BSc verslag TECHNISCHE WISKUNDE

Tree-child Network Containment

ROBBERT HUIJSMAN

Technische Universiteit Delft

Begeleiders

Dr.ir. L.J.J. van Iersel, R. Janssen, MSc en Y. Murakami, MSc

Overige commissieleden

Dr.B. van den Dries

July, 2019 Delft

Abstract

Interest in phylogenetic trees for histories of species and DNA has spawned many problems, one of which is TreeContainment; a problem that asks whether a tree is contained within a network. The TreeContainment problem is proven to be NP-hard for general trees and networks, however it is solvable in polynomial time for networks that meet the tree-child restriction. An algorithm to solve TreeContainment for binary tree-child networks has been created previously with quadratic running time (van Iersel, Semple, Steel, 2010). Janssen and Murakami have recently created a new algorithm that solves a larger problem NetworkContainment, for semi-binary tree-child networks (Janssen, Murakami, 2019). This new algorithm uses tree-child sequences introduced by Linz and Semple, but there has not been an implementation of it until now. In this paper I show an implementation (using Python) of this algorithm, in which I have made a modification that increases its speed on networks with large indegrees. Furthermore I have proven in this paper that the output of this algorithm remains correct under this modification, and that the running time of the modified algorithm is now linear without requiring a constant maximum indegree at all.

1 Introduction

Phylogenetic trees are commonly used in evolutionary biology to represent links in history between species. At times, networks are used instead of trees to display the presence of events such as hybridization or horizontal gene transfer. Many theoretical problems arise from these networks, one of which is NetworkContainment. This problem asks the question of whether a network is contained within another network.

In general, this problem is NP-hard, but an algorithm has been created by Janssen and Murakami which solves this problem for semi-binary tree-child networks in linear time (Janssen, Murakami, 2019). Additionally, they have shown that this algorithm runs in linear time, given that the maximum indegree of the networks is constant. This algorithm finds a cherry picking sequence that reduces the first network to a single leaf, and then applies this sequence to the second network. In their paper, Janssen and Murakami have shown that the second network is contained within the first network if and only if this cherry picking sequence also reduces the second network to a single leaf.

This paper contains an implementation of this algorithm, alongside a number of related theorems and proofs and an improvement to the algorithm that increases its speed. The text starts with several theorems and proofs that show that the cherry picking theory behind the algorithm works. The algorithm consists of a number of functions, each displayed in the next section of this paper alongside the reasoning behind their code and a theoretical proof for the time complexity of their code. The text continues with a section containing several theorems and proofs that establish the correctness of the improved algorithms output. A final section is dedicated to a number of tests that demonstrate the speed of the improved algorithm in practice, and show a comparison between the direct implementation and the improved implementation. Lastly the full code is displayed at the bottom of the text.

2 Definitions

Definition 2.1. A *phylogenetic network* on a non-empty taxa X is a directed acyclic graph made up of at most 4 different kinds of nodes.

1. A single *root* with indegree 0 and outdegree 1 (in this report).
2. *Leaves* with indegree 1 and outdegree 0, labeled bijectively with X.
3. *Tree nodes* with indegree 1 and outdegree at least 2.
4. *Reticulation nodes* with indegree at least 2 and outdegree 1.

The root and a leaf must be included in every phylogenetic network. From now on I will refer to these phylogenetic networks as simply *networks*. If each tree node in a network has outdegree 2 then the network is called *semi-binary*. If, additionally, each reticulation node in a network has indegree 2, then the network is called *binary*. I will assume throughout this paper, that all networks are semi-binary.

If there is an edge (x, y) between x and y, then x is a *parent* of y and y is a *child* of x. A network is *tree-child* if every node that is not a leaf has a child that is either a tree node or a leaf (see figure 1).

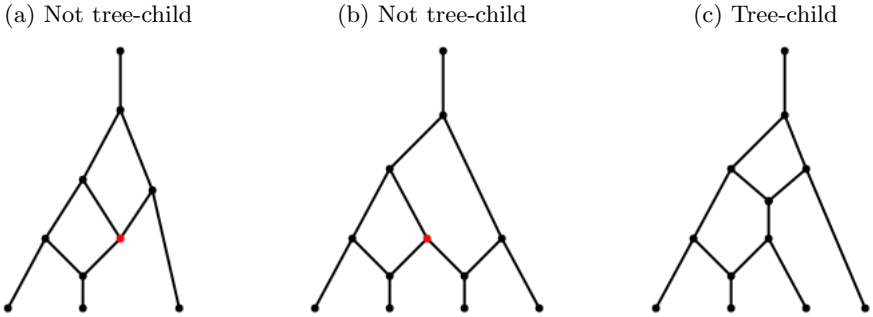


Figure 1: Examples of networks, one of which is tree-child. The red nodes are non-leaf nodes that do not have a child that is a leaf node or a tree node.

Definition 2.2. Let N and N' be networks on the finite sets of leaves L and L' respectively such that $L' \subset L$. An *embedding* of N' in N is an injective function that maps every node of N' to a node of N such that their leaves are identical, and maps every edge in N' to edge disjoint paths in N such that the end points of the edges and paths are identical. If an embedding of N' in N exists, then N' is a *subnetwork* of N.

When dealing with nodes in multiple networks, I will often refer to a node x in a network N as x^N.

2.1 Cherry-picking sequences

Definition 2.3. Let S be a sequence of ordered pairs of two leaves. If every second coordinate of an ordered pair in S is a first coordinate of a different pair in the remainder of S, or a second coordinate of the last pair, then S is a *cherry picking sequence*.

If, additionally, no first coordinate of an ordered pair in S is used as second coordinate of an ordered pair in the remainder of S, then S is a *tree-child sequence*.

Let $0 \leq n \leq |S|$. Then $S_{[:n]}$ is a subsequence of S containing only the first n ordered pairs, and $S_{[n:]}$ is the subsequence of S containing only the last $|S| - n + 1$ ordered pairs.

Definition 2.4. Let x and y be leaves in a network, and let p_x and p_y denote the parents of x and y respectively. An ordered pair (x, y) is called a *cherry* if $p_x = p_y$. An ordered pair (x, y) is called a *reticulated cherry* if p_x is a reticulation node and p_y is also the parent of p_x. See figure 2.

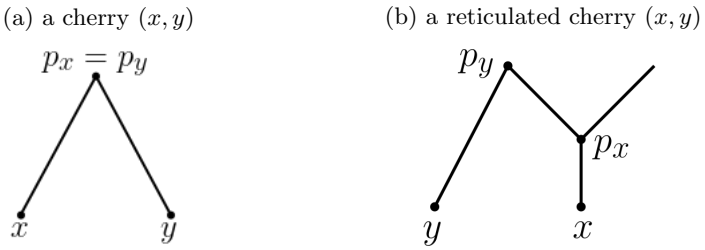


Figure 2: The cherry and reticulated cherry structures

Definition 2.5. Let (x, y) be a cherry, let p be the parent of x and y and let pp be the parent of p. The act of *picking a cherry* (x, y) or *reducing a cherry* (x, y) is defined as taking the following steps (see figure 3):

1. Remove the leaf x and the edge (p, x) connecting it to its parent.
2. Remove the node p and its incident edges (pp, p) and (p, y), and add an edge (pp, y) connecting y to pp.

Note that after this first step, the node p has indegree 1 and outdegree 1, and does not fit the description of any of the four earlier mentioned types of nodes in phylogenetic networks. The second step is therefore taken to solve this problem and replace this node that connects pp and y with a simple edge (pp, y). I may refer to this method of removing nodes with indegree 1 and outdegree 1 as "cleaning up".

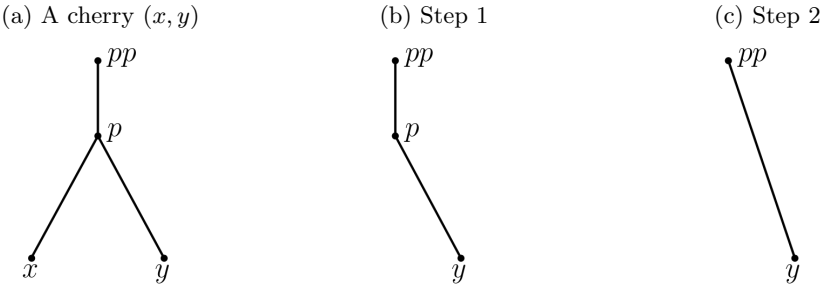


Figure 3: Picking a cherry (x, y)

Definition 2.6. Let (x, y) be a reticulated cherry, let p_x be the parent of x, let p_y be the parent of y, let pp_x be another parent of p_x that is not p_y and let pp_y be the parent of p_y. The act of *picking a reticulated cherry* (x, y) or *reducing a reticulated cherry* (x, y) is defined as taking the following steps (see Figure 4):

1. Remove the edge (p_y, p_x) between p_y and p_x.
2. Remove the node p_y and its incident edges (pp_y, p_y) and (p_y, y), and add an edge (pp_y, y) connecting y to pp_y.
3. If p_x has indegree 1 after the edge removal, remove the node p_x and its incident edges (pp_x, p_x) and (p_x, x) and add the edge (pp_x, x).

Note that this third step is only required if, prior to the reduction, p_x is a reticulation node with indegree 2.

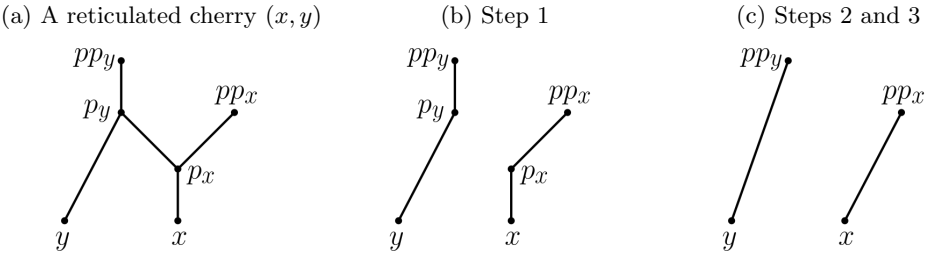


Figure 4: Picking a reticulated cherry (x, y) when p_x has indegree 2

Definition 2.7. Applying a sequence S of pairs to a network N is the act of reducing each pair of S in N in the order they appear in S. The resulting network is written as NS.

