Reconstructing a level-1-network from quartets
Bachelor project

Willem Sonke
Eindhoven University of Technology

Supervisors:
Judith Keijsper & Rudi Pendavingh

May 23, 2013
Abstract

This paper deals with the problem of reconstructing a phylogenetic network from quartets. Specifically a polynomial-time algorithm was implemented to reconstruct trees and level-1-networks from quartet data that satisfies certain conditions. This algorithm uses linear algebra methods to construct the set of orders the taxa can have along the boundary when drawing a planar embedding of the network. Then, the network can be reconstructed. Several experiments are performed to validate the implementation and measure the accuracy and performance of the algorithm.
## Contents

1 Introduction ................................................................. 4

2 Networks and quartets .................................................... 5

3 Cyclic orders ..................................................................... 7
   3.1 Cyclic orders and encodings ........................................ 7
   3.2 The subspace $\mathbb{U}^X$ .................................................. 8
   3.3 Requirements on the quartet set ...................................... 9

4 Sets of splits ..................................................................... 10
   4.1 Splits ........................................................................ 11
   4.2 Cross-free and laminar sets .......................................... 12

5 Algorithm .......................................................................... 14
   5.1 Determining $\mathbb{U}^X(\Omega)$ as an affine subspace .......... 14
   5.2 Finding one cyclic order ................................................ 16
   5.3 Constructing a level-1-network ....................................... 18
   5.4 Detecting non-cyclic vectors .......................................... 21
   5.5 The entire algorithm .................................................... 22

6 Results .............................................................................. 22
   6.1 Performance on trees .................................................... 23
   6.2 Accuracy when leaving out quartets from trees ................ 23

7 Conclusion .......................................................................... 25
1 Introduction

In phylogenetics, a common problem is reconstructing the ancestry of a group of species. One way of approaching this is by constructing a so-called phylogenetic tree. Such a tree indicates how the species (more commonly called taxa) evolved.

Phylogenetic trees can either be rooted or unrooted. Rooted trees are appropriate when one species is known to be the common ancestor of all others. For rooted trees, already a lot is known. In many cases however, unrooted trees are more useful; for example, it can happen that we do not know the ancestor, or that we are not interested in determining it. This paper will only deal with unrooted phylogenetic trees and networks.

It is possible to construct a phylogenetic tree by hand, but techniques are being developed to use certain data to construct trees automatically. A major example is the usage of DNA information. When the DNA sequence of several taxa is known, one can find patterns in it, known as quartets.

However, trees are often not enough to handle real data. When species have a non-treelike evolution, for example when species exchange genetic material, a phylogenetic tree is not appropriate to model the situation. To cope with this, an extension of phylogenetic trees was created: the phylogenetic network. In such a network, cycles are allowed to model a non-treelike evolution. In many cases it is sufficient to consider so-called (unrooted) level-1-networks, phylogenetic networks that are not ‘too connected’. Level-k-networks in general were defined first in 2011 by Gambette, Berry and Paul [2], to distinguish between networks of different levels of connectedness. We will give a formal definition in section 2.

Deciding whether a phylogenetic tree exists that is consistent with a set of quartets \( Q \) is known to be NP-complete in the general case. This was proven by Steel in 1992 [5] (Theorem 1), by reduction from another NP-complete problem called Betweenness. The same holds for level-1-networks, as proven by Gambette [2] (Theorem 4).

However, if we place requirements on \( Q \), we can do better. Gambette already showed that an \( O(n^4) \) algorithm exists that decides whether \( Q \) contains all quartets consistent with a level-1-network. There, also the concept of a dense quartet set is introduced. A set of quartets is dense if it contains at least one quartet on every 4-subset of taxa.

Gambette mentions the problem whether it is possible to decide in polynomial time whether a level-1-network exists with which a dense quartet set is consistent. This report describes an implementation of a novel algorithm by Keijzer and Pendavingh [3] that indeed does this, and analyses its performance. The algorithm can even loosen the requirements a bit further: \( Q \) does not have to be dense; the algorithm also works for so-called 4-out-of-5 quartet sets. We will see this in section 3.3.

To test the algorithm, we implemented it in Java, and several experiments were performed to measure the algorithm’s accuracy and performance.

Outline of this paper  The first part of this paper reviews the theoretical backgrounds of the algorithm. We start by giving the basic definitions and posing the main question of the paper in section 2. We then study an intermediate representation of the network, which we can construct from the quartets, in section 3; in section 4 we will see how to retrieve a network from this representation. The remainder of the paper contains an elaborate description of the algorithm: section 5 describes the algorithm in detail, and in section 6 we performed some experiments using the implementation and report the results.
Networks and quartets

In this section we start with some basic definitions, and we discuss the main question. In this entire paper, graphs are considered to be undirected, except where noted otherwise.

**Definition 2.1** (Networks and trees). A connected graph $N$ is called a (phylogenetic) network on a set of taxa $X$ if all its vertices have degree 3 (internal vertices) or 1 (leaves), and if its leaves are bijectively labeled by $X$. A (phylogenetic) tree is a network without cycles.

We will now introduce the concept of a level-$k$-network; for this, we choose the same definition as Gambette [2].

**Definition 2.2** (Level-$k$-networks). Let $N$ be a network. We first define the following.

- A **cut-edge** of $N$ is an edge $e$ such that $N$ is not connected anymore if $e$ is removed.
- Now, a **blob** of $N$ is a maximal subgraph of $N$ not containing any cut-edge.

A network $N$ is called a level-$k$-network (for some $k \in \mathbb{N}$) if it is possible to remove at most $k$ edges from every blob of $N$ to obtain a tree.

Some examples of networks are drawn in Figure 1. Since in this paper we will only discuss trees and level-1-networks, an easier characterization of level-1-networks is useful. We first need some definitions to accomplish this.

Two paths in a graph are **vertex-disjoint** if there is no vertex in the graph that is used by both paths. We call two paths **internally vertex-disjoint** if these paths are vertex-disjoint after removing the start and end point of both paths. A set consisting of three or more paths is called (internally) vertex-disjoint if these paths are pairwise (internally) vertex-disjoint. Similarly, two paths are **edge-disjoint** if they do not share any edges. A circuit is a connected graph in which all vertices have degree 2.

The following lemma now provides a simple characterization of level-1-networks.

**Lemma 2.3.** Let $N$ be a network. The following statements are equivalent:

1. $N$ is a level-1-network;
2. every two vertices in $N$ are connected by at most 2 internally vertex-disjoint paths.
**Proof.** 
1. ⇒ 2. Take two arbitrary vertices v and w from N. If they are in different blobs, any path between v and w must include the cut-edge between the blobs. So, there are not even two internally vertex-disjoint paths connecting v and w. Thus, assume they are in the same blob. Now, if there are three or more internally vertex-disjoint paths, then we are not able to remove one edge from the blob and get a tree, since there will be a cycle left. So, there must be at most 2 internally vertex-disjoint paths between v and w.

