NetRAX

Maximum-Likelihood Phylogenetic Network Inference without Incomplete Lineage Sorting

Sarah Lutteropp[1], Celine Scornavacca[2], Alexey M. Kozlov[1], Benoit Morel[1], and Alexandros Stamatakis[1,3]

[1] The Exelixis Lab, Scientific Computing Group, Heidelberg Institute for Theoretical Studies, Heidelberg 69118, Germany
[2] Institut des Sciences de l'Évolution Université de Montpellier, CNRS, IRD, EPHE Place Eugène Bataillon 34095, Montpellier Cedex 05, France
[3] Department of Informatics, Institute of Theoretical Informatics, Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany

Abstract. We present NetRAX, a tool for maximum-likelihood inference of phylogenetic networks in the absence of incomplete lineage sorting. Our software is available under the GNU GPL v3 license at https://github.com/lutteropp/NetRAX.

It does not model it, but we can still use it.

1 Acknowledgements

This work was partially funded by the Klaus Tschira foundation.

or negligible.

2 Introduction and Related Work

We can not always describe the evolution of species via a phylogenetic tree. For example, there is hybridization among plants and some bacteria can acquire genes from other bacterial species. In these cases, a phylogenetic network is better suited to explain evolutionary relationships.

2.1 What is a Phylogenetic Network?

In literature, there exists a plethora of different phylogenetic network definitions. Here, we focus on phylogenetic networks which we characterize by a set of displayed trees. We assume that incomplete lineage sorting is not present in the dataset.

A binary phylogenetic network N is a single-source, directed, acyclic graph. We call its source node the *root* node of N. There are three types of nodes in a binary phylogenetic network:

Apart from the root,

- Internal tree nodes with 1 incoming edge and 2 outgoing edges,
- reticulation nodes with 2 incoming edges and 1 outgoing edge, and
- leaf nodes with one incoming edge and no outgoing edges.

Each edge in a phylogenetic network has a branch length and a probability. The incoming edges of a reticulation node (called hybridization edges) have inheritance probabilities assigned to them which must sum up to 1.0. The probability of a non-hybridization edge is 1.0.

Often in phylogenetic inference, a multiple sequence alignment (MSA) A has multiple partitions $\mathcal{A}_1, \ldots, \mathcal{A}_p$. A partition consists of a set of sites for which we believe that they evolved together (e.g., sites of a single gene), following the same evolutionary process. In NetRAX, we allow for branch lengths and inheritance probabilities being either linked among all partitions (this means, all partitions share the same branch lengths and probabilities), or unlinked branch lengths. By default, we use linked branch lengths.

I don't think you should say this here It is too early Maybe in section 4 ?

We assume that neither incomplete lineage sorting (ILS) nor recombination has taken place in the set of sites within a partition A_i. We make this assumption because it reduces the computational complexity, and ILS is not an issue if species did not split recently.

We characterize a phylogenetic network by its set of induced displayed trees (see Figures 2.1 and 2.1).

Please define

A phylogenetic network on a set of n taxa has exactly n leaves. We characterize each displayed tree by the probabilities of the hybridization edges taken in order to obtain the tree. The probability $P(T|N)$ of a displayed tree T in a phylogenetic network N is the product of the probabilities of the hybridization edges that display T.

We can improve this later (let's wait for the "related work" section)

It is not clear that you want to compute the probability wrt an alignment. What I would do is:
- say what is a network [done already]
- say what it describe [reticulate evolution] and how [via displayed trees, what they are, how to get them and their probab.]

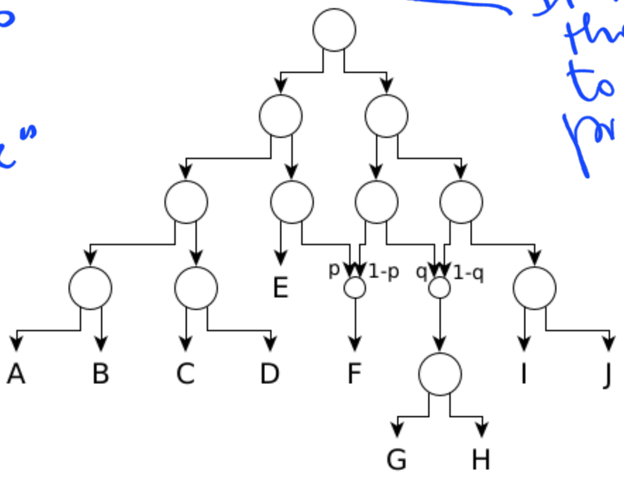


Fig. 1: A phylogenetic network with two reticulation nodes.

I would like the figure to be drawn

without the big rounds (e.g. the figures in the paper with Daniel Huson. I can draw them)

But it is not urgent at all.

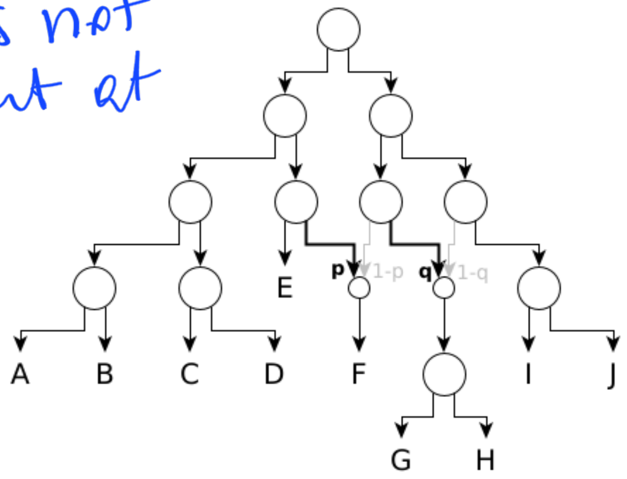


Fig. 2: A displayed tree in a phylogenetic network. The probability of the shown displayed tree is the product of its taken reticulation parents' probabilities, which is $p * q$.

2.2 Related Work

Past approaches for inferring phylogenetic networks include NEPAL [8], PhyloDAG [10], and PhyloNet [14].

In PhyloNet,

Celines moves papers ([5] and cite) Empirical dataset paper ([6])

Raxml paper [9] libpll [1] and pll-modules [2] MPFR C++ [7]

non-treelike evolutionary events

3 Preliminaries

3.1 Multiple Sequence Alignment

TODO: Explain what is a MSA, what is a partitioned MSA, what are MSA sites and patterns

3.2 Phylogenetic Tree

TODO: Explain what is a phylogenetic tree

3.3 Felsenstein Algorithm

TODO: Explain Felsenstein algorithm and CLVs

3.4 Newton-Raphson Optimization

TODO: Explain idea behind Newton-Raphson optimization

3.5 Brent Optimization

TODO: Explain idea behind Brent optimization

In this definition, branch lenghts are always linked. Maybe given the more general definition and then say, after eq.(4), that our default is linked? This is needed for section 5.

4 Phylogenetic Network Likelihood Model

We characterize a phylogenetic network by its set of displayed trees, because we assume that incomplete lineage sorting has not taken place in the dataset.

4.1 Recap: Tree Loglikelihood

Definition 1. *Site-based Phylogenetic Tree Loglikelihood*

Let $T = (V, E)$ be a phylogenetic tree. Let A be a partitioned MSA with partitions $\mathcal{A}_1, \ldots, \mathcal{A}_p$. Let ϑ be the parameter vector, storing branch lengths and likelihood model parameters. The likelihood of T given A is:

and θ

~~$L(T|A) = \prod_{i=1}^{p} L(T|\mathcal{A}_i, \vartheta) = \prod_{i=1}^{p} \prod_{s \in \mathcal{A}_i} L(T|s, \vartheta)$~~ (1)

In order to avoid numerical issues, one usually focuses on tree loglikelihood:

$$\log L(T|\overset{\theta}{A}) = \sum_{i=1}^{p} \log L(T|\mathcal{A}_i, \vartheta) = \sum_{i=1}^{p} \sum_{s \in \mathcal{A}_i} \log L(T|s, \vartheta). \tag{2}$$

?

Definition 2 (Likelihood of a Phylogenetic Tree). *Let $T = (V, E)$ be a phylogenetic tree. Let A be a partitioned MSA with partitions $\mathcal{A}_1, \ldots, \mathcal{A}_p$. Let ϑ be the parameter vector, storing branch lengths and likelihood model parameters. The likelihood of T given A is:*

$$L(T|A, \vartheta)) = \prod_{i=1}^{p} L(T|\mathcal{A}_i, \vartheta) = \prod_{i=1}^{p} \prod_{s \in \mathcal{A}_i} L(T|s, \vartheta)$$

4.2 Network Loglikelihood

Use this notation in sec. 2.1

Definition 3 (Probability of a Displayed Tree). *Let $N = (V, E)$ be a phylogenetic network and T be a displayed tree of N. Let E_r be the set of reticulation edges that need to be taken in order to obtain/generate T. Let $P(e)$ be the probability of an edge e. The probability of T in N is then:*

$$P(T|N) = \prod_{e \in E_r} P(e).$$

Let $N = (V, E)$ be a phylogenetic network with set of displayed trees $\mathcal{T}(N)$. Let A be a partitioned MSA with partitions $\mathcal{A}_1, \ldots, \mathcal{A}_p$. Let ϑ be the parameter vector, storing network branch lengths and other likelihood model parameters.

We define the likelihood of a phylogenetic network as the product over the per-partition likelihoods:

Say that this is possible since we consider partitions as independent

$$L(N|A, \vartheta) = \prod_{i=1}^{p} L(N|\mathcal{A}_i, \vartheta).$$

To avoid numerical underflow, we focus on the network loglikelihood:

$$\log L(N|A, \vartheta) = \sum_{i=1}^{p} \log L(N|\mathcal{A}_i, \vartheta).$$

We assess and implement two versions for computing the loglikelihood on partitioned phylogenetic networks. They both aggregate over the loglikelihood of the phylogenetic trees displayed by the network:

1. **Weighted Average Version**:

$$\log L(N|\mathcal{A}_i, \vartheta) = \log \left(\sum_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N) \right). \tag{3}$$

2. **Best Tree Version**:

$$\log L(N|\mathcal{A}_i,\vartheta) = \log\left(\max_{T\in\mathcal{T}(N)} L(T|\mathcal{A}_i,\vartheta) * P(T|N)\right). \tag{4}$$

In the weighted average version, the likelihood of a network for a given partition is the weighted average over the displayed tree likelihoods. We use the sum here, because the probability of event A or B to occur is the sum of the probability of observing A and the probability of observing B. The weighted average we have in here is thus the expected value, if we treat each displayed tree as a statistical event. To avoid numerical problems, we use arbitrary-precision arithmetics (using MPFR C++ (`http://www.holoborodko.com/pavel/mpfr/`)) to compute $L(N|\mathcal{A}_i)$ from the displayed tree likelihoods.

We use libpll and pll-modules to compute displayed tree likelihoods via the standard Felsenstein pruning algorithm.

5 Branch Length Model

No need for a section here, a subsection is enough.

In its current implementation, NetRAX supports two branch length models: Under the linked branches model, we share the same set of branch lengths among *all* partitions. Under the unlinked branches model, each partition has its own independent set of branch lengths.

What about scaled? Why don't we support it? We should say something here

The branch length model choice has an effect on which reticulations we can recover. Figure 5 shows an example network with a reticulation that is impossible to infer under the unlinked branches model.

Very convincing!

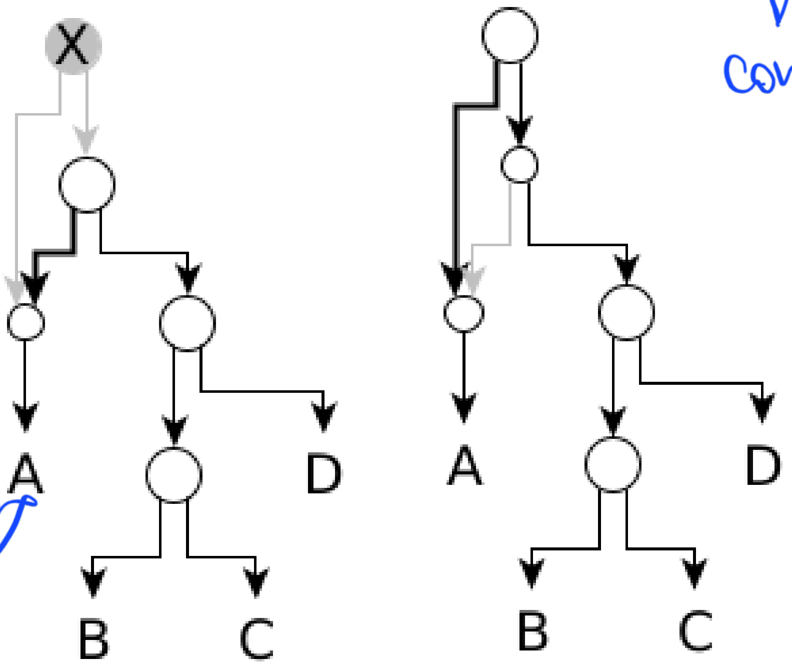


Fig. 3: Two displayed trees in a phylogenetic network. Both displayed trees induce the same topology after collapsing simple paths. They only differ in some branch lengths.

I think that the CLVS part of this section has to be clarified. It is not clear how these CLVs are computed on the network and how they are used to compute the actual likelihood of the network. The main ideas are there already, but need to be expanded and clarified. An example on a small network is necessary!

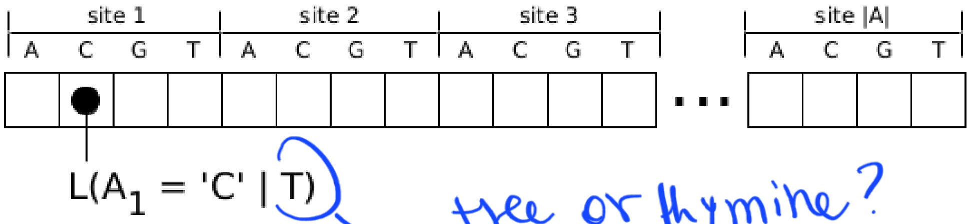
6 Computing the Likelihood of a Phylogenetic Network

Key ideas:

- Use per-node displayed tree CLVs for faster incremental computation of network loglikelihood.
- Share per-node CLVs among displayed trees where-ever possible. Summarize reticulation parent choices that lead to the same displayed tree.

In order to compute the pyhylogenetic network loglikelihood, we first need to compute the loglikelihoods and probabilities of all trees displayed by the network. To avoid numerical underflow, we use the MPFR C++ wrapper [7] to compute the network loglikelihood given the displayed tree likelihoods and displayed tree probabilities.