If this resulting network NS contains only 1 leaf, then S *reduces* N to a leaf. A network that can be reduced to a leaf is called a *cherry picking network*.

3 Containment and cherry picking

This section contains a small number of lemmas and proofs that show the relation between cherry picking sequences that reduce networks and network containment. The ideas of these lemmas and their proofs were created by Murakami and Janssen (Janssen, Murakami, 2019).

Lemma 1. Let N and N' be binary cherry picking networks on the finite sets of leaves L and L' respectively, and let $L' \subseteq L$. Then N' is a subnetwork of N if there exists a cherry picking sequence S of N (it only includes pairs that reduce cherries or reticulated cherries in N) that reduces N and N' to the same leaf.

Proof. Let S' be the cherry picking sequence containing only the pairs from S that are used in reducing N', in the order they occur in S. Let $f : \{1, ..., |S'|\} \to \{1, ..., |S|\}$ be the function that sends a given index of a pair in S' to the corresponding index of that pair in S ($S'_n = S_{f(n)}$).

Since S' contains only the pairs from S that are used in reducing N', while still in the order they occur in S, we have that $S'_i > S'_j$ for $i > j$. Since additionally $|S| \geq |S'|$, the index of a given pair in S is either identical to its index in S' or larger. Therefore, f is a strictly increasing function ($f(x) > f(y)$ for $x > y$).

This proof shows that the cherry picking network $N'S'_{[:n]}$ is a subnetwork of the cherry picking network $NS_{[:f(n)]}$ using backward induction. Note that $N'S'_{[:0]} = N'$ since $S'_{[:0]}$ is an sequence without elements and let $f(0) = f(1) - 1$.

The base case of the induction claims that the lemma holds for $n = |S'| - 1$. Let the pair (x, y) be the last cherry in the cherry picking sequence: $(x, y) = S'_{|S'|}$. If (x, y) is the last cherry, then $N'S'_{[:|S'|-1]} = (x, y)$ and the cherry picking network exists out of only the leaves x and y. Since this pair is an element of S', it is also an element of S and its index can be found with f. Therefore the cherry picking network $NS_{[:f(|S'|-1)]}$ also has the leaves x and y. Since $N'S'_{[:|S'|-1]}$ consists solely of the cherry (x, y), there exists an embedding of $N'S'_{[:|S'|-1]}$ into $NS_{[:f(|S'|-1)]}$.

To create this embedding, map x to x, map y to y and map the root of $N'S'_{[:|S'|-1]}$ to the root of $NS_{[:f(|S'|-1)]}$. There exists a path p from the root of $NS_{[:f(|S'|-1)]}$ to x. Let z be the node along this path, from where a disjoint path q to y exists (note that this could be the tree node whose parent is the root of $NS_{[:f(|S'|-1)]}$). This node z always exists, since x and y are separate nodes. Now embed the parent p of x and y to this node z. Embed the edge between the root of $N'S'_{[:|S'|-1]}$ and p, to the part of the path p, that goes from the root of $NS_{[:f(|S'|-1)]}$ to the node z. Embed the edge between the parent p and x to the part of the the path p, that goes from the node z to x. Finally, embed the edge between the parent p and y to the path q. This embedding is correct, and therefore, $N'S'_{[:|S'|-1]}$ is a subnetwork of $NS_{[:f(|S'|-1)]}$.

This proves the base case $n = |S'|-1$ of the induction. To prove the induction step, we must prove that $N'S'_{[:n]}$ is a subnetwork of $NS_{[:f(n+1)-1]}$ given the induction hypothesis which states that $N'S'_{[:n+2-1]} = N'S'_{[:n+1]}$ is a subnetwork of $NS_{[:f(n+2)-1]}$. Since f is a strictly increasing function, $NS_{[:f(n+2)-1]}$ is a subnetwork of $NS_{[:f(n+1)]}$. Combining this and the induction hypothesis gives that $N'S'_{:n+1}$ is a subnetwork of $NS_{[:f(n+1)]}$.

Since $N'S'_{[:n+1]}$ is a subnetwork of $NS_{[:f(n+1)]}$, there is an embedding E of $N'S'_{[:n+1]}$ in $NS_{[:f(n+1)]}$. To show that $N'S'_{[:n]}$ is a subnetwork of $NS_{[:f(n+1)-1]}$, we extend this embedding E to an extended embedding E'. Let $S'_{n+1} = (x, y)$, let p_x be the parent of x and let p_y be the parent of y, both in $N'S'_{[:n]}$. Additionally, let pp_x be the parent of p_x and let pp_y be the parent of p_y, both in $N'S'_{:n}$, chosen so that they correspond with the parents $p_x^{N'S'_{[:n+1]}}$ of x and $p_y^{N'S'_{[:n+1]}}$ of y respectively. If a pair (x, y) is reduced, the nodes x, p_x and p_y can potentially be removed. Additionally, the edges (p_y, y), (pp_y, p_y), (p_x, x), (pp_x, p_x), (p_x, p_y) and edges incident to p_x could be removed. Therefore these nodes and edges must be mapped explicitly in the extended embedding E', while the other nodes and edges that are not involved in the cherry (x, y) can be

extended naturally.

In this extension E' of the embedding E, the node p_y is mapped to the node p_y in $NS_{[:f(n+1)-1]}$. The edge (pp_y, p_y) is mapped to the path from $(E(p_y^{N'S'_{[:n+1]}}$ to $p_y^{NS_{[:f(n+1)-1]}}$ (these are colored blue in figures 5 and 6). The edge (p_y, y) is mapped to the edge $(p_y^{NS_{[:f(n+1)-1]}}, y)$ (these are colored green in figures 5 and 6). This divides the previous edge $(p_y^{N'S'_{[:n+1]}}, y)$ of the embedding E into the two new edges. In the original embedding E, the leaf p_y was not used since it didn't exist in $N'S'_{[:n]}$. The further mappings are divided into two separate cases:

1. **$N'S'_{[:n+1]}$ has the leaf x**

   In this case, the leaf x is mapped to the leaf x in $NS_{[:f(n+1)-1]}$ already by the natural extension of the embedding A. The node p_x is mapped to $p_x^{NS_{[:f(n+1)-1]}}$. The edge (p_y, p_x) is mapped to the edge $(p_y^{NS_{[:f(n+1)-1]}}, p_x^{NS_{[:f(n+1)-1]}})$ (these are colored red in figure 5). In the original embedding E, the node $p_x^{NS_{[:f(n+1)-1]}}$ was not used since it wasn't in $N'S'_{[:n]}$ and neither was the edge $(p_y^{NS_{[:f(n+1)-1]}}, p_x^{NS_{[:f(n+1)-1]}})$. The edge (pp_x, p_x) is mapped to the path from $(E(p_x^{N'S'_{[:n+1]}}$ to $p_x^{NS_{[:f(n+1)-1]}}$ (these are colored purple in figure 5). the edge (p_x, x) is mapped to the edge $(p_x^{NS_{[:f(n+1)-1]}}, x)$ (this edge is colored gold in figure 5). This divides the previous edge $p_x^{N'S'_{[:n+1]}}, x)$ of the embedding E into the two new edges. Therefore, these new mappings of the extended embedding E' are disjoint.

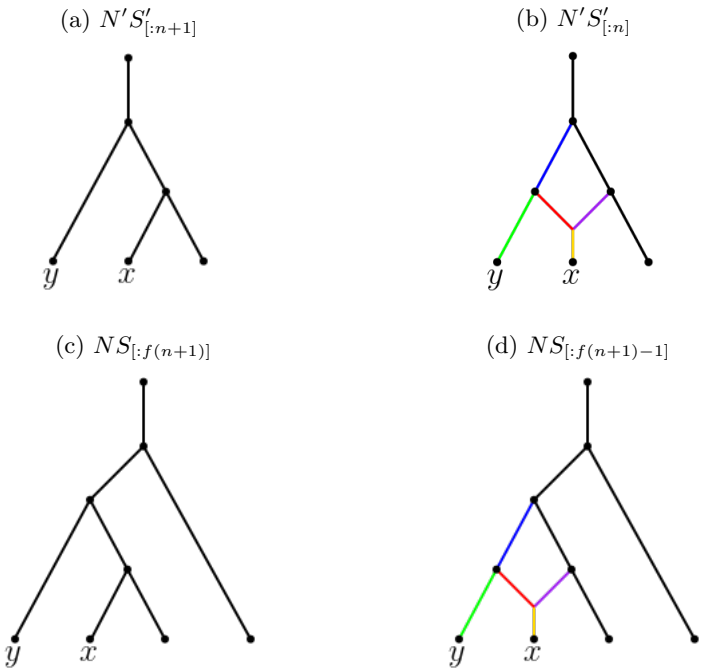


Figure 5: An example where $N'S'_{[:n+1]}$ has the leaf x

2. **$N'S'_{[:n+1]}$ does not have the leaf x**

In this case, the leaf x is mapped to the leaf $x^{NS_{[:f(n+1)-1]}}$. The edge (p_y, x) is mapped to a path from $p_y^{NS_{[:f(n+1)-1]}}$ to $x^{NS_{[:f(n+1)-1]}}$ (these are colored red in figure 6). In the original embedding E, the leaf $x^{NS_{[:f(n+1)-1]}}$ was not used since it didn't exist in $N'S'_{[:n]}$. The node $p_y^{NS_{[:f(n+1)-1]}}$ doesn't exist in $N'S'_{[:n]}$, meaning that the edge (p_y, x) was not used either in E. Therefore, these new mappings of the extended embedding E' are disjoint.