2. ⇒ 1. Take a blob from N. Since every two vertices in N are connected by at most 2 internally vertex-disjoint paths, every two vertices v and w in the blob will be connected by exactly 2 internally vertex-disjoint paths (because a blob does not contain any cut-edges). Thus, the graph induced by the vertices in the blob has to be a circuit. We thus are able to remove an arbitrary edge from every blob, and we will obtain a tree.

Stated otherwise, the lemma shows that a network is a level-1-network if and only if its blobs are circuits. It follows that level-1-networks are planar.

Having defined level-1-networks, we will now continue by considering quartets.

**Definition 2.4 (Quartets).** A quartet on X is a partition, in two sets of two elements, on four distinct elements from X. For a, b, c, d ∈ X we denote the quartet \{(a, b), (c, d)\} by ab|cd. We say a network N is consistent with a quartet ab|cd if N contains two vertex-disjoint paths from a to b and from c to d, respectively.

A network N is consistent with a set of quartets Ω if it is consistent with all ab|cd ∈ Ω.

For example, the network given in Figure 1b is consistent with ab|ed, while the tree in Figure 1a is not. We often represent a quartet by a small tree, as shown in Figure 2, since such a tree depicts the smallest possible network consistent with the quartet. Furthermore, note that a network N is consistent with a quartet ab|cd if and only if this tree representation of ab|cd is a subgraph of N. Note that, for a given set of four elements, say \{a, b, c, d\}, there are only three distinct quartets possible: ab|cd, ac|bd and ad|bc. These indeed exactly correspond to the trees possible on those four elements.

It is of course possible to enumerate all quartets consistent with a network N. This can be done in \(O(n^5(1 + \alpha(n, n)))\) time [2], where n is the number of taxa and \(\alpha\) is the inverse of the Ackermann function; \(\alpha\) only grows extremely slowly. The main question posed in this paper is how to do this the other way round. So, we are given a set of quartets, and we are required to find a network consistent with these quartets. More formally, we will study the following problem.

**Problem 2.5 (Level-k-Quartet-Consistency).** Given \(k \in \mathbb{N}\) and a set of quartets Ω, decide whether a level-k-network exists consistent with Ω.

In 1992, Steel proved that Level-0-Quartet-Consistency is NP-complete [5]. Also Level-1-Quartet-Consistency is NP-complete [2]. However, when requirements are placed on the quartets in Ω, it is possible to solve the problem (and also reconstruct such a
Cyclic orders

The proposed algorithm first constructs an intermediate representation for the network, namely the set of so-called cyclic orders of the network. In this section, we will present a method to convert a given set of quartets to a set of cyclic orders of the network the quartets are taken from. Then, in the next sections, we will use this representation to reconstruct the network.

3.1 Cyclic orders and encodings

Let us first define what a cyclic order is.

**Definition 3.1** (Cyclic order). A linear order on a set of taxa X is a bijection \( f : X \rightarrow \{1, 2, \ldots, |X|\} \) (an ordered list containing the taxa in X). We denote a linear order as a list, for example \([acdbe]\) is the linear order with \( f(a) = 1, f(b) = 4 \) and so on.

Now, a cyclic order is an equivalence class of linear orders modulo circular shifts. This means that for example \([abdce], [bdcea] \text{ and } [dceab] \) represent the same cyclic ordering.

An alternative way to define cyclic orders is as an injection \( f : X \rightarrow S^1 \). Thus, with this definition, a cyclic order assigns every taxon a position on the unit circle, and no two taxa are assigned the same position. In the present paper, we use the above definition for defining cyclic orders, since this will be easier in the description of the algorithm.

We will now show that cyclic orders can be encoded as binary vectors.

**Definition 3.3** (Triple). For a set of taxa X we define a triple as an ordered 3-tuple of distinct taxa. We denote such a triple, where \( a, b, c \in X \), by \([abc]\). The set of all triples of X is denoted by \([X]\), so

\[ [X] := \{[abc] | a, b, c \in X \text{ and } a, b, c \text{ distinct} \} \]

We now encode a cyclic order \( f \) by a vector over GF(2). GF(2) is the field with two elements, called “0” and “1”. The following tables show the addition and multiplication for two elements of GF(2):

\[
\begin{array}{c|c|c}
+ & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\quad \text{and} \quad
\begin{array}{c|c|c}
\cdot & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

We have the following definition for the encoding of a cyclic order.

network) in polynomial time, as shown by Keijser and Pendavingh [3]. Section 3.3 details these requirements.
We are now going to study restrictions of vectors to four elements. The following lemma shows why this is useful.

**Lemma 3.** Let \( f \in GF(2)^{|X|} \) be a set, and let \( x \in GF(2)^{|X|} \). The following properties are equivalent:

1. \( x \) is cyclic;
2. for all distinct \( a, b, c, d \in X \):
   a. \( x[abc] + x[bc] = 0 \) (rotation);
   b. \( x[abc] + x[acb] = 1 \) (anti-symmetry);
   c. \( (x[abc] = 1) \land (x[acd] = 1) \Rightarrow x[abd] = 1 \) (transitivity).

**Proof.**
- **1. \Rightarrow 2.** Assume \( x \) is cyclic, so there is an \( f \) such that \( x = x^f \). If \( x[abc] = 1 \), then \( f \) places \( a, b, c \) in this order on the unit circle, thus, also \( x[bc] = 1 \). If however \( x[abc] = 0 \), then also \( x[bc] = 0 \) by the same reasoning, from which property 2a follows. Property 2b can be proven similarly. For property 2c, consider that, if \( a, b, c \) are placed on this order, and \( a, c, d \) are also in this order, \( a, b, d \) will be in the same order.

- **2. \Rightarrow 1.** We need to show that we can construct a cyclic order \( f \) such that \( x = x^f \). We prove this by induction on \( |X| \). For \( |X| = 2 \) it is trivial that any \( f \) will do. Take an \( x \) on \( |X| \geq 3 \) taxa. Pick one taxon \( z \) arbitrarily. Now by the induction hypothesis, we can construct a cyclic order \( g \) such that \( x|_{X \setminus z} = x^g \). (Here, \( x|_{X \setminus z} \) is the restriction of \( x \) to \( |X \setminus z| \).)

From \( g \), we can construct an \( f \) such that \( x = x^f \): insert \( z \) on the circle between some \( a \) and \( a^+ \) if \( x[aza^+] = 1 \). There indeed is always an \( a \) with \( x[aza^+] = 1 \): assume not, so \( x[aza^+] = 0 \) for every \( a \). Pick an arbitrary \( a \neq z \in X \). By anti-symmetry, \( x[zaa^+] = 1 \), and also \( x[zaa^+a^+] = 1 \). By transitivity, we get \( x[zaa^{++}] = 1 \). We can continue this reasoning to get \( x[zaa^{3+}] = 1, \ldots, x[zaa^-] = 1 \). By rotation we get \( x[aza^-] = 1 \). But we assumed that \( x[aza^-] = 0 \), which gives the required contradiction.

### 3.2 The subspace \( U^X \)

We are now going to study restrictions of vectors to four elements. The following lemma shows why this is useful.

