [GB05] corresponds to a reticulate network in this graph.

calculate a reticulate network from a set of splits. These methods are introduced in Chapter 4. We believe that being able to calculate an explicit interpretation from an implicit one and vice versa is an important tool, that can lead to an deeper understanding of phylogenetic networks. Furthermore, these *transformations* allow the analysis of the underlying data by methods from both research fields. For example, one could apply methods for filtering split networks [HM03; HSW06] to reduce the complexity of the data or to screen for possible false positive information. In turn, this may facilitate the calculation of a reticulate network (an example of this approach is given in 4.6).

The first methods that we introduced for the calculation of a reticulate network from a split network, visualized the results by modifying the drawing of a splits graph. This approach does well for the calculations we have published so far [HKLS05; HK05; HK07], but the results of such a modification for more complex reticulate networks will most likely not be as satisfying. Consequently, the visualization of reticulate networks became a problem that needed to be solved. The result of our research on this topic is introduced in Chapter 5. In addition to the visualization algorithm, we introduce an extension of SplitsTree 4 [HB06], that integrates reticulate networks into the program.

In general, SplitsTree 4 provides a framework for the calculation and visualization of phylogenetic trees and networks. The program contains a vast variety of phylogenetic methods such as evolutionary distance methods, maximum likelihood distances, tree building methods, split network methods

and visualization algorithms. An advantage of SplitsTree 4 is that all algorithms are integrated into its framework architecture via an interface-driven class loader, making it an easy task to incorporate new methods into the program. All methods in SplitsTree 4 implement a specific interface that allows the program to identify plugins dynamically and to integrate them into the framework at runtime. Unfortunately, SplitsTree 4 never provided an application for the user which permitted an easy access to this part of the program. To address this, we developed a system for the administration of these *plugins*. In addition, our system also supports the development and distribution of new plugins.

To visualize important information within a phylogenetic tree, one has to add an annotation. One method to do so is the annotation of important clades. Because of the increase of the genetic sequence information available to the public, the taxonomic size of phylogenetic analysis increases rapidly. The annotation of a phylogenetic tree containing hundreds or even thousands of taxa can be very time-consuming. A tight integration of an annotation into a phylogenetic software can help reduce the time needed. We have extended the graph library used by SplitsTree 4 to integrate the basic methods needed for such an annotation. The extension is presented in Chapter 7. Furthermore, we introduce an application of the extension that provides an automatic annotation of phylogenetic trees.

Chapter 2

Background

In this chapter, we introduce the basic notations and mathematical concepts for this thesis. The first part deals mainly with split networks and their applications. The second part has its focus on reticulate networks and introduces their mathematical foundations, and also gives a basical introduction into their history.

2.1 Splits and Clusters

A *tree* $T = (V, E)$ is a connected acyclic graph with a vertex set V and an edge set E. A vertex of degree one is called a *leaf* of T and the set of all leaves is called the *leaf set* of T.

A *rooted tree* $T = (V, E, \varrho)$ is a tree (V, E) that has exactly one distinguished vertex called the *root*, denoted by ϱ. A rooted tree T has a natural ordering where $v \leq v'$, if v lies on the path from the root to v'. If $v \leq v'$, we say that v is an *ancestor* of v' and v' is a *descendant* of v. For any edge e, we denote $\alpha(e)$ to be the *ancestor* and $\beta(e)$ to be the *descendant* of e.

An $X-$*tree* is an ordered pair (T, λ), where T is a tree and $\lambda : X \to V$ is a map from the set of taxa X to the set of nodes V with the property that, for each v in V of degree at most two, $v \in \lambda(X)$. Roughly speaking, an $X-$tree is a tree that is labeled on certain vertices. One special case that we are particularly interested in is the one where all leaves of an $X-$tree are labeled by an unique element of X. A *rooted phylogenetic* $X-$*tree* is a pair (T, λ), where $T = (V, E, \varrho)$ is a rooted tree and $\lambda : X \to V$ is a bijection from the taxon set X to the leaf set of T.

Any bipartition of X is called an $X-$*split* and if A and B are the two subsets of the bipartition, we denote the $X-$split as $\frac{A}{B}$. If the underlying set of taxa is obvious, we omit the X and call $\frac{A}{B}$ a *split* (we do not distinguish between the equivalent splits $\frac{A}{B}$ and $\frac{B}{A}$).

Two splits $s = \frac{A}{B}, s' = \frac{A'}{B'}$ are called *compatible* if and only if at least one of the intersections $A \cap A'$, $A \cap B'$, $B \cap A'$ or $B \cap B'$ is empty; otherwise, they are called *incompatible*. Furthermore, a set of splits Σ is called *compatible* if and only if all pairs of splits in Σ are compatible.

Let $T = (V, E)$ be an $X-$tree. For an edge e in T, the *split* s of e is defined by the bipartition of X that is formed by the two connected

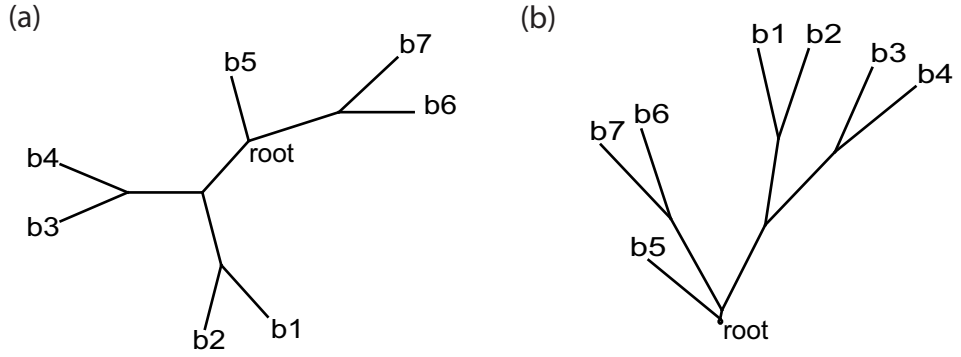


Figure 2.1: **Encoding Splits and Clusters** The $X-$tree shown in (a) contains the $X-$splits: $\{\frac{b1}{b2,b3,b4,b5,b6,b7,\varrho}, \frac{b1,b2}{b3,b4,b5,b6,b7,\varrho}, \frac{b1,b2,b3,b4}{b5,b6,b7,\varrho}, \frac{b1,b2,b3,b4,b5,\varrho}{b6,b7}, \frac{b1,b2,b5,b6,b7,\varrho}{b3,b4}, \frac{b1,b3,b4,b5,b6,b7,\varrho}{b2}, \frac{b1,b2,b4,b5,b6,b7,\varrho}{b3}, \frac{b1,b2,b3,b5,b6,b7,\varrho}{b4}, \frac{b1,b2,b3,b4,b6,b7,\varrho}{b5}, \frac{b1,b2,b3,b4,b5,b7,\varrho}{b6}, \frac{b1,b2,b3,b4,b5,b6,\varrho}{b7}\}$. Rooting the $X-$tree at vertex ϱ results in the tree shown in (b). The cluster encoding of this rooted tree is: $\{\ \{b1,b2,b3,b4,b6,b7\}, \{b1,b2,b3,b4\}, \{b1,b2\}, \{b3,b4\}, \{b6,b7\}, \{b1\}, \{b2\}, \{b3\}, \{b4\}, \{b5\}, \{b6\}, \{b7\}\}$.

components of T obtained by removing e from the graph. The set of all splits Σ obtainable from T in this way is called the *splits encoding* $\Sigma(T)$ of T. An X-tree T' is called a *refinement* of an X-tree T if $\Sigma(T) \subseteq \Sigma(T')$, that is, if $T' = T$ or if T can be obtained by contracting some edges of T'.

The conceptual importance of splits in phylogeny is highlighted by the *Buneman Theorem* [Bun71]:

Theorem 2.1 *Let Σ be a collection of $X-$splits. Then, an $X-$tree T exists such that $\Sigma = \Sigma(T)$ if and only if Σ is compatible. Moreover, if such an $X-$tree exists, then, up to isomorphism, T is unique.*

If T is a rooted $X-$tree, we can define the *cluster* c of e to be the set of descendant taxons i.e. the labeled vertices of the subtree below $\beta(e)$. The set of all clusters $\mathcal{H}$ obtainable from T in this way is called the *cluster encoding* $\mathcal{H}(T)$ of T.

For any $X-$tree (T, λ) one can construct a rooted $(X\backslash x)$-tree (T^x, λ^x) by choosing T^x to be the rooted tree obtained from T by selecting $\lambda(x)$ as root and $\lambda^x = \lambda(X\backslash x)$. Following the argument in [SS03], this construction gives a bijection between the set of $X-$trees and the set of rooted $(X\backslash x)-$trees. For the split encoding $\Sigma(T)$ of an $X-$tree T and the cluster encoding $\mathcal{H}(T^x)$ of the constructed rooted $(X\backslash x)-$tree T^x, one can construct a bijection between $\Sigma(T)$ and $\mathcal{H}(T^x)$ with the essential property that each split $s = \frac{A}{B}$ in $\Sigma(T)$ is mapped onto the cluster c in $\mathcal{H}(T^x)$ for which $A = c$. The inverse function φ^{-1} mapes each cluster c in $\mathcal{H}(T^x)$ onto the split $s = c|\{X\backslash c\}$.

In an analogy to splits, two clusters, c and c' are called *compatible* if and only if at least one of the three sets $c \cap c'$, $c\backslash(c \cap c')$ and $c'\backslash(c \cap c')$ is empty; otherwise, the two clusters are called *incompatible*. A set of clusters $\mathcal{H}$ is *compatible* if and only if all pairs of clusters in $\mathcal{H}$ are compatible. A compatible set of clusters is also called a *hierarchy* on X. Because of the equivalence between a set of splits and a set of clusters for rooted trees, the following equivalence theorem also holds:

Theorem 2.2 *Let $\mathcal{H}$ be a collection of non-empty subsets of X. Then, a rooted $X-$tree T exists such that $\mathcal{H} = \mathcal{H}(T)$ if and only if $\mathcal{H}$ is a hierarchy on X. Moreover, if such an $X-$tree exists, then, up to isomorphism, T is unique.*

For a more detailed introduction to splits and clusters, see [SS03].

2.2 Split Networks

In this section, we introduce the basic concepts of *split networks*, which are used to visualize arbitrary sets of splits. Let $G = (V, E)$ be a finite connected graph and let C denote a set of *colors*. An *edge coloring* is a map $\varsigma : E \to C$ and a path p in G is called *properly colored* if each edge in p is colored differently. A coloring ς is called a *isometric coloring*, if all shortest paths between two vertices in G are properly colored and use the same set of colors. A pair (G, ς) consisting of a bipartite, connected, finite, simple graph $G = (V, E)$ and an isometric coloring $\varsigma : E \to C$ is called a *splits graph*. The most important property of a splits graph is the following:

Theorem 2.3 *Let $(G = (V, E), \varsigma : E \to C)$ be a splits graph. For any $c \in C$, it holds that the graph G_c obtained by deleting all edges of color c in G consists of precisely two separate connected components, denoted by $G_c^0 = (V_c^0, E_c^0)$ and $G_c^1 = (V_c^1, E_c^1)$*

Let Σ denote a set of $X-$splits. A splits graph $(G = (V, E), \varsigma : E \to C := \Sigma)$, together with a vertex labeling $\nu : X \to V$, is said to represent Σ if for every split $\frac{A}{B}$ in Σ, the deletion of all edges colored by $\frac{A}{B}$ produces a graph consisting of precisely two components, where one component contains all vertices labeled by A and the other contains all vertices labeled by B. A splits graph representing a set of splits Σ is called a *split network* $SN(\Sigma)$. If the splits graph of a set of splits Σ is is rooted, one can replace Σ by the corresponding set of clusters $\mathcal{H} := \varphi(\Sigma)$, thus obtaining a split network for a set of clusters $SN(\mathcal{H})$. More details about splits graphs can be found in [DH04].

A *netted component* $Z(\mathcal{H})$ of a split network $SN(\mathcal{H})$ is a maximal set of vertices and edges such that any two vertices $v, w \in Z(\mathcal{H})$ are connected by two different vertex-disjoint paths in $Z(\mathcal{H})$ (in graph-theoretic terminology, a $2-$connected component). Suppose we have a set of clusters $\mathcal{H}$. The *incompatibility graph* $IG(\mathcal{H}) = (V, E)$ of $\mathcal{H}$ has a vertex set $V = \mathcal{H}$ and an edge set E, in which two vertices are connected if and only if they are incompatible.

It is a simple observation that any two clusters $c, c' \in \mathcal{H}$ are incompatible if and only if the edges representing c and c' are contained in the same netted component; see [BD92]. Consequently, for a set of clusters $\mathcal{H}$ a one-to-one correspondence exists between the non-trivial connected components of the incompatibility graph $IG(\mathcal{H})$ and the $2-$connected components of $Z(\mathcal{H})$.

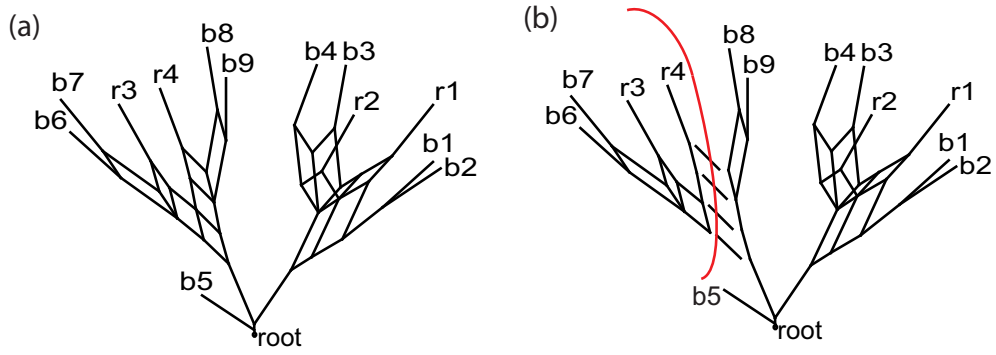


Figure 2.2: **An Example of a Split Network** The graph in (a) is a split network. Figure (b) shows the removal of a bundle of four parallel edges, producing two connected components, which represent the split $\frac{b1,b2,b3,b4,b5,b8,b9,r1,r2,\varrho}{b6,b7,r3,r4}$.

2.3 Consensus and Z-Super Networks

In many situations, the joint analysis of a set of phylogenetic trees is an important step. The problem on how to join a set of phylogenetic trees can be solved in different ways. The solution depends on the set of phylogenetic trees $\mathcal{T}$ given. In the case where all trees in $\mathcal{T}$ are defined on the same set of labels, the *Consensus method* [HM03] can be used. The method makes use of the Buneman Theorem and counts the number of trees in which a given split occurs. That is, let $\mathcal{T} = \{T_1, \ldots, T_k\}$ be a set of trees and denote $\Sigma(\mathcal{T}) = \cup_{T \in \mathcal{T}} \Sigma(T)$ as the set of all splits that occurs in $\mathcal{T}$. The $p-$*consensus* ($p \in [0, 1)$) of $\mathcal{T}$ is defined as the set of splits:

$$\Sigma(p) = \{s \in \Sigma(\mathcal{T}) : |\{T \in \mathcal{T} : s \in \Sigma(T)\}| > pk\}.$$

If $p = 0.5$, the consensus is called the *majority consensus* and it is easy to show that for any majority consensus, the resulting set of splits is compatible. Finally, the $p-$*Consensus network* $C^p(\mathcal{T})$ for the set of trees $\mathcal{T}$ is a split network representing all splits in $\Sigma(p)$.

If the set of trees $\mathcal{T}$ is partial (i.e. not all trees contain the same set of labels), the solution is not as straightforward. One method that can be used to join sets of partial trees is the $Z-$*closure* [HDKS04]. The basic idea of the $Z-$closure is to expand the set of partial splits $\Sigma(\mathcal{T})$ until a certain convergence is achieved. The rule that is used to expand these partial splits is called the $Z-$*rule* and was first introduced by C. Meacham in the context of inferring phylogenies from multi-state characters [DS04; Mea83; SS01]:

> *For any two splits* $S_1 = \frac{A_1}{B_1} \in \Sigma$ *and* $S_2 = \frac{A_2}{B_2} \in \Sigma$*:*
> *if* $A_1 \cap A_2 \neq \emptyset$, $A_2 \cap B_1 \neq \emptyset$, $B_1 \cap B_2 \neq \emptyset$ *and* $A_1 \cap B_2 = \emptyset$, *then*
> *replace* S_1 *and* S_2 *by* $S_1' = \frac{A_1}{B_1 \cup B_2}$ *and* $S_2' = \frac{A_1 \cup A_2}{B_2}$.

Repeatedly applying this rule to all splits originally contained in, or derived from $\Sigma(\mathcal{T})$ until the rule yields no new splits, produces an *order dependent Z-closure* $\bar{\Sigma}'$. The *complete Z-closure* $\bar{\Sigma}$ is the set of all splits that occurs in at least one order dependent Z-closure. Finally the *Z-closure network* $Z(\mathcal{T})$ for the set of trees $\mathcal{T}$ is a split network representing all *full* $X-$splits in $\bar{\Sigma}^*$.

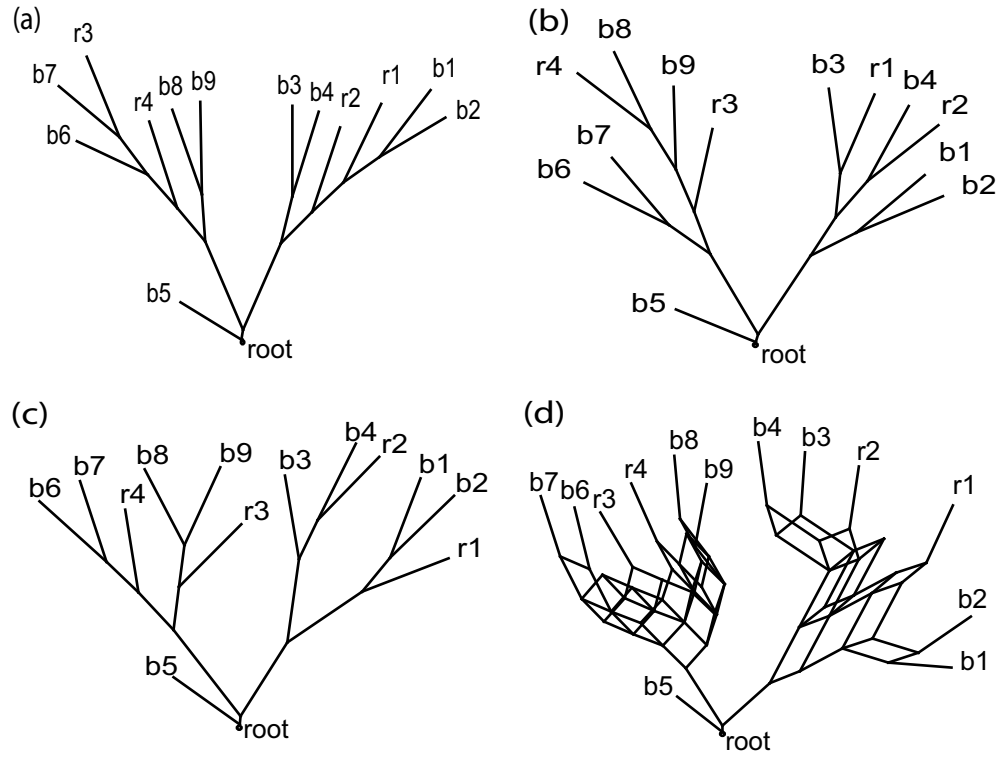


Figure 2.3: **Consensus Network of Three Trees** Figures (*a*), (*b*) and (*c*) show different trees, defined on a common set of taxa. The figure (*d*) shows the consensus network representing all splits in the trees.

2.4 Reticulate Networks

Reticulate networks belong to the family of explicit phylogenetic networks. They comprise, for example, hybridization networks and ancestral recombination graphs. In the following, we will introduce the basic mathematical concepts of reticulate networks and then give an overview on recent progress in this research field.

Mathematical Background

Definition 2.1 *Let X be a set of taxa. A* rooted reticulate network $N = N(V, E, \lambda)$ on X *is a connected, directed acyclic graph with vertex set V, edge set E and vertex labeling $\lambda : X \to V$, such that:*

1. *precisely one distinguished vertex ϱ exists, called the* root*,*

2. *every vertex $n \in V$ is either a* tree vertex*, $v \in V_T$ that has exactly one ancestor, or a* reticulation vertex *(also called a* reticulation*) $r \in V_R$ that has exactly two ancestors,*

3. *every edge is either a* tree edge *leading to a vertex of indegree one, or a* reticulation edge *leading to a vertex of indegree two, and*

4. *the set of* leaves L *(vertices with no descendants) consists only of tree vertices and is labeled by the set of taxa* X*, that is,* λ *maps* X *bijectively onto* L*.*

A reticulate network N' is called a *refinement* of a reticulate network N, if N can be obtained from N' by contracting tree edges. Let N be a reticulate network on X with k reticulation vertices $r_1, \ldots, r_k$. For any such reticulation r_i, let $p(r_i)$ and $q(r_i)$ denote the two associated reticulation edges. Furthermore, we call the vertices $\alpha(p(r_i))$ and $\alpha(q(r_i))$ the *connecting vertices* (or *connectors*) of r_i. We can obtain an X-tree from N by choosing and removing one reticulation edge $p(r_i)$ or $q(r_i)$ for each reticulation r_i. The set of trees $\mathcal{T}(N)$ obtainable in this way is called the *induced* set of trees or trees that *can be sampled* from N. Clearly, the number of different trees that can be sampled from a network N with k reticulations is $|\mathcal{T}(N)| \leq 2^k$.

It follows from these definitions that each reticulation vertex (or *reticulation*, for short) $r \in V_R$ is contained in one or more cycles (in the undirected graph corresponding to G) of the form $C = (r, p(r), w_1, e_1, \ldots, e_{k-1}, w_k, q(r), r)$, with $w_i \in V$ and $e_i \in E \setminus \{p(r), q(r)\}$ for all i (Note that additionally, r can be contained in one or more cycles that do not contain $p(r)$ and $q(r)$). Any such cycle C is called a *reticulation cycle*. Note that a reticulation r possesses at most one reticulation cycle C of which the backbone (i.e. all edges in C except $p(r)$ and $q(r)$) B contains tree edges only; in this case, we call the cycle a *tree cycle* and denote it by $C(r)$.

We say that two reticulations $r, r' \in V_R$ are *dependent* if a cycle that contains both r and r' exists. If no such cycle exists, then r and r' are called independent. Note that in this case, none of the cycles that contain r share any edges with a cycle that contains r'. More precisely, we say that r and r' are *directly dependent* if both r and r' possess a tree cycle and their tree cycles share at least one edge. A reticulation network is called *weakly dependent* if for any dependent pair of reticulation r_1, r_n, a chain of reticulations $r_1, \ldots, r_n$ exists such that any pair r_i, r_{i+1} is directly dependent. We call a rooted weakly dependent reticulation network a *galled network*. It follows from the definition of a galled network that for any two tree edges e_t, e'_t contained in a common cycle C, a chain of reticulations $r_1, \ldots, r_n$ exists, such that e_t is an element of the tree cycle of r_1, e'_t is an element of the tree cycle of r_n and any pair r_i, r_{i+1} is directly dependent. A reticulation that is independent of all other reticulations is also called a *gall* [GEL03] and a reticulate network N that contains only galls is called a *galled tree*, which is a special type of galled network. We say that a reticulate network N is *overlapping*, if every set of dependent reticulations in N has the property that all corresponding reticulation cycles intersect "nicely" along a common "backbone".

Suppose that $N = (V, E, \lambda)$ is a reticulate network and denote the set of tree edges of N by E_T. For any edge e define $R(e)$ to be the set of reticulations for which e is an element of the tree cycle, and for any set of edges E, we define $R(E) := \cup_{e \in E} R(e)$. For any vertex w, let X_w be the set of all descendant taxons of w and let X_w^T be the set of all descendant taxa of w that are reachable from w through a path not containing any of the reticulations from $R(E)$ where E is the set of incoming edges of w.

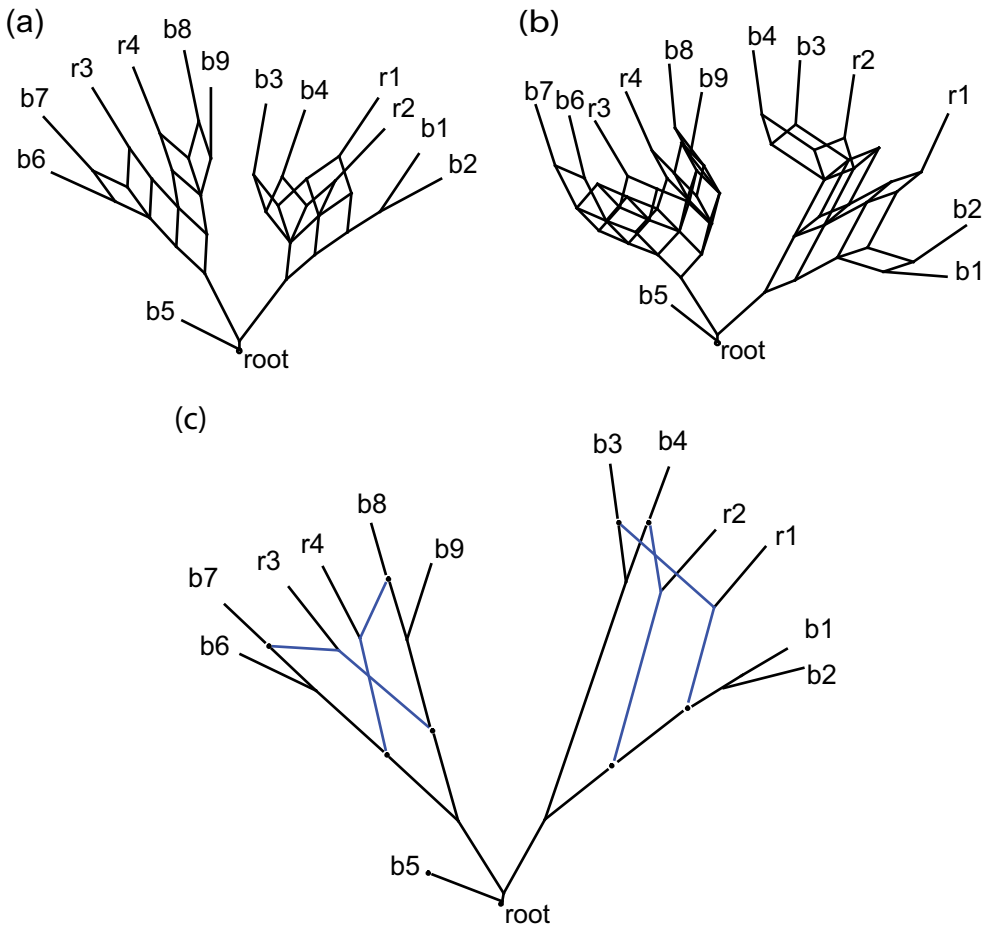


Figure 2.4: **Example of a Reticulate Network** The top row presents two consensus networks. The first one represents the consensus of all splits trees (a) and (b) of Figure 2.3. The second consensus network represents all splits of the three trees (a), (b) and (c) of Figure 2.3. Both split networks are examples of an implicit representation of horizontal transfer of genetic information within the gene trees. An explicit representation can be seen in (c), which is a reticulate network. Interestingly, the split sets of both consensus networks can be sampled from this reticulate network.

Note that for a reticulation r we have $X_r^T = X_r$, and that for any two directly dependent reticulations r, r', the sets X_r and $X_{r'}$ are necessarily disjoint in a galled network. If W is an arbitrary set of vertices, let X_W be the union of all X_w, $w \in W$, and analogously, let X_W^T be the union of all X_w^T, $w \in W$.

A vertex w of N with an incoming edge e is called *degenerated* if $X_w = X_{R(e)}$. If follows directly from this definition that all taxa descending from a degenerated vertex e can only be reached from $\beta(e)$ through a path containing a reticulation in $R(e)$.

The set of clusters $\mathcal{H}(e)$ corresponding to a tree edge e can now be defined as $\{X_w^T \cup X_R \mid R \subseteq R(e)\}$. We denote the union of all $\mathcal{H}(e)$ with e being an edge in E as $\mathcal{H}(E)$. We define $\mathcal{H}(N) := \bigcup_{e \in E_T} \mathcal{H}(e)$ and say that N *induces* $\mathcal{H}$ if $\mathcal{H} \subseteq \mathcal{H}(N)$. Note that $\mathcal{H}(N)$ is equal to the set of clusters $\cup_{T \in \mathcal{T}(N)} \mathcal{H}(T)$.

Sometimes we need an explicit description of a set of clusters, as we often have sets of clusters with a constant and a variable part. For this purpose we define $\mathcal{H}(\mathcal{C}, \mathcal{V}) = \{\mathcal{C} \cup V \mid V \in \mathcal{V}\}$ where $\mathcal{C}$ is a set of taxa (the constant taxa, that occur in all the clusters) and $\mathcal{V}$ is a collection of sets of taxa (the variable parts).

Whenever we try to reconstruct a reticulate network, we are interested in a minimal solution. Classic minimality is defined as the minimal number of reticulations needed for the reconstruction [GEL03]. Here we will define minimality more restrictivly.

Definition 2.2 *A reticulate network N is called* minimal with respect to $\mathcal{H}$ *if and only if N is a network with a minimal number of reticulations that induces the clusters $\mathcal{H}$, N minimizes the number of pairs of dependent reticulations, and within all those networks, minimizes the number of clusters that are sampled from edges that lie within reticulation cycles, then the number of edges that lie within it for each tree cycle and finally, the number of edges.*

For a set of clusters $\mathcal{H}$ induced by N, there may be clusters c that correspond to more than one tree edge. To eliminate this ambiguity with respect to the minimality of the network, we will choose one of the edges using the following three rules: (1) if at least one of the possible edges is not contained in any cycle we choose an edge outside of any cycle that is as far down the network as possible; (2) otherwise, if the clusters can be sampled from non-degenerate tree edges, we choose the edge that is as far down the network as possible; (3) otherwise, if the clusters can only be sampled from degenerate tree edges, we choose any one.

We denote $\mathcal{H}^S(e)$ to the set of clusters that are sampled from e using these two rules, hence $\mathcal{H}^S(e) \subseteq \mathcal{H}(e)$ and $\mathcal{H}^S(E)$ is the union of all $\mathcal{H}^S(e)$ where e is an edge in E.

Since we are often not directly interested in the set $\mathcal{H}$, but rather in the fact that this set of clusters $\mathcal{H}$ exists such that the galled network is minimal with respect to that set, we will suppress $\mathcal{H}$ and just state that the galled network is minimal.

Research History

Many analyses that use reticulate networks are based on bipartite data (binary alignments, splits etc.). For most algorithms available, the evolutionary model, under which the binary alignments are assumed to have evolved is the *infinite site model* [Kim69]. The main assumption of the model is that any position within an alignment may only mutate once. In the case of binary characters, this corresponds to a transition from state 0 to state 1 or vice versa. In general, the primitive state is assumed to be 0, and if it is known, the model only supports changes from the primitive state 0 to state 1. Even though the infinite site model is quite restrictive, the model is still a good approximation, even if the underlying dataset is moderately violating the assumptions of the model. [Rog92].

One interesting question associated with reticulate networks is how to reconstruct them, given a set of bipartite data and keeping the number of reticulations minimal. Minimizing the number of recombination events is motivated by the scientific principle of parsimony, since these events are assumed to occur very rarely in evolution. The general problem can be formulated as follows:

Problem 2.1 Parsimonious Reticulate Network Problem: *Given a set of bipartite data A, construct a reticulate network N that supports A and contains a minimal number of reticulations.*

It has been shown that the reconstruction of a reticulate network with a minimal number of reticulations is $NP-$hard in general [WZZ01; BS06], giving rise to a number of questions that need to be addressed: Is there a possibility of obtaining a lower and upper bound for the minimal number of reticulations events needed for a reconstruction? How can one perfectly, or greedily, compute a (minimal) reticulate network? How good are the results of a reconstruction? Is there a consistent way of decomposing a phylogenetic network into smaller independent problems?

A first lower bound for the calculation of the minimal number of reticulation was introduced by Hudson and Kaplan [HK85] which we will outline later. Other bounds have been proposed in [MG03; GH04; SH04; BB05; LSH05; BS06].

Two sites in an alignment A of binary sequences are said to be *incompatible* if they contain all four gametic types $0-0$, $0-1$, $1-0$ and $1-1$. If two sites are incompatible, they must have evolved using at least one recombination event. This test for incompatibility between two sites is known as the *four gamete test*. If the ancestral state is known (which is, as explained at the beginning of this section, $0-0$), two sites are incompatible if they contain the gametic types $0-1$, $1-0$ and $1-1$.

The *Hudson and Kaplan (HK)* bound [HK85] calculates a lower bound for the number of recombination events in a given alignment A of binary characters where this alignment has evolved under the infinite sites model. The bound is computed in two steps. The first step fills a matrix M, where $M(i,j)$ is 1 if the sites i and j in A are incompatible, and 0 otherwise.

The matrix represents a set of local bounds for all pairs of sites. These local bounds can be joined together to obtain a global bound, either using the original algorithm introduced in the article of Hudson and Kaplan (Appendix 2 in [HK85]), or the *Composite Method* introduced by Myers and Griffith in [MG03]. The Composite Method uses an integer linear programming formulation of the problem, that can be solved using a dynamic programming algorithm.

