Delft University of Technology

Bsc report applied mathematics

Tail move rearrangement algorithm for rooted binary phylogenetic networks

Dutch title: Tail move herschikkingsalgoritme voor gewortelde binaire fylogenetische netwerken

Selma Husanovic

Supervisors:
Remie Janssen
Leo van Iersel

August 12, 2020

Contents

1 Introduction

Phylogenetic networks are graphs that depict the evolutionary history of species. In *phylogenetics*, a biological branch that studies the relatedness of groups of organisms, these types of networks play a prominent role. From molecular data of taxa, phylogenetic networks are reconstructed and subsequently studied, in order to gain insight into the processes underlying evolution [1].

Figure 1 gives a simple example of a phylogenetic network. The leaves are labeled and represent related species. The graph is directed and out-branching, the direction of the edges indicating the direction of evolution, which is why this is also called a *rooted* phylogenetic network. It has nodes that correspond to the event of one species splitting up into two (e.g. the blue node). Moreover, the network is characterized by reticulation events where two species merge or transfer some genetic material, resulting into one species [2] (e.g. the red node). Since each node has at most two outgoing edges, the graph in Figure 1 is an example of a rooted *binary* phylogenetic network.

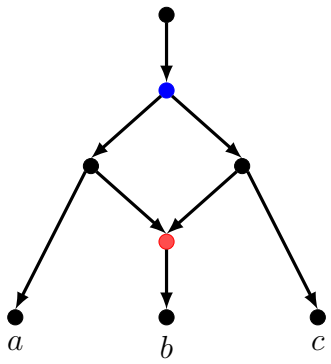

Figure 1: A simple example of a rooted binary phylogenetic network. The blue node represents a split, the red one a reticulation.

A from DNA-data reconstructed phylogenetic network is merely one possible evolutionary history of a group of organisms. When conducting research into the true relatedness of this set of species, it is of great importance to take all possible evolutionary histories into account. In other words: explore the *space of phylogenetic networks*. This is realised by applying *rearrangement moves* to the original network, in order to generate alternative ones from it [3]. These moves, when applied systematically, have the property to transform a given network into any other similar one. The focus of this report will be on one of such rearrangement moves, called the *tail move* (see Figure 2).

Recently, it has been shown by Janssen et al. [4] that, given a rooted binary phylogenetic network, it is possible to generate any other alternative phylogenetic network with the same number of reticulations on the same leaf set, using tail moves only. On top of that, an upper bound on the number of tail moves necessary to realise this is found. The upper bound proof is constructive, which means it is actually possible to find a sequence of tail moves that turns one network into another.

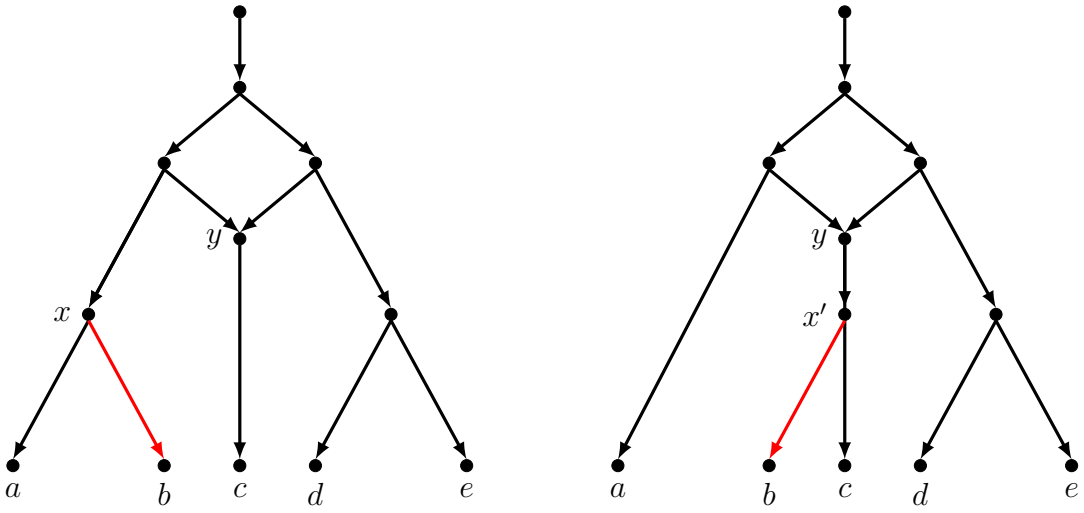

Figure 2: Example of a *tail move*: in the left network the tail of edge (x, b) is moved to (y, c), which results in the network on the right.

The aim of this report is to implement the constructive upper bound proof (of Lemma 4.6 in [4]) as an algorithm, and to assess the quality of its results. The workings of this tail move rearrangement algorithm should be such that, given two networks with the same number of reticulations on the same leaf set, it finds a sequence of tail moves to transform one network into the other. This means the algorithm also gives an upper bound on the *tail distance* (the minimal number of tail moves needed to make the transformation). The quality of the distance as determined by the algorithm shall be evaluated, by comparing it to the exact tail distance for a range of small networks. In order to improve upon the algorithm performance, four improvement proposals will be formulated and tested. Ultimately, the aim is to determine which version of the algorithm, if any, performs best and calculates a distance that is closest to the tail distance.

2 Definitions and properties

This section gives an overview of the mathematical definitions and corresponding properties from [4] that are of interest in this report.

2.1 Phylogenetic networks

Definition 2.1. A *directed graph* is a graph in which each edge is assigned a direction. Formally, a directed graph is a pair $D = (V, E)$ where V is a finite set of nodes and E is a set of ordered pairs of nodes called edges. For an edge (x, y) we say that x is the *tail* and y the *head*. The direction of the edge is indicated by an arrow pointing towards the head. D is called *acyclic* if it contains no circuits.

Definition 2.2. In a directed graph D the *indegree* of a node v is the number of edges in D whose head is v. Similarly, the *outdegree* of v is the number of edges in D whose tail is v.

Definition 2.3. In a directed graph a *leaf* is a node with indegree one and outdegree zero. A *root* is a node with indegree zero and outdegree one. A *tree node* is a node with indegree one and outdegree two. A *reticulation* is a node with indegree two and outdegree one.

Definition 2.4. Let X be a finite set of taxa. A *rooted binary phylogenetic network* $N = (V, E)$ is a directed acyclic graph with no parallel edges. The leaves are bijectively labelled by X. There is a unique root and all other nodes are either tree nodes or reticulations. We write $V(N)$, $L(N)$ to denote the nodes en leaves of N, respectively. For a set of nodes $Y \subseteq V$, we write $N[Y]$ to denote the subgraph of N induced by Y, and we write $N \setminus Y$ to denote $N[V \setminus Y]$. We write $|N|$ to denote the number of nodes in N.

Let u and v be nodes in a network. We say u is *above* v and v is *below* u if $u \leq v$ in the order induced by the directed graph underlying the network. Similarly, if $e = (x, y)$ is a directed edge of the network and u a node, then we say that e is *above* u if y is above u and e is *below* u if x is below u. Let $e = (u, v)$ be an edge of a phylogenetic network, then we say that u is a *parent* of v, or u is *directly above* v and v is a *child* of u, or v is *directly below* u. See Figure 3 for an illustration.

Definition 2.5. Let N, N' be two directed acyclic graphs (includig networks) with some nodes labelled with X. Then, an *isomorphism* between N and N' is a bijection $\phi : V(N) \to V(N')$ such that:

- Two nodes u, $v \in V(N)$ are adjacent in N if and only if $\phi(u)$ and $\phi(v)$ are adjacent in N'.

- For any labelled node $u \in L(N)$, $\phi(u)$ is the node $L(N')$ that has the same label as u.

We say N and N' are *isomorphic* if there exists an isomorphism between N and N'.

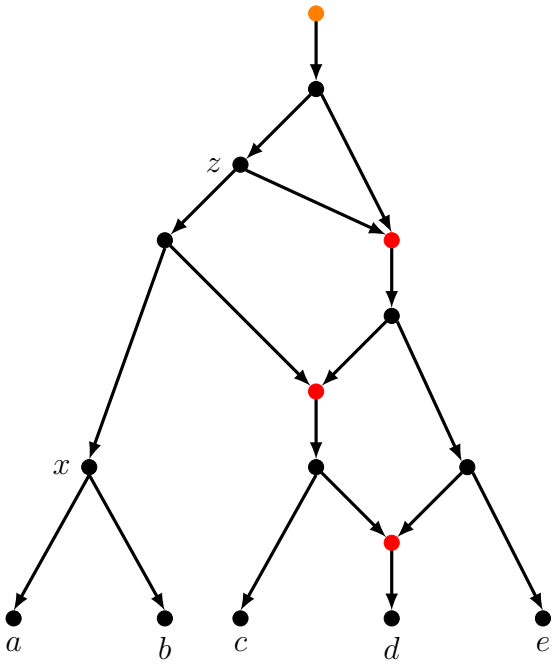

Figure 3: Another example of a rooted binary phylogenetic network with five labelled leaves, a root (orange) and three reticulations (red). The parent of a is x and the children of x are a and b. Furthermore, we can see that z is above all other labelled nodes and also above, for example, edge (x, a).

Lemma 2.6. *Any two rooted binary phylogenetic networks on the same leaf set X, with the same number of reticulations, also have the same number of nodes and edges.*

Proof. Let $N = (V, E)$ be a rooted binary phylogenetic network on X with t tree nodes and r reticulations. Then we have that the number of nodes is equal to:

$$|V| = t + r + |X| + 1.$$

Now, the sum of indegrees of the nodes of N is equal to the number of edges $|E|$, which in turn is equal to the sum of outdegrees of the nodes:

$$t + 2r + |X| = |E| = 1 + 2t + r.$$

From this it follows that $|V| = 2r + 2|X|$ and $|E| = 3r + 2|X| - 1$. □

2.2 Tail moves

Definition 2.7. Let $e = (u, v)$ and f be edges of a network. A *tail move* of e to f consists of the following steps:

1. Deleting e;
2. Suppressing the indegree-1 outdegree-1 node u;
3. Subdividing f with a new node u';

4. Adding the edge (u', v).

A tail move is only allowed if the resulting directed graph is still a network, in correspondence with Definition 2.4.

Figure 4: A tail move as described in Definition 2.7, with from-edge (x, y).

A tail move has the property of being reversible. The new edge that arises after deleting e and suppressing the indegree-1 outdegree-1 node u, is called the *from-edge* of the tail move. This edge is where the moved tail originally was attached to and where it should return to, in order to reverse the move, see Figure 4.

Definition 2.8. Let $e = (u, v)$ be an edge in a network. Then e is called *non-movable* if u is the root, if u is a reticulation, or if the removal of e followed by suppressing u creates parallel edges. Otherwise, e is called *movable*.

There is only one possible scenario in which moving a tail of edge (u, v) can result in parallel edges: when the parent of u forms a triangle with u and the child of u not being v:

Definition 2.9. Let N be a network and let x, u and y be nodes of N. We say that x, u and y form a *triangle* if there are edges (x, u), (u, y) and (x, y). The edge (x, y) is called the *long edge* and (u, y) is called the *bottom edge* of the triangle.

The interesting case now is when u is a tree node:

Observation 2.10. *Let x, u and y form a triangle in a network, in the same way as in definition 2.9, and let v be the other child of u. The edge (u, v) is non-movable because this would create parallel edges from x to y. However, the edges (u,y) and (x,y) are movable, and if (u, y) is moved sufficiently far up (i.e., destroying the triangle), the new edge (x, v) is also movable.*

An illustration of Definition 2.9 and Observation 2.10 can be seen in Figure 5. Due to the binary nature of the networks considered in this paper, the following is a direct consequence of Observation 2.10:

Observation 2.11. *Let u be a tree node, then at least one of its child edges is movable, because at most one of these has its tail in a triangle not containing its head.*

Note that movability of an edge does not imply that a valid tail move is possible. It only tells us that the tail can be safely removed, without violating the definition of a network, not that it can be reattached anywhere else. The following observation characterized valid tail moves:

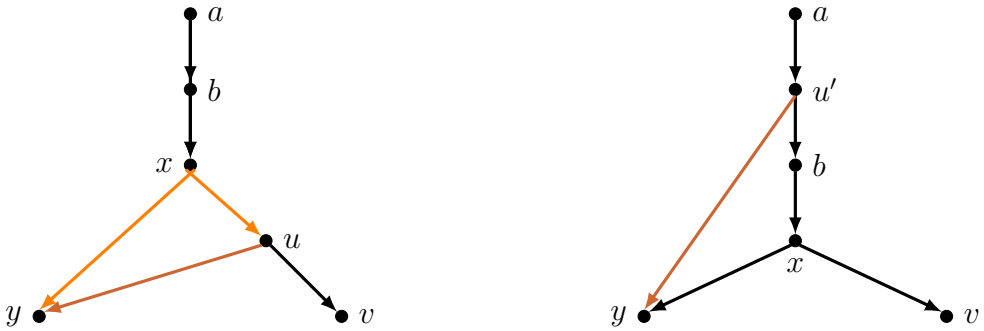

Figure 5: Illustration of Definition 2.9 and Observation 2.10. On the left, x, u and y form a triangle with long edge (x, y) and bottom edge (u, y). The edge (u, v) is non-movable, since deleting it and suppressing u would create parallel edges between x and y. However, (u, y) is movable and when moving its tail to (a, b), we obtain the network fragment on the right. Now (x, v) can be moved.

Observation 2.12. *The tail of an edge $e = (u, v)$ can be moved to another edge $f = (s, t)$ if and only if the following conditions hold:*

1. *e is movable;*
2. *f is not below v;*
3. *$t \neq v$*

The first condition ensures that the tail can be removed, the second that no cycles are created and the third that no parallel edges are created. Note that these conditions imply that moving an edge *up* is allowed, i.e. moving it to another edge that is above it. In particular, a movable tail can always be moved to the root edge. However, note that it is not certain that this results in a non-isomorphic network.

Figure 6 depicts two networks for which there are no non-trivial tail moves leading to a different network. The movable edges are colored green. Note that every valid tail move involving these edges either results in exactly the same network, or an isomorphic one.