**Lemma 3.6.** Let \( x \in GF(2)^{|X|} \). Then \( x \) is cyclic if and only if \( x|_{[(a, b, c, d)]} \) (the restriction of \( x \) to \( [(a, b, c, d)] \)) is cyclic for all distinct \( a, b, c, d \in X \).
Cyclic orders

In Table 1, we enumerate all vectors in $\text{GF}(2)^{\{a,b,c,d\}}$ such that $x[abc] + x[abd] + x[acd] + x[bcd] = 0$. Note that all cyclic orders are in the table; thus, they all satisfy $x[abc] + x[abd] + x[acd] + x[bcd] = 0$. This proves the following lemma.

**Lemma 3.7 (Transitivity).** Let $X$ be a set, and let $x \in \text{GF}(2)^{|X|}$ be a cyclic vector. Then for all distinct $a, b, c, d \in X$:

$$x[abc] + x[abd] + x[acd] + x[bcd] = 0.$$  

Using Lemma 3.5 and Lemma 3.7, we conclude that for a cyclic $x \in \text{GF}(2)^{|X|}$ the following equations hold:

1. $x[abc] + x[aba] = 0$;
2. $x[abc] + x[acb] = 1$;

The set of all $x \in \text{GF}(2)^{|X|}$ satisfying these equations for all distinct $a, b, c, d \in X$ is denoted by $U^X$. A subset $W$ of a vector space $V$ is called an **affine subspace** if $W = \{x \in V \mid Ax = b\}$, for some fixed matrix $A$ and vector $b$. Thus, we see that $U^X$ is an affine subspace of $\text{GF}(2)^{|X|}$.

The following lemma shows that, since will be only interested in cyclic vectors, we will be able to restrict ourselves to $U^X$ in the algorithm. We can formalize this in the following lemma.

**Lemma 3.8.** $U^X$ contains all cyclic vectors in $\text{GF}(2)^{|X|}$.

Note that not all vectors in $U^X$ are necessarily cyclic. In the following section, we will show exactly which vectors are non-cyclic, and how we are able to remove them while keeping the subspace affine.

### 3.3 Requirements on the quartet set

Until now, we did not take into account the set of quartets, given as input to the algorithm. In this section, we are going to remove orders that are not possible according to quartets in this input set, and it turns out that this also (under certain assumptions) removes the non-cyclic vectors.

<table>
<thead>
<tr>
<th>cyclic order</th>
<th>$x[abc]$</th>
<th>$x[abd]$</th>
<th>$x[acd]$</th>
<th>$x[bcd]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[abcd]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[abdc]</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[acbd]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[acdb]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[adbc]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[adcb]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>not cyclic</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>not cyclic</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
To see this, we will take a closer look at the vectors in $U^{[a,b,c,d]}$. Take a look again at Table 1. There are two vectors in $U^{[a,b,c,d]}$ that are actually not cyclic; unfortunately there is no linear equation to remove exactly them. That means that it is not possible to remove exactly the two cyclic vectors, while keeping the resulting subspace affine. However, note that both non-cyclic vectors satisfy $x[abc] \neq x[abd]$. Adding the equation $x[abc] = x[abd]$ would (alongside the two non-cyclic vectors) also remove the vectors corresponding to $[acb|d]$ and $[adb|c]$ from the subspace. These two vectors are exactly those that are not possible according to a quartet $ab|cd$.

Considering this, we define the affine subspace $U^X(ab|cd)$ as the subspace of $U^X$ that additionally satisfies $x[abc] = x[abd]$. For a set of quartets $\Omega$, we furthermore define $U^X(\Omega)$ as follows:

$$U^X(\Omega) = U^X \cap \bigcap_{ab|cd \in \Omega} U^X(ab|cd).$$

It is clear that $U^X(\Omega)$ also is an affine subspace.

Of course, this ‘trick’ of removing the non-cyclic vectors from $U^X$ does not necessarily work if $\Omega$ fails to contain at least one quartet on every quadruple from $X$. A quartet set that does contain a quartet on every quadruple is called dense according to the following definition.

**Definition 3.9** (Dense quartet set). We call a set of quartets $\Omega$ dense if $\Omega$ contains at least one quartet on the taxa of every $\{a,b,c,d\} \subseteq X$.

Now, we can state this more formally in the following theorem.

**Theorem 3.10.** If $\Omega$ is a dense quartet set, $U^X(\Omega)$ is cyclic.

*Proof.* Take distinct $a, b, c, d \in X$ arbitrarily, and take some $x \in U^{[\{a,b,c,d\}]}(\Omega)$. Because also $x \in U^{[\{a,b,c,d\}]}$, $x$ satisfies the three equations (1) – (3).

Assuming $\Omega$ is dense, it contains at least one of $ab|cd$, $ac|bd$ and $ad|bc$. Without loss of generality, assume that it contains $ab|cd$; if is was another, just swap the taxa. This means $x$ satisfies $x[abc] = x[abd]$. We can see in Table 1 that $x$ is cyclic since it satisfies this equation.

Now, by Lemma 3.6, every $x \in U^X(\Omega)$ is cyclic, which proves the theorem. \[\square\]

It turns out that we are able to weaken the requirement: $U^X(\Omega)$ is also cyclic if $\Omega$ contains, on every five taxa, at least four quartets.

**Definition 3.11** (4-out-of-5 quartet set). We call a set of quartets $\Omega$ 4-out-of-5 if $\Omega$ contains at least four quartets on the taxa of every 5-subset of $X$.

The following theorem is a consequence of Lemma 6 from the article by Keijsper and Pendavingh [3].

**Theorem 3.12.** If $\Omega$ is a 4-out-of-5 quartet set, $U^X(\Omega)$ is cyclic.

Note that this theorem is a generalization of Lemma 3.10, since this lemma could only guarantee that $U^X(\Omega)$ is cyclic when $\Omega$ contained five quartets of every 5-subset. This theorem lowers this bound to four quartets.

### 4 Sets of splits

Using the results of the previous sections, we are now able to express the space of all possible cyclic orders of a level-1-network as an affine subspace $U^X(\Omega)$. In the present section, we will define the concept of a split and study properties of the set of all splits of a level-1-network.
4. Sets of splits

Figure 3: \( S = \{c, d\} \) and \( X \setminus S = \{a, b, e\} \) are two examples of splits in this network. The corresponding cut-edge is indicated by a heavier line.

4.1 Splits

We first define what a split is.

**Definition 4.1 (Split).** Let \( N \) be a network. A set \( S \subseteq X \) is called a **split** for \( N \) if

- \( S \) and \( X \setminus S \) are both non-empty;
- there is a subset \( W \) of \( N \)'s vertices such that \( S = X \cap W \) (\( W \) contains all taxa from \( S \));
- \( d_N(W) \leq 1 \) (\( W \) is connected with at most one edge to the remainder of the network).

Alternatively, if \( S \) is a split for \( N \), we also say that \( N \) has split \( S \). A split \( S \) is called **trivial** if \( |S| \leq 1 \) or \( |X \setminus S| \leq 1 \).

