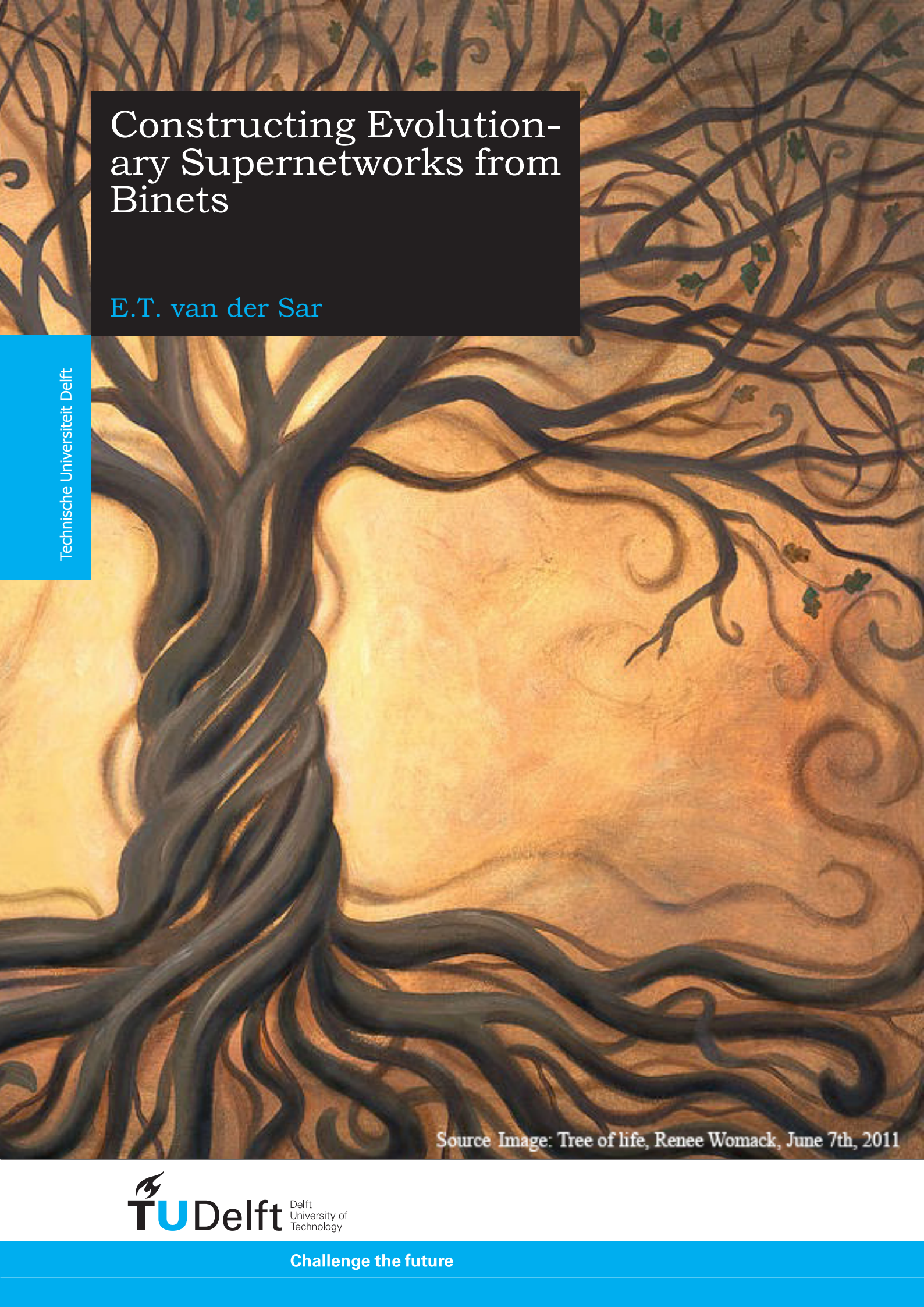




Constructing Evolutionary Supernetworks from Binets

E.T. van der Sar

Technische Universiteit Delft

Source Image: Tree of life, Renee Womack, June 7th, 2011

**Technische Universiteit Delft
Faculteit Elektrotechniek, Wiskunde en Informatica
Delft Institute of Applied Mathematics**

**Evolutionaire Supernetwerken Construeren vanuit
Binets**

**(Engelse titel: Constructing Evolutionary
Supernetworks from Binets)**

Verslag ten behoeve van het
Delft Institute of Applied Mathematics
als onderdeel ter verkrijging

van de graad van

**BACHELOR OF SCIENCE
in
TECHNISCHE WISKUNDE**

door

Erica van der Sar

**Delft, Nederland
Juli 2015**

BSc verslag TECHNISCHE WISKUNDE

“Evolutionaire Supernetwerken Construeren vanuit Binets”
(Engelse titel: “Constructing Evolutionary Supernetworks from Binets”)

ERICA VAN DER SAR

Technische Universiteit Delft

Begeleider

Dr.ir. L.J.J van Iersel

Overige commissieleden

Dr.ir. M.C. Veraar

Dr. ir. M. Keijzer

Juli, 2015

Delft

Abstract

Constructing the evolutionary history of species is a task with a lot of mathematics involved. Given a collection of small pieces of history we try to construct the bigger picture. The evolutionary history of species can be represented in a *phylogenetic tree*. For a given set of smaller phylogenetic trees a bigger *supertree* can be constructed. There are several algorithms that try to find a solution to construct a supertree that represents the given set of trees the best.

However, in some cases there are non-treelike evolutionary histories. These evolutionary histories have to be presented in a *phylogenetic network*. Constructing a *supernetwork* from smaller phylogenetic networks is a new field of research. Up to now most research has focused on tree-like networks, but recently a new algorithm was proposed to decide if there exists a supernetwork that contains all given *binets* (a type of small networks), and construct the supernetwork if so.

In this thesis this algorithm will be analyzed. Subsequently a new algorithm will be presented to construct a supernetwork from a given set of binets even if there exists no supernetwork that contains all given binets. To get some ideas and inspiration about constructing supernetworks, algorithms to construct supertrees are analyzed first.

In addition it will be proved that there always exists a supernetwork that contains at least one third of the given binets.

E.T. van der Sar
Delft, July 2015

Contents

1	Introduction	1
1.1	Phylogenetic Networks	1
1.1.1	Mathematical Definition	2
1.2	Challenge	3
2	Constructing Supertrees	5
2.1	Algorithm: BUILD	5
2.2	Algorithm: MinCutSupertree	8
2.2.1	ModifiedMinCut	10
2.3	Algorithh: Max Cut triplets	13
3	Constructing Supernetworks from Binets	17
3.1	Algorithm: BuildNW	17
3.2	When does the algorithm fail?	19
3.3	Algorithm: BuildNW2	24
3.4	Lower bound binets in supernetwork	31
4	Discussion	33
	Bibliography	37

1

Introduction

In biology the evolutionary tree or “tree of life” as Darwin called it is a very complicated piece of history. It is a tree that should capture all evolution of all species on earth. Even only in the Netherlands there live over 35.000 species; animals, plants, molds and other organisms. [1]

You can imagine that reconstructing such a “tree of life” can be a very difficult task, even if we just want to reconstruct a part of the history of only a few species. Mathematics appears to play an important role in the reconstruction of the history in an evolutionary tree.

A biological evolutionary tree can be displayed as a graph, as shown in Figure 1.1, and is called a phylogenetic tree.

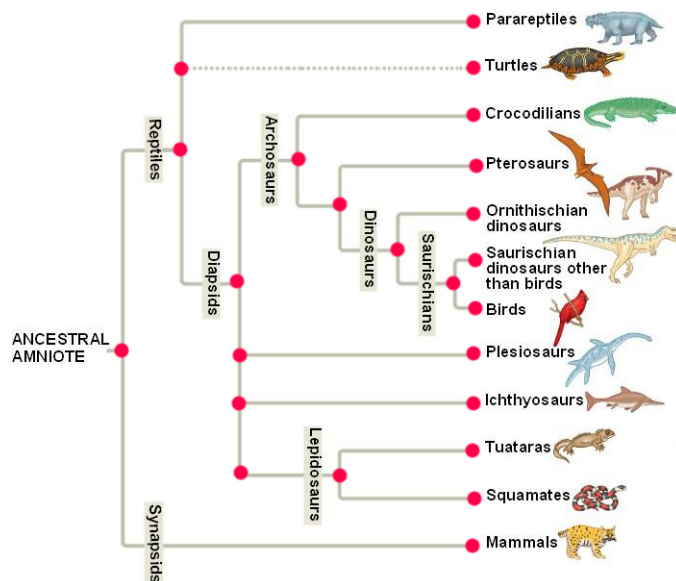


Figure 1.1: A phylogenetic tree with a vertex for each species and an edge between ancestors and direct descendants.[2]

In this chapter I will first explain why not all evolutionary history can be displayed in a phylogenetic tree, why a phylogenetic network sometimes is needed, what phylogenetic networks are (mathematical definition) and at last I will explain the problem that I am going to look at in the further chapters.

1.1. Phylogenetic Networks

By biological definition one species is different from another species if they cannot reproduce. This means that if a species separates into two or more different species, these different species cannot be

crossed into one new species. This would mean that the history of species can always be displayed by a tree.

But in practice there are some cases that make it a little bit more difficult. It appears that some evolutionary histories cannot be displayed in a phylogenetic tree. Through causes like horizontal gene transfer (HGT) and hybridization, it is possible that two different species exchange their DNA. In this way a new species can derive from two different ones. This results in an evolutionary history of species that cannot be caught by a tree, but has to be displayed by a network. This is called a phylogenetic network, see Figure 1.2 for an example.

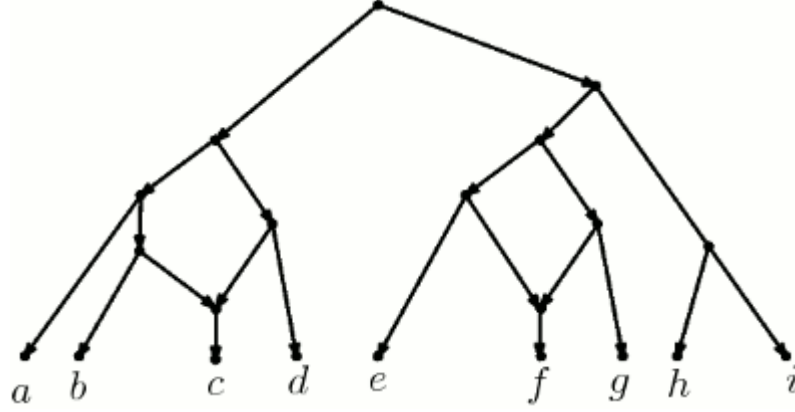


Figure 1.2: Example of a phylogenetic network. Species c is a descendant of ancestors of species b and d . Also species f is a descendant of two different species.[3]

1.1.1.1. Mathematical Definition

Before I am going to work with phylogenetic networks I will first give a proper mathematical definition.[4]

Definition 1.1.1. Let N be a **phylogenetic network** (or $\mathcal{X}$ -network) on a set $\mathcal{X} = \{a, b, \dots\}$. $\mathcal{X}$ is a finite set of labels, which can represent the species. We will refer to the elements of $\mathcal{X}$ as *taxa*.

N consists of a graph and a function. $N = (G, f)$, where

- $G = (V, E)$ is a *directed*, acyclic graph. The set V consists of different types of vertices: $V = \{r\} \cup V_L \cup V_T \cup V_N$, where
 - r is the *root* of N . $\text{indeg}(r) = 0$
 - V_L are the *leaves* of N . $\forall v \in V_L : \text{indeg}(v) = 1, \text{outdeg}(v) = 0$
 - V_T are the *tree-nodes* of N . $\forall v \in V_T : \text{indeg}(v) = 1, \text{outdeg}(v) > 1$
 - V_N are the *network-nodes* of N . $\forall v \in V_N : \text{indeg}(v) = 2, \text{outdeg}(v) = 1$

E is the set of edges between the nodes. These represent the relationships between the species.

- $f : V_L \rightarrow \mathcal{X}$ is a bijective *leaf-labeling* function.

Figure 1.3 will make this definition even more clear.

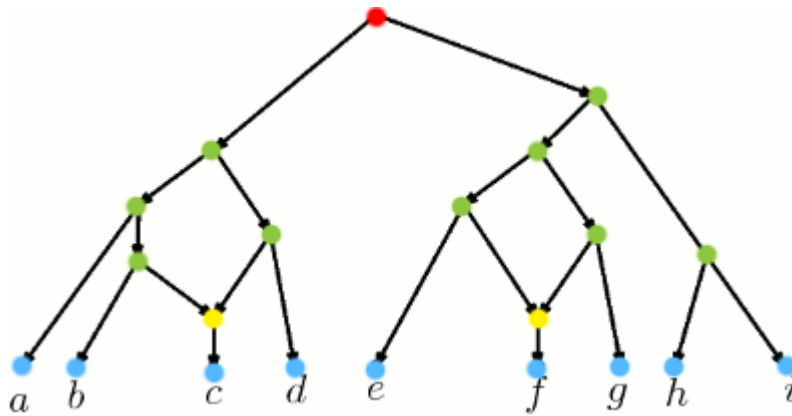


Figure 1.3: A phylogenetic network where the different types of vertices are indicated with a color. Red: $\{r\}$, green: V_T , yellow: V_N , blue: V_L .

Remark: A phylogenetic network with $V_N = 0$ is a phylogenetic tree.

In this report I will only focus on phylogenetic networks (or networks for short) that are called *binary level-1 networks*. [5] Binary means that all vertices have outdegree is at most two. A network is called binary level-1 if any two cycles are disjoint. Furthermore will all networks be single rooted (only one root), just like in Definition 1.1.1.

1.2. Challenge

Small networks with only two leaves are called *binets*. The challenge that I have been assigned to is:

"Given a set of binets, find a supernetwork that contains most of these binets."

For a pair of taxa $\{a, b\}$ there are three possible binets, shown in Figure 1.4. The first and second binet seem similar, but they represent something different. In the first binet b is a descendant from two different species and a is a descendant of only one species. In the second binet it is exactly the other way around. I will call the first case "a binet with b below a ", notation: $N(b; a)$.

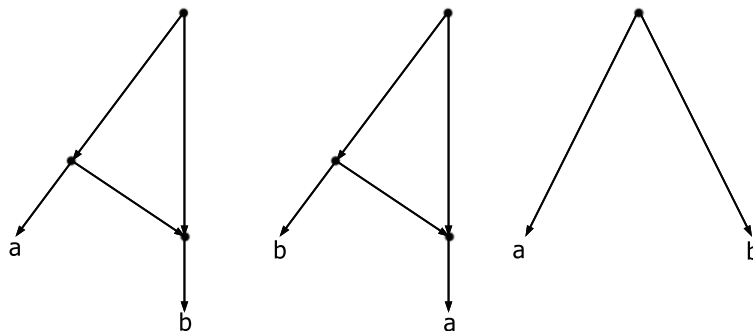


Figure 1.4: All possible binets with the leaf set $\{a, b\}$

To show that a large network consists of binets, I will give the following example:

Example 1.2.1. Suppose we want to derive the binets for the taxa sets $\{a, b\}$, $\{c, d\}$ and $\{f, h\}$ from the network in Figure 1.2. The first step is to find all directed paths from r to the set of taxa. See Figure 1.5a for all directed paths to each set.

The next step is to delete all paths above the *lowest stable ancestor*. The lowest stable ancestor (LSA) of a set of taxa $\{x, y\}$ is the lowest vertex $v \in N$ that is contained in every directed path from the root r to the leaves in $\{x, y\}$.

Now replace all pieces of paths where no network-nodes or tree-nodes are involved (edge-node-...-edge) by only one edge. In the resulting graph there can be some parallel edges. This is the

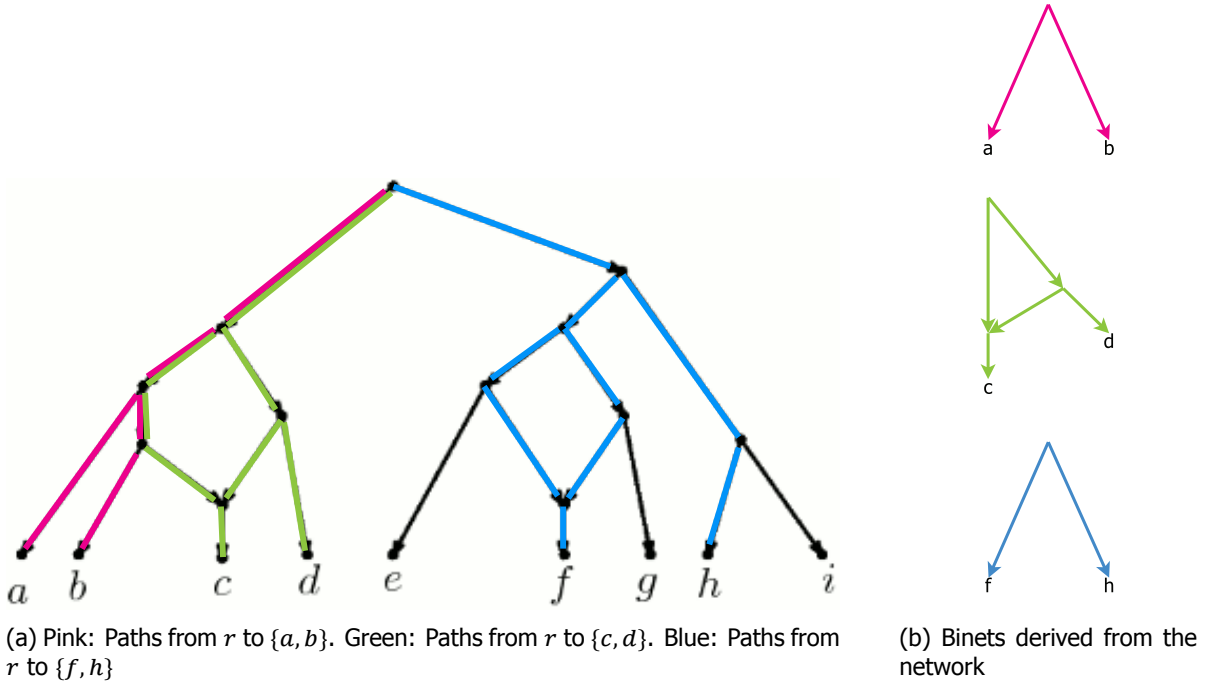


Figure 1.5

case if we try to find a binet for $\{f, h\}$. Contract everything with parallel edges. If this results again to pieces of paths were no network-nodes or tree-nodes are involved (edge-node-...-edge) replace these pieces by just one edge.

