Parental Parsimony on Phylogenetic Networks

Computing the Maximum Parsimony Scores of Phylogenetic Networks

Niels Jonkman

Bachelor Graduation Thesis

Parental Parsimony on Phylogenetic Networks

Computing the Maximum Parsimony Scores of Phylogenetic Networks

Bachelor Graduation Thesis

Niels Jonkman
4289765

June 30, 2017

Project duration:	april, 2016 – june 29, 2017	
Thesis committee:	Dr. ir. L. J. J. van Iersel,	TU Delft, supervisor
	M. B. L. Jones,	TU Delft, supervisor
	Dr. C. Kraaikamp,	TU Delft
	Dr. J. G. Spandaw,	TU Delft

Faculty of Electrical Engineering, Mathematic and Computer Science (EEMCS)
Delft University of Technology

Delft
University of
Technology

Preface

The thesis that lies before you is the result of a project that was done under the supervision of Leo van Iersel and Mark Jones. I want to thank my supervisers and mentors for their support. Without their guidance, this thesis would have been an impossible project for me. Futhermore, I would like to thank my parents, for their unconditional loving and their trust in my abilities. This helps me to bring out the best in me.

W. N. Jonkman
June 2017

Abstract

Phylogenetic networks represent the evolutionary history of organisms and the relationships among them. In the field of phylogenetics research has been done to reconstruct these networks. Several methods for reconstructing those networks have been developed over the years. One of them is the softwired parsimony score and another, a fairly new method, is the parental parsimony method. In theory this is a better and more accurate method, but before this thesis, it was not tested yet. In this thesis, we compare the softwired and the parental parsimony method in practice by implementing the methods and calculating their parsimony scores. The results will show that the parental parsimony is a better and more accurate method in practice as well.

Table of Contents

Introduction

On the Origin of Species, published on 24 November 1859, is a scientific piece of literature that played a hugh role in the evolutionary biology. They even say that Darwin's book was the foundation of evolutionary biology. Soon after the publication of this book, the rumors came. People were wandering whether this book implies that we, humans, are the descendants of Apes. Some people thought, and some people still think, that gorillas, chimpanzees or orang-utans are our ancestors. But this is not true. Not a single ape or monkey that is alive today, is an ancestor of the human race. However, you could consider us cousins.

We share the same ancestor with some apes and we are close relatives, according to DNA research. The study that does this research is called phylogenetics. Phylogenetics is the study of the evolutionary history of organisms and the relationships among them. In this field, in this study, biologist are trying to reconstruct the tree of life: this is a tree or a graph (in the combinatorial sense of the word) that shows all the animal species or taxa and the relations between them. Knowing how these trees work and what they look like, is very important knowlegde, when you try to understand the processes of molecular evolution.

Phylogenetics studies the evolution of species from common ancestors and it shows its results in a graph. But how do we evaluate these evolutionary changes? Most of those changes, we have not seen ourselves. We only know which species live now and what their DNA looks like. This thesis is about these phylogenetic networks and how we can reconstruct these networks. When given a particular network and given the DNA sequences of the species that live now, we want to assign DNA sequences to the internal nodes of this network. In this way, we can determine what kind of species the ancestors of these species are. There are several mathematical methods to do this, but in this thesis a method that is called the *Maximum Parental Parsimony* will be discussed.

This method calculates the network that describes the evolutionary changes the best, according to the parsimony principle. This principle basically says that the less evolutionary changes has occurred, the more it is likely that the calculated network is the correct network. The parental parsimony also calculates the (Parental) *Parsimony Score* for a network, which tells us how many evolutionary changes has occurred. This score can be used to compare different parsimony based methods.

This thesis is about writing and implementing an algorithm for this parental parsimony and calculating the network that describes the evolutionary changes the best. The parsimony score of multiple networks will be calculated for this method, which will give an indication whether this method is better than another parsimony based method that is currently used, the *softwired parsimony*. That is the main issue in this thesis: is the parental parsimony a better method, than the softwired parsimony? Before a more formal description of the problem that is solved in this thesis will be given, some terms and theories that are important for the understanding of this problem will be discussed.

Something else the reader should know about this thesis, is that before writing this thesis, literature study has been done. This thesis is heavily based on the following literature:

- Cplex | ampl. http://ampl.com/products/solvers/solvers-we-sell/cplex/?gclid= Cj0KEQjw4cLKBRCZmNTvyovvj-4BEiQAl$_sgQuEihHt - 5Pj3LQFR9$ $t8N0_kQ8qUnwuHBXjc_2exjRvEaAklF8P8HAQ.(Accessedon$06/29/2017(CPL)
- Mpnet: Maximum parsimony on networks.http://homepages.cwi.nl/ iersel/MPNet/. (Accessed on 06/29/2017 (MPN)
- Diestel, R. (2000).Graphentheory. Springer. (3)
- Fischer, M., Van Iersel, L., Kelk, S., and Scornavacca, C. (2015). On computing the maxi-mum parsimony score of a phylogenetic network (4)
- Papadimitriou, C. H. and Steiglitz, K. (1982).Combinatorial optimization: algorithms andcomplexity. Courier Corporation (5)
- Van Iersel, L.Parental Parsimony: a new definition of parsimony for phylogenetic networks. (Van Iersel et al.)

Preliminairy

4-1 Phylogenetic Networks

Phylogenetics is the study of the evolutionairy history of organisms and the relationships among them. A *phylogenetic network* is therefore a network that graphically displays these relationships. For the display of these relationships, mathematical graphs are used, where the nodes or vertices of the graph correspond with taxa and the edges display the relationships between these taxa.

A *graph*, in the combinatorial sense of the word, consists points, which are called *nodes* or *vertices* and lines between these points, which are called *edges.* These nodes and edges can be labeled. A *directed* network or graph is a graph where every edge has a starting point and an end point. With this in mind, for every node, we can define a *indegree* and *outdegree*: this is the number of edges coming in and out of a node respectively. When there is not a node in a graph with a indegree of 2 or more, we speak of a (phylogenetic) *tree.* When the maximum in and outdegree of a node is 2 and the the total degree of every node is 3, then the graph is *binary.* A phylogenetic network or graph also has *leaves* and *roots.* Leaves are the nodes with an outdegree of zero and the roots are the nodes with an indegree of zero. We call a phylogenetic network *rooted* if there is exactly one node with an indegree of zero. Phylogenetic networks are also connected, which means that there is an undirected path from every node to every other node. Or in other words, there is a directed path from the root to every node.

In this thesis only binary, directed *acyclic* graphs will be discussed. Acyclic means that directed cycles are not permitted. A cycle is a number of vertices that are connected in a closed chain. To shorten notation, we will use the following notation for networks: $N = (V, E)$, where N represents the network, $V(N)$ is the set of nodes of N and $E(N)$ is the set of edges of N. The set of leaves will be denoted as $L(N)$. When we speak of a rooted phylogenetic network, the root of the network is denoted as ρ_N.

Now consider the set of taxa (species) X, then a *phylogenetic network on X* is a directed, acyclic graph of which every leaf is labeled by exactly one element from X. An example of such a network is given in figure 4-1. Note that in this figure and in the rest of the figures in this thesis, edges are not drawn as directed, to make the figures more clear, but they are still considered as directed away from the root. Considering a rooted, binary phylogenetic network $N = (V, E)$, the *reticulation nodes* of N are exactly the nodes of V with an

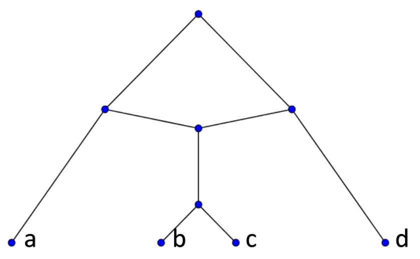

Figure 4-1: A network N_1 on $X = \{a, b, c, d\}$

indegree of 2. Now the conclusion can be drawn that N is a (phylogenetic) *tree* if N has no reticulation nodes, otherwise we speak of a *phylogenetic network*.

When considering two vertices $u, v \in V$, we could say that u is a *parent* of v and v a *child* of u, when there is a directed edge (u, v) from u to v in $E(N)$. To abbreviate, we use the notation $par(v)$ to denote te set of parents of v in N. Just as in biology, the terms *ancestor* and *descendant* can be used as well. We let u be ancestor of v and v a descendant of u when there is a directed path from u to v. Because there is always a path from a node to itself, we say that v is a *trivial* ancestor and a trivial descendant of itself.

4-2 Parsimony

More information is now know about phylogenetic networks, so the parsimony-based methods for the reconstruction of phylogenetic networks and finding the parsimony scores of these networks can be discussed. The problem these methods solve is the following: which species are represented by the internal nodes of the phylogenetic networks? When you know what kind of DNA is represented by the nodes, you know the kind of specie the internal node represents. With the parsimony based methods, you basically look at the DNA sequences of these species that are known and then you try to assign DNA sequences to the internal nodes of the network, in a way that should be correct according to the parsimony principle.

The concept of *maximum parsimony*, this principle, is very useful in the reconstructing of evolutionary networks. This principle is used in multiple disciplines of science and it basically says that when you have two possible answers or explanations that fits the evidence of a problem, then the simplest one is probably the correct one.

When reconstructing an evolutionary network, the primary assumption that is made, is that the character changes in the DNA sequences do not occur often. So in our case, the most simple explanation that explains the data the best, is the one where the phylogenetic network has the least evolutionary changes for every character in the DNA sequences. Notice that with *maximum* parsimony, the network has a *minimum* amount of changes.

When using the parsimony method on the given data - a network with leaves labeled with an alignment of characters, each character of the sequence will be evaluated independently. Therefore, a parsimony score $PS(c_j|T)$ for each character c_j of a phylogenetic tree T will be calculated independently. Then you can calculate the score of an entire DNA sequence of length n by summing over the different characters. The score $PS(c_j|T)$ can be calculated after the assigning of the internal nodes. $PS(c_j|T)$ is the sum over all edges e of T of $PS(c_j|e)$, where $PS(c_j|e)$ is the cost of the change per state for character c_j over edge e. This means that if at one end of an egde $e = uv$ the node u has the states A, C and node v has the states A, T, then one substitution had to take place. Now if you multiply this one substitution by the cost of a change, you get $PS(c_j|e)$. Since only the DNA sequences of the leaves are known, we need to assign the internal nodes sequences in such a way that the combination of them minimizes $PS(c_j|T)$. It is possible to do this for the entire sequence, but in this thesis, only a single character will be considered at the time. After the score per character is calculated, you can easily calculate the score for the entire alignment, by adding the scores of each character together.

Given a set of taxa X and $p \in \mathbb{N}$, a *p-state character* α on X is a function from X to $\{1, ..., p\}$. α is binary if $p = 2$. This means that α is a function that labels the leafs - the nodes labeled with taxa - with a states. These states corresponds with the possible characters for a position in a DNA alignment. For example, a DNA alignment $ABCAC$ corresponds with the states

12312, where 1 is the state of the character on first position in the alignment. Now let α be such a function and let T be a rooted phylogenetic tree on X. Then an *extension* τ of α is a p-state character on $V(T)$ if $\forall x \in X$ it holds that $\tau(x) = \alpha(x)$. This means that the extension τ is a function that labels all the nodes of T with states, with the constraint that α and τ give the leaves the same states. Now consider such a τ and an edge $uv \in E(T)$, then the *change* $c_\tau(uv)$ on uv with respect to τ is 0 if $\tau(u) = \tau(v)$ and considered to be 1 otherwise. Now given a tree T on X and a p-state character α on X, when we take the minimum over all the extenstions τ of α, the *parsimony score* on a tree T and α is defined as

$$PS(T,\alpha) = \min_{\tau} \sum_{uv \in E(T)} c_\tau(uv). \tag{4-1}$$

4-3 Hardwired and Softwired Parsimony

Multiple combinatorial methods for reconstructing phylogenetic networks have been developed over the last couple of years. Some of those methods are parsimony based. Three of those are the exact methods that will be discussed in this thesis. When one moves the scope from phylogenetic trees to phylogenetic networks, there are two definitions of maximum parsimony methods are already known and there is one method that that is still being developed.

The first method is the *hardwired* parsimony method. The goal for this method is to assign states to the internal nodes of the network, in such a way that the cost is minimized. The cost is the number of edges that has a particular state at one end a different state at the other. Ofcourse, we can define a parsimony score for this method. Given a network N on X and a p-state character α on X, the *hardwired parsimony score* of N and α can be defined as follows:

$$PS_{hw}(N,\alpha) = \min_{\tau} \sum_{uv \in E(T)} c_\tau(uv) \tag{4-2}$$

The minimum is taken over all the extensions τ of α to $V(N)$. Note that the hardwired parsimony score is exactly the same as the parsimony score for trees, as defined above.

When looking at phylogenetic trees, this method is well suited to reconstruct the tree en show how the species have evolved. When we look at evolutionary histories in which a phenomenon called reticulate evolution has occurred, this method no longer is the optimal one. Reticulate evolution are events where a specie or a taxon inherits DNA from more than one ancestor. A reticulation node represents such an event in a phylogenetic network.

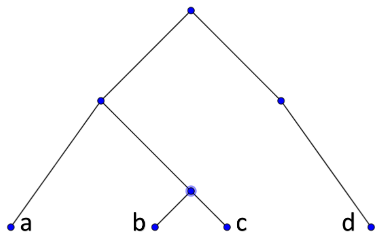

Figure 4-2: A tree displayed by the network N_1 on $X = \{a, b, c, d\}$.

For a reticulation node, this method could count two changes, where it should count only one. For example, consider a reticulation node r with parents u and v. Let r have the character state 1, while u and v have character states 0. Since edge (u, r) has state 1 at one and and state 0 at the other end, a change has occurred. Therefore, the hardwired parsimony score counts a change over this edge. For the edge (v, r) the same holds. So in this example, the parsimony score counts two changes. However, looking from a phylogenetic point of view, it could be possible that the taxa that is represented by r had inherited it's state character from only one parent, and therefore only the character state of one parent had to change. Therefore, the parsimony score, which represents the amount of evolutionary changes had to take place, is too high. Therefore this method is not the optimal one when looking at a phylogenetic network.

In this thesis, as already explained above, the hypothesis of site independence is made. This means that it is assumed that every different character of a DNA sequence independently evolves through the network. The evolution of a single character is still best described by a tree, but for the entire alignment a phylogenetic network describes the evolution better, because of the presence reticulate events. That is why for the second method, which is called the *softwired parsimony*, the parsimony score of a character on a network is defined as the (hardwired) parsimony score of the best tree *displayed* by the network. In other words: the softwired parsimony score is the lowest parsimony score of any tree displayed by the network. A *displayed tree* of a network N is a tree T that we get if we take a subgraph of N and then suppress all the nodes of in- and out-degree 1. That is, for a node v with an in- and out-degree of 1 in the subgraph of N, we take the parent u and the child w and then write an edge in T from u to w. We choose the notation $T(N)$ to abbreviate the set of all phylogenetic trees on X that are displayed by N. An example of a displayed tree can be found in figure 4-2. There you see a displayed of network N_1, given in figure 4-1 We can define the softwired parsimony score of a network N on a set of taxa X and a p-state character α on X as the minimum parsimony score of any tree on X displayed by N:

$$PS_{sw}(N,\alpha) = \min_{T\in T(N)} \min_{\tau} \sum_{uv\in E(T)} c_{\tau}(uv). \tag{4-3}$$

In figure 4-3 you can see an example of a network where the softwired parsimony is applied.