See Figure 3 for an example of a split. It is clear that if \( S \) is a split for \( N \), \( X \setminus S \) is also a split for \( N \). Note that a set with one element \( x \in X \) is always a split, since a vertex that is labeled by a taxon has degree 1 by the definition of a network.

We already briefly came across the concept of a cut-edge in Section 2. We repeat the definition here for reference: a **cut-edge** of \( N \) is an edge \( e \) such that \( N \) is not connected anymore if \( e \) is removed. So, when removing a cut-edge, \( N \) is divided in two connected components \( W \) and \( X \setminus W \). This is equivalent to saying that \( d_N(W) = 1 \). Thus, the set \( S = X \cap W \) is a split for \( N \).

On the other hand, if some \( S \) is a split for \( N \), we can find a cut-edge between \( W \) and \( X \setminus W \) by definition. We thus notice that splits and cut-edges are related: every pair of complementary splits corresponds to a cut-edge and every cut-edge corresponds to a pair of complementary splits. In Figure 3, the corresponding cut-edge is also indicated.

Using the following lemma, we are able to determine whether a given subset of taxa is a split.

**Lemma 4.2.** Let \( N \) be a level-1-network, \( t \) a cyclic order consistent with \( N \), and \( S \subseteq X \) such that \( |S| \geq 1 \) and \( |X \setminus S| \geq 1 \). The following statements are equivalent:

1. \( S \) is a split for \( N \);
2. the taxa from \( S \) are consecutive on \( t \), and \( N \) is also consistent with \( rev_S(t) \).
Here, \( \text{rev}_S(f) \) is a cyclic order obtained by reversing on \( f \) the taxa from \( S \). (For example, \( \text{rev}_{\{b,c,d\}}([\text{ecbda}]) = [\text{edbca}] \).)

**Proof.** The correctness is easy to see if \( S \) is a trivial split, so assume that \( S \) is not trivial.

- \( 1. \Rightarrow 2. \) Let \( S \) be a split for \( N \). This split, as we saw above, corresponds to a cut-edge. Thus, it is clear that the taxa on \( S \) are consecutive on \( f \). Furthermore, \( N \) is consistent with \( f \), so there is a planar embedding of \( N \) with this order of leaves around the outer face. We can reverse the order of the taxa in \( S \) in this drawing by flipping one side of the cut-edge. Thus, \( N \) is also consistent with \( \text{rev}_S(f) \).

- \( 2. \Rightarrow 1. \) If \( N \) is also consistent with both \( f \) and \( \text{rev}_S(f) \), we know that there are planar embeddings of \( N \) with the order of leaves around the outer face given by \( f \) and \( \text{rev}_S(f) \). Assume that two vertex-disjoint paths exist connecting \( S \) and \( X \) \( \setminus S \). These paths would cross in one of these planar embeddings, which is impossible (because of the planarity). So, it is not possible for two vertex-disjoint paths to exist connecting \( S \) and \( X \) \( \setminus S \).

Now, we are going to use that all internal vertices of \( N \) have degree 3. This implies that any two edge-disjoint paths in \( N \) are also vertex-disjoint. Thus, there are also no two edge-disjoint paths connecting \( S \) and \( X \) \( \setminus S \). Since Menger’s theorem states that the maximum number of edge-disjoint paths is equal to the minimal number of edges to remove to disconnect \( S \) and \( X \) \( \setminus S \), we can conclude that indeed only one edge from \( N \) needs to be removed. This is by definition a cut-edge, hence \( S \) is a split for \( N \).

So if we construct one cyclic order \( f \), we know that all splits have to be consecutive on this order. This will (in Section 5) give an efficient way to find splits.

### 4.2 Cross-free and laminar sets

We will now discuss sets of splits, specifically, we are interested in the set of splits of a level-1-network. Such a set is always cross-free according to the following definition.

**Definition 4.3** (Cross-free and laminar sets). Consider a set \( S \) of splits, and let \( X \) be the set of taxa.

- \( S \) is said to be **cross-free** if for all \( S_1, S_2 \in S \):
  
  \[
  S_1 \subseteq S_2, \quad S_2 \subseteq S_1, \quad S_1 \cap S_2 = \emptyset \quad \text{or} \quad S_1 \cup S_2 = X.
  \]

- \( S \) is said to be **laminar** if for all \( S_1, S_2 \in S \):
  
  \[
  S_1 \subseteq S_2, \quad S_2 \subseteq S_1 \quad \text{or} \quad S_1 \cap S_2 = \emptyset.
  \]

Note that if \( S \) is laminar, \( S \) is also cross-free. Moreover, if we pick a fixed taxon \( x \in X \) and take all sets in a cross-free set \( S \) that do not contain \( x \), the resulting set

\[
S' := \{ S \in S \mid x \notin S \}
\]

is laminar. See section 13.4 of Schrijver [4] for more details about laminar and cross-free sets.
Figure 4: Depiction of a laminar set (left). In this figure, all sets in the laminar set are indicated by an oval. Its rooted tree-representation is constructed by having vertices representing every element of the laminar set, and adding an arc between two vertices if the set represented by one is a direct subset of the other (right).

Figure 5: The level-1-network generated in Lemma 4.4 from the rooted tree-representation shown in the right of Figure 4.

The result important to us, is that there is an easy bijection between laminar sets and rooted (directed) trees as follows. If $S$ is a laminar set, construct its rooted tree by adding a vertex for every $S \in S$, and adding an arc $(S_1, S_2)$ if

$$S_2 \subseteq S_1 \quad \text{and} \quad \exists S_3 \in S : S_2 \subseteq S_3 \subseteq S_1,$$

that is, if $S_1$ is a direct subset of $S_2$. We will call such a tree the rooted tree-representation of $S$. In Figure 4 a laminar set is represented graphically, and its rooted tree-representation is also shown.

We are now going to study the set of splits in a level-1-network $N$. Such a set is always cross-free. The following lemma connects the rooted tree-representation of a set of splits to the level-1-network having these splits.

**Lemma 4.4.** Let $S$ be a cross-free set. Pick an arbitrary $x$ and let $S' := \{S \in S \mid x \notin S\}$, so $S'$ is laminar. Now construct its rooted tree-representation as outlined above, remove the directions of the arcs (that is, make the graph undirected) and finally replace all vertices with degree $> 3$ by a circuit (the order does not matter); call the resulting network $N$. (See Figure 5 for an example of this process.) Then $S$ is the set of splits for $N$.

**Proof.** Observe that the arcs in the rooted tree-representation exactly correspond to the cut-edges of $N$. Furthermore, the cut-edges of $N$ correspond to the splits for $N$. So, for every arc $(S_1, S_2)$ in the rooted tree-representation, $N$ has a split $S_2$ and a split $X \setminus S_2$. Thus, the $S \in S$ are exactly the splits for $N$. 

\[\]
5 Algorithm

In this section we describe how the actual algorithm works, based on the theoretical results from the previous sections. Given as input a set of quartets \( \Omega \), the algorithm takes the following steps:

- Determine a matrix \((A|b)\) such that \(U^X(\Omega) = (x|Ax = b)\) (Section 5.1).
- Find one cyclic order (Section 5.2).
- Using this order and the matrix, construct a level-1-network (Section 5.3).