The binets for $\{a, b\}$, $\{c, d\}$ and $\{f, h\}$ derived from the network are shown in Figure 1.5b.

A general algorithm to find binets from a supernetwork is described in Algorithm 1.

Algorithm 1 Derive binets(N)

Input: A phylogenetic network $N = (G, f)$ where $G = (V, E)$

Output: Binet for each pair of taxa

- 1: **for** each pair $\{x, y\} \in N$ **do**
 - 2: $B\{x, y\} =$ all directed paths from the root r to x and from r to y
 - 3: $B\{x, y\} = B\{x, y\} \setminus$ everything above the LSA of x and y
 - 4: Replace $\rightarrow\rightarrow$ in $B\{x, y\}$ by $\rightarrow$
 - 5: Contract parallel edges in $B\{x, y\}$
 - 6: Replace $\rightarrow\rightarrow$ in $B\{x, y\}$ by $\rightarrow$
 - 7: **end for**
 - 8: **return** All binets $B\{x, y\}$
-

In Algorithm 1 the network has to be given and by running the algorithm the binets contained in the network can be found. My problem is exactly the other way around. I want to find an algorithm that for any given set of binets constructs a supernetwork containing most of the given binets.

One may wonder why, for a given set of binets, it would not be possible to construct a supernetwork containing all of the given binets. The answer is relatively simple: In the set of given binets there can be a binet $N(b; a)$ and a binet $N(a; b)$. In reality both cannot be true, because if b is a result of a , a cannot also be a result of b . This means one of the binets is clearly wrong. An interesting question is: why would the data contain wrong binets? This is because the binets cannot be generated with hundred percent certainty, thus the given data existing of binets can be unsure.

In Chapter 2 I will look at a well-studied problem that is similar to this one. In Chapter 3 I will get back to the problem described above.

2

Constructing Supertrees

In the phylogenetic literature a problem very similar to the problem in my research has been evaluated a lot. Namely:

"Given a set of triplets, find a supertree."

A *supertree* is a tree based on a given set of smaller trees, containing all taxa in the given trees. *Triplets* are trees with three leaves. For a set of taxa $\{a, b, c\}$ there are three possible triplets, see Figure 2.1. One of the possibilities is a tree where $\{a, b\}$ have the most recent common ancestor, denoted by $T(a, b; c)$, other possibilities are $T(a, c; b)$ and $T(b, c; a)$. In this chapter most given trees are triplets.

Sometimes it is possible that there exists a supertree contains all triplets, but that is not always the case.

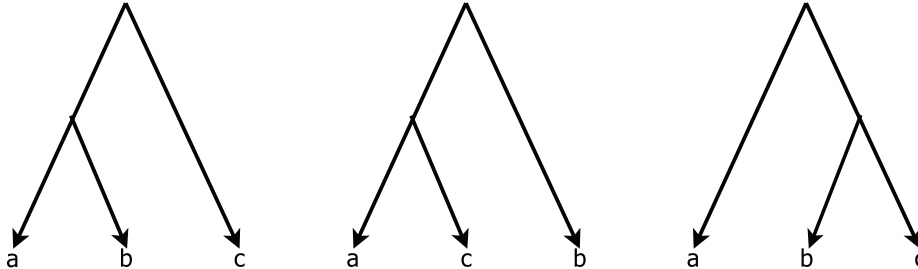


Figure 2.1: All possible triplets with $\{a, b, c\}$. From left to right: $T(a, b; c)$, $T(a, c; b)$, $T(b, c; a)$

Not all of the algorithms that have been created are looking for a supertree containing *most* of the given triplets (or other given small trees), but they do find a tree that still contains information from the triplets.

In this chapter I will explain and discuss a few different algorithms for building a supertree from a set of smaller trees.

2.1. Algorithm: BUILD

The algorithm *BUILD* constructs, given a set of conditions, a supertree satisfying all conditions if it exists. If the conditions are not *compatible*, there does not exist a supertree that can satisfy all of the given conditions. In this case the algorithm will return "the null tree". This algorithm was first described in [6].

The set of conditions can be given in the form of small trees (for example triplets), but they can also

look like this:

$$\begin{aligned} (a, b) &< (c, d) \\ (a, c) &< (b, d) \\ (e, d) &< (a, d) \end{aligned} \tag{2.1}$$

The notation “ $(a, b) < (c, d)$ ” means that the lowest common ancestor of $\{a, b\}$ is a proper descendant of the lowest common ancestor of $\{c, d\}$.

I will explain how the algorithm BUILD works with two examples. In Example 2.1.1 BUILD will be applied to the conditions in (2.1). In Example 2.1.2 BUILD will be applied to triplets.

Example 2.1.1. BUILD makes groups of the taxa which have a recent common ancestor. Since $\{a, b\}$ have a more recent common ancestor than $\{c, d\}$ in the conditions (2.1), $\{a, b\}$ will be together in a group. For the same reason $\{a, c\}$ and $\{e, d\}$ should be grouped together also.

This gives us the groups $S_1 = \{a, b, c\}$ and $S_2 = \{e, d\}$. For each group we build an edge from the root. If there is only one group, there is no tree that can satisfy all conditions.

The next step is to look for each group S_i at the conditions restricted to that group. In this example there are no conditions restricted only to the set S_1 or only to the set S_2 .

The resulting supertree is constructed in Figure 2.2.

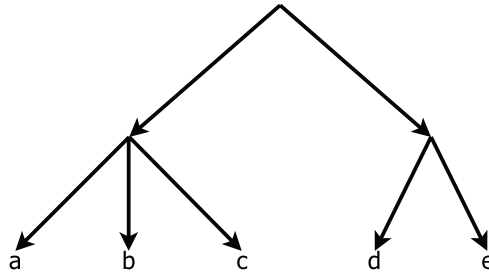


Figure 2.2: Supertree result of the conditions (2.1).

One of the conditions we have seen in the example above is that if $(a, b) < (c, d)$, then a and b should be together in a group. Next to this condition there is also the following condition: If $(a, b) < (c, d)$ and if $\{c, d\}$ are in one group, then the group that contains $\{a, b\}$ should be merged with the group that contains $\{c, d\}$. [6] However, this step doesn't apply in the example above.

Example 2.1.2. Suppose we have been given a set of triplets:

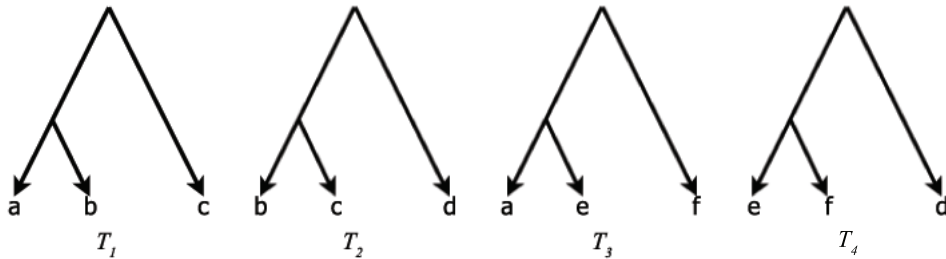


Figure 2.3: Set of compatible triplets $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$.

In this example we make groups of taxa that have a recent common ancestor, just like we did in Example 2.1.1. To make this more visual we make a graph $S_{\mathcal{T}|\mathcal{X}}$. To construct this graph knowledge of the following definitions is needed:

Definition 2.1.1. Suppose T is a rooted phylogenetic tree on $\mathcal{X}$.

1. A *cluster* is a subset $X \subseteq \mathcal{X}$, where X consists of all descendants of a certain vertex v of T . A cluster is *proper* if this vertex is not the root r .

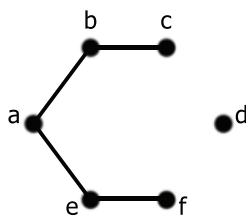
2. $T|\mathcal{X}$ is a subtree of T induced by the taxa in $\mathcal{X}$. Which means that only the leaves of the taxa in $\mathcal{X}$ are left in the subtree $T|\mathcal{X}$
3. $\mathcal{T}|\mathcal{X} = \{T_j|\mathcal{X} : T_j \in \mathcal{T}\}$

In Figure 2.4a the graph $S_{\mathcal{T}|\mathcal{X}}$ is constructed. $S_{\mathcal{T}|\mathcal{X}}$ has a vertex for each taxon in $\mathcal{X}$ and an edge between taxa that are together in a *proper cluster* in one of the trees in $\mathcal{T}|\mathcal{X}$.

For each connected component $S_1, \dots, S_n$ in graph $S_{\mathcal{T}|\mathcal{X}}$ we build an edge from the root, see Figure 2.4b. If there is only one connected component, then there is no tree that can satisfy all the triplets.

The next step is to repeat this process for each connected component. Thus we will build a graph $S_{\mathcal{T}|\mathcal{X}_i}$, where $\mathcal{X}_i = \{x : x \in S_i\}$ for $i = 1, \dots, n$, see Figure 2.4c. Make a new node for a root r_i for the taxa $\mathcal{X}_i$ and attach this root to the edge build for these taxa. Again build an edge from the root for each connected component, see Figure 2.4d.

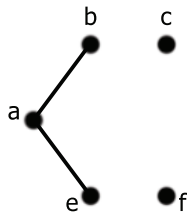
We repeat these steps until there are no connected components left, see Figure 2.4e. This will eventually result to the supertree in Figure 2.4f.



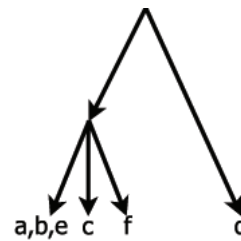
(a) $S_{\mathcal{T}|\mathcal{X}}$ in the first iteration.



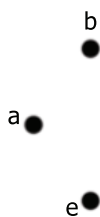
(b) The supertree in the first iteration.



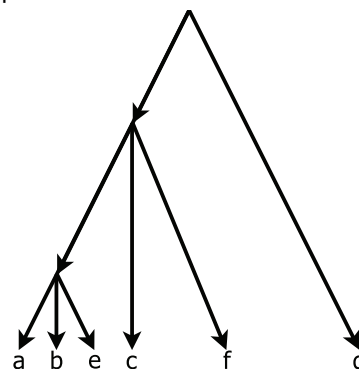
(c) $S_{\mathcal{T}|\mathcal{X}_1}$ in the second iteration. Here $\mathcal{X}_1 = \{a, b, c, e, f\}$.



(d) The supertree in the second iteration.



(e) $S_{\mathcal{T}|\mathcal{X}_1}$ in the last iteration. Here $\mathcal{X}_1 = \{a, b, e\}$.



(f) The supertree after the last iteration.

Figure 2.4: An example of the use of the algorithm BUILD.

Algorithm 2 BUILD($\mathcal{T}, \mathcal{X}$)**Input:** A set of triplets $\mathcal{T} = \{T_1, \dots, T_n\}$ and a set of taxa $\mathcal{X} = \{a, b, \dots\}$ **Output:** A supertree T containing all triplets in $\mathcal{T}$ if it exists

```

1: if  $\mathcal{X}$  consists of a single node  $x$  then
2:    $T = x$  return  $T$ 
3: else
4:   Construct  $S_{\mathcal{T}|\mathcal{X}}$ 
5:   if  $S_{\mathcal{T}|\mathcal{X}}$  is connected then
6:     return the null tree
7:   else
8:     for each connected component  $S_1, \dots, S_m \subseteq S(\mathcal{T})$  do
9:        $\mathcal{X}_i = \{x : x \in S_i\}$ 
10:       $T_i := \text{BUILD}(\mathcal{T}, \mathcal{X}_i)$ 
11:    end for
12:    Construct  $T$  with a new node for the root and the children of the root are the roots of the
    trees  $T_i, i = 1, \dots, m$ 
13:  end if
14: end if
15: return  $T$ 

```

The algorithms *MinCutSupertree* and *Max Cut triplets* work similar as the algorithm in Example 2.1.2. That is why Algorithm 2 is a (slightly adjusted) version of the BUILD algorithm for triplets. [6][7]

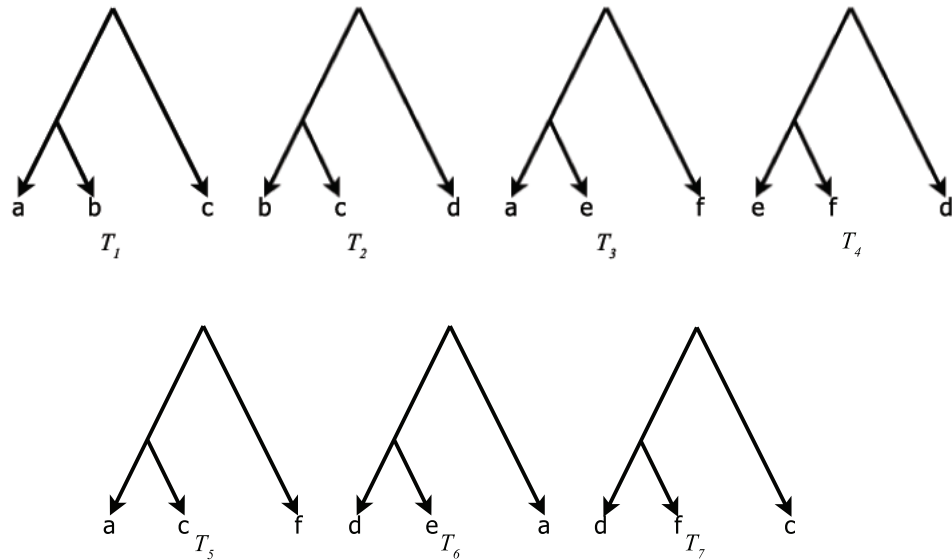
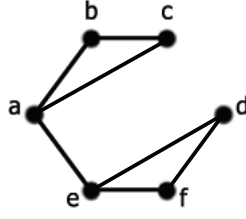
2.2. Algorithm: MinCutSupertree

Figure 2.5: An example of a given set of triplets $\mathcal{T} = \{T_1, \dots, T_7\}$. The set $\mathcal{T}$ is not compatible anymore.

Suppose that the triplets $\{T_5, T_6, T_7\}$ are added to the triplets of Example 2.1.2. All given triplets are represented in Figure 2.5. If we now try to apply the algorithm BUILD on the set of triplets $\mathcal{T} = \{T_1, \dots, T_7\}$, the graph $S_{\mathcal{T}|\mathcal{X}}$ will look like Figure 2.6. This graph is connected, so the response of the algorithm BUILD will be the null tree.

Figure 2.6: Graph $S_{T|X}$ based on $T = \{T_1, \dots, T_7\}$

In this case there is no tree that can satisfy all triplets. If we still want to build a supertree with these triplets as input we will have to use a different supertree method.

One of the methods that does return a supertree even if the input trees are incompatible is `MinCutSupertree` [8].

The algorithm `MinCutSupertree` returns the same tree as the algorithm `OneTree` described in [9] if all the trees in T are compatible. `OneTree` is similar to the algorithm `BUILD`, but for the algorithm `OneTree` the input consists of triplets and *fans*: trees with only one internal vertex, thus only one node next to the leaves. In Algorithm 3 `MinCutSupertree` is displayed.

Algorithm 3 `MinCutSupertree`(T, X, w)

Input: A set of triplets $T = \{T_1, \dots, T_n\}$, a weight function $w : T \rightarrow \mathbb{Q}^+$ and a set of taxa $X = \{a, b, \dots\}$

Output: A supertree T

```

1: if  $X$  consists of a single node  $x$  then
2:    $T = x$  return  $T$ 
3: else
4:   Construct  $S_{T|X}$ 
5:   if  $S_{T|X}$  is disconnected then
6:     Let  $S_i$  be the connected components
7:   else
8:     Construct  $S_{T|X}/E_T^{\max}$ 
9:     Delete all edges of  $S_{T|X}/E_T^{\max}$  that lie in a minimum-weight cut set.
10:    Let  $S_i$  be the resulting connected components
11:   end if
12:   for each component  $S_i$  do
13:      $X_i = \{x : x \in S_i\}$ 
14:      $T_i := \text{MinCutSupertree}(T, X_i, w)$ 
15:   end for
16:   Construct  $T$  with a new node for the root and the children of the root are the roots of the trees
       $T_i$ 
17: end if
18: return  $T$ 

```

The algorithm `MinCutSupertree` requires next to the triplets also a weight function, $w : \{T_1, \dots, T_n\} \rightarrow \mathbb{Q}^+$, as input. If there is no weight function given, we say $w(T_i) = 1 \forall i \in \{1, \dots, n\}$.