We use the Felsenstein pruning algorithm [4] as implemented in libpll for computing the loglikelihood of a phylogenetic tree. For this, libpll stores a conditional likelihood vector (CLV) for each node in the tree (see `https://github.com/xflouris/libpll/wiki/Computing-the-likelihood-of-a-tree`). A CLV stores the per-site likelihoods for a subtree rooted at a given node (see Figure 6). ?



tree or thymine? Not very clear.

Fig. 4: A Conditional Likelihood Vector (CLV) for a node in a phylogenetic tree T and a MSA partition A.

6.1 Iteration Order

Instead of processing one displayed tree after another, we conduct a bottom-up traversal (reversed topological sort) of the phylogenetic network. For each node v we visit, we update the CLVs for each of the displayed subtree topologies present at the node v. We represent each of these displayed subtree topologies by a so called ReticulationConfigSet. The ReticulationConfigSet stores all possible ways to select reticulation parents which result in the given subtree topology. Note that multiple reticulation parent choices can lead to the same underlying displayed tree topology, because we effectively collapse simple paths and prune dead nodes when computing the likelihood.

6.2 Incremental Likelihood Computation

To avoid recomputing all CLVs from scratch after we modified the network topology, we maintain a clv_valid array for all nodes and all displayed trees. This array tells us which CLVs were not affected by changing the network. Analogously, – as already provided by libpll –, we keep a pmatrix_valid array which stores information about which transition probability matrices P for which branch length values we need to recompute.

Say when we do not need to recompute.

What is this?

6.3 Handling Single-child Nodes

When setting the reticulation node parents (activating one parent, deactivating the other parent) in order to display a particular tree, we end up with a tree-like structure which is not strictly bifurcating. After choosing the active reticulation parents, each reticulation only has one active parent and one active child. However, libpll requires a strictly bifurcating tree for its likelihood computations. In order to handle these paths and to avoid complicated code that temporarily merges nodes and branches together, we use the following trick: We introduce a new, “fake” node. The CLV of this “fake” node consists of only 1-s. A skipped node is a node that has only one active child. Whenever we encounter a skipped node, we temporarily add a second child to it, with branch length zero and this all-1s “fake” CLV.

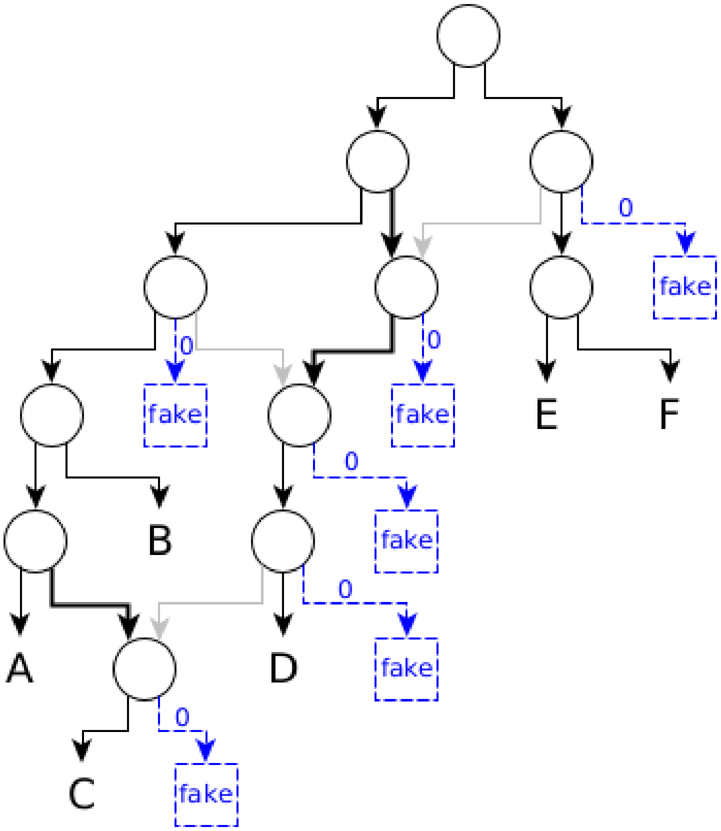


Fig. 5: A displayed tree which contains nodes that have only one active child. For likelihood computation using libpll, we add an imaginary fake node with branch length zero as the second active child of all one-child nodes.

6.4 Skipping Dead Nodes

add ref.

Depending on which reticulation parents we select, it can happen that a large part of the resulting displayed tree consists of dead nodes (see image below). A dead node is a non-leaf node that either has no active children, or is the only active child of a dead node, or a node with no active parent and only a single active child. We pre-detect such nodes in order to reduce the total number of required CLV update operatons. We also handle the case where we need to assign root node for a given displayed tree to another node, like in the situation presented in Figure 6.~~4~~

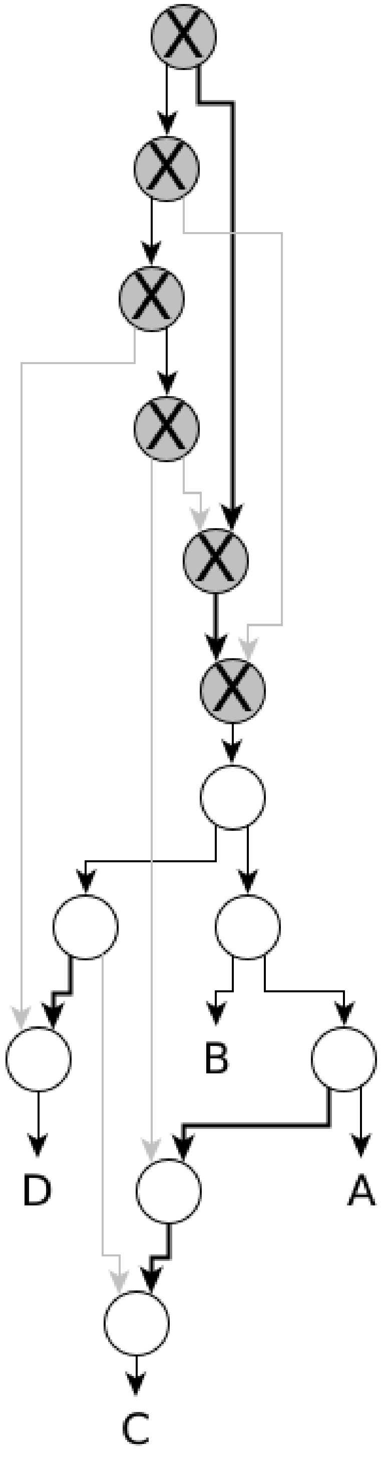
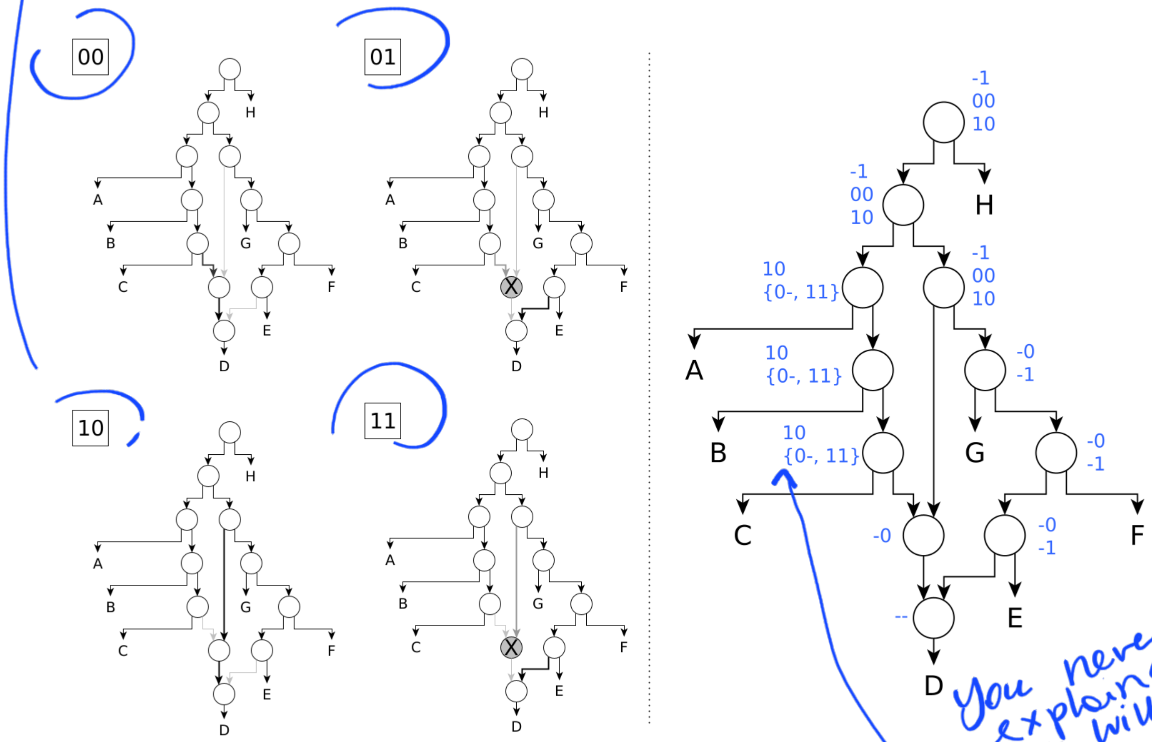


Fig. 6: Dead and therefore skipped nodes in a given displayed tree. In this figure, we mark dead nodes by the letter X. Note that the original virtual root node belongs to the set of dead nodes. For this displayed tree, we use the highest non-dead node as its temporary virtual root node.

explain the meaning of 00 01 10 11.

6.5 Sharing CLVs among Displayed Trees

Naively, one would require number_of_network_nodes * number_of_displayed_trees CLV vectors to calculate the likelihood of a network. However, there exist regions in a network that are exactly identical for several of the displayed trees. In this case, we can share per-node CLVs among displayed trees.



You never explained this, you will need to explain what these numbers are.

Fig. 7: A network an its displayed trees. On the right side, we see which reticulation configurations need their own CLV vector.

For each node v in a rooted phylogenetic network, we store as many CLVs as there are different displayed subtree topologies rooted at the v.

We use libpll and pll-modules to compute the displayed tree likelihoods. See Appendix 21 for the adaptations needed in order to use libpll and pll-modules for networks.

Processing a node When processing a node via a bottom-up traversal, we iterate over all combinations of reticulation configurations from the node's left and right child. If a combination is valid (this is, both children configurations agree on reticulation parent choice), we create a new NodeDisplayedTreeData at the node for this combination (TODO: Add pseudocode and explain NodeDisplayedTreeData).

Say why we need them.
Why not fusion with section 8?
Or move to the appendix?

7 Phylogenetic Network Loglikelihood Derivatives

7.1 Derivatives of Tree Loglikelihood and Tree Likelihood

The derivatives of tree loglikelihood are:

$$(\log L(T|\mathcal{A}_i,\vartheta))' = \sum_{s\in\mathcal{A}_i} \log L(T|s,\vartheta)' \quad = \sum_{s\in\mathcal{A}_i} \frac{L'(T|s,\vartheta)}{L(T|s,\vartheta)}. \tag{5}$$

$$\begin{aligned}(\log L(T|\mathcal{A}_i,\vartheta))'' &= \left(\sum_{s\in\mathcal{A}_i} \frac{L'(T|s,\vartheta)}{L(T|s,\vartheta)}\right)' \\ &= \sum_{s\in\mathcal{A}_i} \left(\frac{L'(T|s,\vartheta)}{L(T|s,\vartheta)}\right)' \\ &= \sum_{s\in\mathcal{A}_i} \frac{L''(T|s,\vartheta) * L(T|s,\vartheta) - (L'(T|s,\vartheta))^2}{(L(T|s,\vartheta))^2}.\end{aligned} \tag{6}$$

We compute the tree likelihood derivatives out of $L(T|\mathcal{A}_i,\vartheta)$, $(\log L(T|\mathcal{A}_i,\vartheta))'$ and $(\log L(T|\mathcal{A}_i,\vartheta))''$ as follows:

$$\begin{aligned}L'(T|\mathcal{A}_i,\vartheta) &= \left(\prod_{s\in\mathcal{A}_i} L(T|s,\vartheta)\right)' \\ &= \left(\prod_{s\in\mathcal{A}_i} L(T|s,\vartheta)\right) * \left(\sum_{s\in\mathcal{A}_i} \frac{L'(T|s,\vartheta)}{L(T|s,\vartheta)}\right) \\ &= L(T|\mathcal{A}_i,\vartheta) * (\log L(T|\mathcal{A}_i,\vartheta))'.\end{aligned} \tag{7}$$

$$\begin{aligned}L''(T|\mathcal{A}_i,\vartheta) &= \big(L(T|\mathcal{A}_i,\vartheta) * (\log L(T|\mathcal{A}_i,\vartheta))'\big)' \\ &= L'(T|\mathcal{A}_i,\vartheta) * (\log L(T|\mathcal{A}_i,\vartheta))' + L(T|\mathcal{A}_i,\vartheta) * (\log L(T|\mathcal{A}_i,\vartheta))''.\end{aligned} \tag{8}$$

7.2 Network Loglikelihood Derivatives

The first and second derivatives of a phylogenetic network loglikelihood (with respect to a changed branch length) are:

$$(\log L(N|A,\vartheta))' = \sum_{i=1}^{p} (\log L(N|\mathcal{A}_i,\vartheta))'. \tag{9}$$

$$(\log L(N|A,\vartheta))'' = \sum_{i=1}^{p} (\log L(N|\mathcal{A}_i,\vartheta))''. \tag{10}$$

Weighted Average Version

$$\begin{aligned}(\log L(N|\mathcal{A}_i,\vartheta))' &= \left(\log\left(\sum_{T\in\mathcal{T}(N)} L(T|\mathcal{A}_i,\vartheta) * P(T|N)\right)\right)' \\ &= \frac{\left(\sum_{T\in\mathcal{T}(N)} L(T|\mathcal{A}_i,\vartheta) * P(T|N)\right)'}{\sum_{T\in\mathcal{T}(N)} L(T|\mathcal{A}_i,\vartheta) * P(T|N)} \\ &= \frac{\sum_{T\in\mathcal{T}(N)} L'(T|\mathcal{A}_i,\vartheta) * P(T|N)}{\sum_{T\in\mathcal{T}(N)} L(T|\mathcal{A}_i,\vartheta) * P(T|N)}.\end{aligned} \tag{11}$$

Let $u := \sum_{T \in \mathcal{T}(N)} L'(T|\mathcal{A}_i, \vartheta) * P(T|N)$ and $v := \Big(\sum_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)\Big)$.