In Section 5.4 we describe how to extend the algorithm to detect whether \(U^X(\Omega)\) contains non-cyclic vectors. Finally, in Section 5.5, we review and analyse the algorithm.

5.1 Determining \(U^X(\Omega)\) as an affine subspace

In this first step of the algorithm, we get a set of quartets \( \Omega \) as input. We want to produce a matrix \((A|b)\) such that \(U^X(\Omega) = \{x|Ax = b\}\).

First, we will show that, because \(U^X(\Omega) \subseteq U^X\) for any quartet set \( \Omega \), we can store the vectors more efficiently. Note that \(|X| = |X| \cdot |X| \cdot |X| - 2\sim |X|^3\), so to store the entire vector, we would need \(\Theta(|X|^3)\) storage space. Furthermore the algorithms used further on (such as reducing the matrix) have to operate on all elements of the vector, so they would have running time \(O(|X|^3)\).

A vector \(x \in U^X\) satisfies equation (1) – (3) for all distinct \(a, b, c, d \in X\). We only need to store \(\Theta(|X|^2)\) elements of \(x\) using the following scheme.

First assign an arbitrary total order \(<\) to the taxa. In this section, we will assume \(X = \{a, b, c, \ldots\}\), and we use the lexicographic order. (In the actual implementation, numbers 0, 1, 2, … are used to represent the taxa, and their natural order is used.) Now we only have to store \(x\)’s value on triples \([i_1 i_2 i_3]\) with the following properties:

- \(i_1\) is the minimum with respect to \(<\) (with our order, this is \(a\));
- \(i_2 < i_3\).

Algorithm 5.1 will derive in constant time \(x[i_1 i_2 i_3]\) on any triple \([i_1 i_2 i_3]\), using only the stored triples.

Algorithm 5.1. Derive-Triple-Value \((x, [i_1 i_2 i_3])\)

```plaintext
1 sort \(i_1, i_2, i_3\) in-place using bubble-sort
2 if an odd number of swaps was needed then \(c \leftarrow 1\) else \(c \leftarrow 0\)
3 if \(i_1 = a\) then
4 return \(x[i_1 i_2 i_3] + c\)  \(\triangleright addition in GF(2)\)
5 else
6 return \(x[a i_1 i_2] + x[a i_1 i_3] + x[a i_2 i_3] + c\)
```

In this algorithm, we use bubble-sort, sorts the elements of the triple in-place. This means that the output of the sort is again stored in variables \(i_1, i_2, i_3\). Since we only need to sort three variables, the time complexity of this step is simply \(O(1)\). During the bubble-sort, we count the
number of swaps. Then, we use \( c \) to correct for the sorting. It is easy to see that the algorithm is correct if \( x \in U^X \).

The following lemma states that the proposed encoding is able to exactly represent the vectors in \( U^X \).

**Lemma 5.2.** Let \( x \in GF(2)^{|X|} \) an arbitrary vector and \( \psi \) the vector obtained by running Algorithm 5.1 on every triple in \( |X| \). Then \( \psi \in U^X \).

**Proof.** To prove that \( \psi(x) \) satisfies equation (1), i.e. \( \psi(x)[abc] + \psi(x)[bca] = 0 \), we have to show that the algorithm returns the same on input \( [abc] \) and \( [bca] \). Without loss of generality, assume \( a < b < c \). Sorting \( [abc] \) will obviously require no swaps, sorting \( [bca] \) requires two swaps and both yield the same result. Thus, \( c \) will be 0 in both cases and both times, the algorithm returns the same value in line 4.

Proofs for equations (2) and (3) are similar.

We are now going to build \((A|b)\). This matrix represents a list of linear equations; we will now show how to choose them to get \( U^X(Q) \). Lemma 5.2 means that we do not need anymore to put equation (1) – (3) in the matrix, since all \( x \in U^X \) satisfy them; we only need to add rows to reduce \( U^X \) to \( U^X(Q) \). Recall that

\[
U^X(Q) = U^X \cap \bigcap_{ab|cd \in Q} U^X(ab|cd),
\]

so we can add an equation for every \( ab|cd \in Q \) separately; namely, for a quartet \( st|uv \) we need to add

\[
x[stu] = x[stv] \iff x[stu] + x[stv] = 0
\]

to the matrix. However, it is possible that we are not storing the values of \( x \) on triples \( [stu] \) and \( [stv] \). We thus need to convert this equation in terms of values we do store. To do this, we distinguish two cases.

- If \( st|uv \) contains \( a \), assume without loss of generality that \( s = a \). (If one of the other elements is \( a \), then we are able to swap elements in the quartet to get \( a \) in the first place.) So the equality we want to represent in the matrix is

\[
x[atu] + x[atv] = 0.
\]

We only store these values in the matrix if \( t < u \) and \( t < v \). If \( t > u \) however, we can use that \( x[atu] = x[aut] + 1 \), and we do store that value. The same holds for the situation that \( t > v \).

- If \( st|uv \) does not contain \( a \), it is a bit more difficult. We can now use equation (3):

\[
x[stu] = x[ast] + x[asu] + x[atu] \quad \text{and} \quad x[stv] = x[ast] + x[asv] + x[atv].
\]

So we can replace \( x[stu] + x[stv] = 0 \) by the equivalent equation (after removing \( 2 \cdot x[ast] = 0 \))

\[
x[asu] + x[atu] + x[asv] + x[atv] = 0.
\]

Now we use the same trick as for the first case if \( s > u \), \( t > u \), \( s > v \) or \( t > v \).

Algorithm 5.3 uses exactly these observations to return a row representing the equation for a quartet.

**Algorithm 5.3.** Quartet-Row(st|uv)
we assume that the quartet is given in canonical form

if \( s = a \) then

sort \( s, t, u \) in-place using bubble-sort
if an odd number of swaps was needed then \( c_1 \leftarrow 1 \) else \( c_1 \leftarrow 0 \)

sort \( s, t, v \) in-place using bubble-sort
if an odd number of swaps was needed then \( c_2 \leftarrow 1 \) else \( c_2 \leftarrow 0 \)

return a row representing \( x[stu] + x[stv] = c_1 + c_2 \)

else

sort \( a, t, u \) in-place using bubble-sort
if an odd number of swaps was needed then \( c_1 \leftarrow 1 \) else \( c_1 \leftarrow 0 \)

sort \( a, t, v \) in-place using bubble-sort
if an odd number of swaps was needed then \( c_2 \leftarrow 1 \) else \( c_2 \leftarrow 0 \)

return a row representing \( x[asu] + x[asv] + x[atu] + x[atv] = c_1 + c_2 \)

Note that this algorithm requires the input quartet to be in canonical form. A quartet \( st|uv \) is in canonical form if \( s < t, u < v \) and \( s < u \). Since we defined quartets in terms of sets, which do not have orders, this definition only makes sense when we need to represent the quartet as four variables, as done in an implementation. Of course, computing the canonical form of a quartet is trivial and can be done in constant time.