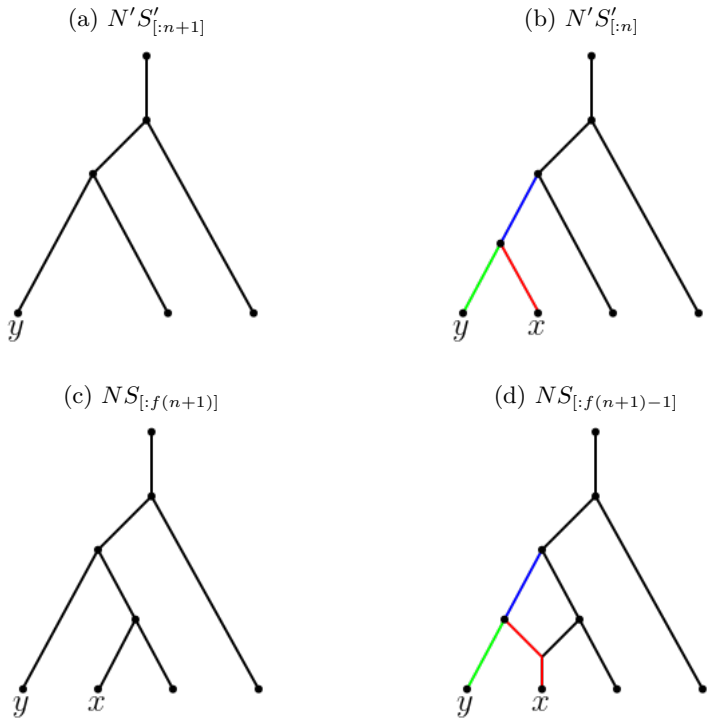


Figure 6: An example where $N'S'_{[:n+1]}$ does not have the leaf x

Extending the embedding E with these mappings ensures that every node or edge in $N'S'_{[:n+1]}$ has been mapped to a unique node or edge from $NS_{[:f(n+1)-1]}$, and all these mappings are disjoint. Therefore, E' is an embedding of $N'S'_{[:n]}$ in $NS_{[:f(n+1)-1]}$. Since there is an embedding, $N'S'_{[:n]}$ is a subnetwork of $NS_{[:f(n+1)-1]}$ which proves the induction step. Thus $N'S'_{[:n]}$ is a subnetwork of $NS_{[:f(n)]}$ for $0 \leq n \leq |S'|$. Therefore $N'S'_{[:0]} = N'$ is a subnetwork of $NS_{[:f(0)]}$. Since $NS_{[:f(0)]}$ is a subnetwork of N, we have that N' is a subnetwork of N (by the transitive property of being a subnetwork). □

Lemma 2. Let N and N' be semi-binary cherry picking networks on the finite sets of leaves L and L' respectively, and let $L' \subseteq L$. Then N' is a subnetwork of N if there exists a cherry picking sequence S that reduces N and N' to the same leaf.

A proof to this lemma can be created by extending the proof for lemma 1.

Lemma 3. If A is a subnetwork of B and B is a subnetwork of C then A is a subnetwork of C.

Proof. Since B is a subnetwork of C, there exists an embedding E_C that injectively maps all

nodes and edges of B to nodes and edge-disjoint paths of C respectively. Since A is a subnetwork of B, there exists an embedding E_B that injectively maps all node and edges of A to nodes and edge-disjoint paths of B respectively. Then every node or edge a in A can be mapped injectively to the node or path $E_C(E_B(a))$ in C. Every edge e in A can be mapped injectively to a edge-disjoint path of C by first using the embedding E_B and splitting up the resulting path into disjoint edges e': $E_B(e) = \bigcup_{e' \in E_B(e)}$. Now every edge e' can be mapped to a path with the second embedding E_C by $\bigcup_{E_C(e') \ : \ e' \in E_B(e)}$. These are edge-disjoint paths, and which means that this is a correct embedding of A into C and therefore A is a subnetwork of C. □

In this paper I will refer to this lemma as the transitive property of being a subnetwork.

Lemma 4. Let N be a tree-child network with N' a tree-child subnetwork of N on the same leaf set X. If S is a tree-child sequence then $N'S$ is a subnetwork of NS.

Proof. This proof shows that $N'S$ is a subnetwork of NS using induction over the length of the tree-child sequence S. The base case of the induction claims that the lemma holds for an empty S. This is easily proven since $NS = N$ and $N'S = N'$ for S empty. To prove the induction step, we must prove that the lemma holds for a tree-child sequence with length $n + 1$ given the induction hypothesis which states that the lemma holds for a tree-child sequence with length n. Let S' be the tree-child sequence S without its last pair (such that $S = S'(x, y)$). This new tree-child sequence S' has length n and due to the induction hypothesis we have that $N'S'$ is a subnetwork of NS'. The rest of this proof is divided into different cases:

1. ***x* or *y* is not in *N'S'***. Since a leaf that is the first coordinate of a pair cannot be used as a second coordinate of a pair in the rest of a tree-child sequence, the leaf y cannot be the first coordinate of any pair in S' (since it is the second coordinate of (x, y)). Since leaves are only removed when they are the first coordinate of a cherry, y must still be in the network $N'S'$. In this case either x or y is not in $N'S'$, and since y is in $N'S'$, x is not. Because $N'S'$ is a subnetwork of NS', there exists an embedding E of $N'S'$ in NS'. However since x is not in $N'S'$, the embedding E does not use the edge that is deleted when (x, y) is reduced. Therefore, $N'S = N'S'$ and $N'S$ can be embedded into NS.

2. ***x* and *y* are in *N'S'***. This case is dependent on whether the pair (x, y) is a cherry, a reticulated cherry or neither in NS':

   (a) **(x, y) is a cherry in *NS'***. Since x and y are in $N'S'$ and form a cherry in NS', they must form a cherry in $N'S'$ as well due to the embedding of $N'S'$ in NS'. Therefore the embedding E requires no alteration in this case, apart from the removal of the edge incident to x in both NS' and $N'S'$.

   (b) **(x, y) is a reticulated cherry in *NS'***. This case is dependent on whether the embedding E uses the edge (p_y, p_x) from the parent p_y of y to the parent p_x of x:

      i. **The embedding uses the edge (p_y, p_x)**. Since this edge (p_y, p_x) is mapped to some edge of $N'S'$, the edges (p_x, x) and (p_y, y) must also be used in the mapping to get to x and y respectively. Therefore there must be a cherry or reticulated cherry (x, y) in $N'S'$. If (x, y) is a reticulated cherry in $N'S'$, reducing (x, y) will remove the same edge in both networks, which means that the embedding requires no alteration.

      If (x, y) is a cherry in $N'S'$, reducing it leaves only the edge (p_y, y) in $N'S$ where p_y is pp_y the parent of p_y in $N'S'$. Since (x, y) is a reticulated cherry in NS',

reducing it leaves (among another edge (p_x, x)) this same edge (p_y, y) which can be mapped to itself in $N'S$.

See figure 7.

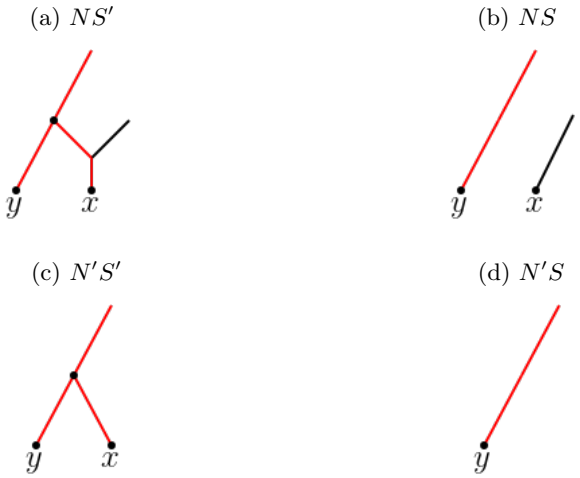


Figure 7: Parts of the networks in the case where the embedding uses the edge (p_y, p_x). Embeddings are shown by the red lines.

ii. **The embedding does not use the edge (p_y, p_x).**

Since $N'S'$ is a subnetwork of NS' and its embedding doesn't utilize the edge (p_y, p_x), it is also a subnetwork of NS that doesn't have this edge (p_y, p_x). Now $N'S$ is a subnetwork of $N'S'$ (since it is the same network with a picked reticulated cherry) and $N'S'$ is a subnetwork of NS, and therefore we have that $N'S$ is a subnetwork of NS due to the transitive property of being a subnetwork.

(c) **(x, y) is neither**.

Since (x, y) is not a cherry nor a reticulated cherry in NS', picking it doesn't alter the network ($NS' = NS$). Again as $N'S$ is a subnetwork of $N'S'$ (since it is the same network with a picked reticulated cherry) and $N'S'$ is a subnetwork of NS' (which is NS), we have that $N'S$ is a subnetwork of NS due to the transitive property of being a subnetwork.

□

Corollary 1. Let N be a tree-child network with a tree-child subnetwork N' on the same leaf set X. If S is a Tree-child sequence that reduces N, then S reduces N' as well.

Theorem 1. Let N and N' be tree-child networks on the same leaf set X. Then N' is a subnetwork of N if and only if any tree-child sequence of N reduces N'.

Proof. This follows from corollary 1 and lemma 2. □

4 The code

The code for this algorithm is written in python using the *networkx* module. This module implements a network (not specifically phylogenetic) class based on an adjacency list implemented using a dictionary of dictionaries. Due to this implementation, actions important for the algorithm such as checking a node's indegree or outdegree, removing a node or edge from the network and accessing a parent or child are done in constant time.

In these functions, variable names such as p (meaning a parent), pp (meaning a parent of a parent) or pc meaning a (child of a parent) are often used. These are not used as formal definitions, but rather as mnemonics.

