DELFT UNIVERSITY OF TECHNOLOGY

BACHELOR THESIS

Orienting phylogenetic networks

Author:
E.A.VERZIJLBERGEN

Supervisors:
dr. ir. L.J.J. van IERSEL
dr. M.E.L. JONES

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Optimization
Delft Institute of Applied Mathematics (DIAM)

June 10, 2019

Declaration of Authorship

"All mathematicians share... a sense of amazement over the infinite depth and mysterious beauty and usefulness of mathematics."

Martin Gardner

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics (DIAM)

Bachelor of Science

Orienting phylogenetic networks

by E.A.VERZIJLBERGEN

This thesis provides you with basic information on graph theory as well as phylogenetic networks, it studies the relationship between undirected (unrooted) and directed (rooted) phylogenetic networks, based on the manuscript 'Rooting for phylogenetic networks' [HvIJ+]. Undirected phylogenetic networks can be oriented to become a directed network. In this manuscript the authors come up with multiple algorithms for orienting phylogenetic networks meeting different characteristics. A network is built up of reticulation vertices (where lineages merge) and tree vertices (where lineages separate). A network can be binary, meaning every node in the body of the network has a degree of three or a network can be non-binary, meaning there are no restrictions to the amount of edges a vertex can have. When a network is binary, an algorithm is described that given the location of the root as well as a set of reticulation points, is used to find an orientation (Algorithm 1). If a network is non-binary, an algorithm is described that given a location of the root as well as the indegree of each vertex, is used to find an orientation (Algorithm 2). Once an orientation is found for a certain undirected network this orientation can be checked to see whether it meets the characteristics of a certain network class. Three network classes are considered. First of all an orientation can be tree-child, meaning that every non-leaf vertex has a child that is not a reticulation. Secondly an orientation can be stack-free, meaning that no reticulation has a child which is a reticulation. And last, an orientation can be valid, meaning that it is stack-free and deleting a single reticulation edge and suppressing its endpoints does not give parallel arcs. When we either did not find an orientation in the class we wanted or want to know all the orientations for a certain network in a certain class, we use Algorithm 3 or 4. The location of the root and the set of reticulation points were necessary input for Algorithm 1; whereas Algorithms 3 and 4 do not require this information. Algorithm 3 returns you the first found orientation in the class you wanted and Algorithm 4 returns you a collection of all orientations in the class. All orientations found by the program are returned in an output format that can be read by a network visualisation website [GGL+].

Acknowledgements

I want to thank all who taught me during my bachelor Applied Mathematics and enlightened me with knowledge. Finishing my bachelor would have been impossible without the aid and support of my family and friends. As most of us know, programming is thinking, trying, and fixing all the error statements to come to a satisfying and well functioning program. This thesis is the result of knowledge gained in the past years.
Special thanking are in order for dr. ir. L.J.J (Leo) van Iersel and dr. M.E.L. (Mark) Jones for guiding me throughout this thesis and dr. ir. M. (Marleen) Keijzer for being part of my Bachelor Committee.

Contents

List of Figures

Chapter 1

Introduction

Phylogenetic networks are graphs used to describe and visualise evolutionary relationships between (for example) genes, chromosomes or species. Networks come in all different layouts, due to the way they are build using different nodes. For most networks we do know which objects are related, often we do not know the direction of the relationship, e.g. which one is the descendent and which is the ancestor. In 2018, 229 new plant and animal species were discovered [Ims]. Due to DNA matching, scientists are able to relate these species to earlier discovered ones. Using these relations it is possible for the new species to be added to a larger network existing of known species on Earth. Since the direction of these new relationships is not known, this network is not entirely directed anymore.
For these networks it might be possible to find an unique orientation. Such an orientation shows the direction of the events, for example time or DNA/gene transfer. Mathematicians, biologists and every specialist in between these fields might benefit from knowing the orientations of a phylogenetic network. Two examples will be provided which occur in nature and connect them to different nodes.

Consider a scenario in which a sloth of polar bears lives on an iceberg. Due to global warming the iceberg splits in two. A part of the sloth of polar bears is separated from the rest and is no longer able to reach them. Now the separated part adapts to the new environment and becomes a new sloth. This situation occurring in nature can be represented as a phylogenetic network, see Figure 1.1.

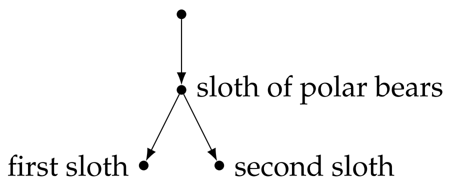

FIGURE 1.1: Division of a sloth of polar bears

Now consider another scenario in which there are adjacent forests in which two packs of wolves live. One day due to a lightning strike there is a forest fire and one of the packs has to migrate to the neighboring forest. The two packs of wolves now live in one forest and the packs merge. This situation occurring in nature can be represented as a phylogenetic network, see Figure 1.2.

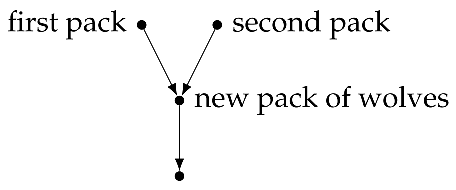

FIGURE 1.2: Uniting two packs of wolves

In the unpublished manuscript: "Rooting for phylogenetic networks" [HvIJ$^+$] by Katharina Huber et al, several fundamental questions regarding the relationship between directed (rooted) and undirected (unrooted) phylogenetic networks are answered. First they ask themselves the following: suppose you know the underlying undirected phylogenetic network of some directed binary phylogenetic network as well as the location of the root and of the reticulation vertices (the vertices where lineages merge.) Does this give you enough information to uniquely reconstruct the directed phylogenetic network? They show that this is indeed the case. In Figure 1.3 you find a complete example of an undirected unrooted binary network and a directed rooted orientation. The squares represent the reticulation vertices, presented in the example of the wolves.

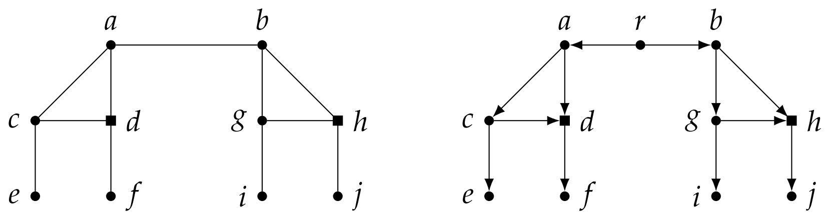

FIGURE 1.3: Example undirected and directed network

Moreover, the authors give a mathematical characterization for when an undirected phylogenetic network, given the locations of the root and reticulation vertices, can be oriented as a directed phylogenetic network. In addition, they give a polynomial-time algorithm to find such an orientation. They also show how these results can be generalized to non-binary networks, if the desired indegrees of the reticulation vertices are specified. Moreover, they show how to apply the algorithm to partly-directed networks, networks in which some of the edges are already oriented. In the second part of the manuscript, they study which undirected binary phylogenetic networks can be oriented to become a directed phylogenetic network of a certain class (with no information about the location of the root or the reticulation vertices). They give an algorithm that is fixed parameter tractable (FPT), this means that the algorithm runs quickly for large input networks as long as the number of reticulation vertices is not too large. The algorithm can be applied to a wide range of classes of networks, including the well-studied classes of Tree-child, Tree-based, Reticulation-visible and Stack-free networks but also to the recently introduced classes of Valid networks [MvIJ$^+$18] and Orchard networks [JM18, ESS19].

Throughout this thesis, definitions, theory and algorithms will be presented to the reader. At first a distinction will be made according to the structure of the graph, a graph will be either binary or non-binary. For each of the two options, an algorithm exists which is able to return an orientation of the network, if one does exist. After this, different network classes will be introduced. The class algorithms will show you how to check if the found orientation belongs to its class. Then another two algorithms will be introduced, these algorithms return either the first found orientation belonging to a desired class or all of the orientations belonging to a desired class, no additional information is needed about the location of the root or the reticulation vertices. In Chapter 8 a manual on how to apply the program on your data set is presented.

The main contribution of this thesis is a practical implementation of the orientation algorithms that were presented in the manuscript [HvIJ+].
The implemented program is applied to multiple example networks [GGL+], the results will be presented in Tables 9.1 and 9.2. For each example network, the program correctly returns the orientation from the website [GGL+] when given the undirected network together with the set of reticulation vertices (binary) or desired indegree (non-binary) and the rooted edge as input. For each orientation it is checked to which network classes it belongs. Furthermore, for each binary example network, the program has been used to find out whether the network has an orientation that belongs to a certain class, if the set of reticulation vertices and rooted edge are assumed to be unknown. This has been done for each of the classes Tree-child, Stack-free and Valid, the running times are presented. At the end you should be able to understand and use the attached python program to orient a phylogenetic network.

Chapter 2

Preliminaries

The following sections will give a short introduction to graph theory (Section 2.1) and specifically phylogenetic networks (Section 2.2). Terminology used in this report will be explained within these sections.

2.1 Graph theory

In this section terminology of graph theory will be explained, these definitions are either given in or based on the definitions of the lecture notes [AvIJ18] of the subject Optimization (Delft University of Technology, TW2020). Graph theory is the study of graphs. *Graphs* are made up of points: *vertices* or *nodes* (V), and lines: *edges* (E). Each edge represents a relation between two nodes, which correspond to different objects.
The official notation of an undirected graph is given by $G = (V, E)$. An edge $e \in E$ is given by the following notation: $e = \{v, w\}$, in this case we say that v and w are *endpoints* of the edge $\{v, w\}$. We say that an edge $e \in E$ is *incident* to each of its endpoints. The *degree* $d(v)$ of a vertex $v \in V$ is the number of edges that are incident to v. A *walk* in a graph $G = (V, E)$ is an ordered list of vertices $(v_0, v_1, ..., v_k)$ such that each connecting edge $\{v_{i-1}, v_i\} \in E$ for all $1 \leq i \leq k$. We call this a *closed walk* in case $v_0 = v_k$. If in this closed walk all v_i are distinct, it is called a *cycle*. In Figure 2.1, we find examples of a walk $\{v_1, v_3, v_4, v_6, v_8, v_7\}$, a closed walk $\{v_1, v_2, v_4, v_6, v_8, v_7, v_4, v_3, v_1\}$ and a cycle $\{v_1, v_2, v_6, v_4, v_7, v_5, v_3, v_1\}$.

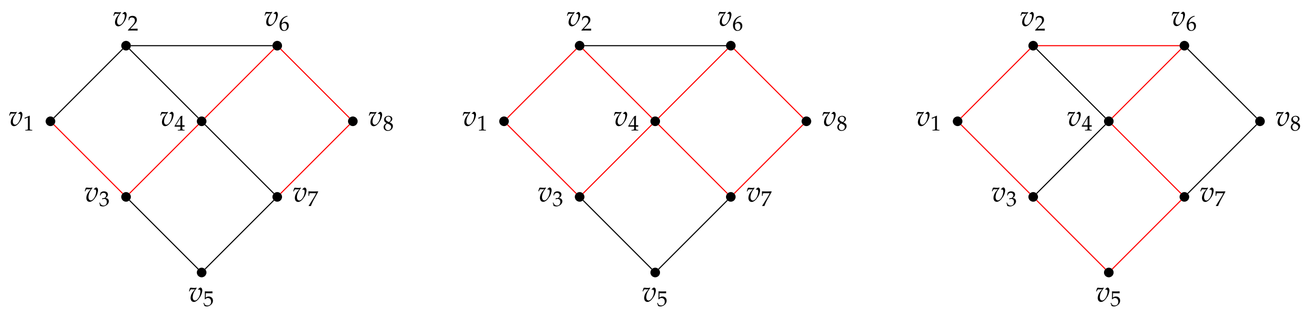

FIGURE 2.1: Example of a walk, a closed walk and a cycle

A *path* is a walk that does not visit any vertex more than once. It is called a *u*-*v* path if u is the first vertex and v is the last vertex on the path. A graph is *connected* if it contains a *u*-*v* path for each pair of vertices $u, v \in V$. A connected graph with no cycles is called a *tree*. If a graph is not connected but does not contain any cycles it is called a *forest*, each of the disconnected components is a tree. See Figure 2.2 for a visualisation of the relation between a tree and a forest.