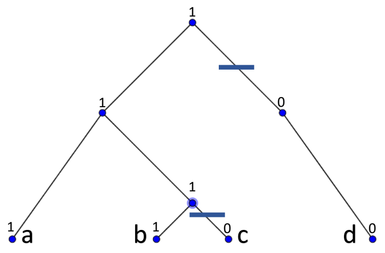

Figure 4-3: Optimal extensions for a tree displayed by N_1 for the softwired parsimony score

The softwired parsimony should be a better method than the hardwired parsimony, but it has a problem as well. Although this method no longer counts two changes for reticulation node, biological speaking this method still gives a problem. A reticulation node can only have one state and it has to choose between the parents of this node. However, biological speaking this is not always what is happening. For example, let us consider the reticulation node r which has two parents u and v, with two different states 0 and 1 respectively. It could be possible that the specie that is represented by this reticulation node, is a specie with both genes in it's population. When looking at the descendants of r, it could be possible that two different descendants have the same states as u and v. From a biological point of view, a evolutionary change has not occurred in this situation. However, the softwired parsimony score will assign to r one of the states of his parents. r has two descendants with different states, so the softwired parsimony will count a change somewhere on one of the path's from r to it's descendants. In this case, the parsimony score of the softwired parsimony is too high. That is why a new variant of parsimony for phylogenetic networks is developed. A variant that does not have the problems as described above. This method is called the parental parsimony.

4-4 Parental Parsimony

The *parental parsimony score* of a phylogenetic network will be explained in this section. This is the parsimony score of a character on a network of the best *parental tree* of the network. A parental tree can be defined intuitively as follows. A parental tree is any tree that can be drawn inside a network, where the nodes of the tree labeled with a taxon coincide with nodes of the network. Note that when drawing, it is allowed to draw multiple branches of the tree through the same network branch. This is not the case with displayed trees, because for a displayed tree you need to take a subgraph of N. A parental tree is not a always subgraph. Biologically speaking, the multiple branches of the drawn tree can represent different versions a gene present in a population of a specie. This way, the problem described for the softwired parsimony is now solved.

A parental tree can be described more specific, but still informal, as follows. A parental tree of N is phylogenetic tree T on X, with two characteristics. One property is that you can map all the nodes of T onto nodes of N. The other property is that you can map all the directed edges in T onto directed paths in N, where the corresponding nodes in the path in N with an in- and out-degree of 1, can be suppressed to obtain T. These properties should be in such a way that the leaves of T labeled with an $x \in X$ are mapped onto the leaves of N labeled with an x, for each $x \in X$. This projection is not injective, therefore different nodes of T can be mapped onto the same node in N.

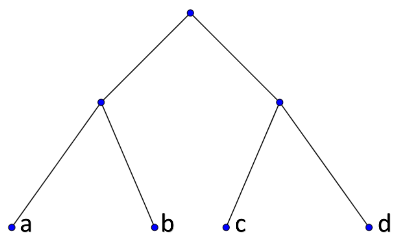

Figure 4-4: A parental tree of the network N_1 on $X = \{a, b, c, d\}$.

To define the parental tree of a phylogenetic network formally, it is first needed define the *multi-labeled* tree $U^*(N)$. A multi-labeld tree is a tree in which a taxon in X can label more than one leaf. The tree U^* is a tree that is derived from N, such that the nodes of $U^*(N)$ are the directed paths in N, starting from ρ_N. For each pair of paths π, π' in N, there is an edge in $U^*(N)$ from π to π' if and only if $\pi' = \pi e$ for some edge in N. Futhermore, each path in N that starts at the root ρ_N, ends at $x \in X$ and is represented by a node in $U^*(N)$, is represented by a node in $U^*(N)$ labeled by x. An example can be found in figure 4-5, where you see the multi-labeled tree $U^*(N_1)$ of network N_1. Now the formal definition of a *parental tree* can be given. A parental tree of N is a phylogenetic tree T on X that is displayed by $U^*(N)$. We abbreviate the set of all trees on X that are parental trees of N with $PT(N)$.

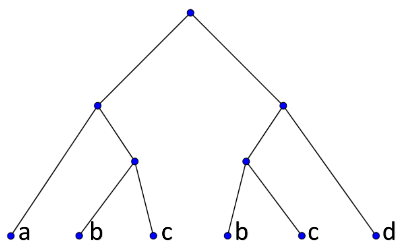

Figure 4-5: The tree $U^*(N_1)$ used in the formal definition of parental trees

With the all the described definitions, the *parental parsimony score* can now be given. The parental parsimony score of a network N and a character α is the hardwired parsimony score of the best parental tree of N. That is, the parental tree that gives the lowest parsimony score. So the parental parsimony score is

$$PS_{pt}(N, \alpha) = \min_{T \in PT(N)} \min_{\tau} \sum_{uv \in E(T)} c_\tau(uv). \tag{4-4}$$

An example of optimal extension for a parental tree is of the given network N_1 in figure 4-1 is given in figure 4-6.

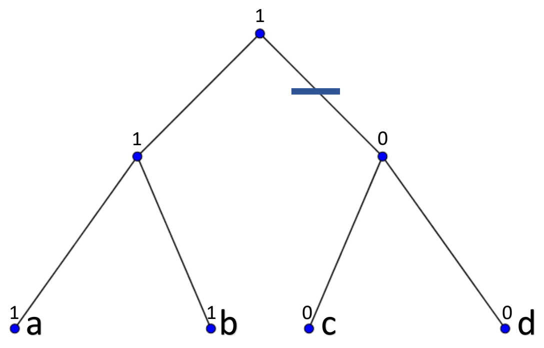

Figure 4-6: Optimal extension for a parental tree of N_1 for the parental parsimony score

Calculating the parental parsimony score of a network N on X and a p-state character α on X is also called the *Parental Parsimony Problem* (PPP).

4-5 Lineage Functions

As described in section 4-4 and in equation 4-4, the parental parsimony score of N and α is the minimum (hardwired) parsimony score of any parental tree of N. One might think that it is needed to calculate all the parental trees of a network, in order to find the parental tree that will give the minimum score. This would be very cumbersome and fortunately, this is not needed. To determine the parental parsimony score, we can characterize parental parsimony with *lineage functions*. A lineage function does not give a the parental tree of the network, but given a lineage function it is easy to find the corresponding parental tree. It does characterize the parsimony score of a tree very well. A lineage function basically does the following. It assigns a set of states to every node in a network. This set of states of a node corresponds to all the branches of parental trees that walk through each node and it tells which states the nodes of these branches have. A formal definition of a lineage function can be given as follows.

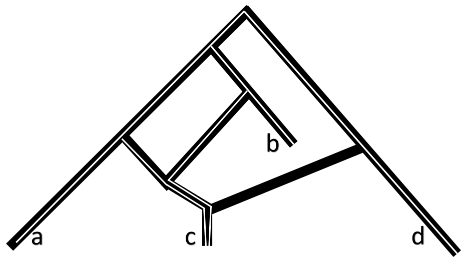

Figure 4-7: A network N_2 on $X = \{a, b, c, d\}$, with a parental tree drawn inside.

Definition 1. *Given a rooted phylogenetic network N on X and an integer p, a (p-width) lineage function on N is the following:*

$$f : V(N) \mapsto \wp(\{1, ..., p\}) \tag{4-5}$$

where $\wp(U)$ refers to the power set of U, which is the set of all subsets of U, including the empty set and U itself.
If $|f(\rho_N)| = 1$, then f is a rooted *lineage function.*
Given a p-state character α on X, if $(\forall x \in X) f(x) = \{\alpha(x)\}$ holds, then f is called a α-consistent *lineage function.*

To determine the parental parsimony score of a network, we use the *weight* of all the possible lineage functions. This weight $w(f)$ of a lineage function f is actually the parsimony score of the parental tree corresponding to f. So if one wants to determine the parental parsimony score of a network N and a p-state character, one should determine the lineage function with the minimum weight. The definition of the weight of a lineage function is a bit similar to the definition of a change in the hardwired parsimony. The weight of a lineage function f on a network N is calculated for every vertex $v \in V(N)$ separately and then the weight of f is the sum over the vertices of N. To determine the weight of a vertex $v \in V$, consider the set of states $f(v)$ of v assigned by lineage function f. Also consider the parents u of v and their assigned set of states $f(u)$. Then the sets $f(u)$ are compared to the set $f(v)$ and the weight of v is the number of states that are assigned to v, but are not assigned to his parents. This is because this is the number of states of the branches through the nodes u that needed to change, in order to let v have the states that f assigned to it. A more formal and a more accurate definition of the weight $w(f)$ of a lineage function f is the following:

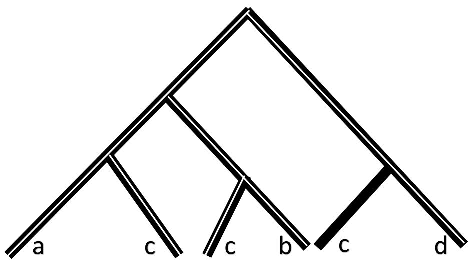

Figure 4-8: The tree $U^*(N_2)$ on $X = \{a, b, c, d\}$, with the same parental tree drawn inside.

Definition 1. *Let f be a lineage function on N. Given a node v of N, the weight of v with respect to f, denoted $w_f(v)$, is defined as*

$$w_f(v) = \begin{cases} 0, & \textit{if } v = \rho_N \\ \text{inf}, & \textit{if } v \neq \rho_N \text{ and } |f(v)| > \sum_{u \in par(v)} |f(u)| \\ |f(v) \setminus \cup_{u \in par(v)} f(u)|, & \text{otherwise} \end{cases} \tag{4-6}$$

The total weight *of f is defined as follows:*

$$w(f) = \sum_{v \in V(N)} w_f(v) \tag{4-7}$$

The first part of the definition of $w_f(v)$ says that the weight of the root ρ_N is zero. This is trivial, because ρ_N is the first vertex of the network and therefore there is not a parent with a character state that can change in order to give ρ_N his state. The second part of the definition says that there is no parental tree corresponding to the given lineage function. This is only the case when there are more branches in a child node required than there are branches traveling trough the parent node(s) to cover this number. For example, consider vertex v in a network with a single parent u. Then a parental tree with one branch going through u and two branches going through v does not exist.

To show that using the parental parsimony and lineages functions do not give the same problem as described for the softwired parsimony in section 4-3, consider the following lineage function on a network. Let us say that a node v has the assigned set {0,1} and let his parents u_1 and u_2 have the assigned sets {0} and {1} respectively. This corresponds with the fact that there are two branches of a parental tree going to v, one from u_1 and one from u_2. As described above, the cost of this for this node is 0, because there are no states assigned to v that are not assigned to u. From a biological point of view, this should be correct, because v could inherit character states from both parents.

As described above, to determine the parental parsimony score of a network, the weight of all the possible lineage functions are considered, and then the smallest one is the parental parsimony score. There are some restrictions. The lineage function needs to be rooted and α-consistent. It needs to be rooted, which means that $|f(\rho_N)| = 1$, because for every parental tree that is displayed by the network, there travels only one branch through the root. So in a network, the lineage function should assign only one state to the root. The lineage function needs to be α-consistent, which means that $(\forall x \in X) f(x) = \{\alpha(x)\}$, because the leaves of the network are already labeled by X. So the lineage functions that should be considered are the ones that correspond to the network and maps the leaves onto the states they should have according to the given the p-state character α. Everything that is said in this section about lineage functions and their relation to parental parsimony, can be summarized in the following equation.

Definition 1. *For any binary network N on X and p-state character α on X,*

$$PS_{pt}(N, \alpha) = \textit{\{w(f): f is a rooted } \alpha\textit{-consistent lineage function on N \}} \tag{4-8}$$

Problem Description

A rooted phylogenetic network is a is a directed acyclic graph of which every leaf is labeled by exactly one element from a set of taxa X, and with only one node that has a in-degree of zero. These networks are used to describe the evolutionary history of organisms and the relationships between them. We want to reconstruct these networks and we want to find out what kind of taxa the internal nodes of such a network should have. To reconstruct such a network, the DNA alignments of the taxa are being analyzed and then one tries to find out what the DNA alignments of the internal nodes of the network should look like. This way we get more knowledge about the process of molecular evolution. To reconstruct a phylogenetic network, there are several methods that can be used. Three the parsimony-based methods will be: the hardwired parsimony, the softwired and the parental parsimony. These methods are based on the principle of parsimony: which says that when a phylogenetic network is reconstructed, there should be as little as possible (evolutionary) changes. For each of those methods, the changes that occur after the reconstruction are counted and they are called the parsimony scores. A maximum parsimony score, is a score as small as possible. Because all three methods are parsimony based, they can be compared. This is done by comparing the parsimony scores of the methods and this way we can find out whether one is better than the other. From a biological point of view, the softwired parsimony is more relevant than the hardwired parsimony, and it certainly delivers a lower parsimony score. So we want to compare the softwired with the parental parsimony. Therefore the main question in this thesis is the following: Is the parental parsimony a better and more accurate method for reconstructing phylogenetic networks than the softwired parsimony?

To solve this question, not only theoretical analysed need to be made, but it also needs to be tested. So first of all, the Parental Parsimony Problem will be solved for an amount of networks. The PPP has as input a network N on X and a p-state character α on X. The output is the parental parsimony score $PS_{pt}(N, \alpha)$. In order to solve this problem, the parental parsimony is characterized with lineage functions. The reason this is done is because the parental parsimony score is the minimum of the weights of all the rooted α-consistent lineage functions on N, which is defined in section 4-5. Fortunately, it is not needed to calculate all the lineage functions in order to get the weight of all the lineage functions on a network. First, the problem of finding the lineage function with the minimum weight, is translated into an ILP formulation, described in section 6-1. To solve this ILP, it was implemented in Java, so that it is possible to use an ILP solver. The software that was used, was the ILP solver CPLEX. This software package assigns the optimal values to the variables of the ILP problem. More information about CPLEX can be found in section 6-2. Then using different methods in Java, this is translated back to our solution: finding the parental parsimony score $PS_{pt}(N, \alpha)$ for every input N and α and reconstructing the network N by assigning states to the internal nodes of N. How this is done, is described in section 6-3. Then the parental parsimony scores are calculated with the model that is written. In the model, the methods to calculate the softwired parsimony are also implemented. Then in section 7 the scores $PS_{pt}(N, \alpha)$ are compared to the scores of the softwired parsimony to see whether the parental parsimony really is a better method.

Methods

To answer the main question of our thesis, the Parental Parsimony Problem and the Softwired Parsimony Problem need to be solved and compared to each other. Because the softwired parsimony problem has already been solved and there is already an algorithm to find the softwired parsimony score for a given network and p-state character, the focus on this thesis lies on solving the PPP.

To solve the PPP - finding the Parental Parsimony Score of a network and a character α - it needs to be translated into mathematics. To translate the problem into mathematics, it needs to be written as a mathematical model that we are able to solve. One way to do this is writing the problem as an *Integer Linear Programming* formulation. More information about the formulation of the ILP is written in 6-1. After formulating the ILP, the ILP will be implemented, so that the software CPLEX can be used to solve the ILP. Section 6-2 contains more information about CPLEX. After CPLEX solved the model, the solution needs to be translated into something that we can read and then the parsimony score and the lineage function can be deduced from this solution. The idea's behind the model that solves the PPP can be found in section 6-3.

In the section 7 the results of calculating the parental parsimony and the softwired parsimony of 60 different networks are presented. The softwired and parental parsimony score will be calculated for these network, several times, with different values assigned to the leaves of the networks. Finally, these scores are compared to each other.