Now we simply call Algorithm 5.3 for every quartet, and fill a matrix with the resulting rows, as demonstrated in Algorithm 5.4. Before returning the matrix, it is first reduced by Gaussian elimination. This does not change the affine space defined by it, but is required by the next steps of the algorithm.

Algorithm 5.4. Construct-Matrix(\( \Omega \))

let \((A | b)\) be a matrix without rows
for every \( st|uv \in \Omega \) do
add the row returned by Quartet-Row(\( st|uv \)) to \((A | b)\)
reduce matrix \((A | b)\)
return \((A | b)\)

5.2 Finding one cyclic order

For the remainder of the algorithm, we want to obtain an arbitrary cyclic order \( f \) with \( x = x^f \) for some \( x \in \mathbb{U}^X(\Omega) \). In the following step, we will use that to find splits efficiently.

We first pick an \( x \in \mathbb{U}^X(\Omega) \), using the reduced matrix \((A|b)\) that represents \( \mathbb{U}^X(\Omega) \). Considering that \((A|b)\) has been reduced by Gaussian elimination, it will be in the following form:

\[
\begin{pmatrix}
I & \ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast & \ast \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots 
\end{pmatrix},
\]

however the columns of \( A \) are permuted. To easily pick a vector \( x \) such that \( Ax = b \), we set the elements belonging to the identity part of \( A \) to the corresponding value from \( b \), and the
remaining elements to 0. So, the structure of the x we pick will be as follows:

\[
\begin{pmatrix}
& \cdots & \cdots & 0 & 0 & \cdots & 0 \\
\end{pmatrix},
\]

where the first part contains vector b. Then, the equation Ax = b reduces to Ib = b, which is true.

Algorithm 5.5 now produces such a vector x by scanning through the matrix.

**Algorithm 5.5.** Pick-Vector((A|b))

1. \(x \leftarrow \text{the vector } (0, 0, \ldots, 0)\) \(\triangleright\) length equal to the number of columns of A
2. \(row \leftarrow 1\)
3. \(\text{for } col = 1 \text{ to the number of columns in } A \text{ do}\)
4. \(\quad \text{if } A(row, col) = 1 \text{ then}\)
5. \(\quad \quad \text{x}(col) = b(row)\)
6. \(\quad \text{row} \leftarrow \text{row} + 1\)
7. \(\text{return } x\)

The algorithm works as follows. It starts in the upper-left corner of the matrix \((row = 1, col = 1)\). Now, when it encounters a 1, \(x\) is being updated with the corresponding value from \(b\), it increments both \(row\) and \(col\); if it encounters a 0, continues searching in the same row by only incrementing \(col\). To illustrate the algorithm, consider the following matrix \((A|b)\):

\[
\begin{pmatrix}
1 & 0 & \ast & 0 & \ast & b_1 \\
0 & 1 & \ast & 0 & \ast & b_2 \\
0 & 0 & 0 & 1 & \ast & b_3 \\
0 & 0 & 0 & 0 & 1 & b_4 \\
0 & 0 & 0 & 0 & 0 & b_5 \\
\end{pmatrix}
\]

The algorithm scans through the matrix as follows:

\[
\begin{pmatrix}
1 & 0 & \ast & 0 & \ast & b_1 \\
0 & 1 & \ast & 0 & \ast & b_2 \\
0 & 0 & 0 & 1 & \ast & b_3 \\
0 & 0 & 0 & 0 & 1 & b_4 \\
0 & 0 & 0 & 0 & 0 & b_5 \\
\end{pmatrix}
\]

This results in the following vector for \(x\):

\[
\begin{pmatrix}
b_1 & b_2 & 0 & b_3 & b_4 \\
\end{pmatrix}^T.
\]

Note that the algorithm fails when \(U^X(\Omega) = \emptyset\). In this case, there are apparently no cyclic orders possible, given the input quartet set \(\Omega\). This for example happens if all three quartets ab|cd, ac|bd and ad|bc are entered. This case should thus be handled by aborting the algorithm and outputting that no level-1-network is possible consistent with the given quartets.

We are now going to reconstruct \(f\) from \(x\). Note that \(x\) may not be cyclic, for the reasons discussed above. However, for now we assume that \(U^X(\Omega)\) only contains cyclic vectors, thus \(x\) must be cyclic. We will handle non-cyclic vectors in Section 5.4.

We are going to use the strategy given in the proof of Lemma 3.5 (part 2 \(\Rightarrow\) 1). This yields Algorithm 5.6.

**Algorithm 5.6.** Vector-To-Cyclic-Order(x)
Algorithm 5.7. Split-List((A|b), f)

1. let $S$ be an empty list
   $\triangleright$ for an arbitrary, but fixed taxon $a$
2. for each set $S$ that is adjacent on $f$ and does not contain $a$ do
3.     if $S$ is a split then add it to $S$
4. return $S$

Note that in an implementation, we can represent $f$ as a list that indicates the order of the taxa on the circle. If we pick $a$ to be the last taxon of this list, we can represent subsets of taxa, as constructed in line 2 of the algorithm, by a begin and end index in the list. Indeed, splits have to be consecutive on $f$ as proven in lemma 4.2. Using this fact, we can simply enumerate all sets that are adjacent on $f$ using two for loops for the begin and end index. For example, if we pick $a$ to be the last taxon (so $f[N] = a$), we can use the following construct to replace line 2–3.

2a. for begin from 0 to $N - 1$ do
2b. for end from begin + 1 to $N - 1$ do
2c. $S \leftarrow \{f[\text{begin}], f[\text{begin} + 1], \ldots, f[\text{end} - 1]\}$
3. if $S$ is a split then add it to $S$

...
Constructing the network from the splits  Having constructed a laminar set of splits, we continue by constructing the network from it. This is actually not very difficult anymore. According to Lemma 4.4, we can use the rooted tree-representation of the laminar set to get a level-1-network that has the same splits as the original network we wanted to reconstruct.

So, the only missing part is an algorithm to get the rooted tree-representation of a laminar set. Of course it is possible to use the definition of the rooted tree-representation directly, but that would require a lot of searching. That is, for every pair of sets in S we would look whether one is a direct subset of the other, which would mean searching through all sets every time.

A quicker approach is the following. We first remove sets of zero or one element (in our application of the algorithm, these are the trivial splits according to Definition 4.1). This is done because they do not carry any information. We could also check this later if we would have lefted them in now.

We start with a star tree on all taxa, as shown in Figure 6 on the left. Now we add the sets from the laminar set one by one, while modifying arcs, to iteratively build the rooted tree-representation.

This step of adding a set S is done as follows. We first locate the taxa from S in the tree. Now we find the lowest common ancestor of these taxa. (In a tree, the lowest common ancestor of a set of vertices V is a node a such that all v ∈ V are reachable from a, and there is no a’ reachable from a such that all v ∈ V are reachable from a’.) We disconnect the taxa from the lowest common ancestor, and attach them all to a new arc that is connected, in turn, to the lowest common ancestor.