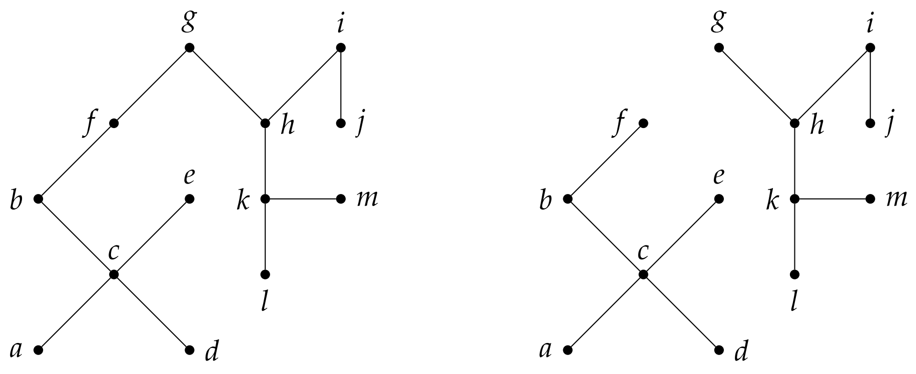

FIGURE 2.2: Example of a tree and a forest

The following lemma holds for trees:

Lemma 2.1.1 *Let $G = (V, E)$ be a graph, then the following are equivalent:*

1. *G is a tree;*
2. *G is connected and $|E| = |V| - 1$;*
3. *G contains no cycles and $|E| = |V| - 1$;*
4. *G contains a unique path between each pair of vertices.*

In case that an edge is directed it is called an *arc*. To relate this to an example occurring in nature: an edge can be drawn when two individuals have DNA in common, an arc can be drawn when we do know which one is the ancestor and which the descendent. The official notation of a directed graph is given by $G = (V, A)$. Taking arc $a \in A$ as $a = \{v, w\}$, we say that v is the *tail* and w is the *head* of the arc. We say that the arc $\{v, w\}$ is *leaving* v and *entering* w. The *indegree* $d^-(v)$ of a vertex v is the number of arcs entering v also called *inedges* of v and the *outdegree* $d^+(v)$ of a vertex v is the number of arcs leaving v also called *outedges* of v. The degree of a vertex might be defined as the sum of the indegree and outdegree.

2.2 Phylogenetic networks

A *phylogenetic network* is any graph used to visualise evolutionary relationships (either abstractly or explicitly) between nucleotide sequences, genes, chromosomes, genomes or species [HRS10]. Phylogenetic networks are mostly directed and rooted (containing a root) but can be undirected and unrooted (not containing a root) as well. The edges or arcs represent evolutionary events such as descent, endosymbiosis, hybridization, or gene transfer.
The rest of this section will explain more about directed rooted phylogenetic networks. A phylogenetic network consists of different sorts of nodes. One of these sorts is called the root, this vertex has no inedges, only outedges. In a rooted network there exists only one root. Besides a root a phylogenetic network also contains leaves, a leaf has no outedges only a single inedge. A network can have multiple leaves.

Besides roots and leaves a directed network exists of two kinds of nodes: tree nodes and reticulation nodes. A tree node has only one inedge and can have multiple outedges. A reticulation node can have multiple inedges as well as multiple outedges. Reticulation- and tree nodes correspond to different kind of evolutionary events. In Chapter 1 two examples are given referring to situations occurring in nature. The scenario of a split of a sloth can be represented as a phylogenetic network. This can be seen in Figure 1.1. This is an example of a tree node, it has only one inedge and multiple, in this case two, outedges. The second scenario, where two packs of wolves merge, can be represented as a phylogenetic network as well. This can be seen in Figure 1.2. This is an example of a reticulation node, it has multiple, in this case two, inedges and can have multiple, in this case one, outedges.

In mathematical terms, a network can be defined as a graph with one root and with leaves labelled with species. The *leaves* have an indegree $d^-(v) = 1$ and outdegree $d^+(v) = 0$. In case the network is directed it has a root which has indegree $d^-(v) = 0$ and outdegree $d^+(v) \geq 1$. In a *binary* network every node of the network that is not a leaf or root has a degree of three. This means that it is either a tree node: v, having indegree $d^-(v) = 1$ and outdegree $d^+(v) = 2$, or a reticulation node: w, having indegree $d^-(w) = 2$ and outdegree $d^+(w) = 1$. In non-binary networks there is no restriction to the number of incident edges a vertex can have. A *non-binary* network just like a binary network exists of a root and leaves as well as two other kind of vertices: A tree node, which has only one inedge and can have multiple outedges, and a reticulation node, which can have multiple inedges as well as multiple outedges.

Suppressing a vertex means deleting it but keeping the connections (paths that travel through it). In Figure 2.3 you find two examples, suppressing the vertex x in a directed network and suppressing the root r in a directed network resulting in an undirected unrooted network.

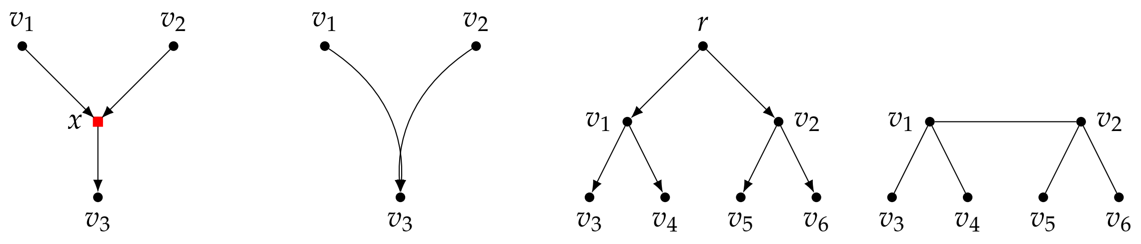

FIGURE 2.3: Examples of suppressing the vertex x in a directed network and suppressing the root r in a directed network resulting in an undirected unrooted network

Given an undirected network N and a directed network N', we say that N is the *underlying network* of N' and N' is an *orientation* of N, if the network derived from N' by replacing all directed arcs with undirected edges and suppressing the former root (if this vertex has degree two) is isomorphic to N. We say that N is *orientable* if it has at least one orientation.

2.3 Network classes

Networks as well as their orientations can be checked for membership of different classes. Some classes overlap and some are sub-classes giving extra characteristics to a network. Examples of overlapping classes are orchard and reticulation-visible, these classes will not be discussed in this thesis. Figure 2.4 gives a representation of how some of the classes of directed networks are related. We will discuss the following classes:

- **Tree-child**: every non-leaf vertex has a child that is not a reticulation.
- **Stack-free**: no reticulation has a child which is a reticulation.

- **Valid**:
  - Stack-free;
  - deleting a single reticulation edge and suppressing its endpoints does not give parallel arcs (definition in Section 2.3.3).

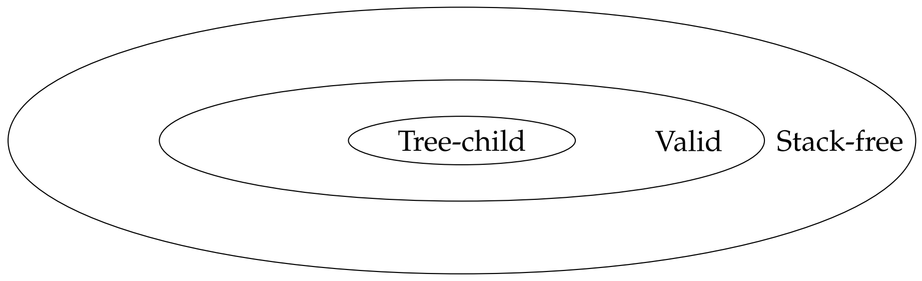

FIGURE 2.4: A visualisation of the relations of the classes.

For each of these classes, the following subsections show how to check algorithmically if a network does belong to the specific class.

2.3.1 Tree-child

To check if a directed network is tree-child we need to check if every non leaf vertex has a child that is not a reticulation. A collection is made containing all the vertices. For each vertex we check if it has children, if a vertex has no children it is a leaf and therefore it is not important for tree-child criterea.
If a vertex is not a leaf, we check if at least one of the children is not a reticulation point. If there exist such a child, we continue to the next vertex.
If all of the vertices fulfil the requirement, the directed network is reported to be tree-child.
If a certain point does not fulfil the requirement the directed network is not tree-child.
The actual program can be found in Appendix A.5.

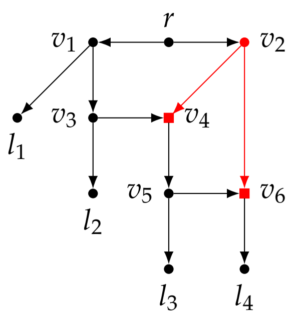

FIGURE 2.5: A binary network that is not tree-child since v_2 has v_4 and v_6, reticulation vertices , as children.

2.3.2 Stack-free

To check if a directed network is stack-free we need to check if there exist no reticulation point with a reticulation as a child. A collection is made containing all the reticulations of a network. For each reticulation we check if none of its children is a reticulation vertex. As soon as one of the children of a reticulation turns out to be a reticulation the directed network is not stack-free.

If all of the reticulation points are checked and non of them has a reticulation as a child we can conclude that the network is stack-free.
The actual program can be found in Appendix A.5.

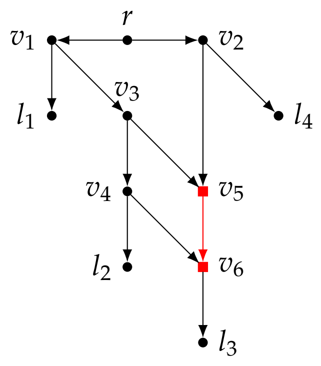

FIGURE 2.6: A binary network that is not stack-free since v_5, a reticulation, has v_6, a reticulation, as a child.

2.3.3 Valid

To check if a directed network is valid we need to check if it is stack-free and if deleting a single reticulation edge and suppressing its endpoints does not give parallel arcs. In case we have $e_1, e_2 \in E$ where $e_1 = \{x, y\}$ and $e_2 = \{x, y\}$ but $e_1 \neq e_2$ we call e_1 and e_2 *parallel arcs*. In binary networks there are two scenarios in which deleting a single reticulation edge and suppressing its endpoints does result in parallel arcs, only one of these is stack-free and can be seen in Figure 2.7. In this figure the reticulation edge (v, r) has to be deleted and the endpoints v and r should be suppressed. When suppressing v this will result in parallel arcs (k, s).

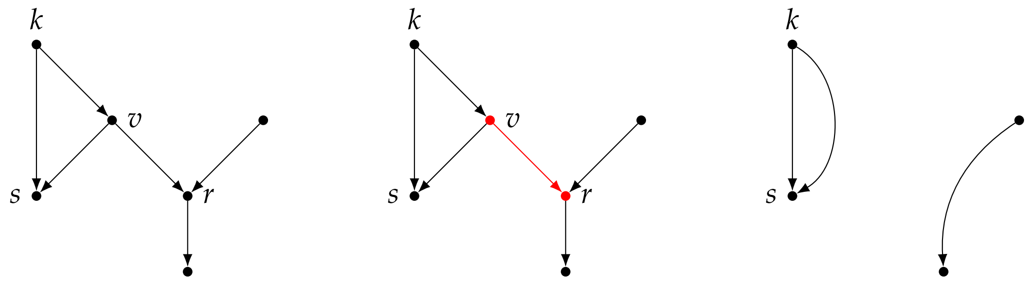

FIGURE 2.7: Part of a binary network in which case deleting a single reticulation edge and suppressing its endpoints does give a parallel arcs.

The algorithm checks if an orientation is valid by iterating over all of the reticulation points. We check if a sibling is a child of one of its grandparents (on that side) as well. When referring to the given example, r is the reticulation point, v is its parent and s is the sibling. s is also a child of k, which is r its grandparent (on the same side as v). The actual program can be found in Appendix A.5.