$$\begin{aligned} u' &= \left(\sum_{T \in \mathcal{T}(N)} L'(T|\mathcal{A}_i, \vartheta) * P(T|N)\right)' \\ &= \sum_{T \in \mathcal{T}(N)} L''(T|\mathcal{A}_i, \vartheta) * P(T|N). \end{aligned} \tag{12}$$

$$\begin{aligned} v' &= \left(\sum_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)\right)' \\ &= \sum_{T \in \mathcal{T}(N)} L'(T|\mathcal{A}_i, \vartheta) * P(T|N). \end{aligned} \tag{13}$$

Using equations 12 and 13, we obtain:

$$\begin{aligned} (\log L(N|\mathcal{A}_i, \vartheta))'' &= \left(\frac{\sum_{T \in \mathcal{T}(N)} L'(T|\mathcal{A}_i, \vartheta) * P(T|N)}{\sum_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)}\right)' \\ &= \left(\frac{u}{v}\right)' \\ &= \frac{u' * v - u * v'}{v^2} \\ &= \frac{\Big(\sum_{T \in \mathcal{T}(N)} L''(T|\mathcal{A}_i, \vartheta) * P(T|N)\Big) * \Big(\sum_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)\Big) - \Big(\sum_{T \in \mathcal{T}(N)} L'(T|\mathcal{A}_i, \vartheta) * P(T|N)\Big)^2}{\Big(\sum_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)\Big)^2}. \end{aligned} \tag{14}$$

Best Tree Version We know that

$$(\max f(x), g(x))' = \begin{cases} f'(x), & \text{if } f(x) > g(x) \\ g'(x), & \text{if } f(x) < g(x) \\ f'(x), & \text{if } f(x) = g(x) \text{ and } f'(x) = g'(x) \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{15}$$

We assume that for empirical use it holds that

$$(L(T_1|\mathcal{A}_i, \vartheta) * P(T_1|N) = L(T_2|\mathcal{A}_i, \vartheta) * P(T_2|N)) \Longleftrightarrow T_1 = T_2. \tag{16}$$

Let $\hat{T}$ be a displayed tree for which $L(T|\mathcal{A}_i, \vartheta) * P(T|N)$ is maximal among all $T \in \mathcal{T}(N)$. With the above assumption, we obtain

$$\begin{aligned} (\log L(N|\mathcal{A}_i, \vartheta))' &= \left(\log\left(\max_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)\right)\right)' \\ &= \frac{\big(\max_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)\big)'}{\max_{T \in \mathcal{T}(N)} L(T|\mathcal{A}_i, \vartheta) * P(T|N)} \\ &= \frac{L'(\hat{T}|\mathcal{A}_i, \vartheta) * P(\hat{T}|N)}{L(\hat{T}|\mathcal{A}_i, \vartheta) * P(\hat{T}|N)} = \frac{L'(\hat{T}|\mathcal{A}_i, \vartheta)}{L(\hat{T}|\mathcal{A}_i, \vartheta)} \\ &= \Big(\log L(\hat{T}|\mathcal{A}_i, \vartheta)\Big)'. \end{aligned} \tag{17}$$

$$(\log L(N|\mathcal{A}_i, \vartheta))'' = \Big(\log L(\hat{T}|\mathcal{A}_i, \vartheta)\Big)''. \tag{18}$$

8 Branch-Length Optimization in Phylogenetic Networks

Key ideas:

- Optimize branches using the Newton-Raphson method.
- Virtually re-root displayed trees during branch-length optimization, in order to save costly CLV recomputations.

When optimizing a branch length b in a phylogenetic network, our goal is to optimize the Phylogenetic Network Loglikelihood function. This is, we aim to find the best branch length assignment $\hat{b}$ in the parameter vector ϑ, such that

$$\log L(N|A,\vartheta) = \sum_{i=1}^{p} \log L(N|\mathcal{A}_i,\vartheta)$$

We put them in sec 7, right?

gets maximized.

As in RAxML-NG, we use pll-modules to optimize branches using the Newton-Raphson optimization method. For this, we need the first and second derivatives of Phylogenetic Network Loglikelihood. We derive them in Appendix ??.

All displayed trees of a network share the same branch lengths. When optimizing the length of a branch, we must thus repeatedly recompute the loglikelihood of all displayed trees where the branch is active. A branch is *active* in a displayed tree, if (see also Figure 8):

- Its source is not a dead node.
- Its target is not a dead node.
- It is not connecting an inactive parent to a reticulation.

Remind that we use the linked model here.

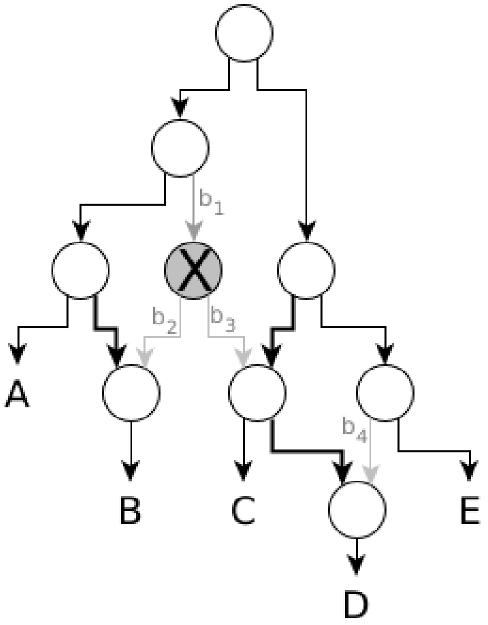


Fig. 8: A displayed tree in a phylogenetic network. The branches b_1, b_2, b_3, and b_4 are inactive in this displayed tree. When optimizing the length of any of the inactive branches, we do not need to recompute the likelihood of this displayed tree.

Appendix 20 describes the rationale against optimizing branches on each of the displayed trees separately (using tree loglikelihood) and then merging them.

8.1 Recap: Virtually Rerooting a Phylogenetic Tree

In order to avoid costly CLV updates when optimizing a branch (u, v) in a tree, RAxML-NG virtually re-roots the tree at the one of the nodes u defining the branch (see Figure 8.1).

After virtually rerooting a tree, we need to update all CLVs that lie on the path between the old and the new virtual root (both ends included). When trying different values for the branch length (u, v) in such a virtually rerooted tree, we can reuse the CLVs stored at u and v without having to recompute them. For this, libpll provides a function compute_edge_loglikelihood.

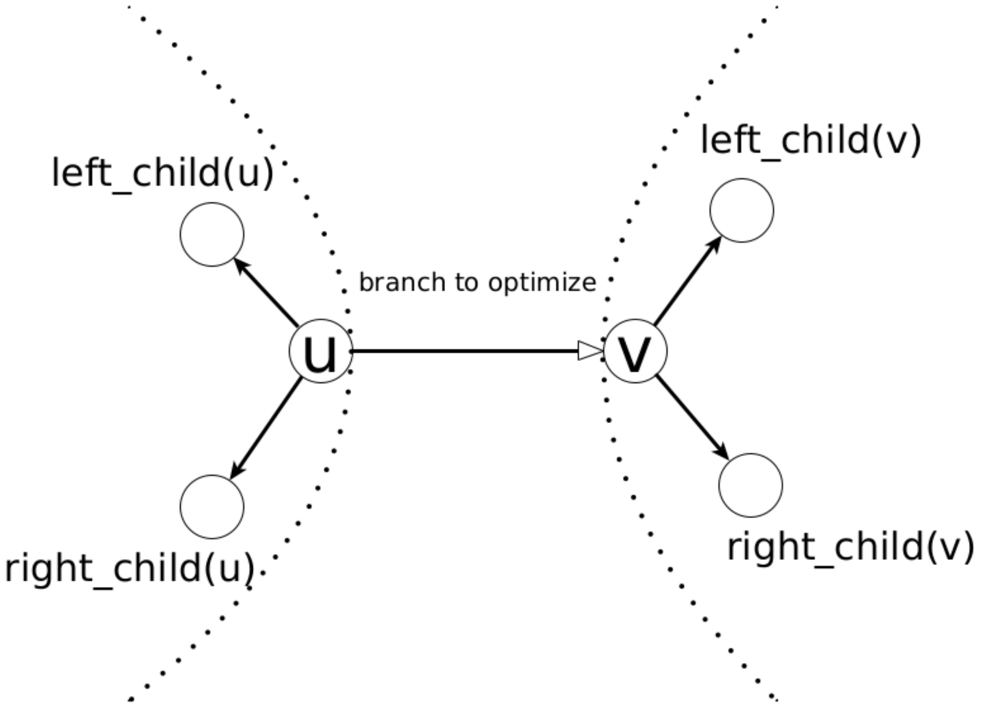


Fig. 9: When optimizing the length of a branch (u, v) in a tree, we virtually re-root the tree at node u, which is the source of the edge whose length we want to optimize. Some edge directions may change in this process. Note that, the phylogenetic likelihood is the same regardeless of the (virtual) root placement under time-reversible models. After the virtual reroot operation, we need to recompute the CLVs that lie on the path between the old virtual root and the new virtual root u.

do not describe this using the past tense

8.2 Virtually Rerooting a Phylogenetic Network

Analogously, for networks, we need to recompute the CLVs which lie on *any* path between the old and new virtual root (both ends included).

The difficulty here is that while we had the fixed network root, a network node always had the same children and the same parent. But now with the virtual rerooting, the edge directions depend on the specific displayed tree we are currently considering (see Figure 8.2). These different edge directions affect the order in which we need to process the CLVs. Thus, for each different set of edge directions, we need to do update the CLVs separately.

I couldn't follow this section.

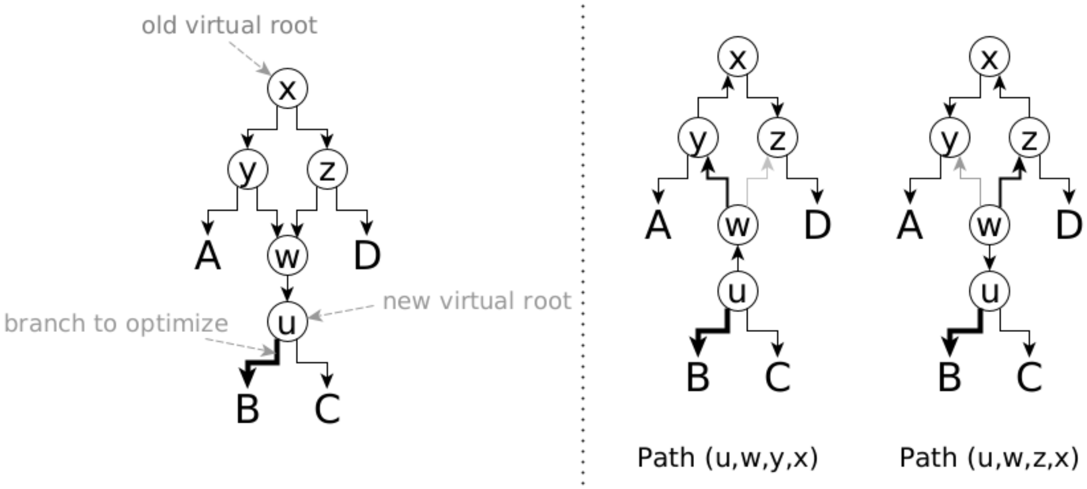


Fig. 10: On path (u,w,y,x), the node y has children A and x. We process the paths and fill the node DisplayedTrees/CLVs accordingly. When updating the CLVs for node z at path (u, w, z, x), node x has child y. The CLVs for y we actually need here are the ones where y has child A (and inactive child w). However, by processing the previous path, we have overwritten the CLVs stored at y to be the ones where x is a child of y.

We devised the following approach for resolving the edge direction problem:

Before processing the next path, we invalidate the old CLVs at all path nodes except for the new_virtual_root node. We detect in advance at which nodes we need to restore the old displayed trees (from how they were when we had old_virtual_root), save and restore them accordingly. (Simple example: Call it with two times the same pmatrix_index. First time recovering the CLVs worked perfectly, second time because the first time overwrote some things we are not able to correctly recover the old CLVs anymore.)

After we finished optimizing a branch, we then re-update the CLVs with regard to the original network root (see Section **??** for a rationale). This means, when optimizing multiple branches, we virtually re-root the network as follows: virtual_root_1 → network_root → virtual_root_2 → network_root → . . . virtual_root_k → network_root.

8.3 Combining Displayed Trees after Virtual Reroot

When updating CLVs for the virtually rerooted displayed trees around a branch *b* we want to optimize, we only update CLVs for the displayed trees where *b* is an active branch. The likelihood of the remaining displayed trees does not change, thus we do not need to recompute it. We store the updated likelihood of all displayed trees in the original, permanent network root node.

For the network loglikelihood evaluation, we need to combine the active displayed trees stored at the source node with the active displayed trees stored at the target node of the branch we want to optimize. (Here we call a displayed tree *active*, if the branch we want to optimize is active in the displayed tree.)

In order to combine a displayed tree stored at the source node and a displayed tree stored at the target node, their reticulation parent choice sets need to be compatible. Two reticulation parent choice sets are compatible, if their intersection is not empty. For example, the intersection of $\{0-, 11\}$ and $\{10, 01\}$ is $\{01\}$.

For when the branch we want to optimize is inactive, we obtain the remaining displayed trees to account for by reusing the old displayed tree loglikelihood data stored in the original network root node before we did the virtual rerooting.

We can reuse them from oldDisplayedTrees as the branch length has no effect in this case.

Again, I could not follow this
section

As displayed trees can have more than one set of reticulation parent choices, we need to re-infer the reticulation parent choices leading to the current inactive displayed tree. For each old displayed tree, we need to recompute the reticulation parent choices leading to this displayed tree.

Algorithm 1: combineTrees: Obtain the active and inactive displayed trees after virtually rerooting a network.

```
Input: The branch $(u, v)$ we want to optimize. The current network annNetwork,
        virtually re-rooted at $u$. The old displayed trees loglikelihood data with respect to
        the original network root, $oldDisplayedTrees$.
Output: The set of active and inactive displayed trees after virtual rerooting, with
        respect to $(u, v)$.
activeTrees ← None
inactiveTrees ← None
for sourceTree ∈ displayedTreesAt(u) do
    for targetTree ∈ displayedTreesAt(v) do
        combinedChoices ← combineReticulationChoices(sourceTree.reticulationChoices,
         targetTree.reticulationChoices)
        if combinedChoices $\neq \emptyset$ and isActiveBranch($(u, v)$, combinedChoices) then
            activeTrees.append({combinedChoices, sourceTree.clv, targetTree.clv})
        end
    end
end
for oldTree ∈ oldDisplayedTrees do
    remainingChoices ← unseenChoices(oldTree.reticulationChoices, combinedTrees)
    if remainingChoices $\neq \emptyset$ then
        inactiveTrees.append({remainingChoices, oldTree.loglh})
    end
end
return (activeTrees, inactiveTrees)
```

Algorithm 2: computeNetworkLoglhBrlenOpt: Compute network loglikelihood after virtual re-rooting.

```
Input: The branch $(u, v)$ we want to optimize. The set of active and inactive displayed
        trees after virtually re-rooting the network at node $u$.
Output: The current network loglikelihood with respect to branch $(u, v)$.
displayed_trees ← $\emptyset$
for activeTree ∈ activeTrees do
    displayed_trees.append(activeTree.reticulationChoices,
     computeEdgeLoglikelihood(activeTree.source_clv, activeTree.target_clv, $(u, v)$))
end
for inactiveTree ∈ inactiveTrees do
    displayed_trees.append(inactiveTree.reticulationChoices, inactiveTree.loglh)
end
return computeLoglikelihood(displayed_trees)
```

8.4 Computing Network Loglikelihood Derivatives

We use the active displayed trees we computed in Algorithm 1 for computing the network loglikelihood derivatives. First, we use libpll to compute the loglikelihood and loglikelihood derivatives for each of the active displayed trees, both with respect to the branch (u, v) whose length we aim to optimize. Then we use our formulas from Section 7 to compute the network loglikelihood derivatives.