Lower bounds for the number of recombination events in a given alignment can be used together with a branch-and-bound approach to reconstruct the minimal recombination history of an alignment. The algorithm of Lyngsoe et.al. [LSH05] uses this approach to compute this type of a history for a given alignment of binary sequences. The algorithm assumes that only single crossovers have occurred and produces a single minimal history. Unfortunately, the program does not calculate all minimal solutions, owing to runtime problems.

The following simpler formulation of the reconstruction problem is tractable and different solutions have been proposed [Mad97; NWL04; GEL03; DGL04]:

Problem 2.2 Parsimonious Galled Tree Problem: *Given a set of bipartite data A, construct a reticulate network N that (a) contains only a minimal number of independent reticulations and (b) generates A, if one exists.*

Because of the hardness of the general reconstruction problem, reducing the runtime for the calculation of a minimal solution is only possible by reducing the complexity of the problem. One way to reduce the complexity of the reconstruction problem is to break the calculation into smaller parts. Importantly, the calculation of each small part of the reconstruction problem must be independent of the calculation of any other part. Consequently, finding a consistent approach to breaking up the calculation has drawn a lot of attention [BB04; HKLS05; GB05; HK07; DGS07]. The formulation of a general conjecture, for the decomposition of reticulate networks was first posed by Dan Gusfield at RECOMB 2005 [GB05]:

Conjecture 2.1 *For any Alignment A of binary characters, a fully-decomposed phylogenetic network always exists for A which has the minimum number of recombinations, out of all possible phylogenetic networks for A.*

At the same conference, we were among the first to prove a special case of this Decomposition Theorem [HKLS05]:

Theorem 2.4 *[Decomposition Theorem] Let N be a reticulate network and let $\mathcal{H}(N)$ be the set of* all *clusters that can be sampled from the network. Each 2-connected component of N contains all and only the splits contained in one connected component of $IG(\mathcal{H}(N))$.*

Similar results, using different definitions, interpretations and proofs, can be found in [BB04; GB05]. This initial formulation gave rise to an algorithm that is able to solve the following problem for a fixed number of reticulations efficiently:

(a)

a	00000 00000
b	01000 00000
c	11000 00000
d	11100 00000
e	01010 00000
f	01011 00000
g	10111 00000
h	00000 01000
i	00000 11000
j	00000 11100
k	00000 01010
l	00000 01011
m	00000 10111

(b)

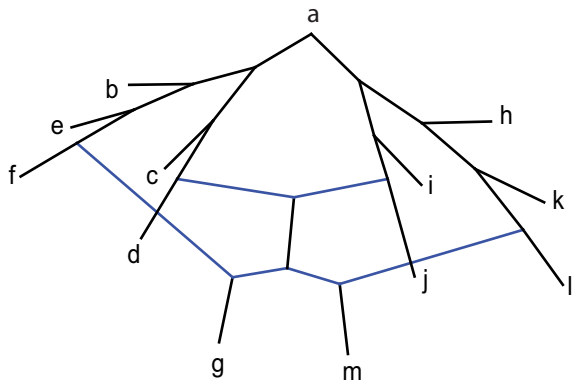


Figure 2.5: **Counter-example for Conjecture 2.2 published in [DGS07]** The alignment in (*a*) generates the minimal phylogenetic network shown in (*b*). Interestingly, the splits induced by the alignment generate two netted components in the incompatibility graph of the splits. One resulting from the first five sites and the other one from the last five sites. Nevertheless, the minimal reticulate network reconstructed from these splits (shown in (*b*)) has only one two-connected component.

Problem 2.3 Parsimonious Overlapping Reticulate Network from Gene Trees Problem: *Given a set of X-trees H construct a reticulate network N containing only independent or overlapping reticulations that supports H and contains a minimal number of reticulations, if one exists.*

Since binary sequences can be interpreted as splits, we were able to extend this solution to handle alignments of binary sequences [HK05].

In our approach, the calculation of a minimal solution depends on the full set of clusters that can be sampled from a reticulate network. In a recent article [HK07], we presented a Decomposition Theorem that holds for non-degenerate galled networks. Following the lead of [GB05], we also proposed a general Decomposition Conjecture:

Conjecture 2.2 *Given an input set of splits Δ, a minimal reticulation network N exists that explains Δ so that for any two tree edges e and f of N we have the following: Any two splits $S \in \Delta \cap \Sigma(e)$ and $S' \in \Delta \cap \Sigma(f)$ are contained in the same connected component of $IG(\Delta)$ if and only if e and f are contained in a common unoriented cycle of N.*

One difference between the conjecture of Gusfield and Bansal to our work is the restrictive formulation of minimality. We use the term *minimality* to describe a unique class of reticulate networks, for which we then prove the Decomposition Theorem. In contrast, Gusfield and Bansal only state at least one reticulate network exists that minimizes the number of reticulations for which the Decomposition Theorem holds. We believe our formulation to be more favorable since the properties of the resulting network are explicitly defined. Interestingly, our conjecture is not true in general, as was shown in [DGS07]. This article provides a very nice counter-example that is shown in Figure 2.5.

In Chapter 3, we will show that Conjecture 2.2 indeed holds for galled networks. Algorithms for computing minimal reticulate networks are shown in Chapter 4. An advantage of our approach is that we can use split networks to obtain a visualization of the clusters. By modifying the split network, we showed that one can obtain a reticulate network [HKLS05] that can also be labeled by mutations on the edges and sequences at internal vertices [HK05]. Nevertheless, obtaining a visualization algorithm for reticulate networks that is independent of split networks seemed to be an important subject. In Chapter 5 we present an optimization strategy that can be used to alter existing drawing algorithms for trees, to obtain visualizations of reticulate networks.

Chapter 3

Decomposing Galled Networks

Galled networks are substantially more general than galled trees. In contrast to the situation in galled trees where reticulation cycles are edge-disjoint, multiple reticulation cycles in a galled network may overlap along a common tree. In this chapter we will focus on the proof of the following Decomposition Theorem:

Theorem 3.1 *[Decomposition Theorem for minimal galled networks] Let N be a minimal galled network that represents $\mathcal{H} \subseteq \mathcal{H}(N)$. Then any two clusters c, c' are contained in the same connected component of $IG(\mathcal{H})$ if and only if two edges e, f exists in N with $c \in (\mathcal{H} \cap \mathcal{H}^S(e))$ and $c' \in (\mathcal{H} \cap \mathcal{H}^S(f))$ that are contained in a common cycle.*

We have published a version of this theorem, restricted to non-degenerated galled networks in [HK07]. The degeneracy of some edges within the networks generates a complex combinatory structure within the sampled clusters. Consequently, the proof of the theorem is rather complex. Basically, one can divide the proof into three different parts. In the first part, we prove some basic structural properties of minimal galled networks. Secondly, we describe the structural properties of clusters generated by degenerated edges in detail. Finally, we will prove the theorem.

To formulate our proof, we have to introduce some additional definitions and results.

Let N be a minimal galled network and r be a reticulation of N with tree cycle $C = (r, p(r), w_1, e_1, \ldots, e_{k-1}, w_k, q(r), r)$.

For $1 \leq i \leq k$, let $\varphi_C(w_i)$ denote the set of taxons reachable from w_i through a path which passes through no edges in C and no reticulations directly dependent on r. For $1 \leq i \leq k-1$, let $R^+_C(e_i)$ denote the set of reticulations r' with $r' \notin R(e_i)$ but $r' \in R(e_j)$ for some $e_j \in C$, e_j descendant of e_i. Finally, let o denote the index in $\{1, \ldots k\}$ for which the outgroup is an element of $\varphi_C(w_o)$.

It is now straightforward to describe the set of clusters of an edge e in the cycle C explicitly as

$$\mathcal{H}(e) = \mathcal{H}(\mathcal{C}^T \cup \mathcal{C}^R, \mathcal{V}) \quad \text{where}$$

$$\mathcal{C}^T = \bigcup\{\varphi_C(w) \mid w \in C,\ w \text{ is a descendant of } e\},$$
$$\mathcal{C}^R = \bigcup\{X_{r'} \mid r' \in R^+_C(e)\} \text{ and}$$
$$\mathcal{V} = \{X_R \mid R \subseteq R(e)\}.$$

The next three lemmatas give insights into the structure of (minimal) galled networks. The first lemma reflects the simple observation that in a galled network the descendants of a reticulation can only be reached from the root of the galled network by paths that contain the reticulation. The second lemma is a prime example of the structural consequences that minimization of the number of edges within the reticulation cycles has on the minimal galled network. The third lemma shows that the degenerated parts within the network consist of at most one edge each. One of the main implications of this insight is that we have to gain a detailed understanding of the combinatoric structure of this edges to be able to prove the decomposition theorem. This part of the proof begins with the definition following Lemma 3.3.

Lemma 3.1 *Let N be a galled network and $C = (r, p(r), w_1, e_1, \ldots, w_i, e_i, \ldots, e_{k-1}, w_k, q(r), r)$ a cycle in N. If for $1 < i < k$, the vertex w_i is a reticulation with either $e_i = p(w_i)$ or $e_{i-1} = p(w_i)$, we have either $e_{i-1} = q(w_i)$ or $e_i = q(w_i)$ respectively.*

Proof: Since a cycle is not oriented, we only show the proof for one direction. That is let w_i be a reticulation and $e_i = p(w_i)$ be its first reticulation edge. Assume e_{i-1} to be a tree egde. Since e_{i-1} and e_i are contained in a cycle, a reticulation r' must exists, such that $\alpha(p(r'))$ is a descendant of w_i and $\alpha(q(r'))$ is not a descendant of w_i. Consequently, w_i is an element of the tree cycle of r'. Since w_i is a reticulation, all the vertices which can be reached from w_i by tree edges are descendants of w_i. Thus, $\alpha(q(r'))$ would be a descendant of w_i, which is a contradiction. So e_{i-1} cannot be a tree edge, which proves the lemma. □

Lemma 3.2 *Let N be a minimal galled network. Furthermore, let r be a reticulation in N with tree cycle $C = (r, p(r), w_1, e_1, \ldots, w_k, q(r), r)$. For $1 < i < k$, we have that if $\varphi_C(w_i) = \emptyset$ then $R(e_{i-1}) \not\subseteq R(e_i)$.*

Proof: Note that for $i = o$, we cannot have $\varphi_C(w_i) = \emptyset$. We show the proof for $i < o$, the case $i > o$ is symmetric. Assume $\varphi_C(w_i) = \emptyset$ and $R(e_{i-1}) \subseteq R(e_i)$.
We will now compare the clusters that can be sampled from e_i to the clusters that can be sampled from e_{i-1}. To this end, we use the explicit description of the clusters: $\mathcal{H}(e_i) = \mathcal{H}(\mathcal{C}_i^T \cup \mathcal{C}_i^R, \mathcal{V}_i)$.

- $\mathcal{C}_i^T = \bigcup\{\varphi_C(w) \mid w \in C,\ w \text{ is a descendant of } e_i\} = \bigcup\{\varphi_C(w_j) \mid 1 \leq j \leq i\}$
  If $\varphi_C(w_i) = \emptyset$ we have $\{\varphi_C(w_j) \mid 1 \leq j \leq i-1\} = \{\varphi_C(w_j) \mid 1 \leq j \leq i\}$, so $\mathcal{C}_{i-1}^T = \mathcal{C}_i^T$.

- $\mathcal{C}_i^R = \bigcup\{X_{r'} \mid r' \in R_C^+(e_i)\} = \bigcup\{R(e_j) \mid j < i\} \setminus R(e_i)$
  First note that $\bigcup\{R(e_j) \mid j < i-1\} \setminus R(e_{i-1}) = \bigcup\{R(e_j) \mid j < i\} \setminus R(e_{i-1})$.
  Since $R(e_{i-1}) \subseteq R(e_i)$ we have $\bigcup\{R(e_j) \mid j < i\} \setminus R(e_i) \subseteq \bigcup\{R(e_j) \mid j < i\} \setminus R(e_{i-1})$
  Now assume these two last sets are not equal. Then there will be a

reticulation r' with $r' \in \bigcup\{R(e_j) \mid j < i\} \setminus R(e_{i-1})$ but $r' \notin \bigcup\{R(e_j) \mid j < i\} \setminus R(e_i)$. So we have $r' \in R(e_j)$ for an index $j < i$ and $r' \in R(e_i)$ but $r' \notin R(e_{i-1})$, which is impossible. Thus, the two sets must be equal, so $\mathcal{C}^R_{i-1} = \mathcal{C}^R_i$.

- $\mathcal{V}_i = \{X_R \mid R \subseteq R(e_i)\}$
  From the assumption $R(e_{i-1}) \subseteq R(e_i)$, it follows directly that $\mathcal{V}_{i-1} \subseteq \mathcal{V}_i$.

Thus, all the clusters that can be sampled from e_{i+1} could also be sampled from e_i, which is a contradiction to the minimality of N! □

Note that since tree cycles are undirected, it also holds that $R(e_i) \not\subseteq R(e_{i-1})$.

Lemma 3.3 *Let N be a minimal galled network and e be an tree edge in N. If every path from $\beta(e)$ to a taxon in $X_{\beta(e)}$ contains a reticulation in $R(e)$, then every reticulation in $R(e)$ is a direct descendant of $\beta(e)$.*

Proof: For every subset R' of $R(e)$, the cluster $X_{R'}$ can be sampled from e. Assume that not all the reticulations in $R(e)$ are direct descendants of $\beta(e)$. Then an edge e' below e exists for which the set R' of reticulations that are descendants of e' is not empty. Now any cluster that can be sampled from e', equals $X_{R''}$, where R'' is a subset of R' and thus also a subset of R. So, any cluster that can be sampled from e' can also be sampled from e contradicting the minimality of N and thus, proving the lemma. □

The next part of the preliminary work for the decomposition theorem is focused on three parts. The first part builds up a mathematical structure to reflect the complex structure that can be sampled from degenerated edges within the reticulation network. The second part proves that for two degenerated edges that share a descending reticulation, sets of sampled clusters, from these edges exists that are contained in the same netted component (Lemma 3.6). Ensuring that these two sets of clusters are contained in one netted component is important, but for the theorem to be of any further value we have to show that at least one other special cluster exists within this netted component that was not sampled from a degenerated edge. This special cluster connects the clusters in the sets to the clusters sampled from the non-degenerated edges in the backbone of the reticulation cycle.

Let H be a set of clusters and R be a set of reticulations, such that every reticulation in R is contained in a cluster in H and every cluster in H is a subset of X_R.

Definition 3.1 *A pair (R', H') with $R' \subseteq R$ and $H' = \{c \in H \mid c \subseteq X_{R'}\}$ is called* closed *with respect to (R, H) if H' is not empty and for all clusters $c \in H$ it holds that $c \cap X_{R'} \in \{\emptyset, c, X_{R'}\}$.*

If we have R' and (R, H) then we can define H' appropriatly. So if $(R', \{c \in H \mid c \subseteq X_{R'}\})$ is closed with respect to (R, H) we say that R' is closed with respect to (R, H).

We say, R' is minimally closed with respect to (R, H) if it is closed and it contains no proper subset that is closed.

Note that the following properties hold for closed subsets: For each set of reticulations R, if we set $H = \{c \mid c \subseteq X_R\}$ then the pair (R, H) is closed with respect to itself . Moreover, if (R', H') is closed with respect to (R, H) and (R'', H'') is closed with respect to (R', H') then R'' is closed with respect to (R, H).

If we take two pairs, (R_1, H_1) and (R_2, H_2), which are both closed with respect to (R, H) then it is easy to check that if $H_1 \cap H_2$ is not empty then $(R_1 \cap R_2, H_1 \cap H_2)$ is also closed with respect to (R, H). However, we cannot conclude that $R_1 \cup R_2$ is also closed with respect to (R, H). But if it is closed, we are interested in the question, whether $H_1 \cup H_2$ is the corresponding set of clusters, that is $H_1 \cup H_2 = \{c \in H \mid c \subseteq X_{R_1} \cup X_{R_2}\}$. To this end, we define:

Definition 3.2 *A pair (R', H') is called* decomposable *if a set of pairs $(R_1, H_1), \ldots, (R_k, H_k)$, which are closed with respect to (R, H), exists such that $R_i \subsetneq R'$ and $\bigcup_{1 \leq i \leq k} H_i = H'$.*

In general, the decomposability of closed sets is of great importance. The clusters that make a closed set non-decomposable often play a crucial role. Therefore we define $\partial H' = \{c \in H' \mid c \notin H_i \text{ for all } (R_i, H_i) \neq (R', H') \text{ closed with respect to } (R', H')\}$.

Note that for every reticulation r in R, there is a non-decomposable closed subset R' of R such that $r \in R' \subseteq R$. So every reticulation is contained in a non-decomposable closed subset.

Lemma 3.4 *Let (R', H') be non-decomposable. If $\partial H' \backslash \{X_{R'}\}$ is non-empty, then for any $r \in R'$ there exists a cluster c in $\partial H' \backslash \{X_{R'}\}$ with $X_r \subseteq c$.*

Proof: Assume no cluster c in $\partial H' \backslash \{X_{R'}\}$ with $X_r \subseteq c$ exists. If r is not contained in any closed subset (R_i, H_i) of (R', H') with $R_i \subsetneq R'$, the set $(R' \backslash \{r\}, H' \backslash \{X_{R'}\})$ would be closed with respect to (R', H') and consequently $\partial H' \backslash \{X_{R'}\}$ would be empty, which contradicts the assumption of the lemma.

Consequently, r must be contained in at least one closed subset (R_i, H_i) of (R', H') with $R_i \subsetneq R'$. Choose (R_i, H_i) to be maximal. We show that $(R' \backslash R_i, H' \backslash H_i)$ is closed with respect to (R', H').

Note that for any cluster c' in $\partial H' \backslash \{X_{R'}\}$ holds $c' \cap X_{R_i} = \emptyset$ and for any cluster c in H' we have $c \cap X_{R_i} \in \{\emptyset, c\}$. Any cluster c with $c \cap X_{R_i} = c$ is contained in H_i. Thus we have for any cluster c in $H' \backslash H_i$ that $c \cap X_{(R' \backslash R_i)} \in \{\emptyset, c, X_{(R' \backslash R_i)}\}$. Furthermore, at least one cluster c in $\partial H' \backslash \{X_{R'}\}$ with $c \cap X_{(R' \backslash R_i)} = c$ exists since $\partial H' \backslash \{X_{R'}\}$ is non-empty. Thus, $(R' \backslash R_i, H' \backslash H_i)$ is closed with respect to (R', H'), which leads to a contradiction of the non-decomposable closedness of (R', H'). □

Lemma 3.5 *Let (R', H') be non-decomposable. Then for any pair of clusters c_1, c_k in $\partial H' \setminus \{X_{R'}\}$, a chain $c_1, c_2, \ldots, c_k$ exists such that c_i in $\partial H' \setminus \{X_{R'}\}$ and c_i is incompatible with c_{i+1}.*

Proof: The proof is separated into two parts. In the first part, we show that no single cluster exists that is compatible with all other clusters, and in the second part, we show the general claim.

Assume a cluster c in $\partial H' \setminus \{X_{R'}\}$ exists that is compatible with all other clusters in $\partial H' \setminus \{X_{R'}\}$. Let $R'' \subseteq R'$ denote the set of all reticulations r'' with $X_{r''} \subseteq c$, that is $X_{R''} = c$. Note that since $X_{R'}$ is not in $\partial H' \setminus \{X_{R'}\}$, it follows that $R'' \subsetneq R'$. We need to show that R'' is closed with respect to (R', H'), so for $c' \in H'$ we want to show that $c' \cap X_{R''} \in \{\emptyset, c', X_{R''}\}$. Because $X_{R''} = c$, we can also show that $c' \cap c \in \{\emptyset, c', c\}$.

- If $c' \in \partial H' \setminus \{X_{R'}\}$ then by the assumption c' is compatible with c, therefore $c' \cap c \in \{\emptyset, c', c\}$.
- If $c' = X_{R'}$ then $c' \cap X_{R''} = X_{R'} \cap X_{R''} = X_{R''}$.
- Finally for (R_i, H_i) closed with respect to (R', H') and $c' \in H_i$, we have $c' \subseteq X_{R_i}$. Observe that since R_i is closed and $c \notin H_i$ we know that $c \cap X_{R_i} \in \{\emptyset, X_{R_i}\}$. If $c \cap X_{R_i} = \emptyset$ then $c' \cap c \subseteq X_{R_i} \cap c = \emptyset$, and if $c \cap X_{R_i} = X_{R_i}$ then $c' \subseteq X_{R_i} \subseteq c$ and so $c \cap c' = c'$.

So R'' is closed with respect to (R', H') and thus, $c = X_{R''}$ can not be in $\partial H' \setminus \{X_{R'}\}$. So it is not possible to find a cluster in $\partial H' \setminus \{X_{R'}\}$ that is compatible with all other clusters in $\partial H' \setminus \{X_{R'}\}$.

Now we can divide the clusters in $\partial H' \setminus \{X_{R'}\}$ into blocks of clusters $H_1, \ldots, H_k$ for which it holds that for any pair of clusters in a block, a chain of clusters exists such that consecutive elements in the chain are incompatible. Note that any cluster in $\partial H' \setminus \{X_{R'}\}$ can only be present in one block. For any block H_i, denote R_i as consisting of the reticulations r for which X_r is present in a cluster in H_i. We will show that R_i is closed with respect to (R', H'):

- Note that for every closed subset R_c of R', either $R_c \cap R_i = \emptyset$ or $R_c \subseteq R_i$. So for every cluster c that is contained in a closed subset (R_c, H_c) of (R', H'), we get $c \cap X_{R_i} \in \{\emptyset, c\}$.
- For c in H_i or $c = X_{R_i}$, we have $c \cap X_{R_i} = c$.
- For c in H_j with $j \neq i$ assume $c \cap X_{R_i} \neq \emptyset$. Then $c_i \in H_i$ exists with $c \cap c_i \neq \emptyset$. Since c and c_i are in different blocks, they must be compatible, so either $c_i \subset c$ or $c \subset c_i$.
  - If $c \subset c_i$ then c is contained in X_{R_i}; therefore $c \cap X_{R_i} = c$.
  - If $c_i \subset c$ then all the clusters in H_i that are incompatible with c_i are also contained in c:
    Assuming that $c_i' \in H_i$ is incompatible with c_i, then $c_i \cap c_i' \neq \emptyset$ and $c_i \not\subseteq c_i'$, therefore $c \cap c_i' \neq \emptyset$ and $c \not\subseteq c_i'$. Because c and c_i' must be compatible, we get $c_i' \subseteq c$. Since every cluster in H_i is connected to c_i by a chain of incompatible clusters, every cluster in H_i is contained in c, thus $c \cap X_{R_i} = X_{R_i}$

So R_i is closed with respect to (R', H'). Since H_i contains clusters from $\partial H'$, this implies that $R_i = R'$ for all blocks.

Assume we have two different blocks, H_1 and H_2. Then for any reticulation r in R', we find two clusters $c_1 \in H_1$ and $c_2 \in H_2$ such that $X_r \subset c_1$ and $X_r \subset c_2$, and thus $c_1 \cap c_2 \neq \emptyset$. Since c_1 and c_2 are in different blocks, they must be compatible, so either $c_1 \subset c_2$ or $c_2 \subset c_1$. Assume with out lost of generality that $c_1 \subset c_2$. By the same argument as above, we infer that in this case every cluster in H_1 must be contained in c_2, so $X_{R_1} \subseteq c_2$. Since $R_1 = R'$, we get $c_2 = X_{R'}$, which is a contradiction since c_2 is in H_2 which is a subset of $\partial H' \setminus \{X_{R'}\}$. So we can not have two different blocks. □

For closed subsets, we are interested in clusters that contain only taxa descending from a certain set of reticulations. This links the degenerated elements of a galled network to closed sets. Let N be a minimal galled network and r a reticulation with tree cycle $C = (r, p(r), w_1, e_1, \ldots, w_k, q(r), r)$, denote E_r^* to be the set of edges in C that lead to degenerated vertices. Note that because of Lemma 3.3 we have $E_r^* \subseteq \{e_1, e_{k-1}\}$. Furthermore, denote E_r^T as any other edge in $C \setminus \{p(r), q(r)\}$.

Lemma 3.6 *Assume N to be a minimal galled network and let r be a reticulation in N with tree cycle $C = (r, p(r), w_1, \ldots, w_k, q(r), r)$ such that $E_r^* = \{e_1, e_{k-1}\}$. We denote (R^1, H^1) to be the maximal non-decomposable closed set with respect to $(R(e_1), \mathcal{H}^S(e_1))$ that contains r (Possibly $R^1 = R(e_1)$). Analogously denote (R^{k-1}, H^{k-1}) to be the maximal non-decomposable closed set with respect to $(R(e_{k-1}), \mathcal{H}^S(e_{k-1}))$ that contains r.*

Then it holds that all clusters in ∂H^1 are contained in the same netted component as all clusters in ∂H^{k-1}.

Proof: Note that as a consequence of Lemma 3.4, at least one cluster c_1 in ∂H^1 exists such that $X_r \subsetneq c_1$ and at least one cluster c_{k-1} in ∂H^{k-1} with $X_r \subsetneq c_{k-1}$. We prove this lemma, by demonstrating that in any case where the two sets of clusters are not contained in the same netted component, we can either sample the clusters of H^1 at e_{k-1} or we can sample the clusters in H^{k-1} at e_1, contradicting the minimality of N.

Since c_1 and c_{k-1} bot contain X_r we have that $c_1 \cap c_{k-1} \neq \emptyset$ and consequently, if both clusters are not incompatible with each other, either $c_1 \subset c_{k-1}$ or $c_{k-1} \subset c_1$.

Let us assume without lost of generality that $c_{k-1} \subset c_1$. If $c_{k-1} = X_{R^{k-1}}$, than we could sample all clusters in H^{k-1} at edge e_1, contradicting the minimality of N.

Thus let c_{k-1} not be equal to $X_{R^{k-1}}$. Bear in mind, that according to Lemma 3.5, for any pair $\overline{c}_1, \overline{c}_l$ of clusters in $\partial H^{k-1} \setminus \{X_{R^{k-1}}\}$ a chain $\overline{c}_1, \overline{c}_2, \ldots, \overline{c}_l$ exists such that $\overline{c}_i$ and $\overline{c}_{i+1}$ are incompatible. Denote C_1 to contain all clusters of $\partial H^{k-1} \setminus \{X_{R^{k-1}}\}$ that are incompatible with c_{k-1} and iterativly denote C_{i+1} to contain those clusters in $\partial H^{k-1} \setminus \{X_{R^{k-1}}\}$ that are not contained in $\bigcup_{j=1}^{i} C_j$, but that are incompatible with a cluster in C_i. We will show via induction that either a cluster in ∂H^{k-1} exists that is incompatible with c_1 or all clusters in ∂H^{k-1} are contained in c_1.

For any cluster $\overline{c}_1$ in C_1 we have $\overline{c}_1 \cap c_{k-1} \neq \emptyset$ and $\overline{c}_1 \backslash (\overline{c}_1 \cap c_{k-1}) \neq \emptyset$. Consequently, $\overline{c}_1 \cap c_1 \neq \emptyset$ and $c_1 \backslash (\overline{c}_1 \cap c_1) \neq \emptyset$. Therefore, either $\overline{c}_1$ is incompatible with c_1, or $\overline{c}_1 \subset c_1$.

Assume that for all clusters $\overline{c}_i$ in C_i, it holds that $\overline{c}_i \subset c_1$. We have to show that any cluster in C_{i+1} is either incompatible with c_1 or contained in it.

For any cluster $\overline{c}_{i+1}$ in C_{i+1}, a cluster $\overline{c}_i$ in C_i exists such that $\overline{c}_{i+1}$ is incompatible with $\overline{c}_i$ and $\overline{c}_i \subset c_1$. Again, we have $\overline{c}_{i+1} \cap c_i \neq \emptyset$ and $\overline{c}_i \backslash (\overline{c}_{i+1} \cap c_i) \neq \emptyset$; consequently, $\overline{c}_{i+1} \cap c_1 \neq \emptyset$ and $c_1 \backslash (\overline{c}_{i+1} \cap c_1) \neq \emptyset$. Thus either $\overline{c}_{i+1}$ is incompatible with c_1, or $\overline{c}_{i+1} \subset c_1$.

Consequently, if all clusters in $\partial H^{k-1} \backslash \{X_{R^{k-1}}\}$ are compatible with c_1, they must be contained in c_1 and thus, could be sampled on edge e_1. Futhermore, because of Lemma 3.4, the cluster $X_{R^{k-1}}$ must also be contained in c_1. This contradicts the minimality of N.

Finally note that a symmetric argument holds for the case $c_1 \subset c_{k-1}$ and consequently the lemma holds.

□

Lemma 3.7 *Let (R', H') be non-decomposable and closed with respect to $(R(E_r^*), \mathcal{H}^S(E_r^*))$. At least one cluster c in $\mathcal{H}^S(E_r^T)$ exists such that c separates $X_{R'}$.*

Proof: First, note that R' contains at least two reticulations: If a reticulation r' exists such that $R' = \{r\}$, then $H' = \{X_r\}$ and the cluster X_r has been sampled from an edge in a reticulation cycle, the network N will not be minimal, since we could add an edge between r and its descendants and then sample X_r from this edge, which is not in a reticulation cycle.

We prove the lemma by contradiction. Suppose none of the cluster sampled from edges in E_r^T seperates $X_{R'}$. Consequently, the minimality of N forces us to use the same tree cycles for all reticulations in R'; thus $E_{r'}^T = E_{r''}^T$ for all $r', r'' \in R'$. We show that in this case N is not minimal:

Remove all the reticulations in R' from N. Attach a new reticulation r_{new} to the first and the last vertices that are shared by all tree cycles of the elements of R'. (Since $E_{r'}^T = E_{r''}^T$ for all $r', r'' \in R'$, the tree cycle of r_{new} contains at least this part of all other tree cycles). Next, attach two new vertices n_1 and n_2 as descending vertices to r_{new}. Choose an arbitrary reticulation r' in R' and reconnect it as a descending vertex to n_1. Finally, reconnect all other reticulation in $R' \setminus \{r'\}$ to n_1 and n_2.

Note that the new network N' which was obtained from N in the way we just described, displays the same clusters as N. Consider a cluster c sampled from an edge e in $E_r^T \cup E_r^*$. If e is in E_r^T, then the cluster still can be sampled from e in the new network. If e is in E_r^*, then we have to discerm whether $X_{r'}$ is contained in c. Note that since (R', H') is closed with respect to $(R(E_r^*), \mathcal{H}^S(E_r^*))$, we know that $c \cap X_{R'} \in \{\emptyset, c, X_{R'}\}$. If $c \cap X_{R'} = \emptyset$, then the cluster c still can be sampled from e. If $c \cap X_{R'} = c$, then $c \subseteq X_{R'}$, and therefore c can be sampled either from the edge connecting n_1 to r_{new} (if $X_{r'}$ is contained in c) or from the edge connecting n_2 to r_{new} (if $X_{r'}$ is not contained in c). Finally, if $c \cap X_{R'} = X_{R'}$, then e is in the tree cycle

of all reticulations in R' and thus is also in the tree cycle of r_{new}, so c still can be sampled from e. Furthermore, note that N' contains exactly as many reticulations as N: In the new network, r' is no longer a reticulation, but in compensation we added a new reticulation r_{new}. However, the important innovation is that r_{new} is not dependent on any reticulations in $R' \setminus \{r'\}$, and consequently N was not minimal. □

Lemma 3.8 *For every subset R' of $R(E_r^*)$ (not necessarily closed) and every cluster c in $\mathcal{H}^S(E_r^T)$ that seperates $X_{R'}$, it holds that c is incompatible with $X_{R'}$.*

Proof: Assume c to be a cluster in E_r^T that seperates $X_{R'}$. From Lemma 3.3, follows that at least one vertex w_j exists such that $\varphi_C(w_j)$ is contained in c. Therefore, c separates $X_{R'}$ and c contains at least one taxon that is not in $X_{R'}$; thus c and $X_{R'}$ are incompatible. □

Lemma 3.9 *Let (R', H') be non-decomposable and closed with respect to (R, H). If a cluster c in $\mathcal{H}^S(E_R^T)$ is incompatible with $X_{R'}$ and $\partial H' \backslash \{X_{R'}\}$ is non-empty, there exists at least one cluster c' in $\partial H' \backslash \{X_{R'}\}$ that is in the same netted component as c.*

Proof: Let $\partial H' \backslash \{X_{R'}\}$ be non-empty and c be a cluster from E_R^T that is incompatible with $X_{R'}$. Assume no c' in $\partial H' \backslash \{X_{R'}\}$ exists such that c and c' are incompatible, so for any cluster c' in $\partial H' \backslash \{X_{R'}\}$ it holds that at least one of the sets $c \cap c'$, $c' \backslash (c \cap c')$ or $c \backslash (c \cap c')$ must be empty. Since at least one vertex w_j exists such that $\varphi_C(w_j)$ is contained in c, $c \backslash (c \cap c')$ cannot be empty. Thus, either $c' \backslash (c \cap c')$ is empty, i.e. c' is contained in c, or $c \cap c'$ is empty. Therefore, $\partial H' \backslash \{X_{R'}\}$ can be divided in two disjoint sets of clusters: the ones that are contained in c and the ones that are disjoint to c. Since these two sets are compatible, we can conclude by Lemma 3.5 that one of them is empty. So either all clusters in $\partial H' \backslash \{X_{R'}\}$ are contained in c, or they all are disjoint to c.