Chapter 3

Problem set-up

The first distinction that is made is based on the degree of the vertices, a network can be either binary or non-binary. For each of these network types exists a different algorithm. These algorithms need different input as well. We describe the input and output for each of these algorithms below.

Definition 3.0.1 *Suppose N' is a directed rooted network, N is a binary undirected unrooted network, e_ρ an edge in N and R a set of vertices in N, we say that N' is a phylogenetic orientation of (N, e_ρ, R) if:*

- *N' is an orientation of N with an inserted root;*
- *the root of N' is inserted in the edge e_ρ;*
- *the reticulation point of N' equal the set R.*

To orient a binary network we need to know the rooted edge $e_\rho \in E$ (this is the edge where the root will be inserted) and the set of reticulation points, R.

ORIENTATION ALGORITHM FOR BINARY NETWORKS
Input: An undirected network N, an edge $e_\rho \in E$, and a subset R of the vertices.
Output: A phylogenetic orientation of (N, e_ρ, R) if it exists and NO otherwise.

Definition 3.0.2 *Suppose N' is a directed rooted network, N is a non-binary undirected unrooted network, e_ρ an edge in N and d^- is the desired indegree of all vertices in N, we say that N' is a phylogenetic orientation of (N, e_ρ, d^-) if:*

- *N' is an orientation of N with an inserted root;*
- *the root of N' is inserted in the edge e_ρ;*
- *the indegree of each point of N' equals the desired indegree d^-.*

To orient a non-binary network we need to know the rooted edge $e_\rho \in E$ and the indegree $d^-(v)$ for each vertex $v \in V$.

ORIENTATION ALGORITHM FOR NON-BINARY NETWORKS
Input: An undirected non-binary network N, an edge $e_\rho \in E$, and the desired indegree $d^-(v)$ with $1 \leq d^-(v) \leq d_N(v)$ for each $v \in V$.
Output: A phylogenetic orientation of (N, e_ρ, d^-) if it exists and NO otherwise.

For each orientation found by one of these algorithms, we can check to which of the classes: tree-child, stack-free and valid, it belongs.
It is possible to search for an orientation of a network which belongs to a desired class. An

orientation belonging to a class C is called a C-orientation.

CLASS-ORIENTATION ALGORITHM RETURNING ONE ORIENTATION
Input: An undirected unrooted binary network N, number of reticulation points k, a class C.
Output: One phylogenetic C-orientation of (N, e_ρ, R) with the number of reticulation points k, if it exists and NO otherwise.

Last, it is possible to return all the orientations of a network belonging to the desired class.

CLASS-ORIENTATION ALGORITHM RETURNING ALL ORIENTATIONS
Input: An undirected unrooted binary network N, number of reticulation points k, a class C.
Output: All phylogenetic C-orientations of (N, e_ρ, R) with the number of reticulation points k, if it exists and NO otherwise.

Chapter 4

Programming choices

This thesis is mostly focused on implementing the algorithms. To turn the written algorithms into actual programs a language had to be chosen. Since Python is an upcoming language and most faculties of Delft University of Technology are switching toward Python, choosing this language will make the use of this program most worthwhile. Throughout the process example networks [GGL+] are used. Their input is placed in a `.txt` file and the python output is printed in the same format which can be used to make a visualisation using a network visualisation website [GGL+]. Once it was decided to use python many other programming choices had to be made. One of the most important ones is the implementation of a network. To represent a network we have chosen to use a dictionary in python [Fou]. Many functions already apply to this class and therefore only the actual algorithms need to be implemented.

To represent the directed graph in Figure 4.1 we use the following code given in Figure 4.2.

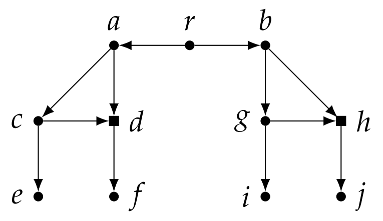

FIGURE 4.1: Example directed graph

```
G = {'r': ['a', 'b'],
     'a': ['c', 'd'],
     'c': ['d', 'e'],
     'e': [],
     'd': ['f'],
     'f': [],
     'b': ['g', 'h']
     'g': ['h', 'i'],
     'i': []
     'h': ['j'],
     'j': []
     }
```

FIGURE 4.2: Python representation of a directed graph

Referring to the code in Figure 4.2: for a line such as $'a'$: $['c', 'd']$ we call a the *key* and c and d the *values*. In an undirected graph, $'a'$: $['c', 'd']$ means that there is an edge between a and c

and between a and d. Hence, a will be a value of the keys c and d. The undirected unrooted graph form of the graph given in Figure 4.2 is given in Figure 4.3 and the corresponding code is given in Figure 4.4. Here we see that b is a value of the key a and a is also a value of the key b.

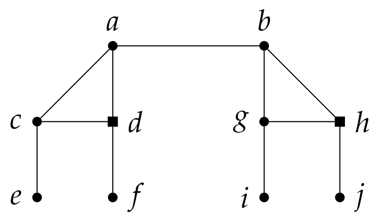

FIGURE 4.3: Example undirected graph

```
G = {'a': ['b', 'c', 'd'],
     'c': ['a', 'd', 'e'],
     'f': ['d'],
     'e': ['c'],
     'd': ['a', 'c', 'f'],
     'b': ['a', 'g', 'h']
     'g': ['b', 'h', 'i'],
     'i': ['g']
     'h': ['b', 'g', 'j'],
     'j': ['h']
     }
```

FIGURE 4.4: Python representation of a undirected graph

Chapter 5

Binary networks

Firstly, simpler type of networks are considered, binary networks. In this chapter will be discussed how to orient a binary network. First the algorithm will be explained and why certain 'if' and 'while' statements are made. Sometimes the algorithm does not return an orientation because not all networks can be oriented. In paragraph 5.2 such an example will be given and explained what the algorithm does in such a case.

5.1 Algorithm 1

In the manuscript [HvIJ+] the authors found an algorithm to generate an orientation of a binary network. The algorithm starts by checking the criterion $|R| = |E| - |V| + 1$. This can be derived using lemma 2.1.1. An oriented network has a lot in common with a tree, except an oriented network does not only consist of tree nodes and therefore the number of reticulation nodes had to be inserted into the formula $|E| = |V| - 1$ resulting in the criterion $|R| = |E| - |V| + 1$. After checking this, the root is inserted and the edges incident are oriented away from the root. Following this, the algorithm continues as long as there are unoriented edges present.

```
Data: An undirected network $N$, an edge $e_\rho \in E$, and a subset $R$ of the vertices
Result: A phylogenetic orientation of $(N, R, e_\rho)$ if it exists and NO otherwise
if $|R| \neq |E| - |V| + 1$ then
    return NO
Subdivide $e_\rho$ by a new vertex $\rho$ and orient the two edges incident to $\rho$ away from $\rho$ ;
while there exists an unoriented edge do
    if there is a vertex $v \in V - R$ with one incoming oriented edge and two incident
      unoriented edges then
        orient the two unoriented edges incident to $v$ away from $v$.
    else if there is a vertex $v \in R$ with two incoming oriented edges and one incident
      unoriented edge then
        orient the unoriented edge incident to $v$ away from $v$
    else
        return NO
end
return the obtained orientation
```

Algorithm 1: Orientation algorithm for binary networks

When implementing Algorithm 1, changes had to be made due to programming choices. Since an unoriented edge is programmed as two oriented edges, one forward and one backward, the total number of edges has to be divided by two to meet the criterion $|R| = |E| - |V| + 1$. Since programming a graph as a dictionary does not give you the opportunity to iterate over edges the while statement is changed. The new idea proposes that the algorithm continues as long as there are too many edges, in other words: there still exist 'double edges', which in this case represent unorientend edges. To know whether "there is a vertex $v \in V - R$ with one incoming oriented edge and two incident unoriented edges" or "there is a vertex $v \in R$ with two incoming oriented edges and one incident unoriented edge" there exist two groups "orientable reticulation nodes" and "orientable tree nodes", each vertex, v, meeting the criteria is added to such a group. The actual implementation can be found in Appendix A.3.

5.2 Reticulation cut

Sometimes no orientation can be found. The criterion $|R| = |E| - |V| + 1$ has already been discussed in the last section. A scenario you can come across is that there are no orientable treenodes left and only reticulation nodes with no inedges or one inedge. The reticulation nodes with one inedge have two unoriented edges but it is not known which is an inedge and which is an outedge. In this case no orientation can be given.

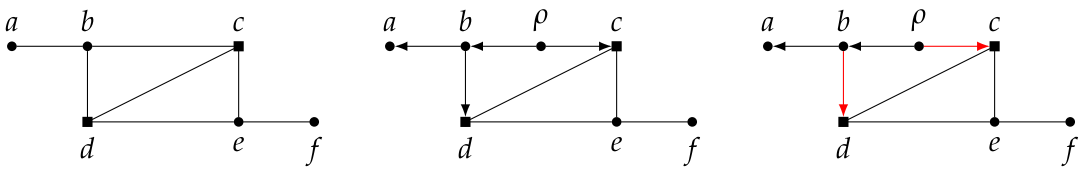

FIGURE 5.1: Orienting a network which results in a reticulation cut

In Figure 5.1 a representation of the algorithm is given in which it results in a reticulation cut. In the figure on the left the unoriented network can be seen. In the middle the root has been inserted and the rooted edge has been oriented as well as the edges from tree node b. After this no edges can be oriented anymore since c and d are both reticulation nodes with only one inedge and two unoriented edges. Therefore the arcs (b, d) and (ρ, c), given in red in the figure on the right, define the reticulation cut. The complete definition of a reticulation cut is written in the following way:

Definition 5.2.1 *Given an undirected binary network N = (V, E, X), a distinguished edge $e_\rho \in E$, and a subset R of the vertices, a reticulation cut is a pair (R', E') with $R' \subseteq R$, |R'| = |E'| and E' is an edge-cut of the network N_ρ obtained by subdividing e_ρ by a new vertex ρ, such that E' has exactly one edge incident to each $r \in R'$ and ρ is not in the same connected component of $N \setminus E'$ as any $r \in R'$.*

The following proposition shows that the existence of a reticulation cut proves that there is no phylogenetic orientation [HvIJ+].

Proposition 5.2.2 *Let N=(V, E, X) be an undirected binary network, $e_\rho \in E$ and $R \subseteq V$. (N, e_ρ, R) is orientable if and only if the following two conditions hold*

1. *(N, e_ρ, R) has no reticulation cut;*
2. *|R| = |E|-|V|+1*

Every time no orientation can be found the algorithm provides you with the set of reticulation edges together forming the reticulation cut.

Chapter 6

Non-binary networks

Binary networks only make up a very small part of all types of graphs. In this chapter will be discussed how to orient a non-binary network. First the algorithm will be explained and why certain 'if' and 'while' statements are made. Sometimes the algorithm does not return a orientation because not every network can be oriented. In paragraph 6.2 such an example will be given and will be explained what the algorithm will do in such a case.

6.1 Algorithm 2

Since not all reticulation nodes have the same number of inedges anymore, the algorithm cannot be provided with a set of all reticulation nodes as done in Algorithm 1. One of the inputs needed to orient a non-binary network is the desired indegree for each $v \in V$. The criterion $|R| = |E| - |V| + 1$ or $|R| + |V| = |E| + 1$ has to be rewritten to a statement involving the indegree. Since R and V are not clearly distinguishable groups, rewriting the statement will result in the new criterion $\sum_{v \in V} d^-(v) = |E| + 1$. After checking this, the root is inserted and the edges incident are oriented away from the root. After that the algorithm continues as long as there are unoriented edges present.

```
Data: An undirected non-binary network $N$, an edge $e_\rho \in E$, and the desired indegree
    $d^-(v)$ with $1 \leq d^-(v) \leq d_N(v)$ for each $v \in V$.
Result: A phylogenetic orientation of $(N, e_\rho, d^-)$ if it exists and NO otherwise
if $\sum_{v \in V} d^-(v) \neq |E| + 1$ then
    return NO
Subdivide $e_\rho$ by a new vertex $\rho$ and orient the two edges incident to $\rho$ away from $\rho$ ;
while there exists an unoriented edge do
    if there is a vertex $v \in V$ with $d^-(v)$ incoming oriented edges and at least one incident
      unoriented edge then
        orient all unoriented edges incident to $v$ away from $v$.
    else
        return NO
end
return the obtained orientation
```

Algorithm 2: Orientation algorithm for non-binary networks

When implementing Algorithm 2, changes had to be made due to programming choices. The same programming choice have been made evolving around the dictionary representation of a graph as discussed in the last chapter. To speed up the algorithm once 'there is a vertex $v \in V$ with $d^-(v)$ incoming oriented edges and at least one incident unoriented edge' this vertex v is added to a group 'orientable' this way we know over which keys to iterate. The actual implementation can be found in Appendix A.4.