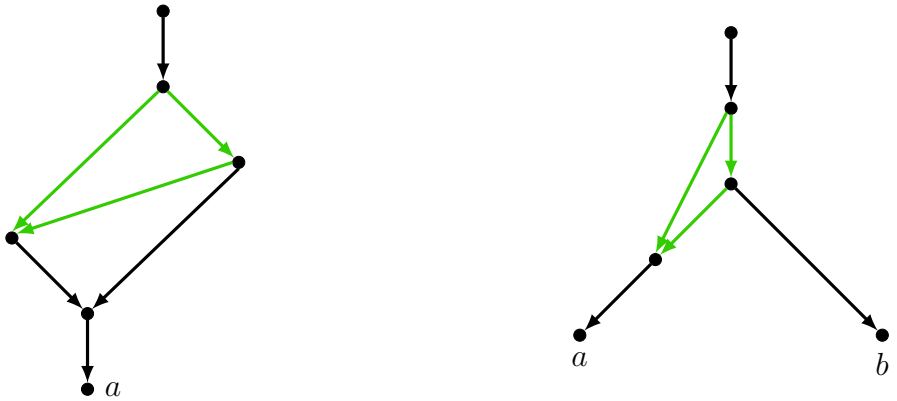

Figure 6: The networks for which there are no non-trivial tail moves.

2.3 Tail distance

Definition 2.13. Let N and N' be two rooted binary phylogenetic networks on the same leaf set X, with the same number of reticulations. We define the *tail distance* $d_{\text{tail}}(N, N')$ as the minimum length of a sequence of tail moves turning N into N'.

The following definition is important for the subsequent Lemma:

Definition 2.14. Given a network N and a set of nodes Y, we say that Y is *downward-closed* if for any $u \in Y$, every child of u is in Y.

Lemma 2.15. (Lemma 4.6 in [4]). *Let N and N' be rooted binary phylogenetic networks on the same leaf set X, with the same number of reticulations, such that they are not the networks depicted in Figure 6. Let $Y \subseteq V(N)$ and $Y' \subseteq V(N')$ be downward-closed sets of nodes such that $L(N) \subseteq Y$, $L(N') \subseteq Y'$ and $N[Y]$ is isomorphic to $N'[Y']$. Then there is a sequence of at most $3|N \setminus Y|$ tail moves turning N into N'.*

Lemma 2.15 plays an important role in this paper, since the tail move rearrangement algorithm that is formulated in the following section is based on the constructive proof of this Lemma (see [4]).

3 Algorithm construction

In this section, the first version of the tail move rearrangement algorithm is formulated, which will form the basis for further improvements. First, a conceptualisation is given, explaining the main working principles. Then, a tail move reversal algorithm (Algorithm 1) is formulated and proven, which is used by the rearrangement algorithm. Lastly, the rearrangement algorithm itself (Algorithm 2) is precisely formulated and proven.

3.1 Conceptualisation

The rearrangement algorithm starts with two rooted binary phylogenetic networks, say N and N', on the same leaf set and with the same number of reticulations. Since they are on the same leaf set, there is an isomorphism between the networks induced by the leaves. For simplicity we write $Y = L(N)$ and $Y' = L(N')$. So there already exists an isomorphism between $N[Y]$ and $N'[Y']$. The aim is to systematically increase the sets Y and Y', by adding each iteration one node from $V(N) \setminus Y$ to Y and one node from $V(N') \setminus Y'$ to Y', such that this extends the isomorphism between the new $N[Y]$ and $N'[Y']$. This continues until $Y = V(N)$ and $Y' = V(N')$, thus having extended the isomorphism to an isomorphism between N and N'.

The algorithm employs tail moves to rearrange either network N or N' per iteration, in order to pave the way for an isomorphism extension. This is done in the following manner. In what follows, a *lowest node* of $N \setminus Y$ (or $N' \setminus Y'$) is a node in $V(N) \setminus Y$ (or in $V(N') \setminus Y'$) such that all descendants of this node are in Y (or in Y'). Taking N' as starting point, the algorithm looks for a *lowest reticulation* of $N' \setminus Y'$. This is a lowest node of $N \setminus Y$, which is a reticulation. Let x' be the single child of u', which then is in Y'. This means that there is a node x in Y such that x' is mapped to x, by the isomorphism between $N[Y]$ and $N'[Y']$. If x has a *reticulation parent* in $N \setminus Y$, a parent node u which is a reticulation, then the isomorphism can be extended, see Figure 7.

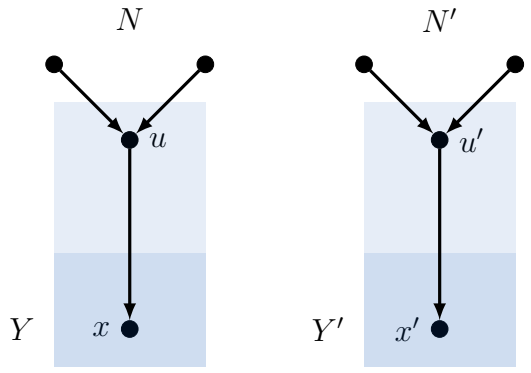

Figure 7: If u' is a lowest reticulation in $N' \setminus Y'$ with child x', then $x' \in Y'$. There is a $x \in Y$ corresponding to x'. If x has a reticulation parent u, the algorithm adds u to Y, u' to Y' and extends the isomorphism by letting u be mapped to u'.

If x does not have a reticulation parent in $N \setminus Y$, the isomorphism cannot be extended directly. There are multiple cases possible then, depending on the structure of the network near x. In each case, the algorithm uses a maximum of three tail moves per iteration to locally rearrange network N, such that x ends up with a reticulation parent and the situation is brought to the same as in Figure 7.

In the case that N' does not have a lowest reticulation, the algorithm turns to network N and looks for a lowest reticulation there. The same process is repeated, but with the roles of N and N' reversed. If neither N nor N' has a lowest reticulation, the algorithm selects a lowest tree node of $N' \setminus Y'$. The children of this node, say x' and y', are mapped to corresponding nodes x and y in Y, again by the isomorphism between $N[Y]$ and $N'[Y']$. If x and y have a common parent in $N \setminus Y$, the isomorphism can easily be extended, see Figure 8. If this is not the case, the algorithm uses tail moves to realise a common parent for x and y, thus making extension possible again.

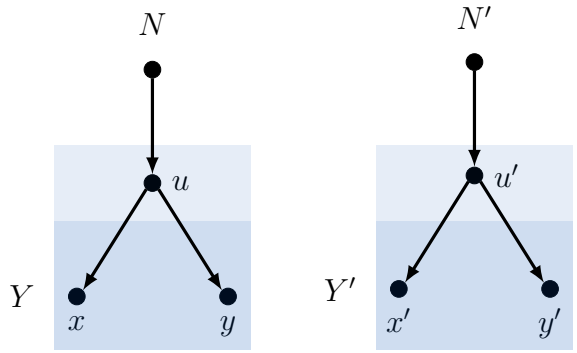

Figure 8: If u' is a lowest reticulation in $N' \setminus Y'$ with children x' and y', then there are nodes $x, y \in Y$ corresponding to x' and y'. If x and y share a reticulation parent u, the algorithm adds u to Y, u' to Y' and extends the isomorphism by letting u be mapped to u'.

The result of the algorithm is an upper bound on the distance between N and N', which is the total number of tail moves used in all the iterations. Next to this, the algorithms keeps track of the tail moves and their order of occurrence for both N and N' separately. The tail moves that occur in N' have the additional record of their from-edges. In the final step, the algorithm combines all information to generate a sequence of moves that directly transforms N into N'. This means that tail moves in N' need to be reversed. This is achieved using Algorithm 1, see the following subsection.

3.2 Tail move reversal algorithm

Algorithm 1

Input:
− isomorphic networks N and N' with the same leaves and same number of reticulations
− $\mu^{N'} = (\mu_1^{N'}, ..., \mu_k^{N'})$ sequence of tail moves in N'' which turns N'' into N'
− $\epsilon^{N'} = (\epsilon_1^{N'}, ..., \epsilon_k^{N'})$ sequence of from-edges corresponding to tail moves in $\mu^{N'}$
Output: τ^N sequence of k tail moves in N such that it reverses the tail moves in $\mu^{N'}$

1: **procedure** Tail Move Reversal
2: $\tau^N = (\ )$
3: $\widetilde{\mu}^{N'} := (\mu_k^{N'}, ..., \mu_1^{N'})$ and $\widetilde{\epsilon}^{N'} := (\epsilon_k^{N'}, ..., \epsilon_k^{N'})$
4: **for all** tail moves (x', y') to (p', q') in $\widetilde{\mu}^{N'}$ with from-edge (m', n') in $\widetilde{\epsilon}^{N'}$ **do**
5: $x'_1 :=$ node in N' arisen from subdividing (p', q')
6: $x, y, m, n :=$ nodes in N which are mapped to x'_1, y', m', n' in N' respectively

execute tail move (x'_1, y') to (m', n') in N'
execute tail move (x, y) to (m, n) in N and add move to τ^N
end for
end procedure

Lemma 3.1. *Let N and N' be isomorphic rooted binary phylogenetic networks on the same leaf set X, with the same number of reticulations. Let $\mu^{N'}$ be a non-empty sequence of tail moves in a network N'', such that it transforms N'' into N'. Let $\epsilon^{N'}$ be a sequence of from-edges corresponding to the tail moves in $\mu^{N'}$. Algorithm 1 constructs a sequence of tail moves τ^N in N, such that it reverses the tail moves in $\mu^{N'}$.*

Proof. First we prove the correctness of Algorithm 1. Since $\mu^{N'}$ is non-empty, we know that we can write $\mu^{N'} = (\mu_1^{N'}, ..., \mu_k^{N'})$ for some $1 \leq k < \infty$. Here each $\mu_i^{N'}$ is a tail move in N'' from an edge e_i to an edge f_i, with $1 \leq i \leq k$. We also know that we can write $\epsilon^{N'} = (\epsilon_1^{N'}, ..., \epsilon_k^{N'})$ for some $1 \leq k < \infty$. Here $\epsilon_i^{N'}$ is the from-edge corresponding to tail move $\mu_i^{N'}$, $1 \leq i \leq k$.
Define $\widetilde{\mu}^{N'}$ as the sequence reversal of $\mu^{N'}$, given by $\widetilde{\mu}^{N'} = (\mu_k^{N'}, ..., \mu_1^{N'})$. In the same way define $\widetilde{\epsilon}^{N'} = (\epsilon_k^{N'}, ..., \epsilon_k^{N'})$. The first member of $\widetilde{\mu}^{N'}$ is the k^{th} tail move done in N'', say moving $e_k = (x', y')$ to $f_k = (p', q')$, which results into N'. Let x'_1 be the new node arisen from subdividing f_k. Since N and N' are isomorphic, we can find nodes $x, y \in V(N)$ such that x is mapped to x'_1 and y is mapped to y'. By the adjacency preserving property of an isomorphism, we know that N contains edge (x, y). The first member of $\widetilde{\epsilon}^{N'}$ is the from-edge, say (m', n') in N' corresponding to $\mu_k^{N'}$. Similarly, we find that there are nodes $m, n \in V(N)$ such that m is mapped to m' and n is mapped to n'. Moreover, we know that N contains edge (m, n). By the reversible property of the tail move, we may reverse the k^{th} move in N'' by moving (x'_1, y') back to (m', n') in N'. But this now is equivalent to moving the tail of (x, y) to (m, n) in N. We call this latter move τ_1^N and let it be the first member of τ^N. After applying this move, N and N' are again isomorphic.
We now look at the second member of $\widetilde{\mu}^{N'}$, which is the $(k-1)^{\text{th}}$ tail move in N''. Now this move results into N'. The second member of $\widetilde{\epsilon}^{N'}$ is the from-edge in N' corresponding to $\mu_{k-1}^{N'}$. Following the same reasoning as before, we can find a tail move in N which is equivalent to reversing the $(k-1)^{\text{th}}$ tail move in N'' and define it τ_2^N. After executing both moves, the isomorphism between N and N' is maintained. We can repeat this procedure for every member of $\widetilde{\mu}^{N'}$ (and correspondingly every member of $\widetilde{\epsilon}^{N'}$). Since in each iteration the isomorphism between N and N' is maintained, we can always find nodes in $V(N)$ which correspond to the nodes in $V(N')$ needed to reverse the tail move. From this, the correctness follows.

Since Algorithm 1 iterates once for each of the k members of $\widetilde{\mu}^{N'}$, with $1 \leq k < \infty$, it follows that it is finite. □

3.3 Tail move rearrangement algorithm

Algorithm 2

Input:
− networks N and N' with the same leaves and same number of reticulations
− $Y = L(N)$ and $Y' = L(N')$
Output: μ^N sequence of tail moves that transforms N into N'

```
1:  procedure Tail Move Rearrangement
2:      distance = 0
3:      while |N \ Y| ≠ 1 do
4:          determine all lowest nodes of N' \ Y'
5:          if there exists a lowest reticulation of N' \ Y' then
6:              u' := arbitrary lowest reticulation of N' \ Y'
7:              x' := child of u'
8:              x := node in Y which is mapped to x' in Y'
9:              z := arbitrary parent of x not in Y
10:             if z is a reticulation then
11:                 add z to Y and add u' to Y'
12:                 let z be mapped to u'
13:                 continue
14:             else if z is not a reticulation then
15:                 if (z, x) is movable then
16:                     u := arbitrary reticulation in N \ Y
17:                     v := child of u
18:                     if x = v then
19:                         add u to Y and add u' to Y'
20:                         let u be mapped to u'
21:                         continue
22:                     else if z = v then
23:                         w := arbitrary parent of u
24:                         y := other child of z
25:                         tail move (z, y) to (w, u) in N
26:                         distance = distance +1
27:                         add u to Y and add u' to Y'
28:                         let u be mapped to u'
29:                         continue
30:                     else
31:                         tail move (z, x) to (u, v) in N
32:                         z_1 := new node arisen from move
33:                         w := arbitrary parent of u
34:                         tail move (z_1, v) to (w, u) in N
35:                         distance = distance +2
36:                         add u to Y and add u' to Y'
37:                         let u be mapped to u'
38:                         continue
```

```
        end if
    else if (z, x) is not movable due to parallel edges then
        find nodes c and d which form a triangle with z with long edge (c, d)
        b := parent of c
        if b is not the root of N then
            a := arbitrary parent of b
            tail move (c, d) to (a, b) in N
            distance = distance +1
            u := arbitrary reticulation in N \ Y
            v := child of u
            if x = v then
                add u to Y and add u' to Y'
                let u be mapped to u'
                continue
            else if z = v then
                w := arbitrary parent of u
                tail move (z, d) to (w, u) in N
                distance = distance +1
                add u to Y and add u' to Y'
                let u be mapped to u'
                continue
            else
                tail move (z, x) to (u, v) in N
                z_1 := new node arisen from move
                tail move (z_1, v) to (w, u) in N
                distance = distance +2
                add u to Y and add u' to Y'
                let u be mapped to u'
                continue
            end if
        else if b is the root of N then
            e := child of d in N
            if x is a tree node then
                s := arbitrary child of x
                t := other child of x
                if t = e then
                    tail move (x, s) to (d, e) in N
                    distance = distance +1
                    add d to Y and add u' to Y'
                    let d be mapped to u'
                    continue
                else if s = e then
                    tail move (x, t) to (d, e) in N
                    distance = distance +1
                    add d to Y and add u' to Y'
                    let d be mapped to u'
```