To understand the algorithm you will first have to get familiar with the graph $S_{T|X}/E_T^{\max}$. We already know $S_{T|X} = (V_X, E_T)$ is the graph with a vertex for each taxon in X and edge $\{x, y\} \in E_T$ if x, y are together in a proper cluster in one of the trees $T_i \in T$.

The graph $S_{T|X}/E_T^{\max}$ can be constructed as follows:

1. Weight each edge $\{x, y\} \in E_T$, by the sum of the weight of the trees that have x, y in a proper cluster.
2. If the weight of an edge $\{x, y\}$ is equal to the sum of the weight of all trees, thus equal to $w_{\text{sum}} := \sum_{T \in T} w(T)$, contract that edge. $\{x, y\} \in E_T^{\max}$ is the set consisting of such edges. If an

edge $\{x, y\}$ is contracted the graph has a node (x, y) .

3. Delete all *loops* (an edge that connects a node to itself) and replace parallel edges by one edge. This edge gets the weight of the sum of all trees that contain a proper cluster of the endpoints of this edge. Thus if one of the points is (x, y) and the other endpoint is z , the weight of this edge is the sum of the weights of the trees that contain either a proper cluster of $\{x, z\}$ or $\{y, z\}$.

The algorithm `BUILD` returns the null tree, because the graph $S_{\mathcal{T}|X}$ is connected. That is why `MinCutSupertree` will delete some edges in $S_{\mathcal{T}|X}/E_{\mathcal{T}}^{\max}$ such that the graph is not connected anymore. The algorithm `MinCutSupertree` is based on deleting all edges in a minimum-weight cut set.

A *minimum-weight cut set* of $S_{\mathcal{T}|X}/E_{\mathcal{T}}^{\max}$ is a set of edges $\{e_1, \dots, e_k\} \in E_{\mathcal{T}}$ such that $S_{\mathcal{T}|X}/E_{\mathcal{T}}^{\max} \setminus \{e_1, \dots, e_k\}$ is not connected and $\sum_{i=1}^k w(e_i)$ is minimized.

If a cluster is represented in all given trees, then we would like too see these clusters also in the resulting supertree. That is why all the edges $\{x, y\} \in E_{\mathcal{T}}^{\max}$ are contracted.

Example 2.2.1. If we apply the algorithm `MinCutSupertree` to the set of triplets $\mathcal{T} = \{T_1, \dots, T_7\}$, the graph $S_{\mathcal{T}|X}/E_{\mathcal{T}}^{\max}$ will look as in Figure 2.7a.

The next step is to find all minimum-weight cut sets and delete those edges. The edge $\{a, e\}$ lays in a minimum-weight cut set, because the graph $(S_{\mathcal{T}|X}/E_{\mathcal{T}}^{\max}) \setminus \{a, e\}$ is disconnected and $w(\{a, e\}) = 1$ which is in this case a minimum-weight cut. This is the only edge in a minimum-weight cut, thus we will only delete this edge to proceed with the algorithm.

We are left with two connected parts S_1 and S_2 of $S_{\mathcal{T}|X}/E_{\mathcal{T}}^{\max}$. Thus we can start constructing the supertree T with two descendants of the root.

For the next step we apply `MinCutSupertree` on $X_1 = \{x : x \in S_1\}$ and X_2 . $S_{\mathcal{T}|X_1}$ and $S_{\mathcal{T}|X_2}$ are both disconnected graphs, see Figures 2.7c and 2.7d, thus we can easily continue the algorithm. This will eventually give us the supertree displayed in Figure 2.7e.

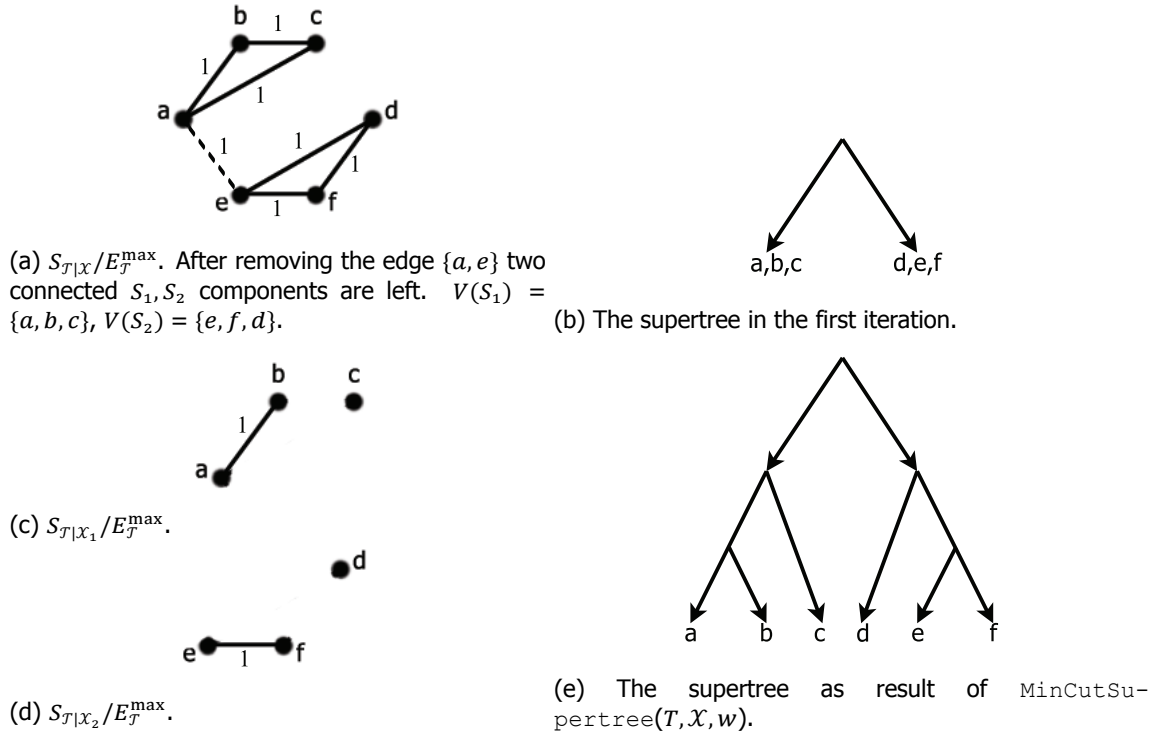


Figure 2.7: An example of `MinCutSupertree` applied to $\mathcal{T} = T_1, \dots, T_7$ in Figures 2.3 and 2.5.

2.2.1. ModifiedMinCut

The algorithm `MinCutSupertree` (MC) can be applied to all kinds of trees, not only to triplets. It can also be that the input trees have different sizes. For instance a given tree could have 5 leaves and another given tree 6.

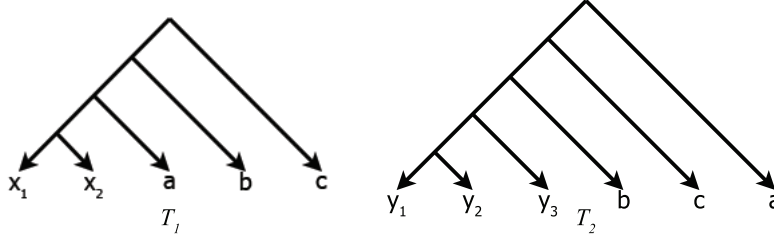
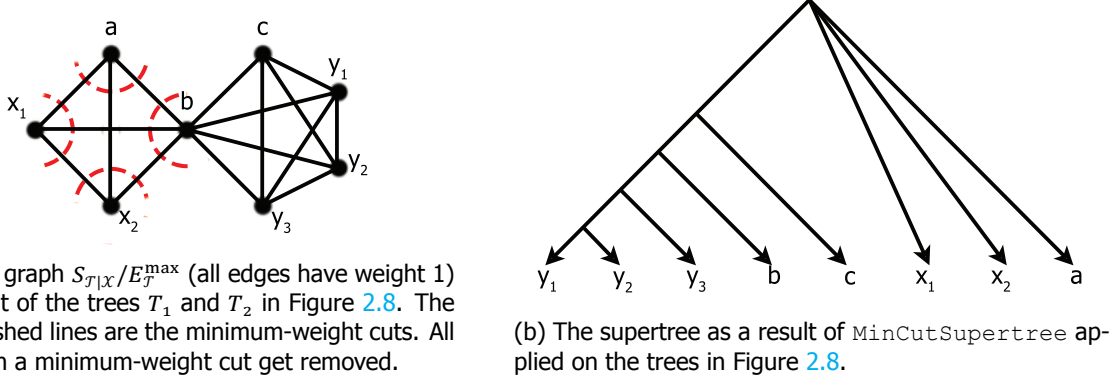


Figure 2.8: Examples of other trees than triplets, that could be given as input data.

Sometimes this difference in number of leaves between trees can lead to an output supertree that is not completely satisfying.

For example look at the trees T_1 and T_2 in Figure 2.8. If the algorithm MC is applied on these two trees, the result will be the supertree in Figure 2.9b. In the given trees $(x_1, x_2) < (b, c)$, this is not represented in the output supertree. Even though there is no tree that suggests something different.

This is because of the way the edges get removed from the graph $S_{\mathcal{T}|\mathcal{X}}/E_{\mathcal{T}}^{\max}$, see Figure 2.9a. Because x_1 and x_2 are in a smaller tree, they are included in fewer proper clusters than y_1, y_2, y_3 . This results in a cut through the edges connected to x_1 and x_2 and therefore the recent common ancestor of $\{x_1, x_2\}$ is not represented in the supertree in Figure 2.9b.



(a) The graph $S_{\mathcal{T}|\mathcal{X}}/E_{\mathcal{T}}^{\max}$ (all edges have weight 1) as result of the trees T_1 and T_2 in Figure 2.8. The red, dashed lines are the minimum-weight cuts. All edges in a minimum-weight cut get removed.

(b) The supertree as a result of MinCutSupertree applied on the trees in Figure 2.8.

Figure 2.9

As said in section 2.2 a good thing about the algorithm MC is that any clusters that are found in all the input trees will get represented in the output supertrees. But if a cluster is not presented in all trees it can get cut even if it is not contradicted by other trees.

Here a concept similar to a cluster will be defined:

Definition 2.2.1. Let T be a rooted phylogenetic tree on $\mathcal{X}$. Suppose $A, B \subseteq \mathcal{X}$. A is a nesting in B if the most recent common ancestor of A is a descendant of the most recent common ancestor of B . Denoted $A <_T B$.

The algorithm MC doesn't pay any attention to the fact that nestings are contradicted or not.

To improve this, the algorithm ModifiedMinCut [10] (MMC) makes distinction between edges in $S_{\mathcal{T}|\mathcal{X}}/E_{\mathcal{T}}^{\max}$ that represent contradicted nestings, uncontradicted nestings, and unanimous nestings.

Definition 2.2.2. 1. A nesting A in B is *contradicted* if in one of the given trees T_i : $A <_{T_i} B$ and in another one T_j : $A <_{T_j} B$.

2. A nesting A in B is *uncontradicted* if $A <_T B$ in all of the given trees that contain A and B .

3. If all given trees contain A and B and the nesting A in B is uncontradicted, then it is an *unanimous nesting*.

The algorithm MC makes sure that you will not cut any edges that are unanimous by contracting the edges in $E_{\mathcal{T}}^{\max}$. The algorithm MMC also tries to minimize the number of uncontradicted edges that get cut in $S_{\mathcal{T}|\mathcal{X}}/E_{\mathcal{T}}^{\max}$. You can see how this works in lines 8-18 in Algorithm 4. To get a clearer idea of the working principle of MMC look at Example 2.2.2.

Algorithm 4 $\text{ModifiedMinCut}(\mathcal{T}, \mathcal{X}, w)$

Input: A set of triplets $\mathcal{T} = \{T_1, \dots, T_n\}$, a weight function $w : \mathcal{T} \rightarrow \mathbb{Q}^+$ and a set of taxa $\mathcal{X} = \{a, b, \dots\}$

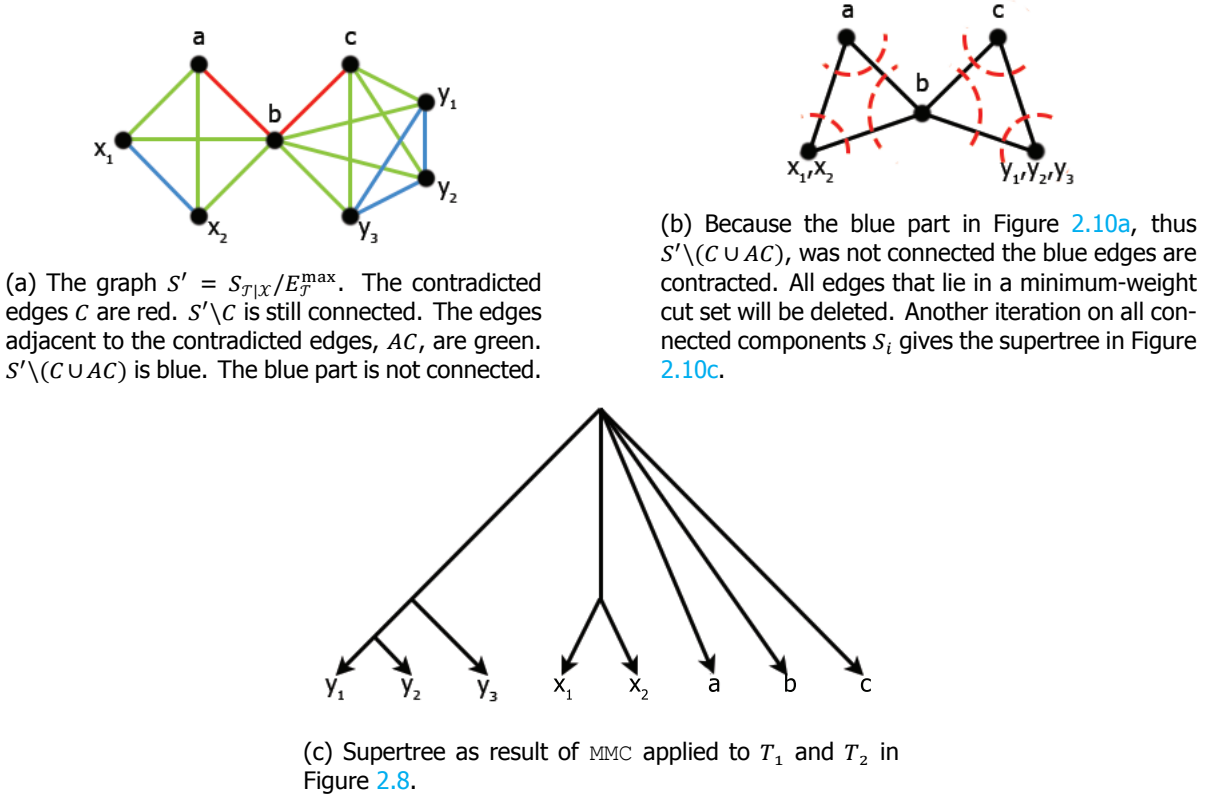
Output: A supertree T

```

1: if  $\mathcal{X}$  consists of a single node  $x$  then
2:    $T = x$  return  $T$ 
3: else
4:   Construct  $S_{\mathcal{T}|\mathcal{X}}$ 
5:   if  $S_{\mathcal{T}|\mathcal{X}}$  is disconnected then
6:     Let  $S_i$  be the connected components
7:   else
8:     Construct  $S' = S_{\mathcal{T}|\mathcal{X}}/E_{\mathcal{T}}^{\max}$ 
9:     Let  $C$  be the set of contradicted edges in  $S'$ 
10:    if  $S' \setminus C$  is disconnected then
11:      contract the connected components of  $S' \setminus C$ 
12:    else
13:      Let  $AC \in S'$  be the set of edges adjacent to the edges in  $C$ 
14:      if  $S' \setminus (C \cup AC)$  is disconnected then
15:        contract the connected components of  $S' \setminus (C \cup AC)$ 
16:      end if
17:    end if
18:    Delete all edges of  $S'$  that lie in a minimum-weight cut set.
19:    Let  $S_i$  be the resulting connected components
20:  end if
21:  for each component  $S_i$  do
22:     $\mathcal{X}_i = \{x : x \in S_i\}$ 
23:     $T_i := \text{ModifiedMinCut}(\mathcal{T}, \mathcal{X}_i, w)$ 
24:  end for
25:  Construct  $T$  with a new node for the root and the children of the root are the roots of the trees
     $T_i$ 
26: end if
27: return  $T$ 

```

Example 2.2.2. In Figure 2.10 the algorithm MMC is applied to the trees T_1 and T_2 in Figure 2.8. As you can see the resulting supertree of MMC represents both nestings $\{x_1, x_2\}$ and $\{y_1, y_2, y_3\}$. This result looks a lot better than the supertree as result of MC .

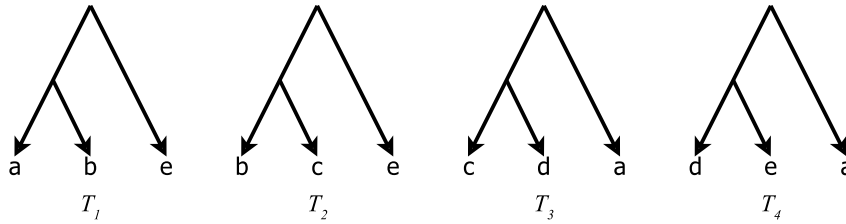
Figure 2.10: Steps of ModifiedMinCut applied to the trees T_1 and T_2 in Figure 2.8.