6-1 Integer Linear Programming Formulation

An ILP formulation of the problem contains an objective function - the function we want to optimize - and the constraints of the problem, written in formula's. With an ILP formulation, all the formulas are linear and all the variables are restricted to be integers. This type of formulation has been chosen because all the restrictions of the problem that is solved are linear. The main reasons for using integer variables in modeling the PPP as a linear program are the following: Some of the variables represent decisions and should therefore only take on the value 0 or 1. These special kind of integer values are called binary variables. The other variables represent quantities that can only be an integer. The parsimony score will be computed and this score is always an integer, because half changes do not exist in our model (and neither in real DNA sequences). Another reason for choosing an ILP formulation to solve the PPP is that although Integer Programming is NP-hard, ILP's can be solved quickly by ILP solvers, such as the software CPLEX that is used. More information about CPLEX can be found in section 6-2 about CPLEX.

The problem of solving the PPP is transformed into solving the problem of finding the minimum weight lineage function, as described in section 4-5. In order to translate the PPP into an ILP, we need to translate the definition of the lineage function and its weight into an objective function, restrictions and variables. Then the objective function will be minimizing the weight.

First of all, the input of the ILP is the same as the input of the PPP. The Input is a rooted phylogenetic network N, with an edge set E, a node set V and a p-state character $\alpha(v)$. For a leaf v, $\alpha(v)$ is the character-state that is given for the leave. ρ_N represents the the root of N. So for the problem the following is given:

- Node set V;
- Edge set E;
- ρ_N: the root of the network;
- p-state character $\alpha(v)$;
- Set states $P = \{1, ..., p\}$;

Consider the lineage functions f, that assigns a set of states to each node $v \in V$. To translate this into the ILP, the binary variable $x_{v,s}$ is used. This variable $x_{v,s}$ represents whether v has state $s \in P$. We denote this input by

$$x_{v,s} = \begin{cases} 1, & \text{if } v \text{ has state } s \\ 0, & \text{otherwise} \end{cases}$$

The lineage functions that are considered need to be rooted and α-consistent. So we want the lineage functions to assign exactly one state to the root ρ_N. This can be denoted by

$$\sum_{s \in P} x_{\rho_N,s} = 1, \quad \text{for the root } \rho_N$$

An α-consistent lineage function means that the lineage function assigns to the leaves of N exactly the states that α has assigned them. We can formulate these requirements in the following two formulas:

$$\begin{aligned} x_{v,s} &= 0 \quad \text{for all } v \in X \text{ , } s \neq \alpha(v) \\ x_{v,\alpha(v)} &= 1 \quad \text{for all } v \in X \end{aligned}$$

We want to translate the weight $w(f)$ of lineage function f into objective function of the ILP. Because $w(f)$ of f is the sum over the weight $w_f(v)$ of nodes v with respect to f, we will translate weights of the nodes to the variables c_v. These variables represent the *cost* of v. In other words, they represent the amount of character states that changed over an edge from parent u to child v . The objective function of the ILP will be the sum of all these variables. So the objective function of our ILP will be the following:

$$\text{minimize} \sum_{v \in V} c_v \tag{6-1}$$

More constraints are needed to translate the weight of lineage functions into the ILP. To help us do that, recall the definition of $w_f(v)$ in definition 1 in section 4-5. It says that if $v \neq \rho_N$ and if $|f(v)| \leq \sum_{u \in par(v)} |f(u)|$, then $w_f(v)$ is $|f(v) \setminus \cup_{u \in par(v)} f(u)|$. A new variable will be introduced. This new variable is $y_{v,s}$ and it indicates whether for node v, the state s is in the set $\{f(v) \setminus \cup_{u \in par(v)} f(u)\}$, where $f(v)$ is the set of states assigned by the lineage function to node v and $f(u)$ is the set of states assigned by the lineage function to node u, a parent of v. This means that if s is in this set, then v has a state s that neither of his parents u have. This way, we check per state s whether there has occurred a change over one of the egdes $(u, v) \in E$. Translated into an constraint, this will give the following equation:

$$y_{v,s} \geq x_{v,s} - \sum_{u \text{ s.t.}(u,v) \in E} x_{u,s} \quad \text{for all } v \in V, s \in P \tag{6-2}$$

Notice that constraint 6-2 actually gives the weight of a node v per state s. So the entire weight of v will be the sum over all the states. The sum over all the states of $y_{v,s}$ will be the cost of v and corresponds to $w_f(v)$ of v. This will give us the following constraint:

$$c_v \geq \sum_{s \in P} y_{v,s} \quad \text{for all } v \in V \tag{6-3}$$

Also notice that constraint 6-2 allows $y_{v,s}$ to be negative. This could happen when v does not have s, but one of his parents u does have s. However, we do not want to $y_{v,s}$ to be negative, because we the weight of a node can neither be negative. So we need the following constraint:

$$y_{v,s} \geq 0 \quad \text{for all } v \in V\ ,\ s \in P \tag{6-4}$$

We need more constraints to fully characterize $w_f(v)$ in our ILP. One of the properties of $w_f(v)$ is that the root of the network gives no weight. This requirement is given by the following constraint:

$$y_{\rho_N,s} \geq 0 \quad \text{for all } s \in P \tag{6-5}$$

One last constraint is needed to fully translate $w_f(v)$ into our ILP formulation. The given definition of $w_f(v)$ says that if $v \neq \rho_N$ and $|f(v) \leq \sum_{u \in par(v)} |f(u)|$ for node $v \in V$ holds, then $w_f(v)$ is infinity. In other words: the considered lineage functions are restricted to assign to v a number of states that is less than or equal to the sum of the amount of states that are assigned to the parents of v. Formulated into a constraint, this will give the following:

$$\sum_{s \in P} x_{v,s} \leq \sum_{u \text{ s.t.}(u,v) \in E} \sum_{s \in P} x_{u,s} \quad \text{for all } v \in V \tag{6-6}$$

To see the correctness of this constraint, consider the following. Because $x_{v,s}$ is equal to 1 when v has state, the amount of states that v has is $\sum_{s \in P} x_{v,s}$. Also, u is a parent of v when there is an edge (u, v) from u to v in E. As explained above, solving the ILP will be done by CPLEX. So the ILP that will be fed to CPLEX, will the summary of everything explained in this section. This will give the ILP formulation and the following input for computing the parental parsimony score of a phylogenetic network $N = (V, E)$:

- Node set V;
- Edge set E;
- ρ_N: the root of the network;
- p-state character $\alpha(v)$. For leaf v, parameter $\alpha(v)$ is the given character state at v;
- Set of states $P = \{1, ..., p\}$;
- Binary variable $x_{v,s}$. This variable indicates whether node v has state s;
- Binary variable $y_{v,s}$. This variable indicates whether for node v, state s is in the set $\{f(v) \setminus \cup_{u \in par(v)} f(u)\}$;
- Variable c_v. This is the cost or the weight of node v;

$$\begin{array}{lll}
\underset{v}{\text{minimize}} & \sum_{v\in V} c_v & \\
\text{Subject to} & \sum_{s\in P} x_{v,s} \leq \sum_{s\in P} \sum_{u \text{ s.t. } (u,v)\in E} x_{u,s} & \text{for all } v \in V\,, \\
& x_{v,s} = 0 & \text{for all } v \in X\,,\ s \neq \alpha(v)\,, \\
& y_{v,s} \geq 0 & \text{for all } v \in V\,,\ s \in P\,, \\
& y_{v,s} \geq x_{v,s} - \sum_{u \text{ s.t. } (u,v)\in E} x_{u,s} & \text{for all } v \in V, s \in P\,, \\
& x_{v,\alpha(v)} = 1 & \text{for all } v \in X\,, \\
& \sum_{s\in P} x_{\rho_N,s} = 1 & \text{for the root } \rho_N\,, \\
& y_{\rho_N,s} = 0 & \text{for all } s \in P\,, \\
& c_v \geq \sum_{s\in P} y_{v,s} & \text{for all } v \in V\,, \\
& x_{v,s} \in \{0,1\}\,. &
\end{array}$$

6-2 CPLEX

To solve the mathematical models that are formulated in section 6-1 the software IBM ILOG CPLEX Optimizer (often referred to simply as CPLEX) from company IBM was used. CPLEX solves large integer programming problems and large linear programming problems, but it also solves convex and non-convex quadratic programming problems and convex quadratically constrained problems. There are different algorithms available in CPLEX. To solve linear programming problems such as the PPP, either primal or dual variants of the simplex method or the barrier interior point method are used. It also uses advanced branch-and-bound, feasibility heuristics and cut generators (CPL).

In this thesis, CPLEX has been made use of in the following way. First we have a network N written in a text file and we want to calculate the softwired and the parental parsimony score score of this network. Then the software development platform Netbeans, written in Java, is used to give CPLEX the correct input. The correct input is also a text file with the specific ILP objective function and constraints for N. How such a specific ILP text file for a network looks, is explained in section 6-3. Then CPLEX will solve this ILP and it will give a file back with the solution. After that, various methods read the solution, determine the parsimony score and give the lineage function that belongs to this solution.

6-3 Source Code OnlyParental

The fist step in solving the PPP for a given network N on X and a p-state character α on X, is formulating the problem as an ILP. This was done in section 6-1. The second step is writing a code in Java, so that this ILP can be implemented. Then this code makes a text file, with a specific ILP formulation for the input N and α. This file is made so that CPLEX is able to read this file and solve the formulated ILP. After CPLEX solved the ILP, it gives the solution and this solution needs to be interpret so that the solution is readable. From this readable solution, the parental parsimony score and the corresponding lineage function is deduced. The code that has been written to make these steps and solves the PPP for a given network N and p-state character α, is called *OnlyParental* and can be found in the Appendix B.

All of the described steps above have already been done for the softwired parsimony. There already exist an implemented algorithm to find the sofwired parsimony score for a given network and a p-state character. The Java code with this algorithm is called MPNet. Because there already exist such an implemented algorithm for the softwired parsimony and because of the fact that the softwired parsimony and the parental parsimony have much in common, the code OnlyParental has much in common with the code MPNet. In fact, before writing OnlyParental, MPNet was analyzed thoroughly. After that, MPNet was modified in order to get the code that is now called OnlyParental.

6-3-1 Main Ideas Behind the OnlyParental Algorithm

To save the reader of this thesis from exhausting reading, not the entire code of OnlyParental is given in this section. The ideas behind OnlyParental will be explained in here, but if the reader that still wants to see the entire code, Only Parental can be found in Appendix B.

To explain the main ideas behind OnlyParental and the main adjustments that are made to MPNet in order to get OnlyParental, we first need to realize what the main differences are between the parental parsimony and the softwired parsimony. There are two main differences that need to be taken into account when implementing the algorithm to find the maximum parental parsimony score. The first primary difference between the parental parsimony and the softwired parsimony, is the fact that nodes are allowed to have multiple states assigned to them when using the parental parsimony, in contrary to the softwired parsimony where a vertex can have only one character state. The other important difference between the parental and the softwired parsimony is the fact that the scores of both methods are calculated differently. For the softwired parsimony, the score is calculated per edge, while for the parental parsimony, the score is calculated per vertex.

Let us explain the steps that OnlyParental goes through, in order to get the parental parsimony score for a given network and a p-state character.

1. The input of the model is a text file with a network and a file with characters, which says with what the leaves are labeled. It is also possible to have as input only a text file with a network and let the model assign random character states to the leaves.

2. OnlyParental reads the network file. It calculates the number of taxa, the number of reticulations, the number of edges, the number of vertices and which vertex is connected to which vertex. It puts all these data in different objects, so that methods that calculate the parsimony score can be build with these objects.

3. It makes a text file and it puts in this file a specific ILP formulation from the given network as input for CPLEX. What is meant by a specific, is the following. It makes the constraint explicit per variable. It makes a String object, and adds the explicit constraints to this String object. For example, if the input is a network N with node set $V(N) = \{1, 2, 3\}$ and set of states $P = \{0, 1\}$, then the strings of the constraint $y_{v,s} \geq 0$, for all $v \in V$, s $\neq \alpha(v)$ will be written as:

    - $y_{1,0} \geq 0$
    - $y_{1,1} \geq 0$
    - $y_{2,0} \geq 0$
    - $y_{2,1} \geq 0$
    - $y_{3,0} \geq 0$
    - $y_{3,1} \geq 0$

Then it translates this String object into an text file that can be fed to CPLEX.

4. When the text file ILP.*txt* is finished, it feeds the text file to CPLEX.
5. Then it runs CPLEX, after which CPLEX will solve the ILP and puts the solution - the values of the variables - in an object.
6. The solution from CPLEX will be translated back to strings, strings that give the values of the variables of the ILP for the optimal solution. Then a method of OnlyParental reads these strings and assigns to every vertex object in the model the correct states.
7. Finally, with 2 methods the model determines the parental parsimony score. One method determines the parsimony directly from the values of the variables of the ILP, the other method gives the parsimony score by calculating the weight of the lineage function after all the states are assigned to the vertices. Two methods determine the parsimony score, one as a double check that the correct parsimony score has been calculated.

6-3-2 Differences and Similarities between MPNet and OnlyParental

Multiple methods in the model MPNet are modified in order to make a model that could calculate the parental parsimony score of a network. First of all, the *Network class* needed to change. This class, which is basically a blueprint for the object *Network()*, was modified because vertices are assigned a set of states in OnlyParental, instead of single states in MPNet. So objects within Network() that first stored a single state, store a set of states in OnlyParental. Objects that stored Strings with sets of states, store Strings with multi-sets of states in OnlyParental.

Fortunately, there were also many code lines that could be left the same. We will walk through every step to explain whether it needed to change or not.

The first two steps of the steps describes above, are exactly the same for MPNet and for OnlyParental.

Step 3 in OnlyParental differs from MPNet. This is the step where the ILP is implemented. This is not unexpected, because the parental parsimony and the softwired parsimony have a different ILP formulation. The way the ILP is implemented and fed to CPLEX is the same, though. Also the step where CPLEX solves the ILP and gives a solution back is the same.

Step 6 in OnlyParental differs from MPNet again. This is due to the fact that the vertex objects of the *Network()* object need to have a set of states instead of a single state. Therefore the method that assignes the states to the vertices is different and the object that keeps track of the states of every node is different.

Step 7 in OnlyParental completely differs from MPNet. This is due to the fact that the parsimony score of OnlyParental is calculated using the weight of the lineage function, while in MPNet the parsimony score is calculated by counting the edges that have a certain state at one end and a different state at the other. OnlyParental uses a method that is called *getScore()*, which is based on the weight of the lineage function. Per vertex v, it calculates the amount of states v has that neither of his parents u have. That will be the weight per vertex with respect to the corresponding lineage function.

Results

In this section we present the results we received after applying the method OnlyParental we formulated in section 6 on the given data. The script of this algoritm can be found in Appendix B [fixme]. As described in section 6, OnlyParental does the following with the given data: it analyses the given data, implements the formulated ILP for our problem, then uses the software package CPLEX to solve the problem for our input data and finally reads and gives us the solution of our problem: finding the maximum parental parsimony score.

7-1 Data Analysis

The data we received consists of a set of 60 different networks. When our method solves the PPP for the given network, it first analyses the network. We found out that there is a great variety in the networks of the given data. The smallest network contains 124 vertices, 139 edges and 16 reticulations, while the biggest network contains 808 vertices, 923 edges and 116 reticulations. For the rest of the networks, the amount of reticulations, vertices and edges lies somewhere between the mentioned numbers. As for the number of taxa of the networks, the following holds: ten networks consist of 50 taxa, ten networks consist of 100 taxa, eleven consist of 150 taxa, ten consist of 200 taxa, ten networks consist of 250 taxa and the last nine networks consist of 300 taxa.