6.2 Degree cut

As well as in the binary case sometimes no orientation can be found. This can be due to a degree cut. Since a reticulation point w does not necessary have two inedges and one outedge but can have multiple inedges as well as multiple outedges it is less easy to provide the user with the degree cut. The algorithm stops once there are no orientable tree nodes and only reticulation nodes with less inedges than their indegree. All these inedges, belonging to reticulation nodes with less inedges than their indegree, belong to the degree cut.

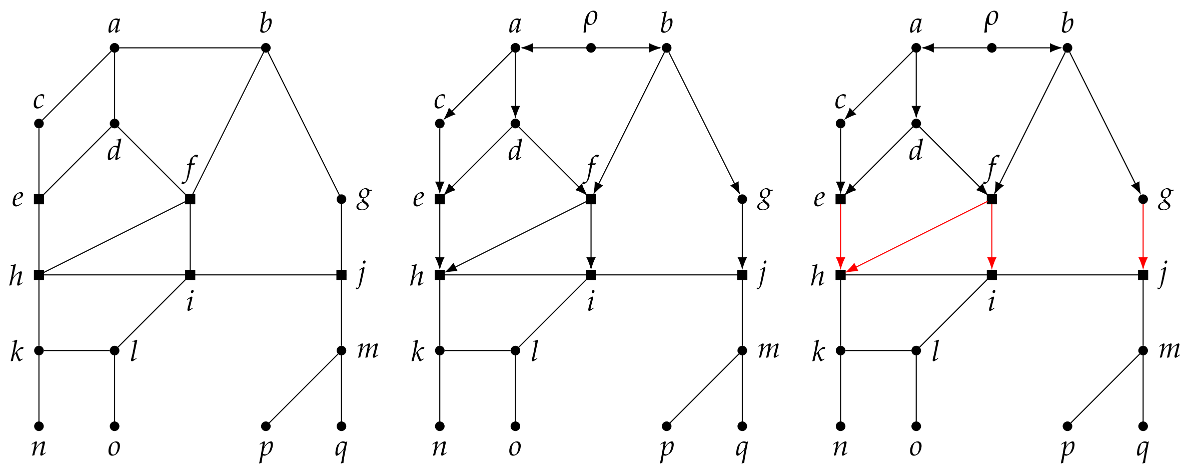

FIGURE 6.1: Orienting a network which results in a degree cut

In Figure 6.1 a representation of the algorithm is given in which it results in a degree cut. In the figure on the left the unoriented network can be seen, in this network all tree nodes have desired indegree one, the reticulation nodes, marked with a square instead of a point, have desired indegree two and only h has desired indegree three. In the middle figure the root has been inserted and the rooted edge has been oriented as well as the edges from orientable tree nodes and reticulation nodes. At one point, no more edges can be oriented since both i and j have desired indegree two and only have one inedge and h has desired indegree three and only has two inedges. Therefore the arcs (e, h), (f, h), (f, i) and (g, j), given in red in the figure on the right, define the degree cut. The complete definition of a degree cut is written in the following way:

Definition 6.2.1 *Given an undirected nonbinary network $N = (V, E, X)$, a distinguished edge $e_\rho \in E$, and the desired in-degree $d^-(v)$ of each vertex $v \in V$, a degree cut is a pair (V', E') with $V' \subseteq V$ such that:*

- *E' is an edge-cut of the network N_ρ obtained by subdividing e_ρ by a new vertex ρ;*
- *each edge in E' is incident to exactly on element of V';*
- *each vertex $v \in V'$ is incident to less than $d^-(v)$ edges in E' and*
- *ρ is not the same connected component of $N \setminus E'$ as any $v \in V'$.*

The following proposition shows that the existence of a degree cut proves that there is no phylogenetic orientation [HvIJ+].

Proposition 6.2.2 *Let N=(V, E, X) be an undirected non-binary network, $e_\rho \in E$ be a distinguished edge and d^- be the desired indegree of each vertex $v \in V$ where $1 \leq d^-(v) \leq d_N(v)$. (N, e_ρ, d^-) is orientable if and only if the following two conditions hold*

1. *(N, e_ρ, d^-) has no degree cut;*
2. $\sum_{v \in V} d^-(v) = |E| + 1$

Every time no orientation can be found the algorithm provides you with the set of edges together forming the degree cut.

Chapter 7

Class orientations

Networks as well as their orientations can be checked for different classes. In this chapter, we describe how to search for an orientation belonging to a specific class. There are two different algorithms using class orientations. Algorithm 3 returns the first orientation it finds and Algorithm 4 returns you all the possibilities. The algorithms need an input consisting of an undirected unrooted binary network, the number of reticulation points and the class to which the orientation should belong. First of all both algorithms loop through all of the edges where the root could be inserted, for each of these options it checks if there exists an orientation for each option of reticulation points. A graph can have multiple orientations corresponding to the same place of the root even though an orientation is unique, this is because each of these orientations has different reticulation points. Every time an orientation is found, the algorithm checks whether the orientation belongs to the given class. If the orientation does belong to the given class it is either returned (Algorithm 3) or added to an array combining all the orientations (Algorithm 4).
These algorithms can only be applied to binary networks, the reason is that the number of reticulation points k is used to generate all combinations of possible sets of reticulation points, this is only sufficient data when orienting binary networks. When orienting non-binary networks the indegree of each vertex should be known.

```
Data: An undirected unrooted binary network $N$, number of reticulation points $k$, a
      class $C$.
Result: One phylogenetic $C$-orientation of $(N, e_\rho, R)$ with the number of reticulation
        points $k$, if it exists and NO otherwise
for each edge $e$ of $N$ do
    for each guess $R \in \binom{V(N)}{k}$ of the $k$ reticulation nodes do
        Compute $N(e,R) =$ BINARY NETWORK ORIENTATION ALGORITHM $(N,e,R)$;
        if $N(e,R)$ is a $C$-orientation then
            Return $N(e,R)$;
    end
end
```

Algorithm 3: Class-orientation algorithm returning one orientation

```
Data: An undirected unrooted binary network $N$, number of reticulation points $k$, a
      class $C$.
Result: All phylogenetic $C$-orientations of $(N, e_\rho, R)$ with the number of reticulation
        points $k$, if it exists and NO otherwise
Set $L :=$ for the root locations and orientations;
for each edge $e$ of $N$ do
    for each guess $R \in \binom{V(N)}{k}$ of the $k$ reticulation nodes do
        Compute $N(e,R) =$ BINARY NETWORK ORIENTATION ALGORITHM $(N, e, R)$;
        if $N(e,R)$ is a $C$-orientation then
            $L = L \cup \{(e, N(e,R))\}$;
            Quit the inner for-loop ;
    end
end
return L
```

Algorithm 4: Class-orientation algorithm returning all orientations

A *biconnected component* of an undirected (non-binary) network is a maximal subgraph that cannot be disconnected by deleting a single vertex. It is called a *blob* if it contains at least three vertices. Algorithms 3 and 4 are based on the third algorithm in the manuscript [HvIJ+]. It differs from the fact that the programmed orientation does not consider blobs, only entire networks. By orienting blobs you reduce the amount of possibilities and therefore minimize the running time. The reason blobs are not considered is that separate dictionaries need to be constructed for each blob on which we can run the algorithm. For many graphs looking at blobs separately would not make a difference since many only have a single blob anyway. Since looking at separate blobs minimizes the running time, different measures have been taken to make sure the algorithm would not take too long. First of all, before generating all combinations of possible sets of reticulation points, leaves are removed from a set of points which will be used to generate all these combinations since a point with only one incident edge can never have an indegree of two. Secondly generating this set of all combinations of possible sets of reticulation points is taken out of the loops, therefore it is only generated once and reused for each possible rooted edge.
The actual implementation of Algorithm 3 and 4 can be found in Appendix A.6.

Chapter 8

Manual

This chapter will give a detailed manual of how to use the actual python program to compute the orientation of a phylogenetic network.

The input for the program should be provided in a `.txt` file, this file should be saved in the same folder as the program. The `.txt` should be named according to the following layout "`graph_.txt`", where at the lower dash a number should be inserted. The `.txt` should have the following layout: each line should proved two connected vertices divided by a single space, there should not be an extra space after the second vertex. Names of vertices can consist of letters, numbers as well as combinations of the two and contain multiple division signs, e.g. dots, lines and lower dashes, but make sure not to use a backslash or blank character (space). See Figure 8.1 for an example.

```
v1 Mbuti
v1 NonAfrican
NonAfrican v2
v2 Eastern_NonAfrican
NonAfrican BasalEurasian
BasalEurasian EEF
Eastern_NonAfrican h1
EEF h1
h1 MA1
```

FIGURE 8.1: `.txt` input example

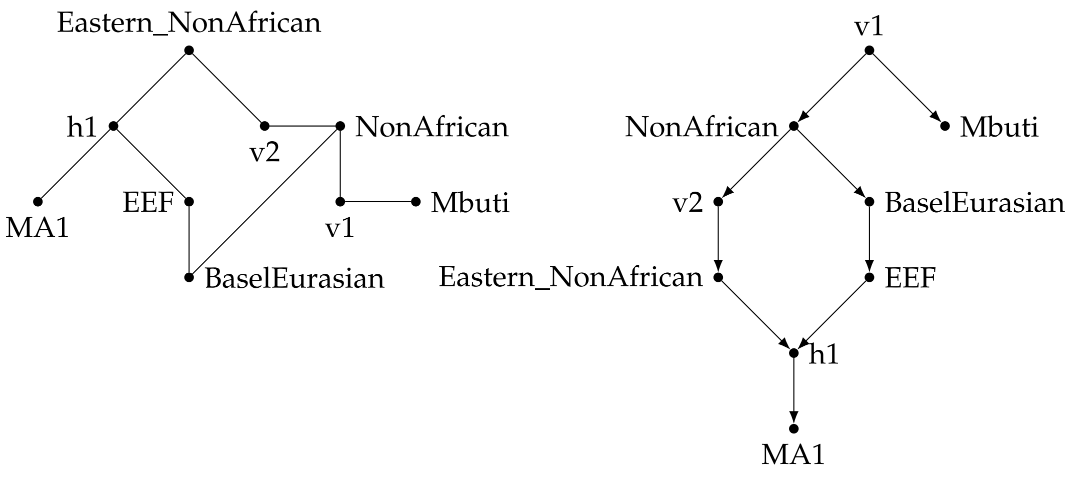

FIGURE 8.2: An undirected (on the left) and directed (on the right) representation of the input given in Figure 8.1

To start the actual program you should run `All.py`. First of all the program will ask what the number of the graph is that you want to open. The answer is the number you inserted in the name of the `.txt` file. Secondly it wants to know if this graph is already directed or not, in case the graph is directed it should also contain the root, otherwise it should not contain the root.
After answering the first two questions it will provide you with the information if the graph is binary or non-binary. Based on whether or not the graph is directed as well as binary or non-binary you might be asked to provide more information.
If the graph is DIRECTED all information necessary is provided.
If the graph is NOT DIRECTED AND BINARY if will ask you to provide it with all of the reticulation points, in case these are known. After the questions about the reticulation points it will ask if you can provide the rooted edge, this is the edge where the root should be inserted.
If the graph is NOT DIRECTED AND NON-BINARY the program will ask you to provide the desired indegree of each of the vertices and afterwards the rooted edge.

Once you have provided the program with all the necessary information it will provide you with the orientation, if one does exist. If you would like to know more about this orientations: it asks you if you would like to know the class of the orientation.