4.1 find_cherry

This function takes a network N and a leaf x as its input, and returns a list of all cherries or reticulated cherries that have the input leaf as their second coordinate.

```
1  def find_cherry(N, x):
2      lst = list()
3      for p in N.predecessors(x):
4          if N.in_degree(p) == 1:
5              for pc in N.successors(p):
6                  if pc != x:
7                      t = N.out_degree(pc)
8                      if t == 0:
9                          lst.append((pc, x))
10                     if t == 1:
11                         for pcc in N.successors(pc):
12                             if N.out_degree(pcc) == 0:
13                                 lst.append((pcc,x))
14     return lst
```

The function starts by creating an empty list and accessing the parent p of the input leaf x. Since $N.predecessors(x)$ returns an iterator over all parents of x, the function uses a for-loop to access the parent of x even though there is only one. It proceeds to check whether the parent p is a tree node by looking at the indegree of p. If its indegree is 1, the parent p is a tree node and the function accesses the other child of the parent pc. The function then checks the outdegree of the other child pc. If this outdegree of pc is 0, pc is also a leaf and the function adds the cherry (pc, x) to the list. If the outdegree of pc is 1, pc is a reticulation and the function accesses the child pcc of pc. It checks whether this pcc is a leaf by checking if its outdegree is 0, and appends the reticulated cherry (pcc, x) to the list if pcc is a leaf.

Note that the returned list will either be empty or have a single element for semi-binary networks, since the node x can only be the second coordinate of either a single cherry or a single reticulated cherry.

Lemma 5. The function $find\_cherry(N, x)$ finds all cherries and reticulated cherries with x as their second coordinate.

Proof. Let x and y be leaves in N and let (y, x) be a cherry in N. Since (y, x) is a cherry, the leaves x and y have the same parent p by definition. The function $find\_cherry(N, x)$ accesses the parent p and checks whether any of its children other than x have outdegree 1, it will find y and add the cherry (y, x) to the list.

Now let (y, x) instead be a reticulated cherry. By definition, p_y is a reticulation node and the parent p_x of x is also the parent of p_y. Since p_y is a reticulation node and p_x is a parent of p_y, the node p_x can not be a reticulation node itself due to the tree-child restriction. The function $find\_cherry(N, x)$ therefore accesses the parent p_x (checking if it has indegree 1) of x and checks whether any of its children other than x have outdegree 1. This if statement will yield the node p_y (since it is the other child of p_x), and the function will continue to check whether its child y has outdegree 0. Since y is a leaf, the function will proceed to add the reticulated cherry (y, x) to the list.

Since this function (as explained in the proof) uses all requirements in the definitions of cherries and reticulated cherries to find these pairs, the function will not output any pair of leaves that is not a cherry or reticulated cherry with with its input leaf as a second coordinate. □

Lemma 6. The running time of $find\_cherry$ is constant.

Proof. Creating an empty list and adding an element to a list take constant time. The leaf x only has a single parent, so accessing it takes constant time. The indegree and outdegree of a node is saved in the networkx implementation of the network, which makes accessing the indegree take constant time. Since the network is semi-binary, accessing the children of p or pc takes constant time and these for-loops both iterate over only 2 nodes. Checking whether 2 nodes are identical takes constant time as well. Since these are all the required actions (and all these required actions take constant time), the function runs in constant time. □

4.2 find_ret_cherry

This function takes a network N and a leaf x as its input and returns a list of all reticulated cherries that have the input leaf as their first coordinate.

```
1  def find_ret_cherry(N, x):
2      lst = list()
3      for p in N.predecessors(x):
4          if N.out_degree(p) == 1:
5              for pp in N.predecessors(p):
6                  for ppc in N.successors(pp):
7                      if ppc != p:
8                          if N.out_degree(ppc) == 0:
9                              lst.append((x, ppc))
10     return lst
```

Like $find\_cherry$, this function starts by creating an empty list and accessing the parent p of the input leaf x. Subsequently it checks whether the parent p is a reticulation by checking whether its outdegree is 1. Since a reticulation has multiple parents, the function iterates through the parents of p using pp as variable for the parent. In this iteration, the function accesses the other child ppc of pp that is not the original parent p. It checks whether this ppc is a leaf by checking if its outdegree is 0, and appends the reticulated cherry (x, ppc) to the list if ppc is a leaf.

Lemma 7. The function $find\_ret\_cherry(N, x)$ finds all reticulated cherries with x as their first coordinate.

Proof. Let x and y be leaves in N and let (x, y) be a reticulated cherry in N. By definition, the parent p_x of x is a reticulation node and the parent p_y of y is also the parent of p_x.

Since p_x is a reticulation node and p_y is a parent of p_x, the node p_y can not be a reticulation node itself due to the tree-child restriction. The function accesses the parent p_x and checks whether it is a reticulation node. It loops through the parents of p_x, one of which is p_y. It then finds the child y of p_y that is not p_x, checks whether it is a leaf, and proceeds to add the reticulated cherry (x, y) to the list.

Since this function (as explained in the proof) uses all requirements in the definition of reticulated cherries to find the reticulated cherry, the function will not output any pair of leaves that is not a reticulated cherry with its input leaf as a first coordinate. □

Lemma 8. The running time of $find\_ret\_cherry$ is $O(\text{indegree(p)})$.

Proof. The leaf x only has a single parent p, so accessing it takes constant time. The parent p however can be a reticulation with multiple parents (using pp as variable for a parent of p). This causes a dependency on the indegree of p for accessing its parents. Because of the tree-child restriction, no parent pp of p can be a reticulation. Therefore they will have two children each and accessing these will take constant time. Since accessing the parents of p is linearly dependant on its indegree and the rest of these actions take constant time, this function's running time is $O(\text{indegree}(p))$ □

4.3 check_cherry

This function takes a network N and 2 leaves (x, y) as its input. The function returns 1 if (x, y) is a cherry, 2 if (x, y) is a reticulated cherry and returns False otherwise.

```
1  def check_cherry(N, (x, y)):
2      if N.has_node(x):
3          if N.has_node(y):
4              for px in N.predecessors(x):
5                  for py in N.predecessors(y):
6                      if px == py:
7                          return 1
8                      if N.out_degree(px) == 1:
9                          if px in N.successors(py)
10                             return 2
11     return False
```

To avoid possible errors, the function starts by checking whether the nodes x and y actually exist within the given network N. It iterates over the parents px of x and py of y and checks if they are identical. If they have a common parent, the function returns 1. If they do not have a common parent, the function checks if px is a reticulation by looking at the outdegree of px. If px is a reticulation, the function checks whether py is in the set of parents of px. If this is true, (x, y) is a reticulated cherry and the function returns 2. In the case where neither 1 or 2 are returned, the function instead returns False.

Note that this function does not actually check whether the nodes x and y are leaves. Why this is not necessary for the algorithm is explained at $find\_tcs$.

Lemma 9. The running time of *check_cherry* is constant.

Proof. Checking whether a node exists within a network, or whether 2 nodes are identical takes a constant time. The leaves x and y both have only a single parent px and py respectively, so accessing these takes a constant time. Checking whether px is contained within the children of

py takes constant time because of the semi-binary tree property. Therefore, *check_cherry* runs in a constant time. □

4.4 reduce_pair

This function takes a network N and a pair (x, y) as its input and reduces this cherry or reticulated cherry within the network. The function returns True if the cherry or reticulated cherry has been successfully reduced and False otherwise.

```
1  def reduce_pair(N, (x, y)):
2      k = check_cherry(N, (x, y))
3      if k == 1:
4          for px in N.predecessors(x):
5              N.remove_node(x)
6              for ppx in N.predecessors(px):
7                  N.remove_node(px)
8                  N.add_edge(ppx,y)
9              return True
10     if k == 2:
11         for px in N.predecessors(x):
12             for py in N.predecessors(y):
13                 N.remove_edge(py,px)
14                 if N.in_degree(px) == 1:
15                     for ppx in N.predecessors(px):
16                         N.add_edge(ppx, x)
17                         N.remove_node(px)
18                 for ppy in N.predecessors(py):
19                     N.add_edge(ppy, y)
20                     N.remove_node(py)
21                 return True
22     return False
```

It starts by using the function *check_cherry* to determine whether the given pair (x, y) is a cherry ($k = 1$) or a reticulated cherry ($k = 2$).

If (x, y) is a cherry, the function accesses the parent px of x before removing x itself from the network. The used networkx function *remove_node* also removes all edges incident to x. After removing x, the parent px now only has a single child. Additional cleanup is done by accessing the parent ppx of the parent px, removing px itself and creating an edge from ppx to x.

If (x, y) is a reticulated cherry, the function accesses both parents px and py of x and y respectively and removes the edge connecting them. Since py was a parent of px, we now have a situation where the (previously) reticulation node px may only have a single remaining parent. The function checks this and cleans up by accessing the parent ppx of px, adding an edge from ppx to x and removing px itself. The parent py of px has also been affected by the removal of the edge between px and py, since it now has only a single child. To solve this issue, the function accesses the parent ppy of py, removes the node py and adds an edge from ppy to y.

Lemma 10. The running time of *reduce_pair* is constant.

Proof. The function *check_cherry* runs in a constant time. Accessing the parents px and py of the leaves x and y respectively takes constant time. The time required to add or remove an edge or node from the network is constant. If the pair (x, y) is a cherry, the parent px of x will be a tree node and accessing its parent ppx takes constant time. If the pair (x, y) is a reticulated cherry, the parent ppx of px is a reticulation. This parent is only accessed when px has a single

edge remaining, which is therefore done in constant time as well. Finally, since *py* is a tree node, accessing its parent *ppy* also takes constant time. Therefore, this function runs in constant time. □

4.5 find_tcs

This function takes a network N as its input and returns a cherry picking sequence in the form of a list.

```
1  def find_tcs(N):
2      lst_todo = list()
3      for x in N.nodes():
4          if N.out_degree(x) == 0:
5              cherry1 = find_cherry(N,x)
6              lst_todo.extend(cherry1)
7      lst_tcs = list()
8      while lst_todo:
9          cherry = lst_todo.pop()
10         k = check_cherry(N, cherry)
11         if (k == 1) or (k == 2):
12             reduce_pair(N, cherry)
13             lst_tcs.append(cherry)
14             lst_todo.extend(find_cherry(N, cherry[1]))
15             lst_todo.extend(find_ret_cherry(N, cherry[1]))
16     return lst_tcs
```

It starts by creating an empty list *lst_todo*. The function iterates through the nodes of the network and checks if a node is a leaf by checking its outdegree. If the node is a leaf, the function uses the *find_cherry* function and extends its output to the list *lst_todo*. After the function has completed this for-loop, the list *lst_todo* contains all initially available cherries and reticulated cherries (see lemma 13 in section 5.1).