85: **continue**
86: **else**
87: tail move (x, s) to (d, e) in N
88: x_1 := new node arisen from move
89: tail move (x_1, e) to (z, t) in N
90: x_2 := new node arisen from move
91: tail move (x_2, t) to (d, s) in N
92: distance = distance +3
93: add d to Y and add u' to Y'
94: let d be mapped to u'
95: **continue**
96: **end if**
97: **else if** e is a tree node **then**
98: s := arbitrary child of e
99: t := other child of e
100: **if** $t = x$ **then**
101: tail move (e, s) to (z, x) in N
102: distance = distance +1
103: add d to Y and add u' to Y'
104: let d be mapped to u'
105: **continue**
106: **else if** $s = x$ **then**
107: tail move (e, t) to (z, x) in N
108: distance = distance +1
109: add d to Y and add u' to Y'
110: let d be mapped to u'
111: **continue**
112: **else**
113: tail move (e, s) to (z, x) in N
114: e_1 := new node arisen from move
115: tail move (e_1, x) to (d, t) in N
116: e_2 := new node arisen from move
117: tail move (e_2, t) to (z, s) in N
118: distance = distance +3
119: add d to Y and add u' to Y'
120: let d be mapped to u'
121: **continue**
122: **end if**
123: **end if**
124: **end if**
125: **end if**
126: **end if**
127: **end if**
128: determine all lowest nodes of $N \setminus Y$
129: repeat lines 5 - 127 of the algorithm but interchange N and N', Y and Y',
130: $N \setminus Y$ and $N' \setminus Y'$. For the tail moves in N' also keep track of the from-edges

```
131:        if there exists neither a lowest reticulation of N \ Y nor of N' \ Y' then
132:            u' := arbitrary lowest node of N' \ Y'
133:            x' := arbitrary child of u'
134:            y' := other child of u'
135:            x := node in Y which is mapped to x' in Y'
136:            y := node in Y which is mapped to y' in Y'
137:            if x and y have a common parent in N \ Y then
138:                u := common parent of x and y in Y
139:                add u to Y and add u' to Y'
140:                let u be mapped to u'
141:                continue
142:            else if x and y do not have a common parent in N \ Y then
143:                z_x := arbitrary parent of x in N \ Y
144:                z_y := arbitrary parent of y in N \ Y
145:                if (z_x, x) is movable then
146:                    tail move (z_x, x) to (z_y, y) in N
147:                    distance = distance +1
148:                    z_x1 := new node arisen from move
149:                    add z_x1 to Y and add u' to Y'
150:                    let z_x1 be mapped to u'
151:                    continue
152:                else if (z_y, y) is movable then
153:                    tail move (z_y, y) to (z_x, x) in N
154:                    distance = distance +1
155:                    z_y1 := new node arisen from move
156:                    add z_y1 to Y and add u' to Y'
157:                    let z_y1 be mapped to u'
158:                    continue
159:                else
160:                    find nodes c_x and d_x which form a triangle with z_x with long edge
161:                    (c_x, d_x)
162:                    find nodes c_y and d_y which form a triangle with z_y with long edge
163:                    (c_y, d_y)
164:                    if c_x is not the child of the root then
165:                        b_x := parent of c_x
166:                        a_x := arbitrary parent of b_x
167:                        tail move (c_x, d_x) to (a_x, b_x) in N
168:                        tail move (z_x, x) to (z_y, y) in N
169:                        distance = distance +2
170:                        z_x1 := new node arisen from move
171:                        add z_x1 to Y and add u' to Y'
172:                        let z_x1 be mapped to u'
173:                        continue
174:                    else if c_y is not the child of the root then
175:                        b_y := parent of c_y
176:                        a_y := arbitrary parent of b_y
```

177: tail move (c_y, d_y) to (a_y, b_y) in N
178: tail move (z_y, y) to (z_x, x) in N
179: distance = distance +2
180: z_{y1} := new node arisen from move
181: add z_{y1} to Y and add u' to Y'
182: let z_{y1} be mapped to u'
183: **continue**
184: **end if**
185: **end if**
186: **end if**
187: **end if**
188: **end while**
189: ρ_N := root of N
190: $\rho_{N'}$:= root of N'
191: add ρ_N to Y and add $\rho_{N'}$ to Y'
192: let ρ_N be mapped to $\rho_{N'}$
193: **end procedure**
194: μ^N := resulting sequence of tail moves in N
195: $\mu^{N'}$:= resulting sequence of tail moves in N'
196: $\epsilon^{N'}$:= resulting sequence of from-edges corresponding to tail moves in N'
197: **if** $\mu^{N'}$ is not empty **then**
198: apply Algorithm 1 to $N, N', \mu^{N'}, \epsilon^{N'}$ and call the resulting sequence τ^N
199: add the terms of τ^N to μ^N
200: **end if**

Theorem 3.2. *Let N and N' be rooted binary phylogenetic networks on the same leaf set X, with the same number of reticulations, such that they are not the networks depicted in Figure 6. Let $Y = L(N)$ and $Y' = L(N')$. Algorithm 2 constructs a sequence of tail moves turning N into N'.*

Proof. First we prove the correctness of the algorithm. We know from Lemma 2.15 (Lemma 4.6 in [4]) that there is a sequence of tail moves turning N into N'. We follow the proof of this Lemma. The existence of all nodes and structures as stated in Algorithm 2 follows from the proof of Lemma 2.15. The proof shows, under the assumption $|N \setminus Y| > 1$, that there are three cases possible, which split into further subcases. As $Y = L(N)$, we know that $|N \setminus Y| > 0$. Since the iteration that starts on line 3 of Algorithm 2 continues as long as $|N \setminus Y| \neq 1$, if follows that in each iteration one of the cases as described in the proof of Lemma 2.15 is applicable.

In each iteration the algorithm determines the lowest nodes of $N' \setminus Y'$ (line 4). If one of the lowest nodes is a reticulation, the algorithm labels is u'. This, line 5, is where we enter case 1 from the proof of Lemma 2.15. Thus we may define x' to be the child of u' and select the node $x \in Y$, such that x is mapped to x' by the isomorphism between $N[Y]$ and $N'[Y']$. Lines 10 - 13 correspond to subcase 1a from the proof of Lemma 2.15. So if z, the parent of x, is a reticulation, we may add z to Y, u' to Y' and extend the isomorphism by letting z be mapped to u'.

If z is not a reticulation (line 14), we enter subcase 1b. Lines 15 - 39 correspond to case 1bi. If (z, x) is movable, the algorithm selects an arbitrary reticulation u in $N \setminus Y$. It proceeds by distinguishing 3 different cases depending on v, the child of u. If $x = v$ (line 18), we are again in case 1a from the proof and so the algorithm repeats lines 11 - 13, but with z replaced by u. If it happens that $z = v$ (line 22), the algorithm considers w, an arbitrary parent of u, and y, the other child of z. From case 1bi it follows that we may move the tail of edge (z, y) to (w, u) in N; add u to Y and u' to Y'; and extend the isomorphism. If neither $x = v$ nor $z = v$ (line 30), it follows from case 1bi that we may perform 2 tail moves in N (lines 31 and 34) and again extend the isomorphism.

Lines 40 - 125 correspond to case 1bii from the proof of Lemma 2.15. If (z, x) is not movable, the algorithm considers the nodes c, d that form a triangle with z. There are two different scenario's: either b, the child of c, is the root (case 1biiA) or it is not the root of N (case 1biiB). Lines 43 - 68 of the algorithm correspond to the former, while lines 69 - 124 correspond to the latter. In the same manner as before, the tail moves performed by the algorithm in each case are as prescribed by the proof of Lemma 2.15. The addition of nodes to Y and Y' and the extension of the isomorphism are justified by the proof as well.

If it so happens that there is no lowest reticulation of $N' \setminus Y'$, the algorithm then determines the lowest nodes of $N \setminus Y$ (line 128). If there exists a lowest reticulation of $N \setminus Y$, we enter case 2 of the proof of Lemma 2.15. Thus we may repeat lines 5 - 127of the algorithm, but interchange N and N', Y and Y', $N \setminus Y$ and $N' \setminus Y'$ (lines 129 - 130).

The event that no lowest node of $N \setminus Y$ nor any lowest node of $N' \setminus Y'$ is a reticulation (line 131), corresponds to case 3 from the proof of Lemma 2.15. Thus we may choose an arbitrary lowest node u' of $N' \setminus Y'$, with x' and y' its children. The algorithm also selects the nodes $x, y \in Y$ such that x is mapped to x' and y to y'. Lines 137 - 141 correspond to case 3a. So if x and y have a common parent u, we may add u to Y, add u' to Y' and extend the isomorphism. Lines 142 - 186 correspond to case 3b, which further splits into 3 subcases: case 3bi (lines 145 - 151), case 3bii (lines 152 - 158) and case 3biii (lines 159 - 185). Again all tail moves and isomorphism extensions follow from the proof of Lemma 2.15.

When the stopping criterion of the loop is met, so when $|N \setminus Y| = 1$, it follows from the proof of Lemma 2.15 that we may add ρ_N, the root of N, to Y and add $\rho_{N'}$, the root of N', to Y'. We may extend the isomorphism by letting ρ_N be mapped to $\rho_{N'}$ (lines 191 - 192). Since the isomorphism between $N[Y]$ and $N'[Y']$ is maintained and properly extended in every possible case of Algorithm 2, the correctness follows.

Lastly, we prove that the algorithm is finite. Each iteration adds precisely one node from $V(N) \setminus Y$ to Y and one node from $V(N') \setminus Y'$ to Y'. Since the stopping criterion is $|N \setminus Y| = 1$, this means the algorithm performs the loop $|N \setminus Y| - 1$ times. Thus the finiteness follows. □

4 Worked examples

In this section, two worked examples are given to demonstrate the operation of Algorithm 2. Example 4.1 uses a network pair with two leaves and two reticulations, and requires only one tail move in N. Example 4.2, where N and N' have three leaves and three reticulations, shows some different cases of the algorithm and additionally requires one tail move in N'. The reversal of this move is illustrated in example 4.2 as well.

4.1 Two leaves and two reticulations

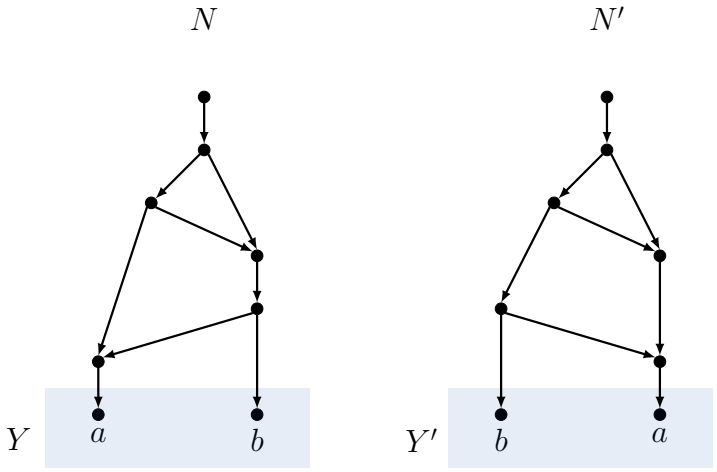

Figure 9: The networks N and N' with two leaves and two reticulations to which Algorithm 2 will be applied. At the start we have $Y = L(N)$ and $Y' = L(N')$ and there is an isomorphism between $N[Y]$ and $N'[Y']$. This will be represented by the blue rectangles.

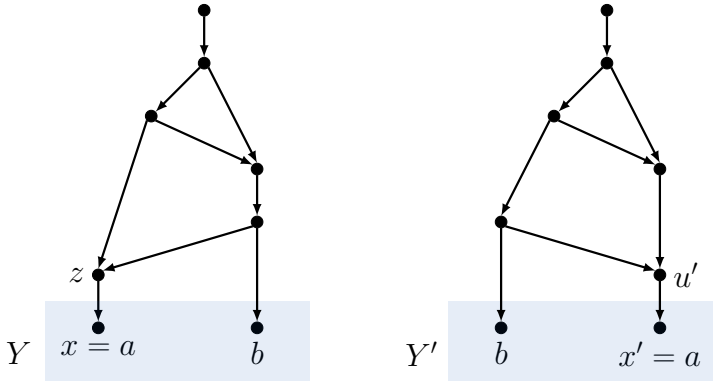

Figure 10: $N' \backslash Y'$ has a lowest reticulation u' with child $x' = a$. x' is mapped to $x \in Y$ with $x = a$. The parent of x is z, which is a reticulation. Thus we are now in line 10 of the algorithm and we may add z to Y and u' to Y'.

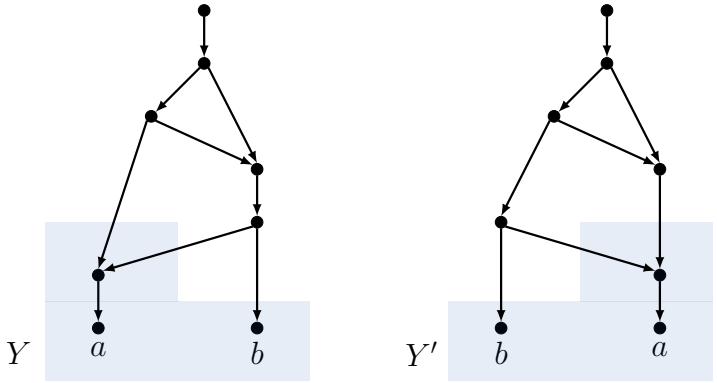

Figure 11: Result of adding z to Y, u' to Y' and extending the isomorphism by letting z be mapped to u'.

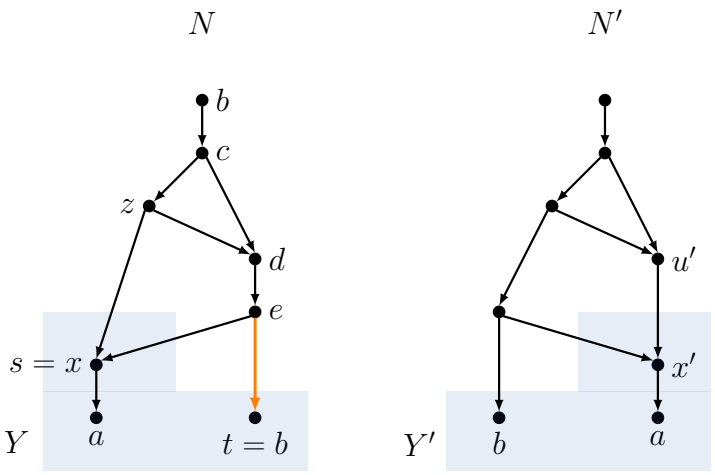

Figure 12: Again $N' \setminus Y'$ has a lowest reticulation u' with child x', so there is a $x \in Y$ with parent z. Here z is not a reticulation and (z, x) is not movable. We can find nodes c, d which form a triangle with z. b is the child of c, which is the root. The child of d is e, which is a tree node with children $s = x$ and $t = b$. We are now in line 106 of the algorithm and we may move (e, t) to (z, x).