7-2 Results

To get the results that are described in this section, the following was done. First of all, the networks of the given data were ordered with respect to the the number of taxa they have and then the networks were numbered, from 1 up to and including 60. The given data set contains networks, but the set does not contain any p-state characters $\alpha(v)$ that assigns states to the leaves. That is why we let the method OnlyParental assign states to the leaves, randomly. This was done with three different sizes of state set P: with two different character states, with three different character states and with four different character states. So for every network, the parental parsimony score and the softwired parsimony score were calculated with three different amounts of character states. Therefore, we ran our algorithm OnlylParental 180 times in total. All the results of these 180 experiments are given in the tables in Appendix A-1, Appendix A-2 and Appendix A-3. In this section a summary of the results is given.

7-2-1 $|P| = 2$

The outcome of the experiments where the size of the set of states P is 2, can be viewed in figure 7-1. Here you see a figure with per network, the corresponding value of the softwired parsimony score (in blue) and the parental parsimony score (in orange). Clearly can be seen, that the parental parsimony score is lower than the softwired parsimony score for all the networks except one, where they are equal. This is confirmed by the tables in the Appendix A-1.

When we take all the softwired scores of the 60 networks together, we get a score of 2744. That is an average softwired parsimony score of 45,7 per network. For the parental parsimony scores, all the scores together add up to 1619 and an average of 27,0. This is nearly the half of the average of the softwired parsimony scores. Also can be deduced that the difference between the scores of the methods is 1125, which comes down to a difference of 18,75 per network. To see all the possible scores, check the tables in Appendix A-1.

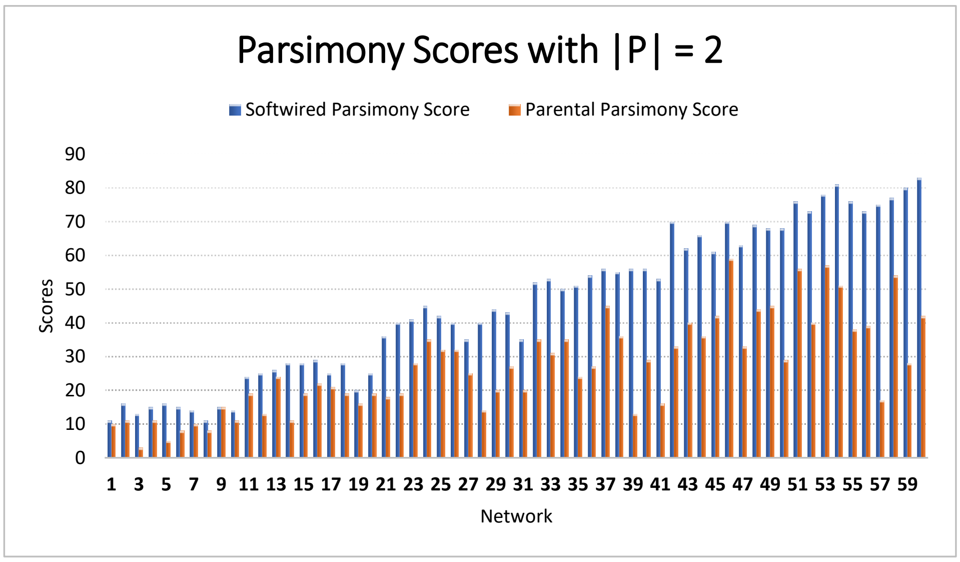

Figure 7-1: Parsimony Scores with $|P| = 2$, with the number of the networks on the x-axis and the corresponding parsimony scores on the y-axis.

7-2-2 $|P| = 3$

For all the experiments where the size of the set of states P is 3 the following holds. Figure 7-2 shows the softwired parsimony score in blue and the parental parsimony score in orange. Also here clearly can be seen that for every network the softwired parsimony score is higher than the parental parsimony score, except for network 9, where they are equal. To see the exact scores of both of the methods of every network, check the tables in Appendix A-2.

All the softwired scores of the 60 networks together add up to 4054. All the scores of the parental parsimony scores add up to 2769. The averages of the softwired and the parental parsimony scores per network are 67,5 and 46,2 respectively. The difference between the scores of the methods is 1282, which comes down to a difference of 21,4 per network.

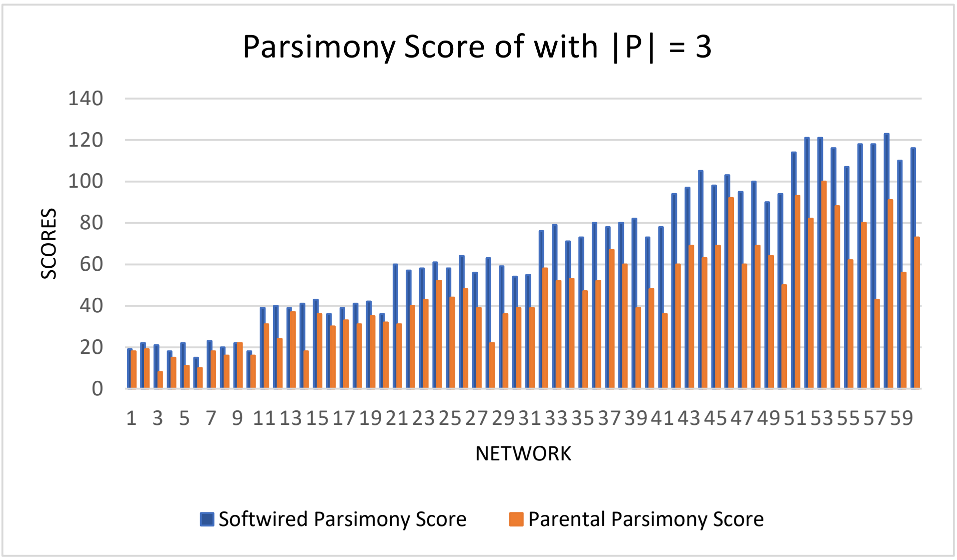

Figure 7-2: Parsimony Scores with $|P| = 3$, with the number of the networks on the x-axis and the corresponding parsimony scores on the y-axis.

7-2-3 $|P| = 4$

The outcome of the experiments where the size of the set of states P is 4, can be viewed in figure 7-3. Here you see a figure with per network, the corresponding value of the softwired parsimony score (in blue) and the parental parsimony score (in orange). Clearly can be seen that the parental parsimony score is lower than the softwired parsimony score for all the networks except for network 9 again, where they are equal. To see the exact scores of the softwired parsimony and the parental parsimony on the networks, check the tables in Appendix A-3.

Furthermore, adding all the softwired parsimony scores together gives a total score of 4923. Doing the same for the parental parsimony, gives a score of 3595. The average scores of the softwired and the parental parsimony per network are 82,0 and 95,1, respectively. The difference between the scores of the methods is 1328 in total and 22,1 per network.

7-2-4 Results Combined

With the 60 networks that were given, we run our method *OnlyParental* three times per method. Each time with a different size of P, as described above. This was done, to see whether we could get more information about the difference between the scores of the parental parsimony and the softwired parsimony.

From the 180 experiments that were done, we made the figures that are shown above. What immediately stood out, was the fact that the parental parsimony score of every network was lower than the softwired parsimony score, regardless of the size of P, except for network 9. For this network, the parsimony scores were exactly the same.

What also stood out, was the fact that the parsimony scores were higher for both methods, when $|P|$ was bigger. This is a bit obvious, because with more different states possible, the

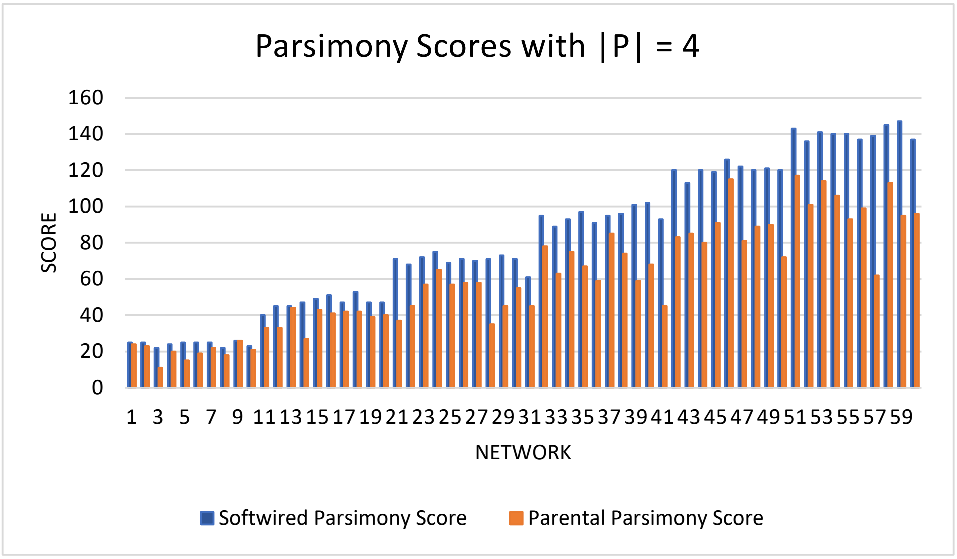

Figure 7-3: Parsimony Scores with $|P| = 4$, with the number of the networks on the x-axis and the corresponding parsimony scores on the y-axis.

leaves had more different states and therefore there had to occur more changes. However, what is also noteworthy, is that it seemed that when there are more states, there is a less significant difference between the two parsimony methods. So we calculated per experiment the difference between the two scores, and divided this by the softwired score. Let us call this score the *relative difference* of the scores. This gave figure 7-4.

In this figure, the x-axis represents the 60 networks and the y-axis represents the relative difference. The gray line represents the experiments where $|P|$ equals two, the orange line represents the experiments where $|P|$ equals three and the blue line represents the experiment where $|P|$ is equal to four. What can be seen in this figure, is that the gray line is almost always above the orange line. Also the orange line is almost always above the blue line. In the results we got, when $|P| = 2$, the relative difference was only in 7 experiments smaller than the relative difference when $|P| = 3$. Also, the relative difference when $|P| = 3$, was only 3 times smaller bigger than the relative difference when $|P| = 4$. This means that the relative difference gets smaller when $|P|$ gets larger. This would imply that the difference between the parsimony scores is indeed less significant when $|P|$ is bigger.

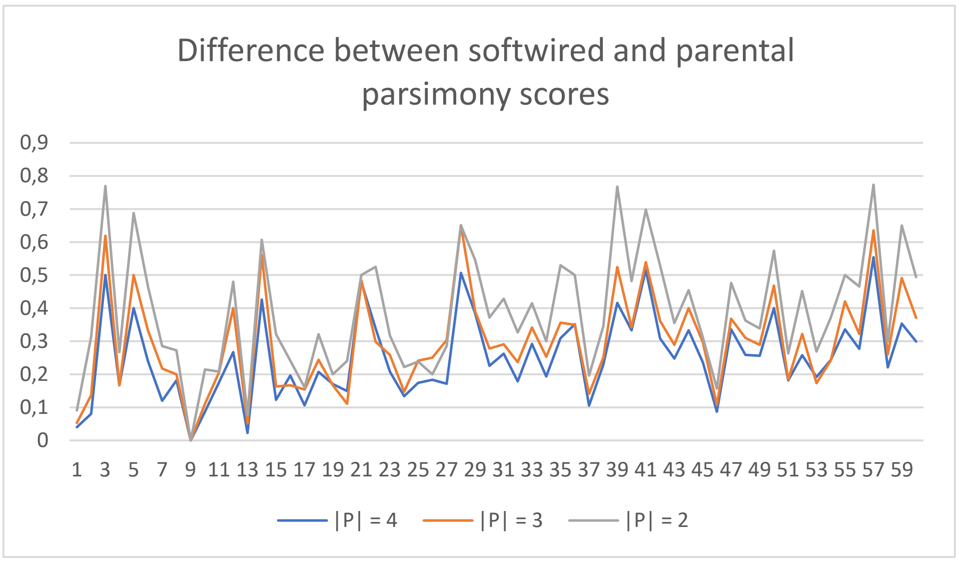

Figure 7-4: Difference between parental parsimony score and softwired parsimony score divided by the softwired parsimony score, per network

Conslusion, Discussion and Recommendations

8-1 Conclusion

In this thesis the following question was addressed: Is the parental parsimony a better and more accurate method for reconstructing phylogenetic networks than the softwired parsimony? To solve this question, 180 experiments where done, where the the parental parsimony score and the softwired parsimony score of 60 different networks were calculated with three different amount of state set sizes and then compared with each other.

To be able to do these experiments, we first needed to find out how we can solve the Parental Parsimony Problem, which gives the parental parsimony score $PS_{pt}(N, \alpha)$ of a network N on a set of taxa X and a p-state character α on X. In order to solve the PPP we characterized the the parental parsimony with lineage functions and translated the PPP into finding the minimum weight lineage function as described in section 4-5. Then we translated this problem into an ILP formulation as described in section 6-1, so that we could implement the problem in Java and use the ILP solver CPLEX. After writing the model *OnlyParental* in Java and implementing an algorithm to find the softwired parsimony scores of an input network as well, we were able to do the experiments to see whether the parental parsimony scores were lower than the softwired parsimony scores.

In the section Results, all the outcomes of the 180 experiments were described. The outcome of every experiment are the two parsimony scores. In every outcome except one, the parental parsimony score was smaller than the softwired parsimony score. For network 9, the scores were equal. We think that this is due to the fact that for this network, the best parental tree is equal to the best tree displayed by the network. For the rest of the tested networks it holds that the parental parsimony scores are significant smaller than the softwired scores. A side note must be added that the larger $|P|$ is, the less significant the difference between the scores is. Recall that the smaller the parsimony score of a parsimony based method is, the more likely it is that the reconstruction the method gives is the correct one. The parental parsimony score is smaller than the softwired score in the most cases and equal to the softwired parsimony score in a few cases. Therefore, we can conclude that parental parsimony is a better and more accurate method for reconstructing phylogenetic networks than the softwired parsimony, but for some networks, the softwired parsimony and the parental parsimony are equal.

8-1-1 Discussion and Recommendations For Further Research

We concluded in the section above that the parental parsimony scores of the tested networks are significant smaller than the softwired parsimony scores of these networks. Also was concluded that the bigger $|P|$ is, the smaller the significance will be. These conclusions immediately cause more questions. How significant is the difference? Which factors have influence on this difference and how do these factors have influence on the difference? Also, for one of the tested networks, the scores were equal. In which cases is the softwired parsimony equal to the parental parsimony? All the described questions can serve as material for further research.

Besides these questions, there are more subjects that would get more attention if we continued this project. The next step of the project would be to improve the script of OnlyParental and let the method graphically display the found network after applying the parental parsimony method. Also, when a network and a character α with the corresponding minimum weight lineage function is given, one could determine what the parental tree of the network would be. Finally we have seen that the computation time of the parental parsimony score in OnlyParental for only one character is sometimes more than two seconds. Calculating the parental parsimony score for a very large network (more than 300 taxa) of one character would take more time and calculating an entire alignment would take even a greater amount of time. For a method that should be able to compute the parental parsimony score quickly, two seconds is a long time. So another next step would be to make the algorithm much faster.

Bibliography

Cplex | ampl. http://ampl.com/products/solvers/solvers-we-sell/cplex/?gclid=Cj0KEQjw4cLKBRCZmNTvyovvj-4BEiQAl_sgQuEihHt-5Pj3LQFR9t8N0_kQ8qUnwuHBXjc_2exjRvEaAklF8P8HAQ. (Accessed on 06/29/2017).

Mpnet: Maximum parsimony on networks. http://homepages.cwi.nl/~iersel/MPNet/. (Accessed on 06/29/2017).

Diestel, R. (2000). *Graphentheory*. Springer.

Fischer, M., Van Iersel, L., Kelk, S., and Scornavacca, C. (2015). On computing the maximum parsimony score of a phylogenetic network. *SIAM Journal on Discrete Mathematics*, 29(1):559–585.