The function continues by creating a second list *lst_tcs* and iterating through the cherries of *lst_todo* until it is empty. In this while-loop, the function removes a cherry (x, y) from *lst_todo* and adds the cherry to *lst_tcs*. The function uses *check_cherry* to see if it's a pair and reduces it with *reduce_pair*. To continue with cherry picking, the function now looks for new pairs that are available after the pair has been reduced. In order to do this, the function adds the outputs of $find\_cherry(N, y)$ and $find\_ret\_cherry(N, y)$ to *lst_todo*. This process and its reasoning is covered in-depth in the proofs of correctness in section 5.

Lemma 11. The running time of *find_tcs* is $O(\#leaves + \#reticulations)$.

Proof. Creating the empty lists *lst_todo* and *lst_tcs* takes constant time. The first for-loop iterates through all nodes in the network. The function *find_cherry* used within this loop runs in a constant time. Extending the contents of *cherry*1 to the list *lst_todo* also takes constant time.

The while-loop runs until the list *lst_todo* is empty. Whenever a cherry or reticulated cherry is reduced in the network, the algorithm removes either a leaf or an edge leading into a reticulation node respectively. Since this function eventually reduces the entire network (theorem 3 in section 5.2, this means that the while-loop will run a number of times equal to the sum of the number of leaves and the number of reticulations.

The list is initially filled by using $find\_cherry$ on every leaf. This will find all initially available cherries (x, y), duplicate cherries (y, x) and reticulated cherries exactly once (lemma 13 in section 5.1). The functions $check\_cherry$, $reduce\_pair$ and $find\_cherry$ all run in constant time. Adding the cherry to the list lst_tcs takes constant time.

The duration of the function $find\_ret\_cherry$ depends on the indegree of the parent node. Since every reticulated cherry (x, y) is found once (theorem 3 in section 5.2) and the while-loop continues until all are reduced, the function $find\_ret\_cherry$ will be used once on every reticulation. Therefore it will iterate over every parent of every reticulation, which means that the total number of iterations done in $find\_ret\_cherry$ is equal to the number of reticulations in the network.

Combining this, the previous contents of the while-loop and the initial for-loop in the creation of the list lst_todo, the running time of the $find\_tcs$ function is $O(\#leaves + \#reticulations)$. □

Note that this result is independent of the maximum indegree of the network.

4.6 Modification

The old version of $find\_tcs$ that corresponds with the theoretical version of the algorithm created by Janssen and Murakami, featured additional code that I have removed to increase the speed of the function. An old version of this code can be seen here:

```
1  def find_tcs(N):
2      lst_todo = list()
3      for x in N.nodes():
4          if N.out_degree(x) == 0:
5              cherry1 = find_cherry(N,x)
6              lst_todo.extend(cherry1)
7      lst_tcs = list()
8      while lst_todo:
9          cherry = lst_todo.pop()
10         k = check_cherry(N, cherry)
11         if k == 1:
12             lst_tcs.append(cherry)
13             reduce_pair(N,cherry)
14             if (cherry[1],cherry[0]) in lst_todo:
15                 lst_todo.remove((cherry[1],cherry[0]))
16             lst_todo.extend(find_cherry(N,cherry[1]))
17             lst_todo.extend(find_ret_cherry(N,cherry[1]))
18         if k == 2:
19             lst_tcs.append(cherry)
20             reduce_pair(N, cherry)
21             lst_todo.extend(find_ret_cherry(N, cherry[0]))
22             lst_todo.extend(find_cherry(N, cherry[1]))
23             lst_todo.extend(find_ret_cherry(N, cherry[1]))
24     return lst_tcs
```

The differences between this old version and the new version are the removal of two actions:

1. In this old version, there exists a check (lines 14 and 15) that removes a cherry (y, x) from the list lst_todo if the cherry (x, y) is reduced. This line was intended to save time by removing a redundant cherry from lst_todo, since this cherry (y, x) couldn't exist in the network anymore after the cherry (x, y) is reduced. In the theoretical version of the algorithm created by Janssen and Murakami, this made sense since lst_todo was supposed to be a set instead of a list. Unfortunately, the *networkx* network classes objects are not hashable, and can therefore not be elements of a set. To solve this issue, lst_todo is a list

instead of a set in this python implementation. However, since searching through the list lst_todo takes $O(|lst\_todo|)$ time (whereas searching through a set is constant), this check did not speed up the algorithm. Worse still, since the size of lst_todo can be linear with the number of leaves and this check takes place in the while-loop, this check can cause a running time that is quadratic with the amount of leaves in the network.

2. In the old version, the case where the pair (x, y) popped from the list lst_todo is a reticulated cherry featured an extra line (Line 21) which added the output of $find\_ret\_cherry(N, x)$ to lst_todo. This line is redundant since the reticulated cherries found by $find\_ret\_cherry(N, x)$ are already elements in the lst_todo. The reasoning behind why these reticulated cherries are already found is explained in section 5, and the effects of this line on the speed of the algorithm are shown in section 6.3.

When these lines are removed, the cases for $k = 1$ and $k = 2$ become identical and can be merged into a single case.

4.7 cps_reduces_network

This function takes a network N and a cherry picking sequence in the form of a list lst as its input. It returns True if the cherry picking sequence reduces network N and False if it does not.

```
1  def cps_reduces_network(N, lst):
2      for cherry in lst:
3          reduce_pair(N, cherry)
4      if N.size() == 1:
5          return True
6      return False
```

The function iterates through the cherry picking sequence using a for-loop and each cherry in the list is reduced with $reduce\_pair$. After this loop is finished, the function checks whether the network is fully reduced by checking the network's size attribute. Note that this $N.size$ gives the number of edges in the network, which means that $N.size() = 1$ corresponds with one edge between the remaining leaf and the root.

Lemma 12. The running time of *cps_reduces_network* is $O(|lst|)$.

Proof. The function $reduce\_pair$ takes constant time. The size of the network is an attribute of the network class and acquiring it takes constant time. Since the for-loop iterates over the entire list lst, this function takes $O(|lst|)$ time. □

4.8 tcs_contains

This function takes a network N and a network M as its input and returns True if N contains M and False if N does not contain M.

```
1  def tcn_contains(N, M):
2      return cps_reduces_network(M, find_tcs(N))
```

This function uses $find\_tcs$ on network N and uses the resulting cherry picking sequence output and the given network M as input for $cps\_reduces\_network$. The output of $cps\_reduces\_network$ is the output of the function.

Theorem 2. The running time of *tcs_contains* (and therefore the algorithm) is $O(\#leaves + \#reticulations)$.

Proof. Since reducing a cherry removes a leaf and reducing a reticulated cherry removes an edge, the list given by $find\_tcs$ will have a length equal to the sum of leaves and reticulations of the network N. Since the function $cps\_reduces\_network$ takes $O(|lst|)$ time, the function $tcn\_contains(N, M)$ (and thus the algorithm) runs in $O(\#leaves + \#reticulations)$. □

5 Discovering pairs

This section serves to prove that the algorithm is indeed capable of reducing a network. For the algorithm to reduce the network, it is important that it finds all pairs available at any given moment. For the time complexity, it is important that the algorithm finds cherries a constant number of times.

5.1 Initial phase

In the initial phase (lines 2 - 6) of the function $find\_tcs$, the list lst_todo is filled with pairs by iterating over all leaves and using the $find\_cherry$ function.

Lemma 13. During the initial phase of $find\_tcs$, all initially available cherries are added to the list lst_todo twice (including cherries with coordinates in reverse order) and all initially available reticulated cherries are added to the list lst_todo once.

Proof. The function $find\_tcs$ finds all pairs with the given leaf as second coordinate once. Since it iterates over all leaves, it will find all initially available pairs at least once. However when reducing cherries in a network, there is no structural difference between reducing the cherry (x, y) or reducing the cherry (y, x) other than the resulting leaf. Therefore initial available cherries are effectively found twice, since the cherry with its coordinates in reverse order will be redundant in the list. For reticulated cherries, this issue doesn't occur since they are ordered. □

5.2 Second phase

After the initial phase, the algorithm searches for new pairs only after reducing an old pair. These new pairs must involve at least one of the leaves associated with the old pair, since the other leaves in the network are not affected when a pair is reduced.

Lemma 14. After a pair (x, y) has been reduced, the combination of functions $find\_cherry(N, y)$ and $find\_ret\_cherry(N, y)$ finds all new available pairs.

Proof. From lemma 5 we have that $find\_cherry(N, x)$ finds all pairs in N with x as their second coordinate. From lemma 6 we have that $find\_ret\_cherry(N, x)$ finds all reticulated cherries in N with x as their first coordinate. The next steps of the proof are divided into two cases:

1. **The reduced pair (x, y) was a cherry.** Since the reduced pair was a cherry (x, y), then the leaf x has been removed. This means that all potential new pairs could be either a cherry or reticulated cherries with y as their first or second coordinate. However, checking for both cherries with y as their first coordinate and y as their second coordinate is redundant, as explained in the proof of lemma 11. Therefore, the function finds all new available pairs (see figure 8) in this case using only the outputs of $find\_cherry(N, y)$ and $find\_ret\_cherry(N, y)$.