Denote R'' as the minimal set of reticulations in R' such that for every cluster c' in $\partial H' \backslash \{X_{R'}\}$ it holds that $c' \subseteq X_{R''}$. Now because of the previous observation, either $X_{R''}$ is contained in c or $X_{R''}$ is disjoint to c, so R'' can not contain all reticulations in R', because c separates $X_{R'}$. To show that R'' is closed with respect to (R', H'), take an arbitrary cluster c' from H' and consider $c' \cap X_{R''}$:

- For c'in $\partial H' \backslash \{X_{R'}\}$ we have $c' \cap X_{R''} = c'$.
- If c' is in any closed subset H_i, then we have to distinguish two cases:
  - If for all reticulations r with $X_r \subseteq c'$, we have $r \notin R''$ then $c' \cap X_{R''} = \emptyset$.
  - If at least one reticulation $r \in R''$ with $X_r \subseteq c'$ exists then there must be a cluster c_r in $\partial H' \backslash \{X_{R'}\}$ with $X_r \subseteq c_r$. Since H_i is closed, we know that $c_r \cap X_{R_i} \in \{\emptyset, c_r, X_{R_i}\}$. Since X_r is contained in c', and thus in X_{R_i}, we know that $c_r \cap X_{R_i}$ is not empty. Since c_r cannot be in H_i, $c_r \cap X_{R_i}$ cannot be c_r. Therefore we get $c_r \cap X_{R_i} = X_{R_i}$, so $c' \subseteq X_{R_i} \subseteq c_r \subseteq X_{R''}$ and thus, $c' \cap X_{R''} = c'$.

Since R'' is closed with respect to (R', H'), a maximally closed subset R_j with $X_{R''} \subseteq X_{R_j}$ must exist, and therefore ever cluster in $\partial H' \backslash \{X_{R'}\}$ would be in H_i, a contradiction to the construction of $\partial H' \backslash \{X_{R'}\}$. □

We have so far shown a group of very useful properties for closed pairs that will help use to more clearly understand the sets of clusters that can be generated from edges within the minimal galled network. For example assume any degenerated edge e for which it holds that $(R(e), \mathcal{H}(e))$ contains two maximally closed pairs $(R_1, H_1), (R_2, H_2)$, such that $\mathcal{H}(e) \backslash \bigcup_{l=1}^{2} H_l$ is empty. The question arrises if this is a contradiction to the minimality of the galled network. Interestingly this is not necesserly the case. If a r_1 in R_1 and a r_2 in R_2 exist such that r_1 and r_2 are directly dependent, keeping the clusters in H_1 and H_2 together on one edge minimizes the network. Otherwise, if no such r_1 and r_2e exists, we would need to map each closed pair onto its own edge, minimizing the pairs of dependent reticulations. We formalize this observation in the following lemma:

Lemma 3.10 *Consider a degenerated edge e and let $(R_1, H_1), \ldots, (R_k, H_k)$ be all maximum closed pairs with respect to $(R(e), \mathcal{H}(e))$. Let a maximum closed pair (R_i, H_i) exist with the property that there is no cluster c in $\overline{\mathcal{H}}(e) = \mathcal{H}(e) \backslash \bigcup_{l=1}^{k} H_l$ with $X_{R_i} \subset c$. For any $j \neq i$, a reticulation r_j in R_j and a reticulation r_i in R_i must exist such that $E^T_{r_j} \cap E^T_{r_i}$ is non-empty.*

Proof: For simplicity of notation, let $i = 1$ and $j = 2$. Assume that a r_1 in R_1 and a r_2 in R_2 exist such that $E^T_{r_1} \cap E^T_{r_2}$ is empty. Furthermore, let there be no cluster c in $\mathcal{H}(e) \backslash \bigcup_{l=1}^{k} H_l$ with $X_{R_i} \subset c$. The following construction leads to a contradiction: Attach a new vertex n_1 via the edge e_1 as a descendant to $\alpha(e)$, remove all reticulations in R_1 from $\beta(e)$ and re-attach them as descendants to n_1. The clusters in H_1 can now be sampled from e_1 and since c in $\mathcal{H}(e) \backslash \bigcup_{l=1}^{k} H_l$ with $X_{R_i} \subset c$ exists, there is no cluster that cannot be sampled in the original galled network that cannot be sampled in the modified galled network. The important innovation is that r_1 and r_2 are independent, since the only edge that was shared by their tree cycles was e; thus we are able to minimize the number of pairs of dependent reticulations further, which is a contradiction to the assumption that N is minimal. □

Now we have all definitions and properties necessary for the proof of the theorem:

Proof: "⇒":

This direction is easy and was first shown in [HKLS05]. Let e and f be two edges not contained in a cycle. Then a *cut-edge* h exists (or at least a *cut-vertex*, which can be refined to provide a cut-edge h) that separates e and f. The edge h induces the same cluster c in every tree $T \in \mathcal{T}(N)$. Thus, every cluster $c' \in \mathcal{H}(N)$ subdivides either c or $X \backslash \{c\}$, but not both sets. This implies the claim.

"⇐":

The proof is separated into four main steps. Let r be an arbitrary reticulation with tree cycle $C = (r, p(r), w_1^r, e_1^r, \ldots, e_{k-1}^r, w_k^r, q(r), r)$ and denote f as the index of the first edge in C that is contained in E_r^T and l as the index of the last edge in C contained in E_r^T. In the first step, we show that a set of clusters in $\mathcal{H}^S(e_f^r)$ exists that is incompatible with a set of clusters in $\mathcal{H}^S(e_l^r)$. In the second step, we show that the sampled clusters $\mathcal{H}^S(E_r^T \backslash \{e_f, e_l\})$ are contained in the same netted component as the clusters of the first step. In these first two steps, we use the symmetric properties that arise from the undirected tree cycles and restrict our attention to the two cases (a) $o \notin \{f, l+1\}$ and (b) $o = l+1$.

The first two steps ensure that for any non-degenerated tree edge e in a cycle of N for which a reticulation r' in $R(e)$ exists such that $e \notin \{e_f^{r'}, e_l^{r'}\}$, the clusters sampled from e are contained in the same netted component. Consequently, we show in the third step that for any non-degenerated tree edge e in N that is contained in a cycle and for which it holds that for any r' in $R(e)$, $e \in \{e_f^{r'}, e_l^{r'}\}$ the set of clusters $\mathcal{H}^S(e)$ are contained in the same netted component.

Finally, in the fourth step, we show that all clusters sampled from a degenerated edge are contained in the same netted component as the clusters sampled in the tree cycles of reticulations that directly decend from the degenerated edge.

First step:

We start out the proof with the case where we assume $E_r^* = \{e_1, e_{k-1}\}$.

Case (a):

Either $(R(e_1^r), \mathcal{H}^S(e_1^r))$ is non-decomposable and closed with respect to itself, or a set $(R_1^1, H_1^1), \ldots, (R_l^1, H_l^1)$ of maximally non-decomposable closed subsets exists such that $\mathcal{H}^S(e_1^r) = \cup_{m=1}^{l} H_m^1$. We denote (R^1, H^1) as either $(R(e_1^r), \mathcal{H}^S(e_1^r))$ or the maximal non-decomposable closed subset that contains r. Using a symmetric argumentation for e_{k-1} we denote (R^{k-1}, H^{k-1}) as either $(R(e_{k-1}^r), \mathcal{H}^S(e_{k-1}^r))$ or as the maximally non-decomposable closed subset that contains r.

Since N is minimal, at least one cluster c_f in $\mathcal{H}^S(e_f^r)$ and a reticulation r_1 in R^1 exists such that $X_{r_1} \subsetneq c_f$. We denote the set $\overrightarrow{\mathcal{H}}_{R^1}^S(e_f^r)$ as consisting of all those clusters in $\mathcal{H}^S(e_f^r)$ that contain at least one X_{r_1} with $r_1 \in R^1$. By symmetry, at least one cluster c_l in $\mathcal{H}^S(e_l^r)$ and a r_{k-1} in R^{k-1} also exists such that $X_{r_{k-1}} \subset c_l$. Analogously to e_f^r, define $\overrightarrow{\mathcal{H}}_{R^{k-1}}^S(e_l^r)$ to consist of all those clusters in $\mathcal{H}^S(e_l^r)$ that contain at least one $X_{r_{k-1}}$ with $r_{k-1} \in R^{k-1}$.

We will show that the sets of clusters $\overrightarrow{\mathcal{H}}_{R^1}^S(e_f^r)$ and $\overrightarrow{\mathcal{H}}_{R^{k-1}}^S(e_l^r)$ are contained in the same netted component. First note that by Lemma 3.6, we know that all the clusters in ∂H^1 and and all the clusters in ∂H^{k-1} are contained in the same netted component if they are non-empty. By Lemma 3.8 and Lemma 3.9, any cluster in $\mathcal{H}^S(e_f^r)$ not completely containing or missing $X_{R^1 \cup R^{k-1}}$ is in the same netted component as ∂H^1 and ∂H^{k-1}. We have to consider two cases, $(a.I)$ all clusters in $\overrightarrow{\mathcal{H}}_{R^1}^S(e_f^r) \cup \overrightarrow{\mathcal{H}}_{R^{k-1}}^S(e_l^r)$ completely

contain $X_{R^1 \cup R^{k-1}}$, or $(a.II)$ at least one cluster exists that does not.

In Case $(a.I)$, the set of clusters $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ and $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ are incompatible since only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ contain $\varphi_C(\beta(e_f))$, clusters in both sets contain $X_{R^1 \cup R^{k-1}}$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e_l))$.

In Case $(a.II)$ any cluster c' in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ or $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ not completely containing $X_{R^1 \cup R^{k-1}}$ is in the same netted component as ∂H^1 and ∂H^{k-1}.

Any cluster in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ completely containing $X_{R^1 \cup R^{k-1}}$ is incompatible with any cluster in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ because only clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e_l))$. Any cluster completely containing $X_{R^1 \cup R^{k-1}}$ shares at last one $X_{r'}$, with $r' \in R^1 \cup R^{k-1}$, with any cluster in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ contain $\varphi_C(\beta(e_f))$ and vice versa. Consequently, all clusters are contained in the same netted component.

Case (b):

Let (R^1, H^1) and (R^{k-1}, H^{k-1}) be defined as in Case (a). Again, since N is minimal, at least one reticulation r_1 in R^1 and a cluster c_f in $\mathcal{H}^S(e^r_f)$ exists such that $X_{r_1} \subsetneq c_f$. Let $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ be the set of all those clusters in $\mathcal{H}^S(e^r_f)$ that contain at least one X_{r_1} with $r_1 \in R^1$. Using a symmetric argument as in the Case (a) and again the minimality of N, at least one cluster c_l in $\mathcal{H}^S(e^r_l)$ and a r_{k-1} in R^{k-1} exist such that $X_{r_{k-1}} \not\subset c_l$. We define $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ as the set of all those clusters in $\mathcal{H}^S(e^r_l)$ that are disjoint to at least one $X_{r_{k-1}}$ with $r_{k-1} \in R^{k-1}$.

Again, we will show that the sets of clusters $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ and $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ are contained in the same netted component. Note that the remarks about ∂H^1 and ∂H^{k-1}, as well as the clusters $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ and $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ given in Case (a), also apply to this case. We have to consider two cases, (I) all clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ completely contain and all clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ are disjoint to $X_{R^1 \cup R^{k-1}}$, or (II) or at least one cluster exists for which Case (I) does not apply.

In Case (I) both sets of clusters are incompatible since only the clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ contain $X_{R^1 \cup R^{k-1}}$, all clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f) \cup \overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e^r_f))$ and only clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e^r_l)) \cup X_{R(e^r_l) \setminus R(e^r_{l-1})}$. Note that either $\varphi_C(\beta(e^r_l))$ is non-empty, or by Lemma 3.2, $X_{R(e^r_l) \setminus R(e^r_{l-1})}$ is non-empty for all clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$.

In Case (II), any cluster c' not completely containing or missing $X_{R^1 \cup R^{k-1}}$ is in the same netted component as ∂H^1 and ∂H^{k-1}. All clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ completely missing $X_{R^1 \cup R^{k-1}}$ are incompatible with all cluster in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$, because only clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e^r_l)) \cup X_{R(e^r_l) \setminus R(e^r_{l-1})}$, clusters of both sets contain $\varphi_C(\beta(e^r_f))$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ contain

elements from $X_{R^1 \cup R^{k-1}}$.

Symmetrically, all clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ completely containing $X_{R^1 \cup R^{k-1}}$ are incompatible with all cluster in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$, because only clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_f)$ completely contain $X_{R^1 \cup R^{k-1}}$. Clusters of both sets contain $\varphi_C(\beta(e^r_f))$ and only clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e^r_l)) \cup X_{R(e^r_l) \setminus R(e^r_{l-1})}$. Consequently, all clusters are contained in the same netted component.

Note that if one of the edges e^r_1, e^r_{k-1} is not be degenerated, this proof can be easily altered. For example, if e^r_{k-1} is not degenerated, we set (R^{k-1}, H^{k-1}) to equal $(\{r\}, \emptyset)$ and the argument given above still holds. Note that in this case, $R^1 \cup R^{k-1} = R^1$, since r is an element of R^{k-1}, and the argument about $(\{r\}, \emptyset)$ being closed can be skipped. If both edges are not degenerated, the set $R^1 \cup R^{k-1}$ becomes $\{r\}$ making case (II) and the argument about the closeness of the pairs redundant.

Second step:

Assuming Case (a):

For any edge e^r_i in C with $f < i <= o$, we define the set $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ as consisting of all those clusters that contain at least one $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$. We now show that these clusters are contained in the same netted component as $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ and $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$. Consequently, we have to show this for all cases given in the first step. Thus, we have to consider Cases $(a.I)$ and $(a.II)$ from the first step.

If we have Case $(a.I)$ the set of clusters $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ is incompatible with $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ since only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ contain $\varphi_C(\beta(e^r_f))$. All clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ completely contain $X_{R^1 \cup R^{k-1}}$ and all clusters in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ contain at least one $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e_l))$.

Any cluster in $\overrightarrow{\mathcal{H}}^S(e_i) \setminus \mathcal{H}^S_{R^1 \cup R^{k-1}}(e^r_i)$ is incompatible with $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ since only clusters in $\mathcal{H}(e^r_i) \setminus \overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ contain $\varphi_C(\beta(e^r_i)) \cup X_{R(e^r_i) \setminus R(e^r_{i-1})}$, the clusters from both sets contain $\varphi_C(\beta(e^r_f))$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$ contain an $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$.

If we have Case $(a.II)$, any cluster not completely containing or missing $X_{R^1 \cup R^{k-1}}$ is contained in the same netted component as ∂H^1 and ∂H^{k-1}. Any cluster in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ completely containing $X_{R^1 \cup R^{k-1}}$ is incompatible with $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$, since only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ contain $\varphi_C(\beta(e^r_f))$. All clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain at least one $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$ and all clusters in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e^r_i)$ completely contain $X_{R^1 \cup R^{k-1}}$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e^r_l)$ contain $\varphi_C(\beta(e_l))$. All clusters in $\overrightarrow{\mathcal{H}}^S(e_i) \setminus \mathcal{H}^S_{R^1 \cup R^{k-1}}(e^r_i)$ that are completely disjoint to $X_{R^1 \cup R^{k-1}}$ are incompatible with the set $\overrightarrow{\mathcal{H}}^S_{R^1}(e^r_f)$, since

the arguments of the above case still hold.

For any edge e_i in C with $o < i < l$ we again define the set $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ to consist of all those clusters that contain at least one $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$.

If we have Case $(a.I)$, the set of clusters $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ is incompatible with $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e_f^r)$ since only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ contain $\varphi_C(\beta(e_l))$, all clusters in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ contain at least one $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$ and al clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e_f^r)$ completely contain $X_{R^1 \cup R^{k-1}}$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e_f^r)$ contain $\varphi_C(\beta(e_f))$. The set of clusters $\mathcal{H}^S(e_i^r) \setminus \overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ is incompatible with $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$ since only clusters in $\mathcal{H}^S(e_i^r) \setminus \overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ contain $\varphi_C(\beta(e_i^r)) \cup X_{R(e_i^r) \setminus R(e_{i+1}^r)}$, the clusters from both sets contain $\varphi_C(\beta(e_l))$, and only clusters in $\overrightarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$ contain $X_{R^1 \cup R^{k-1}}$.

If we have Case $(a.II)$, any cluster not completely containing or missing $X_{R^1 \cup R^{k-1}}$ is contained in the same netted component as ∂H^1 and ∂H^{k-1}. The other clusters are analogous to Case $(a.I)$.

Assuming Case (b):

For any edge e_i^r with $f < i < l$, we define the set $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ as that consisting of all those clusters that contain at least one $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$.

If we take Case $(b.I)$, this set is incompatible with $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$ since only clusters in $\overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ contain an $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$, the clusters in both sets contain $\varphi_C(\beta(e_f^r))$, and only clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$ contain $\varphi_C(\beta(e_l^r)) \cup X_{R(e_l) \setminus R(e_{l-1})}$.

On the other side, all clusters in $\mathcal{H}^S(e_i^r) \setminus \overrightarrow{\mathcal{H}}^S_{R^1 \cup R^{k-1}}(e_i^r)$ are incompatible with $\overrightarrow{\mathcal{H}}^S_{R^1}(e_f^r)$ as the same arguments as in Case $(a.I)$ hold.

If we take Case $(b.II)$, note that all clusters in $\mathcal{H}^S(e_i^r)$ that are not completely containing or missing $X_{R^1 \cup R^{k-1}}$ are contained in the same netted component as $\overrightarrow{\mathcal{H}}^S_{R^1}(e_f^r)$ and $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$. Any cluster in $\mathcal{H}^S(e_i^r)$ completely containing $X_{R^1 \cup R^{k-1}}$ is incompatible with $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$, since the first set completely contains $X_{R^1 \cup R^{k-1}}$ and all clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$ are disjoint to at least one $X_{r'}$ with $r' \in R^1 \cup R^{k-1}$, th clusters in both sets contain $\varphi_C(\beta(e_f))$, and only clusters in $\overleftarrow{\mathcal{H}}^S_{R^{k-1}}(e_l^r)$ contain $\varphi_C(\beta(e_l^r)) \cup X_{R(e_l) \setminus R(e_{l-1})}$.

Any cluster in $\mathcal{H}^S(e_i^r)$ that is disjoint to $X_{R^1 \cup R^{k-1}}$ is incompatible with $\mathcal{H}^S_{R^1}(e_f^r)$, since any cluster of the first kind contains $\varphi_C(\beta(e_i^r)) \cup X_{R(e_i) \setminus R(e_{i-1})}$. The clusters of both sets contain $\varphi_C(\beta(e_f))$, and all cluster in $\mathcal{H}^S_{R^1}(e_f^r)$ contain at least one $X_{r'}$ with $r' \in R^1$.

Third step:

We have to show that for any edge e that (1) is contained in a cycle in N and (2) for which it holds that for any reticulation $r' \in R(e)$ the edge

e is contained in $\{e_f^{r'}, e_l^{r'}\}$, the set of clusters sampled from this edge are contained in the same netted component.

We first denote $R(\beta(e))$ as that containing those reticulations r' in $R(e)$ for which $\beta(e) \in \{w_f^{r'}, w_{l+1}^{r'}\}$ and analogously define $R(\alpha(e))$. Furthermore, define the set $\mathcal{R}(\beta(e))$ as that consisting of the following sets of reticulations:

- $R' = \{r'\}$, where $r' \in R(e)$ is a direct descendant of $\beta(e)$;
- $R' \subseteq R(e^d)$, where e^d is a degenerated edge directly descending from $\beta(e)$, and a reticulation $r' \in R(e)$ exists such that R' is the maximal proper closure of r' in $(R(e^d), \mathcal{H}(e^d))$.

There are two cases to consider: (I) either $R(\beta(e))$ or $R(\alpha(e))$ is empty, and (II) $R(\beta(e))$ and $R(\alpha(e))$ are non-empty.

Assume Case (I):

First assume $R(\alpha(e))$ to be empty. Note that $R(e) \subseteq \cup_{R' \in \mathcal{R}(\beta(e))} R'$ and the sets in $\mathcal{R}(\beta(e))$ reflect those sets that are discussed in the first step of the proof.

If $R(e) \subsetneq \cup_{R' \in \mathcal{R}(\beta(e))} R'$, a $R' \in \mathcal{R}(\beta(e))$ exists such that $R' \cap R(e) \neq R'$. Furthermore, a $r' \in R'$ exists that is not an element of $R(e)$. Consequently, all $c \in \mathcal{H}^S(e)$ must contain $X_{r'}$. From the definition of $\mathcal{R}(\beta(e))$, follows that there is an $r'' \in R'$ with $r'' \in R(e)$. Considering the tree cycle of r'', it follows from the definition of $\mathcal{R}(\beta(e))$ that $e \in \{e_f^{r''}, e_l^{r''}\}$. We choose $e_f^{r''} = e$ without loss of generality and recall from step one that the set of clusters $\overrightarrow{\mathcal{H}}_{R'}^S(e_f^{r''})$ is contained in the same netted component. Since $\overrightarrow{\mathcal{H}}_{R'}^S(e_f^{r''}) = \mathcal{H}^S(e)$, the proposition follows.

Now let $R(e) = \cup_{R' \in \mathcal{R}(\beta(e))} R'$. Any cluster $c \in \mathcal{H}^S(e)$ must contain at least one $X_{r'}$ with $r' \in R(e)$, otherwise it will be sampled outside of all cycles. For any pair of clusters $c, c' \in \mathcal{H}^S(e)$, either a pair $r', r'' \in R'$ exists (note that $r' = r''$ is possible) with $R' \in \mathcal{R}(\beta(e))$, such that $X_{r'} \subsetneq c$ and $X_{r''} \subsetneq c'$ or there does not exists such a pair. If the pair r', r'' exists, we may select any $r' \in R'$ and consider the tree cycle of r'. Again, it follows from the definition of $\mathcal{R}(\beta(e))$ that $e \in \{e_f^{r'}, e_l^{r'}\}$. We choose $e_f^{r'} = e$ without loss of generality and recall from step one that all clusters in $\overrightarrow{\mathcal{H}}_{R'}^S(e_f^{r'})$ are contained in the same netted component and consequently c and c' are as well. If no such set R' exists both clusters are incompatible with each other, since $r' \neq r''$ and both clusters contain $\varphi_C(\beta(e))$.

Now assume $R(\beta(e))$ to be empty and note that in this case we have $\alpha(e) = w_o^{r'}$ for all $r' \in R(e)$. We define $\mathcal{R}(\alpha(e))$ analogously to $\mathcal{R}(\beta(e))$ and again $R(e) \subseteq \cup_{R' \in \mathcal{R}(\alpha(e))} R'$. If $R(e) \subsetneq \cup_{R' \in \mathcal{R}(\alpha(e))} R'$, a $R' \in \mathcal{R}(\alpha(e))$ exists such that $R' \cap R(e) \neq R'$. Furthermore, a $r' \in R'$ exists that is not an element of $R(e)$ and consequently, all $c \in \mathcal{H}(e)$ must be disjoint to $X_{r'}$. From the definition of $\mathcal{R}(\alpha(e))$, follows that there is an $r'' \in R'$ with $r'' \in R(e)$. Considering the tree cycle of r'', it follows from the definition of $\mathcal{R}(\alpha(e))$ that $e \in \{e_f^{r''}, e_l^{r''}\}$. We choose $e_l^{r''} = e$ without loss of generality and recall from

step one that the set of clusters $\overleftarrow{\mathcal{H}}^{S}_{R'}(e_l^{r''})$ is contained in the same netted component. Since $\overleftarrow{\mathcal{H}}^{S}_{R'}(e_l^{r''}) = \mathcal{H}(e)$, the proposition follows.

Now let $R(e) = \cup_{R' \in \mathcal{R}(\alpha(e))} R'$. Any cluster $c \in \mathcal{H}^S(e)$ must be disjoint to at least one $X_{r'}$, where $r' \in R(e)$, otherwise the cluster would be sampled outside of all cycles. Again, for any pair of clusters $c, c' \in \mathcal{H}^S(e)$, we have that either pair $r', r'' \in R'$ exists with $R' \in \mathcal{R}(\alpha(e))$ such that $X_{r'} \not\subseteq c$ and $X_{r''} \not\subseteq c'$, or we do not. If the pair r', r'' exists, we may select any $r' \in R'$ and consider the tree cycle of r'. It follows from the definition of $\mathcal{R}(\alpha(e))$ that $e \in \{e_f^{r'}, e_l^{r'}\}$. We choose $e_l^{r'} = e$ without loss of generality and recall from step one that all clusters in $\overleftarrow{\mathcal{H}}^{S}_{R'}(e_l^{r'})$ are contained in the same netted component and consequently c and c' as well. If these two reticulations do not exist, both clusters are incompatible with each other, since $r' \neq r''$ and both clusters contain $\varphi_C(\beta(e_f^{r'}))$.

Assuming Case (II):

We define two sets of clusters: the set $H(\beta(e))$, for which each element contains at least one $X_{r'}$ with $r' \in R(\beta(e))$; and the set $H(\alpha(e))$, for which each element misses at least one $X_{r'}$ with $r' \in R(\alpha(e))$. It is straightforward to see from Case (I) that the clusters in $H(\beta(e))$ are contained in one netted component and the clusters in $H(\alpha(e))$ are also contained in one netted component. We have to show that the elements of these two sets and the cluster that contains $X_{R(\alpha(e))}$ and excludes $X_{R(\beta(e))}$ are contained in the same netted component. If $H(\alpha(e))$ and $H(\beta(e))$ are not contained in the same netted component, they must be disjoint. This can only be accomplished if for any cluster c in $H(\beta(e))$ it holds that any cluster c' in $H(\alpha(e))$ is contained in c. But than the network would not be minimal, i.e. we could add a new vertex in the middle of e and reconnect the directly descending edges of $\alpha(e)$ and $\beta(e)$ associated with $R(\alpha(e))$ and $R(\beta(e))$ to this vertex, making both sets of reticulations independent. Consequently, at least one cluster c^* must exist that violates this rule, that is c^* is an element of $H(\alpha(e))$ and $H(\beta(e))$. Consequently, the cluster is excluding at least one $X_{r'}$ with $r' \in R(\alpha(e))$. As the cluster contains at least one $X_{r''}$ with $r'' \in R(\beta(e))$ it is incompatible with the cluster that contains $X_{R(\alpha(e))}$ and excludes $X_{R(\beta(e))}$.

Fourth step:

In this step, we have to prove that all clusters sampled from a degenerated edge are contained in the same netted component. Thus, let e be a degenerated edge in N. Furthermore, let r be a reticulation in $R(e)$ and consequently, $E_r^* \neq \emptyset$. We will show that the set of clusters $\mathcal{H}^S(E_r^*)$ is contained in the same netted component as the set of clusters in $\mathcal{H}^S(E_r^T)$. First, remember that according to Lemma 3.7, for any set (R', H') that is non-decomposable and closed with respect to $(R(E_r^*), \mathcal{H}^S(E_r^*))$ at least one cluster c^T in $\mathcal{H}^S(E_r^T)$ exists that separates $X_{R'}$. From Lemma 3.8, we know that $X_{R'}$ and c^T are incompatible. Furthermore, from Lemma 3.9, we know that if $\partial H' \backslash \{X_{R'}\}$ is non-empty, at least one cluster c' in $\partial H' \backslash \{X_{R'}\}$ exists that is incompatible with c^T. Finally, Lemma 3.5 ensures, that all clusters in $\partial H' \backslash \{X_{R'}\}$ are contained in the same netted component. Thus for any set (R', H') that

is non-decomposable and closed with respect to $(R(E_r^*), \mathcal{H}^S(E_r^*))$ the set of clusters $\partial H'$ is contained in the same netted component as $\mathcal{H}(E_r^T)$.

We have to show that for any cluster c in $\mathcal{H}^S(E_r^*)$ a set (R', H') exists that is non-decomposable and closed with respect to $(R(E_r^*), \mathcal{H}^S(E_r^*))$ such that c is contained in $\partial H'$. For any set(R', H') that is closed with respect to $(R(E_r^*), \mathcal{H}^S(E_r^*))$ and for which $\partial H'$ is non-empty the set $\partial H'$ is contained in the same netted component as $\mathcal{H}(E_r^T)$. If we denote $((H_1, R_1), \ldots, (H_k, R_k))$ as the set of all closed sets of $(R(E_r^*), \mathcal{H}^S(E_r^*))$ with the property that $R_i \subsetneq R'$, then by definition, $H' = \partial H' \cup \bigcup_i H_i$. For any set (R_i, H_i), this argument holds as well, and since $(R(E_r^*), \mathcal{H}^S(E_r^*))$ is closed with respect to itself, for any cluster c in $\mathcal{H}^S(E_r^*)$ a set (R', H') non-decomposable and closed with respect to $(R(E_r^*), \mathcal{H}^S(E_r^*))$ exists such that c is contained in $\partial H'$. □

Chapter 4

Calculating Galled Networks

In this chapter, we give a detailed introduction to our solution to the problem of constructing galled networks from a set of splits, and we show how converting the underlying (sequence) information into splits on the one hand simplifies the reconstruction but, on the other hand raises new questions.

A split network can be seen as a direct way to represent a set of binary classifications of a group of taxa. The information stored within this split network is only a subset of the information stored in a set of trees or an alignment, even if we use all splits within the set of trees or all splits that can be generated from the alignment. For example, given a set of splits we can not decide which split was generated by which tree; this information is lost in the conversion. Therefore, ensuring that the information we lose is not essential for the calculation of an explicit network is important. The general problem one would like to solve can be stated as:

Problem 4.1 Parsimonious non-degenerated Galled Network from Split Set Problem: *Given a set of splits Σ, construct a minimal non-degenerated galled network N that represents Σ.*

From the definition of a non-degenerated galled network, we know that a backbone tree exists to which the reticulations can be connected. We propose an algorithm which applies this property by identifying those minimal sets $R \subset X$ with which $\Sigma_{|X\setminus R}$ is compatible. If the set is compatible, the algorithm can use it as a possible split encoding for the backbone tree. The backbone tree is accepted by the algorithm, if for all reticulations $r \in R$, the incompatible splits in $\Sigma_{|X\setminus(R\setminus\{r\})}$ form a path within it. If they do, the algorithm connects each reticulation to the start and end edge of the path. In detail, the algorithm works as follows:

Algorithm 4.1 (Construct minimal galled network from splits)
Input: Set of splits Σ on X
Output: Minimal galled network N that explains Σ, if one exists, or fail.

for $k = lowerbound, \ldots, upperbound$ **do**
  for each *possible choice of a subset $R \subset X$ of cardinality k* **do**
    if $\Sigma' := \Sigma|_{X\setminus R}$ *is compatible* **then**
      Build a rooted backbone *tree T that represents Σ'*

for each *putative reticulation vertex* $r \in R$ **do**
Let E *be the set of all edges in the backbone tree for which there exists an incompatible split in* $\Sigma_{|X\setminus(R\setminus\{r\})}$
if E *is contained in a path in* T **then**
attach r *in the middle of the start- and end-edge of such a shortest path*
else *we can not attach all vertices of* R*, try next choice of* R
if *all* $r \in R$ *were successfully attached and the resulting network* N *represents* Σ **then**
return N
return *fail.*

Whenever the algorithm succeeds, we have a solution to the "Parsimonious Non-degenerated galled network from Split Set Problem". Because of the minimality of the non-degenerated galled network, all reticulations are connected to one vertex in the backbone tree (the one added in the middle of an edge), and there might even be more than one reticulation connected to such a vertex. If we would like to resolve these connecting vertices further, the resulting galled network will no longer be minimal. Interestingly, the number of reticulations does not change and those the relaxed definition of minimality given by [GEL03] still applies. Refining a galled network is indeed an interesting topic and we will discuss it in detail in the next section.

Refining the connections of the reticulations is a post-modification of the resulting minimal galled network. The subsequent application of several modifications may be desirable. Therefore, we see a *complete reconstruction* as a set of steps, some of which are mandatory and some of which are not. In detail, the complete reconstruction can be divided into the following steps:

4.1.a) **Bound the number of reticulations:**
Find a lower and upper bound for the number of reticulations.

4.1.b) **Reconstruct a minimal galled network:**
Use Algorithm 4.1 to reconstruct a minimal galled network.

4.1.c) **Apply additional constrains or modifications (optional):**
Check if additional constrains hold, given the solution, or apply modifications to the minimal galled network found.

Finding lower and upper bounds for reticulate networks has been described elsewhere (for example, see [BB04; LSH05; MG03]) and an algorithm solving the second step has been presented above. In the next section, we show a post-modification, that may be applied to minimal galled networks.

Resolving the Connecting Vertices

We have already discussed the loss of information that is connected with transforming sets of trees or alignments into splits. In this section, we show that as a consequence of the transformation, the resolution of the connecting vertices is ambiguous, for a set of splits, and that we need additional information to solve this problem in a meaningful way.