2.3. Algorithm: Max Cut triplets

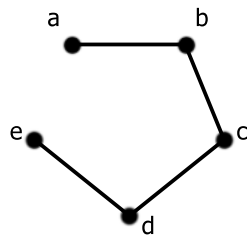
MAX CUT triplets[11] is another algorithm based on the algorithms BUILD and MC. Just like MMC, this algorithm also tries to improve the output supertree in MC.

First an example of triplets where MC does not return a satisfying supertree is given.

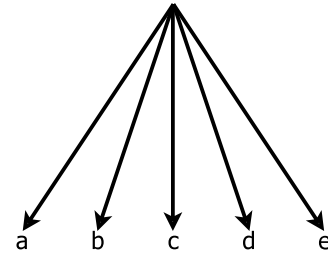
Example 2.3.1. Suppose the following triplets $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$ in Figure 2.11 are given.

Figure 2.11: Example of a set of triplets $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$.

Constructing the graph $S_{T|X}$ gives Figure 2.12a. This graph exists of one connected component thus there is no supertree that can satisfy all triplets. Constructing $S_{T|X}/E_T^{\max}$ does not give any new information, because all clusters are only represented in one triplet, thus all edges get weight 1. This means that all edges are in a minimum-weight cut and all edges will get removed. This results in the supertree represented in Figure 2.12b.



(a) The graph $S_{T|X}$ based on the triplets $\mathcal{T}$ in Figure 2.11. All edges in this graph are in a minimum-weight cut set.



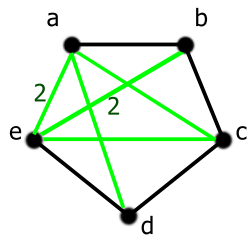
(b) The supertree result of `MinCutSupertree` applied on the triplets $\mathcal{T}$ in Figure 2.11.

Figure 2.12

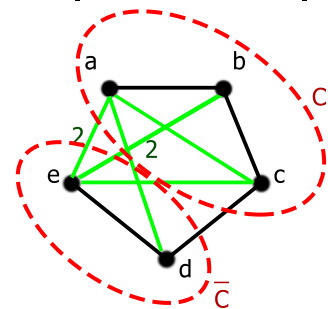
The supertree represented in Figure 2.12b is not a very satisfying result. None of the information of the given triplets is visible in the output supertree. There are several triplets that confirm that taxa a and b are not closely related to taxa e , but in the resulting supertree each taxon seems to be just as closely related.

The algorithm `MAX CUT triplets` tries to maximize the number of satisfied triplets. For a triplet $T(i, j; k)$, the algorithm `MinCutSupertree` only makes an edge between $\{i, j\}$ when constructing the graph $S_{T|X}$, these are called *bad edges* in [11]. `MAX CUT triplets` also makes edges between $\{i, k\}$ and $\{j, k\}$, called *good edges* in [11].

In Figure 2.13a these good edges are added to the graph $S_{T|X}$ in Figure 2.12a.



(a) Bad edges in black and good edges in green. Two edges have weight 2 because this relation is confirmed in 2 trees.



(b) A cut $(C, \bar{C})$ that maximizes the ratio good versus bad edges.

Figure 2.13

Instead of a minimum-weight cut, `MAX CUT triplets` searches for a cut $(C, \bar{C})$ that maximizes the ratio good versus bad edges. See Figure 2.13b for an illustration. The cut results in the graph in Figure 2.14a. Now it is possible to construct a supertree. See Figure 2.14b for the resulting supertree of the `MAX CUT triplets` algorithm.

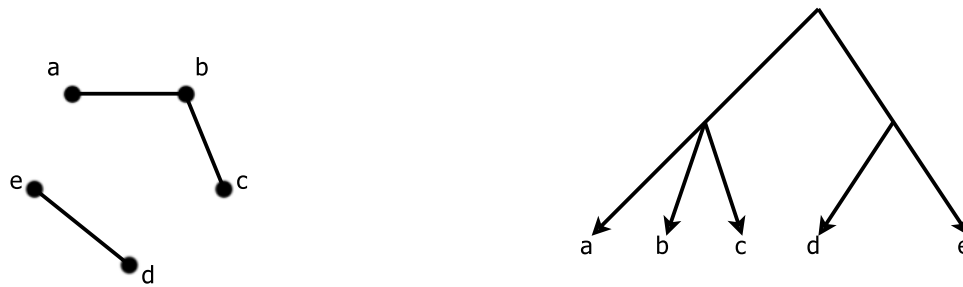
(a) The graph $S_{T|X}$ after the cut.(b) The supertree result of MAX CUT triplets applied on the triplets $\mathcal{T}$ in Figure 2.11.

Figure 2.14

It is now clear how the algorithm MAX CUT triplets works and how this can result in a better supertree than the algorithm MinCutSupertree. The supertree in Figure 2.12b constructed with MinCutSupertree displayed none of the 4 given triplets, while the supertree in Figure 2.14b constructed with MAX CUT triplets displays 3 out of 4 given triplets (T_1, T_2, T_4).

In Algorithm 5, MAX CUT triplets is represented.

Algorithm 5 MAX CUT triplets($\mathcal{T}, \mathcal{X}$)

Input: A set of triplets $\mathcal{T} = \{T_1, \dots, T_n\}$, a weight function $w : \mathcal{T} \rightarrow \mathbb{Q}^+$ and a set of taxa $\mathcal{X} = \{a, b, \dots\}$

Output: A supertree T

```

1: if  $\mathcal{X}$  consists of a single node  $x$  then
2:    $T = x$  return  $T$ 
3: else
4:   Construct  $S' = S_{\mathcal{T}|\mathcal{X}}$ 
5:   if  $S'$  is disconnected then
6:     Let  $S_i$  be the connected components
7:   else
8:     Denote the edges created in step 4 as bad edges
9:     For every triplet  $T(i, j; k)$  add good edges  $(i, k)$  and  $(j, k)$  to the graph  $S'$ 
10:    Weight the good edges  $\{x, y\}$  as follows:  $w(\{x, y\}) = \{\sum w(T) : T(z, x; y), \text{ where } z \in \mathcal{X}\}$ 
11:    Weight the bad edges  $\{x, y\}$  the same:  $w(\{x, y\}) = \{\sum w(T) : T(x, y; z), \text{ where } z \in \mathcal{X}\}$ .
12:    Find a cut  $(C, \bar{C})$  in  $S'$  such that the ratio of the weight of the good edges versus the weight
of the bad edges in the cut is maximized.
13:    Let  $S_i$  be the remaining connected components
14:   end if
15:   for each component  $S_i$  do
16:      $\mathcal{X}_i = \{x : x \in S_i\}$ 
17:      $T_i := \text{MAX CUT triplets}(\mathcal{T}, \mathcal{X}_i)$ 
18:   end for
19:   Construct  $T$  with a new node for the root and the children of the root are the roots of the trees
 $T_i$ 
20: end if
21: return  $T$ 

```

3

Constructing Supernetworks from Binets

We have seen in Chapter 2 that there are several algorithms to build a supertree given a set of triplets. In this chapter an algorithm to construct a supernetwork from a given set of binets will be explained and analyzed. This algorithm returns the null network if the binets contradict with each other, thus if there does not exist a supernetwork that contains all of the given binets.

Furthermore I will search for an adjustment to the original algorithm to construct a supernetwork of the given binets if the original algorithm returns the null network. Also I will present a lower bound to the fraction of binets that should at least be able to fit in a supernetwork.

3.1. Algorithm: BuildNW

In this section an existing algorithm will be explained which constructs a binary supernetwork containing all given binets, if it exists. The algorithm is represented in Algorithm 6. I called the algorithm *BuildNW*. This is not how it is called in [5] but it is easier to reference to, if the algorithm has a name. The reason to name it *BuildNW* is because it shows similarities with the algorithm BUILD in Section 2.1 and NW stands for Network.

To get a good idea how the algorithm works, take a look at Example 3.1.1.

Example 3.1.1. Suppose that the set binets $\mathbb{B} = \{B_1, \dots, B_5\}$ in Figure 3.1 are given.

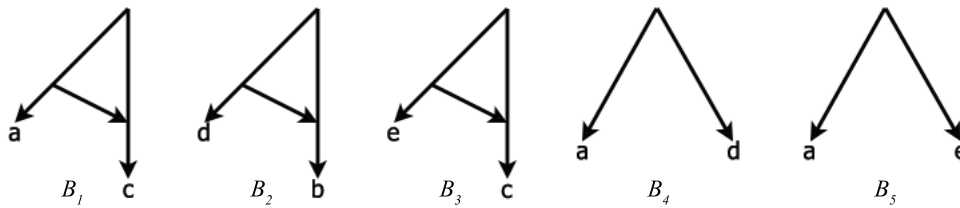


Figure 3.1: $\mathbb{B} = \{B_1, \dots, B_5\}$.

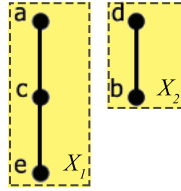
One of the first steps of the algorithm *BuildNW* is to construct $\mathcal{R}(\mathbb{B})$, see line 4 of Algorithm Alg:BuildNW. For the given binets graph $\mathcal{R}(\mathbb{B})$ will look like Figure 3.2.

Algorithm 6 BuildNW($\mathbb{B}$)**Input:** A set of binets $\mathbb{B} = \{B_1, \dots, B_n\}$ and a set of taxa $\mathcal{X} = \{a, b, \dots\}$ **Output:** A binary supernetwork N containing all binets in $\mathbb{B}$ if it exists

```

1: if  $\mathcal{X}$  consists of a single node  $x$  then
2:    $N = x$ 
3: else
4:   Construct  $\mathcal{R}(\mathbb{B})$ : A vertex for each taxon in  $\mathcal{X}$  and an edge between  $\{x, y\}$  if  $N(x; y) \vee N(y; x) \in \mathbb{B}$ 
5:   if  $\mathcal{R}(\mathbb{B})$  is disconnected and has connected components  $X_1, \dots, X_p$  then
6:      $N_i = \text{BuildNW}(\mathbb{B}|X_i)$ ,  $i = 1, \dots, p$ 
7:     Construct  $N$  with a node  $r$  for the root and with edges from  $r$  to the roots of the networks  $N_i$ , such that  $N$  is binary
8:   else
9:     Construct  $\mathcal{K}(\mathbb{B})$ : A vertex for each taxon in  $X_1$  and an edge between  $\{x, y\}$  if  $T(x, y) \in \mathbb{B}$ 
10:    Construct  $\Omega(\mathbb{B})$ : A vertex,  $\pi_1, \dots, \pi_n$ , for each connected component of  $\mathcal{K}(\mathbb{B})$  and an arc  $(\pi_1, \pi_2)$  if for  $x \in V(\pi_1)$  and  $y \in V(\pi_2)$ :  $N(y; x) \in \mathbb{B}$ 
11:    Search for a strict subset  $U \subset V(\Omega(\mathbb{B}))$  such that no arcs are going to  $U$ 
12:    if there is no subset  $U$  that satisfies then
13:      return The null network
14:    else
15:      Construct  $N_{up} = \text{BuildNW}(\mathbb{B}|U)$ 
16:      Call  $L = \mathcal{X} \setminus U$  and construct  $N_{low} = \text{BuildNW}(\mathbb{B}|L)$ 
17:      Build  $N = N(N_{low}; N_{up})$ 
18:    end if
19:  end if
20: end if
21: return  $N$ 

```

Figure 3.2: $\mathcal{R}(\mathbb{B})$ constructed for $\mathbb{B} = \{B_1, \dots, B_5\}$ in Figure 3.1 consists of two connected components X_1 and X_2 .

The graph $\mathcal{R}(\mathbb{B})$ is disconnected, there are two connected components X_1 and X_2 in $\mathcal{R}(\mathbb{B})$.

To construct N_1 we repeat the algorithm for $\mathbb{B}|X_1$, which are the binets $\mathbb{B}$ induced by the taxa in X_1 . If we construct $\mathcal{R}(\mathbb{B}|X_1)$, the result is exactly the connected component X_1 in $\mathcal{R}(\mathbb{B})$, thus $\mathcal{R}(\mathbb{B}|X_1)$ is connected.

The next step is to construct $\mathcal{K}(\mathbb{B}|X_1)$. After $\mathcal{K}$ construct $\Omega(\mathbb{B}|X_1)$. See Figure 3.3a.



(a) The solid lines are $\mathcal{K}(\mathbb{B}|X_1)$ and the dashed arcs with the red vertices are $\Omega(\mathbb{B}|X_1)$.

(b) The supernet N_1 , part of the supernet N . Constructed with the information of the graphs $\mathcal{K}(\mathbb{B}|X_1)$ and $\Omega(\mathbb{B}|X_1)$ in Figure 3.3a.

Figure 3.3: Constructing N_1 by applying the algorithm `BuildNW` to $\mathbb{B}|X_1$.

The next step is to search for a subset $U \subset V(\Omega(\mathbb{B})) = \{\pi_1, \dots, \pi_n\}$ where no arcs are going to. In other words such that there are no arcs (π_i, π_j) with $\pi_i \notin U$ and $\pi_j \in U$. In the case of Figure 3.3a there exists such a subset U , namely $U = \{\pi_1\} (= \{a, b\})$.

In this case $L = X_1 - U = \{c\}$. Now we can build N_1 with a network with `BuildNW( $\mathbb{B}|L$ )` below `BuildNW( $\mathbb{B}|U$ )`, thus $N_1 = N(\text{BuildNW}(\mathbb{B}|L) ; \text{BuildNW}(\mathbb{B}|U))$. N_1 will look like Figure 3.3b if we have recursively computed networks `BuildNW( $\mathbb{B}|L$ )` and `BuildNW( $\mathbb{B}|U$ )`. If we have also constructed N_2 we can attach the roots of N_1 and N_2 to the root of the supernet. See Figure 3.4 for the final supernet.

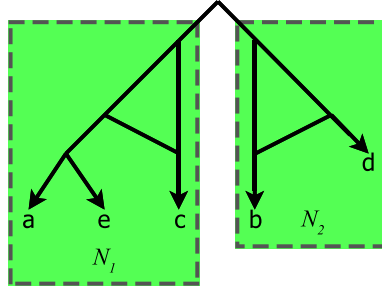


Figure 3.4: Supernet as result of the set $\mathbb{B} = \{B_1, \dots, B_5\}$.

3.2. When does the algorithm fail?

Suppose $\Omega(\mathbb{B})$ (in line 10) of the algorithm `BuildNW` is constructed. If there is no strict subset U such that no arcs are going to U , the algorithm stops and returns "the null network". This means that there is no supernet that can contain all of the given binets.

What I want is a supernet that contains most of the given binets. To search for a good solution to find this supernet I will first examine in what cases the algorithm `BuildNW` returns the null network.

Figures 3.5 and 3.6 are simple examples for which the algorithm `BuildNW` returns the null network.

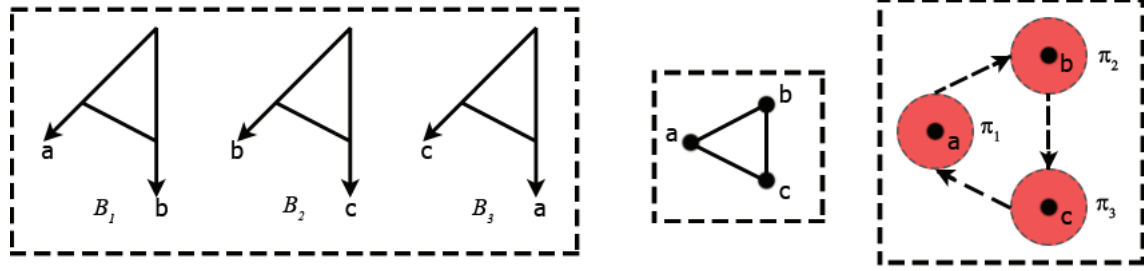


Figure 3.5: $\mathbb{B} = \{B_1, B_2, B_3\}$ is an example of a set of binets that have the null network as result. See the second box for the graph $\mathcal{R}(\mathbb{B})$ this graph is connected, thus we make $\mathcal{K}(\mathbb{B})$ of all binets, see the last box. There are no trees in the set of binets thus a, b, c are all in another “connected” part of $\mathcal{K}(\mathbb{B})$. Finally if we make $\Omega(\mathbb{B})$ we can see that the algorithm will return the null network, because there is no strict subset $U \subset V(\Omega)$ such that has no arcs going into U .

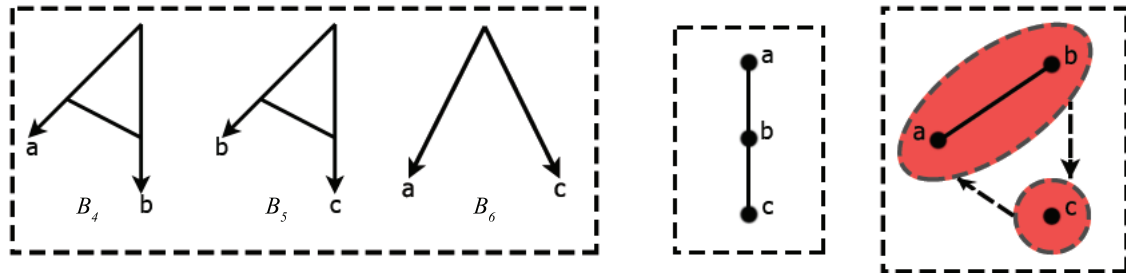


Figure 3.6: $\mathbb{B} = \{B_4, B_5, B_6\}$. Again in the second box graph $\mathcal{R}(\mathbb{B})$ is constructed and in the third box $\mathcal{K}(\mathbb{B})$ and $\Omega(\mathbb{B})$. Here the return will also be a null network, because as you can see there is no strict subset $U \subset V(\Omega)$ such that no arcs are going into U .