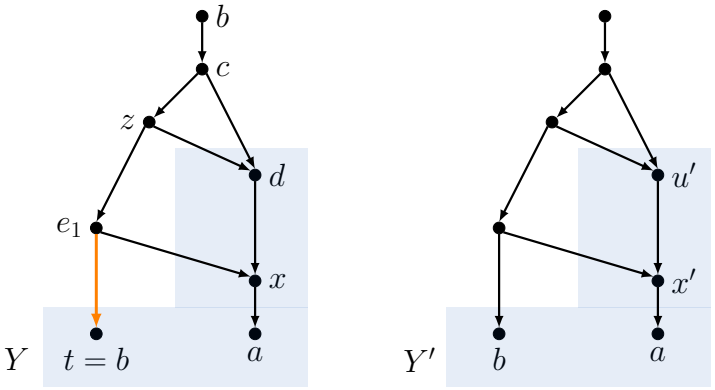

Figure 13: Result of the tail move in N. Now x has a reticulation parent and we may add d to Y and add u' to Y'. We let d be mapped to u'. The distance is now equal to 1.

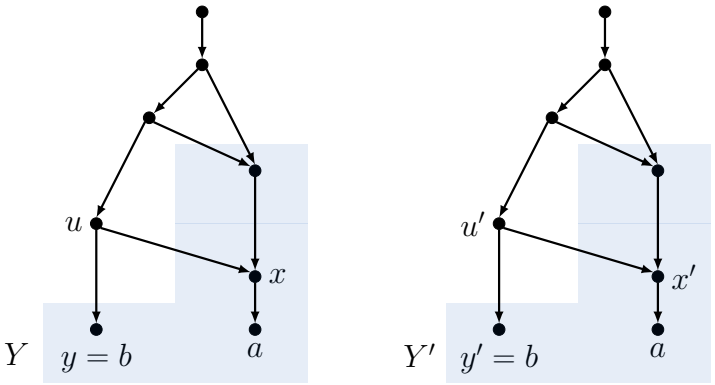

Figure 14: Now neither $N' \setminus Y'$ nor $N \setminus Y$ has a lowest reticulation. We take the lowest tree node u' in $N' \setminus Y'$ with children x', y'. There are nodes $x, y \in Y$ such that x, y is mapped to x', y' respectively. Since x and y have a common parent u not in Y, we are now in line 137 of the algorithm and we may add u to Y and add u' to Y'. We let u be mapped to u'.

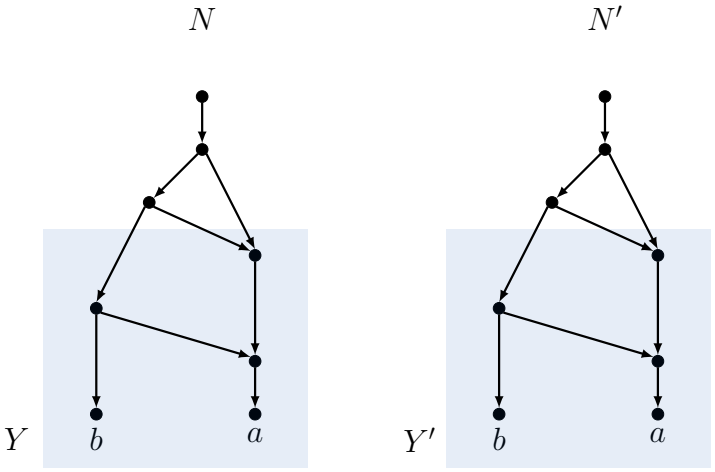

Figure 15: Result of adding u to Y, u' to Y' and extending the isomorphism.

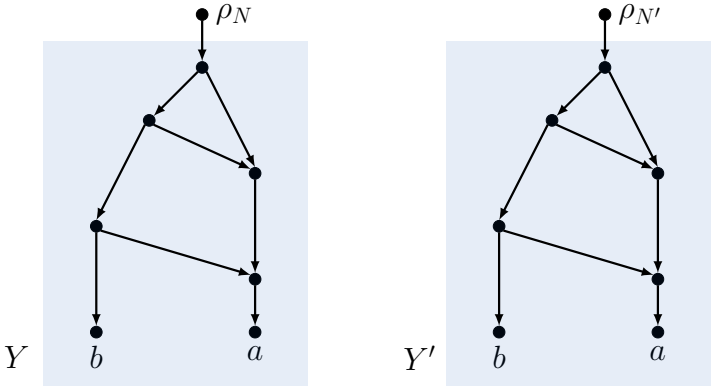

Figure 16: Result of the next two iterations, which repeat lines 131 - 141 of the algorithm. Now $N \setminus Y$ contains only one node, the root ρ_N. Similarly, $N' \setminus Y'$ contains only $\rho_{N'}$. We may add ρ_N to Y and $\rho_{N'}$ to Y' and extend the isomorphism.

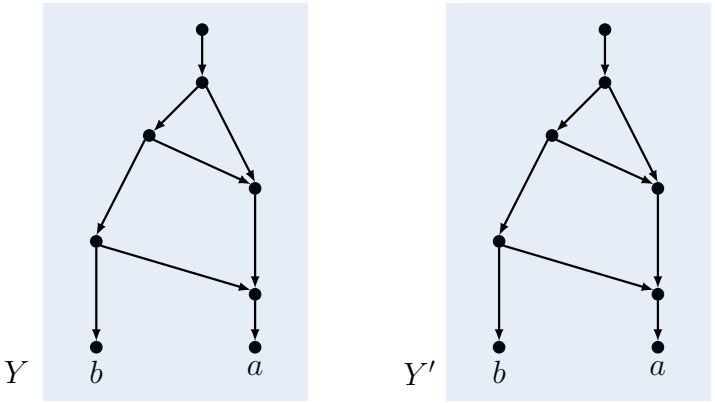

Figure 17: Now $Y = N$ and $Y' = N'$, so N and N' are isomorphic. The algorithm has used only one tail move, so the final distance is 1. Since there were no moves in N', the sequence that turns N into N' is given by the one tail move in N.

4.2 Three leaves and three reticulations

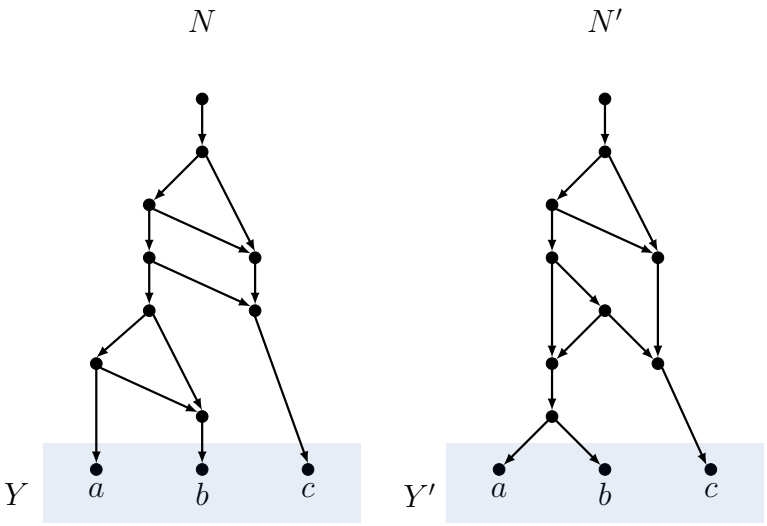

Figure 18: The networks N and N' with three leaves and three reticulations to which Algorithm 2 will be applied. At the start we have $Y = L(N)$ and $Y' = L(N')$ and there is an isomorphism between $N[Y]$ and $N'[Y']$. This will again be represented by the blue rectangles.

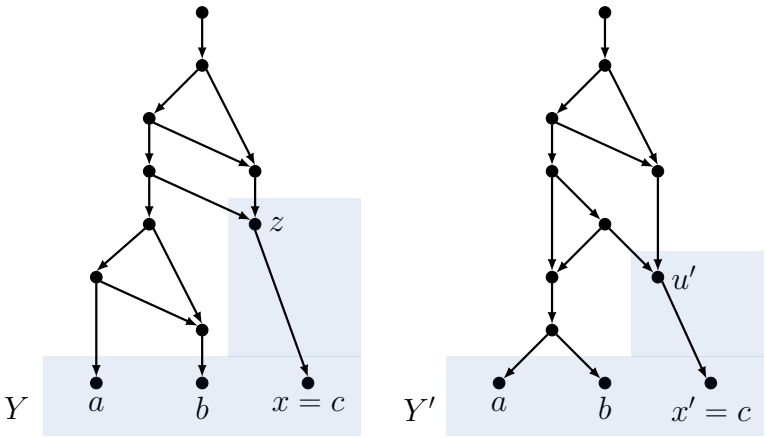

Figure 19: $N' \setminus Y'$ has a lowest reticulation u' with child $x' = c$. x' is mapped to $x \in Y$ with $x = c$. The parent of x is z, which is a reticulation. Thus we are now in line 10 of the algorithm and we may add z to Y and u' to Y'.

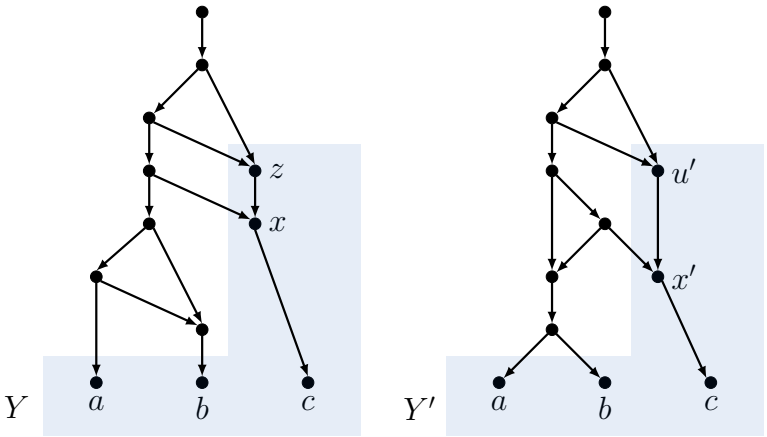

Figure 20: $N' \setminus Y'$ has again a lowest reticulation u' with child x'. x' is mapped to $x \in Y$. The parent of x is z, which is a reticulation. Thus we are again in line 10 of the algorithm and we may add z to Y and u' to Y'.

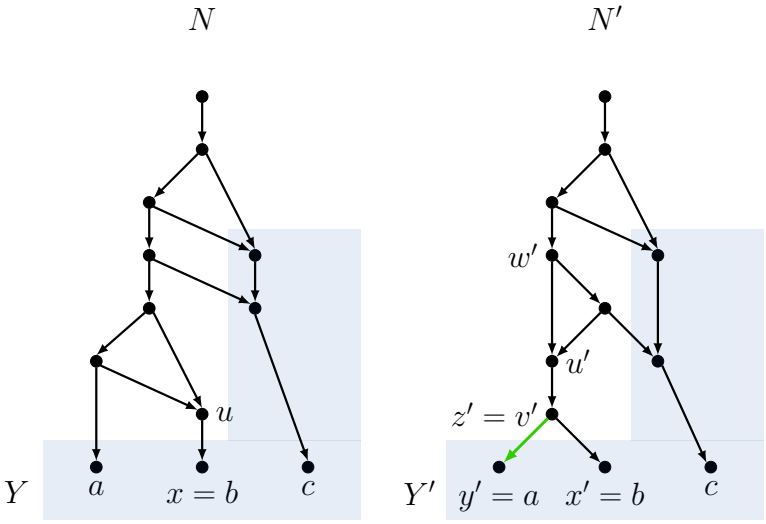

Figure 21: Now $N' \setminus Y'$ has no lowest reticulation, but $N \setminus Y$ has a lowest reticulation u with child x. There is a $x' \in Y'$ with parent z'. Here z' is not a reticulation and (z', x') is movable. We consider the reticulation u' in N' with child $v' = z'$, as well as y', the other child of z. We may move (z', y') to (w', u') in N'.

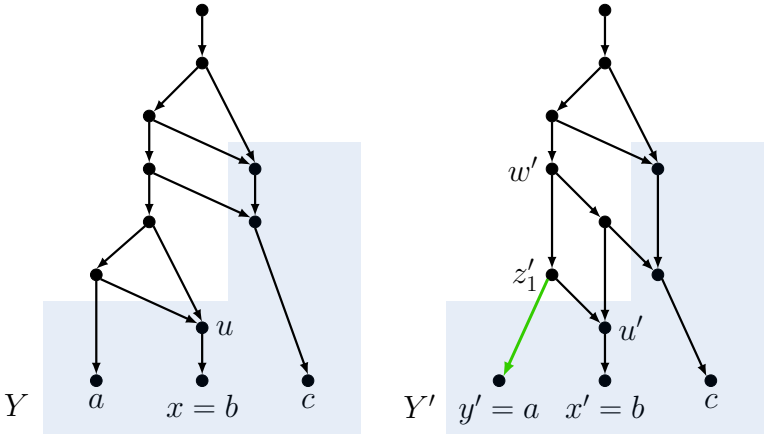

Figure 22: Result of the tail move in N'. Note that the returning edge for this move is (u', b). Now x' has a reticulation parent and we may add u' to Y' and add u to Y. We let u' be mapped to u. The distance is now equal to 1.

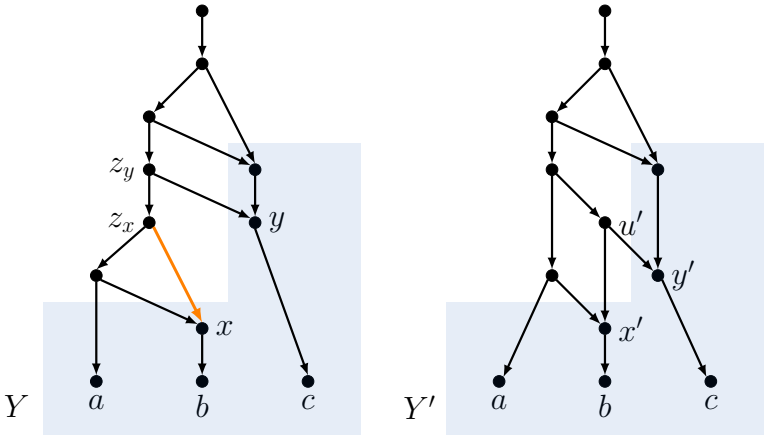

Figure 23: Now neither $N' \setminus Y'$ nor $N \setminus Y$ has a lowest reticulation. We take the lowest tree node u' in $N' \setminus Y'$ with children x', y'. There are nodes $x, y \in Y$ such that x, y is mapped to x', y' respectively. x and y do not have a common parent not in Y and we consider z_x, z_y, the respective parents of x, y. Since (z_x, x) is movable, we are now in line 145 of the algorithm and we may move (z_x, x) to (z_y, y).