9 Optimizing Non-Topology Parameters

When optimizing non-topology parameters, our goal is to optimize the Phylogenetic Network Loglikelihood function. This is, we aim to find optimal parameters in the parameter vector ϑ, such that we maximize

$$\log L(N|A,\vartheta) = \sum_{i=1}^{p} \log L(N|\mathcal{A}_i,\vartheta).$$

9.1 Likelihood Model Parameter Optimization

We reuse the TreeInfo class from RAxML-NG for calling likelihood model optimization. As this uses only pll_partition_t objects, we can directly reuse the pre-existing likelihood model optimization for trees to optimize the likelihood model parameters for networks.

9.2 Optimization of Reticulation Probabilities

We use Brent optimization (as provided by libpll) for optimizing the first-parent probability of a reticulation. Say a bit more?

9.3 Complete Nontopology Optimization Routine

NetRAX provides full optimization of all nontopology parameters (likelihood model, branch lengths, reticulation probabilities) in three variants: QUICK, NORMAL, and SLOW. These variants differ in how often we repeat an optimization step (see Algorithm 3).

Algorithm 3: optimizeNonTopology: Optimize all non-topology parameters.

```
Input: The current network annNetwork with parameter vector $\vartheta$; the optimization type type.
Output: The current network, with optimized non-topology parameters in $\vartheta$.
score_epsilon ← 0.01
max_rounds_slow ← 2
act_rounds_slow ← 0
got_better_slow ← True
while got_better_slow and act_rounds_slow < max_rounds_slow do
    score_slow_before ← scoreNetwork(annNetwork)
    got_better_slow ← False
    do_brlen_opt ← True
    do_model_opt ← True
    do_reticulation_opt ← True
    got_better ← True
    while got_better do
        got_better ← False
        score_before ← scoreNetwork(annNetwork)
        if do_model_opt then
            score_before_modelopt ← scoreNetwork(annNetwork)
            optimizeModel(annNetwork)
            score_after_modelopt ← scoreNetwork(annNetwork)
            if score_before_modelopt - score_after_modelopt < score_epsilon then
                do_model_opt ← False
            end
        end
        if do_brlen_opt then
            score_before_brlenopt ← scoreNetwork(annNetwork)
            optimizeBranches(annNetwork)
            score_after_brlenopt ← scoreNetwork(annNetwork)
            if score_before_brlenopt - score_after_brlenopt < score_epsilon then
                do_brlen_opt ← False
            end
        end
        if do_reticulation_opt then
            score_before_reticulation_opt ← scoreNetwork(annNetwork)
            optimizeReticulations(annNetwork)
            score_after_reticulation_opt ← scoreNetwork(annNetwork)
            if score_before_reticulation_opt - score_after_reticulation_opt > score_epsilon then
                do_reticulation_opt ← False
            end
        end
        score_after ← scoreNetwork(annNetwork)
        if score_before - score_after > score_epsilon then
            if type ≠ QUICK then
                got_better ← True
            end
        end
    end
    score_slow_after ← scoreNetwork(annNetwork)
    if score_before - score_after > score_epsilon and type = SLOW then
        got_better_slow ← True
    end
    act_rounds_slow += 1
end
return annNetwork
```

quickly define
somewhere
these.
functions

10 Network Moves

NetRAX implements the following rooted network topology rearrangement moves, as well as reversal (undo) operations for them: rNNI move, rSPR move, rSPR1 move, head move, tail move, delta plus move, delta minus move, arc removal move, arc insertion move. These are all the moves from Gambette *et al.* [5].

When undoing a move, we restore the original topology and branch lengths. Doing or undoing a move also invalidates some CLV vectors. (When a CLV is invalid, it means that we cannot reuse its entries and need to recompute it.)

Since we require all CLV and pmatrix indices to be consecutive, we sometimes need to swap them before performing or undoing a move.

10.1 rNNI Move

An rNNI move has the following properties:

- No arcs $\{u, t\}$ or $\{s, v\}$ are present before the move.
- The nodes u and v exchange their neighbors.
- The edge direction between u and v may change.
- The in- and outdegrees of s and t do not change.
- The resulting network needs to be acyclic.

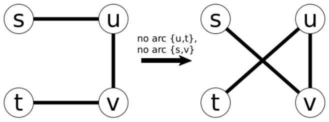


Fig. 11: Generalized visualization of a rNNI move.

When implementing an rNNI move, there are seven cases to consider (see Figure 10.1).

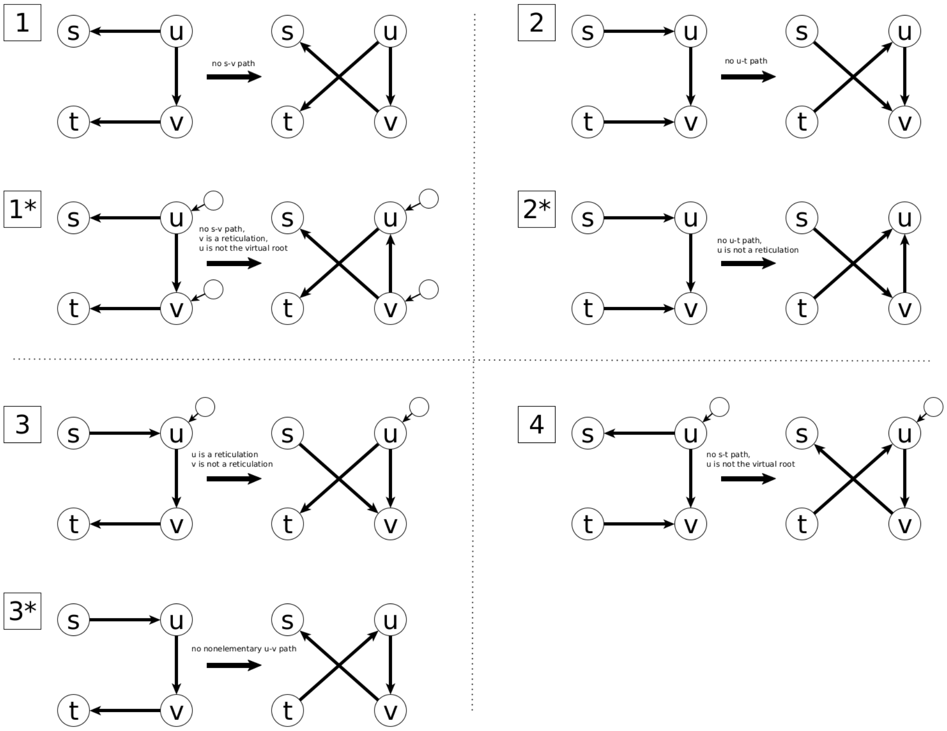


Fig. 12: All seven cases that can occur when implementing a rNNI move.

10.2 rSPR Move

An rSPR move has the following properties:

- No arcs $x'z$, zy', xy are present before the move.
- The resulting network needs to be acyclic.

The arcs xy, yz are the *donor arcs* and the arc $x'y'$ is the *recipient arc*.

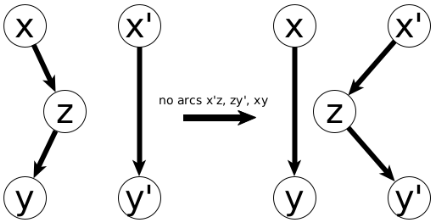


Fig. 13: Generalized visualization of a rSPR move.

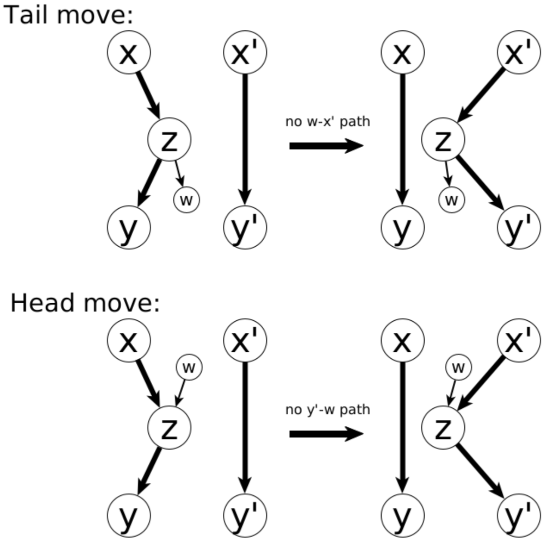


Fig. 14: An rSPR move is either tail-moving or head-moving.

A rSPR1 move is a special kind of rSPR move where the recipient arc is incident to one of the donor arcs (i.e., $x = y'$, $x' = x$, $x' = y$, or $y = y'$).

10.3 Arc Removal/Insertion Move, Delta Plus/Minus Move

An arc removal move removes an arc uv from a reticulation v. When performing an arc removal move, we

- remove 1 bifurcation
- remove 1 reticulation
- remove 5 arcs
- add 2 new arcs

An arc insertion move chooses two distinct arcs ab and cd, with cd not being ancestral to ab. When performing an arc insertion move, we

- add 1 bifurcation

- add 1 reticulation
- remove 2 arcs
- add 5 new arcs

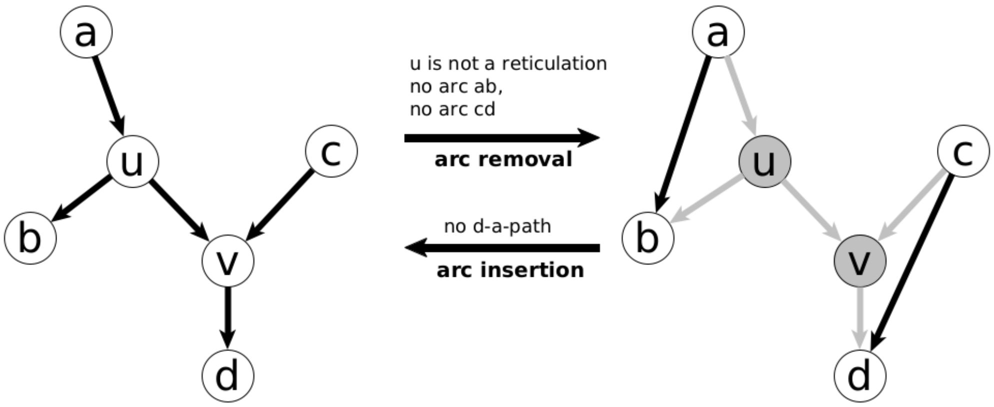


Fig. 15: Arc removal and arc insertion move.

A delta minus move is an arc removal move where $a = c$, $b = d$, or $b = c$. A delta plus move is an arc insertion move where $a = c$, $b = d$, or $b = c$.

Adding a Node or an Edge Adding an element is easy: We search for an unused space in the underlying non-consecutive array in the network, create a new object there, and assign it the highest active index + 1.

Deleting a Node or an Edge When removing a node or an edge, we need to ensure that CLV, pmatrix, and reticulation indices remain consecutive after the operation. Thus, we swap the element at the current highest index with the element we want to remove. We update all references to both affected elements. This essentially swaps the indices associated with the element we want to remove and the element that was previously referred to by the highest index.

The references we need to update include:

- nodes_by_index, edges_by_index, reticulations_by_index
- edge_pmatrix_index and node_clv_index of the links
- reticulation_index, link_to_first_parent and link_to_second_parent in the ReticulationData
- reticulation_choices in the DisplayedTreeData stored at each node in the network
- all references to clv_index and pmatrix_index in the current move data

Too much detail on the data structure?

11 Network Search

Key ideas:

- Pre-filter and rank move candidates to take the best one.
- Search in waves by move type, trying horizonal moves after accepting an arc insertion move.

11.1 Search in Waves

Scoring a Network NetRAX supports vertical topology-rearrangement moves that can increase (ArcInsertion move, DeltaPlus move) or decrease (ArcRemoval move, DeltaMinus move) the number of reticulations in a network. Because model complexity changes when adding or removing reticulations from a network, we cannot compare networks directly via their loglikelihood. For this, NetRAX implements AIC, AICc, and BIC scoring. By default, NetRAX uses the BIC score to compare different networks. Park and Nakleh [11] showed that using BIC performs best in network searches.

The BIC score of a network N with r reticulations on a partitioned MSA A is:

$$\text{BIC}(N|A) = -2 * \log L(N|A) + \text{n\_free\_parameters} * \log(\text{sample\_size}).$$

The free parameters consist of substitution model parameters, reticulation first-parent probabilities, and branch lengths. The sample size is the product of the number of taxa and the number of MSA sites.

Start Networks NetRAX can start the network topology search from:

- A user-specified start network
- A random tree (generated with RAxML-NG)
- A maximum-parsimony tree (generated with RAxML-NG)

Why? I am just being curious.

So far, NetRAX cannot directly start a RAxML-NG tree search in order to initiate the network search from the best tree found by RAxML-NG. In order to obtain this behavior, the user has to carry out the RAxML-NG tree search first and then pass the maximum-likelihood tree inferred by RAxML-NG to NetRAX. We recommend using NetRAX with a maximum likelihood tree inferred by a preceding RAxML-NG run.

Outer Search Loop The current NetRAX search algorithm iterates over the possible move types in the following order: ArcRemovalMove, RSPRMove, RNNIMove, ArcInsertionMove.

For each move type, we have a set of candidate move operations (e.g., there are multiple resulting networks we can reach by doing a single move of a given type on the network – each of these possible moves is a candidate move). Searching and evaluating all move candidates varies in speed across move types, because for some move types there are more possible candidates than for others. In order to reduce the number of candidates to evaluate, we only consider moves within a pre-specified search radius (by default, we use a radius of 5) around each node.

Whenever NetRAX finds a better network than the currently best one, it overwrites the best network found so far on the file system.

NetRAX has an outer search loop 4 and an inner search loop 5.