Since the reconstruction of a minimal galled network ensures that the tree cycles of all reticulations will be maximal, resolving the connecting vertices will lead to a placement of the connectors for any reticulation, on the first and last tree edge of the associated tree cycle. So we actually resolve tree edges and not tree vertices. Let $R(e)$ denote the set of all reticulations connected an edge e of the backbone tree. In Figure 4.1, we show an example of two reticulations connected to an edge in the backbone tree. Each of the individual diagram represents one of the possible combinations. The splits that can be generated from this edge are $\left\{\frac{A}{\{r_1,r_2\}\cup B}, \frac{A\cup\{r_1\}}{B\cup\{r_2\}}, \frac{A\cup\{r_2\}}{B\cup\{r_1\}}, \frac{A\cup\{r_1,r_2\}}{B}\right\}$ for Figures $(a), (b), (d)$and (e); and $\left\{\frac{A}{B\cup\{r_1,r_2\}}, \frac{A\cup\{r_1\}}{B\cup\{r_2\}}, \frac{A\cup\{r_1,r_2\}}{B}\right\}$ for Figures (e) and (f). Since none of the split sets is unique, the problem cannot be solved unambiguously for two reticulations connecting to an edge of the backbone tree and consequently, cannot be solved in general given a set of splits.

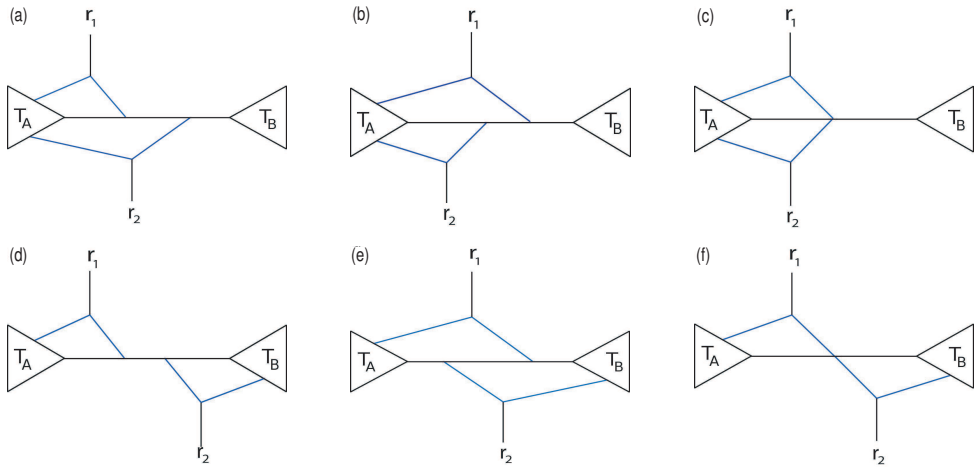


Figure 4.1: **The Placement of Connectors is Ambiguous** An edge in a backbone tree is indicated by the horizontal line, and the two subtrees are labeled T_A and T_B. Two reticulations, r_1 and r_2 connect to this edge. The second reticulation edge of r_1 is always connected to the subtree T_A. The second reticulation edge of r_2 is either connected to the subtree T_A (upper part of the figure) or to the subtree T_B (lower part of the figure). Either r_1 is closer to T_A than r_2 ((a), (d)) or r_2 is closer to T_A than r_1 ((b), (e)) or both are equally distant from T_A ((e), (f)).

Facing this ambiguity, the question arises how to resolve a minimal galled network in a meaningful way. Assume the splits were generated from a set of trees $\mathcal{T} = \{T_1, \ldots, T_n\}$. Some of these trees may be refinements of the backbone tree in the way that some edges of the backbone tree are (partially) resolved in these. Thus, these trees would give rise to an ordering of the reticulations associated with such edges and consequently the ordering in which we can resolve these edges, leading to a refinement of the minimal galled network.

The general idea behind this approach is to define an *evolutionary block* as a set of splits Σ_{EB} that is a subset of all splits Σ, for which it is assumed that the splits within this evolutionary block form one consistent evolutionary event. For example, the splits in each tree of $\mathcal{T}$ can be seen as such an

evolutionary block. For each tree edge and each evolutionary block Σ_{EB}, we may define a partial ordering. Let $Ord_P(e)$ be the set of all partial orderings we obtained from a set of evolutionary blocks with respect to an edge e of the backbone tree. It is straight forward to construct a directed graph from the set of all these partial orderings. We call this graph the *ordering graph* $OG(e)$.

For simplicity, we add two artifical vertices to the sorting processes, representing the source $\alpha(e)$ and sink $\beta(e)$ of the edge. Consequently, $\alpha(e)$ represents the smallest and $\beta(e)$ the largest element with respect to all orderings in the graph. To find a solution, we first have to collapse any cycles within the ordering graph to one vertex, since these represent conflicts within the ordering. Using a Depth First Search, we obtain an ordering of all vertices of $OG(e)$ (see [THCS01], Chapter 6 for details on topological sorting). With this ordering, we can refine the edge e of the minimal galled network and by applying this ordering to all tree edges in the reticulate network, we are able to fully refine it.

In the simplest case, we use all splits as one evolutionary block. This might lead to orderings for connecting vertices, but we would like to point out that there is no certainty about these orderings, since the ordering may be caused by lack of additional information. In the latter, we show how evolutionary blocks can be defined for hybridization and recombination networks, and how these can be used to obtain partial orderings for the ordering graph.

4.1 Hybridization Networks

A hybridization network is a special kind of reticulate network, that is aimed at explaining a set of (partial gene) trees $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ in terms of hybridization. More precisely, a reticulate network N is the hybridization network of $\mathcal{T}$, if $\mathcal{T}$ can be sampled from N.

As explained in Section 2.3, a set of gene trees $\mathcal{T}$ can be combined into one set of splits, using either the Consensus method if all trees in $\mathcal{T}$ contain the same set of labels, and the Z-closure method otherwise. Since Algorithm 4.1 is independent of the method that generates the set of splits, it is also possible to use a different method. The generated set of splits can then be used as input for the calculation.

Next we explain how to obtain partial orderings for a connecting edge from a set of $X-$trees $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. Assume that N is a minimal galled network such that the set of splits $\Sigma(\mathcal{T})$ is represented by N. For each $X-$tree T in $\mathcal{T}$, $T_{|X\setminus R}$ is equal to the backbone tree BT. For each tree edge e in the backbone tree BT, a path in the tree T exists that gives a (partial) ordering of the reticulations that connect to e. Thus, the trees $\mathcal{T}$ give rise to a set of orderings for e. We use this set as input for the ordering graph $OG(e)$. For example, in Figure 4.2, the partial orderings given for the backbone edge to which r_1 and r_2 connect are: $\alpha(e) < r_2 < r_1 < \beta(e)$ for tree (a), $\alpha(e) < \beta(e)$ for tree (b) and $\alpha(e) < r_1 < \beta(e)$ for tree (c).

Notably, if $\mathcal{T}$ corresponds to the complete set of trees that can be sampled

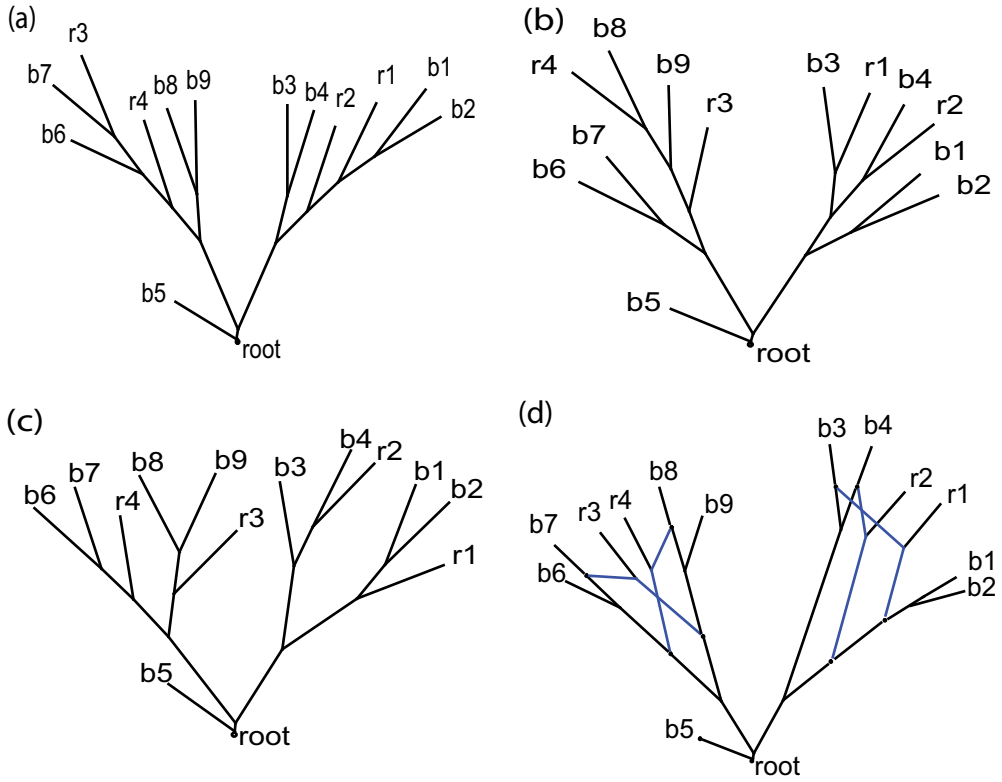


Figure 4.2: **Ordering Reticulations Along an Edge** Figures (a), (b) and (c) show the trees introduced in Section 2.3. The last figure shows the hybridization network that is generated by the consensus network containing all splits. The partial orderings given for the backbone edge to which r_1 and r_2 connect are: $\alpha(e) < r_2 < r_1 < \beta(e)$ for tree (a), $\alpha(e) < \beta(e)$ for tree (b) and $\alpha(e) < r_1 < \beta(e)$ for tree (c).

from a galled network N, the topological sorting applied to the orderings given by $\mathcal{T}$ will give rise to the correct ordering and consequently, will allow the algorithm to successfully reconstruct N.

Example for a Hybridization Network

For the first example, we obtained three gene trees relating different fungal species from TreeBASE [SDPE94], that were published in [PB03]. These trees are based on the *mitochondrial small subunit ribosomal DNA*, the *nuclear internal transcribed spacer* and on a part of the *glyceraldehyde-3-phosphate* gene. We constructed the set of all splits using the Z-closure method and obtained the network shown in Figure 4.3. We applied Algorithm 4.1 to the split network and obtained the hybridization network shown in Figure 4.4. We found a total of seven netted components. All netted components could be resolved to present a hybridization history. Even though our algorithm was able to calculate the history, the plausibility of the networks is somewhat unclear. The overall split network seemed to contain a lot of distortion and consequently, the result should be seen as a proof of concept.

The second example contains two gene trees of buttercups based on the chloroplast JSA region and the nuclear ITS gene[LMH$^+$01]. These two trees are known to contain reticulate events [HSW06]. In [HKLS05], we were able to reconstruct a hybridization network by removing problematic splits from the consensus network shown in Figure 4.5. Fortunately, a later article [HSW06] provided a method for reducing distortion in phylogenetic networks, which, when applied to this example, enables Algorithm 4.1 to provide a simple hybridization history, shown in Figure 4.6.

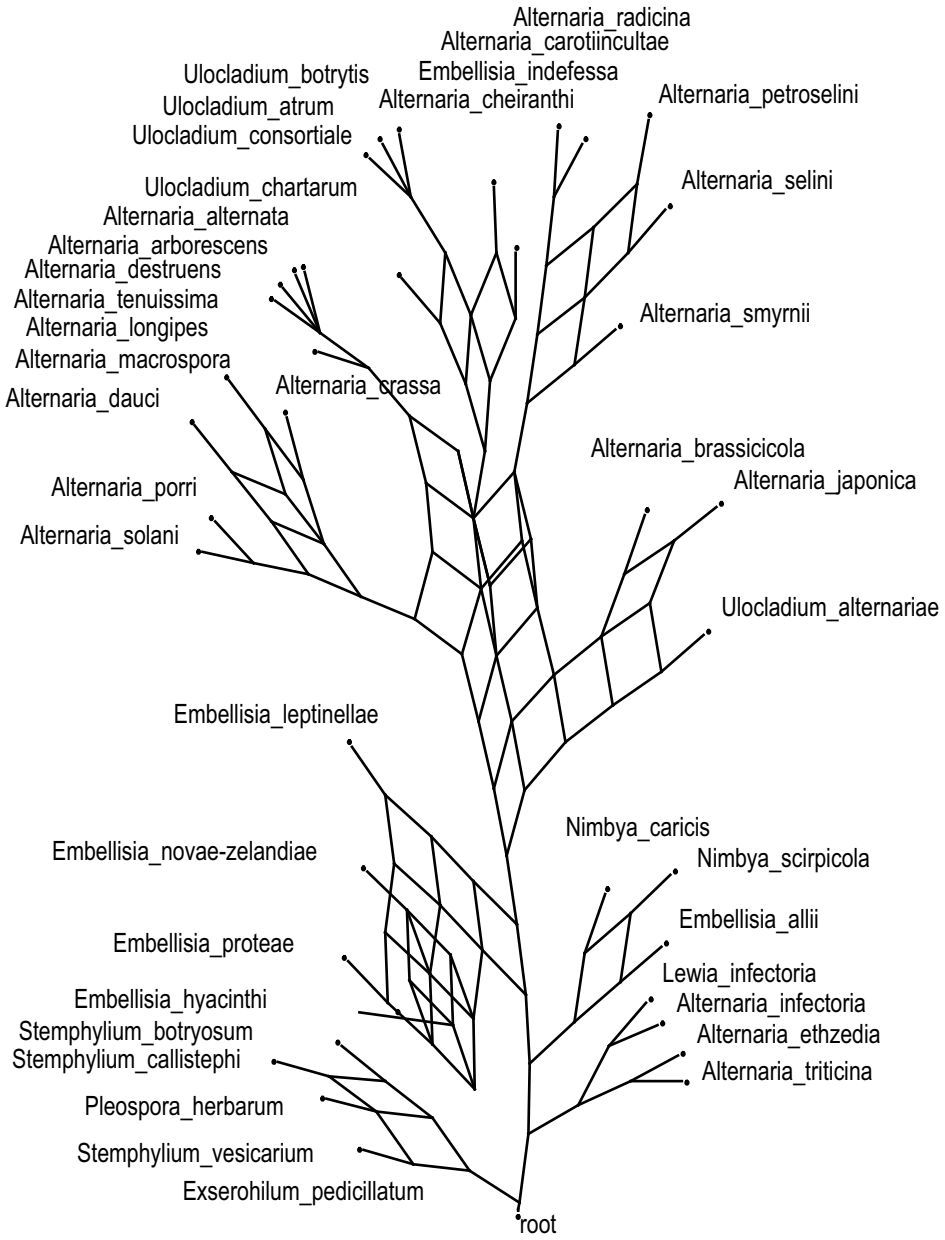


Figure 4.3: **Example of a hybridization network** The split network that represents all splits present in the three gene trees taken from [PB03].

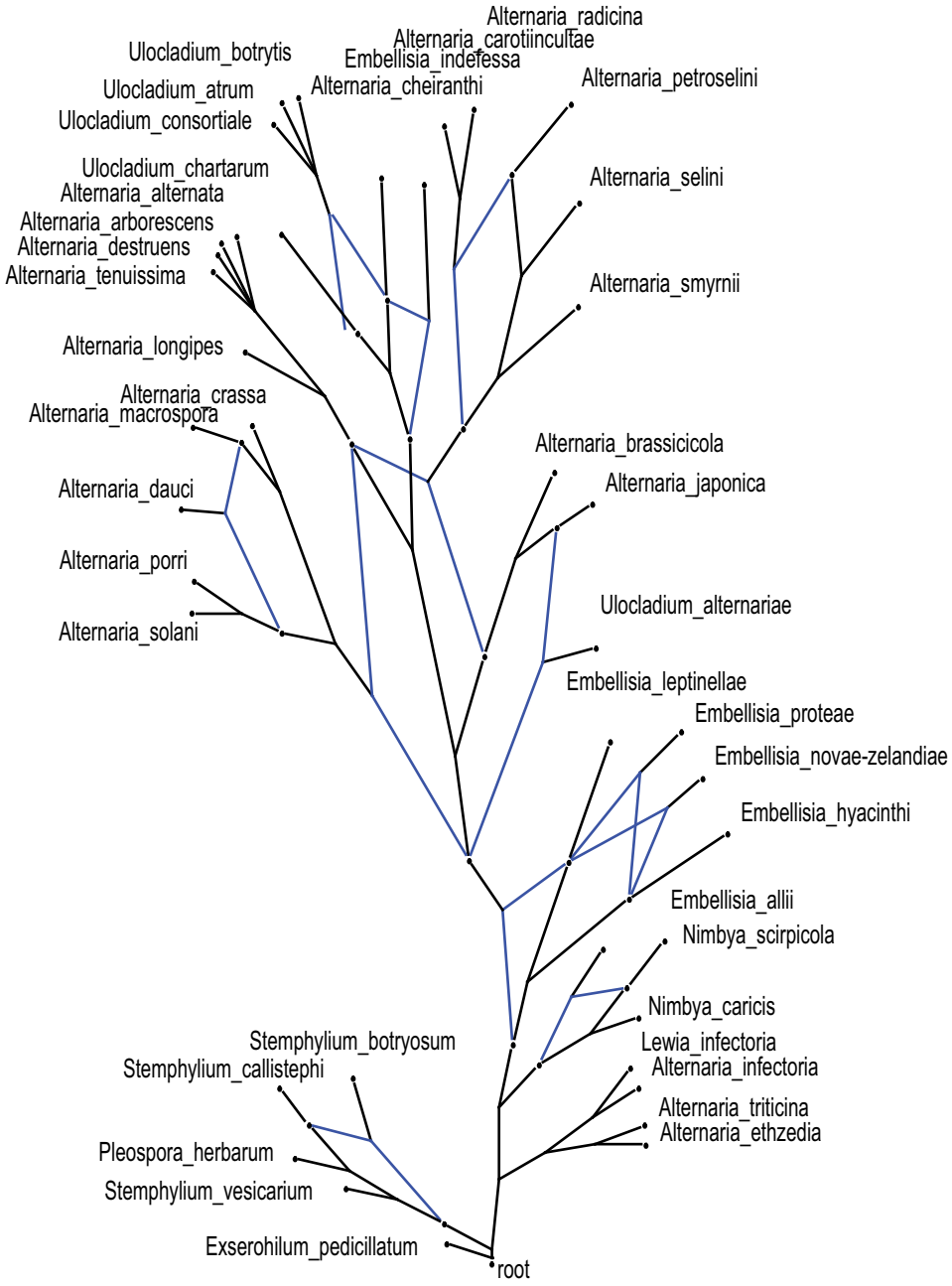


Figure 4.4: **Example of a hybridization network** Corresponds to the hybridization network calculated by our algorithm. The network is meant as a proof of concept and as an example of a reticulation network containing a number of netted components (the example contains seven), rather than a biological meaningful application.

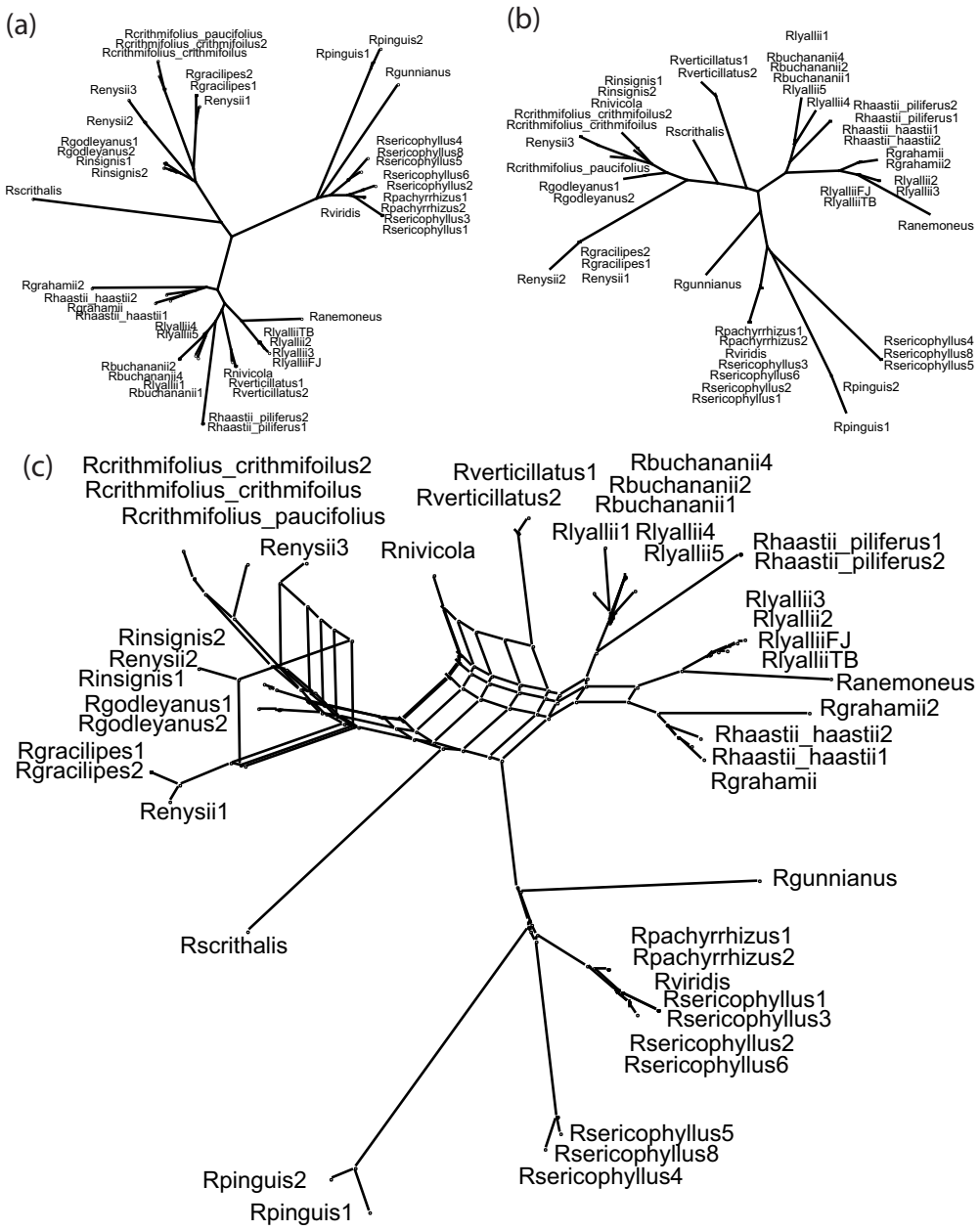


Figure 4.5: **Buttercup Gene Trees and Consensus Network** The top row shows two gene trees of 46 buttercups, based on the chloroplast ITS region ((a)) and the nuclear JSA gene ((b)) [LMH$^+$01]. Figure (c) shows the consensus network of all splits in the two gene trees.

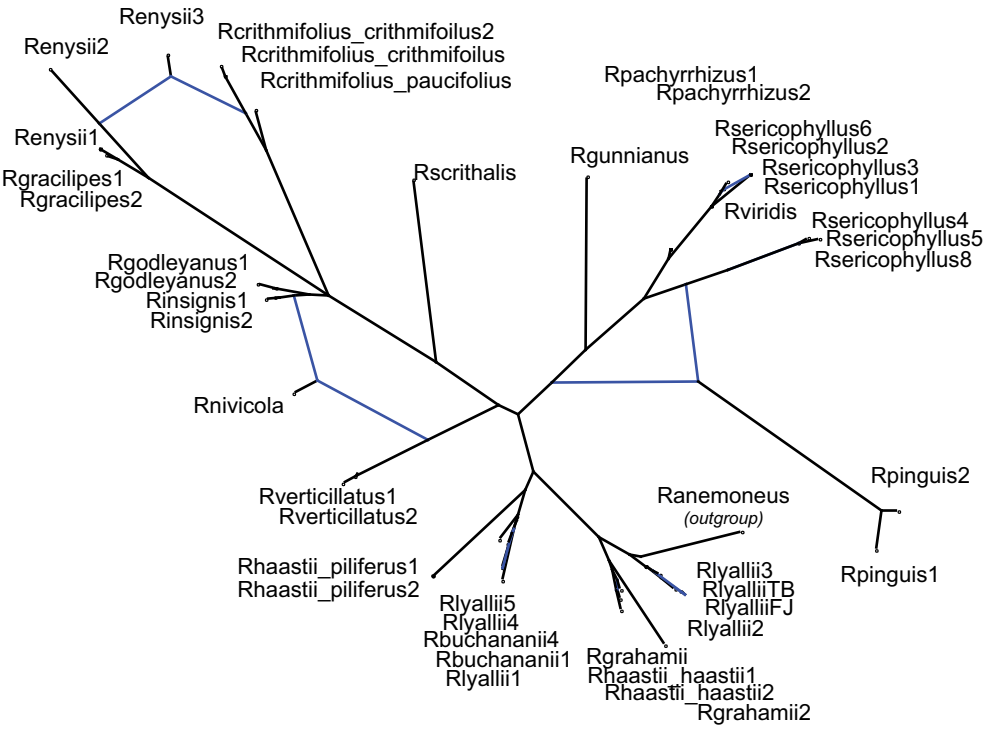


Figure 4.6: **Buttercup Hybridization Network** The hybridization network that can be obtained by first using the *FilterSupernetwork* method [HSW06], allowing each split to have, at most, one distortion in both trees; and secondly, applying Algorithm 4.1 to the resulting split network. This example has first been published in [HKLS05] without the FilteredSuperNetwork method. The result shows clearly that *R. Nivicola* is a hybrid arising between *R. insignis* and *R. verticallatus*, which is known in literature [HSW06]. Furthermore, the network indicates two further possible hybridization scenarios: *R. enysii3* as a hybrid arising between *R. enysii2* and *R. crithmifolius crithmifolius*, and *R. pinguis* which seems to have arisen from *R. secricophyllus* and some unknown other species.

4.2 Recombination Networks

In this section, we look at the problem of determining a recombination network that explains a given alignment A of binary sequences which have evolved under a model of mutation-, speciation- and recombination events. This problem is of interest in population genetics [GM96; Hei93; Hud83], were the concept of an *ancestor recombination graph* was introduced and studied, particularly from a statistical point of view. In the following, we make some simplifying assumptions:

1. all sequences have a common ancestor (which is not necessarily true in population studies),
2. any position in the alignment A can mutate at most once in the network (known as the "infinite sites" model), and optionally,
3. recombinations are always single crossovers.

Condition 1 is required such that the arising network has a single root vertex. Condition 2 allows us to interpret every non-constant column in a given alignment A of binary sequences as defining a split $S = \frac{A}{B}$ of the taxon set X, as discussed below. The third condition is not required for our approach, but an additional filter step can be used in our algorithm to return only those solutions that have this property. We define a recombination network as follows:

Definition 4.1 *Given an alignment A of binary sequences of length n, a* recombination network N that explains A *is a reticulation network, together with [DGL04]:*

1. *a labeling γ of all vertices by binary sequences of length n, such that the leaves of N are appropriately labeled by A,*
2. *a corresponding labeling δ of each tree edge e by those positions that change along e, and optionally*
3. *a corresponding labeling ρ of each reticulation vertex v_r indicating the crossover position for the recombination at v_r.*

In the approach described here, labeling a reticulation vertex v_r with a crossover position $\rho(v_r)$ is redundant, as we can easily infer the possible crossover positions from $\delta(p_r)$ and $\delta(q_r)$. However, if an additional method is being used to determine an optimal crossover position independently, then the labeling ρ can be of use.

Let $X = \{x_1, \ldots, x_n\}$ be a set of taxa and let

$$A = \begin{vmatrix} a_{11} & a_{12} & \ldots a_{1m} \\ a_{21} & a_{22} & \ldots a_{2m} \\ & & \ldots \\ a_{n1} & a_{n2} & \ldots a_{nm} \end{vmatrix}$$

be a corresponding alignment of binary sequences. Any non-constant column i in A gives rise to an X-split $\frac{\{x_j|a_{ji}=0\}}{\{x_j|a_{ji}=1\}}$. Let $\Sigma(A) = \{S_1, \ldots, S_k\}$, with $k \leq m$ and denote the set of all splits that can be obtained in this way. We generate a map ρ that maps each column i of A onto the corresponding split in $\Sigma(A)$. We say that the split network $SN(A) := SN(\Sigma(A))$ *represents* A.

Let N be a galled network that represents $SN(A)$. In the following, we describe how to find a solution for resolving the connecting vertices in N. For each edge e in the backbone tree BT of N, we denote $\rho^{-1}(e) = \cup_{s\in\Sigma(e)}\rho^{-1}(s)$ as the set of all positions in the alignment that support e. We choose those intervals $\mathcal{I}$ of $\rho^{-1}(e)$ that are maximal compatible as evolutionary blocks. For each interval $I \in \mathcal{I}$, the set of splits $\Sigma(I) = \cup_{i\in I}\rho(i)$ defines a split network $SN(\Sigma(I))$ that gives rise to a partial ordering of e. We use the set of all such partial orderings, obtained from $\mathcal{I}$ to build the ordering graph $OG(e)$ and to resolve the connecting vertices. Note that this method does not guarantee a complete ordering of each edge in the backbone tree.

After we have found a galled network N, we label the vertices and edges in a two-step approach. First, we define a labeling γ_{BT} of the vertices in the backbone tree by binary sequences and then define a labeling γ_N that will completely label the vertices of the galled network N.

Let $v \in T$ be the vertex corresponding to some fixed taxon $x_1 \in X$. We set $\gamma_T(v)$ as equal to the input sequence a_1 associated with x_1. Then, we traverse the backbone tree BT and do the following: let e be the edge that we cross in the traversal from vertex a to b. Assume that a has already been assigned a label $\gamma_{BT}(a) = w_1 \ldots w_m$ and that b has not. We obtain $\gamma_{BT}(b) = z_1 \ldots z_m$ by setting $z_i = 1 - w_i$ for each column i of A with $\rho(i) \in \Sigma(e)$, and $z_i = w_i$ otherwise.

To obtain the labeling γ_N, of the galled network N, we set the label $\gamma_N(v_r)$ of a reticulation vertex v_r to the input sequence associated with the corresponding taxon. We now need to label the connector vertices of N. For each edge e in the backbone tree BT with connectors $\alpha(e), w_1, \ldots, w_k, \beta(e)$ let W_j^C denote the set of reticulations connecting to vertex w_j. Let $\delta_{j,k}$ be equal to one, if for all vertices in W_j^C. the label at position k is the same and zero otherwise. We begin labeling the connectors by w_1. To obtain the label $\gamma_N(w_j) = u_1 \ldots u_m$ of the connector w_j let the label of w_{j-1} and $\beta(e)$ be given by $\gamma_N(v_{j-1}) = y_1 \ldots y_m$ and $\gamma_{BT}(\alpha(e)) = z_1 \ldots z_m$ respectively. If $y_i = z_i$, we set $u_i = y_i$, or if $z_i \neq y_i$ and $\delta_{j,k} = 1$, we set $u_i = z_i$; or we set $u_i = 2$.

The labeling of a connector vertex is not necessarily binary, since the ordering of the connecting vertices might not be completely resolved. But any ambiguous labeling is restricted to one edge of the backbone tree. Note that it is indeed possible to define a binary labeling of the connector vertex. However, the labeling may result in an increase of crossovers. The labeling for fully resolved reticulate networks is shown in [HK05].

For any edge e, we define $\delta(e)$ as the *sequence delta* between $\alpha(e)$ and $\beta(e)$, that is, the set of all positions in the labeling for which either $\alpha(e)$ and $\beta(e)$ are binary and differ, or for which $\alpha(e)$ and $\beta(e)$ differ and $\alpha(e)$ is

ambiguous. The labeling of the edges depends on the root ϱ that is given but again the dependency only affects one edge of the backbone tree at a time. The problem here again is that if we have an incomplete ordering of an edge e in the backbone tree, we might be unable to define exactly when a mutation actually happened. If this is so, we place the ambiguous states outside of the reticulation cycle [GEL03; HK05].

Assume that we are given an alignment A of binary sequences of length m, for a set of taxa $X = \{x_1, \ldots, x_n\}$. The following algorithm computes a recombination network N for A:

Algorithm 4.2 (Construct galled network from binary sequences)
Input: Alignment A of binary sequences
Output: Minimal galled network N that explains A, if one exists, or fail.

Compute $\Sigma(A)$ and the mapping σ of columns to splits.
Compute the split network $SN(A)$.
Compute a minimal galled network N.
if *N contains a solution* **then**
  Apply modifications.
  Compute the dissolving of the connecting vertices.
  Compute the vertex labeling γ for the backbone tree BT
  and extend it to the reticulate network N.
  For each edge e, set $\delta(e)$ to the sequence delta between $\alpha(e)$ and $\beta(e)$.
else *return* fail

Example of a Recombination Network

We obtained a set of binary characters from an analysis of a restriction map from the rDNA cistron [KBR98]. The analysis was performed on 16 species of the mosquito subfamily *Culicinae* together with the outgroup *Anopheles albimanus*. The analysis scored 26 sites of which 18 were polymorphic. In the original publication, this dataset was analyzed using a number of different tree-reconstruction methods with inconclusive results. We included this set in an earlier publication and were able to obtain a solution for a subset of the complete taxon set [HK05].

The complete split network is shown in Figure 4.7 (*b*). The resulting recombination network is shown in Figure 4.8. In Figure 4.8 (*a*), we show the recombination network and the labeling of those edges that are important to the recombinations. The algorithm was able to distinguish the emergence of a possible recombination of *Aedes triseriatus* from the emergence of *Armigeres subalbatus* and *Aedes triseriatus*. In Figure 4.8 (*b*), we show the labels of the vertices involved in the recombination scenarios.