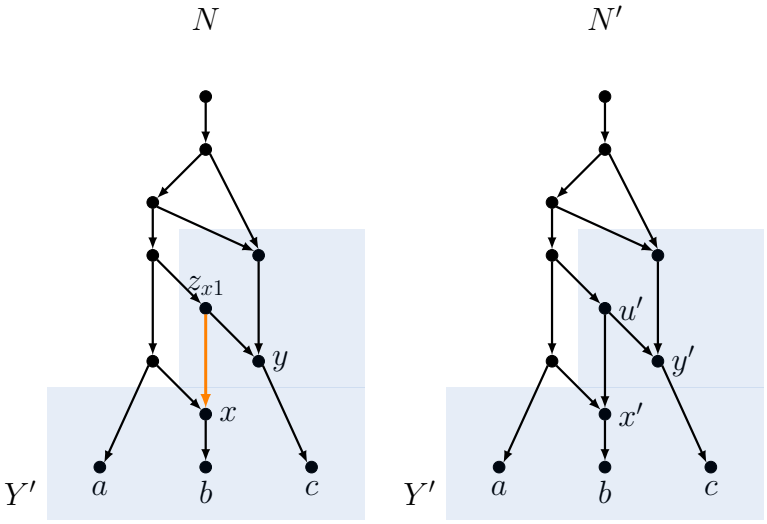

Figure 24: Result of the tail move in N. Now x and y have a common parent z_{x1} in $N \backslash Y$. We may add z_{x1} to Y and add u' to Y'. The distance is now equal to 2.

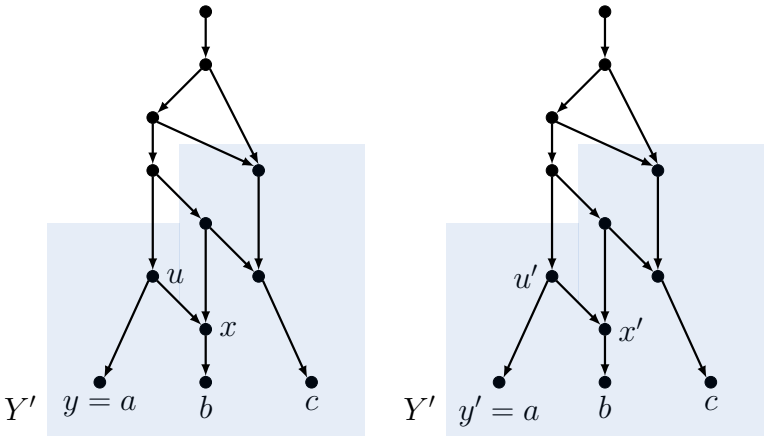

Figure 25: Again neither $N' \setminus Y'$ nor $N \setminus Y$ has a lowest reticulation. We take the lowest tree node u' in $N' \setminus Y'$ with children x', y'. There are nodes $x, y \in Y$ such that x, y is mapped to x', y' respectively. Since x and y have a common parent u not in Y, we are now in line 137 of the algorithm and we may add u to Y and add u' to Y'.

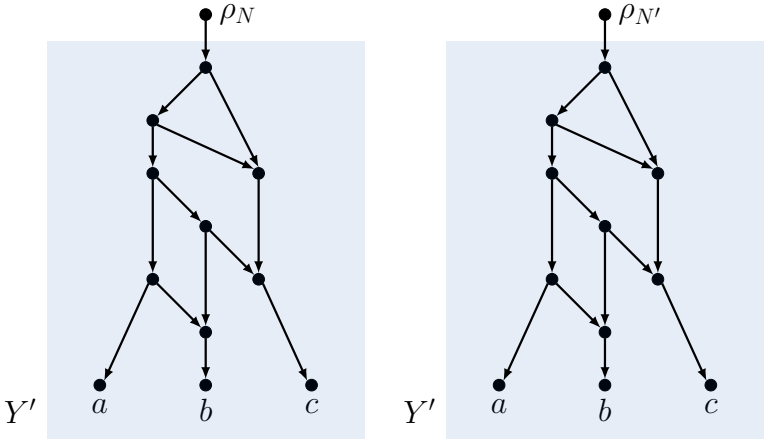

Figure 26: Result of the next three iterations, which repeat lines 131 - 141 of the algorithm. Now $N \setminus Y$ contains only one node, the root ρ_N. Similarly, $N' \backslash Y'$ contains only $\rho_{N'}$. We may add ρ_N to Y and $\rho_{N'}$ to Y'.

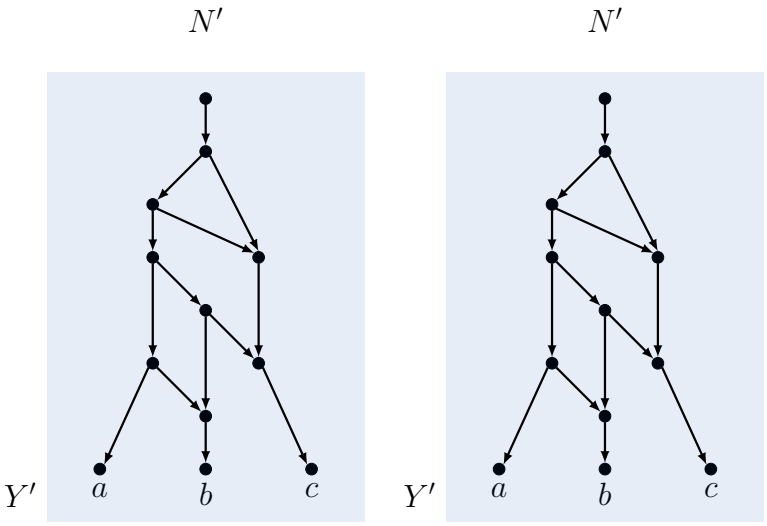

Figure 27: Now $Y = N$ and $Y' = N'$, so N and N' are isomorphic. The algorithm has used two tail move, so the final distance is 2.

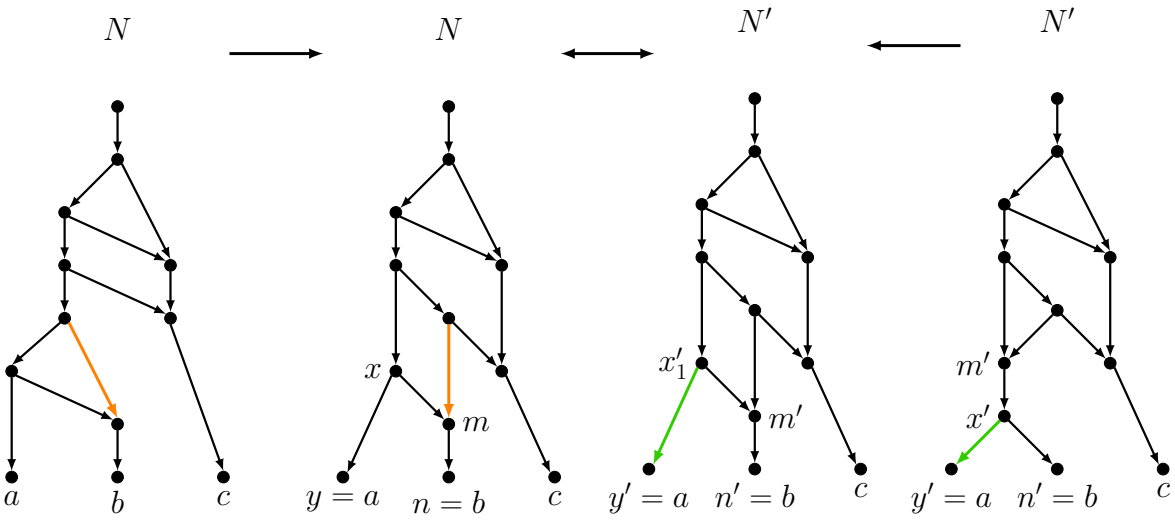

Figure 28: Summary of the rearrangements. From the left we see the tail move in N, from the right we see the tail move in N'. This results in the two middle figures, where N and N' are isomorphic. Since there is a tail move in N', we use Algorithm 1 to reverse it. Tot this end, we relabel the relevant nodes in N'. So the move in N' is moving the tail of (x', y'), which results into a new edge x'_1, with from-edge (m', n'). We can find nodes $x, y, m, n \in V(N)$ such that they are mapped to $x'_1, y', m', n' \in V(N')$, respectively. The reversal of the tail move in N' is now equivalent to moving (x, y) to (m, n) in N. The sequence of tail moves that turns N into N' now consists of two tail moves in N.

5 Algorithm improvements

In this section, four proposals are made for improvement of Algorithm 2, which will be called Algorithms 2.1 through 2.4. For each improvement a motivation is given, supported by illustrations demonstrating the benefits for different network structures. A precise formulation of the changes in the algorithm is given as well.

5.1 Prioritise choosing reticulation parent

The first improvement has to do with an arbitrary node selection Algorithm 2 does very early on, in line 9. It first selects an arbitrary parent z of x, and only then looks to see whether z is a reticulations. However, in the situation that x has more than one parent, this can give rise to unnecessary tail moves. See Figures 29 and 30. Therefor, the first proposal for improvement will be to prioritise choosing a reticulation parent of x.

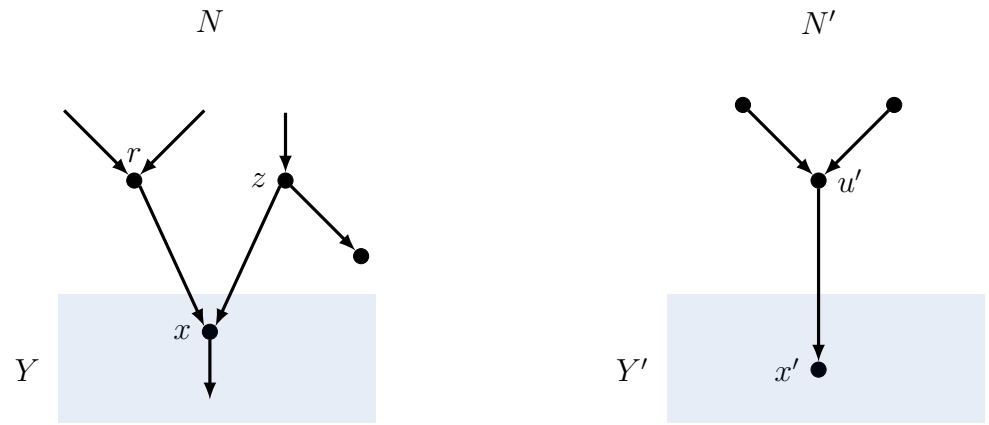

Figure 29: Network fragments of N and N' whose structure can cause a problem. In line 9 the algorithm chooses an arbitrary parent z of x not in Y. Ideally, the choice would be r, the reticulation parent of x, since then we may simply add this node and extend the isomorphism. However, if it happens that the algorithm chooses z as in this figure, and if (z, x) is movable, there is a chance that in line 16 a reticulation in $N \setminus Y$ is selected other than r (if there are more than 2). This results into tail moves in N, where there should be none.

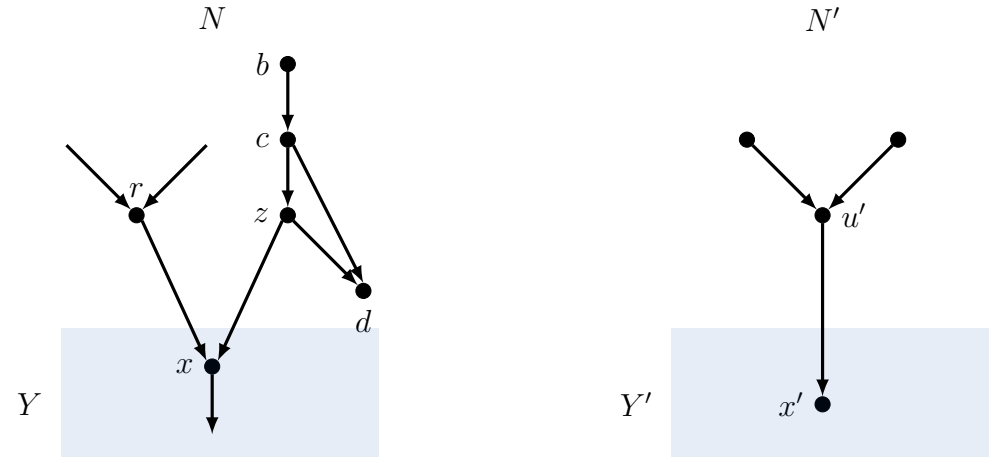

Figure 30: Similar situation to Figure 29 where the algorithm selects the parent z of x, instead of the reticulation parent r. Here (z, x) is not movable, so we are in line 40. If b is not the root, then in line 45 a tail move is applied. If in line 47 a reticulation other than r in $N \setminus Y$ is chosen, this results in even more tail moves. Similarly, if b is the root (line 69) this always results in tail moves, even though there should be none in this case.

To make this improvement precise, lines 9 - 14 of Algorithm 2 should be replaced by the following:

Algorithm 2.1 First improvement

```
 9: if x has a reticulation parent not in Y then
10:     z := arbitrary reticulation parent of x not in Y
11:     add z to Y and add u′ to Y′
12:     let z be mapped to u′
13:     continue
14: else
15:     z := arbitrary parent of x not in Y
16: end if
```

Furthermore, lines 18 - 21 and 49 - 52 of Algorithm 2 may be removed, since they are now redundant.

5.2 Prioritise choosing z such that it has a reticulation parent

There are certain cases in the algorithm that require but one tail move to extend the isomorphism. Whether or not these cases are met, often depends on a previous arbitrary selection of a node. The second improvement is to update Algorithm 2.1 to first try a specific node selection, in order to meet the case that requires the least tail moves. Figures 31 and 32 show in which situations this applies.