This is summarized in Algorithm 5.8. For brevity, in this algorithm and its discussion, we will identify labelled vertices with their taxa. Furthermore, reach(v) denotes the set of taxa reachable from a vertex v; for a set of vertices V we define reach(V) = \bigcup_{v \in V} \text{reach}(v).

Algorithm 5.8. Rooted-Tree-Representation(S)

1. let T be a star tree containing vertices for all taxa
2. for each set S in S do
3.   find the lowest common ancestor a for the taxa in S; let A be a’s children
4.   find the unique V ⊆ A such that reach(V) = S
5.   create a new vertex v’ with children V
6.   remove the arcs from a to the vertices in V; add an arc from a to v’
7. return T

We will now prove that Rooted-Tree-Representation gives a correct result.

Lemma 5.9. Consider an execution of algorithm Rooted-Tree-Representation on an arbitrary laminar set S.

a) Let v be a vertex created in line 5–6. reach(v) is the same after every subsequent execution of line 6.

b) (Line 4 is possible) After each execution of line 3 there is exactly one V ⊆ A such that reach(V) = S.

c) (Line 5 is possible) For such V, |V| ≥ 2.

d) (Loop invariant) After every iteration of the for loop, N is the rooted tree-representation of the laminar set consisting of the trivial splits and the splits already picked to be S in line 2.
Figure 6: Example of two steps of the Rooted-Tree-Representation algorithm. The algorithm starts with a star graph (left). Then, S ← {a, b} is picked from S, and these taxa are split off from the original tree (middle). Finally, S ← {a, b, c} is picked, and these three taxa are split off (right).

Figure 7: Sketch of the situation in the proof of Lemma 5.9 b).

e) Let T be the output of Rooted-Tree-Representation on a laminar set S. Then T is the rooted tree-representation for the laminar set consisting of the trivial splits and the splits in S.

Proof.

a) In the algorithm, reach(v) only changes in line 6 of every iteration. However, all vertices reachable from v via a will now still be reachable via a and v', and reach(v) does not change otherwise. So, it stays the same during the algorithm.

b) There exists at least one. We can use a simple algorithm to find one: start with V = ∅ and until reach(V) = S, pick a not yet reachable s ∈ S and insert a v ∈ A from which s is reachable.

For this to work, we still have to prove that in every v1 ∈ A that is added, only s ∈ S are reachable. Note that at least one s1 ∈ S is reachable from v1, else we would not have added it. Assume that also a s′1 /∈ S would be reachable from v1. Since a is the lowest common ancestor for the taxa in S, there has to be another v2 ∈ A such that at least one s2 ∈ S is reachable. So S contains s1 and s2, and a is the lowest common ancestor for {s1, s2}. See Figure 7 for a sketch of this situation.

Let us now consider v1. This vertex must originate from line 5–6 in the algorithm, since vertices are not generated in another place. At the time of its creation, a set in S must have existed, call it S', so that after line 6 reach(v1) = S'. From a) we deduce that this is still the case now. Thus, s1 and s′1 are both in S'. This is a contradiction with S being
laminar, indeed, $S$ and $S'$ are not subsets of each other, and still their intersection is not empty (it contains $s_1$).

There exists at most one. Assume that, in any execution of line 4 of Rooted-Tree-Representation, there exist two of such subsets. Since $T$ does not contain cycles (this is always the case during the execution, since never any arc is added that introduces a cycle) and every taxon only occurs once, from at least one of the $v \in A$ there are no taxa reachable. Because of a), such a vertex would have been created with no taxa reachable, so $\emptyset \in S$, which is impossible since we removed all sets with zero or one element from $S$. This gives a contradiction.

c) Assume that $V$ contains exactly one element. Note that $|S| \geq 2$, since we removed all sets with zero or one element beforehand. This means that $a$ has one child $a'$ that contains exactly all taxa in $S$, which contradicts the fact that $a$ is the lowest common ancestor of the taxa in $S$. So $|V| \geq 2$.

d) The loop invariant is initialized correctly, since the rooted tree-representation of all trivial splits is the star tree constructed in line 1 of the algorithm.

We thus need to prove that the loop invariant is maintained in every loop iteration. Assume that the loop invariant was true at the end of the previous loop iteration. Furthermore, let $S'$ be the set of splits already picked, and assume that in the current loop iteration $S \in S$ is picked to add. So we have at the beginning of the iteration an arc $(S_1, S_2)$ in $T$ if

$$S_2 \subseteq S_1 \text{ and } \neg \exists S_3 \in S : S_2 \subseteq S_3 \subseteq S_1.$$ 

Now, in line 3, we find the lowest common ancestor $a$ of all taxa in $S$; note that $S$ is a direct subset of $\text{reach}(a)$. Consider now the $V$ found in line 4. Every $\text{reach}(v)$ is no longer a direct subset of $a$ because of the addition of $S$. Instead, they are a direct subset of $S$. In line 6, we thus correctly remove the arcs from $a$ to $V$, and add arcs from $v'$ to $V$.

e) This follows immediately from the loop invariant in d).

Now, to construct a network from the space $U^X(\Omega)$ that we already determined, we chain the algorithms Split-List and Rooted-Tree-Representation. See Algorithm 5.10.

**Algorithm 5.10.** **Construct-Network**((A|b), f)

1. $S \leftarrow \text{Split-List}((A|b), f)$
2. remove all trivial splits from $S$
3. $N \leftarrow \text{Rooted-Tree-Representation}(S)$
4. make $N$ undirected
5. replace all vertices from $N$ with degree $> 3$ by a circuit $\triangleright$ in the order given by $f$
6. return $N$

### 5.4 Detecting non-cyclic vectors

The only requirement needed for the algorithms in the previous sections to work is that $U^X(\Omega)$ only contains cyclic vectors. In this case, the output of the algorithm is always correct. Of course, when $U^X(\Omega)$ contains non-cyclic vectors, the algorithm will not work.
It is desirable to be able to detect when this situation happens. A simple approach would be to use Theorem 3.12 and reject $\emptyset$ if it does not contain at least four quartets on every 5-subset. If this is the case, the algorithm always gives a correct output. However, the condition is a bit strict, since the theorem gives only a necessary condition for $U^X(\Omega)$ to be cyclic. $U^X(\Omega)$ can also be cyclic if the condition does not hold, and the algorithm will work in such a case.

A less strict approach is allowing any input, but aborting when $U^X(\Omega)$ contains non-cyclic vectors. Thus, we need to extend the algorithms given earlier to take into account the possibility that non-cyclic vectors occur. However, it is not easy to be sure that $U^X(\Omega)$ is indeed cyclic. Keijzer and Pendavingh [3] give a way to do this; in the present report (and the implementation) we do not discuss this method. In this section we will thus consider how well the current algorithm handles the situation.

Firstly, Pick-Vector may pick a non-cyclic vector $x \in U^X(\Omega)$. Then, Vector-To-Cyclic-Order is going to fail. This can happen in two ways.