The last part of the program will only be available if the graph is BINARY.
This is the part where the algorithm returns orientations belonging to a desired class. At first it will ask you if you want to check if there is a possible orientation for a place of the root and a combination of reticulation points. If you agree it will ask to which class you want the orientation to belong to. This part will provide you with the first found orientation.
After this it will ask you the same questions but provide you with all of the orientations belonging to the desired class.

All returned orientations are provided in the same layout as the input. Therefore the output can be copied and paste into the website [GGL+] and provide a visualisation of the orientation.

Chapter 9

Data

Throughout the process examples from the website: Visualization of a rooted phylogenetic network [GGL$^+$] are used. The input the website uses is gathered from 31 sources. These sources are listed on the website and clicking directly at the examples at the bottom of the page will load the corresponding network in the form needed to visualise. The same format can be used by the orientation program. The 31 example networks are used as data to check all the possibilities of the orientation program. The results are presented in Tables 9.1 and 9.2. To generate these tables the following is done.

Firstly the directed rooted network is loaded in the program. From this network we could already conclude whether the network was binary or not. The program changes this network to an undirected and unrooted network, for this an orientation was found for which I checked whether it is the same as the orientation from the website, this was the case for all example networks.

After this I replace each directed arc by an undirected edge and I removed the root from the input data and added an edge connecting the vertices earlier incident to the root. This input was again inserted into the algorithm after which I provided the program with the necessary information: if binary: the rooted edge and the reticulation vertices, if non-binary: the rooted edge and the indegree of each vertex.
If the generated orientation did match the website orientation the third column was checked. If there are question marks inserted this means in the non-binary case that the graph was probably very large and inserting the desired indegree would have been a lot of work, and in the binary case for example for graph 6 there were 32 reticulation points.

Using the results from either both or one of the tests, a class check was done. The results are shown in the rows of "1st orientation".

For each of the binary undirected unrooted networks it is checked if an orientation belonging to each of the three classes does exist, if one does exist the first found orientation is returned. For each of these example networks (for which the algorithm run entirely) it was possible to find an orientation beloging to each of the classes. The running times are presented in the table.

The for each of the binary undirected unrooted networks Algorithm 4 finds all orientations belonging to each of the three classes. The running times are presented in the table, these exclude the printing times. In one case the printing took a substantially long time. For graph 30, printing all orientations took for each class about 11 minutes, in each case longer than the actual algorithm. For some graphs there are question marks at the places where running times should be, this is the case for graphs 6, 11 and 26. For these graphs running the algorithm combined with the printing took too long, but the algorithm did not crash. When looking at the other examples we know that the algorithm would have given a result eventually but three extra networks give not as much surplus value as the amount of effort and time it takes finding and printing their orientations. Checking if a class orientation exists and checking for all class orientations are only done for binary networks since this involves using Algorithms 3 and 4, which again refer to Algorithm 1 which only works for binary networks.

As can be seen most running times are acceptable and are matters of seconds. For a few graphs the running time took longer, about 40 to 80 minutes. These graphs are significantly larger and especially for the graphs with a lot of suitable orientations the printing took noteworthy long. This last algorithm returning you all of the possible options regarding one class, running for less than two hours is acceptable in my opinion. To run this algorithm almost no knowledge of the network is required and it provides you with a wide range of information.

TABLE 9.1: First part of the results of the example networks

			1st orientation			an orientation			all orientations		
graph	binary	*	tree-child	stack-free	valid	tree-child	stack-free	valid	tree-child	stack-free	valid
1	×	✓	yes	yes	yes						
2	×	??	yes	yes	yes						
3	×	✓	yes	yes	yes						
4	✓	✓	yes	yes	yes	0.008	0.008	0.016	12.15	12.15	9.216
5	×	✓	no	yes	yes						
6	✓	??	no	no	no	??	??	??	??	??	??
7	×	✓	no	no	no						
8	✓	✓	no	yes	yes	0.030	0.009	0.011	0.362	0.367	0.383
9	×	✓	no	yes	no						
10	×	✓	no	no	no	0.026	0.012	0.010	2.196	2.134	2.154
11	×	✓	no	yes	yes	0.512	0.304	0.307	??	??	??
12	✓	✓	yes	yes	yes	0.009	0.008	0.009	0.068	0.070	0.072
13	×	✓	no	no	no						
14	×	✓	no	no	no						
15	✓	✓	yes	yes	yes	0.011	0.010	0.011	1.048	1.021	1.067

*This column is checked in case the generated orientation does match the website orientation.

TABLE 9.2: Second part of the results of the example networks

			1st orientation			an orientation			all orientations		
graph	binary	*	tree-child	stack-free	valid	tree-child	stack-free	valid	tree-child	stack-free	valid
16	✓	✓	no	no	no	0.016	0.018	0.016	1.504	1.537	1.665
17	✓	✓	yes	yes	yes	0.021	0.010	0.011	2.815	2.803	3.052
18	×	✓	no	yes	yes						
19	×	??	no	no	no						
20	×	??	no	no	no						
21	✓	✓	no	yes	yes	0.119	0.047	0.047	124.1	123.3	126.3
22	✓	✓	yes	yes	yes	0.001	0.002	0.002	0.078	0.075	0.078
23	×	✓	no	yes	yes						
24	✓	✓	no	yes	yes	397	0.076	0.077	2309	2374	2406
25	✓	✓	yes	yes	yes	0.371	0.369	0.373	102.3	103.4	109.7
26	✓	✓	no	no	no	4218	5120	3690	??	??	??
27	✓	✓	yes	yes	yes	0.014	0.015	0.012	6.660	6.600	6.952
28	✓	??	no	no	no						
29	×	✓	no	yes	yes						
30	✓	✓	yes	yes	yes	0.018	0.017	0.019	638.5	625.0	640.0
31	✓	✓	no	yes	yes	0.035	0.011	0.014	48.37	47.92	48.45

*This column is checked in case the generated orientation does match the website orientation.

Chapter 10

Conclusion

In this chapter the main results of this thesis will be discussed. A program has been developed that takes as input a directed rooted network as well as an undirected unrooted network. At first will be checked whether this network is binary or non-binary.
The program can then run one of the following two algorithms.

ORIENTATION ALGORITHM FOR BINARY NETWORKS
Input: An undirected network N, an edge $e_\rho \in E$, and a subset R of the vertices.
Output: A phylogenetic orientation of (N, R, e_ρ) if it exists and NO otherwise.

ORIENTATION ALGORITHM FOR NON-BINARY NETWORKS
Input: An undirected non-binary network N, an edge $e_\rho \in E$, and the desired indegree $d^-(v)$ with $1 \leq d^-(v) \leq d_N(v)$ for each $v \in V$.
Output: A phylogenetic orientation of (N, e_ρ, d^-) if it exists and NO otherwise.

Looking at the test data can be concluded that these algorithms always return the correct orientation if it exists. In case there is a reticulation cut, in the binary case, or the degree cut, in the non-binary case, the edges belonging will be returned. This has been checked using the examples given in Figure 5.1 and 6.1.

Moreover, the program can run the following two algorithms.

CLASS-ORIENTATION ALGORITHM RETURNING ONE ORIENTATION
Input: An undirected unrooted binary network N, number of reticulation points k, a class C.
Output: One phylogenetic C-orientation of (N, e_ρ, R) with the number of reticulation points k, if it exists and NO otherwise.

CLASS-ORIENTATION ALGORITHM RETURNING ALL ORIENTATIONS
Input: An undirected unrooted binary network N, number of reticulation points k, a class C.
Output: All phylogenetic C-orientations of (N, e_ρ, R) with the number of reticulation points k, if it exists and NO otherwise.

Looking at the test data can be concluded that for each of these example networks (for which we let the algorithm run entirely) it was possible to find an orientation belonging to each of the classes. For most graphs it was also possible to return all orientations belonging to a required class, doing this within acceptable running time.

Chapter 11

Discussion

In this chapter we will discuss choices that could have been made differently and how that might result into different outcomes, followed by possible extensions to the program as well as the theory.
The first made programming choice is the representation of a network as a python dictionary. Due to this choice it is not possible to iterate over the edges of the network in the first two algorithms. If a different implementation is chosen for a network this could result into iterating over the edges. You could divide these edges in two groups, directed and undirected in this case.

In the manuscript this thesis is based on besides Tree-child, Stack-free and Valid also the classes Reticulation-visible, Orchard and Tree-based are discussed. Their definitions are given below:

- **Tree-based**: it can be obtained from a rooted tree (of which the root may have out-degree 1 or 2) by subdividing arcs of the tree (any number of times) and adding arcs between the subdividing vertices [FS15, JvI18].

In Figure 11.1 we can see a directed network and it underlying tree-base. In Figure 11.2 at the top left we see the rooted tree on which the network of 11.1 is based. In the top right figure the edges $\{r, l_1\}$ and $\{v_2, l_1\}$ have been subdivided by the vertices v_1 and v_4, between which an arc has been added. In the bottom left figure the edges $\{v_4, l_2\}$ and $\{v_6, l_3\}$ have been subdivided by the vertices v_5 and v_8, between which an arc has been added. In the bottom right figure the edges $\{v_1, l_1\}$ and $\{v_5, l_2\}$ have been subdivided by the vertices v_3 and v_7, between which an arc has been added. Now we have constructed the network from a tree-base.