The network search starts with the fastest move type (which is the ArcRemovalMove). Only if a topology search loop did not lead to a better network topology with the specified move type, it continues with the next-slower move type. The search loop ends when even the slowest/most thorough move type does not yield a better network topology likelihood.

NetRAX uses a greedy hillclimbing approach for optimizing the network topology (see Algorithm 4).

I didn't get this.

I am not sure we need an algorithm for alg.
Maybe we can say it with 4-7
words and move the algorithms to the
appendix?

```
Algorithm 4: The outer network search loop, iterating over move types.
Input: The current best-scoring network annNetwork, the search radius searchRadius
Output: The best-scoring network found in the search bestNetwork
bestNetwork ← annNetwork
continueSearch ← True
while continueSearch do
    continueSearch ← False
    for moveType in {ArcRemovalMove, rSPR move, rNNI move, ArcInsertionMove} do
        newNetwork ← innerSearch(annNetwork, moveType, searchRadius)
        if scoreNetwork(newNetwork) < scoreNetwork(bestNetwork) then
            bestNetwork ← newNetwork
            continueSearch ← True
        end
    end
end
return bestNetwork
```

11.2 Inner Search Loop (Topology Search Algorithm, for a given move type)

When gathering possible move candidates, NetRAX by default uses a search radius of 5. This is, for each node in the network, we only consider rSPR moves and arc insertion moves within radius of 5 nodes around the current node. The searchRadius parameter does not affect our choice of rNNI or arc removal moves.

When accepting a move, we optimize all branch lengths, reticulation probabilities, and model parameters in the new best network. We also update the clv index and pmatrix index references in the set of remaining candidate moves, according to the index remappings induced by the accepted move. When updating the old move candidates, we also remove the now invalid ones.

Filtering Move Candidates (to find the promising ones) When filtering the set of move candidates, we use three different filtering modes. These differ in the number of branches we optimize before evaluating a candidate.

- FilterType::PREFILTER – Do not optimize branch lengths. (Exception: For prefiltering an ArcInsertionMove, we need to optimize the length of the newly introduced branch, because we do not have a usueful initial guess for it.)
- FilterType::RANK – Optimize branches directly affected by the move.
- FilterType::CHOOSE – Optimize all branches in the network.

Elbow Method In the elbow method [13] [12], we sort the list of pre-scored move candidates by increasing BIC score. We need to find the point with the largest distance to the line from the first to the last candidate; this point corresponds to our chosen cutoff value. We keep all candidates with score smaller-or-equal than this cutoff score value (see Figure 33).

Again, do we need an algorithm?

Algorithm 5: The inner network search loop, for a given move type.

```
Input: The current best-scoring network annNetwork, the current move type moveType, the
       search radius searchRadius
Output: The best-scoring network found in the search bestNetwork
bestNetwork ← annNetwork
continueSearch ← True
candidates ← gatherCandidates(annNetwork, moveType, searchRadius)
triedWithAll ← True
while continueSearch do
    continueSearch ← False
    candidates ← filterCandidates(bestNetwork, candidates, FilterType::PREFILTER)
    chosenMove ← selectBestCandidate(bestNetwork, candidates)
    if chosenMove then
        bestNetwork ← acceptMove(annNetwork, chosenMove)
        if $moveType \neq ArcInsertionMove$ then
            continueSearch ← True
            candidates ← updateOldCandidates(candidates, chosenMove)
            triedWithAll ← False
            if $candidates == \emptyset$ then
                candidates ← gatherCandidates(annNetwork, moveType, searchRadius)
                triedWithAll ← True
            end
        end
    end
    else if $triedWithAll == False$ then
        candidates ← gatherCandidates(annNetwork, moveType, searchRadius)
        triedWithAll ← True
        continueSearch ← True
    end
end
return bestNetwork
```

Algorithm 6: filterCandidates: Reduce the set of move candidates.

```
Input: The current best-scoring network annNetwork, the set of move candidates candidates,
       the filter type filterType
Output: The reduced set of move candidates.
oldScore ← scoreNetwork(annNetwork)
filteredCandidates ← None
scoredCandidates ← None
oldModel ← extractModel(annNetwork)
for cand ∈ candidates do
    performMove(annNetwork, cand)
    if filterType == FilterType::PREFILTER then
        if cand.moveType == ArcInsertionMove then
            optimizeNewBranch(annNetwork, cand.u_v_pmatrix_index)
            optimizeNewReticulation(annNetwork, cand.new_reticulation_index)
        end
    end
    else if filterType == FilterType::RANK then
        optimizeBranches(annNetwork, getBranchesAffectedByMove(cand))
        optimizeReticulations(annNetwork)
    end
    else
        // filterType == FilterType::CHOOSE
        optimizeAllBranches(annNetwork)
        optimizeReticulations(annNetwork)
    end
    actScore = scoreNetwork(annNetwork)
    scoredCandidates.append(cand, actScore)
    undoMove(annNetwork, cand)
    assignModel(annNetwork, oldModel)
end
sortByIncreasingScore(scoredCandidates)
nKeep ← elbowMethod(scoredCandidates)
if filterType == FilterType::PREFILTER then
    nKeep ← max{nKeep, numberOfCandidatesWithBetterBIC(scoredCandidates, oldScore)}
end
scoredCandidates.resize(nKeep)
return all candidates moves from the remaining scoredCandidates array
```

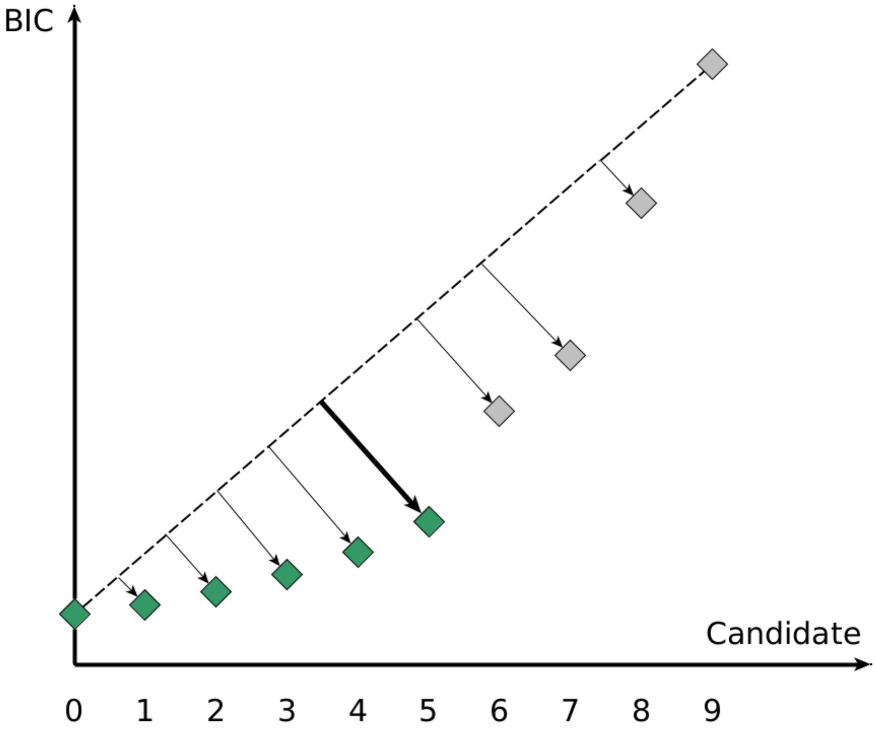


Fig. 16: The elbow method. We keep all candidates with BIC smaller-or-equal to the cutoff value. In this example, we keep the first six candidates, candidates 0 to 5.

Deciding on the most promising move candidate In order to determine the most promising move candidate, we filter the candidates until we identified the one that improves the BIC the most.

?

```
Algorithm 7: selectBestCandidate: Finding the best score-improving move candidate.
Input: The current best-scoring network annNetwork, the set of move candidates
       candidates
Output: The best score-improving move to apply to the network.
chosenMove ← None
rankCandidates(annNetwork, candidates, FilterType::RANK)
chooseCandidates(annNetwork, candidates, FilterType::CHOOSE)
if $candidates \neq \emptyset$ then
    chosenMove ← candidates[0]
end
return $chosenMove$
```

Updating a list of Move Candidates After we accepted a move in the search, we want to reuse other promising candidate moves. Because of index remappings, we need to also update the indices stored in the remaining move candidates after accepting a move. For this, each Move object stores a list of remapped CLV indices, remapped pmatrix indices, and remapped reticulation indices. After accepting a move, we update the other move candidates by applying the index remappings from the accepted move in order. When undoing a move, we perform its index remappings in reverse order. We also update the stored CLV and pmatrix index references in the move objects.

I stopped here.

12 Implementation

Key ideas:

- Modular software architecture
- MPI parallelization of loglikelihood computation and computation of loglikelihood derivatives over sites in the MSA

12.1 Modular Architecture

TODO: Picture showing how we split up NetRAX into separate modules

12.2 Parallelization

We parallelized both displayed-tree loglikelihoood and displayed tree loglikelihood derivatives computation in NetRAX over the MSA sites by using MPI. For this, we reuse the `ParallelContext` class and load balancing solutions provided by RAxML-NG.

In order to avoid redundant computations and making best use of per-tree MPI parallelization, we first iterate over nodes in a network. Then, we go over all displayed trees stored in a node. We precompute data (`prob_invar` and `diagpable` in libpll) that remains unchanged among computations for multiple displayed trees.

13 Simulation of Phylogenetic Networks and Sequences

TODO: Describe the simulator (I found some old text from Celine and pasted it here)

The simulator simulates an ultrametric networks and extracts displayed trees out of the simulated network, one for partition. Then simulates sequences on the output displayed trees.

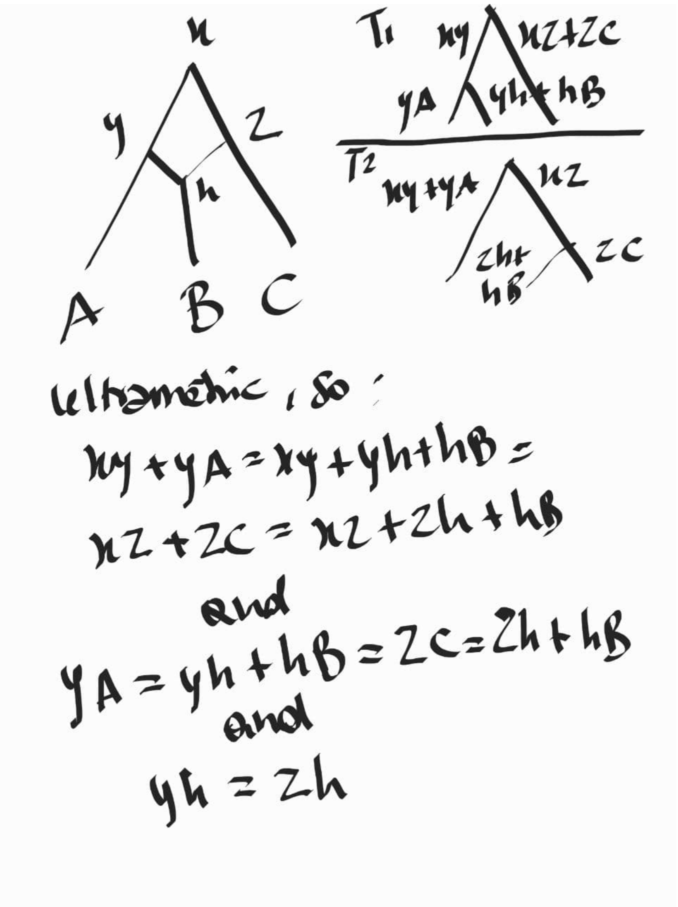


Fig. 17: Ultrametric trees from the simulator.

We are going to simulate our networks under the birth-hybridization process of Zhang *et al.* [15]. In this process, we have a speciation rate l (lambda), a hybridisation rate v (nu) and the time we run the process t_0.

From their paper: "The simulator starts from the time of origin (t_0) with one species. A species splits into two (speciation) with rate l, and two species merge into one (hybridization) with rate v. At the moment of k branches, the total rate of change is r_tot = k l + choose (k 2) * v. We generate a waiting time ∼ exp(r_tot) and a random variable u ∼ U(0,1), If u ¡ kl=r_tot, we randomly select a branch to split; otherwise, we randomly select two branches to join, and generate an inheritance probability c ∼ U(0,1). The simulator stops at time t_0."

No transfer from now, because NetRAX cannot recover branches with no length.

Given such generated network, we extract a random tree, using the inheritance probabilities at the reticulated nodes: for each reticulated node x, we draw a number y ∼ U(0,1) and we choose the first parent p_x if y ¡ inheritanceProb of the edge p_x,x. If we do this,we have atree but not a binary one. Then we clean the tree from the dead nodes and 1-indegree 1-outdegree nodes, as Sarah explained in her document, and we obtain a binary tree. On this binary tree, we simulate N sequences with Seq-Gen-1.3.4 (currently under this model -mHKY -t3.0 -f0.3,0.2,0.2,0.3 we can change this). We do this K times, obtaining an alignment of length K*N in which each group of N sites has its tree.

In their paper, they use this parameters in the simulations

$\lambda - \nu \sim$ exp(0.1) with mean 10 $\tau_0 \sim$ exp(10) with mean 0.1 $\nu/\lambda \sim$ Beta distribution with alpha = 1 and beta = 1 (same as uniform distribution between 0 and 1) $\gamma \sim$ Uniform distribution (between 0 and 1)

As I said in a slack message, this process gives Yule-type networks, that act as the Yule-type trees, see example below for trees.

Once they start having a certain amount of nodes, things get crazy and they start speciating and hybridising a lot. It is not the focus of our paper to correctly simulate networks (we are not in a Bayesian framework, we do not need priors) and this is the best we can for now (for the next paper, we will do better, working on it). So we can simply throw the networks we do not like as done in line 143-147 of this simulator, which only simulate networks and not trees and sequences.

We use seq-gen for simulating sequences on the displayed trees with the following parameters: TODO

13.1 Excluding Weird Networks

Since our simulator simulates ultrametric networks,

TODO: Explain how we exclude weird networks. With number of pairs and number of equal pairs... This network should have a weirdness of 1.0, because all its displayed trees induce the same bipartitions.

There is still another problem though... now that I fixed the network weirdness computation, it turns out that we have a lot of ”weird” simulated networks (meaning, displayed trees inducing the same bipartition sets). These networks have reticulations which we cannot infer with NetRAX (or any other tool) because the simulated networks are ultrametric, making the data indistinguishable from not having the problematic reticulations.

Simply excluding these networks sounds better to me though, since the BIC stats on the weird networks are obvious (”MSA 100 percent supports a tree? Well, let’s infer a tree then.”)