(a) Reticulated cherry (y, z) (b) Reticulated cherry (z, y) (c) Cherry (y, z) or (z, y)

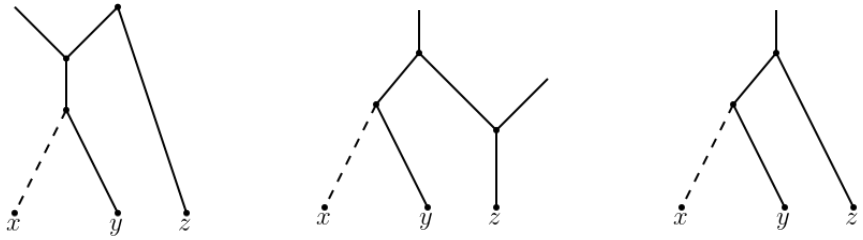


Figure 8: Possible new pairs if (x, y) is a cherry

2. **The reduced pair (x, y) was a reticulated cherry.** Since the reduced pair was a reticulated cherry, the edge between the parent p_x of x and the parent p_y of y has been removed. The parent p_y of y was a tree node (due to the tree-child restriction), and its parent pp_y of p_y can be either a reticulation node or a tree node. If pp_y is a tree node, the removal of the edge between p_y and the parent p_x of x can lead to a new cherry with coordinate y[1]. If pp_y is a reticulation node, the removal of the edge can lead to a new reticulated cherry with y as it's first or second coordinate (as seen in figure 9). Let p_x again be the parent of x. The possible pairs with x are divided into two cases:

   (a) **The indegree of p_x was larger than two prior to reducing (x, y).** If the parent px of x had an indegree of more than two prior to reducing the cherry, it remains a reticulation node when its edge to py is removed. This means that any reticulated cherries using x as their first coordinate will remain reticulated cherries in the network.

   If (x, y) was found during the initial phase, then the other reticulated cherries using x as a first coordinate have also been found by using $find\_cherry$ with their second leaf as input (since $find\_cherry$ has been used with every leaf as input during the initial phase).

   If (x, y) was found during the second phase, then it must have been found with the function $find\_ret\_cherry(N, x)$. However since $find\_ret\_cherry(N, x)$ finds all reticulated cherries with x as their first coordinate, these other possible reticulated cherries have already been found.

   (b) **The indegree of p_x was exactly two prior to reducing (x, y).** If p_x had indegree 2, it will have indegree 1 after the removal of the edge between p_y and p_x and the cleaning up process of $reduce\_pair$ will remove it entirely. Now x will be connected with an edge to the parent pp_x of p_x. Since p_x was a reticulation node in the original network, ppx cannot be a reticulation node because of the tree-child restriction. This means that x can now only be part of a cherry. If x is now part of a cherry (x, z) with z another leaf in the network, this means that x would have been part of a reticulated cherry (x, z) before reducing the cherry (x, y). Since both are written the same way, and (x, z) should be discovered if (x, y) has been discovered, it is not necessary to add any new cherries with coordinate x.

[1]This new cherry could potentially even be (x, y) if $pp_y = pp_x$, rendering it possible that a pair occurs multiple times in a cherry picking sequence.

(a) Reticulated cherry (y, z) (b) Reticulated cherry (z, y) (c) Cherry (y, z) or (z, y)

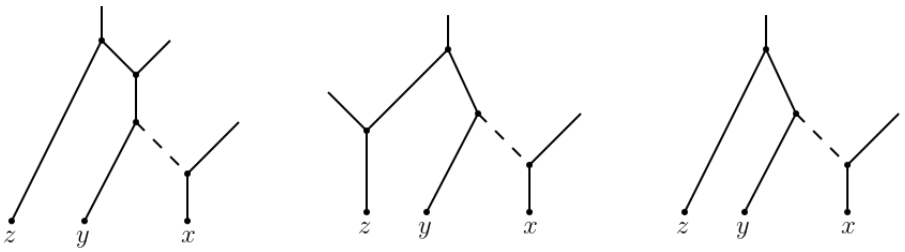


Figure 9: Possible new pairs if (x, y) is a reticulated cherry

□

Finding all cherries at least once is crucial for the correctness of the algorithm. However, for the running time of the algorithm, it is important that all cherries are found at most once. To show this, I have the following lemma:

Lemma 15. After a pair (x, y) has been reduced, the combination of functions $find\_cherry(N, y)$ and $find\_ret\_cherry(N, y)$ finds all new available pairs.

Proof. To prove this, I show that a cherry cannot be found multiple times during a single loop, and that a cherry cannot be found again in subsequent loops.

According to lemma 5 and lemma 6, the function $find\_cherry(N, y)$ finds all pairs with y as their second coordinate and $find\_ret\_cherry(N, y)$ finds all reticulated cherries with y as their first coordinate. These 2 functions will never find the same pair, since $find\_ret\_cherry(N, y)$ cannot find cherries and reticulated cherries are ordered (the reticulated cherries (x, y) and (y, x) cannot both exist).

After the initial phase, a pair cannot be found again if it has already been found. If the pair (x, y) is a cherry, neither x nor y can be part of another pair that contains a different leaf. That means that these leaves x and y can only be found after the cherry (x, y) is reduced. However the leaf x will be removed from the network when the cherry is reduced, rendering any rediscovery of the cherry (x, y) impossible.

If the pair (x, y) is a reticulated cherry, x can be part of a reticulated cherry (x, z). If this reticulated cherry (x, z) were to be reduced, the function would remove the edge between x and z and check for new pairs using $find\_cherry(N, z)$ and $find\_ret\_cherry(N, z)$ Since the function only looks for reticulated cherries with first or second coordinate z, it will never rediscover (x, y). □

Lemma 16. Every semi-binary tree-child network on taxa X with $|X| > 1$ has at least one cherry or reticulated cherry.

Proof. The idea behind this proof was created by Bordewich and Semple (Bordewich and Semple, 2007). If N is a tree, then N definitely has a cherry. If N is not a tree, N must have at least one reticulation. Let x be the reticulation node such that the path from the root to x is the longest (passes through the largest number of nodes). Since N is tree-child, both of the parents of x must be tree nodes. Note that the path from the root to x passes through one of these parents.

Let p be the parent of x through which this path travels. Since p is a tree node, let y be the other child of p that is not x.

Due to the tree-child restriction, y must either be a leaf or a tree node. We divide this into two separate cases:

1. ***y* is a leaf**. Let z be the child of x. Then since x is a reticulation, its child z must be either a leaf or a tree node due to the tree-child restriction. If z is a leaf, then the pair (z, y) is a reticulated cherry (see figure 10). If z is a tree node, then there will be two or more leaves reachable through z without passing through a reticulation (since z was the parent furthest away from the root), and at least one cherry will be formed out of these leaves.

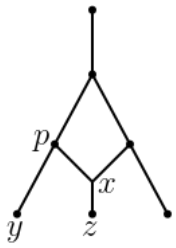


Figure 10: Example of a network where y is a leaf

2. ***y* is a tree node**. If y is a tree node, then there will be two or more leaves reachable through y without passing through a reticulation (again since z was the parent furthest away from the root), and at least one cherry will be formed out of these leaves.

□

Theorem 3. Let N be a semi-binary tree-child network. The function $find\_tcs(N)$ finds a tree-child sequence of N, while iterating through every initial cherry of N twice and while iterating through every other cherry or reticulated cherry of N once.

Proof. This follows from lemma 13, lemma 14, lemma 15 and lemma 16. □

6 Tests and results

This section covers several tests used to determine the linearity of the algorithm in practice.

6.1 Generating the random networks

For the data set used in the test functions, I have used a piece of code written by Remie Janssen that generates networks using tree-child sequences. I have modified this code slightly to generate a folder of 10000 test files, which each have two networks. The first network ranges from 10 to 1000 leaves and 10 to 1000 reticulations in steps of 10. This second network has the same amount of leaves (the same leaves as the first network) and half the reticulations of the first network, and has a 50% chance to be contained within the first network by using the same tree-child sequence

to generate it. If this second network does not pass this 50% chance, it will instead be generated randomly[2].

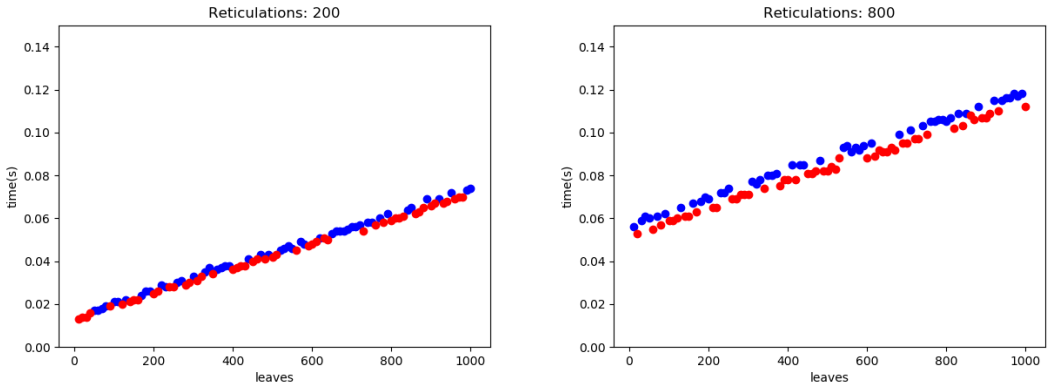
6.2 Plots

This large set of data files can be used to draw useful 2d plots that shows the running time of the algorithm, by either taking a constant number of leaves and letting the number of reticulations vary, or by taking a constant number of reticulations and letting the number of leaves vary. Figures 11(a) and 11(b) use a constant number of reticulations to show the linearity in the number of leaves, while figures 11(c) and 11(d) use a constant number of leaves to show the linearity in the number of reticulations. In these figures, a test case is shown with a blue dot if the algorithm returns True, and a test case is shown with a red dot if the algorithm returns False.

[2]This random network could theoretically be contained within the first network, but for large networks, the chance of this happening is extremely small.