Papadimitriou, C. H. and Steiglitz, K. (1982). *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

Van Iersel, L., Jones, M., and Scornavacca, C. Parental parsimony: a new definition of parsimony for phylogenetic networks.

Appendices

Appendix A: Results of the Experiments

A-1 Appendix A.1: Tables with $|P| = 2$

Network number	1	2	3	4	5	6	7	8	9	10
Number of character states	2	2	2	2	2	2	2	2	2	2
Number of taxa	50	50	50	50	50	50	50	50	50	50
Number of reticulations	17	16	17	16	17	18	16	20	17	16
Softwired Parsimony Score	11	16	13	15	16	15	14	11	15	14
Parental Parsimony Score	10	11	3	11	5	8	10	8	15	11

Network number	11	12	13	14	15	16	17	18	19	20
Number of character states	2	2	2	2	2	2	2	2	2	2
Number of taxa	100	100	100	100	100	100	100	100	100	100
number of reticulations	39	37	37	35	38	36	38	38	33	39
Softwired Parsimony Score	24	25	26	28	28	29	25	28	20	25
Parental Parsimony Score	19	13	24	11	19	22	21	19	16	19

Network number	21	22	23	24	25	26	27	28	29	30
Number of character states	2	2	2	2	2	2	2	2	2	2
Number of taxa	150	150	150	150	150	150	150	150	150	150
Number of reticulations	53	54	53	48	53	53	53	58	55	58
Softwired Parsimony Score	36	40	41	45	42	40	35	40	44	43
Parental Parsimony Score	18	19	28	35	32	32	25	14	20	27

Network number	31	32	33	34	35	36	37	38	39	40
Number of character states	2	2	2	2	2	2	2	2	2	2
Number of taxa	150	200	200	200	200	200	200	200	200	200
number of reticulations	56	71	76	75	77	73	70	70	70	74
Softwired Parsimony Score	35	52	53	50	51	54	56	55	56	56
Parental Parsimony Score	20	35	31	35	24	27	45	36	13	29

Network number	41	42	43	44	45	46	47	48	49	50
Number of character states	2	2	2	2	2	2	2	2	2	2
Number of taxa	200	250	250	250	250	250	250	250	250	250
number of reticulations	72	92	93	92	96	91	87	89	90	93
Softwired Parsimony Score	53	70	62	66	61	70	63	69	68	68
Parental Parsimony Score	16	33	40	36	42	59	33	44	45	29

Network number	51	52	53	54	55	56	57	58	59	60
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	300	300	300	300	300	300	300	300	300	300
number of reticulations	72	92	93	92	96	91	87	89	90	93
Softwired Parsimony Score	76	73	78	81	76	73	75	77	80	83
Parental Parsimony Score	56	40	57	51	38	39	17	54	28	42

A-2 Appendix A.1: Tables with $|P| = 3$

Network number	1	2	3	4	5	6	7	8	9	10
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	50	50	50	50	50	50	50	50	50	50
Number of reticulations	17	16	17	16	17	18	16	20	17	16
Softwired Parsimony Score	19	22	21	18	22	15	23	20	22	18
Parental Parsimony Score	18	19	8	15	11	10	18	16	22	16

Network number	11	12	13	14	15	16	17	18	19	20
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	100	100	100	100	100	100	100	100	100	100
number of reticulations	39	37	37	35	38	36	38	38	33	39
Softwired Parsimony Score	39	40	39	41	43	36	39	41	42	39
Parental Parsimony Score	31	24	37	18	36	30	33	31	35	32

Network number	21	22	23	24	25	26	27	28	29	30
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	150	150	150	150	150	150	150	150	150	150
Number of reticulations	53	54	53	48	53	53	53	58	55	58
Softwired Parsimony Score	60	57	58	61	58	64	56	63	59	54
Parental Parsimony Score	31	40	43	52	44	48	39	22	36	39

Network number	31	32	33	34	35	36	37	38	39	40
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	150	200	200	200	200	200	200	200	200	200
number of reticulations	56	71	76	75	77	73	70	70	70	74
Softwired Parsimony Score	55	76	79	71	73	80	78	80	82	73
Parental Parsimony Score	39	58	52	53	47	52	67	60	39	48

Network number	41	42	43	44	45	46	47	48	49	50
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	200	250	250	250	250	250	250	250	250	250
number of reticulations	72	92	93	92	96	91	87	89	90	93
Softwired Parsimony Score	78	94	97	105	98	103	95	100	90	94
Parental Parsimony Score	36	60	69	63	69	92	60	69	64	50

Network number	51	52	53	54	55	56	57	58	59	60
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	300	300	300	300	300	300	300	300	300	300
number of reticulations	72	92	93	92	96	91	87	89	90	93
Softwired Parsimony Score	114	121	121	116	107	118	118	123	110	116
Parental Parsimony Score	93	82	100	88	62	80	43	91	56	73

A-3 Appendix A.1: Tables with $|P| = 4$

Network number	1	2	3	4	5	6	7	8	9	10
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	50	50	50	50	50	50	50	50	50	50
Number of reticulations	17	16	17	16	17	18	16	20	17	16
Softwired Parsimony Score	25	25	22	24	25	25	25	22	26	23
Parental Parsimony Score	24	23	11	20	15	19	22	18	26	21
Network number	11	12	13	14	15	16	17	18	19	20
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	100	100	100	100	100	100	100	100	100	100
number of reticulations	39	37	37	35	38	36	38	38	33	39
Softwired Parsimony Score	40	45	45	47	49	51	47	53	47	47
Parental Parsimony Score	33	33	44	27	43	41	42	42	39	40
Network number	21	22	23	24	25	26	27	28	29	30
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	150	150	150	150	150	150	150	150	150	150
Number of reticulations	53	54	53	48	53	53	53	58	55	58
Softwired Parsimony Score	71	68	72	75	69	71	70	71	73	71
Parental Parsimony Score	37	45	57	65	57	58	58	35	45	55
Network number	31	32	33	34	35	36	37	38	39	40
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	150	200	200	200	200	200	200	200	200	200
number of reticulations	56	71	76	75	77	73	70	70	70	74
Softwired Parsimony Score	61	95	89	93	97	91	95	96	101	102
Parental Parsimony Score	45	78	63	75	67	59	85	74	59	68
Network number	41	42	43	44	45	46	47	48	49	50
Number of character states	3	3	3	3	3	3	3	3	3	3
Number of taxa	200	250	250	250	250	250	250	250	250	250
number of reticulations	72	92	93	92	96	91	87	89	90	93
Softwired Parsimony Score	93	120	113	120	119	126	122	120	121	120
Parental Parsimony Score	45	83	85	80	91	115	81	89	90	72
Network number	51	52	53	54	55	56	57	58	59	60
Number of character states	4	4	4	4	4	4	4	4	4	4
Number of taxa	300	300	300	300	300	300	300	300	300	300
number of reticulations	72	92	93	92	96	91	87	89	90	93
Softwired Parsimony Score	143	136	141	140	140	137	139	145	147	137
Parental Parsimony Score	117	101	114	106	93	99	62	113	95	96