(a)

Aedes albopictus	1111010101 0100010101 010010
Aedes aegypti	1111010100 0100010101 000010
Aedes seatoi	1111010101 0100010101 010000
Aedes flavopictus	1111010101 0100010101 010010
Aedes alcasidi	1111010101 0100010101 010000
Aedes katherinensis	1111010101 0100010101 010000
Aedes polynesiensis	1111010100 0100010101 010010
Aedes triseriatus	1011010100 0110010101 000000
Aedes atropalpus	1011010100 0100010111 000010
Aedes epactius	1011010100 0100010111 000010
Haemagogus equinus	1011010100 0110010101 010000
Armigeres subalbatus	1011010100 0100010101 000000
Culex pipiens	1111011100 0100011101 001011
Tripteroides bambusa	1111011100 0100010101 000010
Sabethes cyaneus	1111010100 1100010101 010000
Anopheles albimanus	1101110110 0101110101 110100

(b)

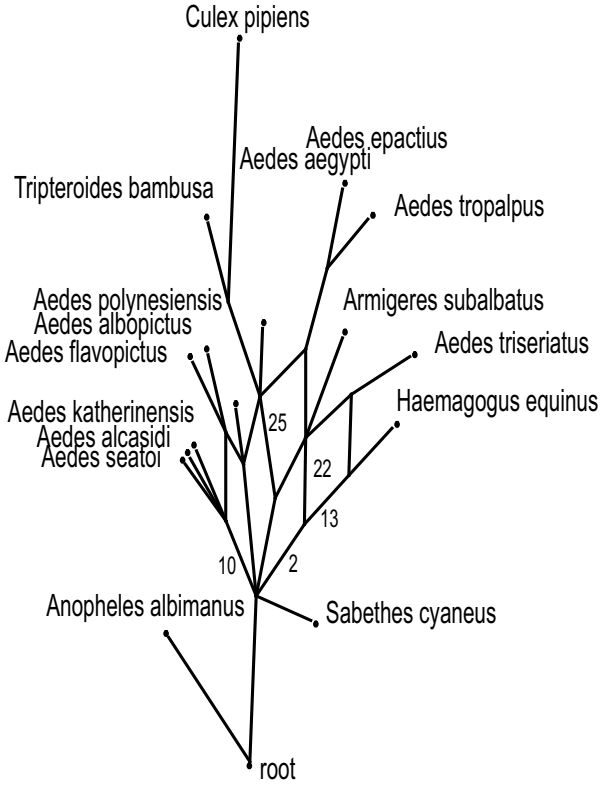


Figure 4.7: **Binary Alignment and Split Network** In (a), we show an alignment of binary sequences obtained from the restriction maps of the rDNA cistron (length ≈ 10 kb) of twelve species of mosquitoes using eight 6 bp recognition restriction enzymes [KBR98]. In (b), the split network computed from the data in (a) is shown. Edges of incompatible splits are labeled by the corresponding mutation positions.

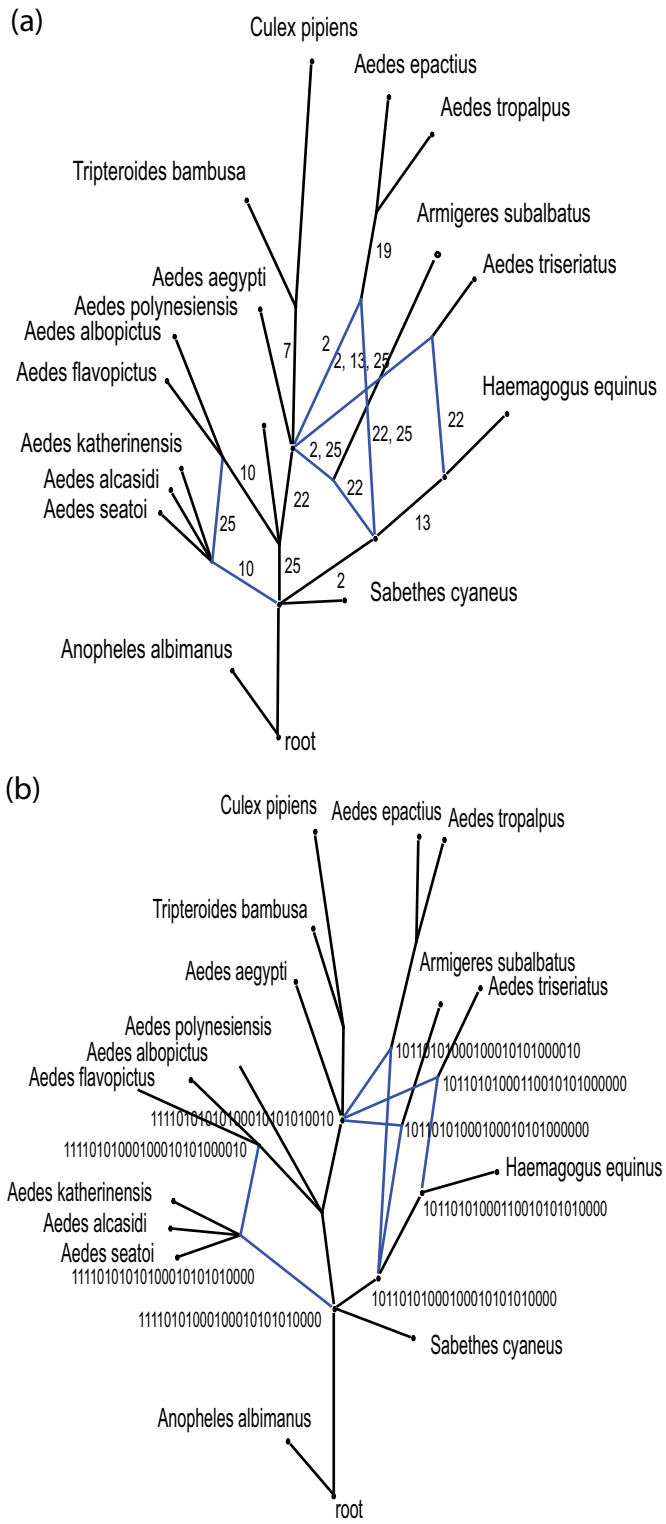


Figure 4.8: **Example of a Recombination Network** In (a), we show the recombination network calculated from the data shown in Figure 4.7. The edges involved in a possible recombination are labeled with their corresponding mutations in the alignment. In (b), we labeled the vertices involved in the possible recombinations with their corresponding sequences.

The second example is taken from a study of gene histories of *Fusarium gaminearum*, the fungus causing wheat scab [KOC00]. The analysis contained 28 sequences of the $TRI101$ protein with a length of 1336 bp. The sequences cluster into seven geographical regions. Prior phylogenetic reconstruction as well as parsimony analysis showed strong evidence for intra-genic recombination within strain 28721. We generated a binary representation of the sequences using a standard encoding method and then applied our method to the resulting binary sequences. The resulting recombination network is shown in Figure 4.9 (b), showing strain 28721 as a recombination of the $Asian$ (strains 13818, 6101, 26156 and 28720) and $African$ (strains 29010, 28436 and 28723) clusters.

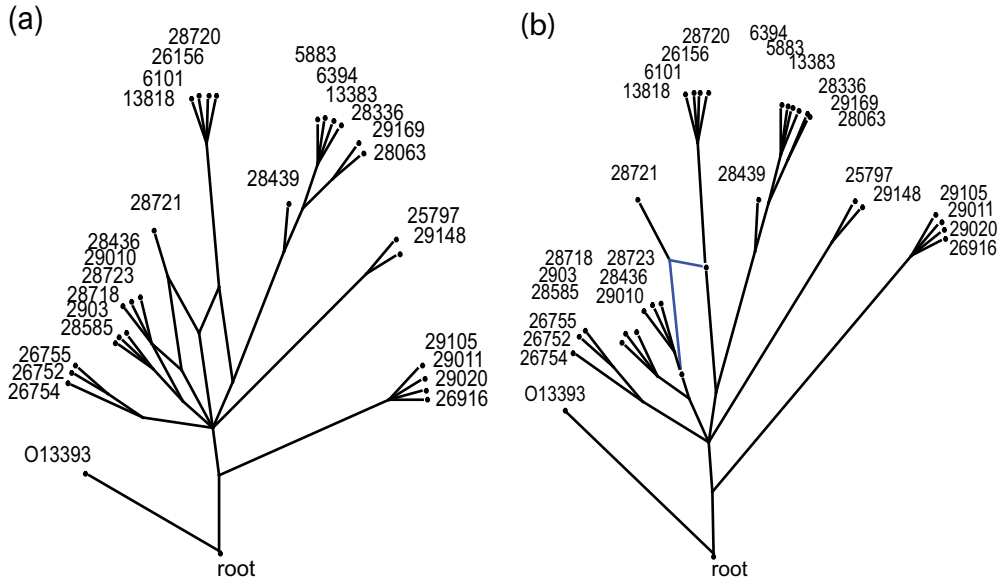


Figure 4.9: **Example of a Recombination Network** In (a), we show the split network of 28 strains of the $TRI101$ gene of *Fusarium gaminearum*. In (b), we show the reconstructed recombination network with strain 28721 as a recombination of the $Asian$ and $African$ clusters.

Chapter 5

Drawing Reticulate Networks

Phylogenetic networks are graphs representing phylogenetic relationships between different taxa, and are usually employed when a tree is not an adequate representation of the data.

Because of the complex nature of phylogenetic networks, not only their calculation but also their visualization is a challenging problem. In general, the visualization methods for phylogenetic trees are based upon the hierarchical structure of the trees; unfortunately, phylogenetic networks do not feature this structure. Furthermore, methods used for visualizing phylogenetic trees graphically display characteristics of the phylogenetic information. An optimal visualization of reticulate networks should preserve these characteristics as much as possible.

The software currently available for the calculation and analysis of reticulate networks consists of a variety of basic implementations of algorithms developed to solve the computational task of network reconstruction [DGL04; GB05; HK07; HKLS05; HK05; SH05]. Most of the software has only command line interfaces which typically lack an appealing visualization of the results. It is, however, essential, for the better integration of reticulate networks into standard phylogenetic analysis, to provide easy access to those methods for biologists and an appealing visualization of the results.

SplitsTree 4 [HB06] incorporates a variety of methods for the calculation, visualization and interpretation of phylogenetic trees and implicit phylogenetic networks. Two main advantages of SplitsTree 4 are the graphical user interface (GUI) and the integration of algorithms via an interface-driven class loader, resulting in an extendable plugin architecture. Furthermore, SplitsTree 4 allows the user to edit the graph interactively and change edge lengths, edge widths, edge labels, vertex labels, vertex positions and colors. These features make SplitsTree 4 very suitable for integrating of complex graphs such as phylogenetic networks, as can be seen, for example, from the integration of split networks.

In this section, we present an extension of SplitsTree 4 that enables the program to handle reticulate networks. The extension solves two important problems: the visualization of these networks and an efficient integration of reticulate networks into SplitsTree 4.

5.1 Basic Notations

Let us first recapitulate some notation from Chapter 2.

A rooted tree T has a natural ordering of the vertices, such that $v \leq v'$ if v lies on the path from the root to v'. If $v \leq v'$, we say that v is an *ancestor* of v' and v' is a *descendant* of v. For any set of vertices V, a vertex v is called *minimal* with respect to V if for all v' in V, it holds that $v \leq v'$. For any edge e, we use $\alpha(e)$ and $\beta(e)$ to denote the source and target of e, respectively.

It follows from the definition of a rooted reticulate network that each reticulation $r \in V_R$ is contained in one or more cycles in the corresponding undirected graph of the form $C = (r, p(r), w_1, e_1, \ldots, e_{k-1}, w_k, q(r), r)$, with $w_i \in V$ and $e_i \in E \setminus \{p(r), q(r)\}$ for all i. (Note that additionally, r can also be contained in one or more cycles that do not contain $p(r)$ and/or $q(r)$). We say that two reticulations $r, r' \in V_R$ are *dependent* if a cycle that contains both r and r' exists.

For any reticulation vertex r, let $p(r)$ and $q(r)$ denote the two associated reticulation edges. Furthermore, let v_p^r and v_q^r denote the two ancestors of r with respect to $p(r)$ and $q(r)$. The *lowest single ancestor* $lsa(r)$ of a reticulation r is the minimum of all vertices in V that is connected to r by two paths p and p' that share no vertices except for $lsa(r)$ and r.

5.2 Drawing Reticulate Networks

One important approach to drawing trees is the equal angle algorithm which was developed by Meacham (see [Fel04]). It guarantees that edges do not intersect and the runtime of the algorithm is linear with respect to the number of leaves. Our algorithm for visualizing recombination networks generalizes this algorithm. To be able to draw reticulate networks, our algorithms adds an ordering step at each vertex, which chooses an optimal ordering of the descending edges, thereby minimizing the number of crossings between reticulation edges and other edges. It can easily be altered for use with any drawing algorithm for trees.

We will start out with a description of the equal angle algorithm and then define some basic properties of the optimization. Finally, we will give solutions to the problem of minimizing crossing edges in a drawing of a reticulate network, and the optimal placement of reticulation vertices.

The equal angle algorithm is a recursive algorithm that starts at an internal vertex of a tree. For each subtree connected to this vertex, it assigns an angle proportional to the fraction of leaves it contains. In the next step, it assigns, to each subtree, a sector of the circle corresponding to its angle and draws the edge to the subtree in the middle of the sector. It places the sector of the subtree in a way that the sector is centered on the end of the branch and the branch is pointing at the bisector of the angle. It then recurses to the starting vertex of the subtree and assigns each newly discovered subtree a fraction of the angle proportional to its size. Each subtree is then placed in the sector of the starting vertex. The recursion is repeated until the algorithm has assigned angles to each branch of the tree. The only

modifications for rooted trees are the explicit starting point at the root of the tree and that only a fraction of the cycle is used to draw the tree. For a detailed description of the algorithm, see [Fel04].

The rooted equal angle algorithm is not directly applicable to a reticulate network since for each reticulation, the algorithm has to decide which of the reticulation edges it wants to use; either choice may be suboptimal. The idea behind our approach is to use neither of them. The *influence* of a reticulation upon the graph structure is bounded by the reticulation and its lowest single ancestor, so therefore we decided to define an *auxiliary edge* between those two vertices and let the algorithm to use the auxiliary edges for the layout of the graph. When the algorithm reaches a vertex, each descending edge is checked for its status (being either a tree-edge, an auxiliary-edge or a reticulation-edge), and only tree- and auxiliary-edges are considered in the drawing process.

With these modifications to the rooted equal angle algorithm, it is possible to visualize reticulate networks, but such a visualization is not very appealing. To obtain an improved method, we will address two key problems. The first problem is the crossing of reticulation edges: even though it can not always be avoided, the number of such events should be minimized. The second problem is that the auxiliary edges are artifical edges and their optimal edge length must be determined. In the following, we will show solutions to these two problems.

Minimizing crossing edges

Edges that cross each other are undesirable when drawing a graph and their number should therefore be minimized. It is well known that solving this problem is, in general, computationally hard [MGS76]. The equal angle algorithm assures that we only have to deal with reticulation edges crossing other edges. Furthermore, the construction of the auxiliary edges implies that edges that can be crossed by the reticulation edges are descendent edges of the lowest single ancestor of the reticulation. The optimization starts at the root of the networks and optimizes the arrangement of the directly descending vertices. It then continues the optimization iteratively at each directly descending vertex in the determined order given and continues until it has optimized all placements.

Let V_v^T be the set of tree vertices directly below a vertex v and let V_v^I be the set of reticulation vertices connected to v by auxiliary edges. We say that a *tree path* $p(v, v')$ from a vertex v to a vertex v' exists if v' is a descendant of v and every edge in $p(v, v')$ is either a tree- or auxiliary-edge. Furthermore, we say that a reticulation r is *easily reachable* from a vertex v if a tree path $p(v, v_p^r)$ exists. Finally, let R_v be the set of all reticulations that are easily reachable from the vertex v.

The set R_v can be divided into those reticulations r for which $v = lsa(r)$, which we will again denote by R_v^G; v is a descendant of $lsa(r)$, denoted by R_v^D; and v is an ancestor of $lsa(r)$, denoted by R_v^A. If v is the root, R_v^D is empty. The set R_v^D can be divided further. Since for a reticulation r in R_v^D, the vertices directly below $lsa(r)$ have been previously sorted, we can denote

$\overline{R}_v^D$ as the set containing those r in R_v^D for which r has a lower rank than the directly descending vertex of $lsa(r)$ leading to v via a tree path.

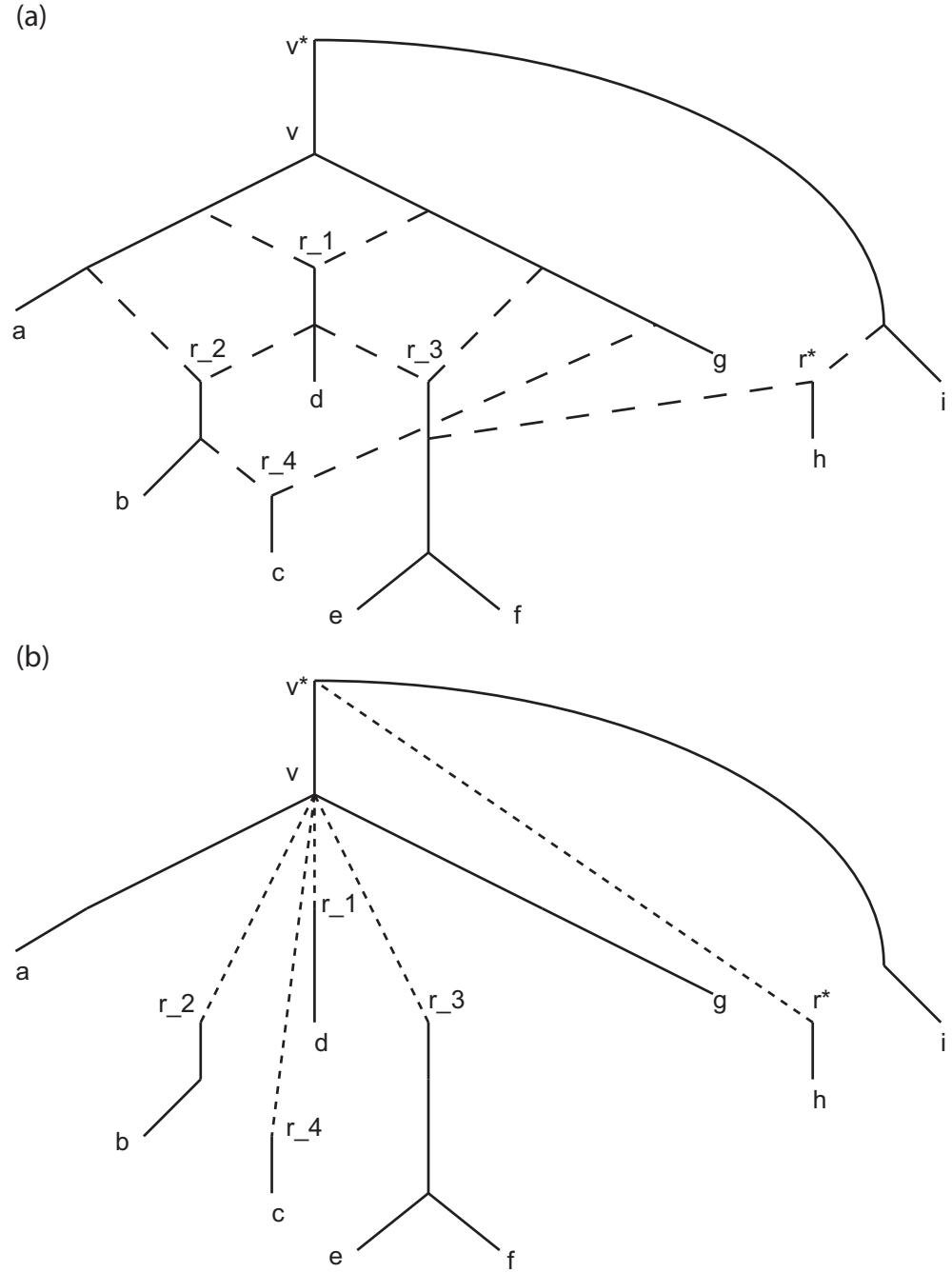


Figure 5.1: **Example of the Layout Optimization** Figure (a) shows an reticulate network. The reticulation edges of the network are shown as dashed lines. Removing the reticulation edges and integrating the auxiliary edges into the networks leads to a tree structure, as shown in (b) (auxiliary edges are drawn as dashed lines). The set of easily reachable edges R_v of the vertex v contains the reticulations $r1, r2, r3, r4$ and r^*. The set R_v^D only contains r^* and is equal to $\overline{R}_v^D$. The placement of r^* has cost 1 since the reticulation edges only cross the edge that leads to leaf g. The placement of $r4$ has cost 2, since the reticulation edges to the right crosses $r1$ and $r3$.

The aim of our optimization is to find a linear arrangement of the vertices

in $V_v^T \cup V_v^I$ such that in the subtrees of the vertices in $V_v^T \cup V_v^I$, the number of reticulation edges intersecting with tree edges is minimized. We define the *optimal linear arrangement graph* $OLA^v(V, E)$ of a vertex v as one that contains a vertex representative for any vertex in $V_v^T \cup V_v^I$. We add a weighted edge between any two vertices (v_i, v_k) in V and set the weight w_{ik} of the edge to $|R_{v_i}^D \cap R_{v_k}^D|$. More formally:

Problem 5.1 *With*

$$x_{ik} = \begin{cases} 1 & if\ vertex\ i\ takes\ position\ k, \\ 0 & otherwise, \end{cases} \quad \forall i, k$$

minimize

$$\sum_{(i,j)\in E} w_{ij} x_{ik} x_{jl}\ |k-l| + \sum_{i\in V} |R_{v_i}^D \cap \overline{R}_n^D|\ x_{ik}\ |k| + \sum_{i\in V} |R_{v_i}^D \cap (R_n^D \setminus \overline{R}_n^D)|\ x_{ik}\ |\,|V|-k|$$

$$subject\ to \sum_{i\in(V)} x_{ik} = 1\ and \sum_{k\in\{1,\ldots,|V|\}} x_{ik} = 1,\ \forall i, k$$

The optimal linear arrangement problem is well known to be $NP-$hard [GJ83]. Nevertheless, this arrangement problem is, in general, less complex than minimizing all crossing edges at once. Interestingly, a couple of additional constraints exist that we may apply to the ordering, leading to a "greedy" solution that works well in most cases. One constraint that can place upon the structure is that any reticulation r should be positioned between v_p^r and v_q^r in the ordering. Consequently, one should place v_p^r and v_q^r before placing r.

Another constraint represents the dependency of the reticulations upon each other. For any pair of reticulation r, r' in R_v^G, we say that r is less than r' if and only if a tree path $p(r, v_p^{r'})$ exists. To meet the first restriction, one has to place r before one can place r'. The graph that can be constructed from the relations between the reticulations must be acyclic, since the reticulation network is acyclic. Consequently, one can use a standard topological sorting algorithm to obtain a linear ordering $Ord_l(R_v^G)$ for the reticulations in R_v^G.

The optimization algorithm iterates through the ordering and at each reticulation r, it first places v_p^r and v_q^r, if necessary, and then r. If all reticulations are placed, the algorithm processes all descending tree edges that have not yet been placed. At each step, the algorithm places the vertex at the position that minimizes the score given in Problem 5.1. After all vertices have been placed in the linear arrangement, the result is returned to the main method.

Optimal placement for reticulation vertices

Having calculated the angle and optimal arrangement for each edge, we have to place the vertices. Tree vertices can be placed in the same way as in the standard equal angle algorithm. But since auxiliary edges do not come with a given length, the algorithm has to calculate an optimal placement for each of the reticulation vertices. Such a placement has to incorporate the conditions of the equal angle algorithm, otherwise one might face unnecessary crossings between edges. Note that there are two cases for which the algorithm has to consider different placement methods. In the first case, the vertices v_p^r and v_q^r of a reticulation r are both different from the lowest single ancestor $lsa(r)$; in the second case, one of them is equal to $lsa(r)$.

In both cases, the algorithm places the reticulation vertex r on the bisector of the sector assigned to its auxiliary edge. In the first case, the distance between r and $lsa(r)$ should be larger than the minimum distance between $lsa(r)$ and the line $l(v_p, v_q)$, indicating that r is a descendant of v_p and v_q. In other words, the algorithm assumes that the angles $v_q v_p r$ and $v_p v_q r$ are positive. In the second case, assume that v_q is equal to $lsa(r)$. The algorithm first calculates the point on the bisector r_t that has the same distance to $lsa(r)$ as v_p and then ensure that the angle between $r_t v_p r$ is positive. We added an option to the algorithm so that the user can specify the (maximum) value of this angle; the standard value is 15°.

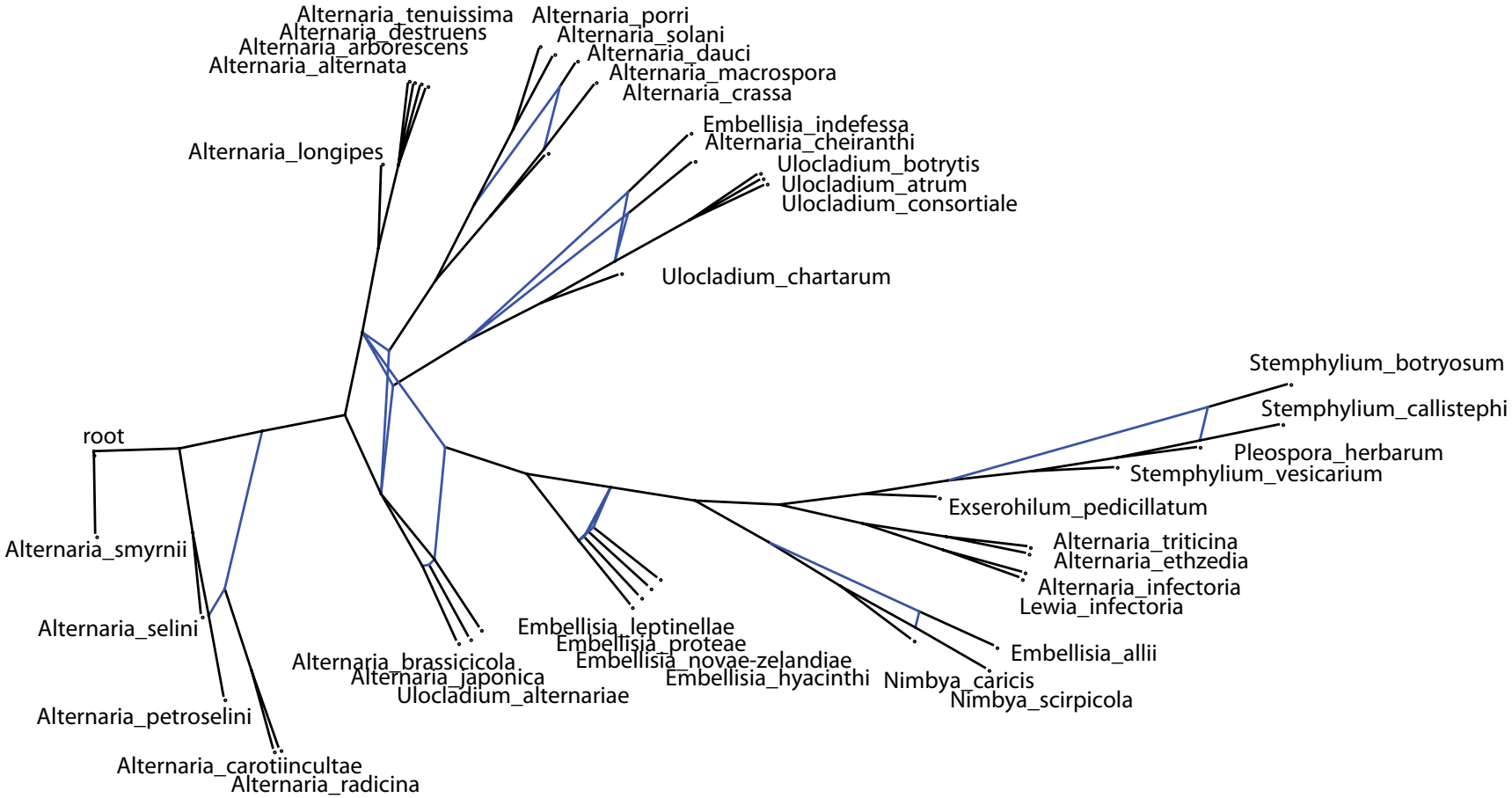


Figure 5.2: **Example of the drawing algorithm** The figure shows the drawing of a reticulation network that we recently published [HK07]. It is based on three gene trees described in [PB03].

5.3 Integration of Reticulate Networks into SplitsTree 4

We started to integrate reticulate networks into SplitsTree 4 in our RECOMB 2005 article [HKLS05]. Originally, such methods were squeezed into the existing data structures within SplitsTree 4. The program itself is built around a group of core classes, each one representing a different type of information. The standard file format of SplitsTree 4 is the *Nexus* [MSM97] file format and each core class has its own Nexus representation (called a *Nexus class*). Consequently, developing a Nexus representation of reticulate networks is essential for their integration into SplitsTree 4.

The integration of a new core class into SplitsTree 4 is a complex process. Most of the functionality had to be extended to support this new class. One of the more interesting elements of this extension is the graphical user interface in the *Algorithm Window* of the SplitsTree 4 program.

The Reticulate Nexus block

To build a Nexus representation for a reticulate network, one needs to find an efficient way to encode it as a string. We decided to use a version of the *extended Newick* (eNewick) [MM06] format. In general, the eNewick format allows labels to be present up to two times within the network. A label is allowed to appear once as a leaf and once as an internal label. Whenever a label occurs twice, the leaf is identified with the internal vertex, thus providing a network with vertices of indegree two.

A lot of research has lately been focused on proving some interesting decomposition theorems [GB05; HK07; HKLS05] for reticulate networks (see also Chapter 3). The general motivation of these theorems is that the calculation of a reticulate network with a minimal number of reticulation events from some given information is hard [WZZ01; BS07]. The idea is to decompose each network into its two-connected components and to calculate the minimal solutions of each two-connected component separately. Following the idea of decomposing reticulate networks, each two-connected component may have several solutions and the possible combinations of these solutions grow exponentially, which is a problem if the number of two-connected components is large. Consequently, we decided that the Nexus representation of the network needs to reflect the two-connected components.

Note that any reticulate network contains either a two-connected component or a tree like element, that contains the root. We call this particular element the *root component*. The two-connected components are called *netted components* and for each netted component, a number of solutions may exist. Any connected component that is not a two-connected component is a *tree component*. Each tree component may appear more than once within the possible configurations. The way in which these three basic elements are combined is left to the user.

We now describe the Nexus notation for reticulate networks; the schematic of this notation is shown in Figure 5.3:

```
BEGIN RETICULATE;
    DIMENSIONS
        NTAX = number-of-taxa
        NROOTCOMPONENTS = number-of-root-components
        NNETTEDCOMPONENTS = number-of-netted-components
        NTREECOMPONENTS = number-of-tree-components;
    [ FORMAT
        [ ACTIVEROOT = position-of-active-root-component ]
        [ ACTIVENETTEDCOMPONENTS =
            positions-of-active-netted-components ]
        [ SHOWLABELS = [ INTERNAL ] [ TREECOMPONENTS ]
            [ NETTEDCOMPONENTS ] ]
    ]
    TREECOMPONENTS
        [ name = tree-component-specification; ]
        [ name = tree-component-specification; ]
        ...
    NETTEDCOMPONENTS
        [ netted Component name = ]
            [ name = netted-component-specification; ]
            [ name = netted-component-specification; ]
            ...
        [ netted Component name = ]
            [ name = netted-component-specification; ]
            [ name = netted-component-specification; ]
            ...
        ...
    ROOTCOMPONENTS
        name = root-component-specification;
        name = root-component-specification;
        ...
END;
```

Figure 5.3: **Reticulate Nexus Block Schematic** Shown is a schematic of the Reticulate nexus block as it its implemented in SplitsTree 4. The block is divided into three parts: *Dimensions* contains all informations about the dimensions of the reticulate network; *Format* is an optional element that describes the configuration of the reticulate network; and *TreeComponents*, *NettedComponents* and *RootComponents* contain the string representations of the reticulate network.

In general, one needs to save the components containing the root in the **RootComponents** section. Any such string should either be formatted in standard eNewick, in Newick format where any two leaves with the same label are identified by the name of a tree component, or in Newick format where at least one leaf is labeled with the name of a netted component.

The **NettedComponents** section contains a list of all two-connected

components. Each one must be identified by a unique name and there must be at least one string representation given for each one . Any such string must either be formated in eNewick, or in Newick format where any two leaves with the same label are labeled with the name of a tree component.

The **TreeComponents** section contains a list of uniquely named strings in Newick format, where leaves can be labeled with the name of netted components.

In the **Dimensions** section, the dimension of the Nexus block is given, i.e. the number of taxa (**NTax**), RootComponents (**NRootComponents**), NettedComponents (**NNettedComponents**) and TreeComponents (**NTreeComponents**).

In the optional **Format** section, the details about the representation of the network are given. The **ActiveRoot** must be a positive integer that is less than **NRootComponents** and specifies the RootComponents that should be used. The **ActiveNettedComponents** must be a list of integers, separated by a blank character, of length **NNettedComponents** specifying, for each netted component, which netted component specification should be used. The **ShowLabels** variable can be used to label internal vertices by the labels given in the **NRootComponents**, **ActiveNettedComponents** or **TreeComponents** sections.