Notice that in case Ω is a directed cycle you will not be able to find a strict subset U that satisfies. In fact there is an even stronger statement:

Lemma 3.2.1. If Ω contains a directed cycle the algorithm will eventually return the null network.

Proof: Suppose Ω is a larger graph that contains a directed cycle. It can be that in the first iterations there exists a strict subset U such that no arcs are going into U . The next step is to keep repeating the algorithm for $\mathbb{B}|U$ and $\mathbb{B}|L$ ($L = \mathcal{X} - U$). L or U contains the set of taxa that form the directed cycle. It is never possible to split the taxa contained in this cycle in U and L , because then there would be arcs going to U from other taxa not in U . Thus the directed cycle will always stay in one group. Eventually the algorithm will be repeated for only the group of taxa in the cycle. This means that in this iteration there is no strict subset U that satisfies. This means we will eventually get the null network as a return of the algorithm. $\square$

Remark: For a subset $X' \in \mathcal{X}$ the graphs $\mathcal{R}(\mathbb{B}|X')$, $\mathcal{K}(\mathbb{B}|X')$ and $\Omega(\mathbb{B}|X')$ are the same as the subgraphs of $\mathcal{R}(\mathbb{B})$, $\mathcal{K}(\mathbb{B})$ and $\Omega(\mathbb{B})$ induced by the taxa in X' . This is remarkable because this is not true for the graphs used for the construction of supertrees: For a subset $X' \in \mathcal{X}$: in general $S_{T|X'}$ is not equal to $S_{T|X}$ restricted to the taxa in X' . This is because for the graph $S_{T|X}$ constructing an edge between taxa $\{x, y\}$ depends on the other nodes in the trees that contain $\{x, y\}$. Constructing an edge $\{x, y\}$ in the graphs $\mathcal{R}, \mathcal{K}, \Omega$ only depends on the taxa $\{x, y\}$ and if they are together in a binet or not.

It is not very convenient that every time we want to examine whether the given set of binets has a solution, we need to draw three separate graphs $\mathcal{R}, \mathcal{K}$ and at last Ω . That is a reason to introduce $\mathcal{R}^*(\mathbb{B})$, which is a combination of the graphs $\mathcal{R}(\mathbb{B})$ and $\Omega(\mathbb{B})$.

Definition 3.2.1. $\mathcal{R}^*(\mathbb{B})$ is a directed graph with a vertex for every taxon in the set of binets $\mathbb{B} = \{B_1, \dots, B_n\}$ and with an arc (x, y) if $N(y; x) \in \mathbb{B}$

In Figure 3.7b this newly defined graph $\mathcal{R}^*(\mathbb{B})$ is constructed for the binets $\mathbb{B} = \{B_1, B_2, B_3\}$ in Figure 3.5. Because the graph $\mathcal{R}^*(\mathbb{B})$ contains a directed cycle, you can immediately see that this set of binets

has no supernetwork solution, even without looking at $\mathcal{K}(\mathbb{B})$.



(a) $\mathcal{R}^* + \mathcal{K}$ for the binets $\mathbb{B} = \{B_1, B_2, B_2\}$.

(b) $\mathcal{R}^* + \mathcal{K}$ for the binets $\mathbb{B} = \{B_4, B_5, B_6\}$.

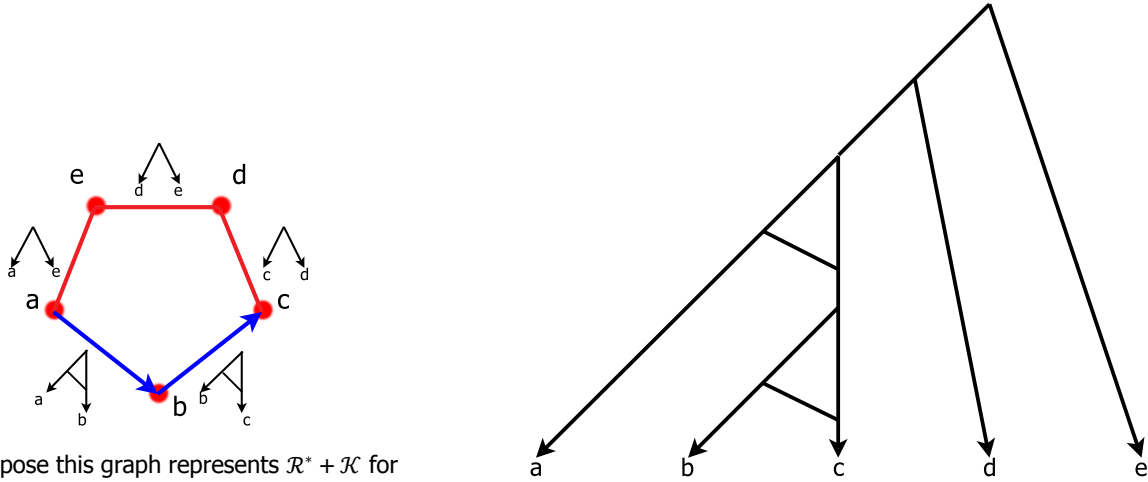
Figure 3.7: In blue the graph $\mathcal{R}^*(\mathbb{B})$ and in red the graph $\mathcal{K}(\mathbb{B})$.

With the binets $\mathbb{B} = \{B_4, B_5, B_6\}$ in example in Figure 3.6 this is different. If only $\mathcal{R}^*$ is applied to this set of binets we will not immediately be able to conclude if there exists a supernetwork that contains all binets or not. In this case it is necessary to construct $\mathcal{K}(\mathbb{B})$ to see that the algorithm will return the null network. Because this new version of $\mathcal{R}^*(\mathbb{B})$ is a combination of $\mathcal{R}(\mathbb{B})$ and $\Omega(\mathbb{B})$, the graph $\mathcal{K}(\mathbb{B})$ is constructed in the same Figure. This way we are able to look for directed cycles between connected components of $\mathcal{K}$ just like in the original algorithm. In Figure 3.7b the graph $\mathcal{R}^* + \mathcal{K}$ is constructed for the binets $\mathbb{B} = \{B_4, B_5, B_6\}$.

Definition 3.2.2. $\mathcal{R}^* + \mathcal{K}$ is a *mixed graph* (with arcs and edges), with the arcs of $\mathcal{R}^*$ and edges of $\mathcal{K}$.

You can see that there is a directed cycle between the connected components of $\mathcal{K}(\mathbb{B})$. In the original algorithm the connected components of $\mathcal{K}(\mathbb{B})$ would be a vertex in $\Omega(\mathbb{B})$. This means that there is a directed cycle between the vertices of $\Omega(\mathbb{B})$.

It seems natural to think that if there is a directed cycle in the graph $\mathcal{R}^* + \mathcal{K}$ between connected components in $\mathcal{K}$, the algorithm returns the null network. To show this is not necessarily true I will give an example in Figure 3.8. In this figure there is a directed cycle between the connected components in $\mathcal{K}$: $C = (\{b\} \rightarrow \{a, e, d, c\} \rightarrow \{b\})$, but there is still a supernetwork that can contain all binets.



(a) Suppose this graph represents $\mathcal{R}^* + \mathcal{K}$ for a given set of binets (see drawn next to the matching edges/arcs). Again blue is $\mathcal{R}^*$ and $\mathcal{K}$ is red.

(b) The supernetwork constructed based on the graph in Figure 3.8a.

Figure 3.8: In this example there is a directed cycle in the graph $\mathcal{R}^* + \mathcal{K}$ between the connected components in $\mathcal{K}$, but we are still able to find a supernetwork that contains all binets.

The reason that we are able to find a supernetwork in Figure 3.8 is that the directed cycle is not in one connected component of $\mathcal{R}^*$. $X_1 = \{a, b, c\}$, $X_2 = \{d\}$ and $X_3 = \{e\}$ are all different connected

components of $\mathcal{R}^*$. This means that in the original graph $\mathcal{R}(B)$ there are also three separate connected components and for the next step $\mathcal{K}(\mathbb{B}|X_1), \mathcal{K}(\mathbb{B}|X_2), \mathcal{K}(\mathbb{B}|X_3)$ would be constructed separately. This results in three graphs $\Omega(\mathbb{B}|X_i)$ without any directed cycles and without problems in finding a subset $U \subset V(\Omega)$ to construct a supernetwork.

Conclusion: constructing $\mathcal{K}(\mathbb{B})$ for all binets when $\mathcal{R}^*(\mathbb{B})$ is disconnected is unnecessary and confusing.

In Figure 3.9 I made two extra binets such that $\mathcal{R}^*(\mathbb{B})$ is in one connected component. $\mathcal{K}(\mathbb{B})$ is constructed in the same figure. We see that there is a directed cycle between the connected components of $\mathcal{K}(\mathbb{B})$. In this case the algorithm will return a null network, because the directed cycle $C = (\{b\} \rightarrow \{a, e, d, c\} \rightarrow \{b\})$ is now in one connected component of $\mathcal{R}^*(\mathbb{B})$.

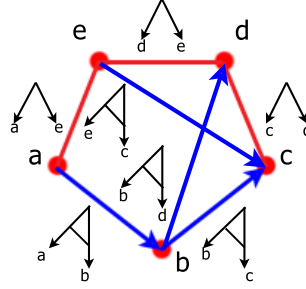


Figure 3.9: $\mathcal{X} = \{a, b, c, d, e\}$ are all in one connected component of $\mathcal{R}^*$ and there is a directed cycle between connected components $\pi_1 = \{a, e, d, c\}$ and $\pi_2 = \{b\}$ of $\mathcal{K}$.

For this reason the following lemma is introduced.

Lemma 3.2.2. Given a set of binets $\mathbb{B}$, with taxa $\mathcal{X} = \{a, b, \dots\}$. Suppose the graph $\mathcal{R}^*(\mathbb{B}) + \mathcal{K}(\mathbb{B})$ is constructed.

If $\exists$ a subset $X' \subseteq \mathcal{X}$ connected in $\mathcal{R}^*(\mathbb{B}|X')$ and with a directed cycle between connected components of $\mathcal{K}(\mathbb{B}|X')$ then the algorithm `BuildNW` would return the null network.

Proof: Suppose there is a given a set of binets $\mathbb{B}$, with taxa $\mathcal{X} = \{a, b, \dots\}$ and for a subset $X' \subset \mathcal{X}$, $\mathcal{R}^*(\mathbb{B}|X')$ is connected and there is a directed cycle between connected components of $\mathcal{K}(\mathbb{B}|X')$. To proof that `BuildNW` will return the null network we look at the original graphs $\mathcal{R}$ and Ω :

The graph $\mathcal{R}(\mathbb{B})$ may not be connected, but if $\mathcal{R}^*(\mathbb{B}|X')$ is connected $\mathcal{R}(\mathbb{B}|X')$ would also be connected. In one of the iterations the algorithm will be done for a subset $(X' \subseteq) X'' \subseteq \mathcal{X}$ for which $\mathcal{R}(\mathbb{B}|X'')$ is connected.

Because $\mathcal{R}(\mathbb{B}|X'')$ is connected, $\mathcal{K}(\mathbb{B}|X'')$ will get constructed. Suppose $\pi_1, \dots, \pi_i$ are the connected components of $\mathcal{K}(\mathbb{B}|X'')$. These connected components would be vertices in $\Omega(\mathbb{B}|X'')$. Thus if there is a directed cycle between connected components $\pi_{c_1}, \dots, \pi_{c_k}$ of $\mathcal{K}(\mathbb{B}|X')$, the graph $\Omega(\mathbb{B}|X'')$ contains a directed cycle. According to lemma 3.2.1 the algorithm will return the null network. $\square$

Does this mean that if there are no directed cycles between connected components of $\mathcal{K}$, `BuildNW` will always return a supernetwork?

If this appears to be true this would indicate that in the case there exists no subset $X' \in \mathcal{X}$ connected in $\mathcal{R}^*$ and with a directed cycle between connected components of $\mathcal{K}$ there, is always a strict subset U in the connected components of $\mathcal{K}$ such that no arcs are going into U .

After a little research I saw that this is not always true. In Figure 3.10 there is an example of binets that gave no directed cycle between components of $\mathcal{K}$ after constructing $\mathcal{R}^* + \mathcal{K}$. Yet there is no strict subset U in the connected components of $\mathcal{K}$ such that no arcs are going into U , because there is only one connected component in $\mathcal{K}$, namely $\pi_1 = \{a, b, c, d\}$.

It seems that if the given binets result to a graph $\mathcal{R}^* + \mathcal{K}$ that is connected in $\mathcal{R}^*$ AND connected in $\mathcal{K}$, the result of `BuildNW` will also be the null network.

Definition 3.2.3. The graph $\mathcal{R}^* + \mathcal{K}$ is called $\mathcal{R}^* + \mathcal{K}$ -connected if the graph is connected in $\mathcal{R}^*$ AND connected in $\mathcal{K}$.

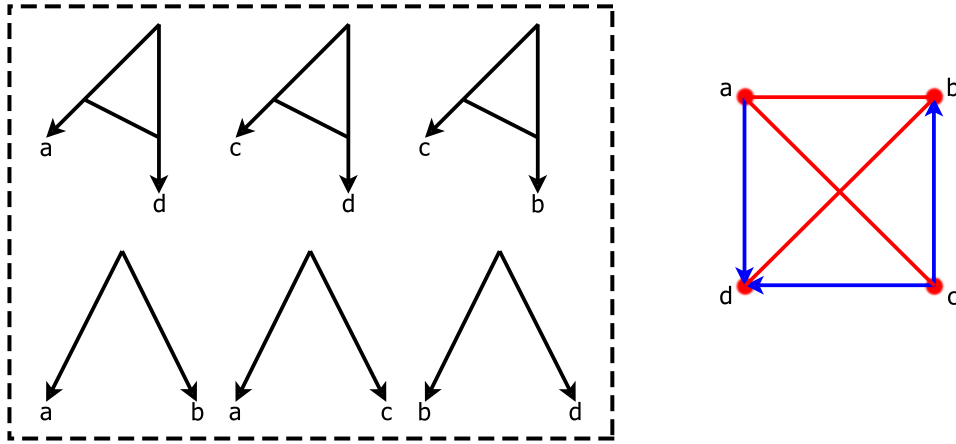


Figure 3.10: An example of a set of binets that result to a graph that is $\mathcal{R}^* + \mathcal{K}$ -connected (but has no directed cycles).

Lemma 3.2.3. Given a set of binets $\mathbb{B}$, with taxa $\mathcal{X} = \{a, b, \dots\}$. Suppose the graph $\mathcal{R}^*(\mathbb{B}) + \mathcal{K}(\mathbb{B})$ is constructed.

If $\exists X' \subseteq \mathcal{X}$ such that the graph $\mathcal{R}^*(\mathbb{B}|X') + \mathcal{K}(\mathbb{B}|X')$ is $\mathcal{R}^* + \mathcal{K}$ -connected, the result of `BuildNW` will be the null network.

Proof: Suppose there is a given set of binets $\mathbb{B}$, with taxa $\mathcal{X} = \{a, b, \dots\}$ and for a subset $X' \subset \mathcal{X}$, the graph $\mathcal{R}^*(\mathbb{B}|X') + \mathcal{K}(\mathbb{B}|X')$ is $\mathcal{R}^* + \mathcal{K}$ -connected. Just like in Lemma 3.2.2 we will look at what this means for the original graphs $\mathcal{R}$ and Ω to prove that `BuildNW` returns the null network:

Just like in Lemma 3.2.2 the algorithm will in be done for a subset taxa $(X' \subseteq) X'' \subseteq \mathcal{X}$ for which $\mathcal{R}(\mathbb{B}|X'')$ is connected. This means $\mathcal{K}(\mathbb{B}|X'')$ will get constructed. The set X' is connected in $\mathcal{K}(\mathbb{B}|X'')$, thus this set becomes one vertex when $\Omega(\mathbb{B}|X'')$ is constructed. This means that in all iterations the set X' will stay together (Remember the remark after Lemma 3.2.1).

Eventually in one of the iterations $\mathcal{R}(\mathbb{B}|X')$ will be constructed. The graph $\mathcal{R}(\mathbb{B}|X')$ is connected. This means $\mathcal{K}(\mathbb{B}|X')$ will get constructed. The graph $\mathcal{K}(\mathbb{B}|X')$ is also connected. This means there is only one connected component π_1 in $\mathcal{K}(\mathbb{B}|X')$, thus $\Omega(\mathbb{B}|X')$ will exist of only one vertex. This means there is no strict subset $U \subset V(\Omega(\mathbb{B}|X'))$ where no arcs are going to. Thus the algorithm `BuildNW` will return the null network. $\square$

Notice: If there are *no* two given binets with the same set of leaves, the smallest version of a graph that is $\mathcal{R}^* + \mathcal{K}$ -connected is a graph with 4 taxa, just like in Figure 3.10. The smallest version of a graph $\mathcal{R}^* + \mathcal{K}$ with a cycle has 3 taxa.