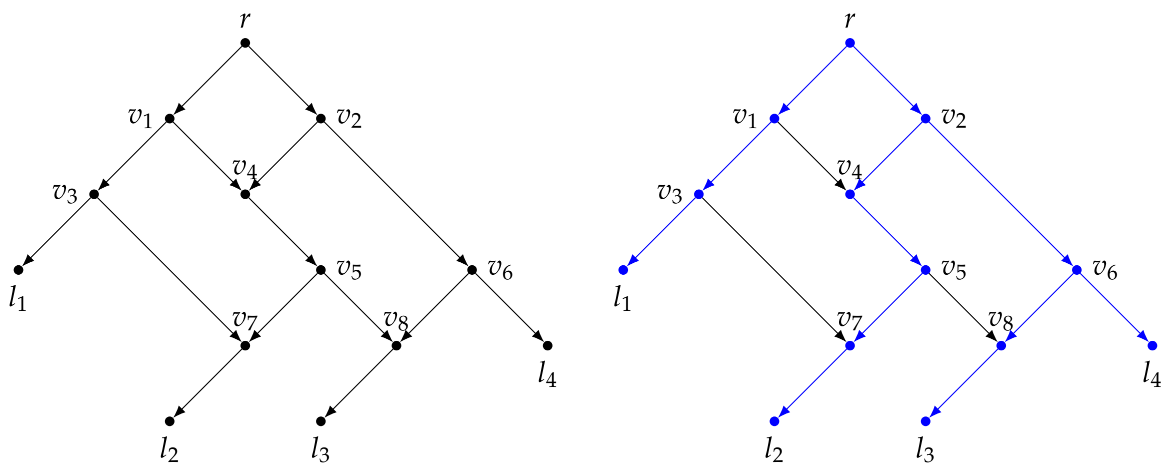

FIGURE 11.1: A network and its underlying tree-base

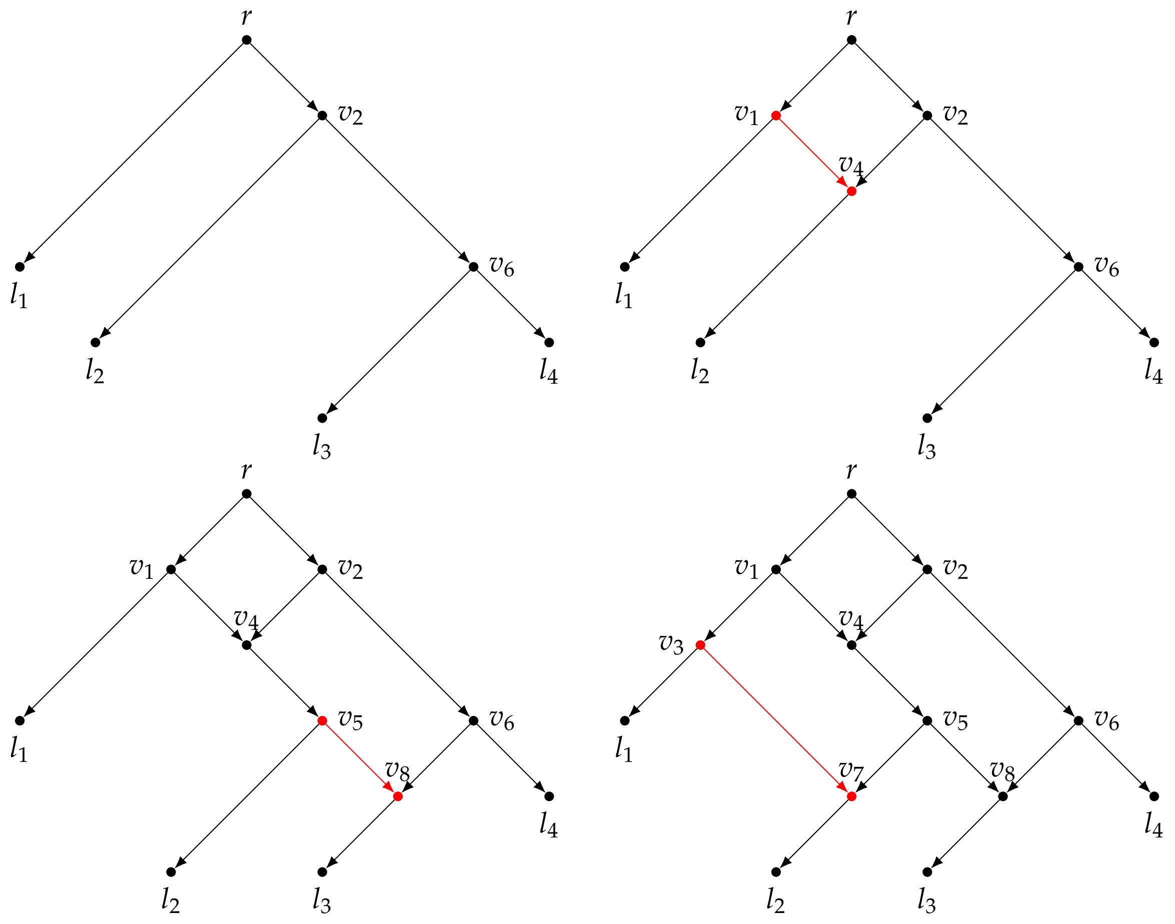

FIGURE 11.2: The network of Figure 11.1 obtained by subdividing arcs of the tree and adding arcs between the subdividing vertices

By deleting a *cut-arc* a connected network becomes a disconnected network.

- **Reticulation-visible**: from each reticulation there is a path to a cut-arc that does not contain any further reticulations.

A further class of directed binary networks is defined based on the notion of cherry picking. Two leaves with a common parent are called a *cherry*. Picking a cherry means deleting one leaf of a cherry and suppressing its parent, see Figure 11.3. A *reticulated cherry* is a pair of leaves connected by an undirected path with two internal vertices, exactly one of which is a reticulation. Picking a reticulated cherry consists of deleting the middle arc of this path and suppressing its endpoints, see Figure 11.4.

- **Orchard**: it can be reduced to a single cherry by picking cherries and reticulated cherries.

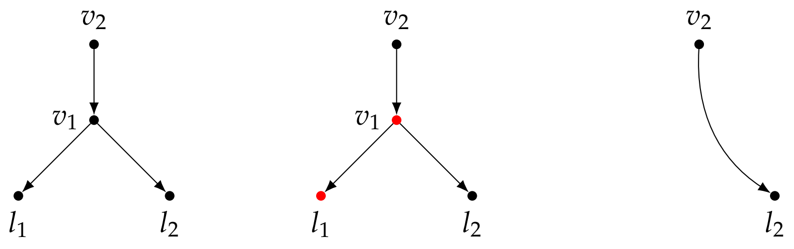

FIGURE 11.3: Cherry picking

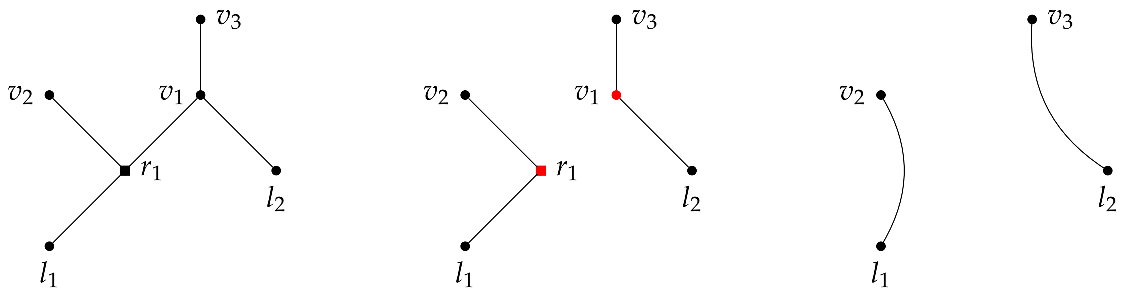

FIGURE 11.4: Reticulate cherry picking

All classes discussed up to now relate to each other in the way shown in Figure 11.5. These class checks could be implemented and by adding them to Algorithms 3 and 4 we are able to extend the algorithms such that they can also find orientations belonging to each of these classes.

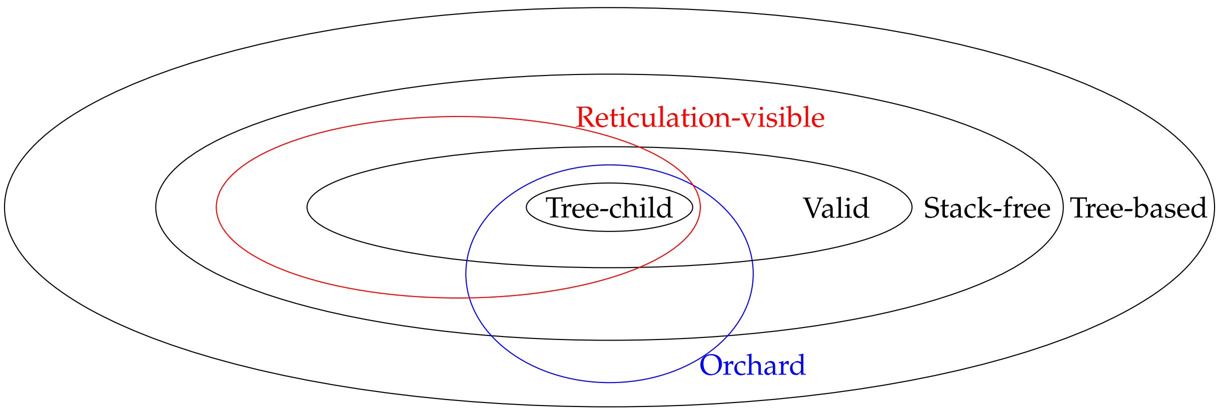

FIGURE 11.5: A visualisation of the relations of the classes.

In phylogenetic networks often a part of the edges is directed and a part is not. A partly-directed network is a mixed graph (a graph that may contain both undirected edges and directed arcs) obtained from an undirected network by orienting a subset of its edges as directed arcs. A semi-directed network is a mixed graph obtained from a directed binary network by unorienting all arcs except for arcs entering reticulation vertices and suppressing the former root [SLA16]. If an orientation is found for a certain undirected network given a rooted edge and set of reticulation points or indegree, this orientation is unique. Therefore it could be checked to match the edges for which you knew the direction. This could of course also be done for the orientations found using Algorithms 3 and 4.

A *biconnected component* of an undirected (non-binary) network is a maximal subgraph that cannot be disconnected by deleting a single vertex. It is called a *blob* if it contains at least three vertices. Algorithms 3 and 4 are changed when comparing it to the matching algorithms represented in the manuscript. These algorithms could be applied to blobs and combining the outcomes results into multiple orientations. By adding the blob part these algorithms have to check less possible options, this could be profitable for the running time.

Appendix A

Appendix

A.1 Program

```
#Name: All.txt

import EdgestoGraph as EtG
import Alg1
import Binarycheck as Bc
import Alg2
import Alg3
import ClassCheck as cc
import time

number = raw_input("What is the number of the graph you want to open? ")
graph  = "graph"+ str(number)+ ".txt"
g = open(graph, "r")

if g.mode == "r":
    S = g.read()

S = S + '\n'
Directed = raw_input("Is this tree directed? If so, it should also contain the root. Yes or
 ↪  No: ")

if Directed == 'Yes':
    dgraph = EtG.StringtoDirectedGraph(S)
    undgraph = EtG.StringtoUndirectedGraph(S)
    undunrgraph = EtG.StringtoUndirectedUnrootedGraph(S)
    Binary = Bc.BinarycheckRoot(undunrgraph)
    root = EtG.Root(dgraph)
    if Binary:
        print "This graph is binary!"
    if not Binary:
        print "This graph is not binary! "
if Directed == 'No':
    undunrgraph = EtG.StringtoUndirectedGraph(S)
    Binary = Bc.BinarycheckRoot(undunrgraph)
    if Binary:
        print "This graph is binary!"
    if not Binary:
        print "This graph is not binary! "
```

```python
    Know1 = ''
    Know2 = ''
    Know3 = ''
    #Reticulation points
    if Binary:
        Know1 = raw_input("Do you know the reticulation points? Yes or No: ")
        R = []
        if Know1 == 'Yes':
            print "Please enter the reticulation points one by one, leave empty when done. "
            while True:
                point = raw_input("Reticulation point = ")
                if len(point)>0:
                    R.append(point)
                else:
                    break
    #degree
    if not Binary:
        degree = []
        Know3 = raw_input("Do you know the indegree of the vertices? Yes or No: ")
        if Know3 == 'Yes':
            print "Please enter the indegree for each point."
            for v in undunrgraph.keys():
                ind =  raw_input("What is the indegree of " + str(v) + " ? ")
                degree.append((v,int(ind)))
    #edge
    if Know1 == 'Yes' or Know2 == 'Yes' or Know3 == 'Yes':
        Know4 = raw_input("Do you know the edge where the root should be inserted? Yes or No: ")
        if Know4 == 'Yes':
            x = raw_input("Where should the root be? The root is connected to (give 1  point first): ")
            y = raw_input("and to: ")
            e_p = (x,y)
        else:
            e_p = ()

if Binary and Directed == 'Yes':
    R = EtG.R(dgraph)
    Orient = Alg1.OrientationAlgRoot(undgraph, root, R)
if not Binary and Directed == 'Yes':
    degree = Alg2.Indegree(dgraph)
    Orient = Alg2.OrientationAlgRoot(undgraph, root, degree)

if Binary and Directed == 'No':
    if R == []:
        RNumber = Alg1.NumberofEdges(undunrgraph)/2-len(undunrgraph)+1
        Orient = Alg3.OneOrientationAlg(undunrgraph, RNumber, '')[1]
    else:
        if e_p == ():
            E = []
            for i in undunrgraph.keys():
                for j in undunrgraph[i]:
                    if (i,j) not in E and (j,i) not in E:
                        E.append((i,j))
            for e_p in E:
                Orient = Alg1.OrientationAlgEp(undunrgraph, e_p, R)
                if isinstance(Orient,dict):
                    print "The edge for which an orientation is found is "+str(e_p)+"."
                    break
        else:
            Orient = Alg1.OrientationAlgEp(undunrgraph, e_p, R)

if not Binary and Directed == 'No':
    if degree == []:
        print "Sorry, not enough information for this program to find an orientation"
    else:
        if e_p == ():
            E = []
            for i in undunrgraph.keys():
                for j in undunrgraph[i]:
                    if (i,j) not in E and (j,i) not in E:
                        E.append((i,j))
            for e_p in E:
                Orient = Alg2.OrientationAlgEp(undunrgraph, e_p, degree)
                if isinstance(Orient,dict):
                    print "The edge for which an orientation is found is "+str(e_p)+"."
                    break
                else:
                    Orient = "No orientation exists for all places of the root"
        else:
            Orient = Alg2.OrientationAlgEp(undunrgraph, e_p, degree)

if isinstance(Orient,dict):
    pOrient = EtG.GraphtoString(Orient) #printable Orientation
    p = raw_input("Do you want the orientation printed? Yes or No: ")
    if p == "Yes":
        print "The Orientation is: \n", pOrient
    #following can only be done if orientation exists
    ClassCheck = raw_input("Do you want to know the class of the orientation? Yes or No: ")

    if ClassCheck == 'Yes':
        if cc.TreeChild(Orient):
            print "The orientation is tree-child!"
        else:
            print "The orientation is not tree-child!"
        if cc.StackFree(Orient):
            print "The orientation is stack-free!"
        else:
            print "The orientation is not stack-free!"
        if cc.Valid(Orient):
            print "The orientation is valid!"
        else:
            print "The orientation is not valid!"
else:
    print Orient

L=""
if Binary:
    Check = raw_input("Do you want to check if there is a possible orientation for a place of
    ↪  the root and a combination of reticulation points? Yes or No: ")

    if Check == 'Yes':
        Class = raw_input("What class should the orientation be? \n For tree-child enter: 1
        ↪  \n For stack-free enter: 2 \n For valid enter:      3 ")
        RNumber = Alg1.NumberofEdges(undunrgraph)/2-len(undunrgraph)+1
        print "The original graph had " + str(RNumber) + " reticulation point(s)."
        start = time.time()
        if Class == '1':
            L = Alg3.OneOrientationAlg(undunrgraph, RNumber, "TreeChild")
        elif Class == '2':
            L = Alg3.OneOrientationAlg(undunrgraph, RNumber, "StackFree")
        elif Class == '3':
            L = Alg3.OneOrientationAlg(undunrgraph, RNumber, "Valid")
        else:
            print "There are no orientations meeting the criteria."
        end = time.time()

        tekst = ""

        if len(L)>0:
            o = EtG.GraphtoString(L[1])
            tekst = tekst + "When the root is inserted in edge " + str(L[0]) + ", the
            ↪  orientation is: " + o + "\n \n"
        elif len(L) == 0:
            tekst = "There are no orientations meeting the criteria."
        print tekst
        print "to find one orientation took (in seconds) :", str(end-start)

L=""
if Binary:
    Check = raw_input("Do you want to check all the possible orientations for all places of
    ↪  the root and all combinations of reticulation points? Yes or No: ")

    if Check == 'Yes':
        Class = raw_input("What class should the orientations be? \n For tree-child enter: 1
        ↪  \n For stack-free enter: 2 \n For valid enter:      3 ")
        RNumber = Alg1.NumberofEdges(undunrgraph)/2-len(undunrgraph)+1
        print "The original graph had " + str(RNumber) + " reticulation point(s)."
        start = time.time()
        if Class == '1':
            L = Alg3.OrientationAlg(undunrgraph, RNumber, "TreeChild")
        elif Class == '2':
            L = Alg3.OrientationAlg(undunrgraph, RNumber, "StackFree")
        elif Class == '3':
            L = Alg3.OrientationAlg(undunrgraph, RNumber, "Valid")
        else:
            print "There are no orientations meeting the criteria."
        end = time.time()

        tekst = ""
        start1 = time.time()
        for i in range(0,len(L)):
            o = EtG.GraphtoString(L[i][1])
            tekst = tekst + "When the root is inserted in edge " + str(L[i][0]) + ", the
             ↪  orientation is: " + o + "\n \n"
        if len(L)==0:
            tekst = "There are no orientations meeting the criteria."

        print tekst
        end1 = time.time()

        print "to find all orientations took (in seconds) :", str(end-start)
        print "printing took (in seconds) :", str(end1-start1)
```