Interactive Exploration of a Result

Another advantage of SplitsTree 4 is the interactivity between the program and the user. Because of the Reticulate class, this interactivity can be optimally used for reticulate networks.

In SplitsTree 4, specific parameters for phylogenetic analyses are configured through the *Algorithm Window*. Besides the algorithms, this window provides the functionality to modify the underlying data. We augmented the window with the new *Reticulate* tab. In addition to the default sub-tab *Method*, we implemented the sub-tab *Filter*. It allows the user to change the root component and the configurations for each netted component interactively. The user can browse through the set of solutions, and changes in the configuration are directly visualized on the main view.

Chapter 6

A Plugin Management System for Java Software

The development of a management system for plugins was motivated by our experience with plugins in SplitsTree 4.

SplitsTree 4 ([HB06]) provides a framework for the calculation and visualization of phylogenetic trees and networks. It is programmed in Java and has a command line interface as well as a graphical user interface. The program contains a vast variety of phylogenetic methods such as evolutionary distance methods, maximum likelihood distances, tree building methods, split-network methods and visualization algorithms. It allows the user to import and export data in different file formats and to save visualizations in various graphics formats. Another advantage of SplitsTree 4 is that all algorithms are integrated into its framework architecture via an interface-driven class loader, making it an easy task to incorporate new methods into the program. All methods in SplitsTree 4 implement a specific interface that allows the program to identify plugins dynamically and to integrate them into the framework at runtime.

One of the main problems associated with plugins is in their distribution. The only possibility for SplitsTree 4 to distribute new plugins so far, was to release a new version, containing these. Every user who wanted to use newly developed methods within SplitsTree 4 had to install the new version of the software. This approach essentially contradicts the design of the software, which allows plugins to be integrated into the software at runtime. This seems to be one reason why only a few plugins have been developed for this application by researchers outside our own research group. Through personal communication it became apparent that there is a community of developers who would like to implement new plugins for SplitsTree 4 and a community of users who would appreciate new methods to be integrated dynamically into the application.

Since most of the applications developed in our group allow plugins to be dynamically integrated at runtime, it became desirable to develop a general system which could be easily adapted for different programs. The design of the developed system has been adapted especially for this purpose and needs only minimal interaction with the underlying software. In this chapter, we present this generic concept and explain how it has been integrated into

SplitsTree 4.

The plugin management system consists of a defined format in which plugins are programmed, a central submission facility for new plugins, a remote access to the centrally stored plugins, and the possibility of sharing plugins privately. The general concept of our solution is that a plugin consists of a particular core class (in the case of SplitsTree 4, this is a *Transformation*) and a set of Java libraries. The developer has to use our software (the Plugin Creator) to create a *Plugin Archive* from these files. The Plugin Archive can then be submitted to our central plugin database (via a web interface) or be distributed directly to the software users. A software user has to download the centrally stored plugins (via the Plugin Overview or the web page) or provide Plugin Archives, to integrate them.

The first section of this chapter describes the design and layout of the management system. The second section focuses on the implementation details and the last section describes the integration into SplitsTree 4.

6.1 The Management System

The management system was designed on the basis of a *use-case* analysis. The use-case diagram for the system is shown in Figure 6.1. Basically, two types of users can be distinguished; the developer and the software user, where the developer extends the software user. Furthermore, there are three phases that can be distinguished: the *Plugin Development*, the *Plugin Distribution* and the *Plugin Usage* phase. The Plugin Development phase is restricted to the developer. In the Plugin Distribution phase, the developer acts as provider and the software user as customer. In the Plugin Usage phase, the software user integrates the plugin into the application. The system provides software for the support of all three phases and consequently presents a complete system for managing plugins.

To adjust the management system to the users needs optimally, we first have to implement the software concept *Plugin*. Based upon the software in our research group that uses plugins, it is assumed that a plugin contains a core class (e.g. implementing a Java interface). A plugin may also contain a set of Java libraries (used by the central class) and source files. In addition to these software elements, a plugin should also contain information about the author, the version and a description. In this context, we have to distinguish between information that is identical in all versions of a plugin (name, author, description etc.) and information that changes from version to version (version number, supported software versions etc.). These two categories of information are associated with each other, but can be addressed independently. Consequently, we allocate a plugin to one category representing the general plugin information and another category containing version specific information. To ensure a smooth transfer of information between the different phases of the management system, the integrity of a plugin has to be guaranteed. To do so, we have defined the structure *Plugin Archive* and developed the *Plugin Creator* application. The Plugin Creator should be used at the end of the Plugin Development phase to generate a Plugin

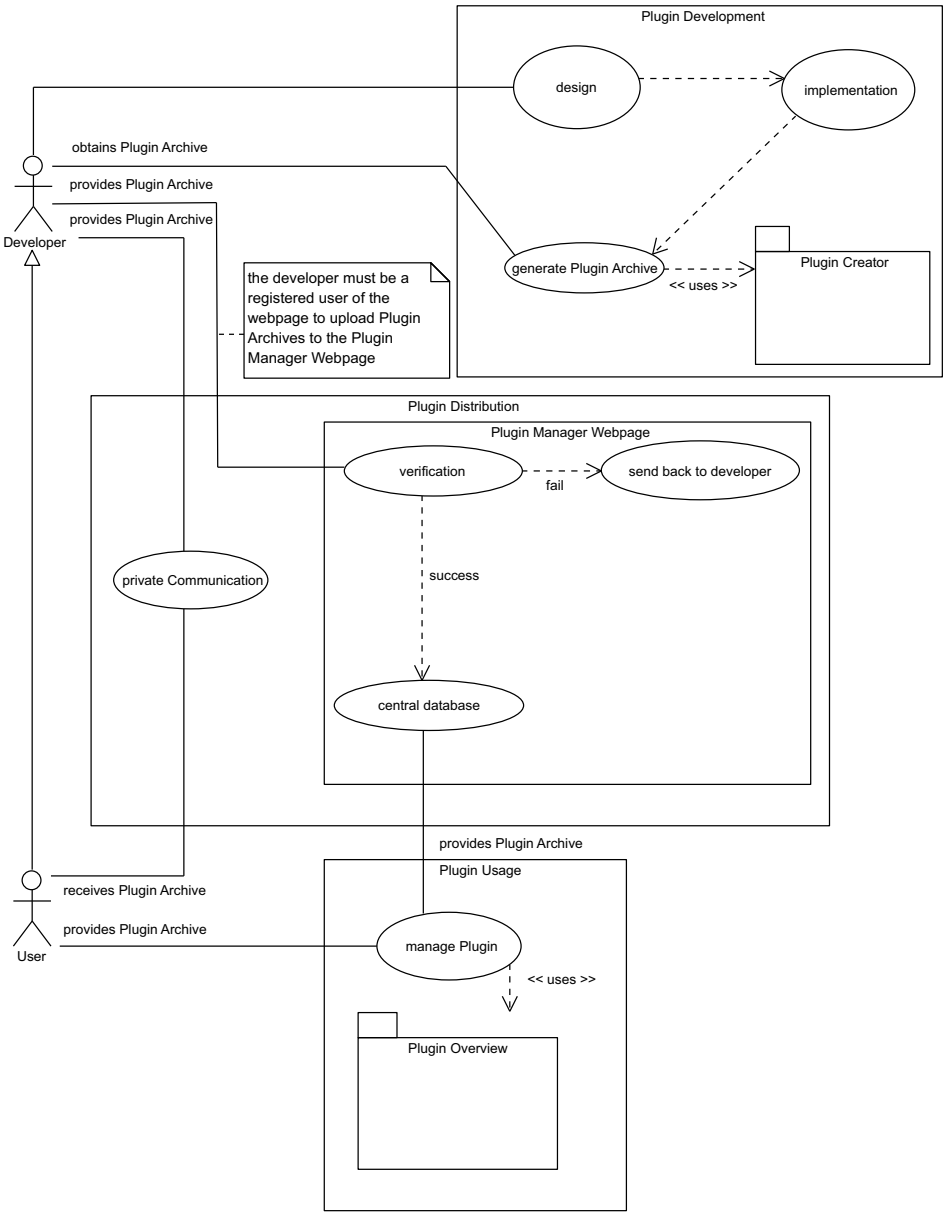


Figure 6.1: **Use-Case Diagram of the Plugin Management System** There are two different kinds of users: the developer and the application user. Furthermore, the system can be divided into three phases: the Plugin Development phase, the Plugin Distribution phase and the Plugin Usage phase. The Plugin Development phase is restricted to the developer, and at the end of this phase, the developer obtains a Plugin Archive. During the Plugin Distribution phase, the developer can either submit his Plugin Archive to the central database, or distribute it directly to the user. The user can access the plugin in the Plugin Usage phase by either providing a Plugin Archive to to the Plugin Overview or by using the application to download plugins from the central database.

Archive from the programmed components.

The Plugin Creator is an application that allows the developer to integrate all necessary information and classes of the plugin into a Plugin Archive. The Plugin Creator is a stand-alone application that is distributed with SplitsTree. The two main options in the Plugin Creator are the creation of a new plugin or the creation of a new version of an already installed plugin. At the

startup, the Plugin Creator will ask for general information (e.g. the plugin name, the name of the developers, email contact, a short description of the plugin and the associated publication). When sufficient information has been given, the application proceeds to collect the plugin main class, additional Java archives and the source files. Finally, the Plugin Creator generates a Plugin Archive in a designated folder.

After the Plugin Development phase is completed, the Plugin Distribution phase begins. There are two ways of distributing plugins: the official and the private way.

For the official way, we have updated the SplitsTree 4 homepage and implemented an area that allows developers to submit Plugin Archives or to manage Plugin Archives that have already been submitted. A developer that would like to submit a new plugin has to register first. After access has been granted, the developer can start uploading Plugin Archives. Furthermore, the developers can discontinue their older or no longer supported plugins, change information of already uploaded plugins, and/or change their personal information.

The private way of distributing plugins is implemented in our software, but should only be used for developmental purposes and not as a regular method of distribution. One apparent problem that is associated with this way of distribution, is the consistency of plugin names and versions. The upload of a Plugin Archive into the central database allows a consistent association of all elements of a plugin. The version-specific information is uniquely associated with the general information and consequently, new versions can always be uniquely linked with their general information. This can not be guaranteed in the unregistered case. Accordingly, the Plugin Creator distinguishes between both cases. However, distributing a developmental version via the central database does not seem appropriate and consequently, the system supports private distribution.

The last phase in the system is the Plugin Usage. A natural dependency exists between the application of the management system and the content application. To minimize the dependency between these two elements, plugins are installed into a local folder that is chosen by the content application. The content application has to integrate the plugins that are stored in this local folder dynamically (for an example, see Section 6.3).

The *Plugin Overview* is the application in the management system that is responsible for the local administration of the plugins. This program not only provides the means for the administration, but also for the integration of new plugins. All available plugins and versions are synchronized to a local plugin database upon request. The local plugin database is stored in a flat file and accessed via a HSQLDB driver (http://hsqldb.org). The user can choose to install, remove, upgrade or downgrade any available plugin from an interactive list. If a plugin is to be installed, the Plugin Manager downloads the necessary Plugin Archive from the central database to a local folder and deploys the files. Alternatively, the user may provide a Plugin Archive to the Plugin Overview via a file chooser. Any up- or downgrade will erase the installed version of the plugin and deploy the new version. The new plugins will be available to the user directly after installation, without restarting

SplitsTree.

Since, in general, company networks block most network ports, we had to develop a solution that provides a connection to the central database using a port that is in typically open, or can be used with a proxy. The port that is especially suited for this is the *HTTP* port. Therefore we have implemented a HTTP-tunneling-script that provides access to the central database via an encrypted connection through HTTP.

6.2 Implementation Details

The implementation of the management system has to include a set of properties that can be deduced from the use-case diagram shown in Figure 6.1. For example, the management of plugins installed by an user is implemented using a local database. This database must be synchronized to the central database that is installed on a web server. Furthermore, the local database must distinguish between plugins added via the web server and those added from a local Plugin Archive.

The central plugin database is a MySQL database and is accessed via a MySQL Connector (http://www.mysql.com). The local plugin database is stored in a flat file and accessed via a HSQLDB driver (http://hsqldb.org). Both databases contain two tables, *Plugin* and *Version*. Every entry in the Version table is associated with exactly one entry in the Plugin table. Whenever an entry is written into the Plugin or Version table of the central database, an unique registration identifier is generated. Plugins in the user database that contain a registration identifier can be identified as plugins from the central database. The central database contains a third table *User*, for the management of the users of the developer web-site.

The access to the databases is controlled by the *database handles*. Each database has its own database handle, which implements the interface *database.DBHandle* (see Figure 6.2). The methods that are implemented by the interface can be divided into two groups:

- those methods for the access to the database: *openDB()* and *closeDB()*; and

- those methods that handle requests to the database: *getPreparedStatement()*, *executeSQLStatement()* and *executeSQLQuery()*.

The method *getPreparedStatement()* is used to avoid SQL-injections, the method *executeSQLStatement()* executes a statement and returns *true* if the execution was successful, and the method *executeSQLQuery()* executes a query and returns the result in a *java.sql.ResultSet*. In addition to the compulsory methods of the interface, the database handle for the local database *database.HSQLHandle* implements the methods *createDB()*, for the initialization of the local database, and the method *syncOfflineDBWithOnlineDB()*, to synchronize the local database with the central database.

For each table in the databases, we have created a Java class representing an entry in the table, i.e. *database.PluginData* for the Plugin table, *database.VersionData* for the Version table and *database.UserData* for the User table. Each of these classes implements the interface *database.DataSet* which provides the fundamental functions *isNew()*, *updateIntoDatabase()*, *writeIntoDatabase()*, *delete()* and *toString()*. Additionally, the classes PluginData and VersionData implement the interface *IODataSet*, which describes the necessary methods for the synchronization between the different elements of the management system, e.g. the input and output handling. The methods *read()* and *readHTML()* initialize the object from a file and the methods *write()* and *writeHTML()* write the object to a file. Additionally, the PluginData class, implements the methods *getVersions()*, which extracts all entries from the Version table that are associated to the PluginData object; and *getInstalledVersion()* to obtain the VersionData object representing the installed version of the plugin. The VersionData class implements the method *isValid()*, which returns *true* if the version is supported by the content application; and the static method *isValidVersion()*, which controls if the version is in a format that is supported by the software. The UserData class is only used by the central database and is used for the management of the users of the web-site. It implements the static methods *checkPassword()*, which returns *true* if the password corresponds to the user; *createPassword()*, which creates a password of a given length; and *encryptString()*, which encrypts a given string using a *sha1* encryption method [NIS95].

To facilitate the management of a plugin and its available versions, we have implemented the additional class *data.Plugin*, which holds exactly one PluginData and one VersionData object. Furthermore, it contains all information about files included in the Plugin Archive of the version (the primary class, Java libraries and source files). Accordingly, the class implements the necessary functionality to generate a Plugin Archive (*zip()*). It can also retrieve all information contained in a Plugin Archive (*readZipFile()*). The method *removeOfflinePluginsWithSameName()* removes all plugins from the given database which are not registered in the central database and have the same name. Finally, the class implements the two static methods *installPlugins()* and *uninstallPlugins()* to install and un-install a given list of plugins into/from a given directory.

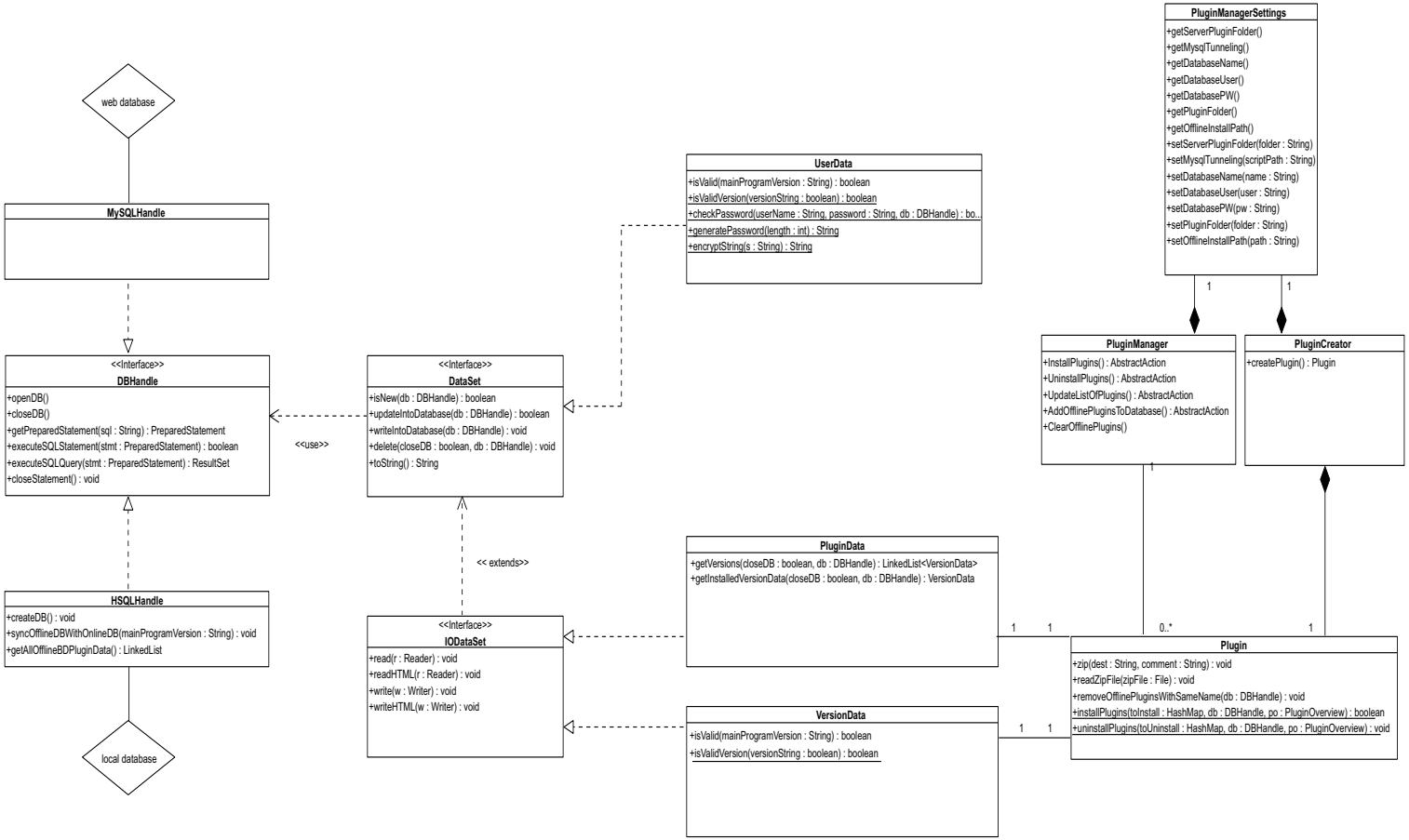


Figure 6.2: **Class Diagram of the Plugin Management System** The class diagram of the management system is divided into three parts. The left-hand part corresponds to the classes that handle the access to the database. The middle part contains the interfaces for the DataSet objects and the objects themselves. The right-hand part describes the User interfaces, along with their support classes.

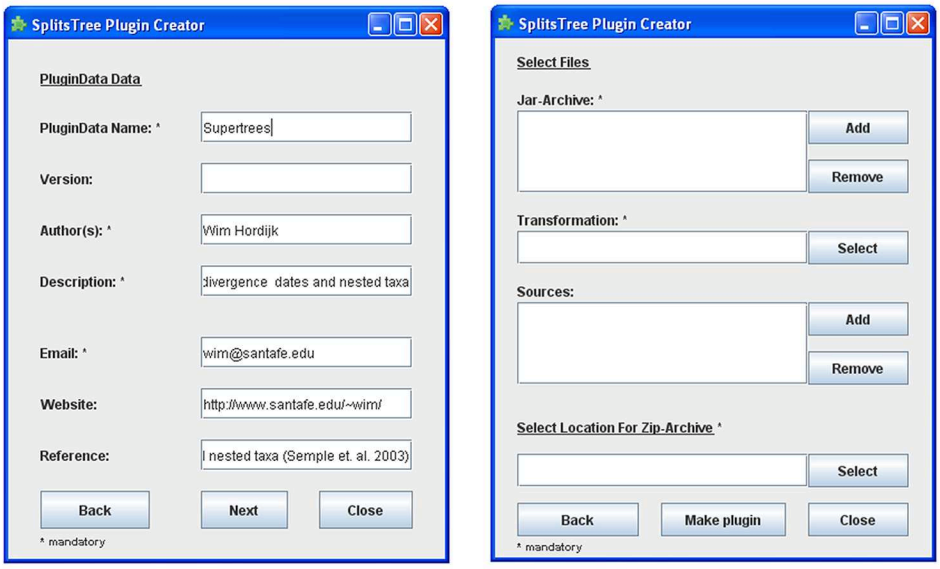


Figure 6.3: **The Plugin Creator** The Plugin Creator is a graphical user interface to the Plugin class that allows users to create a Plugin Archive. The first frame of the software (left) prompts for the necessary plugin and user information. The locations of the plugin files and the output directory are entered in the second frame.

The user interface to the management system is provided by the two classes *gui.PluginCreator* and *gui.PluginOverview*.

The Plugin Creator is shown in Figure 6.3. Basically, it is a graphical user interface for the Plugin class. It fills in all the necessary information of the Plugin class, so that the *zip()* method creates a valid Plugin Archive. The Plugin Creator is divided into two parts. In the first part, the user provides the necessary general information. The mandatory fields *Plugin Name*, *Version*, *Author(s)* and *email* must be filled with valid information and the additional fields *Web-site* and *Reference* are optional. In the second part, the user can supply the files of the plugin. The *Transformation* is the main class of the plugin and is mandatory. In addition, the user can also supply Java libraries needed by the main class in the *Jar-Archive* field. Finally, the Plugin Archive will be generated at the given location.

The PluginOverview is shown in Figure 6.4. It is a GUI-driven interface to the local database, that manages the locally installed plugins. It provides six basic functions:

- **Install** to install plugins that are marked in the *Select* column of the list;

- **Add from File** to add a Plugin from a *Plugin Archive* (download-able from the SplitsTree web-page or privately shared) to the list;

- **Update** to update the list of plugins using the central database (access to the Internet is mandatory for this option to work);

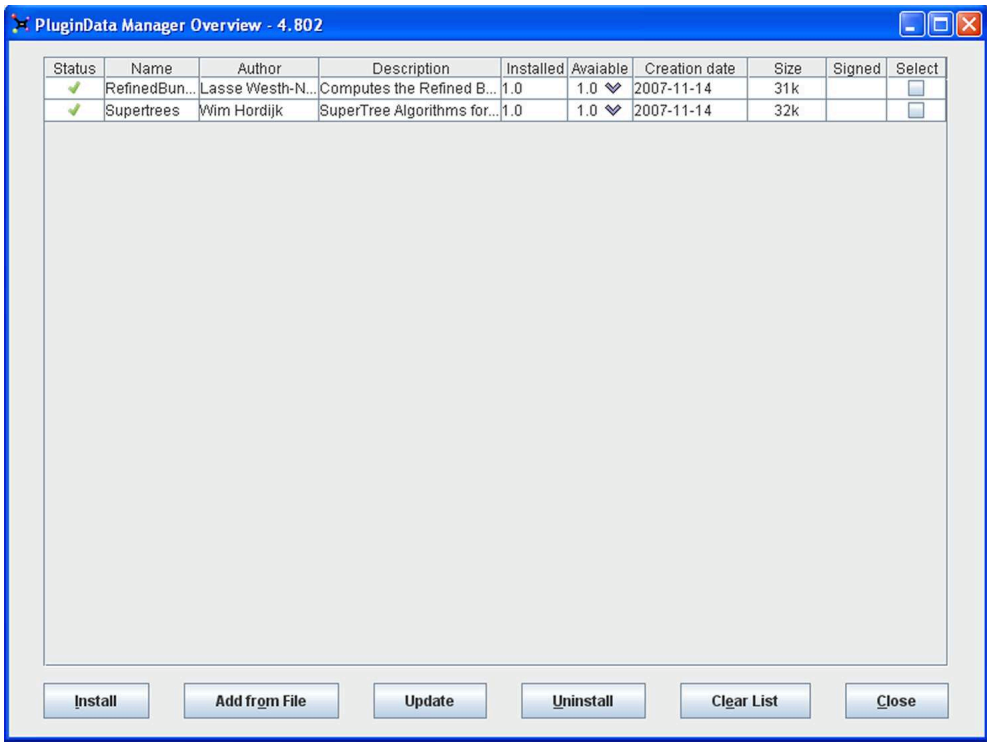


Figure 6.4: **The Plugin Overview** The Plugin Overview contains a JTable that displays the local plugin database, and a set of buttons to manipulate the local database and to install or un-install the plugins into/from the designated folder.

- **Uninstall** to remove plugins that are marked in the *Select* column of the list;

- **Clear List** to remove all plugins from the list that are not installed and not part of the central database.

- **Close** to close the PluginOverview.

Since the management system is designed to serve for more than one software, there is a central configuration class *gui.PluginManagerSettings*. Most of the configuration elements of the class are independent of the application but some must be set at the startup. To accomplish the interaction between the PluginManagerSettings and the Plugin Creator or Plugin Overview, the two user interfaces do not contain a *main()* function. To start them, one has to write a class that configures a PluginManagerSettings object and creates one of the user interfaces using this object. The settings that must be set before starting one of the user interfaces, are:

- **pluginFolder** the local folder where plugins should be stored,

- **serverPluginFolder** the location on the server where the Plugin Archives are stored,

- **databaseName** the database name for the main application,

- **MysqlTunneling** the location on the server of the MYSQL-tunneling script.

- **mainProgramVersion** the version of the main application.

6.3 Integration into SplitsTree 4

SplitsTree 4 provides a set of core objects that are used to store the data. Each of these objects reflects one type of data, for example, distances, trees or splits. Also, each object corresponds to a "Nexus block" ([MSM97]), which is the standard file format of SplitsTree 4. Any plugin in SplitsTree 4 *transforms* one data object into another data object. These transformations are directed from *Unaligned* → *Characters* → *Distances* → *Trees* → *Quartets* → *Splits* → *Network*. For example, a Characters object can be transformed into a Network object, but a Network object can not be transformed into a Splits object. For a plugin to be recognized by SplitsTree 4, it has to implement one of the interfaces corresponding to these possible transformations. We have implemented an interface for each possible transformation and named these accordingly. For example, a transformation from Distances to Splits must implement the interface *Distances2Splits*.

Consequently, a plugin in SplitsTree 4 consists of a *Transformation* and potentially some additional classes that must be provided in a Java library. Using the software *splits.progs.SplitsTreePluginCreator*, the developer can generate a Plugin Archive for SplitsTree 4. This Plugin Archive can then be uploaded into the central database or privately distributed. In the *Window* menu of SplitsTree 4, we added the new sub-menu *Plugin Manager* that will start the Plugin Overview interface.

In addition to these two applications, we have redesigned the SplitsTree 4 homepage and added some new elements. The SplitsTree 4 web-site can be found at:

http://www-ab2.informatik.uni-tuebingen.de/splitstreePluginManager/.

The publicly available pages contain sections for downloads, documentation, web-start and developers. The documentation page contains the SplitsTree 4 manual and the accumulated FAQs. The SplitsTree 4 application, as well as publicly available plugins, can be obtained from the download section. It is possible to start SplitsTree 4 as a web-start application. A link to this feature is provided in the web-start section. Since only registered users are able to upload plugins into the system, the developer section contains a link that leads to a registration form. At the registration page, a new user has to enter some personal information (username, name, e-mail and affiliation). Additionally, to prevent automatic registration, each user has to insert a $CAPTCHA$ string [LvAL04] that is displayed as an image. Upon successful registration, the system sends an email with the user information and a generated password to the administrator. If the information provided is valid, the administrator sends the password to the new user.

Registered user can log into the system in the *developer* section. The developer pages contain sections for an individual's plugins (private plug-

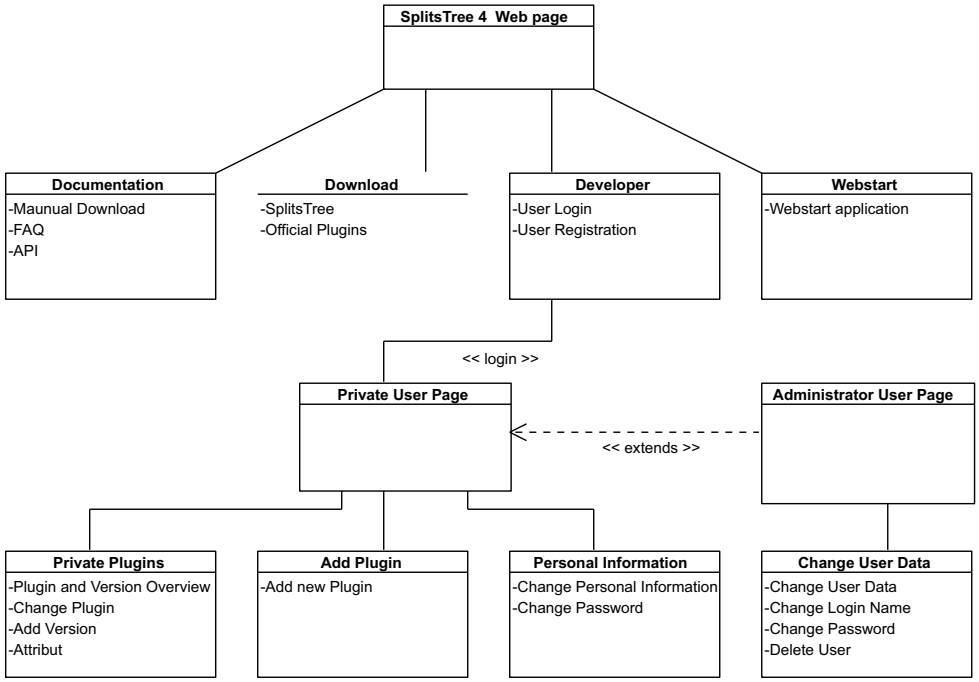


Figure 6.5: **Structure Diagram of the SplitsTree Web-site** The SplitsTree 4 homepage is divided into four main areas. The Documentation area contains the manual, FAQ and API of the software. The Download area contains the installer for SplitsTree 4 and a list of all official plugins. The Web-start area starts SplitsTree 4 as a web-start application. The new area Developer is the interface for the developer section of the web site. Any registered user can log into this area. The user can manage the plugins, add new plugins and administrate his/her user profile. The administrator has additional access to the management site for the users.

ins), a section to upload new plugins (add plugins), and a section for his/her personal information (change personal data). The *private plugin* section provides the developer with a list of his/her plugins. The edit page of any of these plugins allows the developer to change the information about the plugin, to add/delete versions of the plugin, or to delete the plugin all together. To upload a new plugin to the system, the *add plugin* section provides the developer with the opportunity to browse his/her file system for the Plugin Archive. The system checks whether the Plugin Archive is valid and, if so, it transfers the plugin information into the database and places the Plugin Archive in a folder on the web-server. If a developer wants to change his/her personal information (including the password), he/she can do so with a form in the *change personal data* section. If the developer is logged in as an administrator, an additional section will be shown. In this section, the administrator is able to change personal user information, the username, or the password of a registered user. It is also possible to delete a registered user completely.

List of Available Plugins for SplitsTree

- **Supertree**:
  Author: Wim Hordijk
  Description: SuperTree Algorithms for ancestral divergence dates and nested taxa
  Email: wim@santafe.edu
  Reference SuperTree Algorithms for ancestral divergence dates and nested Taxa (Semple et. al. 2003)[SDH+04]

- **RefinedBunemanTree**:
  Author: Lasse Westh-Nielsen and Christian N. S. Pedersen
  Description: Computes the Refined Buneman Tree
  Email: lasse@birc.dk
  Reference: Computes the Refined Buneman Tree (Brodal et al. 2002) [BFO+03]

Chapter 7

Annotation of Phylogenetic Graphs using Jloda

The integration of additional information into phylogenetic graphs is a common technique to clarify complex results. A simple example of such an annotation is the labeling of edges in a phylogenetic graph with bootstrap values. Another possibility is the labeling of functional or taxonomic groups as can be seen in Figure 7.1. In the following, we will use the term *glyph* to represent either a geometric object or a string that is used to label a graph.

The integration of glyphs into phylogenetic software is a challenging problem. The software must not only allow the interaction with the actual phylogenetic graph, but also the interactive integration of additional geometric objects into the graphical display. However, integrating these annotations should only lead to a minimal increase of the complexity in the software, since usability is important for the success of any software.

The library *Jloda* has been developed in our research group as a graph library especially aimed at phylogenetic graphs. Jloda provides the graphical and functional basis of many of our software projects (SplitsTree, Dendroscope, Megan). One of the main features of the library is an interactive user interface to modify the underlying graph. Nevertheless, the library has not been intended to provide secondary graphical objects besides the phylogenetic graph.

Consequently, Jloda had to be adjusted to provide an interactive integration of glyphs into the interface. Furthermore, a variety of glyphs had to be implemented. In this context, two additional design elements had to be taken into account. Firstly, Jloda displays a graph with the help of its own transformation class that maps the graph from its coordinate system to the coordinate system of the displaying Java panel. Secondly, Jloda saves its graphs in a simple string-based format and consequently, glyphs must support this system.