It still makes sense to have them in the simulator though, since extinctions etc in biology can lead to such ”weird” networks. I just don’t see a good reason for wasting more compute hours running NetRAX on them, if their induced MSA is indistinguishable from having a tree...

ignore ”undetectable”/”weird” reticulations weird networks (e.g., displayed trees having the same bipartitions)

14 Experimental Setup

14.1 Simulated Data

We simulated phylogenetic networks using our simulator from Section 13. Our simulation used the linked branch lengths model (meaning, all partitions share the same set of branch lengths). We simulated a partition with 1000 MSA sites for displayed tree in the simulated network.

In all experiments on simulated data, we compared our inferred network with the true simulated network using BIC and the network topology distances discussed in Section 15.

For each simulated dataset, we inferred a Maximum-Likelihood tree with RAxML-NG. We then did the following experiments, using NetRAX with both LikelihoodModel.AVERAGE and LikelihoodModel.BEST:

A: Standard In our standard experiment setting, we assessed NetRAX inference quality with different number of taxa and number of reticulations. We simulated networks with $n\_taxa = \{10, 15, 20, 25, 30, 35, 40\}$ and $n\_reticulations = \{1, 2, 3, 4\}$, one network for each combination of number of taxa and reticulation count. We used probability 0.5 for each reticulation. We started the NetRAX inference from the RAxML-NG maximum-likelihood tree.

B: Reticulation Probability In our reticulation probability experiment setting, we simulated a single dataset with 20 taxa and one reticulation. We then varied the first parent probability of the simulated reticulation to be in $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. We started the NetRAX inference from the RAxML-NG maximum-likelihood tree.

C: Brlen Scaler In our brlen scaler experiment setting, we simulated a single network with 40 taxa and 4 reticulations, using probability 0.5 for each reticulation. Before simulating the sequences for this dataset, we multiplied all branches in the simulated network by $s \in \{1, 2, 4, 8\}$. We started the NetRAX inference from the RAxML-NG maximum-likelihood tree.

D: Unpartitioned Data In our unpartitioned data setting, we used the same setting as in experiment A, but merged all simulated partitions into a single partition before running tree or network inference.

E: Multiple Starting Trees In our multiple starting trees setting, we used the same setting as in experiment A, but ran NetRAX inference using 3 random and 3 maximum parsimony starting trees and compared the network inference result to the one obtained with starting from only the RAxML-NG maximum-likelihood tree.

F: Scrambled Partitions In our scrambled partitions experiment setting, we simulated a single dataset with 30 taxa and 3 reticulations. Before running NetRAX inference (using the RAxML-NG maximum-likelihood tree as starting tree), we randomly scrambled the partitions such that $p \in \{0\%, 10\%, 20\%, 30\%, 40\%, 50\%, 60\%, 70\%, 80\%, 90\%, 100\%\}$ of the sites from each partition were randomly reassigned to other partitions.

G: Fixed number of taxa and reticulations, 100 datasets In this experiment setting, we simulated 100 networks with 40 taxa and 4 reticulations, using probability 0.5 for each reticulation. We started the NetRAX inference from the RAxML-NG maximum-likelihood tree.

H: Different Alignment Size In this experiment setting, we simulated a single network with 30 taxa and 3 reticulations, using probability 0.5 for each reticulation. We simulated $\{100, 500, 1000, 5000, 10000\}$ sites per partition. We started the NetRAX inferences from the RAxML-NG maximum-likelihood trees.

14.2 Empirical Data

We want to run NetRAX on this empirical dataset (from https://advances.sciencemag.org/content/5/5/eaav9188) which is available for download at https://bioweb.supagro.inra.fr/WheatRelativeHistory/index.php?menu=download

We need the IndividualAlignments data, which we merge together into one big MSA (I wrote a script for it here: https://github.com/lutteropp/NetRAX/blob/master/scripts/merge_gene_alignments.py) Each gene is its own partition. The standard RAxML model (GTR+GAMMA) is okay We have 47 individuals over 17 species in the dataset. There are 1387815 patterns in the merged MSA, and 8738 partitions In order to create a MSA based on species, we can either use extended DNA alphabet or, more elegant version: Chapter 5 of Alexeys PhD thesis (https://cme.h-its.org/exelixis/pubs/dissAlexey.pdf)

That's the tree inferred by RAxML-NG, which I will use as start network for NetRAX:

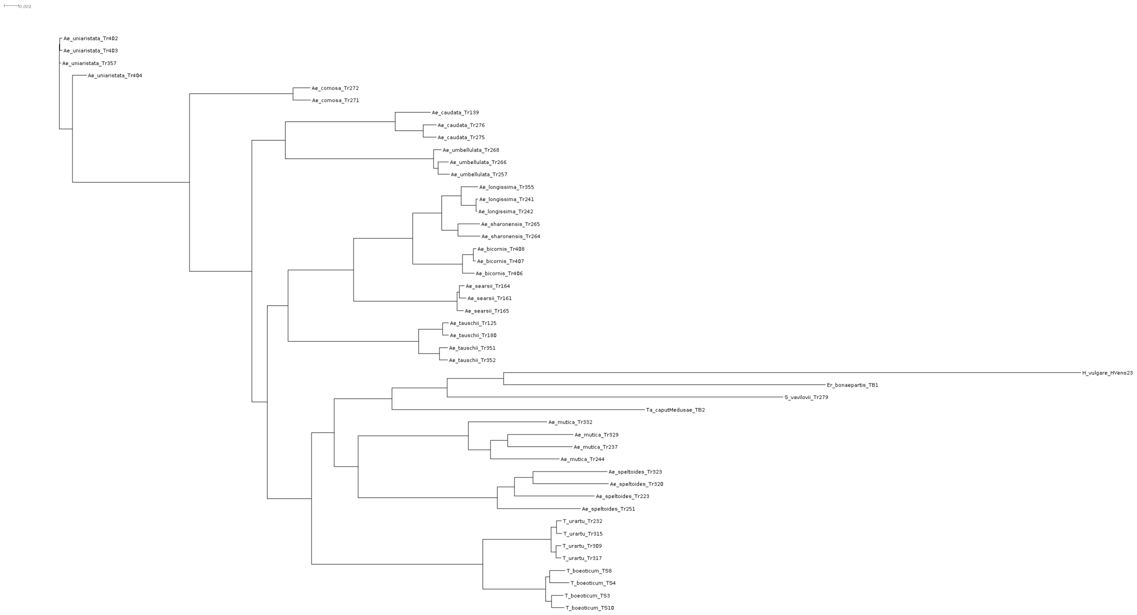


Fig. 18: That's the tree inferred by RAxML-NG, which I will use as start network for NetRAX

RAxML-NG vanilla (with 10 random and 10 parsimony starting trees, site repeats and coarse-grain load balancing disabled, taking binary MSA file as input, on 20 cluster nodes) on the big empirical dataset needs about 4 hours in total, which is 15 minutes per starting tree. I used the master branch of RAxML-NG for tree inference, commit 6db1154d64b5b6f65c09dccdc4576a8e5ab87195.

14.3 Evaluation Metrics

We use the following for evaluating quality of results:

- Number of reticulations in the network
- Likelihood-based evaluation metrics
  - Loglikelihood of the network
  - BIC score of the network
  - AIC score of the network
  - cAIC score of the network
- Topology-based evaluation metrics
  - hardwired network distance
  - softwired network distance
  - displayed trees network distance
  - tripartition network distance
  - nested labels network distance
  - path multiplicity network distance

15 Topology-based evaluation metrics

See the book Phylogenetic Networks: Concepts, Algorithms and Applications

I have just figured out that we can easily plot relative distance versions (in range [0.0, 1.0]) of all topological network distances. When looking at the definitions in @celines network book, they all are of the form: (—symmetric difference between A and B—) divided by 2. -¿ We just need to change them to be (—symmetric difference between A and B—) divided by (—A union B—) and there we go. Then, we will get relative distances. These will make nicer plots.

So we need to diverge from the distance definitions in Celines network book: We will explicitly discard the trivial bipartitions/clusters/whatever in our own distance implementations.

Totally ok for discarding trivial bipartitions and use the denominator you suggested and not 2

Definition 4 (Cluster). *A* cluster *in a rooted phylogenetic network is a non-empty proper subset of the set of taxa present in the network. The cluster induced by an edge in the network is the set of taxa "below" the edge. This is, the set of taxa that are descendants of the edge's target node.*

- Two clusters are *compatible*, if they are disjoint or one cluster contains the other.
- For obtaining the set of *hardwired* clusters, we collect the clusters induced by every edge in the network. Here, we have all reticulation edges activated at once. With the hardwired interpretation, one edge induces a single cluster.
- For obtaining the set of *softwired* clusters, we go through each tree displayed by the network and add the set of clusters induced by the displayed tree to the total set. Here, we toggle active and inactive reticulation edges when going through the displayed trees. With the softwired interpretation, one edge induces a set of clusters (up to one per displayed tree).

- A *cluster* induced by an edge is the set of taxa descending from the target node of the edge.
- For *hardwired* clusters, we keep all reticulation edges activated at once, and each edge induces a single cluster this way.
- For *softwired* clusters, we go through the displayed trees one-by-one, switching reticulation edges on and off during the process. Here, each edge induces a set of clusters (up to one per displayed tree).

Definition 5 (Unrooted Softwired Distance). *For a given network N, $T(N)$ are the displayed trees of N and $B(N)$ are the set of all bipartitions of the trees in $T(N)$. Then, the softwired unrooted distance between two networks N_1 and N_2 is*

$$\frac{|B(N_1) \triangle B(N_2)|}{|B(N_1)| + |B(N_2)|}$$

16 Results and Discussion

Spoiler: If you look at the unrooted softwired network distance, we are getting great results starting from only RAxML-NG best tree. Often even relative distance zero, even with 40 taxa and 4 reticulations!

I also noticed that LikelihoodModel.AVERAGE always performed better-or-equal (and yes, sometimes slightly better!) than LikelihoodModel.BEST in our simulated datasets. Which is a surprising result because LikelihoodModel.BEST should be fine for our simulated data (since we simulated each partition on a single displayed tree). My explanation attempt is that the NetRAX network search gets stuck in local optima. Also, non-surprising as it requires less computations, LikelihoodModel.BEST was always the faster one.

Some more initial spoiler informations I see from closely looking at the CSV file (the one I posted above) from the standard experiment round: The relative unrooted softwired network distance is damn good, overall There were two cases where that distance was a bit higher: In Case 1, we found a different network with slightly better BIC score than the true network In Case 2, there were two near-zero branches in the simulated network In one of the setting (with 10 taxa and 1 reticulation), BIC preferred a tree. In all other settings, we always inferred the correct number of reticulations when using LikelihoodModel.AVERAGE. In the rare cases where LikelihoodModel.AVERAGE performed better than LikelihoodModel.BEST, it was because LikelihoodModel.BEST inferred 1 reticulation less than LikelihoodModel.AVERAGE. (edited) 9:25 I hereby conclude that the slower-to-evaluate LikelihoodModel.AVERAGE is the better choice regarding quality of inference results. This is because it seems to perform slightly better when it comes to avoiding local optima in the search. It is not an inherent advantage of the model per se, but happens when interacting with the currently implemented network search algorithm.

now it's prefiltering for 7 reticulations :exploding_head: ... (the theoretical maximum we could end up with here is 16, as there are 16 partitions in the dataset) I need to abort this experiment and run it with way less taxa and reticulations to start with! It already gets very clear that LikelihoodModel.BEST is garbage if the passed partitions are dirty. Still interesting to see what will happen with LikelihoodModel.AVERAGE, that one should work out just fine. (edited) 8:48 I am 99% sure that the error we will get is an out of memory error. Because now with prefiltering 7 reticulations (meaning we have 14 displayed trees to keep track of, and our current implementation keeps all CLVs for all displayed trees in memory), NetRAX already uses 10 out of 16 GB RAM on the PhD laptop.

essentially what is happening here is garbage in, garbage out. Because when having a partition, we assume all sites belonging to the partition evolved together. With the scrambling, we are violating this assumption.

Interesting. With 30 taxa 3 reticulations, we did not have the rapid reticulation growth in the scramble partitions experiment. Instead, both likelihood models performed equally bad the more messy the partitions got, LikelhoodModel.AVERAGE was only slightly better, but comparable to LikelihoodModel.BEST. (edited) 4:05 both LikelihoodModel.BEST and LikelihoodModel.AVERAGE had the overestimating number of reticulations issue as soon as 30% of the sites were scrambled among partitions. They just overestimated the number of reticulations less (by just 1) than before.

17 Comparison with other Tools (TODO: Which ones?)

18 Future Work

18.1 Runtime Improvements

Improve virtual re-rooting during Branch-Length Optimization Currently, we always reroot back to original root before going on with the next branch to optimize. A possible alternative solution would be: Store which nodes were the children used in the DisplayedTreeData. Then, it makes sense to store multiple DisplayedTreeData's in a node, with the same reticulation choices but different set of children. And then one has to make sure to also check for compatible children setting when looking at a DisplayedTreeData from a child at a current node (it is only compatible if the current node does not show up in the list of children). To do final tree logl evaluation right, keep a flag stating whether a tree was newly added or not. And only evaluate using the newly added trees.

So far, after we finished optimizing a branch, we then re-update the CLVs with regard to the original network root. This means we have virtual_root_1 → network_root → virtual_root_2 → network_root → virtual_root_3 → network_root.

Regarding minimizing the total number of CLVs to recompute, the solution proposed here implements virtual_root_1 → virtual_root_2 → virtual_root_3 → network_root.

Speed up Branch-Length and Model Optimization Do model optimization less often – currently, we do it at the beginning, once at the end, and every time we accept a move. Another future work thing to try later: Currently, NetRAX optimizes all branch lengths in the choosing phase. Maybe it's enough to just optimize the branches within a certain radius. Speedup from this shouldn't be super large though, as we typically have less than 10 candidate networks to score in the choosing phase. But maybe it's also enough to just optimize branches within a radius when accepting a move (currently, we do full modelopt, reticulation opt, and full brlen opt when accepting a move).

While still nowhere used, NetRAX offers functionality to compute loglikelihood on just a subset of displayed trees. Explore if we can use it in nontopology parameter optimiziation. When doing minor local optimizations, maybe it suffices to sample likelihood over the most probable displayed trees?

Faster Candidate Filtering – Pseudolikelihood When scoring potential move candidates in the filtering phases during the network search algorithm, NetRAX frequently computes Phylogenetic Network Loglikelihood. We can speed up the initial candidate filtering by using a faster likelihood function, such as pseudolikelihood [10], as a proxy for the real network likelihood. One can also investigate how the wrong NEPAL-likelihood (where blob-optimization works and where we can define a network CLV) performs if we are in a horizontal move search round.

Alternatively: When enumerating all possible candidates for the given move type, use parsimony scores to decide which candidates to test.