Appendix B: Script of method OnlyParental

```
import java.io.*;
import java.util.*;
import ilog.concert.*;
import ilog.cplex.*;

public class OnlyParental {

    // uses CPLEX
    public static boolean DEBUG = false;
    public static boolean SILENT = false;
    static int GAP = 9999;

    public static void main(String[] args) {
        //System.out.println("parental parsimony score is" + ps);

        Long seed = new Long(487641078);
        Random generator = new Random(seed);
        int maxstates = 0;

        if (args.length < 2 || args.length > 9) {
            System.out.println("----------- OnlyParental -----------");
            System.out.println("Software for computing the (hardwired and ...
                softwired) maximum parsimony scores of a phylogenetic network");
            System.out.println("Uses CPLEX");
            System.out.println("----------- USAGE -----------");
            System.out.println("java OnlyParental network.tree sequences.fasta ...
                [options]");
            System.out.println("network.tree should contain at least one network ...
                in e-newick format");
            System.out.println("sequences.fasta should contain, on each line, a ...
                taxon name followed by a space and a character state, or a ...
                sequence of character states");
            System.out.println("---------- OPTIONS ----------");
            System.out.println("--nolabels\t hides taxon labels");
            System.out.println("--nostates\t hides character states");
            System.out.println("--softwired\t only compute the softwired ...
                parsimony score, not the hardwired one");
            System.out.println("--hardwired\t only compute the hardwired ...
                parsimony score, not the softwired one");
            //System.out.println("--approx\t compute an approximation (faster)");
            System.out.println("--rand k\t use character states randomly chosen ...
                from 1 to k");
            System.out.println("--silent k\t do not show intermediate results");
            System.out.println("-Djava.library.path=[path to cplex.jar]\t to ...
                tell java the location of cplex.jar");
            return;
        }

        boolean nolabels = false;
        boolean nostates = false;
        boolean onlysoftwired = false;
        boolean onlyhardwired = false;
        boolean rand = false;
        int num_states = -1;
        boolean relax = false;
        for (int i = 1; i < args.length; i++) {
            if (args[i].equals("--nolabels")) {
                nolabels = true;
            } else if (args[i].equals("--nostates")) {
                nostates = true;
            } else if (args[i].equals("--softwired")) {
                onlysoftwired = true;
            } else if (args[i].equals("--hardwired")) {
                onlyhardwired = true;
            } else if (args[i].equals("--silent")) {
                SILENT = true;
            } else if (args[i].equals("--approx")) {
                relax = true;
            } else if (args[i].equals("--rand")) {
                rand = true;
                try {
                   num_states = Integer.parseInt(args[i + 1]);
                } catch (NumberFormatException nfe) {
                   System.out.println("** Integer number of states required");
                   return;
                }
                i++;
            } else if (i > 1) {
                System.out.println("Unknown option: " + args[i]);
                return;
            }
        }

        String netwerkFile = args[0];
        if (!SILENT) {
            System.out.println("\\ ** Reading e-newick from " + netwerkFile + ...
                "...");
        }

        File file = new File(netwerkFile);
        BufferedReader reader = null;
        String newick = null;
        Vector<String> newicks = new Vector();
        try {
            reader = new BufferedReader(new FileReader(file));
            while ((newick = reader.readLine()) != null) {
                newicks.add(newick);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            return;
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        Vector<Integer> num_taxa = new Vector();
        Vector<Integer> num_retic = new Vector();
        Vector<Integer> hardwired_score = new Vector();
        Vector<Integer> softwired_score = new Vector();
        Vector<Double> hardwired_time = new Vector();
        Vector<Double> softwired_time = new Vector();

        // loop through input networks
        for (String n : newicks) {
            if(n.equals("")) continue;

            if (newicks.indexOf(n) > 0 & !SILENT) {
                System.out.println("***** Results so far *****");
                System.out.println("***** Random seed: " + seed);
                System.out.println("***** Numbers of taxa: " + num_taxa.toString());
                System.out.println("***** Numbers of reticulations: " + ...
                    num_retic.toString());
                System.out.println("***** Hardwired parsimony scores: " + ...
                    hardwired_score.toString());
                System.out.println("***** Softwired parsimony scores: " + ...
                    softwired_score.toString());
                System.out.println("***** Computation time hardwired parsimony ...
                    score: " + hardwired_time.toString());
                System.out.println("***** Computation time softwired parsimony ...
                    score: " + softwired_time.toString());
            }

            System.out.println("\\ ** Processing network " + (newicks.indexOf(n) ...
                + 1) + " out of " + newicks.size());

//          System.out.println("\\ ** Read the following:");
//          System.out.println("\\ " + newick);
            Long hardwiredTime = new Long(0);
            Long softwiredTime = new Long(0);

            Network.TAXON_LABELS = new Vector();
            Network.STATE_LABELS = new Vector();
            if (!SILENT) {
                System.out.println("\\ ** Parsing...");
            }
            Network N = Network.newick2netwerk(n);

            Vector<Vector<Character>> allStates = new Vector();
            Vector<String> taxa = new Vector();

            if (!rand) {
                // read character states from file
                String characterFile = args[1];
                if (!SILENT) {
                    System.out.println("\\ ** Reading character data from " + ...
                        characterFile + "...");
                }

                file = new File(characterFile);
                String record = null;
                try {
                    reader = new BufferedReader(new FileReader(file));
                    Vector<Character> states = new Vector();
                    while ((record = reader.readLine()) != null) {
                        if (record.length() == 0 || record.startsWith("//")) {
                            continue; // ignore comments and empty lines
                        }
                        if (record.startsWith(">")) {
                            if (!states.isEmpty()) {
                                allStates.add(states);
                                states = new Vector();
                            }
                            String[] data = record.split(" ");
                            taxa.add(data[0].substring(1));
                            if (data.length > 1) {
                                for (char a : data[1].toCharArray()) {
                                    states.add(a);
                                }
                            }
                        } else {
                            String[] data = record.split(" ");
                            if (data.length > 1) {
                                if (!states.isEmpty()) {
                                    allStates.add(states);
                                    states = new Vector();
                                }
                                taxa.add(data[0]);
                                for (char a : data[1].toCharArray()) {
                                    states.add(a);
                                }
                            } else {
                                for (char a : data[0].toCharArray()) {
                                    states.add(a);
                                }
                            }
                        }
                    }
                    if (!states.isEmpty()) {
                        allStates.add(states);
                    }
                    reader.close();
                } catch (FileNotFoundException e) {
                    e.printStackTrace();
                    return;
                } catch (IOException e) {
                    e.printStackTrace();
                    return;
                }
            } else {
                // assign random states
                if (!SILENT) {
```

```
        System.out.println("\\ ** Assigning random character states");
    }
    for (int i = 0; i < Network.TAXON_LABELS.size(); i++) {
        Vector<Character> states = new Vector();
        int c = generator.nextInt(num_states) + 1;
        char cc = (char) (c + 65);
        states.add(cc);
        allStates.add(states);
        taxa.add(Network.TAXON_LABELS.elementAt(i));
    }
}

if (taxa.size() != Network.TAXON_LABELS.size()) {
    System.out.println("Error: character and network do not have the ...
        same number of taxa.");
    System.out.println("Network taxa: " + ...
        Network.TAXON_LABELS.toString());
    System.out.println("Character taxa: " + taxa.toString());
    return;
} else if (!taxa.containsAll(Network.TAXON_LABELS)) {
    // System.out.println(taxa.toString());
    // System.out.println(Network.TAXON_LABELS);
    System.out.println("Error: character and network do not have ...
        identical taxon sets.");
    System.out.println("Network taxa: " + ...
        Network.TAXON_LABELS.toString());
    System.out.println("Character taxa: " + taxa.toString());
    return;
}

int ss = 0; // softwired parsimony score
int hs = 0; // hardwired parsimony score
String uOutFile = netwerkFile + ".hardwiredPS.dot";
String uPDFFile = uOutFile + ".pdf";
String rOutFile = netwerkFile + ".softwiredPS.dot";
String rPDFFile = rOutFile + ".pdf";
String eol = System.getProperty("line.separator");

// construct vector with all vertices
// this also numbers the vertices
N.cleanNetwork();
int[] num = new int[1];
num[0] = Network.TAXON_LABELS.size() + 1;
Vector<Network> vertices = N.getVertices(num);

if (!OnlyParental.SILENT) {
    System.out.println("\\ ** Network has " + ...
        Network.TAXON_LABELS.size() + " taxa.");
}
if (!OnlyParental.SILENT) {
    System.out.println("\\ ** Network has " + vertices.size() + " ...
        vertices.");
}

// construct vector with all edges
Vector<Vector<Network>> edges = N.getEdges();
if (!OnlyParental.SILENT) {
    System.out.println("\\ ** Network has " + edges.size() + " edges.");
```

```
}

// construct vector with all reticulations
// this also numbers the vertices
N.cleanNetwork();
num[0] = Network.TAXON_LABELS.size() + 1;
Vector<Network> reticulations = N.getReticulations(num);
if (!OnlyParental.SILENT) {
    System.out.println("\\ ** Network has " + reticulations.size() + ...
        " reticulations.");
}

for (int state_index = 0; state_index < ...
    allStates.elementAt(0).size(); state_index++) {

    int hcs = 0; // hardwired parsimony score of this character
    int scs = 0; // softwired parsimony score of this character

    if (!SILENT && allStates.elementAt(0).size() > 1) {
        System.out.println("\\ ** Processing character " + ...
            (state_index + 1) + " out of " + ...
            allStates.elementAt(0).size() + "...");
    }

    // clear existing character states
    N.clearStates();
    //N.cleanNetwork();
    Network.STATE_LABELS = new Vector();

    // add character data to network
    int k = N.setCharacterStates(taxa, allStates, state_index);
    if (k > maxstates) {
        maxstates = k;
    }

    if (Network.STATE_LABELS.isEmpty()) {
        System.out.println("\\ ** No states.");
        continue;
    }

    if (Network.STATE_LABELS.size() == 1) {
        System.out.println("\\ ** Only one state.");
        continue;
    }

    if (!OnlyParental.SILENT) {
        System.out.println("\\ ** Character has " + k + " states.");
    }

    // compute parsimony scores
    // first softwired
    if (!onlyhardwired) {

        if (!SILENT) {
            System.out.println("** ----- Solving softwired ILP with ...
                CPLEX ------");
        }
        Long startingTime = System.currentTimeMillis();
```

```
            // compute softwired parsimony score
            computePS(true, relax, N, vertices, edges, reticulations);
            if (relax) {
                // in case some states are still -1 we assing them a value
                N.roundStates(true);
            }
            scs = N.getScore(true);
            N.resetSeen();

            N.finaliseStates(state_index, true);
            N.resetSeen();

            //update time
            softwiredTime += System.currentTimeMillis() - startingTime;
            if (!SILENT) {
                System.out.println("** ----- Finished solving softwired ...
                    ILP with CPLEX -----");
            }
        }

        // now hardwired
        int rel_hcs = 0;
    /*    if (!onlysoftwired) {

            Long startingTime = System.currentTimeMillis();
            if (!SILENT) {
                System.out.println("** ----- Solving hardwired ILP with ...
                    CPLEX ------");
            }

            // compute hardwired parsimony score
            N.clearInternalStates();
            computePS(false, relax, N, vertices, edges, reticulations);
            N.roundStates(false);
            N.resetSeen();
            hcs = N.getScore(false);
            N.resetSeen();
            N.finaliseStates(state_index, false);
            N.resetSeen();

            //update time
            hardwiredTime += System.currentTimeMillis() - startingTime;
            if (!SILENT) {
                System.out.println("** ----- Finished solving hardwired ...
                    ILP with CPLEX -----");
            }
        }*/

        ss += scs;
      //  hs += hcs;
    }

    // compute edge support
    N.computeRetEdgeSupport();

    // output
    Vector<String> networkStrings;
```

```
if (!onlyhardwired) {

    if (newicks.size() == 1) {
        // write output network to file
        networkStrings = N.toDot(nolabels, nostates, true);
        try {
            BufferedWriter out = new BufferedWriter(new ...
                FileWriter(rOutFile));
            for (String line : networkStrings) {
                out.write(line + eol);
            }
            out.close();
            if (!SILENT) {
                System.out.println("** Network for softwired ...
                    parsimony score in DOT format has been written ...
                    to: " + rOutFile);
            }

        } catch (IOException e) {
            System.out.println("** Could not write output network to ...
                file.");
        }

        try {
            String line;
            Process p = Runtime.getRuntime().exec("dot -Tpdf " + ...
                rOutFile + " -O");
            BufferedReader bre = new BufferedReader(new ...
                InputStreamReader(p.getErrorStream()));
            while ((line = bre.readLine()) != null) {
                if (!SILENT) {
                    System.out.println(line);
                }
            }
            bre.close();
            p.waitFor();
            System.out.println("** Network for softwired parsimony ...
                score in PDF format has been written to: " + rPDFFile);

        } catch (Exception err) {
            System.out.println("** Could not convert network to PDF ...
                format.");
        }
    }
}

if (!onlysoftwired) {

    if (newicks.size() == 1) {

        networkStrings = N.toDot(nolabels, nostates, false);
        try {
            BufferedWriter out = new BufferedWriter(new ...
                FileWriter(uOutFile));
            for (String line : networkStrings) {
                out.write(line + eol);
            }
            out.close();
        } catch (IOException e) {
            return;
        }

        if (!SILENT) {
            System.out.println("** Network for hardwired parsimony ...
                score in DOT format has been written to: " + uOutFile);
        }

        try {
            String line;
            Process p = Runtime.getRuntime().exec("dot -Tpdf " + ...
                uOutFile + " -O ");
            BufferedReader bre = new BufferedReader(new ...
                InputStreamReader(p.getErrorStream()));
            while ((line = bre.readLine()) != null) {
                System.out.println(line);
            }
            bre.close();
            p.waitFor();
            if (!SILENT) {
                System.out.println("** Network for hardwired ...
                    parsimony score in PDF format has been written ...
                    to: " + uPDFFile);
            }

        } catch (Exception err) {
            System.out.println("** Could not convert network to PDF ...
                format.");
        }
    }
}

N.cleanNetwork();
if (!SILENT) {
    System.out.println("***** Number of taxa: " + ...
        Network.TAXON_LABELS.size());
}
if (!SILENT) {
    System.out.println("***** Number of reticulations: " + ...
        reticulations.size());
    System.out.println("***** Number of characters: " + ...
        allStates.elementAt(0).size());
    System.out.println("***** Number of character states: " + ...
        maxstates);
}

if (!onlysoftwired & !SILENT) {
    System.out.println("***** Hardwired Parsimony Score: " + hs);
}
if (!onlyhardwired & !SILENT) {
    System.out.println("***** Softwired Parsimony Score, no, ...
        Parental Parsimony Score: " + ss);
}

if (!SILENT) {
    System.out.println("***** Computation time for Hardwired ...
        Parsimony Score " + hardwiredTime / 1000 + " seconds.");
```

```
                System.out.println("***** Computation time for Softwired ...
                    Parsimony Score: " + softwiredTime / 1000 + " seconds.");
            }
            num_taxa.add(Network.TAXON_LABELS.size());
            num_retic.add(reticulations.size());
            hardwired_score.add(hs);
            softwired_score.add(ss);
            hardwired_time.add(hardwiredTime / 1000.0);
            softwired_time.add(softwiredTime / 1000.0);

        }

        if (newicks.size() > 1) {
            System.out.println("***** Finished all networks");
            System.out.println("***** Random seed: " + seed);
            System.out.println("***** Numbers of taxa: " + num_taxa.toString());
            System.out.println("***** Numbers of reticulations: " + ...
                num_retic.toString());
            System.out.println("***** Hardwired parsimony scores: " + ...
                hardwired_score.toString());
            System.out.println("***** Softwired parsimony scores: " + ...
                softwired_score.toString());
            System.out.println("***** Computation time hardwired parsimony ...
                score: " + hardwired_time.toString());
            System.out.println("***** Computation time softwired parsimony ...
                score: " + softwired_time.toString());
        }

        // this can be use to compute average computation times per run
        if (newicks.size() == 60) {
            for (int run = 0; run < 6; run++) {
                double hwsum = 0;
                double swsum = 0;
                for (int i = run * 10; i < run * 10 + 10; i++) {
                    hwsum += hardwired_time.elementAt(i);
                    swsum += softwired_time.elementAt(i);
                }
                System.out.println("***** Avg computation time hardwired run " + ...
                    (run + 1) + ": " + (hwsum / 10.0));
                System.out.println("***** Avg computation time softwired run " + ...
                    (run + 1) + ": " + (swsum / 10.0));
            }
        }
    }


    public static double computePS(boolean softwired, boolean relax, Network N, ...
        Vector<Network> vertices, Vector<Vector<Network>> edges, ...
        Vector<Network> reticulations) {
        double ps = -1; // the parsimony score

        // Vector of Strings with ILP formulation
        Vector<String> ILPStrings = new Vector();
        String eol = System.getProperty("line.separator");

        // generate ILP for CPLEX
        int k = Network.STATE_LABELS.size();      //int k is number of states
        ILPStrings.add("MINIMIZE");      // writes "minimize"
```

```
// objective function
String obj = "";
boolean gotone = false;                          //need to start with c_w
for (Network v : vertices){                        //loops through vertices
    if (v.state == GAP) {                          //state cant be gap
        continue;
    }
    if (gotone) {
        obj += " + ";
    }
    gotone = true;
    obj += "c_" + v.number;
}


ILPStrings.add(obj);
ILPStrings.add("Subject To");

//vertex constraints
//constraint 6: sum over states of x_root,s = 1
for (Network v : vertices) {      //loop trough vertices
    if (v.isRoot) {
        String con = "";          //make a string
        for (int s = 0; s < k; s++) {   //loop through states
            if (s > 0) {          //start with an x first
                con += " + ";
            }
            con += "x_" + v.number + "," + s; //x_number of root states
        }
        con += " = 1";
        ILPStrings.add(con);              //so now som over states of ...
            x_root,s = 1.
    }

}
//constraint 8: weight y of root is 0 for every state s
    for (Network v : vertices) {
        if (v.isRoot) {
            for (int s = 0; s < k; s++) {  //loop through states
                String con = "y_" + v.number + "," + s + " = 0";
                ILPStrings.add(con);
            }
        }
    }

    //constraint 3: weight y_v,s ≥0
    for (Network v : vertices) {
        for (int s = 0; s < k; s++) {
            String con = "y_" + v.number + "," + s + " ≥0";
            ILPStrings.add(con);
        }
    }


    //constraint 9: c_v - sum over all states of y_v,s ≥0
    for (Network v : vertices) {
        String con = "";
        for (int s = 0; s < k; s++) {   //loop through states
```

```
            con += " - y_" + v.number + "," + s; //y_number of root states
        }
        ILPStrings.add("c_" + v.number + con + " ≥" + 0);
    }

    //constraint 2 and 5: x_v,s = 0 for every v in X and s != a(v), and ...
        x_v,s = 1 if s = a(v)
    for (Network v : vertices) {     // loop through all vertices
        if (v.isLeaf) {            //we are only interested in the leaves