Since glyphs can move freely within the coordinate system of the graph, the system must employ a two-step transformation. The first transformation represents the affine relocation of the glyph within the coordinate system of the graph. The second transformation maps the glyph from the coordinate system of the graph onto the displaying Java panel. It is worthwhile to note that each glyph has its own transformation for the affine relocation

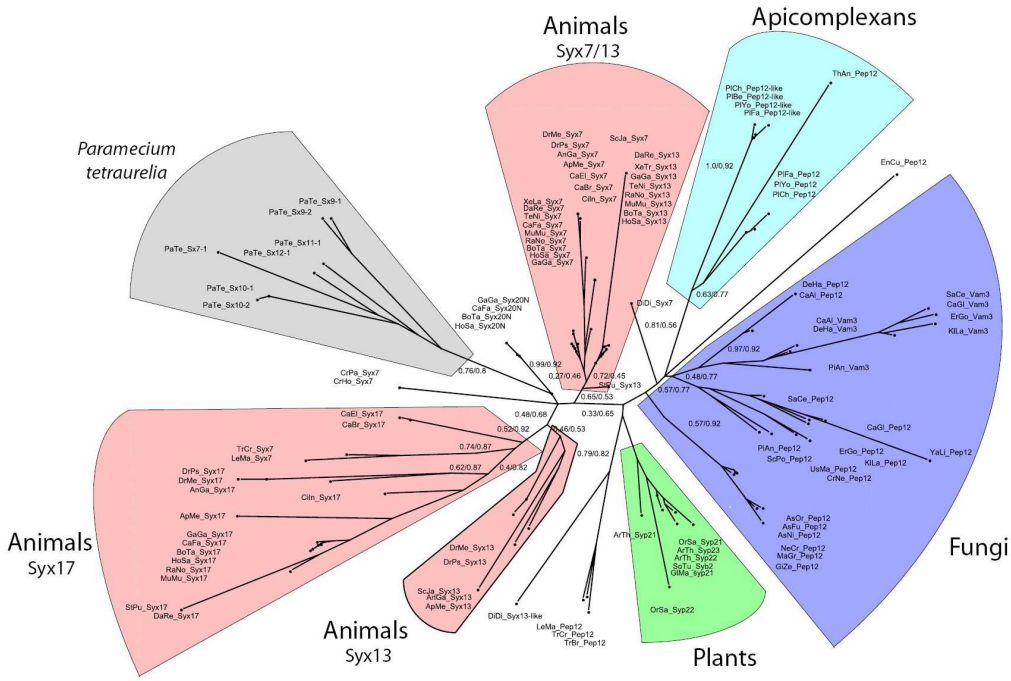


Figure 7.1: **Example of an Annotation of a Phylogenetic Tree** Unrooted phylogenetic tree relating Syntaxin members involved in endosomal trafficking, published in [TKF07]. The figure reflects the increase of Syntaxins within the animal kingdom. Furthermore, fungi and apicomplexans species seem to possess at least two distinct proteins related to endosomal trafficking,whereas plants seem to have undergone at least two independent genome duplications.

within the coordinate system of the graph, but all glyphs use the same second transformation. Furthermore, for the interactivity of the glyphs, changes on the Java panel must be transformed back to the coordinate system of the graph.

In this chapter, we will introduce an implementation of interactive glyphs into Jloda, which enables an user to integrate additional information into a graph, and the integration of this functionality into SplitsTree 4. Finally, we will introduce an algorithm for drawing phylogenetic graphs that provides an automatic visualization of annotated groups within the graph.

7.1 Integration into Jloda

As mentioned above, Jloda is a library developed in our group specifically for phylogenetic graphs. The core graph classes of the library are contained in the package *jloda.graph*. All classes representing a graph are deduced from the class *jloda.graph.Graph*. The deduced classes representing phylogenetic graphs are contained in the package *jloda.phylo*. Furthermore, the library provides an interactive interface for the visualization of graphs contained in *jloda.graphview*. The primary class for the visualization of a graph is *jloda.graphview.GraphView*, which extends the Java core object *java.awt.JPanel*. Each GraphView has exactly one graph associated with

it. Jloda itself not only provides the foundations for the visualization, but also the interactive access to it. Furthermore, the library contains implementations of some standard phylogenetic drawing algorithms contained in *jloda.phylo*.

Jloda provides two primary elements: the implementation of graphs and an implementation for their interactive visualization. This substructure of the library is not only reflected by the breakdown into two distinct packages, but also by the dependency of the two classes Graph and GraphView upon each other. Every graph class in Jloda has its own coordinate system (*Graph World*), which is independent of the coordinate system of the GraphView class (*Device World*). The Graph World can be mapped onto the Device World and vice versa, using the class *jloda.util.Transform*. This way of implementing the connection between these two elements has the benefit that a developer working on visualization algorithms can focus on the Graph class, whereas a developer interested in developing a different view can focus on the GraphView class. Another benefit is that the (costly) calculation of the layout is independent of its visualization, thus changing the size, rotation or mirroring of the Device World does not imply the recalculation of the layout, but only adaption of the Transform class.

The conceptual integration of the glyphs into the program operation is shown in Figure 7.2. All glyphs are deduced from a shape (usually *java.awt.geom.Rectangle2D*) and contain an internal transformation of the shape onto the *Affine Glyph World*. This internal transformation allows, as the name suggests, the glyph to be affinely transformed and is designed so that it allows the Affine Glyph World to be mapped onto the Graph World via an identity mapping. Accordingly, one can use the class Transform already included in Jloda to map a glyph onto the Device World. In return, any user interaction with a glyph in the Device World is inversely mapped onto the Affine Glyph World, allowing the internal transformation to be adapted according to the changes within the Affine Glyph World. An important advantage of this approach is the simple representation of a glyph as a string. Only the underlying dimensions of the shape, the configuration of the internal transformation and the appearance of the glyph have to be saved.

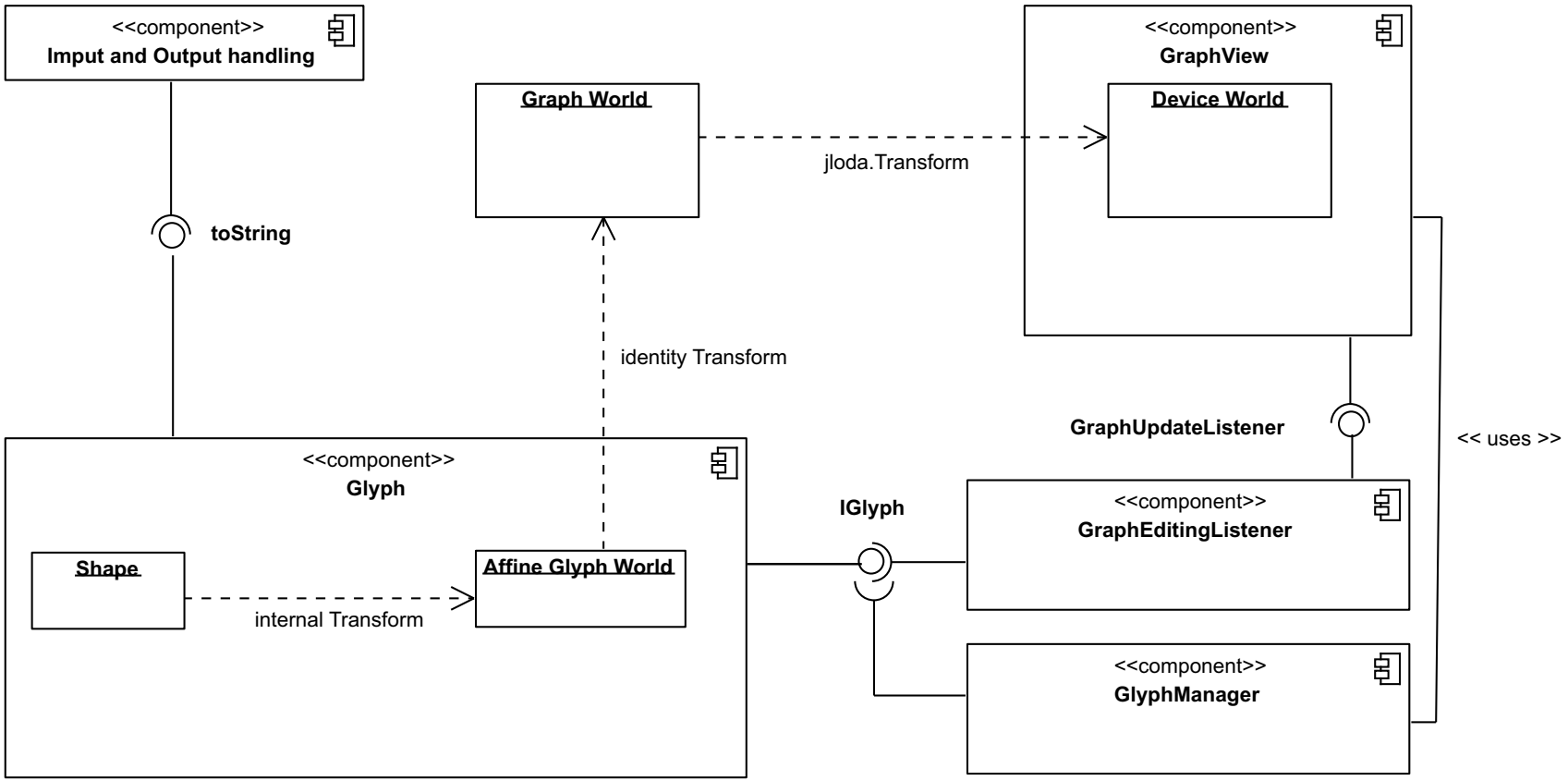


Figure 7.2: **Component Diagram of the Integration of Glyphs into Jloda** Glyphs are integrated into Jloda using two components. The GraphView is extended with the Glyph Manager. It manages the glyphs that are added to the GraphView. The GlyphEditingListener implements the GraphUpdateListener interface and provides the interactivity between the glyphs in the GraphView and the user. Both components use the IGlyph interface of the glyphs for access. All glyphs contain an internal transformation to map the underlying shape to the Affine Glyph World. The transformed shapes are mapped via an identity transformation to the Glyph World. The Glyph World is than mapped to the Device World using the Jloda.Transform object.

To separate the implementation of the glyph management from the GlyphView class, we have implemented the *jloda.graphview.glyphs.GlyphManager* class that centralizes the administration. Each GlyphManager class is associated with one GraphView class and is responsible for the access and modification of the glyphs contained in the GraphView. The GlyphManager class provides the necessary methods for the creation (*addGlyph()*), deletion (*deleteGlyph()*) and retrieval (*getGlyph()*) of a glyph. Additionally, the GlyphManager provides methods for selection (*setGlyphSelected()*) and deselection (*removeSelectedGlyph()*). Furthermore, one can change the appearance of selected glyphs, for example by using the GlyphManager methods *updateFontOfSelectedGlyphs()*, *updateLineColorOfSelectedGlyph()* or *updateFontDStyleOfSelectedGlyphs()*. The default settings for adding new glyphs can be altered using the methods *setFont()*, *setFontStyle()*, *setLineWidth()*, *setArcType()* etc.. The default settings can be restored using the methods *resetGlyphStyle()*.

Each glyph supports different visualizations and ways to modify its appearance. Different appearances can be set using the method *setSelectedGlyphDrawMode()*; selection of the next supported form uses *setNextSelectedGlyphDrawMode()*.

The order in which glyphs are drawn can be changed using the methods *decreaseSelectedGlyphsLevel()*, *increaseSelectedGlyphsLevel()*, *setAbsoluteMinLevelForSelectedGlyphs()* and *setAbsoluteMaxLevelForSelectedGlyphs()*. Furthermore, one can retrieve glyphs ordered by their level using the methods *getGlyphsSortedByLevel()* and *getSelectedGlyphsSortedByLevel()*. Glyphs that have been added to the graph are managed according to the interfaces they implement. A developer can use the GlyphManager to access the glyphs and does not need to care about the detailed properties each glyph implements.

Jloda allows adjustment of the interactivity of the GraphView surface by replacing the listener *jloda.graphview.GraphUpdateAdapter* with a class that implements the interface *jloda.graphview.GraphUpdateListener*. We have designed the class *GlyphEditingListener* that implements this interface and provides an interactive access to the glyphs. If this listener is activated, it allows a user to modify the glyphs interactively (e.g. scale, translate, resize, rotate). Details about modifying glyphs are discussed in Section 7.3.

Due to their design the glyphs present an independent extension of Jloda, which minimizes modifications to the core implementation.

7.2 Implementation Details

In this section, we describe the design and implementation of the glyphs and discuss some problems related to specific glyphs. One problem is the interface for user interaction with the glyphs, e.g. depending on the action of the user, the software must decide on how to change the glyph. To solve this problem, we decided to use specific buttons for each possible action, one for each change to the glyph. The glyph is then modified in reaction to button and mouse movements. This is a simple, yet effective way to provide the

necessary flexibility for glyph-drawing. The buttons are drawn directly onto the Java pane containing the phylogenetic graph. Furthermore, the buttons have to adapt to the transformations applied to the underlying glyph. Consequently we implemented our own buttons, which we describe in detail in Section 7.2. The more complex design of the glyphs is also described in Section 7.2. All classes related to this project are contained in the new package *jloda.graphview.glyphs*.

Buttons

Buttons are small graphics that are used to control the interaction of the user with the glyphs. To ensure this basic functionality, all buttons have to implement the interface *jloda.graphview.glyphs.IButton*. Methods that need to be implemented for a button are *draw()* (which draws the button using the Graphics2D Object provided), *hit()* (returns *true* if the given Point2D is contained in the boundary of the button), *setLocation()* (sets the location of the button), *getLocation()* (returns the location of the button) and *getStyle()* (returns the drawing style of a button, either *IButton.RECTANGLE* or *IButton.CYCLE*) methods. All buttons support both drawing styles and the default drawing style used is *IButton.CYCLE*, since the appearance is smoother than the *IButton.RECTANGLE* style. The size of a button depends upon its style and is set at compile time in the interface class using final integers. All implemented buttons are deduced from the base class *jloda.graphview.glyphs.ButtonBase* which implements the method *getTopLeftCorner()*. This method is used by all glyph drawing methods. We have implemented five different button types, and each button type has its own *draw()* method that provides a unique appearance. The class diagram of the buttons can be seen in Figure 7.3. Examples of the buttons can be seen in Figure 7.5

Glyphs

We have implemented glyphs for basic geometric figures such as *lines* (LineGlyph) *rectangles* (RectangleGlyph), *sectors of an oval* (Arc2DGlyph and the special case OvalGlyph) and *strings* (StringGlyph). All graphic elements are implemented on the basis of the java shape classes and generalized according to the special needs. As can be seen from Figure 7.4, glyphs can implement up to three different interfaces. The interface *Jloda.graphview.glyphs.IGlyph*, which assures the presence of the basic geometric operations, is mandatory for every glyph. Additionally, a glyph can implement the interface *Jloda.graphview.glyphs.IBoundableGlyph*, which allows the glyph to draw special objects at the end and start of the glyph as can be seen in Figure 7.5. The third interface, *Jloda.graphview.glyphs.IArcGlyph*, provides the possibility of interactively changing the internal arcs of glyphs as shown in Figure 7.5.

The methods of the interface IGlyph can be divided into four basic groups:

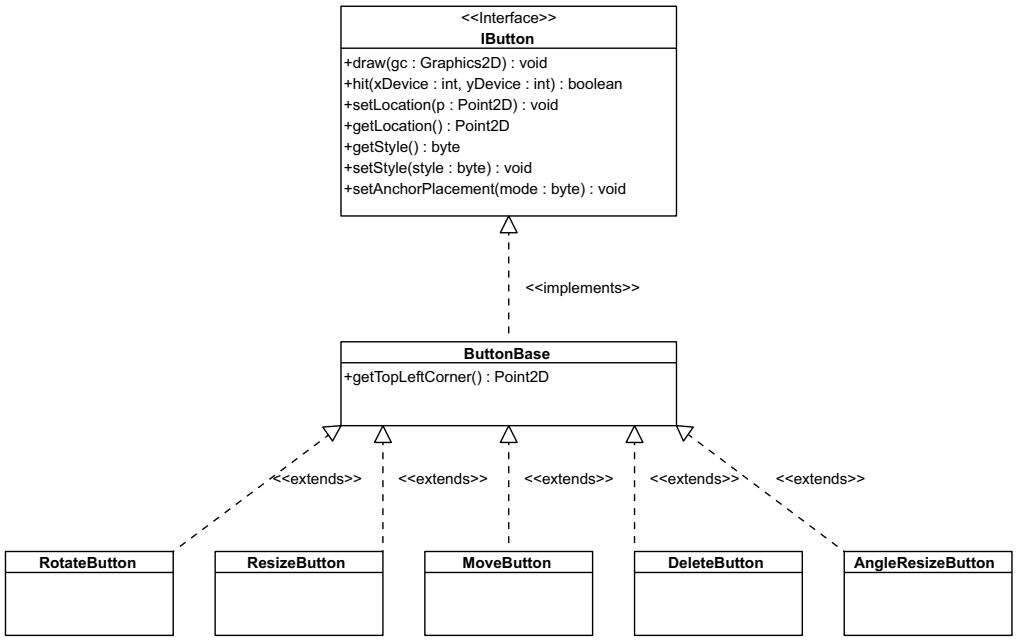


Figure 7.3: **Class Diagram of the Design of the Button Classes** All Buttons used in Jloda implement the IButton interface, which provides the basic functionality needed. The ButtonBase class is a basic implementation of the IButton interface. All Button classes extend this basic implementation.

- methods that assure the interactivity in the Device World: *translate()*, *resize()*, *rotateAbout()*, *hit()*, *getCenter()* and *getBounds()*;

- methods that map the glyph onto the Device World: *draw()*;

- methods that change the appearance of the glyph: *getLineWidth()*, *setLineWidth()*, *getLineColor()*, *setLineColor()*, *getFillColor()*, *setFillColor()*, *getLabelLocation()*, *setLabelLocation()*, *getFont()*, *setFont()*; and

- supporting methods: *getSupportedEditModi()*, *toString()*, *getLabel()* and *setLabel()*.

The methods *translate()*, *resize()* and *hit()* take values from the Device World and the Transform object as input. The methods *getCenter()* and *getBounds()* return the center and bounds of the glyph mapped from the Affine Glyph World to the Device World. The method *draw()* uses the given Transformation class to map the glyph to the Device World and draws it, using the given Graphics2D object.

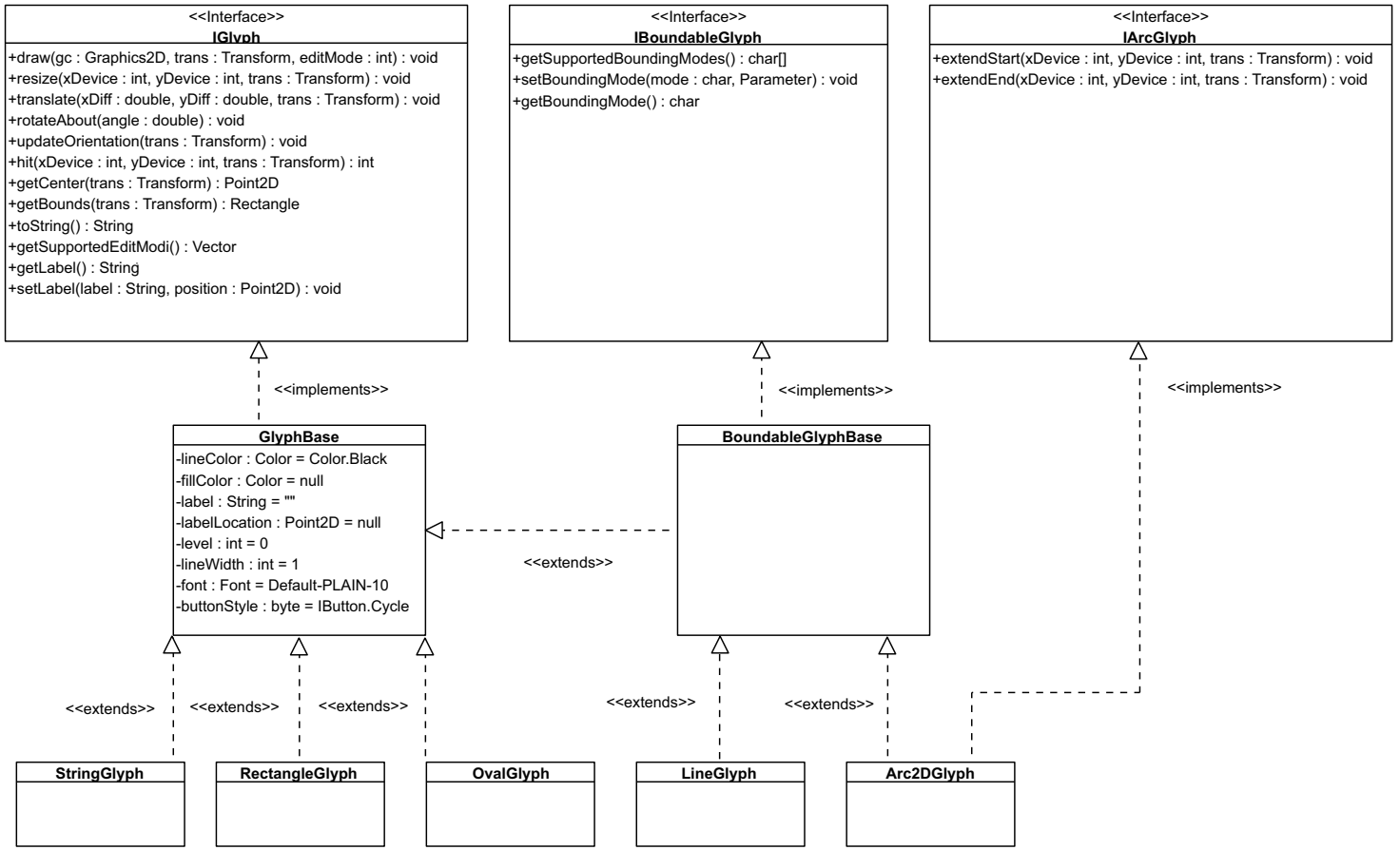


Figure 7.4: **Class Diagram of the Design of the Glyph Classes** Glyphs in Jloda can implement up to three different interfaces. The basic interface that must be implemented by all glyphs is the IGlyph interface, which provides the basic functionality. The BoundableIGlyph and IArcGlyph interfaces augment the glyphs with additional functionality. A basic implementation of the IGlyph interface is provided by the GlyphBase class, and the BoundableGlyphBase class is an extension of the GlyphBase class that also implements the IBoundableGlyph interface. The five glyphs implemented in Jloda either extend the GlyphBase or BoundableGlyphBase classes.

The Glyphs do not implement the interface directly, but rather extend the base class *jloda.graphview.glyphs.GlyphBase* ,which provides the functionality to change the appearance of the glyph. The basic appearance properties of a glyph are *lineWidth*, *lineColor* (defining the border appearence of the glyph), *fillColor* (defining the background color), *label*, *labelLocation* (a label of the glyph and its location in the Affine Glyph World), *font* (the font of the label), *buttonStyle* (the style of the editing buttons, which is either *IGlyh.CIRCULAR* or *IGlyph.RECTANGLE*) and *level*. The level of a glyph defines the order in which the glyphs and the graph are drawn. The graph has the fixed level 0 and the level of a glyph can be any integer value. The *draw()* method contained in GraphView, draws all glyphs in the order given by their levels, starting at the minimum level. As soon as it reaches level 0 it draws the graph and then continues with glyphs that have a higher level.

In addition to the IGlyph interface, the LineGlyph and Arc2DGlyph also implement the interface IBoundableGlyph, which allows the end points of the glyphs to be drawn as either orthogonal lines, circles, squares or arrowheads. This is well suited to, for example marking taxonomic groups within the graph. An example is given in Figure 7.5. In addition, to the IBoundableGlyph interface, the Arc2DGlyph class also implements the interface IArcGlyph, allowing the user to change the start and extend angle of the arc interactively.

In general, the transformations allow a user to mirror glyphs, with the obvious exception of the class StringGlyph. Instead of mirroring strings (making them unreadable), we have implemented a special extension in the draw() method of this class. When a mirroring is performed by the Transform class, the method calculates the relative rotation of the glyph in the Device World from the bounding rectangle and rotates the string accordingly. This modification ensures that the string is readable at all the time. An example of the mirroring of a string can be seen in Figure 7.5.

Each glyph contains a set of possible drawing modes (*IGlyph.ACTIVE*, *IGlyph.INACTIVE*, *IGlyph.RESIZEANGLE*), allowing the glyph to adjust its appearence depending upon the mode that is selected when calling the *draw()* method. The default mode is IGlyph.INACTIVE. If the mode is changed to IGlyph.ACTIVE, the user can edit the size, rotation and location of the glyph or delete it. The mode IGlyph.RESIZEANGLE is used to support the IArcGlyph interface and allows the user to resize the angle of the Arc2DGlyph object, as can be seen in Figure 7.5.

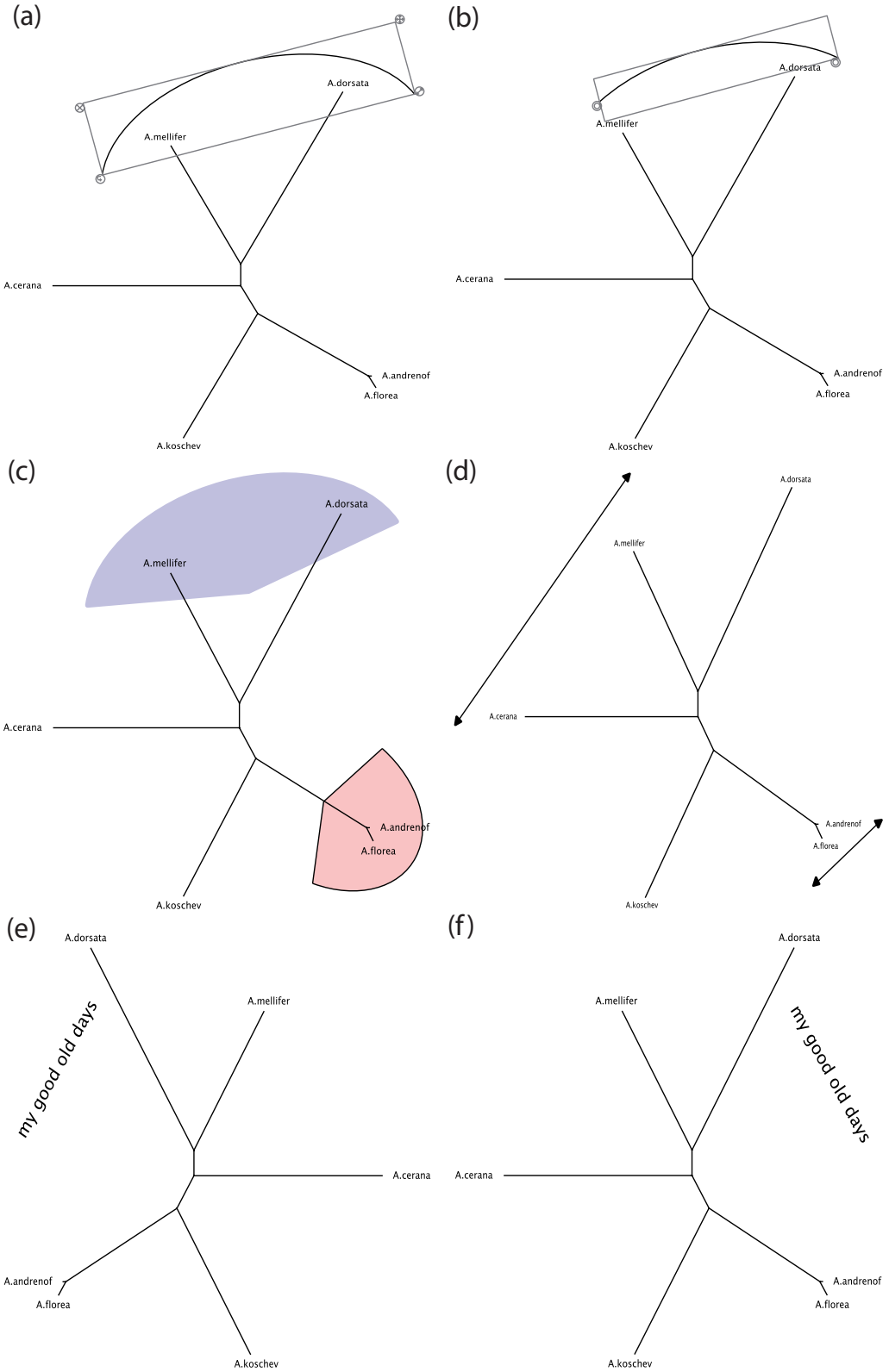


Figure 7.5: **Example of Glyphs** Figures (a) and (b) show an example of a selected Arc2DGlyph. The glyph shown in Figure (a) is rotated around its center so that it fits the two taxa labels. The start and extend angle of the glyph shown in Figure (b) has been altered to fit the two taxa labels better. The two Arc2DGlyphs shown in Figure (c) have been altered with different styles. Figure (d) shows two LineGlyphs with arrowheads at their ends. In the last two figures ((e) and (f)), we show a string before and after mirroring.

7.3 Integration of Glyphs into SplitsTree 4

We have integrated glyphs into SplitsTree 4 using two different approaches. The first part of the integration allows the user to manually add, edit and remove glyphs by hand using a GUI, and the second part is an automatic visualization algorithm for annotated groups within a phylogenetic tree or network.

The interaction of the user with the glyphs within SplitsTree 4 is controlled by two software elements. The first element SplitsTree uses is the GlyphUpdateListener described in Section 7.1, to allow the user to change the size, rotation and location of the glyph or to delete it. To create new glyphs and to change the visual properties of the glyphs, we have implemented a GUI that is shown in Figure 7.6. When the user activates this GUI, the GraphUpdateListener that is used by the GraphView is stored and replaced by the GlyphUpdateListener. The glyphs can then be edited using the mouse for activation and reshaping, and the GUI for changing their appearence.

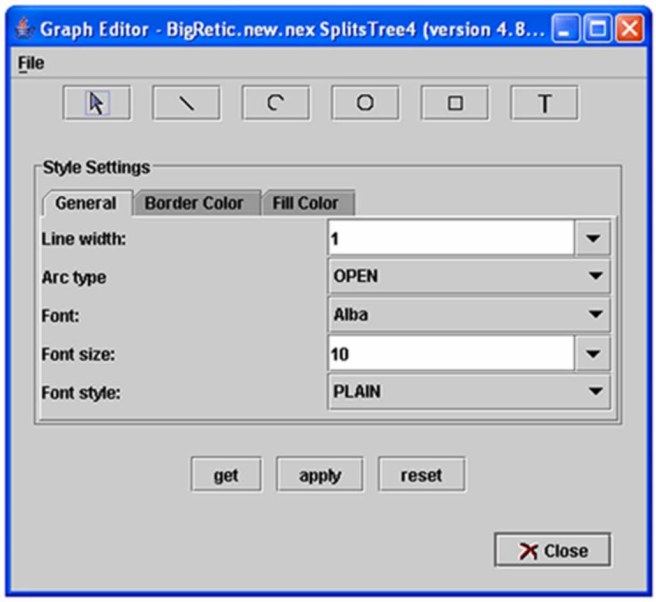


Figure 7.6: **The Graph Editor for the Integration of Glyphs into a GraphView** The top six buttons of the Graph Editor are used to create the different glyphs. The style settings of a selected glyph can be altered using the middle panel of the GUI. The *Get* button obtains the style settings of a glyph, the *Apply* button changes the style of selected glyphs according to the given style, and the *Reset* button resets all glyph properties to their default values.

We implemented a special GUI to interface the automatic visualization algorithm [Sch06]. The *TaxaSet Viewer* is used to group sets of taxa in a hierarchical fashion. The visualization algorithm uses the created hierarchy to extend the given GraphView with glyphs. The idea of the algorithm is to find the best fitting arc to be drawn around the positions of a set of taxa. This is done by finding the optimal placement of a cycle around the taxa vertices and truncate it to an arc at the positions of the first and the last vertex of the designated set. The algorithm requires a circular ordering of

all taxa. For instance, the Equal Angle Algorithm provides this property by assigning each taxon a unique angle.

In the first step, the algorithm extracts all groups from the hierarchy each group is assigned the level it has in the hierarchy. The level is used later in the algorithm to shift groups of taxa outwards to avoid collisions with the arcs of overlapping groups. In the second step, the vertices are sorted according to the given cyclic ordering. If the relevant taxa of a marked group are not consecutive, they are split into consecutive subgroups. In the next step, the initial and final vertex of a set with respect to the circular ordering is extracted, and the optimal arc for the set of taxa is calculated. In the last step, the algorithm optimizes the arcs and creates the glyphs according to the calculated settings. Details about calculating the arc can be found in [Sch06].