Using Ancestral States Instead of enumerating all candidate networks for a given move type, we can compute ancestral states for each internal node in the network. Using these ancestral states, we can make informed predictions about which moves would lead to a higher-scoring network. We expect the use of ancestral states to be especially promising when pre-identifying promising arc insertion move candidates. Moreover, when doing branch length optimization, we can use the genetic distance between ancestral states in the network before applying a move to provide an initial educated guess for newly inserted or redirected branches.

Pre-identify the most promising move candidates this way, and try them first. If we already found a highly promising BIC-improving network this way, we may skip scoring the remaining move candidates.

We can either find a way to define ancestral states in a network directly, or make use of the ancestral states in its displayed trees. For computing ancestral state genetic distance between two nodes in a network, we can then use the minimum genetic distance between ancestral states in its displayed trees, weighted by displayed tree probability.

Hash already encountered networks In order to not revisit the same network more than once, we suggest keeping all accepted and already encountered network topologies during the search in a hash set (use a LRU cache?).

Avoid isomorphic Networks – Identifiability Issues

- Discuss identifiability issues: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4388854/`
- identifiability issue, different networks can have same displayed trees –¿ graph isomorphism on rooted DAGs for checking network topology
- Skip move candidates that would lead to an indistinguishable network.

identifiability issue, different networks can have same displayed trees –¿ graph isomorphism on rooted DAGs for checking network topology

Inferring large networks – Divide and Conquer The current NetRAX software is still too slow to perform network inference on large dataset with a potentially high number of reticulations directly. The number of candidate moves to evaluate in each filtering step rises with the number of taxa in the dataset. Thus, we propose the following pipeline:

1. Infer a maximum-likelihood phylogenetic tree on the dataset, using RAxML-NG.
2. Delimit species on this tree by using mPTP.
3. Collapse each delimited species in the tree, replacing each species by its most recent common ancestor. Use RAxML-NG to compute ancestral state sequences for the species roots, using them in the reduced MSA.
4. Using NetRAX, infer a phylogenetic network on this reduced dataset.
5. Uncollapse the species in the inferred reduced network, replacing each species root with its species subtree.
6. Perform a last NetRAX search on the uncollapsed network, in order to refine the final network.

We expect that in the last pipeline step, the network does not change much. Intuitively, we expect that the placement of reticulations will undergo minor changes. Thus, we propose restricting the set of candidate moves in the last refinement NetRAX run to moves that relocate a reticulation parent or child within a small radius. The underlying assumption within our proposed solution is that reticulations are less likely to have split up species individuals.

An alternative implementation of the above idea is:

1. Infer RAxML-NG best tree on the entire dataset.
2. Run mptp species delimitation on it.
3. Call NetRAX with both RAxML-NG best tree and mptp species delimination output file.
4. In the NetRAX run, do the following:
   (a) For each delimited species root, mark all nodes in the subtree rooted at the species root as FORBIDDEN.
   (b) Do a network inference search where we do not allow moves to include any of the FORBIDDEN nodes.
   (c) Remove the FORBIDDEN mark from all nodes.
   (d) Do a second network inference search run, starting from the best inferred network so far.

Actually collapse simple paths instead of using fake nodes If we want to reduce the number of CLV update operations and are willing to increase code complexity, we can collapse simple paths on a per-displayed-tree basis.

Improve Dealing with Numerical Issues Avoid using the slow MPFR multiprecision library: implement Kahan's summation algorithm (`https://en.wikipedia.org/wiki/Kahan_summation_algorithm`), compute faster logarithms.

18.2 Improving Inference Quality, Escaping Local Optima

Avoid local optima Replace the simple greedy network search algorithm by reversible-jump MCMC or a simulated annealing approach.

Improving Inference Quality – Promising State Queue NetRAX currently does not provide checkpointing. After adding checkpointing functionality that allows us to restart the search from any intermediary search state, we propose the following approach to improve our inferency quality: Keep alternative choices in a PromisingStateQueue. We can implement this PromisingStateQueue as a priority queue, scoring each alternative promising state by its BIC score. When we have to choose 1 out of 5 highly promising candidates in the choosing phase, why discard the others? In order to not revisit the same network more than once, we suggest keeping all accepted and already encountered network topologies during the search in a hash set (use a LRU cache?). This way, when restarting the search from another promising state than the one taken in the initial search phase, we avoid revisiting the same network topology more than once and can stop the redundant repeated search steps that would follow from such a network. We expect the PromisingStateQueue approach to help NetRAX avoid getting stuck in local optima, while at the same time leading to faster total inference time compared to entirely restarting the search from multiple random or parsimony trees.

Improve Network Root Choice ... Midpoint rooting on start tree?

Scrambling try to escape local optima by scrambling the best found network

Better start networks Start from maximum parsimony networks.

Delay BIC Decision I figured out why NetRAX sometimes infers 1 reticulation less than the true number of reticulations: It's a design flaw in the search algorithm! In the current version, if the best found arc insertion move before the search stops led to a worse BIC than we had for the network before, the search stops. Even if that arc insertion move led to a slightly better loglikelihood! Instead, what we should do is performing an additional search run where we take the arc insertion move that worsened BIC the least, and do a full round of horizontal moves on it. Only after that we should decide whether to stop the search or not. This change would require some medium-sized code changes though. Thus I am just adding it to the Future Work section.

18.3 Additional Features

More network likelihood Definitions Support more network likelihood definitions. Keep in mind that there is work which uses HMMs to model site dependency: `https://pdfs.semanticscholar.org/592f/40608fb1fe5b642d790b054551e1e7ef6135.pdf`

Improve Simulation of Phylogenetic Networks Do more simulations 3 simulators (celine, sarah, empirical datasets)

Support scaled branch lengths model Most of the NetRAX codebase already supports the scaled branch lengths model (where partitions share branch lengths, but each partition has its own scaling factors). Only the function that computes displayed tree likelihood derivatives still has a missing TODO in it.

Improve General Software Quality and User-Friendliness

- Implement checkpointing.
- Add CI/CD and some better unit tests for it.
- Improve software quality (more unit tests, check with softwipe, test if it works on a Mac, add checkpointing)
- Add support for PTHREADS. Currently, only the MPI version works. It's likely something with the CMakeLists.txt
- Provide a graphical user interface, showing some BIC progress plots, automatically drawing the network after each accepted move - maybe even animate the current network, showing how the moves get performed.
- Write a user manual.
- Improve user friendliness (Write a user manual, make a website for NetRAX, provide a GUI, offer a webservice, support more data formats, add Python and R bindings, distribute NetRAX where biologists can find it (Bioconda etc))

19 Conclusion

TODO (I am waiting for the experiments to finish before writing the conclusion)

20 Appendix: Rationale for not doing branch length optimization on each displayed tree separately and then taking the weighted average

What we tried, but didn't work We tried to optimize branch lengths separately for each displayed tree, and then use the weighted average over the per-displayed-tree optimized branch lengths as branch length for the network. But on random data the optimal-brlen variance among displayed trees was too high. Also, we found out that we can not easily map branches close to reticulation nodes between displayed trees, as in a displayed tree these branches exist only as a sum of multiple network branches (see Figure below). For the same reason, it is also difficult (if not impossible) to properly map back branches of displayed trees to the network. Even if we fix the outgoing edge of a reticulation to have branch length zero, we still have problematic branches (problematic here means that in some displayed trees, the network branch does not appear exactly, but only as a sum of multiple network branches on a path).

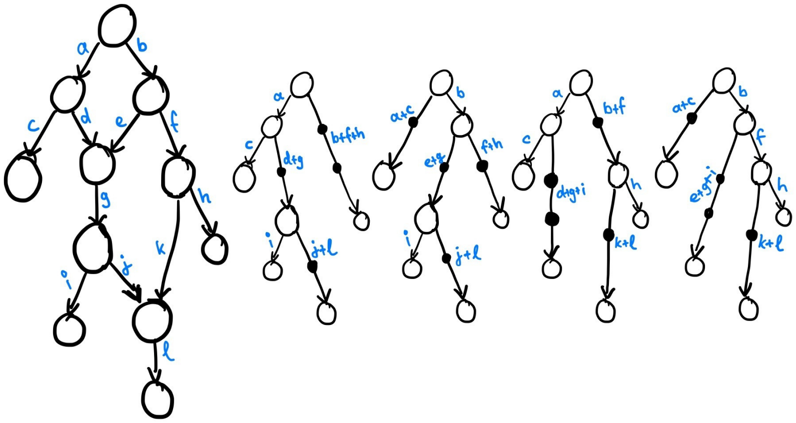


Fig. 19: Some network branches (e.g., branch d) are not present in any displayed tree of the network (after collapsing simple paths), but only appear as part of a sum of branches.

21 Appendix: Wrapping and Adapting libpll/pll-modules/RAxML-NG to work with Networks

supplementIf we put this in the supplement, we need a recall of the libpll and pll-modules/ Otherwise the reader cannot understand this part. In order to understand network likelihood computation in NetRAX, it is crucial to understand how tree likelihood computation in libpll and pll-modules works. In NetRAX, we extend the approach of libpll and pll-modules to networks. In particular, the reader must be familiar with the Felsenstein pruning algorithm and the per-node conditional likelihood vectors (CLVs) used in the algorithm. One also needs to know that libpll updates CLVs by specifying an operations array. An entry in this operations array is of type pll_operation_t and consists (among other things, like scaler indices and pmatrix indices) of the parent clv index, the clv index of the first child, the clv index of the second child. We must provide these entries such that they correspond to a post-order traversal of the subtree of interest. A clv index is the identifier of a node, and a pmatrix index is the identifier of a branch.

A detailed explanation of CLVs and tree likelihood computation is available here: `https://github.com/xflouris/libpll/wiki/Computing-the-likelihood-of-a-tree`

21.1 Fake Treeinfo

All operations in libpll, pll-modules, and RAxML-NG require phylogenetic trees. In order to make some of them work on networks as well, we implement substantial changes in forked versions of their repositories. Most functions provided by pll-modules require a gigantic pllmod_treeinfo_t data structure. This structure stores, among things useful for us (such as pll_partition_t information), a pll_utree_t tree. For our use with networks, we created a fake treeinfo data structure with an empty tree, but overrode some of its parameters (like number of nodes in the “tree”). We also added function pointers to both pll_treeinfo_t and the TreeInfo class from RAxML-NG. We need these function pointers to call our own network likelihood function within the branch-length and model optimization procedures.

22 Appendix: Network Data Format

We use a similar data structure as the utree in libpll (https://github.com/xflouris/libpll/wiki/Tree-structure). In libpll, a tree is a dynamic, unrooted structure, with a pointer to a "virtual root node". Instead of storing a network as a rooted, hierarchical structure, we store its underlying topology in an unrooted version. This simplifies dynamic changes to the network topology, as required by the topology rearrangement moves (see Section 10). We keep the information about which node in this unrooted topology corresponds to the root node in a "virtual root node" pointer. We further also mantain a vector of pointers to reticulation nodes for easier access. In an unrooted network topology, reticulation nodes are indistinguishable from normal tree nodes. Thus, every node also stores a pointer to a ReticulationData object. For normal tree nodes, this pointer is NULL. The ReticulationData object stores pointers to the first and second parent of a reticulation node, as well as the probabilities of taking the first or second parent of the reticulation node.

The equivalent to a post-order traversal in a tree is a (reversed) topological sort in a directed acyclic graph. We store a vector of pointers to nodes in (reversed) topological traversal order. We need this for filling the libpll operations array. In the libpll operations array, the operations have to in bottom-up order. We use the operations array in the Felsenstein pruning algorithm for likelihood computation (see https://github.com/xflouris/libpll/wiki/Computing-the-likelihood-of-a-tree).

Supported I/O Formats NetRAX reads and writes phylogenetic networks in the Extended Newick Format [3].

22.1 Network I/O and Data Structure

We support both reading and writing the Extended Newick Format for networks (https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-532). When reading the input network file, we first parse the network into an intermediate rooted network data structure (with a top-level trifurcation), and then transform it into our unrooted network data structure with a virtual root node (see image below). We do fault tolerant parsing, ignoring superfluous extra braces around the root node in the extended Newick file.

The final network data structure we use is analogous to the tree data structure used in RAxML-NG (https://doi.org/10.1093/bioinformatics/btz305)/pll-modules(https://github.com/ddarriba/pll-modules)/libpll (https://github.com/xflouris/libpll-2) and in the genesis toolkit (https://doi.org/10.1093/bioinformatics/btaa070). This is, we store the network topology as an unrooted and undirected structure, with a pointer to a designated virtual root node. This virtual root node gives us (most) implicit edge directions. The advantage of this data structure i wanted to say that the link information about child-parent relationship is already in the figure (the child link) and it is confusing because it is not introduced yet is that we can change the root location without explicitly changing the direction of all edges. In contrast to phylogenetic trees, we need to store some additional information to correctly keep track of the reticulation nodes (see paragraph after the image).

We initially parse a phylogenetic network into a rooted data structure. In order to convert the rooted data structure into the unrooted data structure with a virtual root, we merge the root node with one of its non-reticulation children, which creates a toplevel trifurcation. Note that not both children of the network root can be reticulations, since a phylogenetic network is a single-source, directed, acyclic graph.

TODO: Update this part. We switched to always ensuring that we have a toplevel bifurcation in the network data structure.

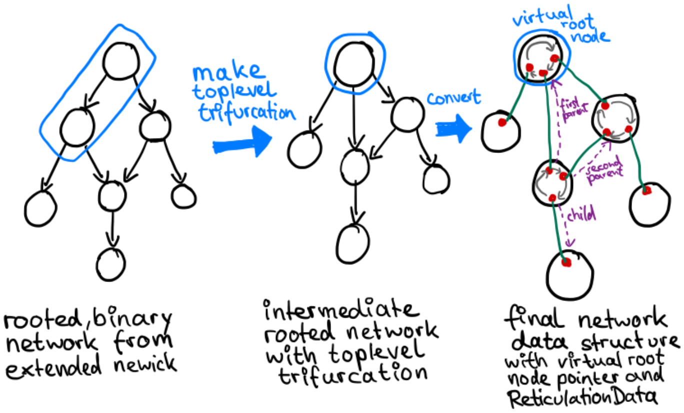

make toplevel trifurcation
convert
virtual root node
first parent
second parent
child
rooted, binary network from extended newick
intermediate rooted network with toplevel trifurcation
final network data structure with virtual root node pointer and ReticulationData

Fig. 20: Converting the phylogenetic network parsed from Extended Newick into the format used within NetRAX.