A.2 Binary check

```
#Name: Binarycheck.txt

def BinarycheckRoot(graph):
    #needs undirected graph
    root=[]
    internal = []
    leaf = []
    for k in graph.keys():
        if len(graph[k])==2:
            root.append(k)
        elif len(graph[k])==3:
            internal.append(k)
        elif len(graph[k])==1:
            leaf.append(k)
    if len(root) == 0: #or len(root) == 1:
        if len(root)+len(internal)+len(leaf)==len(graph):
            return True
        else:
            return False
    else: return False
```

A.3 Algorithm one

```
#Name: Alg1.txt

import ReticulationCut as RC
import copy

```

```
def NumberofEdges(graph):
    lengths = [len(v) for v in graph.values()]
    sum = 0
    for num in lengths:
        sum += num
    return sum

def OrientationAlgEp(graph1, e_p, R):
    graph = copy.deepcopy(graph1)
    if len(R) != NumberofEdges(graph)/2 - len(graph)+1:
        return "NO, wrong summation"
    NOE = NumberofEdges(graph)

    #subdevide e_p by a new vertex p and orient the two edges incident to p away from p
    graph["p"]=[e_p[0], e_p[1]]

    #remove a to b and b to a
    x = graph.get(e_p[0])
    x.remove(e_p[1])
    graph.update({ e_p[0]:x})
    y = graph.get(e_p[1])
    y.remove(e_p[0])
    graph.update({ e_p[1]:y})

    orientable_R = []
    treenode = dict.keys(graph)
    orientable_treenode= []
    for r in R:
        treenode.remove(r)
    for i in [0,1]:
        if e_p[i] in treenode:
            if len(graph[e_p[i]])>0:
                orientable_treenode.append(e_p[i])

    while NumberofEdges(graph) > NOE/2+1+1:
        while len(orientable_treenode)>0:
            for v in orientable_treenode:
                x = graph[v]
                if len(graph[v]) != 2:
                    return "NO, wrong treenode"
                y = graph.get(x[0])
                y.remove(v)
                graph.update({x[0]:y})
                if x[0] in treenode:
                    if len(graph[x[0]])==2:
                        orientable_treenode.append(x[0])
                elif x[0] in R:
                    if len(graph[x[0]])==1:
                        orientable_R.append(x[0])
                z = graph.get(x[1])
                z.remove(v)
                graph.update({x[1]:z})
                if x[1] in treenode:
                    if len(graph[x[1]])==2:
                        orientable_treenode.append(x[1])
                elif x[1] in R:
                    if len(graph[x[1]])==1:
                        orientable_R.append(x[1])
                orientable_treenode.remove(v)
        while len(orientable_R)>0:
            for i in orientable_R:
                if len(graph[i])==1: #always in case binary
                    x = graph[i]
                    y = graph[x[0]]
                    y.remove(i)
                    if x[0] in treenode:
                        if len(y)==2:
                            orientable_treenode.append(x[0])
                    elif x[0] in R: #in case not funneled
                        if len(y)==1:
                            orientable_R.append(x[0])
                    graph.update({x[0]:y})
                    orientable_R.remove(i)
                if len(graph[i])==0:
                    orientable_R.remove(i)


        if len(orientable_treenode)==0 and len(orientable_R)==0:
            for i in graph.keys(): #check if still undirected edges, in case reticulation cut
                for j in graph.keys():
                    for k in range(0,len(graph[j])):
                        if i == graph[j][k]:
                            for l in range(0,len(graph[i])):
                                if j==graph[i][l]:
                                    RetCut= RC.ReticulationCutBin(graph,R)
                                    return "NO, there exists a reticulation cut, namely:" +
                                     ↪  str(RetCut)
            break #else no reticulation cut
    return graph

def OrientationAlgRoot(graph, Root, R):
    Root = Root[0]
    if len(R) != NumberofEdges(graph)/2 - len(graph) +1: #-1-1
        return "Eerste NO"
    NOE = NumberofEdges(graph)

    x = graph[graph[Root][0]]
    x.remove(Root)
    graph.update({graph[Root][0]:x})
    if len(graph[Root])>1:
        y = graph[graph[Root][1]]
        y.remove(Root)
        graph.update({graph[Root][1]:y})

    orientable_R = []
    treenode = dict.keys(graph)
    orientable_treenode= []
    for r in R:
        treenode.remove(r)
    x = graph[Root][0]
    if x in treenode:
        if len(graph[x])==2:
            orientable_treenode.append(x)
    if len(graph[Root])>1:
        y = graph[Root][1]
        if y in treenode:
            if len(graph[y])==2:
                orientable_treenode.append(y)
    while NumberofEdges(graph) > NOE/2+1+1:
        while len(orientable_treenode)>0:
            for v in orientable_treenode:
                x = graph[v]
                y = graph.get(x[0])
                y.remove(v)
                graph.update({x[0]:y})
                if x[0] in treenode:
                    if len(graph[x[0]])==2:
                        orientable_treenode.append(x[0])
                elif x[0] in R:
                    if len(graph[x[0]])==1:
                        orientable_R.append(x[0])
                z = graph.get(x[1])
                z.remove(v)
                graph.update({x[1]:z})
                if x[1] in treenode:
                    if len(graph[x[1]])==2:
                        orientable_treenode.append(x[1])
                elif x[1] in R:
                    if len(graph[x[1]])==1:
                        orientable_R.append(x[1])
                orientable_treenode.remove(v)
        while len(orientable_R)>0:
            for i in orientable_R:
                if len(graph[i])==1:
                    x = graph[i]
                    y = graph[x[0]]
                    y.remove(i)
                    if x[0] in treenode:
                        if len(y)==2:
                            orientable_treenode.append(x[0])
                    elif x[0] in R: #in case not funneled
                        if len(y)==1:
                            orientable_R.append(x[0])
                    graph.update({x[0]:y})
                    orientable_R.remove(i)
        if len(orientable_treenode)==0 and len(orientable_R)==0:
            for i in graph.keys(): #check if still undirected edges, in case reticulation cut
                for j in graph.keys():
                    for k in range(0,len(graph[j])):
                        if i ==graph[j][k]:
                            for l in range(0,len(graph[i])):
                                if j==graph[i][l]:
                                    RetCut= RC.ReticulationCutBin(graph,R)
                                    return "NO, there exists a reticulation cut, namely:" +
                                     ↪  str(RetCut)
            break
    return graph
```