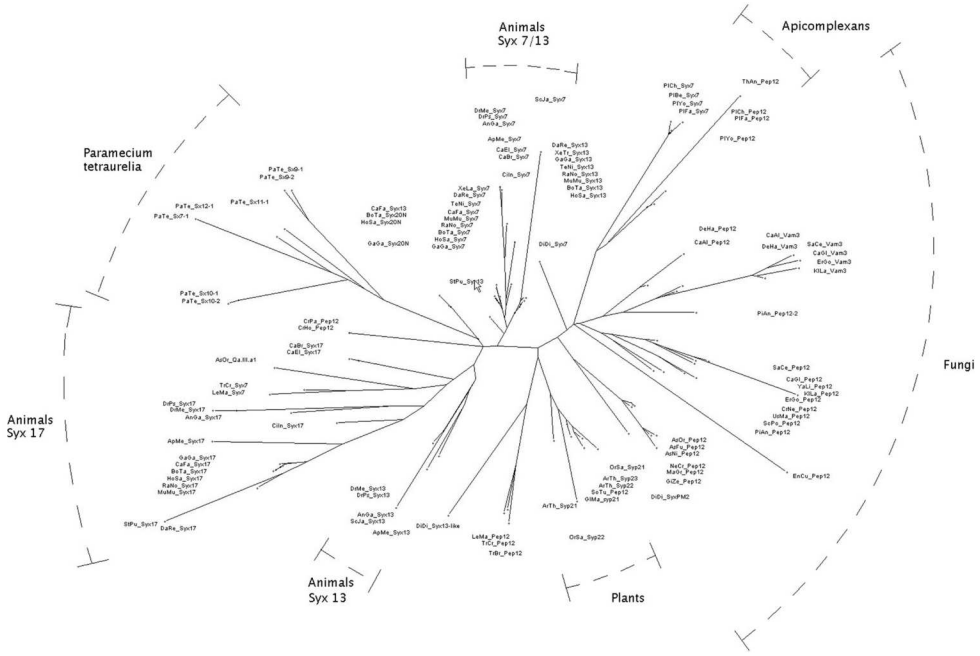


Figure 7.7: **Example of the Automatic Annotation Algorithm** This shows the phylogenetic tree discussed in the introduction of Chapter 7.1. The annotations have been drawn using the automatic annotation algorithm.

Chapter 8

Discussion

The decomposition theorem is an important tool for the efficient computation of minimal reticulate networks. We have given the proof that a one-to-one correspondence exists between the netted components of a splits network and the maximal two-connected components of a minimal (eventually degenerated) galled network. An interesting property of this decomposition theorem is that it only holds for a rather restrictive definition of minimality. If the minimality would be relaxed, for example, by not minimizing the pairs of dependent reticulation in the galled network, the decomposition theorem is no longer valid as the example in Figure 8.1 shows. Most interestingly, the special properties of a galled network imply that minimizing the number of reticulations and minimizing the number of pairs of dependent reticulations are not mutually exclusive. In general, this does not hold and it is a challenging question whether the decomposition theorem holds for either one of these two minimization objectives.

Another interesting facet of our work is that minimizing the number of edges within the network leads to a maximal depth of degenerated edges of one. The set of clusters that can be sampled from a degenerated edge is the set of all possible combinations of the directly descending reticulations. Consequently, it seems important to more clearly understand the role of these degenerated edges in the evolutionary process and the conflicting signals that can be sampled from these.

If we wish to alter the calculation algorithm given in Chapter 4 in such a way that it applies to a degenerated galled network, we have to account for a group of problems. The first one would be the placement of degenerated edges within the backbone tree, which seems to be a complex process. Furthermore, allowing for degeneration within a galled network, makes it clear that dissolving the connecting vertices of a degenerated galled network is a mandatory step in the reconstruction, rather than optional, as it is for the non-degenerated galled network.

Because of the minimality of a galled network, the influence of a false negative (or missing) split would most probably not result in an incorrect solution. Nevertheless, it would be interesting to see how a reticulate network changes when the given set of splits is shrinking. Another facet is formulating minimal generating sets for reticulate networks, or at least for galled networks, which could provide valuable insights into the combinatorial

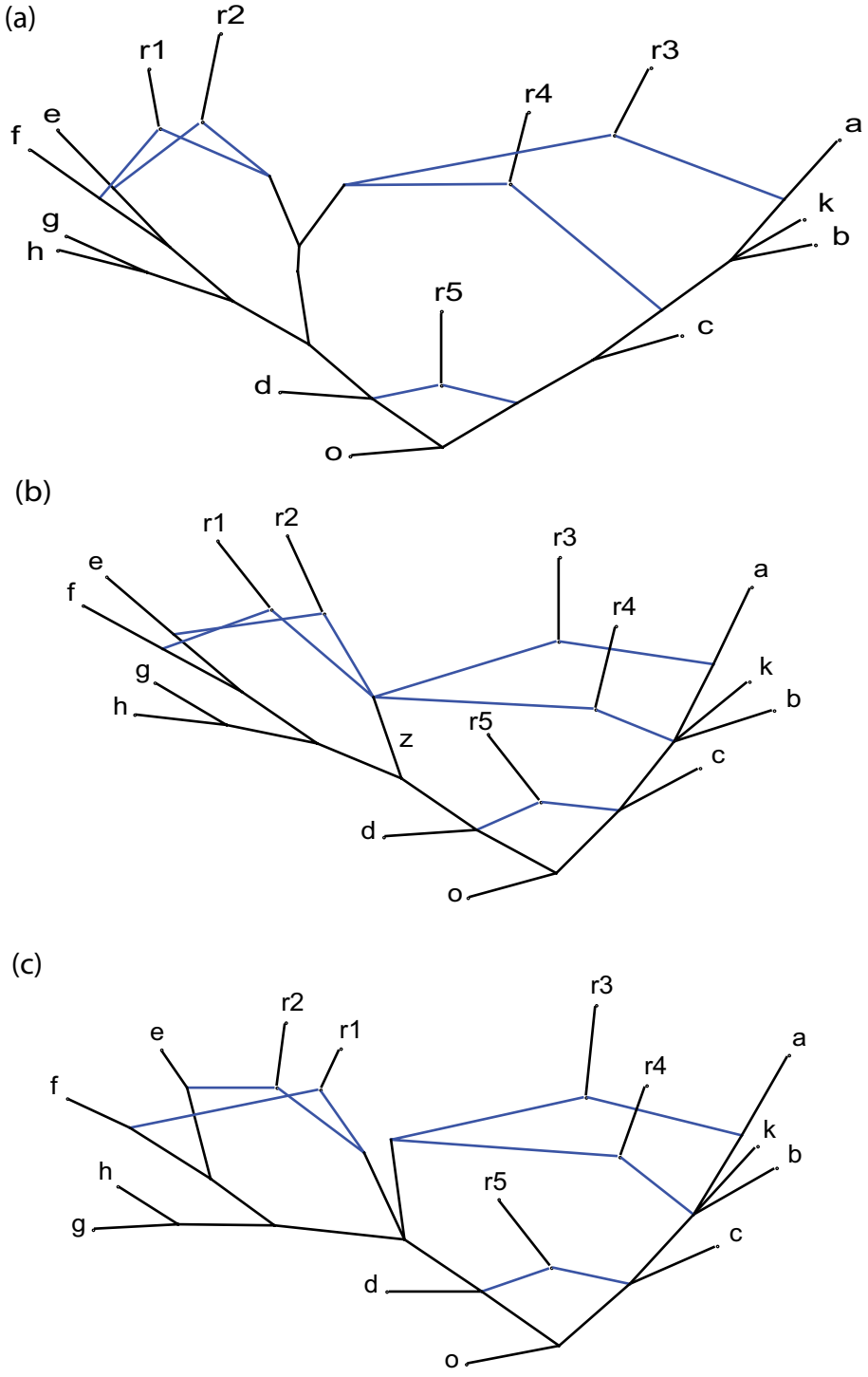


Figure 8.1: **Minimal Galled Networks** Three different galled networks are shown. The galled network in (a) is not minimal. If we sample $\mathcal{H}(N)$ from (a), we see that (b) is a minimal representation of $\mathcal{H}$. If the set of clusters $\mathcal{H}(z)$ of the edge z would comprise the two clusters $X_{r_1} \cup X_{r_2}$ and $X_{r_3} \cup X_{r_4}$ only, (b) is no longer minimal, but (c) is. The only difference between (b) and (c) is that the number of pairwise dependent reticulations is smaller in (c) than it is in (b). In fact, this is a counter-example to the *Decomposition Theorem* in the case where we only minimize the number of reticulations and the number of edges (as defined in [HK07]), since the cluster $X_{r_1} \cup X_{r_2}$ is not incompatible with any cluster that is an element of the tree cycle of r_3 or r_4.

structure of these graphs. More importantly, the problem of false positive information within a given set of splits seems to have a greater influence on the reconstruction process. There are two ways to address this problem: filtering false positive splits from the input set before reconstruction begins, or identifying false positive splits while reconstructing the network. There are some solutions available for the first approach (for example, see [HSW06]), but so far we have not seen a solution for the second one. Finally, the reconstruction may lead to more than one minimal solution. Ordering the given solutions by some likelihood would be an considerable advantage.

One aim of this thesis was getting better access to reticulate network methods for users. We believe that SplitsTree 4 is perfectly suited to provide this. Not only does SplitsTree 4 already provides a variety of methods for calculating splits networks, but the application is also designed in such a way that it allows new methods to be integrated easily. Nevertheless, for an successful integration, we had to develop a visualization method. The algorithm we presented in this thesis solves this problem. Together with the integration of the reticulate Nexus class into SplitsTree 4, we have implemented an easy-to-use and powerful tool for reticulate networks. We believe that modifying the graph with auxiliary edges is the best way to solve the computational problem. It enables us to formulate the problem as a local optimization, for which we have given a fast and accurate greedy solution. Nevertheless, we believe that developing further local optimization strategies may result in even better solutions. It might be interesting to calculate the optimal solution, since the problem is small in most cases. Furthermore, it could be of some interest to choose different weights for the crossings, depending upon the relative length of the subtrees. Finally, we believe that extending the algorithm to un-rooted reticulate networks may also prove an interesting topic.

Developing a Plugin Management System for SplitsTree 4 is another step aimed at providing better access to reticulate networks. Even though the system is applicable to any phylogenetic method for SplitsTree 4, the development of this application allows us to distribute newly developed method for SplitsTree 4 more easily. Basically, the Plugin Management System is only rounding off the underlying software, since SplitsTree 4 has always been able to integrate new method dynamically. The only part missing in SplitsTree 4, in this connection, was the ability for users to find and manage newly available methods easily. Since SplitsTree 4 is not the only software in our research group that uses Java-based plugin technology, we focused on developing a system that was as independent of the underlying software as possible. We believe that our system represents a good balance between software independency and usability. Further work that could be of interest is an automatic update mechanism for SplitsTree 4 itself, allowing users to update their version of the application dynamically.

Since the reconstruction of large phylogenetic trees and networks has become a common technique, visualizing the substantial elements within such a large phylogenetic graph is an important element of phylogenetic analysis. In general, this annotation is done by hand and can take up large amounts of time. Developing an automatic annotation algorithm for phylogenetic trees

seemed to be an interesting subject. The foundation for such an integration was the availability of basic geometrical shapes and text in the visualization library. Unfortunately, integrating such elements into a graph library is a complex task. The solution we presented in this thesis provides the user of the Jloda library with the ability to add this type of annotation as additional elements into their phylogenetic graph. Furthermore, we provided an algorithm that is able to annotate a given taxonomy into the phylogenetic graph, thus providing fast and accurate insights into the structure. The main problem with such an annotation is its quality. Consequently, we provided the user with the possibility to edit the added elements by hand, changing their size, rotation, color etc.. Having provided the underlying infrastructure for an automatic annotation tool, it is an interesting next step to provide further algorithms for the this task.

Appendix A

Publications

A.1 Published Manuscripts

1. Daniel H. Huson and Tobias H. Kloepper. **Beyond Galled Trees - Decomposition and Computation of Galled Networks**. Proceedings of RECOMB 2007, Lecture Notes in Computer Science, 2007, volume 4453, pages 211-225.

   > Reticulate networks are a type of phylogenetic network that are used to represent reticulate evolution involving hybridization, horizontal gene transfer or recombination. The simplest form of these networks are galled trees, in which all reticulations are independent of each other. This paper introduces a more general class of reticulate networks, that we call galled networks, in which reticulations are not necessarily independent, but may overlap in a tree-like manner. We prove a Decomposition Theorem for these networks that has important consequences for their computation, and present a fixed-parameter-tractable algorithm for computing such networks from trees or binary sequences. We provide a robust implementation of the algorithm and illustrate its use on two biological datasets, one based on a set of three gene-trees and the other based on a set of binary characters obtained from a restriction site map.

2. Daniel H. Huson and Tobias H. Kloepper. **Computing Recombination Networks from Binary Sequences**. In Proceedings of ECCB 2005, Bioinformatics, 2005, volume 21, supplemental 2, pages ii 159-165.

   > Phylogenetic networks are becoming an important tool in molecular evolution, as the evolutionary role of reticulate events such as hybridization, horizontal gene transfer and recombination is becoming more evident, and as the available data is dramatically increasing in quantity and quality. This paper addresses the problem of computing a most parsimonious recombination network for an alignment of binary

sequences that are assumed to have arisen under the "infinite sites" model of evolution, with recombinations.

Using the concept of a splits network as the underlying data-structure, this paper shows how a recent method designed for the computation of hybridization networks can be extended to also compute recombination networks. The proposed approach is illustrated using a number of real biological datasets and the algorithm will be made available as part of the SplitsTree 4 program.

3. Daniel H. Huson, Tobias H. Kloepper, Pete J. Lockhart and Mike A. Steel. **Reconstruction of Reticulate Networks from Gene Trees**. Proceedings of RECOMB 2005, Lecture Notes in Computer Science, 2005, volume 3500, pages 233-249.

   One of the simplest evolutionary models has molecular sequences evolving from a common ancestor down a bifurcating phylogenetic tree, experiencing point-mutations along the way. However, empirical analyses of different genes indicate that the evolution of genomes is often more complex than can be represented by such a model. Thus, the following problem is of significant interest in molecular evolution: Given a set of molecular sequences, compute a reticulate network that explains the data using a minimal number of reticulations. This paper makes four contributions toward solving this problem. First, it shows that there exists a one-to-one correspondence between the tangles in a reticulate network, the connected components of the associated incompatibility graph and the netted components of the associated splits graph. Second, it provides an algorithm that computes a most parsimonious reticulate network in polynomial time, if the reticulations contained in any tangle have a certain overlapping property, and if the number of reticulations contained in any given tangle is bounded by a constant. Third, an algorithm for drawing reticulate networks is described and a robust and flexible implementation of the algorithms is provided. Fourth, the paper presents a statistical test for distinguishing between reticulations due to hybridization, and ones due to other events such as lineage sorting or tree-estimation error.

4. Tobias H. Kloepper and Daniel H. Huson. **Integration of explicit phylogenetic networks into SplitsTree 4**. Accepted to BMC Evolutionary Biology.

   SplitsTree provides a framework for the calculation of phylogenetic trees and networks. It contains a wide variety of methods for the import/export, calculation and visualization of phylogenetic information. The software is developed

in Java and implements a command line tool as well as a graphical user interface.

In this article we, present solutions to two important problems in the field of phylogenetic networks. The first problem is the visualization of explicit phylogenetic networks. To solve this, we present a modified version of the equal angle algorithm that naturally integrates reticulations into the layout process and thus leads to an appealing visualization of these networks. The second problem is the availability of explicit phylogenetic network methods for the general user. To advance the usage of explicit phylogenetic networks by biologists further, we present an extension to the SplitsTree framework that integrates these networks. By addressing these two problems, SplitsTree is the first program that incorporates *implicit* and *explicit* network methods with standard phylogenetic tree methods in a graphical user interface environment.

A.2 Other Published Manuscripts

1. David Bryant, Daniel H. Huson, Tobias H. Kloepper and Kay Nieselt-Struwe. **Distance Corrections on Recombinant Sequences** . Proceedings of WABI 2003, Lecture Notes in Computer Science, 2003, volume 2812, pages 271-286.

   Sequences that have evolved under recombination have a lsquomosaicrsquo structure, with different portions of the alignment having evolved on different trees. In this paper we study the effect of mosaic sequence structure on pairwise distance estimates. If we apply standard distance corrections to sequences that evolved on more than one tree then we are, in effect, correcting according to an incorrect model. We derive tight bounds on the error introduced by this model mis-specification and discuss the ramifications for phylogenetic analysis in the presence of recombination.

2. Daniel H. Huson, Tobias Dezulian, Tobias H. Kloepper and Mike Steel. **Phylogenetic Super-Networks from Partial Trees**. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2004, volume 1, pages 151-158.

   In practice, one is often faced with incomplete phylogenetic data, such as a collection of partial trees or partial splits. This paper poses the problem of inferring a phylogenetic super-network from such data and provides an efficient algorithm for doing so, called the Z-closure method. Additionally, the questions of assigning lengths to the edges of the network, and how to restrict the “dimensionality” of the

network, are addressed. Applications to a set of five published partial gene trees relating different fungal species, and to six published partial gene trees relating different grasses, illustrate the usefulness of the method and an experimental study confirms its potential. The method is implemented as a plug-in for the program SplitsTree4.

3. Tobias H. Kloepper, C. Nickias Kienle and Dirk Fasshauer. **An elaborate classification of SNARE proteins sheds light on the conservation of the eukaryotic endomembrane system**. Molecular Biology of the Cell, 2007, volumne 18(9), pages 3463-3471.

   Proteins of the SNARE (soluble N-ethylmalemide-sensitive factor attachment protein receptor) family are essential for the fusion of transport vesicles with an acceptor membrane. Despite considerable sequence divergence, their mechanism of action is conserved: heterologous sets assemble into membrane-bridging SNARE complexes, in effect driving membrane fusion. Within the cell, distinct functional SNARE units are involved in different trafficking steps. These functional units are conserved across species and probably reflect the conservation of the particular transport step. Here, we have systematically analyzed SNARE sequences from 145 different species and have established a highly accurate classification for all SNARE proteins. Principally, all SNAREs split into four basic types, reflecting their position in the four-helix bundle complex. Among these four basic types, we established 20 SNARE subclasses that probably represent the original repertoire of a eukaryotic cenancestor. This repertoire has been modulated independently in different lines of organisms. Our data are in line with the notion that the ur-eukaryotic cell was already equipped with the various compartments found incontemporary cells. Possibly, the development of these compartments is closely intertwined with episodes of duplication and divergence of a prototypic SNARE unit.

Appendix B

Contributions

1. **Chapter 3- Decomposing Galled Networks**.

   The fundamental ideas in this chapter was published in the RECOMB2007 article by Daniel Huson and myself [HK07]. The proof presented in this thesis is a generalization of the one presented there. The idea and structure of the proof are my own. However, Regula Rupp has greatly influenced the writing of the proof with her deep understanding of the mathematical problems related to it.

2. **Chapter 4- Calculating Galled Networks**.

   The design and implementation have been inspired by our first article on this field [HKLS05]. The idea of how to label the resulting minimal galled network with sequences and mutations was developed in collaboration with Daniel Huson.

3. **Chapter 5- Drawing Explicit Phylogenetic Networks**.

   This work started in 2005, when Phillippe Gambette visited our group and I started discussing this problem with him. The solutions we discussed never resulted in a working visualization. Nevertheless, it gave me the idea of the algorithm presented here. Thanks to Daniel Huson's deep understanding of SplitsTree, he was able to assist me in the integration of the Reticulate Nexus class into the application.

4. **Chapter 6- A Plugin Management System for Java**.

   The Plugin Management System was first implemented in the diploma thesis of Sabine Luff [Luf07]. Unfortunately, the implementation was unstable and, in many parts, not consistent. Consequently, I redesigned and reimplemented the complete system and kept only the layout of the GUIs, for which I had given specific instructions. The web page of the project was designed with and implemented by C. Nickias Kienle.

5. **Chapter 7- Annotation of Phylogenetic Graphs using Jloda**.

   The system was designed with Daniel Huson. The implementation was done by myself. The automatic annotation algorithm was designed with and implemented by Andreas Schmidt [Sch06].

Bibliography

[BB04] V. Bafna and V. Bansal. The number of recombination events in a sample history: conflict graph and lower bounds. *IEEE/ACM Transactions in Computational Biology and Bioinformatics*, 1(2):78–90, 2004.

[BB05] V. Bafna and V. Bansal. Improved recombination lower bounds for haplotype data. In *Proceedings of the Ninth International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 569–584, 2005.

[BD92] H.-J. Bandelt and A. W. M. Dress. A canonical decomposition theory for metrics on a finite set. *Advances in Mathematics*, 92:47–105, 1992.

[BFO+03] G.S. Brodal, R. Fagerberg, A. Östlin, C.N.S.Pedersen, and S.S. Rao. Computing refined buneman trees in cubic time. *Lecture Notes in Computer Science*, 2812:259–270, 2003. Springer Verlag.

[BS06] M. Bordewich and C. Semple. Computing the minimum number of hybridisation events for a consistent evolutionary history. submitted, 2006.

[BS07] M. Bordewich and C. Semple. Computing the hybridization number of two phylogenetic trees is fixed-parameter tractable. *IEEE/ACM Trans. Comp. Biol. and BioInf.*, 4(3):458–466, 2007.

[Bun71] P. Buneman. The recovery of trees from measures of dissimilarity. In F. R. Hodson, D. G. Kendall, and P. Tautu, editors, *Mathematics in the Archaeological and Historical Sciences*, pages 387–395. Edinburgh University Press, 1971.

[Dar59] Charles M.A. Darwin. *The Origin of Species - by Means of Natural Selection.* John Murray, 1859.

[Dar37] Charles M.A. Darwin. First notebook on transmutation of species, 1937.

[DB07] W. . Doolittle and E. Bapteste. Inaugural Article: Pattern pluralism and the Tree of Life hypothesis. *Proceedings of the National Academy of Sciences*, 104(7):2043–2049, 2007.

[DGL04] S. Eddhu D. Gusfield and C. Langley. The fine structure of galls in phylogenetic networks. *INFORMS J. of Computing Special Issue on Computational Biology*, 16(4):459–469, 2004.

[DGS07] V. Bafina D. Gusfield, V. Bansal and Y.S. Song. A decomposition theory for phylogenetic networks and incompatible characters. *J. Comput. Biol.*, 2007. In press.

[DH04] A. W. M. Dress and D. H. Huson. Constructing splits graphs. *IEEE/ACM Transactions in Computational Biology and Bioinformatics*, 1(3):109–115, 2004.

[DS04] T. Dezulian and M. A. Steel. Phylogenetic closure operations and homoplasy-free evolution. In *Proceedings of the meeting of the International Federation of Classification Societies (IFCS) 2004, ed. D. Banks, L. House, F. R. McMorris, P. Arabie, and W. Gaul*, pages 395–416. Springer-Verlag, Berlin, 2004.

[Fel04] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., 2004.

[GB05] D. Gusfield and V. Bansal. A fundamental decomposition theory for phylogenetic networks and incompatible characters. In *Proceedings of the Ninth International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 217–232, 2005.

[GEL03] D. Gusfield, S. Eddhu, and C. Langley. Efficient reconstruction of phylogenetic networks with constrained recombination. In *Proceedings of the 2003 IEEE CSB Bioinformatics Conference*, pages 173–213, 2003.

[GH04] D. Gusfield and D. Hickerson. A new lower bound on the number of needed recombination nodes in both unrooted and rooted phylogenetic networks. *Technical Report ICD-ECS-06, University of California, Davis*, 2004.

[GJ83] M.R. Garey and D.S. Johnson. Crossing number is np-complete. *SIAM J. Alg. Discr. Math.*, 4:312–316, 1983.

[GM96] R. C. Griffiths and P. Marjoram. Ancestral inference from samples of DNA sequences with recombination. *J. Computational Biology*, 3:479–502, 1996.

[Hae74] Ernst H.P.A. Haeckel. *Anthropogenie oder Entwicklungsgeschichte des Menschen*. Wilhelm Engelmann, Leipzig, 1874.

[HB06] D. H. Huson and D. Bryant. Application of phylogenetic networks in evolutionary studies. *Molecular Biology and Evolution*, 23:254–267, 2006. Software available from `www.splitstree.org`.

[HDKS04] D. H. Huson, T. Dezulian, T. Kloepper, and M. A. Steel. Phylogenetic super-networks from partial trees. *IEEE/ACM Transactions in Computational Biology and Bioinformatics*, 1(4):151–158, 2004.

[Hei93] J. Hein. A heuristic method to reconstruct the history of sequences subject to recombination. *J. Mol. Evol.*, 36:396–405, 1993.

[HK85] R. R. Hudson and N. L. Kaplan. Statistical properties of the number of recombination events in the history of a sample of DNA sequences. *Genetics*, 111:147–164, 1985.

[HK05] D.H. Huson and T.H. Kloepper. Computing recombination networks from binary sequences. *Bioinformatics*, 21(suppl. 2):ii159–ii165, 2005. ECCB.

[HK07] Daniel H. Huson and Tobias H. Kloepper. Beyond galled trees - decomposition and computation of galled networks. accepted to RECOMB2007, 2007.

[HKLS05] D.H. Huson, T. Kloepper, P.J. Lockhart, and M.A. Steel. Reconstruction of reticulate networks from gene trees. In *Proceedings of the Ninth International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 233–249, 2005.

[HLT04] M. Hallett, J. Largergren, and A. Tofigh. Simultaneous identification of duplications and lateral transfers. In *Proceedings of the Eight International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 347–356, 2004.

[HM03] B. Holland and V. Moulton. Consensus networks: A method for visualizing incompatibilities in collections of trees. In G. Benson and R. Page, editors, *Proceedings of "Workshop on Algorithms in Bioinformatics"*, volume 2812 of *LNBI*, pages 165–176. Springer, 2003.

[HSW06] D.H. Huson, M.A. Steel, and J. Whitfield. Reducing distortion in phylogenetic networks. In P. Bücher and B.M.E. Moret, editors, *Algorithms in Bioinformatics*, LNBI 4175, pages 150–161, 2006.

[Hud83] R. R. Hudson. Properties of the neutral allele model with intergenic recombination. *Theoretical Population Biology*, 23:183–201, 1983.

[KBR98] A. Kumar, W.C. Black, and K.S. Rai. An estimate of phylogenetic relationships among culicine mosquitoes using a restriction map of the rDNA cistron. *Insect Molecular Biology*, 7(4):367–373, 1998.

[Ket55] H.B.D. Kettlewell. Selection experiments on industrial melanism in the lepidoptera. *Heredity*, 9:323–342, 1955.

[Ket56] H.B.D. Kettlewell. Further selection experiments on industrial melanism in the lepidoptera. *Heredity*, 10:287–301, 1956.

[Kim69] M. Kimura. The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutation. *Genetics*, 61:893–903, 1969.

[KOC00] B. K. Tacke K. O'Donnell, H. C. Kistler and H. H. Casper. Gene genealogies reveal global phylogeographic structure and reproductive isolation among lineages of fusarium graminearum, the fungus causing wheat scab. *Proc. Natl. Acad. Sci. USA*, 97(14):7905–7910, 2000.

[LMH+01] P. J. Lockhart, P. A. McLenachan, D. Havell, D. Glenny, D. H. Huson, and U. Jensen. Phylogeny, dispersal and radiation of New Zealand alpine buttercups: molecular evidence under split decomposition. *Ann Missouri Bot Gard*, 88:458–477, 2001.

[LR04] C. R. Linder and L. H. Rieseberg. Reconstructing patterns of reticulate evolution in plants. *Am. J. Bot.*, 91(10):1700–1708, 2004.

[LSH05] Rune B. Lyngsø, Yun S. Song, and Jotun Hein. Minimum recombination histories by branch and bound. In *WABI*, pages 239–250, 2005.

[Luf07] S. Luff. Implementation of a plugin management system for splitstree. Master's thesis, University of Tuebingen, Center for Bioinformatics, 2007.

[LvAL04] M. Blum L. von Ahn and J. Langford. Telling humans and computers apart automatically, 2004.

[Mad97] W. P. Maddison. Gene trees in species trees. *Syst. Biol.*, 46(3):523–536, 1997.

[Maj02] Michael E.N. Majerus. *Moths*. HarperCollins(UK), 2002.

[Mea83] C. A. Meacham. Theoretical and computational considerations of the compatibility of qualitative taxonomic characters. In J. Felsenstein, editor, *Numerical Taxonomy*, volume G1 of *NATO ASI Series*. Springer, Berlin, 1983.

[Mer05] C. Mereschkowsky. über natur und ursprung der chromatophoren im pflanzenreiche. *Biol. Centralbl.*, 25:593–604, 1905.

[MG03] S. R. Myers and R. C. Griffiths. Bounds on the minimal number of recombination events in a sample history. *Genetics*, 163:375–394, 2003.

[MGL99] G. Burger M.W. Gray and B.F. Lang. Mitochondrial Evolution. *Science*, 283(5407):1476–1481, 1999.

[MGS76] D.S. Johnson M.R. Garey and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[MM06] M.M. Morin and B.M.E. Moret. Netgen: generating phylogenetic networks with diploid hybrids. *Bioinformatics*, 22(15):1921–1923, 2006.

[MSM97] D.R. Maddison, D.L. Swofford, and W.P. Maddison. NEXUS: an extendible file format for systematic information. *System. Bio.*, 46(4):590–621, 1997.

[NIS95] NIST. Secure hash standard. *Federal Information Processing Standard*, FIPS-180-1, 1995.

[NWL04] L. Nakhleh, T. Warnow, and C. R. Linder. Reconstructing reticulate evolution in species - theory and practice. In *Proceedings of the Eight International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 337–346, 2004.

[PB03] B. M. Pryor and D. M. Bigelow. Molecular characterization of Embellisia and Nimbya species and their relationship to Alternaria, Ulocladium and Stemphylium. *Mycologia*, 95(6):1141–1154, 2003.

[Rog92] A. Rogers. Error introduced by the infinite-site model. *Mol. Biol. Evol.*, 9(6):1181–1184, 1992.

[Sch83] A.F.W. Schimper. über die entwicklung der chlorophyllkörner und farbkörper. *Bot. Zeitung*, 41:105–114,121–131,137–146,153–162, 1883.

[Sch06] A. Schmidt. Taxa set highlighting for splitstree 4. Master's thesis, University of Tuebingen, Center for Bioinformatics, 2006.

[SDH+04] C. Semple, P. Daniel, W. Hordijk, R.D.M. Page, and M. Steel. Supertree algorithms for ancestral divergence dates and nested taxa. *Bioinformatics*, 20(15):2355–2360, 2004.

[SDJ04] M.T. Brown S.D. Dyall and P.J. Johnson. Ancient Invasions: From Endosymbionts to Organelles. *Science*, 304(5668):253–257, 2004.

[SDPE94] M. J. Sanderson, M. J. Donoghue, W. Piel, and T. Eriksson. Treebase: a prototype database of phylogenetic analyses and an interactive tool for browsing the phylogeny of life. *Amer. Jour. Bot.*, 81(6):183, 1994.

[SH04] Y.S. Song and J. Hein. On the minimum number of recombination events in the evolutionary history of DNA sequences. *J. Math. Biol.*, 48:160–186, 2004.

[SH05] Y.S. Song and J. Hein. Constructing minimal ancestral recombination graphs. *J. Comp. Biol.*, 12:147–169, 2005.

[SS01] C. Semple and M. A. Steel. Tree recontruction via a closure operation on partial splits. In *Computational Biology (proceedings of JOBIM 2000), LNCS 2066*. Springer-Verlag, 2001.

[SS03] C. Semple and M. A. Steel. *Phylogenetics*. Oxford University Press, 2003.

[SZ00] T. Sang and Y. Zhong. Testing hybrization hypotheses based on incongruent gene trees. *System. Biol.*, 49(3):422–424, 2000.

[THCS01] R. L. Rivest T. H. Cohen, C.E. Leiserson and C. Stein. *Introduction to Algorithms, second edition*. PWS, 2001.

[TKF07] C.N. Kienle T.H. Kloepper and D. Fasshauer. An elaborate classification of snare proteins sheds light on the conservation of the eukaryotic endomembrane system. *Mol. Biol. Cell*, 18(9):3463–3471, 2007.

[Wal58] Alfred R. Wallace. On the tendency of varieties to depart indefinitely from the original type. In *Proceedings of the Linnean Society of London*, volume 3, pages 53–62, 1858.

[WZZ01] L. Wang, K. Zhang, and L. Zhang. Perfect phylogenetic networks with recombination. *Journal of Computational Biology*, 8(1):69–78, 2001.

Lebens- und Bildungsweg

Tobias Heinz Klöpper, geboren am 27. Mai 1977 in Bremen

1983 - 1987	Besuch der Grundschule an der Robinsbalje
1987 - 1994	Besuch der Gesamntschule Hermannsburg
1994 - 1995	Besuch der Lapeer East High School (Michigan, USA)
06/1995	Honorary Diploma
1995 - 1997	Besuch des Schulzentrums Huchting
06/1997	Abitur
09/1997 - 02/2003	Studium der Mathematik an der Universität Göttingen
02/2003	Diplomarbeit mit dem Titel *Stationäre Prozesse und fast sichere Grenzwertsätze*
01/2001 - 06/2002	Angestellter bei Prof. Dr. Peter Gruss, Abteilung für Mollekulare Zellbiologie, Max-Plank-Institut für biophysikalische Chemie Göttingen, für Programmierarbeiten im Rahmen des AMGEN- Projekt
03/2003 - 01/2004	Wissenschaftlicher Mitarbeiter bei Dr. Kay Nieselt, Institut für Informatik, Arbeitsbereich Proteomics Algorithmen und Simulation
seit 02/2004	Wissenschaftlicher Mitarbeiter bei Prof. Dr. Daniel H. Huson, Universität Tübingen, Institut für Informatik, Arbeitsbereich Algorithmen der Bioinformatik. Anfertigung einer Dissertation mit dem Titel: *Algorithms for the Calculation and Visualisation of Phylogenetic Networks* Ko-Betreuung durch Prof. Dr. David Bryant, Universität Auckland