(a) Constant 200 reticulations

(b) Constant 800 reticulations



(c) Constant 200 leaves

(d) Constant 800 leaves

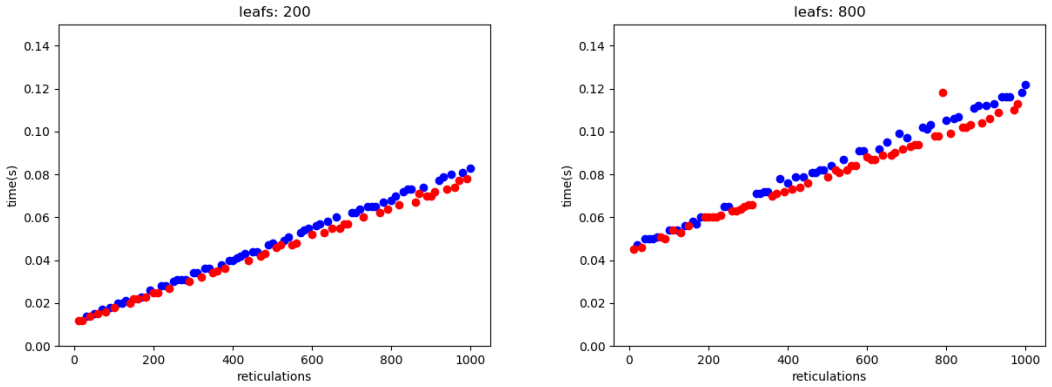
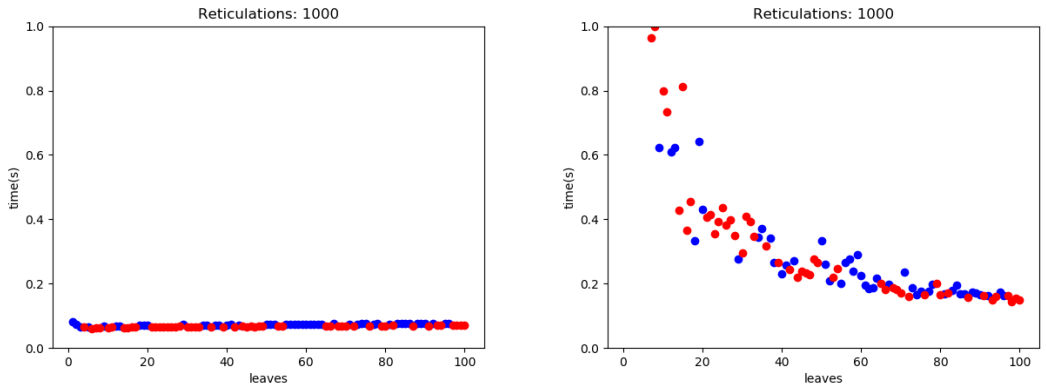


Figure 11: Plots showing the running time of the algorithm (in seconds) when tested on various sized networks.

6.3 Effects of the modification

Without the modification mentioned previously in Section 4.6, the algorithm will add the output of $find\_ret\_cherry$ (using the first coordinate of the reduced pair) to the list. As shown in section 5.2, this line is not necessary and will therefore cause the while-loop to iterate over redundant reticulated cherries. Initially, this may not seem like it would cause much trouble since these pairs will not pass the *check_cherry* check (as they can only be reduced once or twice) and therefore not waste too much time. However, these redundant reticulated cherries are added for each removed reticulation. For a reticulation node with n reticulated cherries, the function will initially remove a reticulation and add $n-1$ redundant reticulated cherries to the list. When another reticulated cherry involving the same reticulation node is processed, the function will remove another edge

leading into a reticulation node and add another $n-2$ redundant reticulated cherries[3]. Since this continues n times until there is are no reticulated cherries left in this reticulation node, this line will add $\sum_{i=1}^{n}(n-i) = n(n-1)/2$ redundant reticulated cherries to the list. The effect of this becomes apparent when testing the algorithm without the modification on a network with few leaves and many reticulations.



(a) With modification

(b) Without modification

Figure 12: The effects of the modification

For this test with a constant number of reticulations equal to 1000, the difference in running time for low numbers of leaves is quite large (see Figure 12). The running times of the first few networks of the algorithm without modification in figure 12 are outside of the window, and have running times ranging from one second to 2.5 seconds.

6.4 True or false

A clear pattern in the plots is the cases with output True taking more running time than the cases with output False. This occurs because the second network in the false cases is generated randomly and will generally differ a lot from the first network. Therefore, most of the pairs in the cherry picking sequence from the first network will not be cherries or reticulated cherries in the second network. When the *cps_reduces_network* function is used, most of the pairs in the cherry picking sequence will not pass the *check_cherry* test in the *reduce_pair* function. This means that a big part of *reduce_pair* is skipped, resulting in a shorter running time. Since this effect depends on the size of the networks, the difference in running time between true and false cases becomes more apparent as the number of reticulations and leaves in the network increase.

6.5 Linear regression test

With linear regression, the algorithm can be tested on its linearity. In this project, I have used the linear regression functions from the scikit-learn linear_model module. These functions predict a linear equation based on the given data, and can show the difference between the data and

[3]Since new pairs are added to the end of the list and popped from the end of the list, there can be no discovery of a reticulated cherry involving the reticulation node by reducing a pair elsewhere in the network, before other reticulated cherries involving the reticulation node are processed.

their prediction. This linear regression is used on the running time data received from using the algorithm on all 10000 test cases.

The coefficient of determination R^2 shows how well the prediction fits with the data. This coefficient generally ranges from 0 to 1, where 0 indicates a really poor fit and 1 indicates a perfect fit. The result of this test gives a R^2 value of 0.9917824501469797, which indicates that the predicted line fits the data very well. The tiny difference between this value and 1 is likely due to the random factors in the generation of the networks, and due to the difference in running time for True and False cases.

The linear regression test also gives an intercept value and a slope value, which together are used to make the predicted equation. The first variable of the slope is the number of leaves, and the second variable is the number of reticulations. According to the results of this test, the predicted equation is given by:

$$t = 5.9298 * 10^{-5} \#leaves + 7.0140 * 10^{-5} \#reticulations - 0.0009$$

The number of reticulations has a higher coefficient than the number of leaves, which is likely due to the extra steps required in the algorithm to deal with reticulated cherries instead of cherries.

Note that since the networks are generated fairly randomly and the speed of the algorithm depends on the computer, these tests will always give slightly differing results.

7 Conclusion and discussion

In this paper I have shown a python implementation of the algorithm created by Janssen and Murakami (Janssen, Murakami, 2019), which solves the NetworkContainment problem for semi binary tree-child networks. I have made a modification that increases its speed and removes its maximum indegree requirement. Furthermore I have proven that this modified algorithm works correctly and that its time complexity is linear, by using both theoretical proofs and practical tests. These practical tests show that not only the algorithm is linear, but its coefficients are small enough to allow the algorithm to solve test cases of large networks (800 reticulations and 800 leaves) in under 0.2 seconds using a ordinary computer.

The algorithm's speed could be improved further by implementing it in a faster programming language, however this wouldn't be too lucrative since its python implementation is already fast enough for its current day uses. What would be more interesting is looking at the theoretical side and changing the algorithm to work on different and more general types of trees and networks.

It would interesting to look at multifurcating trees, in which tree nodes can have more than two children. When dealing with these types of trees, a tree node with three or more children can also be represented by multiple nested tree nodes with two children. However, this gives rise to the problem that not all sequences that reduce the tree node with three or more children also reduce the nested tree nodes. One might wonder whether there's a fast way to find a sequence that reduces both networks, by looking at both networks.

Another interesting type of network is the more general cherry picking network. These can be reduced by cherry picking sequences, however, they don't necessarily contain a tree if there exists a cherry picking sequence that reduces both. The exact requirements for a cherry picking sequence to reduce both are still unproven.

Additionally, one might look at hardness proofs for TreeContainment for these cherry picking networks. For general networks, this problem is NP-complete but for tree child networks it is linear. The problem for cherry picking networks could be linear, NP-complete or fall somewhere in between, but where is unknown.

8 References

Bordewich, M. and C. Semple (2016). Determining phylogenetic networks from inter-taxa distances. *Journal of mathematical biology*, 73 (2), 283–303.

Janssen, R. and Murakami, Y. (2019). Solving Phylogenetic Network Containment Problems using Cherry-picking Sequences. arXiv preprint arXiv:1812.08065.

Linz, S. and Semple, C. (2017). Attaching leaves and picking cherries to characterise the hybridisation number for a set of phylogenies. Advances in Applied Mathematics, 105, 102-129.

van Iersel, L., C. Semple, and M. Steel (2010). Locating a tree in a phylogenetic network. *Information Processing Letters, 110* (23), 1037–1043.

9 Raw code

The algorithm and test functions:

```
import networkx as nx
import ast
import time
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

def find_cherry(N, x):
    lst = list()
    for p in N.predecessors(x):
        if N.in_degree(p) == 1:
            for pc in N.successors(p):
                if pc != x:
                    t = N.out_degree(pc)
                    if t == 0:
                        lst.append((pc, x))
                    if t == 1:
                        for pcc in N.successors(pc):
                            if N.out_degree(pcc) == 0:
                                lst.append((pcc,x))
    return lst


def find_ret_cherry(N, x):
    lst = list()
    for p in N.predecessors(x):
        if N.out_degree(p) == 1:
            for pp in N.predecessors(p):
                for ppc in N.successors(pp):
                    if ppc != p:
                        if N.out_degree(ppc) == 0:
                            lst.append((x, ppc))
    return lst