- In line 3 no $i$ exists such that $x[f[i] \cdot f[i + 1]] = 1$. In this case, $x$ is clearly not cyclic.

- The algorithm succeeds, but returns an incorrect answer. This can happen since the algorithm does not check all elements of $x$. However, we can simply detect this situation by checking (after the algorithm finishes) whether indeed $x = x^f$. Thus, we calculate $x^f$ from $f$, and compare it with $x$.

In both cases, we can conclude that $x \in U^X(\Omega)$. Thus, we can detect problems Pick-Vector. However, if Pick-Vector picked a cyclic vector, that of course does not imply that all vectors in $U^X(\Omega)$ are cyclic. Thus, in the next step of the algorithm (Split-List and Rooted-Tree-Representation), we still need to consider non-cyclic vectors.

Consider the Rooted-Tree-Representation algorithm. If $U^X(\Omega)$ is not cyclic, then the splits found by Split-List may not be a laminar set. In such a case, Rooted-Tree-Representation will not be able to find a rooted tree representation. Instead, it will in one of the loop iterations not be able to find a $V \subseteq A$ such that $\text{reach}(V) = S$ (line 4). Thus, also in this case we can abort the algorithm.

However, since the algorithm does not consider all splits, but only those that are adjacent on $f$, the set of splits returned by Split-List may be laminar. The constructed network will then be probably incorrect.

Summarizing, the algorithm always works on 4-out-of-5 quartet sets. If this is not guaranteed, the algorithm may still work (if $U^X(\Omega)$ is cyclic), or it may detect the problem, or it may return an incorrect network.

5.5 The entire algorithm

The entire algorithm is outlined in Figure 8.

6 Results

The algorithm was implemented in Java. Several tests were performed to see how efficient the algorithm is in practice, and to determine its accuracy on both synthetic and real data sets.

Initially, custom routines in Java were implemented to perform the matrix operations over GF(2). They were quite slow however, and needed a lot of memory. Fortunately, much more
efficient algorithms are implemented in the \textit{M4RI} library for matrix operations over GF(2) \cite{1}. This library makes effective use of, for example, the ability of computers to do XOR operations on entire words efficiently. By using the JNI (Java Native Interface), we extended our implementation (written in Java) to use \textit{M4RI} for all matrix computations (algorithm \textit{Construct-Matrix}).

All performance measurements were done on a laptop with an Intel i5 processor and 4 GB of memory.

6.1 Performance on trees

To get an impression of the performance of the implementation, we implemented a simple algorithm to generate a random tree on \( n \) taxa and to enumerate all quartets from that. We then executed this algorithm for \( n = 20, 22, \ldots, 50 \). (For smaller \( n \), the times became immeasurably small.) We then measured the time to

- construct the matrix \((A|b)\) representing \( U^X(\Omega) \);
- reduce the generated matrix (both using the matrix operations implemented in Java, and using the \textit{M4RI} library);
- find the splits and reconstruct the network from it.

To improve the accuracy, every measurement was done five times; then, the average of the results was taken.

The results are given in Figure 9, Figure 10 and Figure 11, respectively. It is clear that reducing the matrix was by far the slowest part of the algorithm when \textit{M4RI} was not used. However, using \textit{M4RI} causes this part to become about 20 times faster, making reconstruction of a 50-taxon tree possible in a few seconds. Furthermore, much larger trees become feasible using \textit{M4RI}.

6.2 Accuracy when leaving out quartets from trees

In this experiment, we want to show how robust the algorithm is when not all quartets are known. To simulate this situation, we generate random trees and determine the quartets of them. Then, we pick a fraction \( \varphi \) of all quartets and use them as input for the algorithm. We are interested in how often the algorithm still succeeds to reconstruct the tree from the set of quartets.
6 Results

Figure 9: Duration in seconds (on a logarithmic scale) of constructing the \((A|b)\) matrix on \(n\) taxa.

Figure 10: Duration in seconds (on a logarithmic scale) of reducing the matrix on \(n\) taxa. The upper graph shows the custom Java routines for matrix reduction; the lower graph shows M4RI.

Figure 11: Duration in seconds (on a logarithmic scale) of reconstructing the network on \(n\) taxa.
We thus need a measure to determine whether the algorithm is able to reconstruct the tree from a quartet set $Q$. Note that the algorithm is able to do this if $U^X(Q)$ only contains cyclic vectors. We can determine this by using the rank of the produced matrix $(A|b)$ as follows. For a tree on $n$ taxa, $U^X(Q)$ should have dimension $n - 2$. If it has higher dimension, it follows that not all vectors are cyclic. Thus, if the dimension $n - 2$ is reached, the algorithm works.

First, we generated trees on 25 taxa; we then find all $(\binom{25}{4}) = 12650$ quartets. We then set $\phi = 0.01, 0.02, \ldots, 0.25$, and compute the fraction of the time that $U^X(Q)$ has dimension 23. The results are shown in Figure 12. We can conclude that for 25 taxa, we need a fraction of about 9.5% of the quartets for the algorithm to work in 50% of all cases.

We define $Q(n)$ as the number of quartets needed to let the algorithm succeed in half the cases. An interesting question now is how this number depends on the number of taxa. We implemented a simple binary search to find an approximation of $Q(n)$. We will denote this number by $Q(n)$. We executed this for 5, 7, $\ldots$, 35 taxa. In Figure 13 we plotted the fraction of quartets needed, that is, $Q(n)$ divided by the total number of quartets.

It is clear that the fraction of needed quartets gets smaller for higher $n$. However, the total number of quartets compatible with a tree over $n$ taxa is $\binom{n}{4}$, which is $O(n^4)$. So, the $Q(n)$ is still increasing quickly. To better investigate how quickly this function increases, we plotted $Q(n)/n^2$, $Q(n)/n^2 \log n$, $Q(n)/n^3$ and $Q(n)/n^3 \log n$. See figure 14.

From these plots, we are able to conclude that $Q(n)$ is not $O(n^2)$ or $O(n^2 \log n)$, since $Q(n)/n^2$ and $Q(n)/n^2 \log n$ are clearly increasing. From this data, it is not clear whether $Q(n)$ is $O(n^3)$ or $O(n^3 \log n)$.

7 Conclusion

In this paper an implementation, based on the algorithm by Keijsper and Pendavingh, was discussed that finds a level-1-network consistent with a set of quartets. The algorithm always works when the set of quartets is dense, or even 4-out-of-5.

The results show that reconstruction is fast for networks smaller than about 50 taxa. Unfortunately, the running time increases quickly for networks with more taxa.
Figure 13: The fraction of quartets that is needed to let the algorithm work in half the cases, for different amounts of taxa.

Figure 14: Plots of our approximation of $Q(n)$. Upper-left: $Q(n)/n^2$, upper-right: $Q(n)/n^2 \log n$, lower-left: $Q(n)/n^3$ and lower-right: $Q(n)/n^3 \log n$. 
Acknowledgements  I would like to thank Judith Keijsper and Rudi Pendavingh who discovered the algorithm and, as my supervisors, helped me a lot during the project.

References