If it is possible to have two given binets with the same set of leaves the smallest version of a graph that is $\mathcal{R}^* + \mathcal{K}$ -connected is of course a graph with 2 taxa, for instance if the binets $N(x; y)$ and $T(x, y)$ are given. The smallest version of a graph that contains a cycle also contains 2 taxa, this would be the case if the binets $N(x; y)$ and $N(y; x)$ are given.

There are now two reasons found for the algorithm to return the null network that turn out to cover all possibilities for the algorithm to return the null network.

Theorem 3.2.1. Given a set of binets $\mathbb{B}$, with taxa $\mathcal{X} = \{a, b, \dots\}$. The algorithm `BuildNW` will return the null network if and only if one of the following two cases appear:

1. $\exists$ a subset $X' \subset \mathcal{X}$ such that $\mathcal{R}^*(\mathbb{B}|X')$ is connected and $\mathcal{R}^*(\mathbb{B}|X') + \mathcal{K}(\mathbb{B}|X')$ has a directed cycle between connected components of $\mathcal{K}^*(\mathbb{B}|X')$.
2. $\exists$ a subset $X' \subset \mathcal{X}$ such that the graph $\mathcal{R}^*(\mathbb{B}|X') + \mathcal{K}(\mathbb{B}|X')$ is $\mathcal{R}^* + \mathcal{K}$ -connected.

Proof:

" $\Leftarrow$ " In Lemma 3.2.2 we have seen that if case 1 appears then the algorithm `BuildNW` would return

the null network. In Lemma 3.2.3 we have seen that if case 2 appears then the algorithm `BuildNW` would return the null network.

" $\Rightarrow$ " Suppose for a given set of binets $\mathbb{B} = \{B_1, \dots, B_n\}$ the algorithm `BuildNW` returned the null network. This means that in one of the iterations applied to a set $X' \subseteq \mathcal{X}$ there was no strict subset $U \subset V(\Omega(\mathbb{B}|X'))$ such that no arcs are going to U . There are two possibilities for that to happen:

- (a) There exists no strict subset $U \subset V(\Omega(\mathbb{B}|X'))$, because there is only one vertex $\pi_1 \in V(\Omega(\mathbb{B}|X'))$.
- (b) There exists a strict subsets $U \subset V(\Omega(\mathbb{B}|X'))$, but there are arcs going to all possible strict subsets U .

Suppose possibility (a) happens. The graph $\Omega(\mathbb{B}|X')$ exist of only one vertex if and only if $\mathcal{K}(\mathbb{B}|X')$ is one connected component. This means that for the set X' , the graph $\mathcal{R}^*(\mathbb{B}|X')$ is connected and $\mathcal{K}(\mathbb{B}|X')$ is also completely connected. Thus case 2 above is true.

Now consider possibility (b). There is always a finite number of taxa $\mathcal{X} = \{x_1, \dots, x_n\}$. This means that the number of taxa in the subset of $X' \subseteq \mathcal{X}$ for which the iteration returns a null network is also finite. This means the number of vertices $\{\pi_1, \dots, \pi_k\} \in V(\Omega(\mathbb{B}|X'))$ is also finite (because $1 < k < n$). Obviously case 2 is not true because $k > 1$. Suppose case 1 is also not true. Thus there is no directed cycle in $\Omega(\mathbb{B}|X')$.

Clearly the graph $\mathcal{R}^*(\mathbb{B}|X')$ has to be connected otherwise $\Omega(\mathbb{B}|X')$ would not be constructed.

If there are arcs going to every subset $U \subset V(\Omega(\mathbb{B}|X'))$, then every vertex in $\Omega(\mathbb{B}|X')$ has an arc going to it. Pick one of the vertices in $\Omega(\mathbb{B}|X')$ and call that vertex v_1 . There is an arc going to v_1 coming from an other vertex, call that vertex v_2 . Also there is an arc going to v_2 coming from an other vertex, v_3 and so on. Thus the arc going to v_{i-1} comes from v_i . When one of the arcs going to v_i comes from a vertex $v \in \{v_1, \dots, v_{i-1}\}$, there is a directed cycle $C = (v \rightarrow v_i \rightarrow v_{i-1} \rightarrow \dots \rightarrow v)$. But there were no directed cycles, thus for every v_i has to be an $v \in V(\Omega(\mathbb{B}|X')) \setminus \{v_1, \dots, v_{i-1}\}$ where the arc comes from. There are only $1 < k < n$ vertices in $\Omega(\mathbb{B}|X')$, thus if we arrive at vertex v_k there is no vertex $v_{k+1} \in V(\Omega(\mathbb{B}|X')) \setminus \{v_1, \dots, v_k\}$, because $V(\Omega(\mathbb{B}|X')) \setminus \{v_1, \dots, v_k\}$ is empty. This means there is no arc going to v_k , which means there would be a subset, namely $\{v_k\} \in V(\Omega(\mathbb{B}|X'))$, where no arcs are going to. This leads to a contradiction! Thus there has to be a directed cycle in $\Omega(\mathbb{B}|X')$.

A directed cycle between vertices of $\Omega(\mathbb{B}|X')$ means a directed cycle in the graph $\mathcal{R}^*(\mathbb{B}|X') + \mathcal{K}(\mathbb{B}|X')$ between connected components of $\mathcal{K}(\mathbb{B}|X')$. $\square$

3.3. Algorithm: `BuildNW2`

I want to make an adjusted version of `BuildNW`. If there does exist a supernetwork with all of the given binets, I want my algorithm to return that supernetwork. If there doesn't exist a supernetwork with all given binets, I want my algorithm to return a supernetwork with many of the given binets, instead of the null network.

From Section 3.2 it is clear when the algorithm `BuildNW` returns the null network. If I still want to construct a supernetwork with the given binets, I will have to remove some binets. The question is: which binets can I remove best, such that the binets left are able to form a supernetwork?

What I want to find are the binets that cause the algorithm to return a null network and to remove these binets such that I can build a supernetwork. Thus:

1. If $\exists$ a subset $X' \subset \mathcal{X}$ such that $\mathcal{R}^*(\mathbb{B}|X')$ is connected and $\mathcal{R}^*(\mathbb{B}|X') + \mathcal{K}(\mathbb{B}|X')$ has a directed cycle between connected components of $\mathcal{K}^*(\mathbb{B}|X')$, I want to remove a binet such that the directed cycle breaks.
2. If $\exists$ a subset $X' \subset \mathcal{X}$ such that $\mathcal{R}^*(\mathbb{B}|X') + \mathcal{K}(\mathbb{B}|X')$ is $\mathcal{R}^* + \mathcal{K}$ -connected, I want to remove a binet such that at least one of the graphs, $\mathcal{R}^*$ or $\mathcal{K}$ is not connected anymore.

First I will show some examples to get a clear image of how the problems can look like and how binets can be removed such that a supernetwork can be found.

Example 3.3.1. Suppose $\mathcal{R}^* + \mathcal{K}$ looks like Figure 3.11a (the same graph as in Figure 3.9). There is a cycle C between the connected components in $\mathcal{K}$, $C = (\{b\} \rightarrow \{a, e, d, c\} \rightarrow \{b\})$. To break the

directed cycle C does not seem very hard. Just remove one of the arcs from $\mathcal{R}^*(\mathbb{B})$ that are in the directed cycle. If I remove the dashed arc, the cycle is broken. I can build a supernetwork from the graph $(\mathcal{R}^* + \mathcal{K}) \setminus (a, b)$ that is left, which is represented in Figure 3.11b. This supernetwork contains all binets from which the graph $\mathcal{R}^* + \mathcal{K}$ was created except for the binet $N(b; a)$.

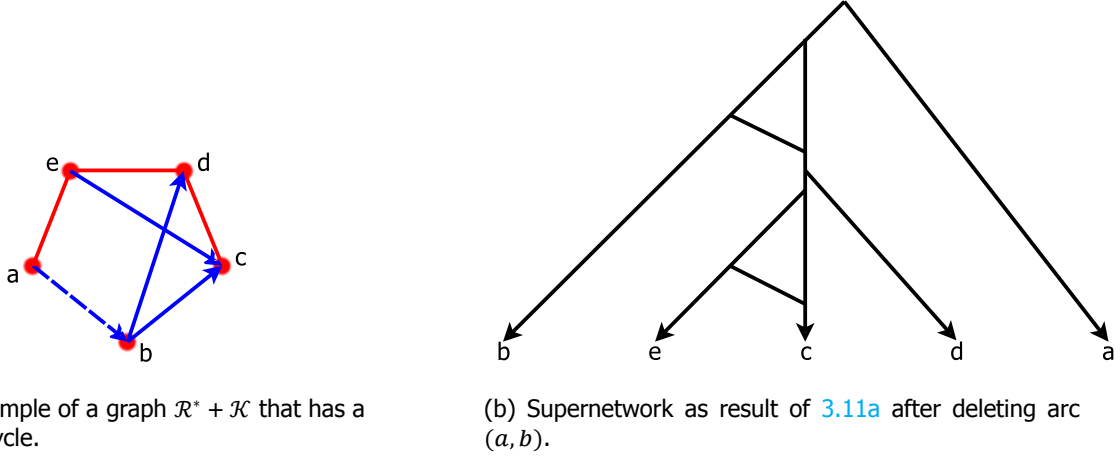


Figure 3.11: After deleting the arc (a, b) in (a), $\{a\}$ is in a different connected component of $\mathcal{R}^*$. Look at the connected component $\{b, c, d, e\} \in \mathcal{R}^*$, for this set of taxa there are two connected components $\{b\}, \{c, d, e\} \in \mathcal{K}$, where $\{b\}$ is now a subset where no arcs are going to. Next step is to construct $N(b, \text{BuildNW}(\mathbb{B}|\{c, d, e\}))$. For the set $\{c, d, e\}$ there are two connected components $\{c, e\}, \{d\} \in \mathcal{R}^*$. This results to the graph in (b).

What is interesting to see is that in this case also other edges or arcs could be removed to break the directed cycle. For any edge/arc $e \in \{\{a, e\}, (a, b), (b, c), (b, d), \{d, e\}, (e, c)\}$ it is possible to construct a supernetwork from the graph $(\mathcal{R}^* + \mathcal{K}) \setminus e$. Only the removal of edge (c, d) does not help.

Example 3.3.2. Suppose $\mathcal{R}^* + \mathcal{K}$ looks like Figure 3.10. In this case we could not find a solution, because this graph is $\mathcal{R}^* + \mathcal{K}$ connected. All edges and arcs have a role in the making the graph $\mathcal{R}^* + \mathcal{K}$ -connected. Just like in Example 3.3.1, there are more edges and arcs that can be removed such that it is possible to construct a supernetwork. In fact in this case for any edge/arc e that you remove in Figure 3.10 the graph $(\mathcal{R}^* + \mathcal{K}) \setminus e$ has a supernetwork.

Remark: Edge or arc e represented a binet, which means by removing e the information from this binet is lost and cannot be found in the resulting supernetwork.

Example 3.3.3. Suppose that a given set of binets results to the graph $\mathcal{R}^* + \mathcal{K}$ constructed in Figure 3.12.

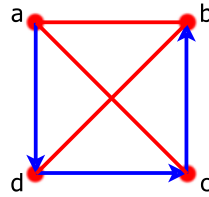


Figure 3.12: An example of a graph $\mathcal{R}^* + \mathcal{K}$ with multiple problems to construct a supertree.

This graph is similar to the one in Example 3.3.2, but in this case not just any edge or arc (x, y) can be removed such that the graph $(\mathcal{R}^* + \mathcal{K}) \setminus (x, y)$ has a supernetwork. This is because there are more problems in the graph $\mathcal{R}^* + \mathcal{K}$, which make that there does not exist a supernetwork containing all binets. Namely:

1. $\{a, b, c, d\}$ is connected in $\mathcal{R}^*$ AND connected in $\mathcal{K}$
2. $\{a, c, d\}$ has a directed cycle between connected components of $\mathcal{K}(\mathbb{B}|\{a, c, d\})$:
 $C_1 = (\{d\} \rightarrow \{a, c\} \rightarrow \{d\})$

3. $\{b, c, d\}$ has a directed cycle between connected components of $\mathcal{K}(\mathbb{B}) \setminus \{b, c, d\}$:
 $C_2 = (\{c\} \rightarrow \{b, d\} \rightarrow \{c\})$
4. $\{a, b, c, d\}$ has a directed cycle between connected components of $\mathcal{K}(\mathbb{B}) \setminus \{(a, c), (b, d)\}$:
 $C_3 = (\{d\} \rightarrow \{c\} \rightarrow \{a, b\} \rightarrow \{d\})$

If for instance edge (a, b) is removed problems 2 and 3 are still not solved. Thus for $(\mathcal{R}^* + \mathcal{K}) \setminus (a, b)$ there is still no supernetwork. The question therefore is:

"If there are n problems found in the graph $\mathcal{R}^* + \mathcal{K}$, will it be necessary to remove n edges or arcs to be able to make a supernetwork? Or is it possible to cleverly choose the edges or arcs such that you can tackle all problems in fewer moves?"

Yes, there is a way to choose the edges/arcs smart such that you can try to tackle most possible problems in one move. See the following procedure.

Define a weight for the edges/arcs.

Definition 3.3.1. Suppose e is an edge/arc in $\mathcal{R}^* + \mathcal{K}$.

$$w(e) := \# \text{ problems } e \text{ is in.}$$

The result of the weight-function w applied on the graph $\mathcal{R}^* + \mathcal{K}$ in Figure 3.12 is visible in Figure 3.13a. It is immediately clear that arc (d, c) is contained in the maximum number of problems, namely 4. This means that $\mathcal{R}^* + \mathcal{K} \setminus (d, c)$ has 4 problems less and thus a supernetwork can be constructed.

In Figure 3.13b the graph $\mathcal{R}^* + \mathcal{K} \setminus (d, c)$ is shown. The graph now consists of 2 connected components $\{a, d\}, \{c, b\} \in \mathcal{R}^*$, therefore there are no issues anymore to construct a supernetwork. See Figure 3.13c for the resulting supernetwork.

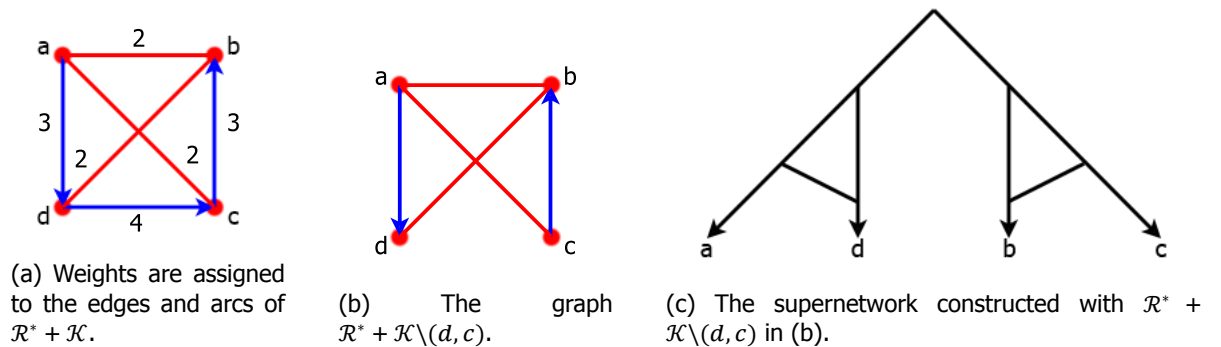
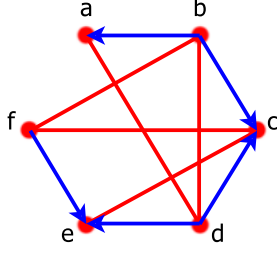


Figure 3.13: Steps for creating a supernetwork for $\mathcal{R}^* + \mathcal{K}$ in Figure 3.12.

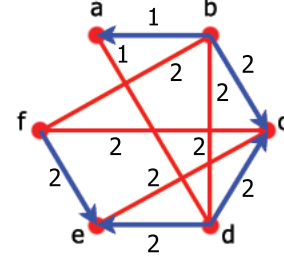
As just seen in Example 3.3.3, the problems causing that there can't be a supernetwork constructed containing all binets, can overlap. The method that is used in this example is kind of similar with the solutions we have seen in Chapter 2.

First the edges and arcs get a weight. In the algorithm `MinCutSupertree` this weight presents how valuable the relationship that this edge presents is. In Example 3.3.3 this weight presents how unreliable the relationship that this edge/arc presents is. In both methods is based on the weight decided if that edge/arc should be removed. In Example 3.3.3, an edge that causes a lot of problems has a high weight, such that there is a good reason to remove that edge.

To give a proper weight to an edge/arc you will have to observe in how many problems an edge/arc is contained. This is unfortunately not always easy. If the graph $\mathcal{R}^* + \mathcal{K}$ gets more complicated, it can be hard to tell how many times an edge or arc is contained in directed cycles or $\mathcal{R}^* + \mathcal{K}$ connected components. For instance a graph that is $\mathcal{R}^* + \mathcal{K}$ connected can contain a subset X' such that $(\mathcal{R}^* + \mathcal{K}) \setminus X'$ is also connected, see Figure 3.14a.



(a) This graph constructed for a set of binets $\mathbb{B}$. The graph is $\mathcal{R}^* + \mathcal{K}$ connected for all given binets, but also for the binets $\mathbb{B}|\{b, c, d, e, f\}$.



(b) Weights assigned to the edges/arcs of $\mathcal{R}^*(\mathbb{B}) + \mathcal{K}(\mathbb{B})$.