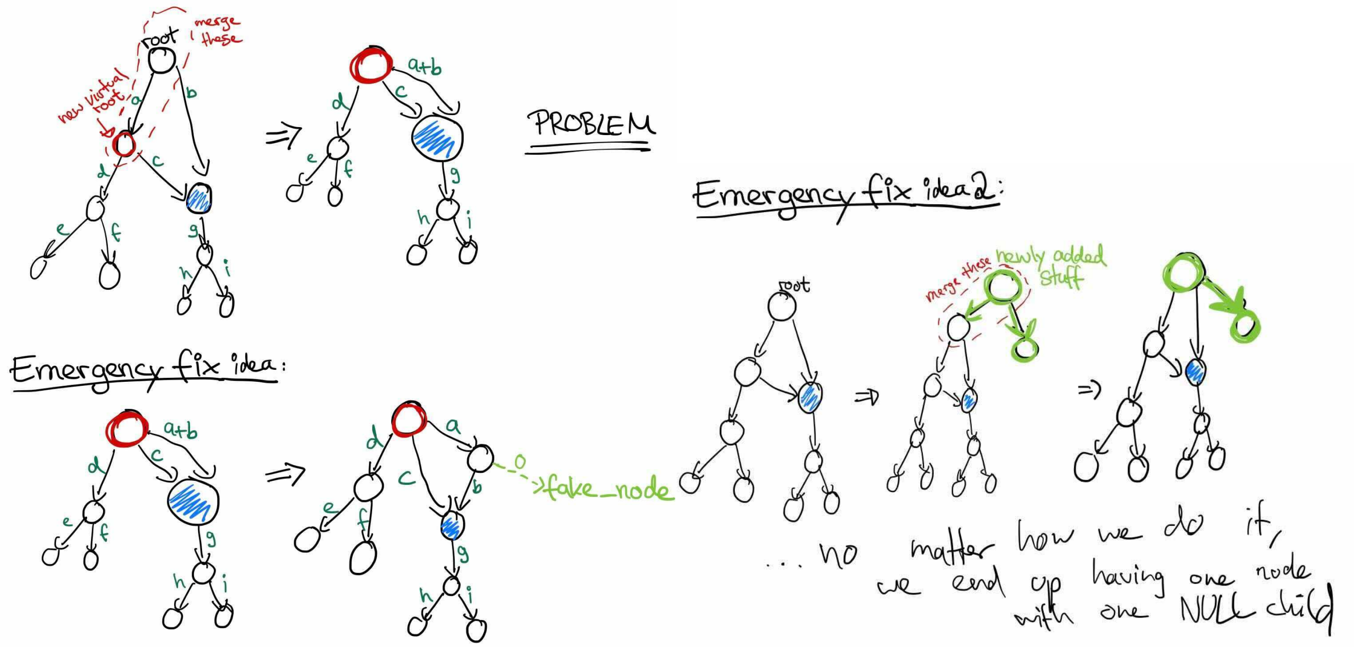


Fig. 21: I need to change some parts in NetRAX, decided to implement Emergency Fix 2.

Handling Reticulation Nodes In a rooted network, a reticulation node has two parents (called first parent and second parent) and one child node. In each displayed tree, exactly one of the parents of each reticulation node is active and the other parent is passive. Each edge to a parent has a probability. We only store the probability of taking the first parent, as the probability of taking the second parent is 1 minus the probability of taking the first parent. We deactivate all edges coming from passive parents in order to obtain a displayed tree. The virtual root node is not sufficient to direct the edges incident to a reticulation node. Thus, each node in our implementation has a NodeType. A node either has the type RETICULATION_NODE or the type BASIC_NODE. Each node stores a unique pointer to a ReticulationData data structure (the pointer is null for non-reticulation nodes). In ReticulationData, we store pointers to both parents of the reticulation node, a pointer to the child of the reticulation node, as well as the probability of taking the first parent node and a flag stating which parent node is currently active/taken.

Important Note about the Virtual Root Choice The following image shows why we need to be cautious about where to place the virtual root, even if we separately store which edges are incoming edges of a reticulation and thus exclude each other. Depending on where we place the virtual root, it does affect in perhaps unexpected ways the edge directions in the displayed trees (what we really don't want because it would rob us of being able to use a modified Felsenstein approach for likelihood computation)too colloquial, but you know that. This is just another reason why we cannot do the branch length optimization on a network the same way as RAxML-NG is doing it on a tree (it re-roots the tree at the branch whose length we want to optimize). Finding all possible placements for the virtual root pointer where the edge directions in the induced displayed trees remain the same is straightforward: We only need to exclude all nodes that reside in the subnetworks "below" reticulation nodes (in their "child" subnetworks).

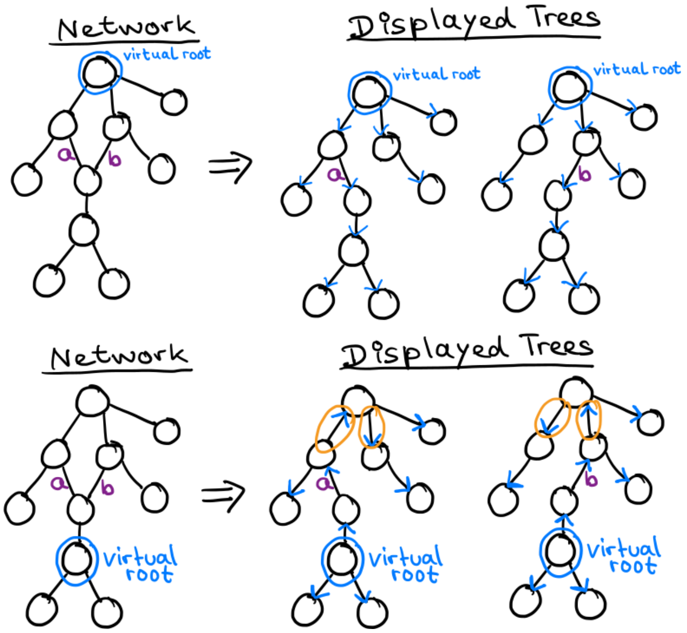


Fig. 22: Virtual root placement affects edge directions. Storing a pointer to a virtual root node and keeping a set of mutually exclusive pairs of edges (the incoming edges of each reticulation) suffices to induce all edge directions.But if we place the virtual root "below" any reticulations, the edge directions in the displayed trees differ. This is bad for reusing CLVs when computing network likelihood.

Overview Overall, we have nodes, links, and edges.

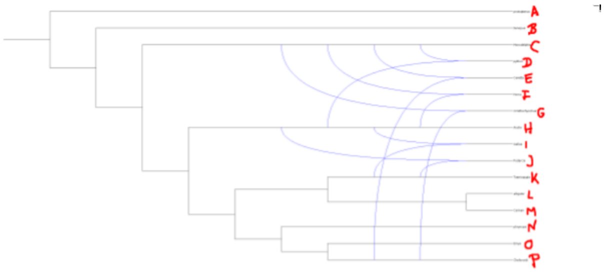
- A node can either be a leaf, an internal tree node, or a reticulation node. A leaf has one link. An internal tree node has 3 links. A reticulation node has 3 links and a non-null unique pointer to a ReticulationData structure, which stores a pointer to the child, a pointer to the first parent, a pointer to the second parent, and a value first_parent_prob (storing the probability of taking the first parent of a reticulation) in the range $[0, ..., 1]$. Each node also has a label. The id of a node is its clv_index.
- A link belongs to a node. It has a pointer to the node it belongs to, to the edge it connects with, and to the outer link. The outer link is in the other node incident to the link's edge. The id of a link is its link_index. do we need this level of detail? supplement?
- An edge has two links. An edge also has a value for its length. The id of an edge is its pmatrix_index.

Restrictions inherited by RAxML-NG/pll-modules/libpll If the network has n tips, then the clv_indices $0..n-1$ belong to tip nodes, and the pmatrix_indices $0...n-1$ belong to edges incident to tip nodes.

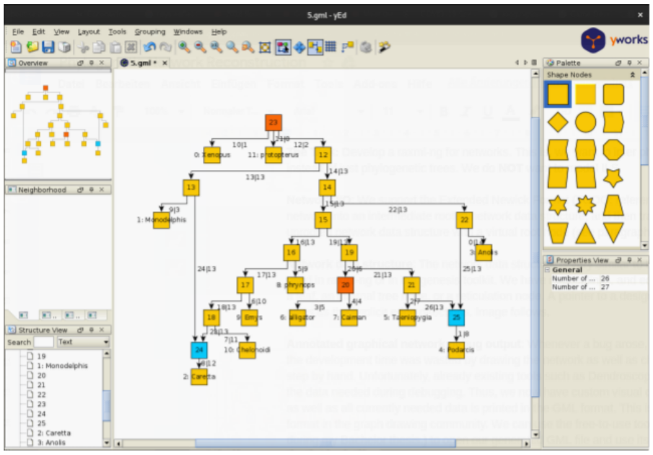
23 Appendix: Debugging

23.1 Annotated Graphical Debug Output

that would go to the supplement Whenever a bug arose, it turned out that we wasted most of the development time by drawing the network as well as changes done to it step by step by hand on paper. Unfortunately, already existing tools such as Dendroscope do not support displaying the data needed during debugging. Moreover, in case of a high number of reticulations, the resulting image obtained from Dendroscope was visually too cluttered to be of use for debugging. Thus, we now have a custom visual debug output. We export rhe network as well as all currently needed data into the GML format, which is a widely used data format in the graph drawing community. We can use the freely available yED Graph Editor (`https://www.yworks.com/products/yed`) to open the generated GML file and use its ”hierarchical layout” function to obtain a helpful readable image for debugging. Unfortunately, we cannot fully automate this step, as this is not allowed in the free version of yEd.

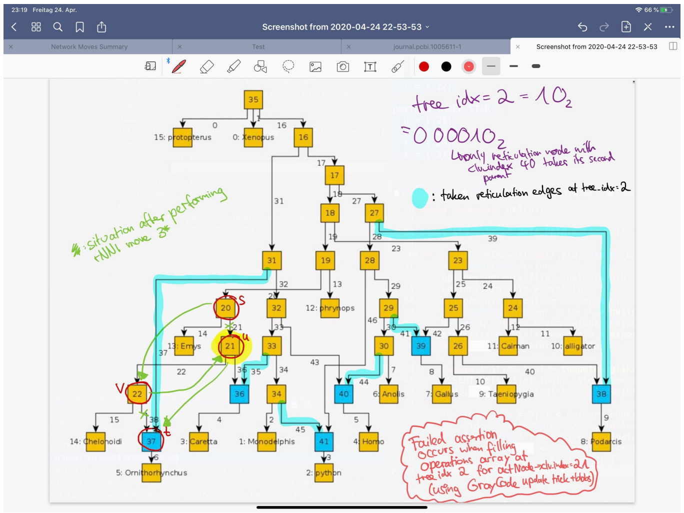


The kind of images one gets from using Dendroscope. This does not help much for debugging our code.



A very helpful network debug image, showing the clv index of each node, the pmatrix index of each edge, the reticulation nodes (in blue), and the megablob roots (in dark orange).

Fig. 23: Example of Dendroscope output vs.õur graphical debug output.



tree idx = 2 = 10$_2$

= 000010$_2$

↳ only reticulation node with clvindex 40 takes its second parent

: taken reticulation edges at tree_idx = 2

: situation after performing rNNI move 3*

Failed assertion occurs when filling operations array at tree idx 2 for actNode->clv_index = 21 (using GrayCode update trick + tables)

Fig. 24: Example showing the usefulness of our graphical debug output.

Further, NetRAX provides extensive debug output. Not only can we convert the displayed trees of a network into pll_utree_t objects, we can also print each of them in NEWICK format. In addition to this standard output, NetRAX can also print detailed graphical debug information by exporting a phylogenetic network into the GML format.

23.2 Automatically Finding Smaller Datasets

I wrote a script for reducing debugging pain for programs that take a partitioned MSA as input. The problem it aims to solve is:

Given A partitioned MSA where your program crashes

Wanted A sub-sampled smaller MSA (less taxa, less sites) where your program already crashes

In its current version, it specializes on NetRAX, only parses MSA in Phylip format, and does some dumb grid search instead of a 2D binary search. But it works. The script discovered a 8 taxa 160 sites dataset where I am already encountering a program crash. This is much more comfortable to debug than the 40 taxa 16000 sites dataset I started from.

References

1. libpll-2. `https://github.com/xflouris/libpll-2.git`
2. pll-modules. `https://github.com/ddarriba/pll-modules`
3. Cardona, G., Rosselló, F., Valiente, G.: Extended newick: it is time for a standard representation of phylogenetic networks. BMC bioinformatics 9(1), 1–8 (2008)
4. Felsenstein, J.: Evolutionary trees from dna sequences: a maximum likelihood approach. Journal of molecular evolution 17(6), 368–376 (1981)
5. Gambette, P., Van Iersel, L., Jones, M., Lafond, M., Pardi, F., Scornavacca, C.: Rearrangement moves on rooted phylogenetic networks. PLoS computational biology 13(8), e1005611 (2017)
6. Glémin, S., Scornavacca, C., Dainat, J., Burgarella, C., Viader, V., Ardisson, M., Sarah, G., Santoni, S., David, J., Ranwez, V.: Pervasive hybridizations in the history of wheat relatives. Science advances 5(5), eaav9188 (2019)
7. Holoborodko, P.: Mpfr c++. `http://www.holoborodko.com/pavel/mpfr/`
8. Jin, G., Nakhleh, L., Snir, S., Tuller, T.: Maximum likelihood of phylogenetic networks. Bioinformatics 22(21), 2604–2611 (08 2006), `https://doi.org/10.1093/bioinformatics/btl452`
9. Kozlov, A.M., Darriba, D., Flouri, T., Morel, B., Stamatakis, A.: Raxml-ng: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference. Bioinformatics 35(21), 4453–4455 (2019)

10. Nguyen, Q., Roos, T.: Likelihood-based inference of phylogenetic networks from sequence data by phylodag. In: International Conference on Algorithms for Computational Biology. pp. 126–140. Springer (2015)
11. Park, H.J., Nakhleh, L.: Inference of reticulate evolutionary histories by maximum likelihood: the performance of information criteria. In: BMC bioinformatics. vol. 13, pp. 1–10. BioMed Central (2012)
12. Perera, A.: Finding the optimal number of clusters for k-means through elbow method using a mathematical approach compared to graphical approach. `https://www.linkedin.com/pulse/finding-optimal-number-clusters-k-means-through-elbow-asanka-perera/`
13. Thorndike, R.L.: Who belongs in the family? Psychometrika 18(4), 267–276 (1953)
14. Wen, D., Yu, Y., Zhu, J., Nakhleh, L.: Inferring phylogenetic networks using phylonet. Systematic biology 67(4), 735–740 (2018)
15. Zhang, C., Ogilvie, H.A., Drummond, A.J., Stadler, T.: Bayesian inference of species networks from multilocus sequence data. Molecular biology and evolution 35(2), 504–517 (2018)