            int a = v.state;
            for (int s = 0; s < k; s++) { //loop through all states
                if (a == s) {         //see if the state is the state of ...
                    the leaf
                    ILPStrings.add("x_" + v.number + "," + s + " = 1");
                } else { //if a!=a,           //if the state we are ...
                    looking at, is not equal to the state of v, x = 0
                    ILPStrings.add("x_" + v.number + "," + s + " = 0");
                }
            }
        }
    }
    // edge constraints, constraint 4                    //y_v,s ≥x_v,s ...
        - sum over parents u x_u,s
    for (Network v : vertices) {     //for every vertex v in V.
        //Network u = v.parent;      // get parent u of v
        if (v.isRoot){
            continue;
        }
        if (v.isLeaf && v.state != GAP) { //if v is leaf, then x_v,a(v)=1
            int s = v.state;
            String xvs = "x_" + v.number + "," + s;
            String yvs = "y_" + v.number + "," + s;
            String xus = "";                  //0 if v is root and u does ...
                not exist.
            boolean first = true; //to see if we already have a first element

            for (Network u : v.parents) { //loops through parents u of v



             //  if (first) {
            //      xus = "x_" + u.number + "," + s;  //if u is first ...
                  parent of v: start the string with this x
             //  } else {
                    xus += " + x_" + u.number + "," + s;
       //      }
                first = false;         //after we have the first element, ...
                    this boolean is false
            }
            // now we have for leaf v everything to make the string
            //String con = yvs + "≥" + xvs + "-1*(" + xus + ")";    ...
                //every variable has to be on the left hand side (LHS)
            String con = yvs + " - " + xvs + xus + " ≥" + 0;
            ILPStrings.add(con);
       } else if (v.isLeaf) {
            //v is a gap
            // no constraint
        } else {
```

```
        for (int s = 0; s < k; s++) {   //loop through all the states
            String xvs = "x_" + v.number + "," + s;
            String yvs = "y_" + v.number + "," + s;
            String xus = "";
            boolean first = true; //to see if we already have a first ...
                element
            for (Network u : v.parents) { //loops through parents u ...
                of v
                //is u parent of v? if Yes: add to string
              //  if (first) {
               //     xus = "x_" + u.number + "," + s;   //if u is ...
                   first parent of v: start the string with this x
               // } else {
                    xus += " + x_" + u.number + "," + s;
              //  }
                first = false;          //after we have the first ...
                    element, this boolean is false
            }

            // now we have for leaf v everything to make the string
            //String con = yvs + "≥" + xvs + "-1 *(" + xus + ")";    ...
                //everything on LHS?
            String con = yvs + " - " + xvs + xus + " ≥" + 0;
            ILPStrings.add(con);
        }
    }
}
   //Constraint 1: sum(over s) x_v,s - sum(over s) sum(over u in ...
       par(v)) x_u,s ≤0
for (Network v : vertices) {     //loop through all vertices
    if(v.isRoot){
        continue;
    }
    String con1 = "";             //make a string for som over x_v,s
    String con2 = "0";         //make a string for sum over s of sum ...
        over edges into v of x_u,s
    for (int s = 0; s < k; s++) {   //loop through states
        if (s > 0) {            //start with an x first
            con1 += " + ";
        }
        con1 += "x_" + v.number + "," + s; //con1 is now sum over s ...
            of x_v,s
    }
    boolean first = true; //to see if we already have a first element
    for (int s = 0; s < k; s++) {   //loop through states
        for (Network u : v.parents) { //need to build something that ...
            gets the parents of v, loops through all vertices
            if (first) {
                con2 = "x_" + u.number + "," + s;   //if u is first ...
                    parent of v: start the string with this x
            } else {
                con2 += " - x_" + u.number + "," + s;
            }
            first = false;        //after we have the first element, ...
                this boolean is false
        }
    }
```

```
        //  String con = con1 + "-1*(" + con2 + ")≤0";    //som x_v,s - ...
            som over som x_u,s ≤0
              String con = con1 + " - " + con2 + " ≤0";           //this ...
                  is better?
            ILPStrings.add(con);
        }

    //constraint 7: x is 0 or 1
    // bounds
    if (relax) {                        //relax mean relaxation: no integers
        ILPStrings.add("Bounds");
        for (Network vertex : vertices) {                                  ...
            //loops through vertices
            if (vertex.isLeaf) {                                           ...
                // dont do anything when x is a leaf, why??
                continue;
            }
            for (int s = 0; s < k; s++) {                                  ...
                //loop through states
                ILPStrings.add("0 ≤x_" + vertex.number + "," + s + " ≤1"); ...
                    //adds bounds, x between 0 and 1
            }
        }
    } else {
        ILPStrings.add("Binary");        //when everything is binary:
        //this will give every variable
        for (Network vertex : vertices) {  //loops through vertices
            if (vertex.isLeaf) {         //doesnt give leafs
                continue;
            }
            for (int s = 0; s < k; s++) {
                ILPStrings.add("x_" + vertex.number + "," + s);
            }
        }
     }

                //---------------------
        ILPStrings.add("End");

       // write ILP to file
    try {
        BufferedWriter out = new BufferedWriter(new FileWriter("ILP.tmp"));
        for (String line : ILPStrings) {
            out.write(line + eol);
        }
        out.close();
    } catch (IOException e) {
        return -1;
    }

    // ----- run CPLEX -----
    try {

        IloCplex cplex = new IloCplex();                                   ...
            //make new CPLEX object: to solve our model

        //! fileName is the name of the file where your ILP is
```

```
cplex.importModel("ILP.tmp");                                           ...
    //imports our ILP Model

//! uncomment this to suppress visual output from cplex
cplex.setOut(null);

//! this is the solving bit
if (cplex.solve()) {                                                    ...
    //returns boolean, whether cplex has found a solution
    IloNumVar[] var = parse(cplex);                                     ...
        //if cplex found a solution, then var is now the solution
                                                                        //(parse ...
                                                                            translates ...
                                                                            from ...
                                                                            cplex ...
                                                                            to ...
                                                                            java

    //! read the optimal solution
    double x[] = cplex.getValues(var);                                  ...
        //puts solution in an double array
    int yCounter = 0;
    if (relax) {                                                        ...
        //i'm not going to relax it, let this be
        // give each vertex a state with maximum x_{v,s} value
        for (int loop = 0; loop < x.length; loop++) {
            String varname = var[loop].getName();
            String[] splitVarName = varname.split("_");
            if (splitVarName[0].equals("x")) {
                // this is an x-variable (indicating character state)
                String[] doubleSplit = splitVarName[1].split(",");
                int vertexNum = Integer.parseInt(doubleSplit[0]);
                int stateValue = Integer.parseInt(doubleSplit[1]);
                Network v = N.getVertex(vertexNum);
                // give the corresponding vertex the right state
                if(x[loop] > v.stateDouble) {
                    v.state = stateValue;
                    v.stateDouble = x[loop];
                }
            } else {
                // this is a c-variable: skip
            }
        }
    } else {
        for (int loop = 0; loop < x.length; loop++) {
            System.out.println(var[loop].getName() + " = " + ...
                x[loop]);   //let us read what the values of the ...
                variables are
            String varname = var[loop].getName(); ...
                                        //varnaam is the String of the ...
                solution in var
            int intvalue = (int) Math.round(x[loop]); ...
                                    // rond de solution af op integer, ...
                noem deze intvalue
            if (intvalue == 1) { ...
                                                        //als ...
                intvalue ==1, dan is x_v,s =1 en gebruiken we hem
```

```
                    String[] splitVarName = varname.split("_"); ...
                                      //splitst de string van varname in ...
                        x/y en v,s
                    if (splitVarName[0].equals("y")) {
                        //this is a y-variable (counting the changes), we ...
                            could do something with this
                        yCounter++; ...
                                                                    // ...
                            This will count the number of y_v,s that are ...
                            1, so basically the parsimony score.


                        /*                  This below is for softwires ...
                            parsimony
                         int edgenum = Integer.parseInt(splitVarName[1]);
                         Vector<Network> edge = ...
                             softwiredILP.edges.elementAt(edgenum);
                         Network u = edge.elementAt(0);
                         Network v = edge.elementAt(1);
                         int index = v.parents.indexOf(u);
                         // record that this edge was switched on for this ...
                             character
                         Vector<Boolean> used = new Vector();
                         used.setSize(v.parents.size());
                         used.set(index, true);
                         v.retEdgeUsed.set(state_index,used);
                          */
                     }

                    if (splitVarName[0].equals("x")) {
                        // this is an x-variable (indicating character state)
                        String[] doubleSplit = splitVarName[1].split(","); ...
                               //splits in x en v,s
                        int vertexNum = Integer.parseInt(doubleSplit[0]); ...
                                //kijkt naar v, dit is vertexNum
                        int stateValue = Integer.parseInt(doubleSplit[1]); ...
                               //kijkt naar s, dit is stateValue
                        // give the corresponding vertex the right state
                        N.setStates(vertexNum, stateValue); ...
                                                //Hier moet iets anders komen,
                        //N.setVectorStates(vertexNum, stateValue);
                        System.out.println("In ComputePS, we have the ...
                            following vertices with the following states: ...
                            "+ vertexNum + " vertexnumm en state value "+ ...
                            stateValue);
                    } else {
                        // this is a c-variable: skip
                    }
                }
            }
            System.out.println("according to computePS and the yCounter, ...
                the parsimony score is "+ yCounter);
        }
    }

    //! this gets the objective function value, rounded to an int
    ps = cplex.getObjValue();
```

```
            //! this deallocates the CPLEX resources
            cplex.end();

        } catch (IloException e) {
            System.out.println("Something went wrong with CPLEX.");
            System.out.println(e.getMessage());
            System.exit(0);
        }

        return ps;
    }




    private static IloNumVar[] parse(IloCplex cplex) throws IloException {
        HashSet<IloNumVar> vars = new HashSet<IloNumVar>();
        Iterator it = cplex.iterator();
        IloLinearNumExpr expr;
        IloLinearNumExprIterator it2;
        while (it.hasNext()) {
            IloAddable thing = (IloAddable) it.next();
            if (thing instanceof IloRange) {
                expr = (IloLinearNumExpr) ((IloRange) thing).getExpr();
                it2 = expr.linearIterator();
                while (it2.hasNext()) {
                    vars.add(it2.nextNumVar());
                }
            } else if (thing instanceof IloObjective) {
                expr = (IloLinearNumExpr) ((IloObjective) thing).getExpr();
                it2 = expr.linearIterator();
                while (it2.hasNext()) {
                    vars.add(it2.nextNumVar());
                }
            } else if (thing instanceof IloSOS1) {
                vars.addAll(Arrays.asList(((IloSOS1) thing).getNumVars()));
            } else if (thing instanceof IloSOS2) {
                vars.addAll(Arrays.asList(((IloSOS2) thing).getNumVars()));
            } else if (thing instanceof IloLPMatrix) {
                vars.addAll(Arrays.asList(((IloLPMatrix) thing).getNumVars()));
            }
        }
        IloNumVar[] varray = vars.toArray(new IloNumVar[1]);
        return varray;
    }

}

class Network {

    static int MAX_RET = 0; // for printing purposes
    static int GAP = 9999;
    static Vector<String> TAXON_LABELS = new Vector();
    static Vector<Character> STATE_LABELS = new Vector();
    Vector<Network> children;
    Vector<Network> parents;
    int state;
    Vector<Integer> MultiStates;                //not sure which one i'm using
    LinkedHashSet<Integer> PStates;        //not sure which one i'm using
    double stateDouble;
    Vector<Character> softwiredStates;
    Vector<String> parentalStates;
    Vector<Character> hardwiredStates;
    Vector<Double> retEdgeSupport;
    boolean isLeaf;
    String label;
    Vector TreeVertices;
    int aafComp;
    boolean isRoot;
    int retNum;
    boolean seen;
    public int number;

    public Network() {
        isLeaf = false;
        label = null;
        parents = new Vector();
        children = new Vector();
        TreeVertices = new Vector();
        MultiStates = new Vector();           //vector of states
        PStates = new LinkedHashSet();   //LinkedHashSet of states, this object ...
            does not contain duplicates
        aafComp = -1;
        retNum = -1;
        seen = false;
        isRoot = false;
        number = 0;
        state = -1;
        stateDouble = -1;
        parentalStates = new Vector();
        hardwiredStates = new Vector();
        softwiredStates = new Vector();
        retEdgeSupport = new Vector();
    }

    public static Network newick2netwerk(String newick) {
        if (newick.endsWith(";")) {
            int lastclosepar = newick.lastIndexOf(")");
            newick = newick.substring(0, lastclosepar + 1);
        } else {
            return null;
        }
        Network N = newick2netwerk(newick, new Vector());
        N.isRoot = true;
        N.cleanNetwork();

        // suppress indegree-1 outdegree-1
        N.suppress();

        return N;
    }

    public boolean setState(int vertex_num, int state) {
        if(this.number == vertex_num) {
            this.state = state;                                 //will be ...
                used for finalise states
            return true;
        }
        for (Network child : children) {
            boolean cb = child.setState(vertex_num, state);
            if(cb) {
                return true;
            }
        }
        return false;
    }



    public boolean setStates(int vertex_num, int state){                 //give ...
        the correct vertex the correct state, N.setStates(vertexNum, stateValue);
        if(this.number == vertex_num){                                   ...
            //checks if we are looking at the correct vertex
            this.PStates.add(state);                                     ...
                //gives the vertex we are looking at, the state we want, PStates ...
                is a LinkedHashSet
            return true;                                                 //then ...
                we are done
        }
        for(Network child: children){                                    //if ...
            we are not looking at the correct vertex, look at the children of ...
            this vertex
            boolean cb = child.setStates(vertex_num, state);             ...
                //check if we want the children
            if(cb){                                                      //loop ...
                through childeren until we found the correct vertex with ...
                corresponding vertex number, this is an iteration
                return true;                                             //we ...
                    are done
            }
        }
        return false;
    }


    public Network getVertex(int vertex_num) {
        if(this.number == vertex_num) {
            return this;
        }
        for (Network child : children) {
            Network v = child.getVertex(vertex_num);
            if(v != null) {
                return v;
            }
        }
        return null;
    }

    public void suppress() {
        for (Network child : children) {
            child.suppress();
            if (child.children.size() == 1 && child.parents.size() == 1) {
                // indegree-1 outdegree-1
                // suppress
                Network grandchild = child.children.elementAt(0);
                children.setElementAt(grandchild, children.indexOf(child));
                grandchild.parents.setElementAt(this, ...
                    grandchild.parents.indexOf(child));
            }
        }
    }

    public static Network newick2netwerk(String newick, Vector<Network> ...
        reticulations) {
        int lastclosepar = newick.lastIndexOf(")");
        int lasthash = newick.lastIndexOf("#");
        int lastcolon = newick.lastIndexOf(":");

        // get rid of weights
        if (lastcolon > lastclosepar & lastcolon > lasthash) {
            return newick2netwerk(newick.substring(0, lastcolon), reticulations);
        }

        Network N = new Network();

        if (newick.startsWith("(")) {
            if (lastclosepar < newick.length() - 1 && newick.charAt(lastclosepar ...
                + 1) == '#') {
                // a new reticulation
                reticulations.add(N);
                N.retNum = new Integer(newick.substring(lastclosepar + 3, ...
                    newick.length()));
                Network child = newick2netwerk(newick.substring(0, lastclosepar ...
                    + 1), reticulations);
                N.children.add(child);
                child.parents.add(N);
                return N;
            } else {
                // split vertex
                int openpar = 0;
                int closepar = 0;
                int start = 1;
                Vector<String> childrenNewick = new Vector();
                for (int i = 0; i < newick.length(); i++) {
                    if (newick.charAt(i) == '(') {
                        openpar++;
                    }
                    if (newick.charAt(i) == ')') {
                        closepar++;
                    }
                    if ((openpar == closepar + 1) && (newick.charAt(i) == ',')) {
                        childrenNewick.add(newick.substring(start, i));
                        start = i + 1;
                    }
                    if (i == newick.length() - 1) {
                        childrenNewick.add(newick.substring(start, i));
                    }
                }

                for (String childNewick : childrenNewick) {
                    Network child = newick2netwerk(childNewick, reticulations);
                    N.children.add(child);
                    child.parents.add(N);
                }
                return N;
            }

        } else {
            if (newick.startsWith("#H")) {
                // a reticulation
                N.retNum = Integer.parseInt(newick.substring(2, newick.length()));
                for (Network reticulation : reticulations) {
                    if (reticulation.retNum == N.retNum) {
                        // an existing reticulation
                        N.children.add(reticulation);
                        reticulation.parents.add(N);
                        return N;
                    }
                }
            } else {
                // a leaf
                if (newick.contains("#")) {
                    // a reticulation leaf
                    int hash = newick.indexOf("#");
                    N.retNum = Integer.parseInt(newick.substring(hash + 2, ...
                        newick.length()));

                    // check if we've already seen this reticulation
                    for (Network reticulation : reticulations) {
                        if (reticulation.retNum == N.retNum) {
                            // an existing reticulation
                            N.children.add(reticulation);
                            reticulation.parents.add(N);
                            return N;
                        }
                    }

                    // apparently this is a new reticulation
                    reticulations.add(N);
                    Network child = new Network();
                    child.isLeaf = true;
                    String lab = newick.substring(0, hash);
                    child.label = lab;
                    N.children.add(child);
                    child.parents.add(N);
                    TAXON_LABELS.add(lab);
                } else {
                    // a normal leaf
                    N.isLeaf = true;
                    N.label = newick;
                    TAXON_LABELS.add(newick);
                }
            }
        }
        return N;
    }

    public String toString() {
        String output;
        // returns eNewick string of the network
```

```
        if (isLeaf) {
            return label + ":1.0";
        }