Figure 3.14

The graph $\mathcal{R}^* + \mathcal{K}$ in Figure 3.14a is connected for edges and arcs between the set $\{a, b, c, d, e, f\}$, but also for edges and arcs between the smaller set $\{b, c, d, e, f\}$. This means that the edge and the arc connected to vertex a are only in one problem and all the other edges and arcs are in two. This can be something you wouldn't see on first notice.

Steps to give weight to the edges:

The weight for the edges/arcs is defined as " $w(e) := \#$ problems e is in" as seen before.

To check in how many problems an edge/arc e is contained, there has to be checked for every subset $E' \in E(\mathcal{R}^* + \mathcal{K})$: $e \in E'$, if this subset E' is a problem. Here $E(\mathcal{R}^* + \mathcal{K})$ is the set of all edges and arcs in $\mathcal{R}^* + \mathcal{K}$.

This can be a lot of work, but there is a way to do this smart. Of course if the algorithm can still construct parts of a supernetwork, thus there are still strict subsets U where no arcs are going, there is no need to look for problems yet. The moment to start counting problems is when in the building process there are no such subsets U anymore. There is now a selection of edges/arcs $E'' \subseteq E(\mathcal{R}^* + \mathcal{K})$ that we need to check for problems.

As seen in section 3.2 all the possible problems are $\mathcal{R}^*$ connected. Thus we only have to look at subsets of edges/arcs $E' \subseteq E'' \subseteq E(\mathcal{R}^* + \mathcal{K})$ for which all vertices are connected in $\mathcal{R}^*$. Which means that subsets of $E(\mathcal{R}^* + \mathcal{K})$ where some of the vertices are only connected by edges in $\mathcal{K}$ are not interesting. For each such subset E' check if this subset causes a problem. A set E' causes a problem if for the vertices that this subset E' connects there exists no strict subsets U where no arcs are going.

The method to remove the edges with the highest weight is described in a new algorithm BuildNW2. See Algorithm 7 at the end of this section. This algorithm is just like BuildNW, but with a few adjustments:

- $\mathcal{R}^*(\mathbb{B})$ is constructed instead of $\mathcal{R}(\mathbb{B})$ and $\Omega(\mathbb{B})$
- The algorithm doesn't repeat itself for each connected component of $\mathcal{R}^*(\mathbb{B})$
- The colored part is added to the algorithm such that the algorithm always returns a supernetwork even though not all binets fit in.

With the notation $\mathbb{B}|E'$ I mean the binets represented by the edges/arcs in E' . Thus $\mathcal{R}^*(\mathbb{B}|E')$ means the graph $\mathcal{R}^*$ based on the binets $\mathbb{B}|E'$.

Example 3.3.4. Here BuildNW2 will be applied in an example. Suppose the graph $\mathcal{R}^*(\mathbb{B})$ is constructed and is connected. Thus $\mathcal{K}(\mathbb{B})$ can be constructed in the same figure. Suppose this gives us the graph in iteration 1 in Figure 3.15. For the first iteration there is a subset U of connected components in $\mathcal{K}(\mathbb{B}|X_i)$ such that no arcs are going to U , namely $\{a, b, d, e, f, g\}$.

Now start building the supernetwork $N = N(\text{BuildNW2}(\mathbb{B}|\{c\}); \text{BuildNW2}(\mathbb{B}|\{a, b, d, e, f, g\}))$.

This means we will have a second iteration for the set of binets $\mathbb{B} = \mathbb{B}|\{a, b, d, e, f, g\}$. This gives the graph in iteration 2, step 1. There are now two connected sets $X_1 = \{a, b, d, f, g\}$ and $X_2 = \{e\}$ in

$\mathcal{R}^*(\mathbb{B})$. Thus $\mathcal{K}(\mathbb{B})$ has to be constructed for both these sets separately. That is why the edge $\{d, e\}$ in iteration 1 is not visible anymore in iteration 2.

The graph $\mathcal{R}^*(\mathbb{B}|X_1) + \mathcal{K}(\mathbb{B}|X_1)$ has no subset U of connected components in $\mathcal{K}(\mathbb{B}|X_1)$ such that no arcs are going to U . Thus the algorithm will assign weights to the edges and arcs in $E(\mathcal{R}^*(\mathbb{B}|X_1) + \mathcal{K}(\mathbb{B}|X_1))$. The problems for this part of the graph are:

1. For the edges/arcs between $\{a, b, d, f, g\}$ the graph is $\mathcal{R}^* + \mathcal{K}$ -connected
2. For the edges/arcs between $\{a, b, f, g\}$ the graphs $\mathcal{R}^* + \mathcal{K}$ -connected
3. There is a directed cycle $C = (\{a\} \rightarrow \{g, b\} \rightarrow \{a\})$

See iteration 2, step 2 for the weights of each edge/arc. The edge $\{g, b\}$ and arcs $(a, b), (a, g)$ all have a maximum weight: 3. It does not matter which edge/arc with maximum weight gets removed, all removals will result in 3 problems less. Here edge $\{g, b\}$ is deleted which results in the graph in iteration 2, step 3. Because there were only 3 problems in this graph, there are now no problems anymore to find a subset U of connected components in $\mathcal{K}(\mathbb{B}|X_i)$ such that no arcs are going to U , namely $U = \{a, b, d, f\}$ satisfies.

Now we can continue building the supernetwork. Two edges will be constructed, one for $\{a, b, d, f, g\}$ and one for $\{e\}$. The root of $N = N(\text{BuildNW2}(\mathbb{B}|\{g\}); \text{BuildNW2}(\mathbb{B}|\{a, b, d, f\}))$ will be attached to one of the edges.

A new iteration will start for the binets $\mathbb{B} = \mathbb{B}|\{a, b, d, f\}$, see iteration 3. Walking through all steps again eventually there will be one more iteration. As you can see in iteration 4 is in iteration 3 is $U = \{d\}$ chosen as subset where no arcs are going to and $L = \{a, b\}$. The final supernetwork is represented in Figure 3.16.

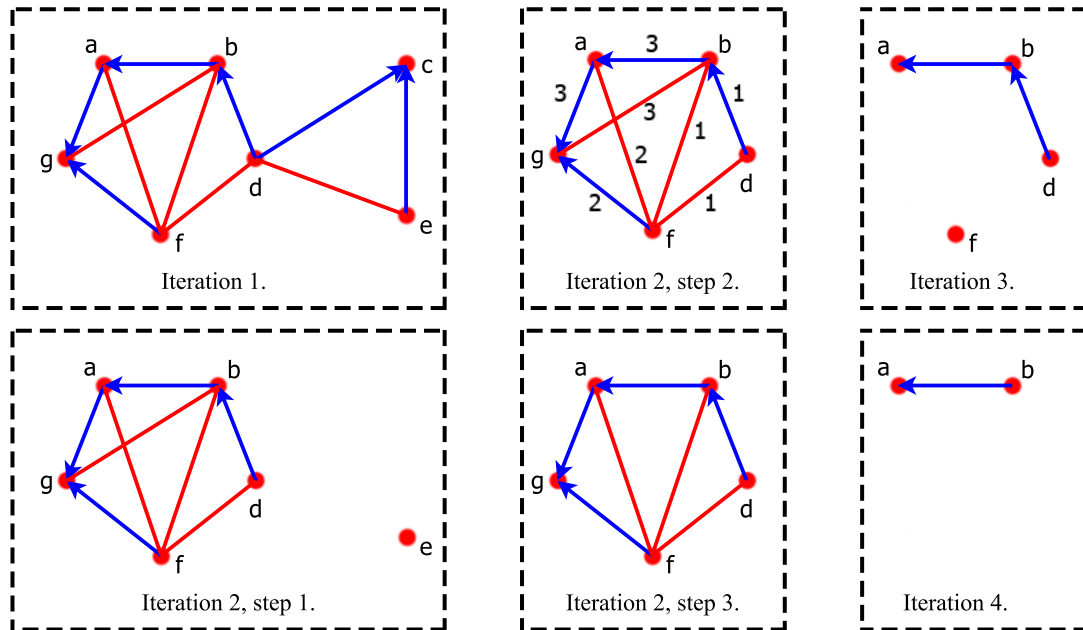


Figure 3.15: All iterations of the `BuildNW2` algorithm on a set of binets.

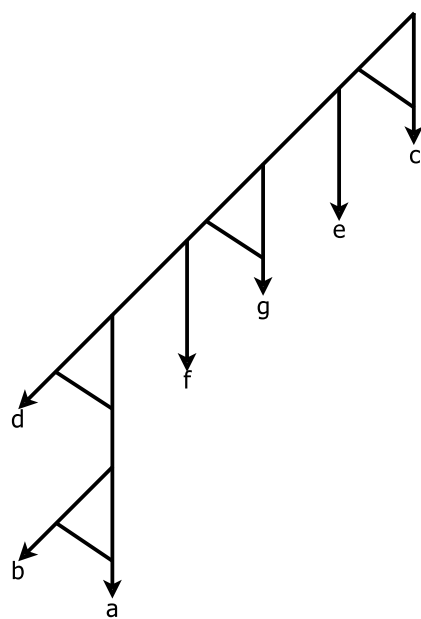


Figure 3.16: The supernet constructed by the graphs in the iterations of Figure 3.15.

Algorithm 7 BuildNW2($\mathbb{B}$)**Input:** A set of binets $\mathbb{B} = \{B_1, \dots, B_n\}$ and a set of taxa $\mathcal{X} = \{a, b, \dots\}$ **Output:** A binary supernetwork N

```

1: Construct  $\mathcal{R}^*(\mathbb{B})$ : A vertex for each taxon in  $\mathcal{X}$  and an arc  $(x, y)$  if  $N(y; x) \in \mathbb{B}$ 
2: for each connected component  $X_1, \dots, X_p$  in  $\mathcal{R}^*(\mathbb{B})$  do
3:   if  $X_i$  consists of a single node  $x$  then
4:      $N_i = x$ 
5:   else
6:     Construct  $\mathcal{K}(\mathbb{B}|X_i)$  in the same figure as  $\mathcal{R}^*(\mathbb{B}|X_i)$  ( $i = 1, \dots, p$ )
7:     Search for a strict subset  $U$  of connected components in  $\mathcal{K}(\mathbb{B}|X_i)$  such that no arcs are going
      to  $U$ 
8:     if there is no subset  $U$  that satisfies then
9:        $w(e) = 0 \ \forall$  edges and arcs  $e \in E(\mathcal{R}^*(\mathbb{B}|X_i) + \mathcal{K}(\mathbb{B}|X_i))$ 
10:      for every subset  $E' \subseteq E(\mathcal{R}^*(\mathbb{B}|X_i) + \mathcal{K}(\mathbb{B}|X_i))$  of edges/arcs do
11:        if all vertices connected by  $E'$  are  $\mathcal{R}^*(\mathbb{B}|E')$ -connected then
12:          if  $\mathcal{K}(\mathbb{B}|E')$  is connected then
13:            for every edge/arc  $e \in E'$  do
14:               $w(e) = w(e) + 1$ 
15:            end for
16:          end if
17:          if there is a cycle  $C = (\pi_1 \rightarrow \dots \rightarrow \pi_m)$  between connected components  $\pi_l \in$ 
 $\mathcal{K}(\mathbb{B}|E')$  ( $l = 1, \dots, m$ ) then
18:            for every edge/arc  $e \in E'$  do
19:              if  $(\mathcal{R}^*(\mathbb{B}|X_i) + \mathcal{K}(\mathbb{B}|X_i)) \setminus e$  does not contain the cycle  $C$  then
20:                 $w(e) = w(e) + 1$ 
21:              end if
22:            end for
23:          end if
24:        end if
25:      end for
26:      Pick an edge/arc  $e$  with maximum weight
27:       $B :=$  the binet that  $e$  represents
28:       $N_i = \text{BuildNW2}((\mathbb{B}|X_i) \setminus B)$ 
29:    end if
30:    Construct  $N_{up} = \text{BuildNW2}(\mathbb{B}|U)$ 
31:    Call  $L = \mathcal{X} \setminus U$  and construct  $N_{low} = \text{BuildNW2}(\mathbb{B}|L)$ 
32:    Build  $N_i = N(N_{low}; N_{up})$ 
33:  end if
34:  if  $p = 1$  then
35:     $N = N_1$ 
36:  else
37:    Construct  $N$  with a node  $r$  for the root and with edges from  $r$  to the roots of the networks
 $N_i$  ( $i = 1, \dots, p$ ), such that  $N$  is binary
38:  end if
39: end for
40: return  $N$ 

```

3.4. Lower bound binets in supernetwork

In Section 3.3 an algorithm has been found that returns a supernetwork even if not all binets can be included. The challenge for this project was "Given a set of binets find a supernetwork with *most* of these binets".

The new algorithm does find a supernetwork but it is not sure if this super network contains *most* of the binets. What I will prove in this section is that there is a lower bound on the fraction of the binets that should at least be able to fit in the supernetwork. Thus if there are less binets contained in the supernetwork than that lower bound, you will be sure there must be a better solution with more binets contained in the supernetwork.

In this section I will prove that this lower bound is $\frac{1}{3}$.

Notice: It is not possible to give a higher bound than $\frac{1}{3}$. If for example the binets $T(a, b)$, $N(a; b)$ and/or $N(a; b)$ are given, only one of them can be chosen to construct a supernetwork.

Theorem 3.4.1. At least $\frac{1}{3}$ of the given binets must fit in a supernetwork.

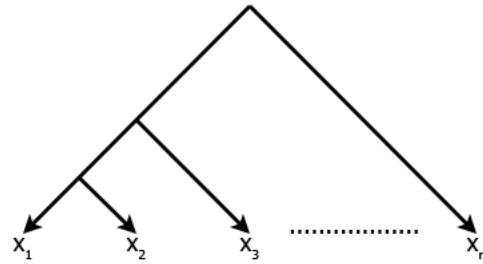
Proof:

Suppose there is a given set of binets for the taxa $\mathcal{X} = \{x_1, \dots, x_n\}$, which can be numbered arbitrarily. For each pair of taxa $\{x_i, x_j\}$ the binets $T(x_i, x_j)$, $N(x_i; x_j)$ and/or $N(x_j; x_i)$ could be given.

If all the network binets get removed, all binets left are of the form $T(x_i, x_j)$. With only tree binets left, there will be no problem to construct a supernetwork. $R^*(\mathbb{B})$ will be completely disconnected, thus the supernetwork will be constructed by making a binary tree with leaf set $\mathcal{X}$, see Figure 3.17b.



(a) $R^*(\mathbb{B})$ if there are only tree binets.



(b) A supernetwork (or supertree) if there are only tree binets.

Figure 3.17

If only binets of the form $N(x_i; x_j)$ are left, where $1 \leq i < j \leq n$, there is also no problem for constructing a supernetwork. $R^*(\mathbb{B})$ will have arcs from some of the nodes in the set $\{x_2, \dots, x_n\}$ to x_1 , from some of the nodes in the set $\{x_3, \dots, x_n\}$ to x_2 and in general from nodes in the set $\{x_{i+1}, \dots, x_n\}$ to x_i . See Figure 3.18a. (This figure seems to imply that all $N(x_i; x_j)$, where $1 \leq i < j \leq n$, are given, but of course it could be that $N(x_i; x_j)$ for certain i, j is not given. Then there would be no arc (x_j, x_i) , but still the following can be applied.) This means that for the first iteration $\{x_2, \dots, x_n\}$ can be chosen as U , for the second iteration can be $U = \{x_3, \dots, x_n\}$ etc.¹ Which results to a supernetwork represented in Figure 3.18b.

¹There are a lot of different possibilities for choosing U . It could also be that for the first iteration $U = x_n$ for the second $U = x_{n-1}$ etc. and this leads to a whole different graph.