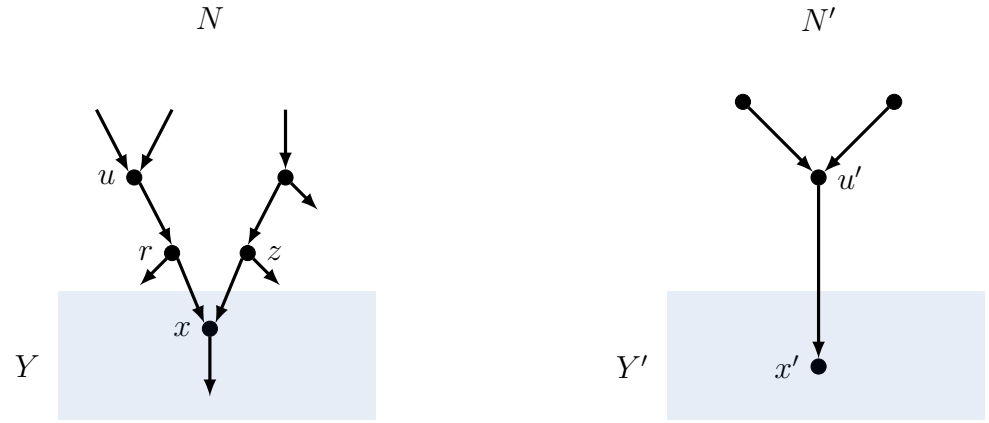

Figure 31: Assume we have the following situation. Since x has no reticulation parent in $N \setminus Y$, Algorithm 2.1 selects z, an arbitrary parent of x. If it happens to be z as in this figure, we know that the algorithm will never meet line 22, since z has no reticulation parent that can be marked u in line 16. Thus it is certain that more than one tail move is needed. However, x has another parent r that does have a reticulation parent. We can force the algorithm to choose r over z and select u its reticulation parent. This results in only one tail move.

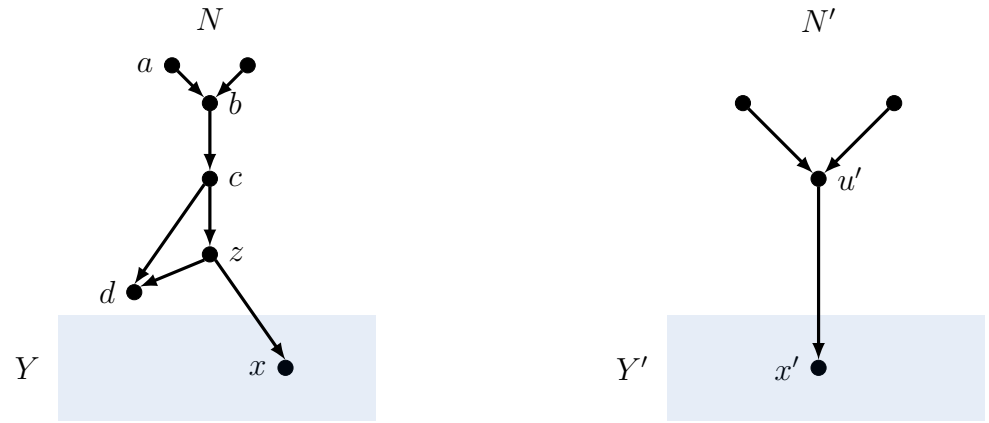

Figure 32: Assume we have the following situation. Since (z, x) is not movable and b is not the root, the algorithm moves (c, d) to (a, b) in N (line 45). The result of this tail move is that b becomes the parent of z. In line 47 the algorithm then chooses an arbitrary reticulation u in $N \setminus Y$. We can update the algorithm to prioritise choosing the reticulation parent $u = b$, since then we have $z = v$ and only one tail move is needed.

To make the second improvement precise: lines 9 - 68 of Algorithm 2 should be replaced by the following (this also contains the first improvement):

Algorithm 2.2 Second improvement

```
    if x has a reticulation parent not in Y then
        z := arbitrary reticulation parent of x not in Y
        add z to Y and add u' to Y'
        let z be mapped to u'
        continue
else
    if x has a parent not in Y which has a reticulation parent then
        z := arbitrary parent of x not in Y which has a reticulation parent
        u := parent of z
        y := other child of z
        tail move (z, y) to (w, u) in N
        distance = distance +1
        add u to Y and add u' to Y'
        let u be mapped to u'
        continue
    else
        z := arbitrary parent of x not in Y
        if (z, x) is movable then
            u := arbitrary reticulation in N \ Y
            v := child of u
            tail move (z, x) to (u, v) in N
            z_1 := new node arisen from move
            w := arbitrary parent of u
            tail move (z_1, v) to (w, u) in N
            distance = distance +2
            add u to Y and add u' to Y'
            let u be mapped to u'
            continue
        else if (z, x) is not movable due to parallel edges then
            find nodes c and d which form a triangle with z with long edge (c, d)
            b := parent of c
            if b is not the root of N then
                a := arbitrary parent of b
                tail move (c, d) to (a, b) in N
                distance = distance +1
                if b is a reticulation then
                    w := arbitrary parent of b
                    tail move (z, d) to (w, b) in N
                    distance = distance +1
                    add b to Y and add u' to Y'
                    let b be mapped to u'
                    continue
                else
                    u := arbitrary reticulation in N \ Y
                    v := child of u
```

```
                    tail move (z, x) to (u, v) in N
                    z_1 := new node arisen from move
                    tail move (z_1, v) to (w, u) in N
                    distance = distance +2
                    add u to Y and add u' to Y'
                    let u be mapped to u'
                    continue
                end if
            end if
        end if
    end if
end if
```

5.3 Prioritise choosing u' such that it results in no tail moves

The third proposal for an improvement of the algorithm is based on refinement of yet another arbitrary node selection. This time the point of interest is the choice of u' in $N' \setminus Y'$. Instead of choosing an arbitrary lowest reticulation or, if there are none, an arbitrary lowest tree node, the algorithm can be forced to prioritise choosing u' such that this leads to no immediate tail moves in the other network. See Figures 33 and 34.

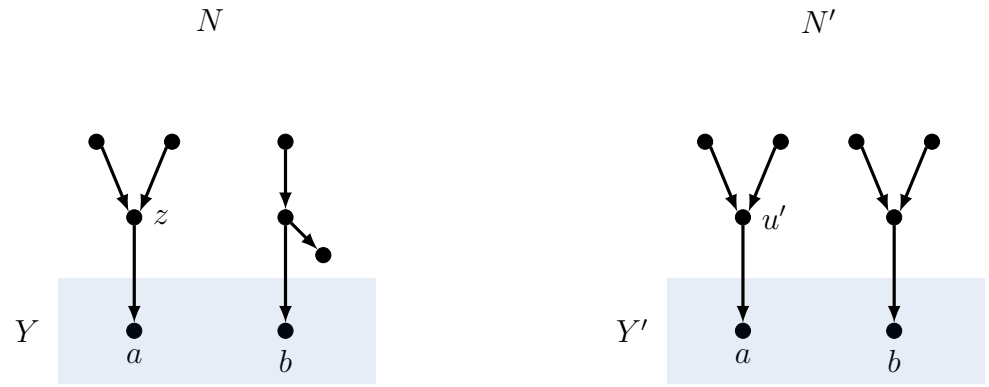

Figure 33: Consider the following situation. Algorithm 2.2 first selects an arbitrary lowest reticulation in $N' \setminus Y'$. There are two possibilities. Choosing u' as in this figure results in no immediate tail move, and z can be added to Y and u' to Y'. This is preferable, since it 'locks' their position. If the choice fell on the other lowest reticulation in $N' \setminus Y'$, the resulting tail moves could alter the structure of N in such a way that in the next iteration the reticulation parent of $a \in Y$ cannot simply be added.

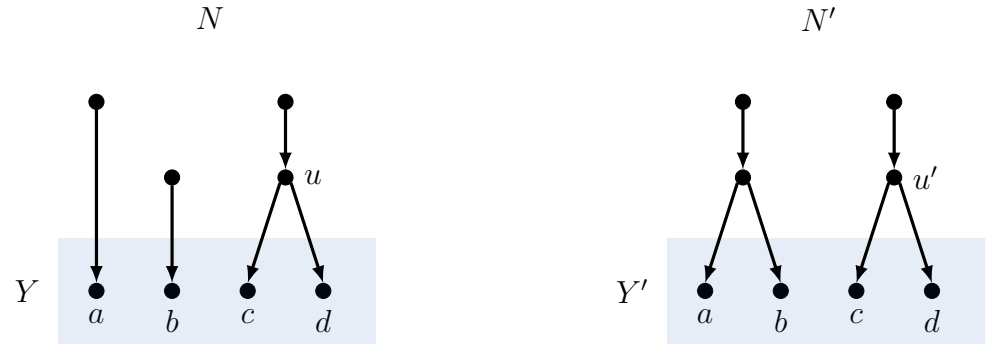

Figure 34: A similar situation as in Figure 33, but with no lowest reticulations. Following the same reasoning, it is preferable to let the algorithm first select a lowest tree node u' in $N' \setminus Y'$ such that in this case $c, d \in Y$ have a common parent u.

To make the third improvement precise: lines 6 - 8 and lines 132 - 136 of Algorithm 2.2 (which are the same lines in Algorithm 2) should be replaced by the following:

Algorithm 2.3 Third improvement

```
 6: for all lowest reticulations of N' \ Y' do
 7:     u' := current lowest reticulation of N' \ Y'
 8:     x' := child of u'
 9:     x := node in Y which is mapped to x' in Y'
10:     if x has a reticulation parent not in Y then
11:         break
12:     end if
13: end for
```

```
132: for all lowest nodes of N' \ Y' do
133:     u' := current lowest node of N' \ Y'
134:     x' := arbitrary child of u'
135:     y' := other child of u'
136:     x := node in Y which is mapped to x' in Y'
137:     y := node in Y which is mapped to y' in Y'
138:     if x and y have a common parent in N \ Y then
139:         break
140:     end if
141: end for
```

5.4 Select u as the closest reticulation to z

The last improvement proposal is an extension of the second improvement. The aim is to completely eliminate the arbitrariness of selecting a reticulation u in $N \setminus Y$, but instead let the algorithm choose u such that it is 'closest' to z. That is: choose reticulation u such that the length of the shortest path from u to z in the underlying undirected graph is

smallest, see Figure 35. The rationale is that if the algorithm is forced to make structural changes as locally as possible, this will improve performance (require less tail moves), since the global structure is minimally affected. Especially networks N and N' that already have a large resemblance, in particular 'higher up' in the network, should benefit from this.

To make this improvement precise, lines 27 and 52 of Algorithm 2.2 should be replaced by the following:

Algorithm 2.4 Fourth improvement

27: u := reticulation in $N \setminus Y$ such that the length of the shortest path to z in the undirected graph underlying N is smallest

52: u := reticulation in $N \setminus Y$ such that the length of the shortest path to z in the undirected graph underlying N is smallest

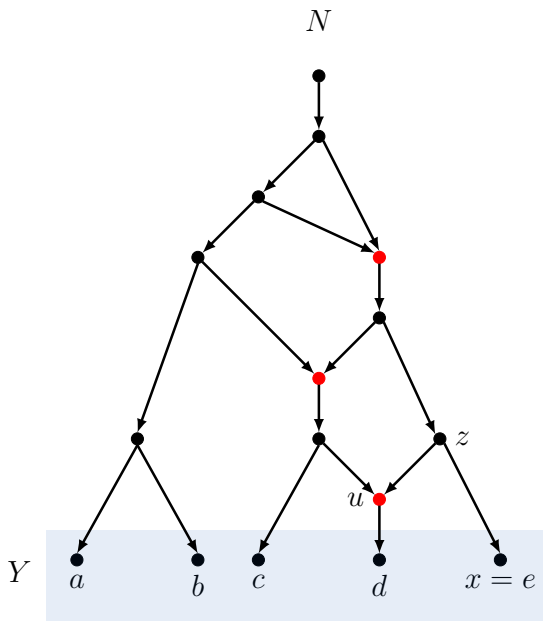

Figure 35: The fourth improvement selects in line 27 the reticulation u in $N \setminus Y$ as in this figure, since the shortest path from it to z in the underlying undirected graph has smallest length (1), compared to all other reticulations in $N \setminus Y$ (in red). The resulting tail moves in N will change the network locally (close to z) and will not affect the structure 'higher up'.

6 Testing and results

6.1 Methodology

Algorithm 2 and its improvement proposals were implemented using MATLAB. In order to test their performance, a comparison is made between the algorithm distance as determined by the different versions of Algorithm 2, and the true tail distance $d_{\text{tail}}(N, N')$ for small networks N and N'. To this end, data was provided by Remie Janssen, who used a breadth-first search algorithm to determine the tail distance for a range of 385 network pairs. A random phylogenetic network generator [5] was used to generate ten networks (numbered 0 through 9) of each desired network type, i.e. networks with the same number of reticulations and leaves. Here networks were used with 3 leaves and 1 through 3 reticulations, 4 leaves and 1 through 3 reticulation, 5 leaves and 1 reticulation. For each network type, the breadth-first search algorithm determined $d_{\text{tail}}(N, N')$, where for N and N' all combinations of the ten networks were used. 84 out of the 385 network pairs N and N' lack the exact value of $d_{\text{tail}}(N, N')$, due to a too long running time. In these cases a lower bound of the tail distance is given. Algorithms 2 through 2.4 were applied to the same range of 385 network combinations, and the resulting distances were recorded.

6.2 Summary results

Table 1 and Figure 36 give a summary of the extended results, which can be found in Appendix A and B. Table 1 summarises the results for the 84 network combinations for which only a lower bound is known, whilst Figure 36 provides an overview of the results for the other 301 networks for which the exact tail distance is known.

Algorithm distance comparison	Improvement			Worsening		
	%	total	average	%	total	average
Algorithm 2.1 vs 2	0	0	0	0	0	0
Algorithm 2.2 vs 2.1	20.2	41	2.4	0	0	0
Algorithm 2.3 vs 2.2	11.9	11	1.1	3.6	3	1
Algorithm 2.4 vs 2.3	20.2	23	1.4	25	39	1.9

Table 1: Summary of Table B.1: results for the 84 network combinations for which a lower bound is known.