        String childString1 = ((Network) children.elementAt(0)).toString();

        if (parents.size() > 1) {
            // reticulation
            if (seen) {
                output = "#H" + retNum;
            } else {
                MAX_RET++;
                retNum = MAX_RET;
                seen = true;
                if (childString1.startsWith("(")) {
                    output = childString1 + "#H" + retNum;
                } else {
                    output = "(" + childString1 + ")" + "#H" + retNum;
                }
            }
            return output;
        }

        output = "(" + childString1;

        for (int i = 1; i < children.size(); i++) {
            output += "," + ((Network) children.elementAt(i)).toString();
        }
        output += "):1.0";
        if (parents.isEmpty()) {
            output += ";";
        }
        return output;
    }

    public void computeRetEdgeSupport() {
        retEdgeSupport = new Vector();
        for(Network parent : parents) {
            retEdgeSupport.add(0.0);
        }
        if (parents.size() > 1) {
            for (int i = 0; i < softwiredStates.size(); i++) {
                char c = softwiredStates.elementAt(i);
                int numParentsWithC = 0;
                for(Network parent : parents) {
                    char d = parent.softwiredStates.elementAt(i);
                    if (c == d) {
                        numParentsWithC++;
                    }
                }
                //double score = 1.0 / (numParentsWithC * softwiredStates.size());
                for (int p = 0; p < parents.size(); p++) {
                    Network parent = parents.elementAt(p);
                    char d = parent.softwiredStates.elementAt(i);
                    if (c == d & numParentsWithC == 1) {
                    //if (c == d) {
                        retEdgeSupport.setElementAt(retEdgeSupport.elementAt(p) + ...
                            1, p);
                    }
                }
            }
            // normalise
            int total = 0;
            for (int p = 0; p < parents.size(); p++) {
                total += retEdgeSupport.elementAt(p);
            }
            if (total != 0) {
                for (int p = 0; p < parents.size(); p++) {
                    retEdgeSupport.setElementAt(retEdgeSupport.elementAt(p) / ...
                        total, p);
                }
            }
        }
        // recurse
        for(Network child : children) {
            child.computeRetEdgeSupport();
        }
    }

    public int setCharacterStates(Vector<String> taxa, ...
        Vector<Vector<Character>> states, int state_index) {
        // returns the number of states
        if (isLeaf) {
            for (String taxon : taxa) {
                if (label.equals(taxon)) {
                    char s = ...
                        states.elementAt(taxa.indexOf(taxon)).elementAt(state_index);
                    if (s != '-' & s != '?') {
                        if (!STATE_LABELS.contains(s)) {
                            STATE_LABELS.add(s);
                        }
                        this.state = STATE_LABELS.indexOf(s);
                    } else {
                        this.state = GAP;
                    }
                }
            }
        } else {
            for (Network child : children) {
                child.setCharacterStates(taxa, states, state_index);
            }
        }
        return STATE_LABELS.size();
    }

    public void resetSeen() {
        seen = false;      if (!isLeaf) {
            for (Network child : children) {
                child.resetSeen();
            }
        }
    }

    public void cleanNetwork() {
        MAX_RET = 0;
        seen = false;
```

```
        retNum = -1;
        number = 0;
        if (!isLeaf) {
            for (Network child : children) {
                child.cleanNetwork();
            }
        }
    }


    //for states in LinkesHashSet
    public int getScore(boolean softwired) {              //input is N: ...
        N.getScore(true)
        // returns the parsimony score
        int s_s = 0;                                  // softwired score, also ...
            weight of lineage function
        int parentCount = 0;                          //will count number of ...
            states a parent of v has , method size did not work
        int stateCount = PStates.size();                          // will count ...
            the number of states v has
        /*
        if (this.state == GAP) {                       //this was some kind of ...
            check, dont know if we will need this.
            return 0;
        }*/

        //first part of lineage function: weight is 0 if v is root
        if(!this.isRoot) {                                              //if this is ...
            the root, weight and score is 0, so we do nothing with the root and ...
            root has no parents
        //second part: weight is inf if v ≠pN and |f(v)| > union(over u in ...
            par(v))[ |f(u)| ]
          /* Iterator<Integer> itr1 = PStates.iterator();
            while(itr1.hasNext()){                                      //loops ...
                through elements of HashSet and thus counts the number of states ...
                v has
              stateCount++;                                            //remember ...
                  that there are no duplicates in a LinkedHashSet
            }
           for(Integer u: PStates){
                stateCount++;
            }*/
            for(Network parent : parents){
                parentCount += parent.PStates.size();
            }
//          for (Network parent : parents) {                            //loops ...
    through parents of v
  //            Iterator<Integer> itr2 = parent.PStates.iterator(); //makes ...
      iterator that goes through the LinkedHashSet of the parent
    //          while(itr2.hasNext()){                                  //loops ...
        through elements: if the iterator hasn't a next element, this is the ...
        last element
      //            parentCount++;                                      //counts ...
          the number of parents in the HashSet of the parent
       //       }
        // }                                                            //we are ...
            going to check whether v has more states than his parents together
```

```
if(stateCount > parentCount){                           // if |f(v)| > ...
    union(over u in par(v))[ |f(u)| ], then w is inf
    s_s = -999999;                                      //weight would ...
        be inf, but with a negative integer we also show that ...
        something is wrong
    System.out.println("the weight of vertex number " + this.number ...
        + " is infinity"); //let us know which vertex goes wrong
}
//third part: weight of the lineage function is |f(v) \ sum(over u ...
    in par(v))[ f(u) ] |
//Iterator<Integer> itr3 = PStates.iterator();
//while(itr3.hasNext()){                           //loop through ...
    elements of f(v), check per element whether Uf(u) has it
Iterator<Integer> itr5 = PStates.iterator();       //check the ...
    elements of PStates
while (itr5.hasNext()){
    System.out.println("elements in PStates of vertex " + ...
        this.number + " are: "+ itr5.next() + ", so the size is "+ ...
        PStates.size() );
}
for(Integer w : PStates){                           //loop through ...
    elements of this.PStates
    //we will put the states of the parent in an arraylist, so that ...
        we can compare these states with state w
    int statesOfParent = 0;                          //counts the ...
        number of states of a parent
    boolean firstParent = true;
    ArrayList statesOfBothParents = new ArrayList();        //we are ...
        going to put states of all parents in this arraylist
    for(Network parent : parents){                  //loop through ...
        parents of this(v)
        //Iterator<Integer> itr4 = parent.PStates.iterator();
        if(firstParent){                             //check if you ...
            have first parent
            for(Integer u: parent.PStates){          //check for first ...
                parent how much states it has
                statesOfParent++;                    //counts the ...
                    number of states of a parent

            }
            firstParent = false;                     //we have had the ...
                first parent
            statesOfBothParents.addAll(parent.PStates);       //add ...
                all states of the first parent in statesOfBothParents
        }
        if(!firstParent){                            //if it is not ...
            the first parent, add all the states from the second ...
            parent in statesOfBothParents, from index statesOfParent
            statesOfBothParents.addAll(statesOfParent,parent.PStates); ...
                //add all states of the first parent, beginning from ...
                the first empty index.
        }

    }
    boolean oneTheSame = false;                          //we will ...
        compare the state w with all the states of the parents
    for(int s = 0; s < statesOfBothParents.size(); s++ ){   //loop ...
        through all the states of all the parents
```

```
                if(w == statesOfBothParents.get(s)){ ...
                                                  //checks whether one of the ...
                    parents has a state s that is the same as state w
                    oneTheSame = true ;                      // says one of ...
                        the parents has one of the same states as w
                }
            }
            if(!oneTheSame){
                s_s++;
            }
        }
    }
    this.seen = true;
    for (Network child : children) {
        if (!child.seen) {
            int childscore = child.getScore(softwired);
            s_s += childscore;
        }
    }
    System.out.println("in getScore, it calculates the following parsimony ...
        scores: "+ s_s);
    return s_s;
}


public Vector<String> toDot(boolean nolabels, boolean nostates, boolean ...
    softwired) {
    // returns vector with the network in dot format
    Vector<String> out = new Vector();
    out.add("strict digraph G {");
    int[] num = new int[1];
    num[0] = 1000;
    this.cleanNetwork();
    out.addAll(this.nodes2dot(num, nolabels, nostates, softwired));
    out.addAll(this.arcs2dot(softwired));
    out.add("}");
    this.cleanNetwork();
    return out;
}

public Vector<String> nodes2dot(int num[], boolean nolabels, boolean ...
    nostates, boolean softwired) {
    Vector<String> out = new Vector();
    if (number != 0) {
        return out; // already visited
    }
    if (isLeaf) {
        // this is a leaf
        number = TAXON_LABELS.indexOf(label) + 1;
        //System.out.println(number + " [shape=circle, width=0.3, label=\"" ...
            + label + " (" + STATE_LABELS.elementAt(state) + ")\"" + "];");
        //System.out.println(number + " [shape=circle, width=0.3, label=\"" ...
            + STATE_LABELS.elementAt(state) + "\"" + "];");
        String slabel = "";
        if (softwired) {
            for (char cstate : softwiredStates) {
                if (cstate == GAP) {
                    slabel += '-';
                } else {
                    slabel += cstate;
                }
            }
        } else {
            for (char cstate : hardwiredStates) {
                if (cstate == GAP) {
                    slabel += '-';
                } else {
                    slabel += cstate;
                }
            }
        }
        if (nolabels) {
            if (!nostates) {
                out.add(number + " [shape=box, width=0.2, label=\"" + slabel ...
                    + "\"" + "];");
            } else {
                out.add(number + " [shape=point];");
            }
        } else {
            if (!nostates) {
                out.add(number + " [shape=box, width=0.2, label=\"" + label ...
                    + "\\n" + slabel + "\"" + "];");
            } else {
                out.add(number + " [shape=none, label=\"" + label + "\"];");
            }
        }
    } else {
        number = num[0];
        //System.out.println(number + " [shape=point];");
        //System.out.println(number + " [shape=circle, width=0.3, label=\"" ...
            + STATE_LABELS.elementAt(state) + "\"" + "];");
        String slabel = "";
        if (softwired) {
            for (char cstate : softwiredStates) {
                slabel += cstate;
            }
        } else {
            for (char cstate : hardwiredStates) {
                slabel += cstate;
            }
        }
        if (!nostates) {
            out.add(number + " [shape=box, width=0.2, label=\"" + slabel + ...
                "\"" + "];");
        } else {
            out.add(number + " [shape=point];");
        }
        for (Network child : children) {
            num[0]++;
            out.addAll(child.nodes2dot(num, nolabels, nostates, softwired));
        }
    }
    return out;
}

public Vector<String> arcs2dot(boolean softwired) {
```

```
// returns the arcs in dot format
Vector<String> out = new Vector();
for (Network child : children) {
    int intlabel = -1;
    double doublelabel = -1;
    /*
     if(child.parents.size()>1 & softwired & SourceCodeTest.USE_CPLEX) {
     // find the fraction of characters using this edge
     int index = child.parents.indexOf(this);
     doublelabel = 0.0;
     for(int s = 0; s < numchar; s++) {
     Boolean b = child.retEdgeUsed.elementAt(s).elementAt(index);
     if(b!=null && b) {
     doublelabel++;
     }
     }
     doublelabel = doublelabel / numchar;
     * */
    if (child.parents.size() > 1 & softwired) {
        // find the fraction of characters that could be inherited over ...
            this edge
        /*
        doublelabel = 0;
        for (int i = 0; i < softwiredStates.size(); i++) {
            if ((softwiredStates.elementAt(i) == ...
                child.softwiredStates.elementAt(i)) | ...
                (child.softwiredStates.elementAt(i) == '-')) {
                doublelabel++;
            }
        }
        doublelabel = doublelabel / numchar;
        */
        doublelabel = ...
            child.retEdgeSupport.elementAt(child.parents.indexOf(this));
        doublelabel = Math.round(doublelabel*10.0) / 10.0;
    } else {
        // find the number of changes on this edge
        intlabel = countChanges(this, child, softwired);
    }
    if (doublelabel != -1) {
        out.add(number + " -> " + child.number + "[color=blue,label=" + ...
            doublelabel + "]");
    } else if (softwiredStates.size() == 1 | hardwiredStates.size() == ...
        1) {
        if (intlabel > 0) {
            out.add(number + " -> " + child.number + "[color=red]");
        } else {
            out.add(number + " -> " + child.number + "[color=black]");
        }
    } else {
        //out.add(number + " -> " + child.number + ...
            "[colorscheme=dark28,color=" + (changes+1) + "]");
        if (intlabel > 0) {
            out.add(number + " -> " + child.number + "[color=red,label=" ...
                + intlabel + "]");
        } else {
            out.add(number + " -> " + child.number + "[color=black]");
        }
```

```
            }
        }
        seen = true;
        for (Network child : children) {
            if (!child.seen) {
                out.addAll(child.arcs2dot(softwired));
            }
        }
        return out;
    }

    public int countChanges(Network vertex, Network child, boolean softwired) {
        int changes = 0;
        if (softwired) {
            for (int i = 0; i < child.softwiredStates.size(); i++) {
                char childstate = child.softwiredStates.elementAt(i);
                if (childstate == '-') {
                    continue;
                }
                boolean change = true;
                for (Network parent : child.parents) {
                    if (parent.softwiredStates.elementAt(i).equals(childstate)) {
                        change = false;
                    }
                }
                if (change) {
                    changes++;
                }
            }
        } else {
            for (int i = 0; i < vertex.hardwiredStates.size(); i++) {
                char mystate = vertex.hardwiredStates.elementAt(i);
                char childstate = child.hardwiredStates.elementAt(i);
                if (childstate == '-') {
                    continue;
                }
                if (childstate != mystate) {
                    changes++;
                }
            }
        }
        return changes;
    }

    public Vector<Network> getVertices(int num[]) {
        Vector out = new Vector();
        if (number != 0) {
            return out; // already visited
        }
        if (isLeaf) {
            // this is a leaf
            number = TAXON_LABELS.indexOf(label) + 1;
            out.add(this);
        } else {
            number = num[0];
            out.add(this);
            for (Network child : children) {
                num[0]++;
                out.addAll(child.getVertices(num));
            }
        }
        return out;
    }

    public Vector<Network> getReticulations(int num[]) {
        Vector out = new Vector();
        if (number != 0) {
            return out; // already visited
        }
        if (parents.size() > 1) {
            out.add(this);
        }
        if (isLeaf) {
            number = TAXON_LABELS.indexOf(label) + 1;
        } else {
            number = num[0];
            for (Network child : children) {
                num[0]++;
                out.addAll(child.getReticulations(num));
            }
        }
        return out;
    }

    public Vector<Vector<Network>> getEdges() {
        Vector out = new Vector();
        for (Network child : children) {
            Vector edge = new Vector();
            edge.add(this);
            edge.add(child);
            out.add(edge);
        }
        seen = true;
        for (Network child : children) {
            if (!child.seen) {
                out.addAll(child.getEdges());
            }
        }
        return out;
    }

    public void roundStates(boolean softwired) {
        // rounding procedure for states
        if (state == -1) {
            if (softwired) {
                // for the softwired parsimony score, we make the state equal to ...
                    the state of at least one parent
                // we choose the parent whose state is equal to the state of a ...
                    maximum number of children
                if (parents.size() > 0) {
                    Network bestParent = null;
                    int bestnum = 0;
                    for (Network parent : parents) {
                        int num = 0;
                        for (Network child : children) {
                            if (parent.state == child.state) {
```

```
                        num++;
                    }
                }
                if (num ≥bestnum) {
                    bestnum = num;
                    bestParent = parent;
                }
            }
            state = bestParent.state;
        } else {
            // the root we give state 0 (if it's not yet integral)
            state = 0;
        }
    } else {
        // for the hardwired parsimony score, we round to the nearest ...
            integer
        state = (int) Math.rint(stateDouble);

        // might want to do something more clever here

    }
}
for (Network child : children) {
    child.roundStates(softwired);
}
}

public void clearStates() {
    state = -1;
    stateDouble = -1;
    //if (!STATE_LABELS.isEmpty()) {
    //    STATE_LABELS = new Vector();
    //}
    for (Network child : children) {
        child.clearStates();
    }
}

public void clearInternalStates() {
    if(!isLeaf) {
        state = -1;
        stateDouble = -1;
    }
    //if (!STATE_LABELS.isEmpty()) {
    //    STATE_LABELS = new Vector();
    //}
    for (Network child : children) {
        child.clearInternalStates();
    }
}



public void finaliseStates(int index, boolean softwired){  //parentalStates ...
    will be a vector with a string (a set) of characters per state_index
    seen = true;                                            //we are going ...
        to iterate through network N, so we need to know if this vertex is ...
        already seen
    String s = new String();                       //create string, we ...
        are going to put the characters in this string
    boolean first = true;                          //first element of ...
        string does not need a ","
    ArrayList ArrayPStates = new ArrayList();      //make new arraylist
    ArrayPStates.addAll(PStates);                  ////Put elements of ...
        LinkedHashSet in a ArrayList, so we can loop through elements

    for(int i = 0; i<ArrayPStates.size(); i++){    //loop through ...
        elements of ArrayPStates
        int intObj = Integer.parseInt(ArrayPStates.get(i).toString()); //we ...
        need to make int's of the Integer Objects in ArrayPStates, for ...
        the STATE_LABELS.elementAt method
        if(first){                                           //only for ...
        the first, we don't need a ","
        s += STATE_LABELS.elementAt(intObj);             //add the ...
            actual characters of state_labels to the string s
        first = false;
        } else{
        s += ", " + STATE_LABELS.elementAt(intObj);         //add the ...
        corresponding character in s
        }
    }
    if(softwired){                                           //bit ...
        unnecessary
        if(index ≥parentalStates.size()){                    //if index ...
        is bigger than the size, we can just add s at the end of ...
        parentalStates. When a new string needs to be placed at this ...
        index, the current string will move one index forward.
        parentalStates.addElement(s);                     // put ...
            the string s in the vector<string> on the correct spot ...
            (state_index)
        }
        else{
        parentalStates.setElementAt(s, index);
        }
    }
}


public Network cloneLeaf() {
    Network leaf = new Network();
    leaf.label = label;
    leaf.isLeaf = true;
    return leaf;
}

public Network getTree(int index) {
    // returns tree obtained by using the reticulation edge specified by ...
        index for each reticulation
    Network tree = new Network();
    if (isLeaf) {
        return (Network) this.cloneLeaf();
    }
    for (Network child : children) {
        if (child.parents.size() == 1 || child.parents.indexOf(this) == ...
        index) {
        Network treeChild = child.getTree(index);
        tree.children.add(treeChild);
        treeChild.parents.add(tree);
        }
    }
    return tree;
}
}
```