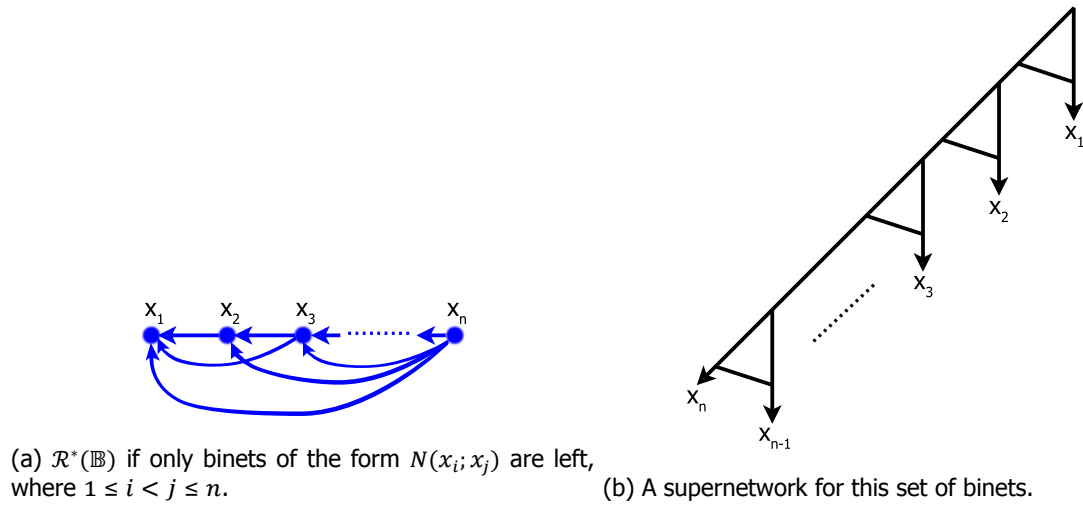
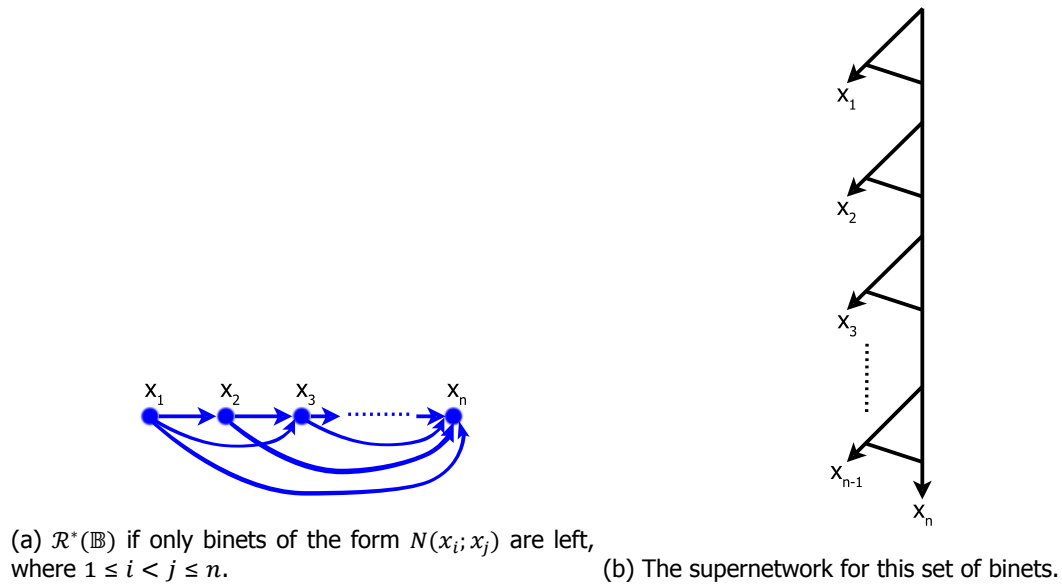


Figure 3.18

For the set of binets of the form $N(x_j; x_i)$, where $1 \leq i < j \leq n$ the same applies. Again there is no problem for constructing a supernet. $R^*(\mathbb{B})$ will have an arc from x_1 to nodes in the set $\{x_2, \dots, x_n\}$, x_2 will have an arc to nodes in the set $\{x_3, \dots, x_n\}$ and in general x_i will have an arc to nodes in the set $\{x_{i+1}, \dots, x_n\}$. See Figure 3.19a. This means for the first iteration U can be $\{x_1\}$ for the second iteration $U = x_2$ etc. Which eventually results in a super network represented in Figure 3.19b



There are three possibilities:

1. $\geq \frac{1}{3}$ of the binets is of the form $T(x_i, x_j)$.
2. $\geq \frac{1}{3}$ of the binets is of the form $N(x_i; x_j)$.
3. $\geq \frac{1}{3}$ of the binets is of the form $N(x_j; x_i)$.

Where $1 \leq i < j \leq n$. Thus as seen above if possibility 1 is true there is a supernet for at least $\frac{1}{3}$ of the binets. If possibility 1 isn't true possibility 2 or 3 must be true and for both cases this means there is a supernet for at least $\frac{1}{3}$ of the binets. $\square$

4

Discussion

In this report a new algorithm, `BuildNW2`, for constructing a supernet from a given set of binets has been presented. `BuildNW2` finds a supernet for a given set of binets even when some of the binets contradict.

In this chapter, I will explain some choices I had to make for the construction of a supernet. I will discuss them and make some suggestions to improve the algorithm in future work. At the end of the chapter I will get back on the research question of this thesis. Also I will make a suggestion for maybe another application of the algorithm.

Construction based on the choices

One of the first steps in the algorithm `BuildNW2` is constructing the graph $\mathcal{R}^*(\mathbb{B})$. If $\mathcal{R}^*(\mathbb{B})$ consists of $n > 1$ connected components a subnetwork N_i ($i = 1, \dots, n$) will be build for each connected component. The supernet N will be constructed with a node r for the root and with edges from r to the roots of N_i , such that N is binary. The networks N_i will be attached in a random order, thus the supernet N could have different outcomes by the choice of the order. If for instance $n = 5$ the subnetworks N_i could be attached as in Figure 4.1a or as in Figure 4.1b. There are $5!$ possible ways to connect the subnetworks. If the supernet in Figure 4.1a is chosen, the taxa in N_1 seem closer related to the taxa in N_2 than to the taxa in N_5 , while actually we do not know anything about it.

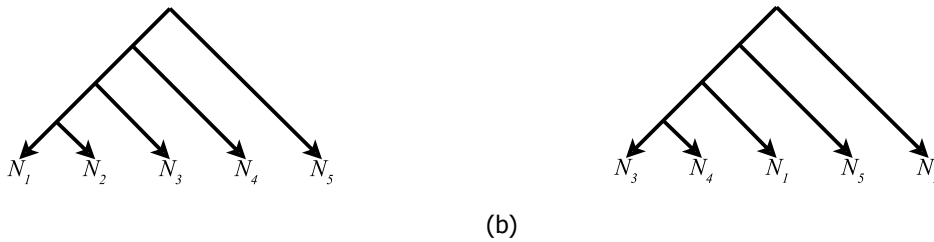


Figure 4.1: Two different supernet structures that could represent the same set of binets.

The reason the subnetworks were attached like this is because we wanted to make a binary level-1 supernet. In the supertree methods in Chapter 2 I have seen that the supertrees did not have to be binary. Why should the supernet be binary? I think it would be better to make the resulting supernet as much as possible based on the information the binets gave us and not based on the choices the constructor makes. That is why I would attach the subnetwork as represented in Figure 4.2 (inspired by the algorithms of supertrees).

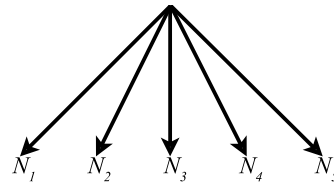
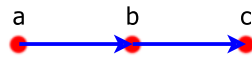


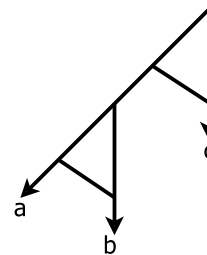
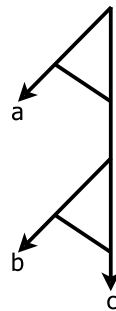
Figure 4.2: A different way to attach the subnetworks to the root of the supernetwork.

Step 7 of the algorithm `BuildNW2` are also based on a choice. In this step there has to be searched for a subset U where no arcs are going to. Sometimes there are more options for such subsets to choose from. For instance, suppose the graph $\mathcal{R}^* + \mathcal{K}$ looks like Figure 4.3a. This gives two options for constructing the supernetwork, see Figures 4.3b and 4.3c. Figure 4.3b implies taxa b and c are closer related than a and b , thus the most recent common ancestor of $\{b, c\}$ is a descendant of the most recent common ancestor of $\{a, b\}$. While in Figure 4.3c it seems that a and b are closer related than b and c .

For the same set of binets different supernetworks can be constructed that imply different relations between species.



(a) This example of a graph $\mathcal{R}^* + \mathcal{K}$ has different solutions.



(b) A supernetwork solution of the graph $\mathcal{R}^* + \mathcal{K}$ in Figure 4.3a.

(c) Another supernetwork solution of the graph $\mathcal{R}^* + \mathcal{K}$ in Figure 4.3a.

Figure 4.3

The difference between two supernetworks caused by the choice of U can be a lot larger if a larger set of binets is given. I do not know if there is a solution for this problem, because the given binets contain no information which connection is “stronger”.

Suppose that weights would be assigned to the binets relating the certainty and the closeness of the historical relation described by that binet. Then there could be a solution for this problem. This I will discuss in the next section.

There is another step in the algorithm `BuildNW2` which is based on a choice. If in line 26 there are multiple edges with a maximum weight, there will be only one edge picked to remove, while all the other edges had just as much reason to get removed. The reason to pick only one edge is because the goal was to put most possible of the given binets in the supernetwork. If the goal would just be to find a supernetwork based on the given binets, it might be better to remove all the edges with a maximum-weight. In the algorithm `MinCutSupertree` all edges that are in a minimum-weight cut set are removed. For the same reason we can apply a similar approach in the algorithm `BuildNW2`: All the edges that are most contradicted by other edges get removed.

In Figure 4.4 this idea is applied to the same graph as in Example 3.3.4. If we also omit the restriction that the supernetwork should be binary, the resulting supernetwork will look like Figure 4.5.

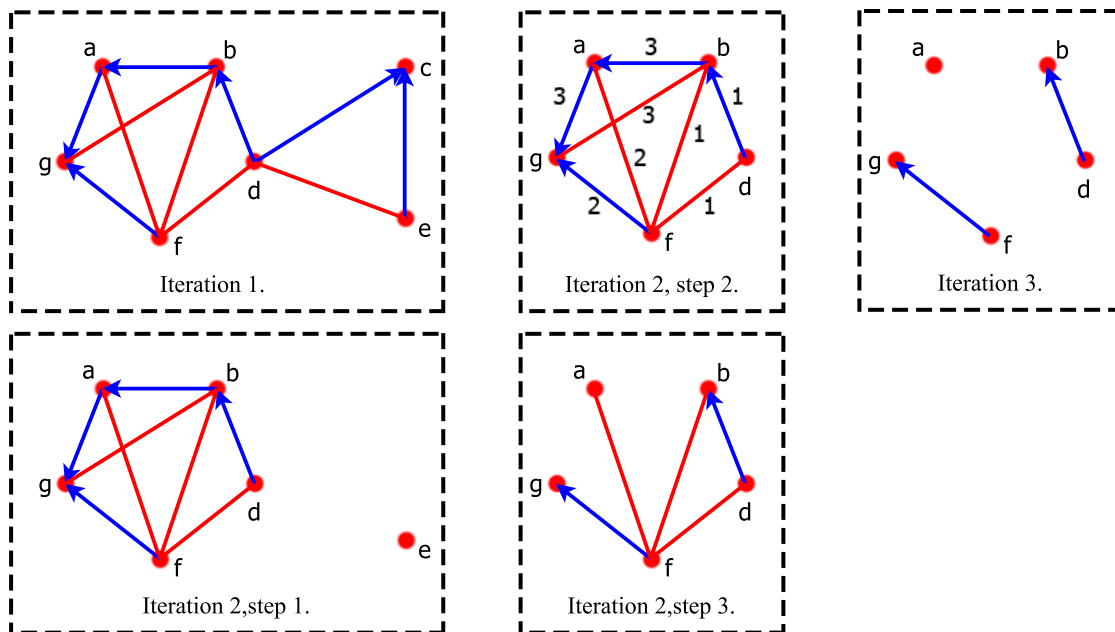


Figure 4.4: Iteration 1 starts with the same graph as in Example 3.3.4. But in step 2 of iteration 2 *all* edges with maximum weight get removed, instead of just one.

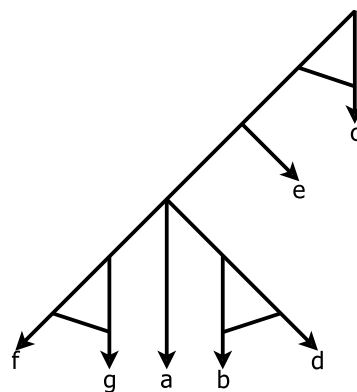


Figure 4.5: A supernet as result of the new ideas.

Option to assign weights to the binets

I can imagine that for a given set of binets, there could be binets with a higher or a lower level of certainty. That is why I think it might be good to also have an option where there could be weights assigned to the binets. In this way you can represent more of the information of the binets that have a higher certainty of than binets with a low certainty in the supernet.

In the `MinCutSupertree` algorithm there is an option for a weight-function for the trees: $w : \mathcal{T} \rightarrow \mathbb{Q}^+$. I think it would be good to have something just like that in the algorithm `BuildNW2` for binets, thus a weight-function $v : \mathbb{B} \rightarrow \mathbb{Q}^+$ (w is already used in the algorithm `BuildNW2`).

How to apply weights to get a better supernet?

Right now the choice of U in step 7 is up to the constructor of the supernet, like discussed above. If there are more subsets U that satisfy step 7 could be adjusted as follows:

Search for strict subsets $U_1, \dots, U_k$ of connected components in $\mathcal{K}(\mathbb{B}|X_i)$ such that no arcs are going to U_j ($j=1, \dots, k$)
for all subsets U_j ($j = 1, \dots, k$) **do**

```

 $\mathbb{B}_{U_j} = \{\text{binets } B \in \mathbb{B} : \text{binets are represented by edges/arcs } e \text{ with one endpoint in } U_j \text{ and one in } \overline{U_j}\}$ 
 $u(U_j) = \sum_{B \in \mathbb{B}_{U_j}} v(B)$  % assigns a weight  $u$  for each  $U_j$ 
end for
Pick  $U_j$  with minimum weight  $u(U_j)$ 

```

Another step where the weights could be used to make a founded choice is step 26. In step 26 there can be several edges that have the highest weight and like described above there is just one edge picked to remove. With a weight function for every binet there might be a better reason to remove one edge before the other one. Suppose before step 26 the following step is added:

```

for each edge  $e \in \mathcal{R}^*(\mathbb{B}|X_i) + \mathcal{K}(\mathbb{B}|X_i)$  do
   $B = \text{the binet this edge represents}$ 
   $w(e) = w(e)/v(B)$ 
end for

```

These are just some ideas of how the algorithm could be adjusted if a weight-function $v : \mathbb{B} \rightarrow \mathbb{Q}^+$ would be added to the input of the algorithm. This can be worked out in more detail in future work.

Lower bound with no two binets with the same pair of leaves

In Section 3.4 I have showed that there is a lower bound of $\frac{1}{3}$ of the total number of binets that should be able to fit in a supernetwork. This lower bound applies for the case two taxa can be presented by several binets. Thus taxa $\{x, y\}$ can be in the binets $T(x, y), N(x; y)$ and $N(y; x)$. Which means, if only these binets were given exactly $\frac{1}{3}$ of the binets can fit in a supernetwork.

If there are no two binets with the same pair of leaves this lower bound is probably higher. That is why the following question might also be interesting:

“What is the lower bound of binets that should be able to fit in a supernetwork if there are no two binets with the same pair of leaves?”

My presumption is that at least $\frac{2}{3}$ of the given binets should be able to fit in a supernetwork, if there are no two binets with the same pair of taxa as leaves. One of the reasons I suspect this is because if there are no two binets with the same pair of leaves there are at least three binets required to lead to a problem. See for instance the examples in Figures 3.5 and 3.6. These problems show that it is not possible to give a higher lower bound than $\frac{2}{3}$.

To prove that this is or is not a lower bound is also something that can be done in future work.

Is the research goal: “Find a supernetwork containing *most* of the given binets” achieved?

The goal of the research was to find a method that would find a supernetwork containing *most* of the given binets. BuildNW2 is a method that returns a supernetwork for the given binets, but I have not proven yet whether it returns a supernetwork with *most* of the given binets. This is also future work.

Trinets

This report was only about looking for an algorithm to construct a supernetwork from a given set of binets. In [5] an algorithm for constructing a supernetwork from a given set of trinets is given. This algorithm also does not return a supernetwork when there is no supernetwork containing all trinets. It would be interesting to see if we could apply a similar solution, like BuildNW2, to construct a supernetwork from a given set of trinets if there is no supernetwork containing all trinets.

Bibliography

- [1] Naturalis, *Natuurinformatie: Hoe groot is de huidige biodiversiteit?* <http://www.natuurinformatie.nl/nnm.dossiers/natuurdatabase.nl/i002078.html> (2000), accessed: 2015-05-18.
- [2] C. o. D. Peter Chen, *Figure: Amniotes comprise*, <http://bio1151.nicerweb.com/Locked/media/ch34/amniotes.html> (2010), accessed: 2015-05-19.
- [3] C. W. . Informatica, *Figure: A level-1 phylogenetic network*, <https://www.cwi.nl/node/2168>, accessed: 2015-05-19.
- [4] L. Nakhleh, *Evolutionary phylogenetic networks: models and issues*, in *Problem solving handbook in computational biology and bioinformatics* (Springer, 2011) pp. 125–158.
- [5] K. Huber, L. van Iersel, V. Moulton, C. Scornavacca, and T. Wu, *Reconstructing phylogenetic level-1 networks from nondense binet and trinet sets*, arXiv preprint arXiv:1411.6804 (2014).
- [6] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman, *Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions*, SIAM Journal on Computing **Volume 10; Issue 3**, pages 405/421 (1981).
- [7] O. R. Bininda-Emonds, J. L. Gittleman, and M. A. Steel, *The (super) tree of life: procedures, problems, and prospects*, Annual Review of Ecology and Systematics **Volume 33**, pages 265/289 (2002).
- [8] C. Semple and M. Steel, *A supertree method for rooted trees*, Discrete Applied Mathematics **Volume 105; Issue 1-3**, pages 147/158 (October 2000).
- [9] M. P. Ng and N. C. Wormald, *Reconstruction of rooted trees from subtrees*, Discrete Applied Mathematics **Volume 69; Issue 1-2**, pages 19/31 (August 1996).
- [10] R. D. Page, *Modified mincut supertrees*, Algorithms in Bioinformatics, Lecture Notes in Computer Science **Volume 2452; Issue 3**, pages 537/551 (2002).
- [11] S. Snir and S. Rao, *Using max cut to enhance rooted trees consistency*, IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) **Volume 3; Issue 4**, pages 323/333 (October 2006).