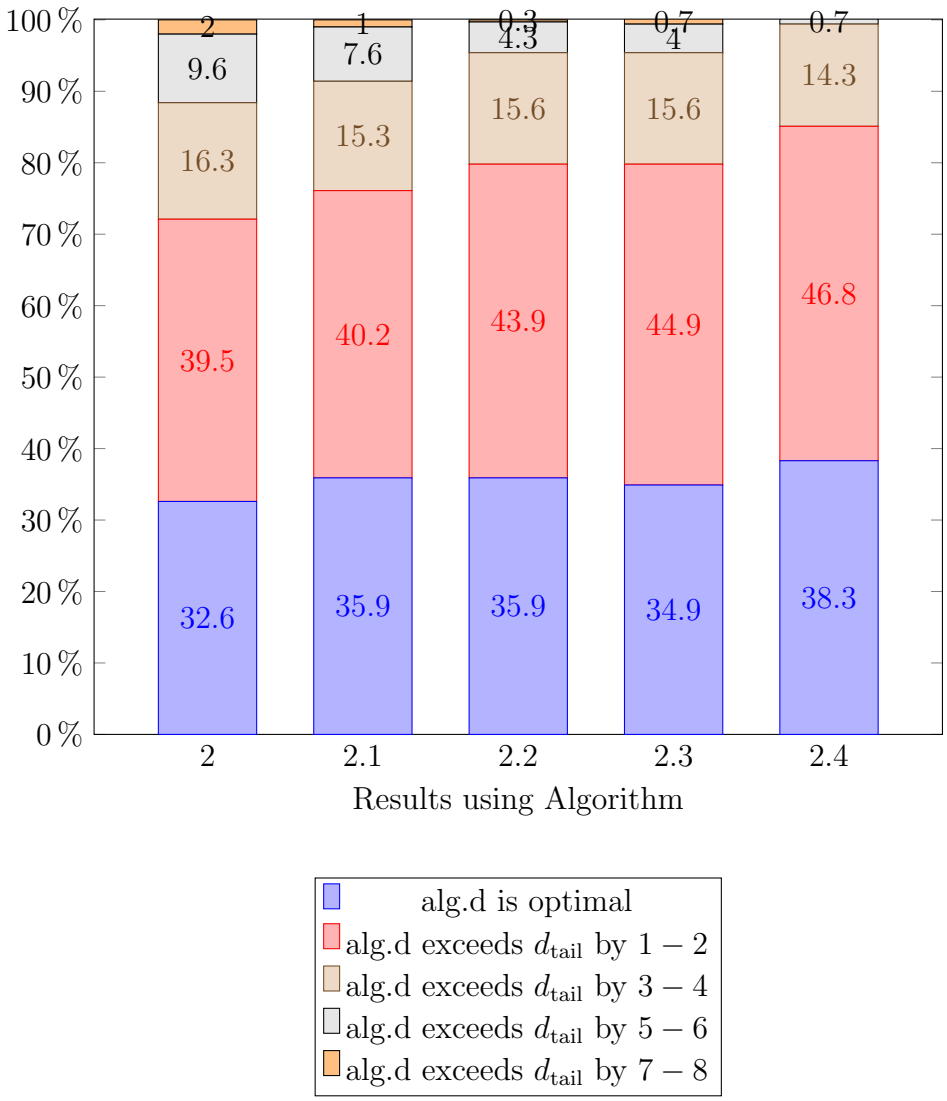

Figure 36: Summary of Table A.1: results for the 301 network combinations for which an exact tail distance is known.

6.3 Interpretation and comparison

Figure 36 and Table 1 enable easy performance comparison and assessment of the different algorithm improvements. When first considering Algorithm 2, from Figure 36 it is clear that it gives the least results of all algorithms. This means that for the tested networks N and N' for which $d_{\text{tail}}(N, N')$ is known, Algorithms 2.1 through 2.4 are indeed an improvement on Algorithm 2. Still, the results of the latter are not unsatisfactory: for almost a third of the 301 tested networks the algorithm calculates a distance that is equal to the tail distance. On top of that, for about 40% of the network pairs it calculates a distance that is only one or two tail moves larger than the tail distance. Larger deviations, with a maximum of 8, are increasingly less frequent. It is harder to evaluate the performance of Algorithm 2 when considering only Table B.1, since no comparison can be made with the exact tail distance. However, larger algorithm distances (with a maximum of 11) seem to

be more frequent compared to Table A.1. Since the network pairs from Table B.1 are not bigger, and are even in the same range (in terms of number of leaves and reticulations) as the ones from Table A.1, it is not likely that the exact tail distances very much exceed the estimated lower bounds. Thus it seems that Algorithm 2 performs worse when applied to the network combinations from Table B.1.

When looking in Table 1 at the comparison of Algorithm 2.1 against Algorithm 2, it can be seen that the former determines the exact same results for all tested networks with a lower bound, as the latter. This does not agree with the results from Figure 36, where a definite improvement can be seen of Algorithm 2.1 over Algorithm 2 for the other 301 network pairs.

In Figure 36, the number of tested networks for which Algorithm 2.2 determines an optimal distance, remains the same compared to Algorithm 2.1. However, the former determines in about 4% more cases a distance that exceeds d_{tail} by only $1 - 2$, while in 4% less cases it calculates larger deviations, in the range $5 - 8$. This shift is a desirable improvement. From Table 1, when Algorithm 2.2 is compared to Algorithm 2.1, it follows that for 20.2% of the tested network combinations, Algorithm 2.2 calculates a smaller distance, with an average distance improvement of 2.4. On the other hand, a larger algorithm distance compared to Algorithm 2.1 does not occur. Thus Table 1 and Figure 36 do agree: the quality of the algorithm distance has improved with Algorithm 2.2.

In the case of Algorithm 2.3, there is again a discrepancy between Figure 36 and Table 1. The former shows a worsening compared to Algorithm 2.2, since the number of networks from Table A.1 for which it calculates an optimal distance has decreased with 1%. However, in Table 1 a slight improvement over Algorithm 2.2 can be seen. For 11.9% of the network pairs from Table B.1, Algorithm 2.3 results in a distance improvement, compared to 3.6% worsening. The improvement, however, is only a total distance improvement of 11. When also taking into account the total distance worsening, which is 3, the net improvement of Algorithm 2.3 for the networks from Table B.1 is rather insignificant.

From Figure 36, it is clear that Algorithm 2.4 gives the best results of all algorithms: for 85% of the tested networks from Table A.1 it determines a distance that is either optimal or exceeds only by one or two tail moves. On top of that, the really large deviations in the range $7 - 8$ have now zero occurrence. Table 1 shows something very different for the networks from Table B.1. Here it is clear that Algorithm 2.4 is not at all an improvement on Algorithm 2.3. However, how this compares to the exact tail distance is unknown. It may be that even though Algorithm 2.4 performs less than Algorithm 2.3, the quality of the algorithm distance is still sufficient.

7 Conclusion and open questions

In this paper, the constructive proof of Lemma 4.6 from [4] was formulated and implemented as an algorithm: Algorithm 2. Given two rooted binary phylogenetic networks on the same leaf set and with the same number of reticulations, this tail move rearrangement algorithm calculates a sequence of tail moves to turn one network into the other, and thus determines an upper bound on the tail distance between the networks. In order to try and bring this algorithm distance as close to the tail distance as possible, four improvements (Algorithms 2.1 through 2.4) were proposed and implemented. To test the different algorithm performances, a comparison was made with the true tail distance for a range of 385 combinations of small networks. This data was made available and was acquired using a breadth-first search technique. For 84 out of the 385 tested network pairs, the exact tail distance was not determined, only a lower bound was known.

It was shown that Algorithm 2 performs least of all algorithms, when applied to the 301 tested network pairs for which an exact tail distance was known. Nevertheless, it still gives adequate results: for 72% of the network pairs it calculates a distance that is either optimal, or exceeds the tail distance by only $1 - 2$ tail moves. Of the proposed improvements, Algorithm 2.3 turned out to be the worst. The best improvement showed to be Algorithm 2.4. However, this ranking proved not to be a rule in general, as it was not in agreement with the results obtained by testing the algorithms on the networks for which only a lower bound was known. Although no comparison with the tail distance was possible, the algorithms seemed to perform worse in these cases. Algorithm 2.4 was even a substantial worsening compared to Algorithm 2.3.

Based on the disparities in the results, it may be concluded that, given two networks N and N', it generally cannot be predicted which version of Algorithm 2 will perform best. Especially for more complicated networks, for which the tail distance is large. In the case of small and relatively simple networks, Algorithm 2.4 provides the best results.

It did not become evident what causes the disparities in the results of this paper. It would be an interesting follow-up research to try and answer this question. A possible approach could be to extensively analyse the structure of the tested networks and try to recognize patterns that indicate a certain performance of the different algorithms. Ideally, this could prompt better algorithm improvements. Another large open question is how the different algorithms perform for much larger and more complicated networks (more leaves, more reticulations). From a biological point of view, an answer to this question is even more befitting, since the evolutionary history of a species is much more complex than the small networks tested in this paper.

8 References

[1] Daniel H. Huson, Regula Rupp, and Celine Scornavacca. *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, 2010.

[2] Daniel H. Huson and Celine Scornavacca. A survey of combinatorial methods for phylogenetic networks. *Genome Biology and Evolution*, 3:23–35, 2011.

[3] Philippe Gambette, Leo van Iersel, Mark Jones, Manuel Lafond, Fabio Pardi, and Celine Scornavacca. Rearrangement moves on rooted phylogenetic networks. *PLoS Computational Biology*, 13(8):1–21, 2017.

[4] Remie Janssen, Mark Jones, Péter L. Erdős, Leo van Iersel, and Celine Scornavacca. Exploring the tiers of rooted phylogenetic network spaces using tail moves. *Bulletin of mathematical biology*, 80(8):2177–2208, 2018.

[5] Louxin Zhang. Phylnet tools. `phylnet.univ-mlv.fr/tools/randomNtkGenerator.php`.

A Results for network pairs with exact tail distance

Table A.1: For 301 network combinations, the algorithm distance as determined by Algorithms 2 through 2.4, in comparison to the exact tail distance.

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
3	1	0	0	0	0	0	0	0	0
3	1	0	1	3	4	4	4	4	4
3	1	0	2	2	3	3	3	3	3
3	1	0	3	3	4	4	4	4	4
3	1	0	4	2	3	3	3	3	3
3	1	0	5	3	3	3	3	3	3
3	1	0	6	1	2	2	2	2	2
3	1	0	7	1	2	2	2	2	2
3	1	0	8	3	4	4	4	4	4
3	1	0	9	3	5	5	5	5	5
3	1	1	1	0	0	0	0	0	0
3	1	1	2	2	3	3	3	3	3
3	1	1	3	0	0	0	0	0	0
3	1	1	4	2	3	3	3	3	3
3	1	1	5	1	1	1	1	1	1
3	1	1	6	2	3	3	3	3	3
3	1	1	7	3	3	3	3	3	3
3	1	1	8	1	3	3	3	3	3
3	1	1	9	1	2	2	2	2	2
3	1	2	2	0	0	0	0	0	0
3	1	2	3	2	3	3	3	3	3
3	1	2	4	0	0	0	0	0	0
3	1	2	5	2	3	3	3	3	3
3	1	2	6	2	3	3	3	3	3
3	1	2	7	1	3	3	3	3	3
3	1	2	8	1	3	3	3	3	3
3	1	2	9	3	3	3	3	3	3
3	1	3	3	0	0	0	0	0	0
3	1	3	4	2	3	3	3	3	3
3	1	3	5	1	1	1	1	1	1
3	1	3	6	2	3	3	3	3	3
3	1	3	7	3	3	3	3	3	3
3	1	3	8	1	3	3	3	3	3
3	1	3	9	1	2	2	2	2	2
3	1	4	4	0	0	0	0	0	0
3	1	4	5	2	3	3	3	3	3
3	1	4	6	2	3	3	3	3	3

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
3	1	4	7	1	3	3	3	3	3
3	1	4	8	1	3	3	3	3	3
3	1	4	9	3	3	3	3	3	3
3	1	5	5	0	0	0	0	0	0
3	1	5	6	2	3	3	3	3	3
3	1	5	7	3	4	4	4	4	4
3	1	5	8	1	2	2	2	2	2
3	1	5	9	1	2	2	2	2	2
3	1	6	6	0	0	0	0	0	0
3	1	6	7	1	2	2	2	2	2
3	1	6	8	3	4	4	4	4	4
3	1	6	9	2	3	3	3	3	3
3	1	7	7	0	0	0	0	0	0
3	1	7	8	2	4	4	4	4	4
3	1	7	9	3	5	5	5	5	5
3	1	8	8	0	0	0	0	0	0
3	1	8	9	2	2	2	2	2	2
3	1	9	9	0	0	0	0	0	0
3	2	0	0	0	0	0	0	0	0
3	2	0	1	3	4	4	4	4	4
3	2	0	2	1	1	1	1	1	1
3	2	0	3	2	2	2	2	2	2
3	2	0	4	4	6	6	6	6	6
3	2	0	5	3	4	4	4	4	6
3	2	0	6	2	6	6	6	2	2
3	2	0	7	4	6	6	6	6	6
3	2	0	8	4	6	6	6	6	5
3	2	0	9	3	4	4	5	5	5
3	2	1	1	0	0	0	0	0	0
3	2	1	2	3	3	3	3	3	3
3	2	1	3	2	4	4	4	4	4
3	2	1	4	3	6	6	6	6	3
3	2	1	5	2	7	7	7	7	3
3	2	1	6	2	4	4	4	4	6
3	2	1	7	3	6	6	6	6	3
3	2	1	8	3	6	6	6	6	3
3	2	1	9	2	6	6	2	2	2
3	2	2	2	0	0	0	0	0	0
3	2	2	3	2	3	3	3	3	3
3	2	2	4	4	6	6	6	6	6
3	2	2	5	3	5	5	5	5	6
3	2	2	6	2	6	6	6	2	6
3	2	2	7	4	6	6	6	6	6

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
3	2	2	8	4	6	6	6	6	6
3	2	2	9	3	5	5	5	5	5
3	2	3	3	0	0	0	0	0	0
3	2	3	4	3	6	6	6	6	4
3	2	3	5	2	4	4	4	4	6
3	2	3	6	2	4	4	4	4	5
3	2	3	7	3	6	6	6	6	4
3	2	3	8	3	6	6	6	6	4
3	2	3	9	2	4	4	5	5	5
3	2	4	4	0	0	0	0	0	0
3	2	4	5	2	6	6	6	6	2
3	2	4	6	3	3	3	3	3	3
3	2	4	7	1	1	1	1	1	1
3	2	4	8	1	3	3	3	3	3
3	2	4	9	2	6	6	6	6	2
3	2	5	5	0	0	0	0	0	0
3	2	5	6	3	5	5	5	5	5
3	2	5	7	2	5	5	5	5	3
3	2	5	8	2	7	7	7	7	5
3	2	5	9	1	1	1	1	1	1
3	2	6	6	0	0	0	0	0	0
3	2	6	7	3	5	5	5	5	5
3	2	6	8	2	5	5	5	5	5
3	2	6	9	3	4	4	4	4	7
3	2	7	7	0	0	0	0	0	0
3	2	7	8	1	2	2	2	2	2
3	2	7	9	2	7	7	7	7	3
3	2	8	8	0	0	0	0	0	0
3	2	8	9	2	7	7	7	7	3
3	2	9	9	0	0	0	0	0	0
3	3	0	0	0	7	0	0	0	0
3	3	0	1	3	5	5	5	5	6
3	3	0	2	2	6	2	2	2	2
3	3	0	5	2	7	4	4	4	4
3	3	0	6	3	10	10	10	10	4
3	3	0	9	3	9	9	9	10	3
3	3	1	1	0	0	0	0	0	0
3	3	1	3	3	8	8	8	8	5
3	3	1	5	2	2	2	2	2	2
3	3	1	7	3	8	8	9	9	8
3	3	1	9	3	9	9	7	7	4
3	3	2	2	0	7	0	0	0	0
3	3	2	5	3	7	4	4	4	4