A.4 Algorithm two

```
#Name: Alg2.txt

import Alg1
import EdgestoGraph as EtG
import Binarycheck as Bc
import Alg2
import Alg3
import ClassCheck as cc
import ReticulationCut as RC
import copy

def Indegree(graph):
    #directed graph needed
    degree = []
    for k in graph.keys():
        for i in graph[k]:
            y = [item for item in degree if item[0]==i]
            if len(y)==0:
                degree.append((i,1))
            else: #len y ==1
                degree.remove((i,y[0][1]))
                z=y[0][1]+1
                degree.append((i,z))
    return degree

def IndegreeSum(degree):
    Sum = 0
    for i in range(0,len(degree)):
        Sum = Sum + degree[i][1]
    return Sum

def OrientationAlgEp(graph1, e_p, degree):
    graph = copy.deepcopy(graph1)
    #non rooted graph needed
    if IndegreeSum(degree) != Alg1.NumberofEdges(graph)/2+1 :
        print IndegreeSum(degree)
        print Alg1.NumberofEdges(graph)
        return "NO (sommatie incorrect)"
    #subdevide e_p by a new vertex p and orient the two edges incident to p away from p
    graph["p"]=[e_p[0], e_p[1]]
    orientable = []
    #remove a to b and b to a
    x = graph.get(e_p[0])
    if e_p[1] in x:
        x.remove(e_p[1])
    graph.update({ e_p[0]:x})
    y = graph.get(e_p[1])
    if e_p[0] in y:
        y.remove(e_p[0])

    graph.update({ e_p[1]:y})
    for i in [0,1]:
        if len([item for item in degree if item[0]==e_p[i]])>0:
            if 1== [item for item in degree if item[0]==e_p[i]][0][1]:
                orientable.append(e_p[i])
        else:
            orientable.append(e_p[i])
    NOE = Alg1.NumberofEdges(graph)
    while Alg1.NumberofEdges(graph) > NOE/2+1+1:
        while len(orientable)>0:
            for v in orientable:
                count = 0
                for j in graph.keys():
                        for k in range(0,len(graph[j])):
                            if v==graph[j][k]:
                                count = count + 1
                if (count-len(graph[v]))== [item for item in degree if item[0]==v][0][1]:
                    l = len(graph[v])
                    for i in range(0,l):
                        x = graph[v]
                        y = graph.get(x[i])
                        y.remove(v)
                        graph.update({x[i]:y})
                        teller = 0
                        for j in graph.keys():
                            for k in range(0,len(graph[j])):
                                if x[i]==graph[j][k]:
                                    teller = teller + 1
                        if len([item for item in degree if item[0]==x[i]])>0:
                            if (teller-len(graph[x[i]]))== [item for item in degree if
                             ↪  item[0]==x[i]][0][1]:
                                orientable.append(x[i])
                orientable.remove(v)
        if len(orientable)==0:
            for i in graph.keys(): #check if still undirected edges, in case reticulation cut
                for j in graph.keys():
                    for k in range(0,len(graph[j])):
                        if i ==graph[j][k]:
```

```python
                            for l in range(0,len(graph[i])):
                                if j==graph[i][l]:
                                    print graph
                                    RetCut = RC.ReticulationCutNonbin(graph,degree)

                                    return "NO, there exists a reticulation cut, namely: " + str(RetCut)
            break #else no reticulation cut

    return graph

def OrientationAlgRoot(graph, Root, degree):
    #rooted graph needed
    Root = Root[0]
    if IndegreeSum(degree)-1 != (Alg1.NumberofEdges(graph)-2)/2 :
        return "NO (sommatie incorrect)"

    orientable = []
    #still turn edge to arc
    for i in range(0,len(graph[Root])):
        x = graph[graph[Root][i]]
        x.remove(Root)
        graph.update({graph[Root][i]:x})
        if len([item for item in degree if item[0]==graph[Root][i]])>0:
            if 1 == [item for item in degree if item[0]==graph[Root][i]][0][1]:
                   orientable.append(graph[Root][i])

    NOE = Alg1.NumberofEdges(graph)
    while Alg1.NumberofEdges(graph) > NOE/2+1+1:
        while len(orientable)>0:
            for v in orientable:
                l = len(graph[v])
                for i in range(0,l):
                    x = graph[v]
                    y = graph.get(x[i])
                    y.remove(v)
                    graph.update({x[i]:y})
                    teller = 0
                    for j in graph.keys():
                        for k in range(0,len(graph[j])):
                            if x[i]==graph[j][k]:
                                teller = teller + 1
                    if len([item for item in degree if item[0]==x[i]])>0:
                        if (teller-len(graph[x[i]]))== [item for item in degree if item[0]==x[i]][0][1]:
                            orientable.append(x[i])

                orientable.remove(v)

        if len(orientable)==0:
            for i in graph.keys(): #check if still undirected edges, in case reticulation cut
                for j in graph.keys():
                    for k in range(0,len(graph[j])):
                        if i ==graph[j][k]:
                            for l in range(0,len(graph[i])):
                                if j==graph[i][l]:
                                    RetCut = RC.ReticulationCutNonbin(graph,degree)
                                    return "NO, there exists a reticulation cut, namely: " +
                                     ↪   str(RetCut)
            break #else no reticulation cut

    return graph
```

A.5 Network classes

```
#Name: ClassCheck.txt

import EdgestoGraph as EtG

def TreeChild(graph):
    #directed graph needed
    #every non leaf vertex has a child that is not a reticulation
    R = EtG.R(graph)
    Check = graph.keys()
    for v in graph.keys():
        if len(graph[v])==0:
            Check.remove(v)
        for w in graph[v]:
            if w not in R:
                if v in Check:
                    Check.remove(v)
    if len(Check)==0:
        return True
    else:
        return False

def StackFree(graph):
    #directed graph needed
    #no reticulation with a reticulation as a child
    R = EtG.R(graph)
    for v in R:
        for w in graph[v]:
            if w in R:
                return False
    return True

def Valid(graph):
    #directed graph needed
    #stack-free and deleting a single reticulation edge and suppressing its endpoints does
     ↪   not give parallel arcs.
    if StackFree(graph):
        R = EtG.R(graph)
        for r in R:
            for v in graph.keys():
                if r in graph[v]:
                    if len(graph[v])==2:
                        if r == graph[v][0]:
                            s = graph[v][1]
                        if r == graph[v][1]:
                            s = graph[v][0]
                        for k in graph.keys():
                            if v in graph[k]:
                                if s in graph[k]:
                                    return False
        return True
    return False
```

A.6 Algorithm three

```
#Name: Alg3.txt

import Alg1
import itertools
import ClassCheck as cc
import Binarycheck as Bc
import time

def RCombinations(V, RNumber):
    start = time.time()
    it = itertools.combinations(V, RNumber)
    RC = [c for c in it]
    end = time.time()
    print "Rcombinations took (seconds): ", str(end-start)
    return RC

    #Alg 4 in thesis
def OrientationAlg(graph, RNumber, Class):
    #Orientation algorithm for undirected unrooted binary networks
    L = []
    #all edges in one set
    E = []
    for i in graph.keys():
        for j in graph[i]:
            if (i,j) not in E and (j,i) not in E:
                E.append((i,j))

    V = graph.keys()
    Ropt = graph.keys()
    for v in V:
        if len(graph[v]) == 1:
            Ropt.remove(v)
    if RNumber > 1:
        RC = RCombinations(Ropt,RNumber)
    if RNumber == 1:
        RC = [{r} for r in Ropt]
    for e_p in E:
        print "e_p = ", e_p
        for rc in RC:
            Orient = Alg1.OrientationAlgEp(graph, e_p, rc)
            if isinstance(Orient,dict):
                if Class == 'TreeChild':
                    if cc.TreeChild(Orient):
                        L.append((e_p,Orient))
                if Class == 'StackFree':
                    if cc.StackFree(Orient):
                        L.append((e_p,Orient))
                if Class == 'Valid':
                    if cc.Valid(Orient):
                        L.append((e_p,Orient))
                if Class == '':
                    L.append((e_p,Orient))
    return L

    #Alg 3 in thesis
def OneOrientationAlg(graph, RNumber, Class):
    #Orientation algorithm for undirected unrooted binary networks

    #all edges in one set
    E = []
    for i in graph.keys():
        for j in graph[i]:
            if (i,j) not in E and (j,i) not in E:
                E.append((i,j))

    V = graph.keys()
    Ropt = graph.keys()
    for v in V:
        if len(graph[v]) == 1:
            Ropt.remove(v)
    if RNumber > 1:
        RC = RCombinations(Ropt,RNumber)
    if RNumber == 1:
        RC = [{r} for r in Ropt]
    for e_p in E:
        print "e_p = ", e_p
        for rc in RC:
            Orient = Alg1.OrientationAlgEp(graph, e_p, rc)
            if isinstance(Orient,dict):
                if Class == 'TreeChild':
                    if cc.TreeChild(Orient):
                        return (e_p,Orient)
                if Class == 'StackFree':
                    if cc.StackFree(Orient):
                        return (e_p,Orient)
                if Class == 'Valid':
                    if cc.Valid(Orient):
                        return (e_p,Orient)
                if Class == '':
                    return (e_p,Orient)
    return []
```

A.7 Layout

```
#Name: EdgestoGraph.txt

def StringtoComb(S):
    combi = S.splitlines()
    pairs = []
    a = ""
    b = ""
    for item in combi:
        for j in range(0,len(item)):
            if item[j] == ' ':
                for k in range(0,j):
                    a = a + item[k]
                for l in range(j+1,len(item)):
                    b = b + item[l]
                pairs.append((a,b))
                a = ""
                b = ""
    return pairs

def CombtoDirectedGraph(P):
    graph={}
    for i in range(0,len(P)):
        a,b = P[i]
        if a in graph.keys():
            x = graph.get(a)
            x.append(b)
            graph.update({a:x})
        else:
            graph[a]=[b]
    return graph

def CombtoUndirectedGraph(P):
    graph={}
    for i in range(0,len(P)):
        a,b = P[i]
        if a in graph.keys():
            x = graph.get(a)
            x.append(b)
            graph.update({a:x})
        else:
            graph[a]=[b]
        if b in graph.keys(): #only if edges not arcs
            x= graph.get(b)
            x.append(a)
            graph.update({b:x})
        else:
            graph[b]=[a]
    return graph


def StringtoDirectedGraph(S):
    P = StringtoComb(S)
    graph = CombtoDirectedGraph(P)
    return graph

def StringtoUndirectedGraph(S):
    P = StringtoComb(S)
    graph = CombtoUndirectedGraph(P)
    return graph

def StringtoUndirectedUnrootedGraph(S):
    DirectedGraph = StringtoDirectedGraph(S)
    root = Root(DirectedGraph)[0]
    UndirectedGraph = StringtoUndirectedGraph(S)
    a = UndirectedGraph[root][0]
    x = UndirectedGraph.get(a)
    x.remove(root)

    if len(UndirectedGraph[root])>1:
        b = UndirectedGraph[root][1]
        y = UndirectedGraph.get(b)
        y.remove(root)
        y.append(a)
        x.append(b)
        UndirectedGraph.update({b:y})
    UndirectedGraph.update({a:x})
    del UndirectedGraph[root]
    return UndirectedGraph


def Root(graph):
    root = graph.keys()
    for i in graph.keys():
        for j in graph.keys():
            for k in range(0,len(graph[j])):
                if i == graph[j][k]:
                    if i in root:
                        root.remove(i)
    return root

def R(graph):
    R = []
    for i in graph.keys():
```

```python
        t = 0
        for j in graph.keys():
            if i in graph[j]:
                t = t + 1
        if t > 1:
            R.append(i)
    return R


def GraphtoComb(graph):
    #directed graph needed
    pairs = []
    for a in graph.keys():
        for b in graph[a]:
            pairs.append((a,b))
    return pairs

def CombtoString(pairs):
    #array with pairs needed
    string = ""
    for pair in pairs:
        string = string + "\n" + pair[0] + " " +pair[1]
    return string

def GraphtoString(graph):
    return CombtoString(GraphtoComb(graph))
```

Bibliography

[AvIJ18] Karen Aardal, Leo van Iersel, and Remie Janssen. Optimization, 2018. Lecture Notes TW2020.

[ESS19] Peter L. Erdős, Charles Semple, and Mike Steel. A class of phylogenetic networks reconstructable from ancestral profiles, 2019. arXiv: 1901.04064.

[Fou] Python Software Foundation. Python, 5.5. dictionaries. https://docs.python.org/2/tutorial/datastructures.html. (accessed: 23.04.2019).

[FS15] Andrew R. Francis and Mike Steel. Which Phylogenetic Networks are Merely Trees with Additional Arcs? *Systematic Biology*, 64(5):768–777, 06 2015.

[GGL$^+$] Philippe Gambette, Andreas Gunawan, Anthony Labarre, Stéphane Vialette, and Louxin Zhang. Visualization of a rooted phylogenetic network. http://phylnet.univ-mlv.fr/recophync/networkDraw.php#form_n26. (accessed: 23.04.2019).

[HRS10] Daniel H. Huson, Regula Rupp, and Celine Scornavacca. *Phylogenetic Networks*. Cambridge University Press, 2010. ISBN: 9780511974076.

[HvIJ$^+$] Katharina Huber, Leo van Iersel, Remies Janssen, Mark Jones, Vincent Moulton, Yukihiro Murakami, and Charles Semples. Rooting for phylogenetic networks.

[Ims] Eleanor Imster. Scientists describe 229 new species in 2018. https://earthsky.org/earth/new-species-2018. (accessed: 30.04.2019).

[JM18] Remie Janssen and Yukihiro Murakami. Solving phylogenetic network containment problems using cherry-picking sequences, 2018. arXiv: 1812.08065.

[JvI18] Laura Jetten and Leo van Iersel. Nonbinary tree-based phylogenetic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(1):205–217, 2018.

[MvIJ$^+$18] Yukihiro Murakami, Leo van Iersel, Remie Janssen, Mark Jones, and Vincent Moulton. Reconstructing tree-child networks from reticulate-edge-deleted subnetworks, 2018. arXiv: 1811.06777.

[SLA16] Claudia Solís-Lemus and Cécile Ané. Inferring phylogenetic networks with maximum pseudolikelihood under incomplete lineage sorting. *PLoS Genetics*, 12(3):e1005896, 2016.