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
3	3	2	6	3	7	7	7	7	5
3	3	2	7	3	9	9	5	5	5
3	3	2	8	3	8	8	8	8	8
3	3	2	9	2	7	7	7	7	4
3	3	3	3	0	0	0	0	0	0
3	3	3	5	3	8	8	8	8	6
3	3	3	7	1	6	6	5	5	5
3	3	3	8	3	7	7	7	7	4
3	3	3	9	3	6	6	6	6	6
3	3	4	4	0	0	0	0	3	0
3	3	4	8	2	4	4	4	4	4
3	3	4	9	3	8	8	7	7	7
3	3	5	5	0	0	0	0	0	0
3	3	5	6	3	7	7	7	7	7
3	3	5	7	2	6	6	6	6	6
3	3	5	9	2	7	7	7	7	6
3	3	6	6	0	5	0	0	0	0
3	3	6	7	3	6	6	5	5	5
3	3	6	9	2	10	10	3	3	3
3	3	7	7	0	0	0	0	3	0
3	3	7	8	4	8	8	7	7	7
3	3	7	9	3	7	7	4	4	4
3	3	8	8	0	7	0	0	0	0
3	3	8	9	3	10	10	7	7	7
3	3	9	9	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0
4	1	0	1	2	2	2	2	2	2
4	1	0	2	2	3	3	3	3	3
4	1	0	3	3	4	4	4	4	4
4	1	0	4	2	4	4	4	4	4
4	1	0	5	2	3	3	3	3	3
4	1	0	6	4	4	4	4	4	4
4	1	0	7	3	4	4	4	4	4
4	1	0	8	1	1	1	1	1	1
4	1	0	9	2	3	3	3	3	3
4	1	1	1	0	0	0	0	0	0
4	1	1	2	2	3	3	3	3	3
4	1	1	3	4	5	5	5	5	5
4	1	1	4	2	3	3	3	3	3
4	1	1	5	2	4	4	4	4	4
4	1	1	6	3	3	3	3	3	3
4	1	1	7	3	4	4	4	4	4
4	1	1	8	1	1	1	1	1	1

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
4	1	1	9	2	2	2	2	2	2
4	1	2	2	0	0	0	0	0	0
4	1	2	3	2	3	3	3	3	3
4	1	2	4	3	5	5	5	5	5
4	1	2	5	1	2	2	2	2	2
4	1	2	6	2	4	4	4	4	4
4	1	2	7	3	4	4	4	4	4
4	1	2	8	1	4	4	4	4	4
4	1	2	9	3	3	3	3	3	3
4	1	3	3	0	0	0	0	0	0
4	1	3	4	2	3	3	3	3	3
4	1	3	5	2	3	3	3	3	3
4	1	3	6	2	3	3	3	3	3
4	1	3	7	2	3	3	3	3	3
4	1	3	8	3	5	5	5	5	5
4	1	3	9	3	4	4	4	4	4
4	1	4	4	0	0	0	0	0	0
4	1	4	5	3	5	5	5	5	5
4	1	4	6	3	4	4	4	4	4
4	1	4	7	3	4	4	4	4	4
4	1	4	8	2	3	3	3	3	3
4	1	4	9	1	2	2	2	2	2
4	1	5	5	0	0	0	0	0	0
4	1	5	6	3	4	4	4	4	4
4	1	5	7	3	4	4	4	4	4
4	1	5	8	2	3	3	3	3	3
4	1	5	9	3	3	3	3	3	3
4	1	6	6	0	0	0	0	0	0
4	1	6	7	3	4	4	4	4	4
4	1	6	8	3	5	5	5	5	5
4	1	6	9	3	5	5	5	5	5
4	1	7	7	0	0	0	0	0	0
4	1	7	8	3	4	4	4	4	4
4	1	7	9	3	3	3	3	3	3
4	1	8	8	0	0	0	0	0	0
4	1	8	9	2	2	2	2	2	2
4	1	9	9	0	0	0	0	0	0
4	2	0	0	0	0	0	0	0	0
4	2	0	1	3	7	7	7	7	7
4	2	0	2	3	7	7	7	6	7
4	2	0	4	2	6	6	5	5	5
4	2	0	6	3	7	7	7	7	7
4	2	0	8	2	6	6	4	4	4

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
4	2	0	9	1	1	1	1	1	1
4	2	1	1	0	0	0	0	0	0
4	2	1	3	2	5	5	5	5	5
4	2	1	4	3	3	3	3	3	3
4	2	1	7	2	5	5	5	5	5
4	2	1	8	3	5	5	5	5	5
4	2	1	9	3	7	7	7	7	7
4	2	2	2	0	0	0	0	0	0
4	2	2	3	3	6	6	6	6	7
4	2	2	5	3	5	5	6	6	6
4	2	2	6	1	6	6	3	3	3
4	2	2	7	3	3	3	4	4	4
4	2	2	8	2	7	7	4	4	4
4	2	2	9	3	5	5	5	5	6
4	2	3	3	0	0	0	0	0	0
4	2	3	4	3	6	6	5	5	5
4	2	3	8	3	6	6	6	5	5
4	2	4	4	0	0	0	0	0	0
4	2	4	8	2	7	7	4	4	4
4	2	4	9	2	5	5	4	4	4
4	2	5	5	0	0	0	0	0	0
4	2	5	6	3	5	5	5	5	5
4	2	5	7	3	4	4	4	4	4
4	2	6	6	0	0	0	0	0	0
4	2	6	7	3	5	5	5	5	5
4	2	6	8	3	7	7	7	7	5
4	2	6	9	3	6	6	6	6	6
4	2	7	7	0	0	0	0	0	0
4	2	7	8	3	6	6	6	5	7
4	2	8	8	0	0	0	0	0	0
4	2	8	9	2	6	6	5	5	5
4	2	9	9	0	0	0	0	0	0
4	3	0	0	0	0	0	0	0	0
4	3	0	6	3	8	8	4	4	6
4	3	0	7	3	9	9	5	5	5
4	3	1	1	0	5	0	0	0	0
4	3	1	4	2	6	2	2	2	2
4	3	1	9	3	7	7	7	7	5
4	3	2	2	0	0	0	0	0	0
4	3	2	3	3	4	4	4	6	4
4	3	3	3	0	0	0	0	3	0
4	3	4	4	0	5	0	0	0	0
4	3	4	6	3	8	8	6	6	5

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
4	3	4	8	3	8	8	8	8	7
4	3	4	9	2	7	7	7	7	4
4	3	5	5	0	0	0	0	2	0
4	3	6	6	0	0	0	0	0	0
4	3	7	7	0	0	0	0	1	0
4	3	8	8	0	6	0	0	0	0
4	3	9	9	0	6	0	0	0	0
5	1	0	0	0	0	0	0	0	0
5	1	0	1	2	3	3	3	3	3
5	1	0	2	3	5	5	5	5	5
5	1	0	3	2	3	3	3	3	3
5	1	0	4	2	3	3	3	3	3
5	1	0	5	3	4	4	4	4	4
5	1	0	6	2	3	3	3	3	3
5	1	0	7	3	4	4	4	4	4
5	1	1	1	0	0	0	0	0	0
5	1	1	3	2	5	5	5	5	5
5	1	1	4	2	3	3	3	3	3
5	1	1	5	3	3	3	3	3	3
5	1	1	6	2	4	4	4	4	4
5	1	1	7	3	5	5	5	5	5
5	1	2	2	0	0	0	0	0	0
5	1	2	6	3	5	5	5	5	5
5	1	2	7	3	5	5	5	5	5
5	1	2	9	3	5	5	5	5	5
5	1	3	3	0	0	0	0	0	0
5	1	3	4	2	4	4	4	4	4
5	1	3	5	2	3	3	3	3	3
5	1	3	6	3	3	3	3	3	3
5	1	3	7	3	6	6	6	6	6
5	1	4	4	0	0	0	0	0	0
5	1	4	5	2	5	5	5	5	5
5	1	4	6	2	5	5	5	5	5
5	1	4	7	2	6	6	6	6	6
5	1	4	8	3	4	4	4	5	4
5	1	4	9	3	4	4	4	4	4
5	1	5	5	0	0	0	0	0	0
5	1	5	6	3	3	3	3	3	3
5	1	5	7	3	6	6	6	6	6
5	1	5	9	3	6	6	6	6	6
5	1	6	6	0	0	0	0	0	0
5	1	6	7	3	3	3	3	3	3
5	1	6	8	3	4	4	4	4	4

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
5	1	6	9	3	3	3	3	3	3
5	1	7	7	0	0	0	0	0	0
5	1	7	8	3	5	5	5	5	5
5	1	8	8	0	0	0	0	0	0
5	1	8	9	2	2	2	2	2	2
5	1	9	9	0	0	0	0	0	0

B Results for network pairs with lower bound

Table B.1: For 84 network combinations, the algorithm distance as determined by Algorithms 2 through 2.4, in comparison to a lower bound of the tail distance.

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
3	3	0	3	≥ 4	8	8	8	7	8
3	3	0	4	≥ 3	9	9	9	8	8
3	3	0	7	≥ 4	8	8	5	5	8
3	3	0	8	≥ 4	7	7	7	7	7
3	3	1	2	≥ 4	5	5	5	5	5
3	3	1	4	≥ 4	9	9	9	9	11
3	3	1	6	≥ 4	9	9	7	7	9
3	3	1	8	≥ 4	11	11	11	11	10
3	3	2	3	≥ 4	9	9	9	8	9
3	3	2	4	≥ 3	10	10	10	10	10
3	3	3	4	≥ 4	7	7	7	6	6
3	3	3	6	≥ 4	8	8	7	6	9
3	3	4	5	≥ 4	8	8	8	8	8
3	3	4	6	≥ 3	10	10	7	7	7
3	3	4	7	≥ 4	7	7	6	6	6
3	3	5	8	≥ 4	10	10	8	8	7
3	3	6	8	≥ 4	9	9	7	7	7
4	2	0	3	≥ 3	8	8	8	8	8
4	2	0	5	≥ 3	5	5	5	5	5
4	2	0	7	≥ 3	5	5	5	5	5
4	2	1	2	≥ 4	5	5	5	5	7
4	2	1	5	≥ 3	6	6	6	4	8
4	2	1	6	≥ 4	7	7	7	7	7
4	2	2	4	≥ 4	7	7	5	5	5
4	2	3	5	≥ 4	7	7	7	6	7
4	2	3	6	≥ 4	7	7	7	7	7
4	2	3	7	≥ 4	5	5	5	5	5
4	2	3	9	≥ 4	8	8	8	8	8
4	2	4	5	≥ 4	7	7	7	6	7
4	2	4	6	≥ 4	7	7	7	7	7
4	2	4	7	≥ 4	5	5	5	5	5
4	2	5	8	≥ 3	6	6	6	6	8
4	2	5	9	≥ 3	7	7	7	7	6
4	2	7	9	≥ 4	7	7	7	7	7
4	3	0	1	≥ 3	8	8	6	6	6
4	3	0	2	≥ 3	10	10	5	5	5
4	3	0	3	≥ 3	9	9	9	9	9

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
4	3	0	4	≥ 3	4	4	4	4	4
4	3	0	5	≥ 3	8	8	8	8	6
4	3	0	8	≥ 3	7	7	7	7	7
4	3	0	9	≥ 3	9	9	9	9	10
4	3	1	2	≥ 3	9	9	9	9	10
4	3	1	3	≥ 3	10	10	10	10	9
4	3	1	5	≥ 3	8	8	5	6	5
4	3	1	6	≥ 3	8	8	6	6	6
4	3	1	7	≥ 3	9	9	6	7	6
4	3	1	8	≥ 3	8	8	8	8	7
4	3	2	4	≥ 3	10	10	8	8	9
4	3	2	5	≥ 3	9	9	9	9	10
4	3	2	6	≥ 3	7	7	7	7	6
4	3	2	7	≥ 3	10	10	8	8	8
4	3	2	8	≥ 3	6	6	6	6	8
4	3	2	9	≥ 3	10	10	10	10	9
4	3	3	4	≥ 3	8	8	8	8	10
4	3	3	5	≥ 3	10	10	10	10	9
4	3	3	6	≥ 3	7	7	7	7	7
4	3	3	7	≥ 3	8	8	8	7	11
4	3	3	8	≥ 3	7	7	7	7	6
4	3	3	9	≥ 3	8	8	8	8	8
4	3	4	5	≥ 3	7	7	7	7	6
4	3	4	7	≥ 3	9	9	5	6	5
4	3	5	6	≥ 3	10	10	10	10	6
4	3	5	7	≥ 3	10	10	10	10	7
4	3	5	8	≥ 3	8	8	8	8	10
4	3	5	9	≥ 3	9	9	9	9	9
4	3	6	7	≥ 3	10	10	8	8	8
4	3	6	8	≥ 3	10	10	10	10	10
4	3	6	9	≥ 3	9	9	9	9	8
4	3	7	8	≥ 3	6	6	6	6	8
4	3	7	9	≥ 3	8	8	8	8	8
4	3	8	9	≥ 3	9	9	9	9	9
5	1	0	8	≥ 3	5	5	5	5	5
5	1	0	9	≥ 3	4	4	4	4	4
5	1	1	2	≥ 4	5	5	5	4	5
5	1	1	8	≥ 4	5	5	5	5	5
5	1	1	9	≥ 4	5	5	5	5	5
5	1	2	3	≥ 3	5	5	5	5	5
5	1	2	4	≥ 4	5	5	5	5	5
5	1	2	5	≥ 3	5	5	5	5	5
5	1	2	8	≥ 3	5	5	5	5	5

Leaves	Reticulations	N	N'	$d_{\text{tail}}(N, N')$	Distance as determined by Algorithm				
					2	2.1	2.2	2.3	2.4
5	1	3	8	≥ 4	6	6	6	6	6
5	1	3	9	≥ 4	5	5	5	5	5
5	1	5	8	≥ 4	6	6	6	6	6
5	1	7	9	≥ 4	4	4	4	4	4