```

```
35
36  def check_cherry(N, (x, y)):
37      if N.has_node(x):
38          if N.has_node(y):
39              for px in N.predecessors(x):
40                  for py in N.predecessors(y):
41                      if px == py:
42                          return 1
43                      if N.out_degree(px) == 1:
44                          #if py in N.predecessors(px):
45                          if px in N.successors(py):
46                              return 2
47      return False
48
49
50  def reduce_pair(N, (x, y)):
51      k = check_cherry(N, (x, y))
52      if k == 1:
53          for px in N.predecessors(x):
54              N.remove_node(x)
55              for ppx in N.predecessors(px):
56                  N.remove_node(px)
57                  N.add_edge(ppx,y)
58              return True
59      if k == 2:
60          for px in N.predecessors(x):
61              for py in N.predecessors(y):
62                  N.remove_edge(py,px)
63                  if N.in_degree(px) == 1:
64                      for ppx in N.predecessors(px):
65                          N.add_edge(ppx, x)
66                          N.remove_node(px)
67                  #if N.out_degree(py) == 1:
68                  for ppy in N.predecessors(py):
69                      N.add_edge(ppy, y)
70                      N.remove_node(py)
71                  return True
72      return False
73
74  def find_tcs(N):
75      lst1 = list()
76      for x in N.nodes():
77          if N.out_degree(x) == 0:
78              cherry1 = find_cherry(N,x)
79              lst1.extend(cherry1)
80      lst2 = list()
81      while lst1:
82          cherry = lst1.pop()
83          k = check_cherry(N, cherry)
84          if (k == 1) or (k == 2):
85              reduce_pair(N, cherry)
86              lst2.append(cherry)
87              lst1.extend(find_cherry(N,cherry[1]))
88              lst1.extend(find_ret_cherry(N,cherry[1]))
89      return lst2
90
91
92  def cps_reduces_network(N, lst):
93      for cherry in lst:
94          reduce_pair(N, cherry)
95      if N.size() == 1:
96          return True
```

```
97      return False
98
99
100 def tcn_contains(N, M):
101     return cps_reduces_network(M, find_tcs(N))
102
103
104 def tester(min,max):
105     start = time.time()
106     for i in range(min, max+1):
107         if i < 10:
108             name = "input000" + str(i) + ".txt"
109         else:
110             if i > 99:
111                 name = "input0" + str(i) + ".txt"
112             else:
113                 name = "input00" + str(i) + ".txt"
114         test = open("SmallDataSet\\" + name, "r")
115         line1 = test.read()
116         line1 = line1.split("\n")
117         M = nx.DiGraph()
118         N = nx.DiGraph()
119         N.add_edges_from(ast.literal_eval(line1[0]))
120         M.add_edges_from(ast.literal_eval(line1[1]))
121         print name, ":", tcn_contains(N, M)
122     end = time.time()
123     print
124     print "time_elapsed:", end - start, "seconds"
125
126
127 def tester_2():
128     truthnumber = 0
129     lst1 = list()
130     lst2 = list()
131     for i in range(1, 101):
132         for j in range(1, 101):
133             if i > 0:           #if i > j:
134                 lst1.append((10*i,10*j))
135
136                 index1 = "0000000" + str(i)
137                 index1 = index1[-4:]
138                 index2 = "0000000" + str(j)
139                 index2 = index2[-4:]
140                 name = "input" + index1 + index2 + ".txt"
141
142                 test = open("DataSet\\" + name, "r")
143                 line1 = test.read()
144                 line1 = line1.split("\n")
145                 M = nx.DiGraph()
146                 N = nx.DiGraph()
147                 N.add_edges_from(ast.literal_eval(line1[0]))
148                 M.add_edges_from(ast.literal_eval(line1[1]))
149
150                 start = time.time()
151                 contains = tcn_contains(N, M)
152                 end = time.time()
153                 print name, ":", contains
154                 lst2.append(end - start)
155                 if contains == True:
156                     truthnumber = truthnumber + 1
157     model = LinearRegression().fit(lst1, lst2)
158     r_sq = model.score(lst1, lst2)
```

```
159     print('coefficient of determination:', r_sq)
160     print('intercept:', model.intercept_)
161     print('slope:', model.coef_)
162     print "number of Trues", truthnumber
163 
164 
165 def tester_3(j, check):
166     lst11 = list()
167     lst12 = list()
168     lst21 = list()
169     lst22 = list()
170     for i in range(1, 101):
171         index1 = "0000000" + str(i)
172         index1 = index1[-4:]
173         index2 = "0000000" + str(j)
174         index2 = index2[-4:]
175         name = "input" + index1 + index2 + ".txt"
176 
177         test = open("DataSet\\" + name, "r")
178         line1 = test.read()
179         line1 = line1.split("\n")
180         M = nx.DiGraph()
181         N = nx.DiGraph()
182         N.add_edges_from(ast.literal_eval(line1[0]))
183         M.add_edges_from(ast.literal_eval(line1[1]))
184 
185         start = time.time()
186         contains = tcn_contains(N, M)
187         print name, ":", contains
188         end = time.time()
189 
190         if contains is True:
191             lst11.append(10 * i)
192             lst21.append(end - start)
193         else:
194             lst12.append(10 * i)
195             lst22.append(end - start)
196 
197         if check is True:
198             M = nx.DiGraph()
199             N = nx.DiGraph()
200             N.add_edges_from(ast.literal_eval(line1[0]))
201             M.add_edges_from(ast.literal_eval(line1[1]))
202             indegree = check_maximum_indegree(N)
203             print "          indegree: " + str(indegree)
204 
205     plt.plot(lst11, lst21, 'bo')
206     plt.plot(lst12, lst22, 'ro')
207 
208     title = "Reticulations: " + str(10*j)
209     plt.title(title)
210     plt.xlabel('leaves')
211     plt.ylabel('time(s)')
212     plt.ylim((0, 0.15))
213     plt.show()
214 
215 
216 def tester_4(i, check):
217     truthnumber = 0
218     lst11 = list()
219     lst12 = list()
220     lst21 = list()
```

```
221     lst22 = list()
222     for j in range(1, 101):
223         index1 = "0000000" + str(i)
224         index1 = index1[-4:]
225         index2 = "0000000" + str(j)
226         index2 = index2[-4:]
227         name = "input" + index1 + index2 + ".txt"
228 
229         test = open("DataSet\\" + name, "r")
230         line1 = test.read()
231         line1 = line1.split("\n")
232         M = nx.DiGraph()
233         N = nx.DiGraph()
234         N.add_edges_from(ast.literal_eval(line1[0]))
235         M.add_edges_from(ast.literal_eval(line1[1]))
236 
237         start = time.time()
238         contains = tcn_contains(N, M)
239         print name, ":", contains
240         end = time.time()
241 
242         if contains is True:
243             lst11.append(10 * j)
244             lst21.append(end - start)
245             truthnumber = truthnumber + 1
246         else:
247             lst12.append(10 * j)
248             lst22.append(end - start)
249 
250         if check is True:
251             M = nx.DiGraph()
252             N = nx.DiGraph()
253             N.add_edges_from(ast.literal_eval(line1[0]))
254             M.add_edges_from(ast.literal_eval(line1[1]))
255             indegree = check_maximum_indegree(N)
256             print "_____indegree:_" + str(indegree)
257 
258     plt.plot(lst11, lst21, 'bo')
259     plt.plot(lst12, lst22, 'ro')
260     print "number_of_Trues:", truthnumber
261     title = "leafs:_" + str(10*i)
262     plt.title(title)
263     plt.xlabel('reticulations')
264     plt.ylabel('time(s)')
265     plt.ylim((0, 0.15))
266     plt.show()
267 
268 
269 
270 def check_maximum_indegree(N):
271     indegree = 0
272     for x in N.nodes():
273         x_indegree = N.in_degree(x)
274         if x_indegree > indegree:
275             indegree = x_indegree
276     return indegree
```

The code used to make network test files:

```
1 import networkx as nx
2 import random
```

```
3  import matplotlib.pyplot as plt
4  import os
5  import ast
6  import re
7  import time
8  from copy import deepcopy
9
10 def SeqToNewick(sequence):
11     return ""
12
13
14 def Newick_To_Network(newick):
15     newick = newick[:-1]
16     newick = newick.replace("(",",[")
17     newick = newick.replace(")",",]")
18     newick = re.sub(r"\]\#H([\d]+)", r",#R\1]", newick)
19     newick = re.sub(r"#([RH])([\d]+)", r"'#\1\2'", newick)
20     nestedtree = ast.literal_eval(newick)
21     edges, leaves, label_set, current_node = NestedList_To_Tree(nestedtree,1)
22     edges.append([0,1])
23     ret_labels = dict()
24     leaf_labels = dict()
25     for l in leaves:
26         if len(l)>2 and (l[:2]=="#H" or l[:2]=="#R"):
27             ret_labels[l[2:]]=[]
28         else:
29             leaf_labels[l]=[]
30     for l in label_set:
31         if len(l[0])>2 and (l[0][:2]=="#H" or l[0][:2]=="#R"):
32             if l[0][1]=='H':
33                 ret_labels[l[0][2:]]+=[l[1]]
34             else:
35                 ret_labels[l[0][2:]]=[l[1]]+ret_labels[l[0][2:]]
36         else:
37             leaf_labels[l[0]]+=[l[1]]
38     network = nx.DiGraph()
39     network.add_edges_from(edges)
40     for retic in ret_labels:
41         r = ret_labels[retic]
42         receiving = r[0]
43         parent_receiving = 0
44         for p in network.predecessors(receiving):
45             parent_receiving = p
46         network.remove_node(receiving)
47         for v in r[1:]:
48             network.add_edge(v,parent_receiving)
49             network = nx.contracted_edge(network,(v,parent_receiving))
50             network.remove_edge(v,v)
51             parent_receiving = v
52     leaves = set()
53     for l in leaf_labels:
54          leaf_labels[l]=leaf_labels[l][0]
55          leaves.add(l)
56     return network, leaves, leaf_labels
57
58
59 def NestedList_To_Tree(nestedList,next_node):
60     edges = []
61     leaves = set()
62     labels = []
63     top_node = next_node
64     current_node = next_node+1
```

```
65      for t in nestedList:
66          edges.append((top_node,current_node))
67          if type(t)==list:
68              extra_edges, extra_leaves, extra_labels, current_node = NestedList_To_Tree(t,current_nod
69          else:
70              extra_edges = []
71              extra_leaves = set([str(t)])
72              extra_labels = [[str(t), current_node]]
73              current_node+=1
74          edges = edges + extra_edges
75          leaves = leaves.union(extra_leaves)
76          labels = labels + extra_labels
77      return edges, leaves, labels, current_node
78
79
80
81
82  ##############################################################################
83  ##############################################################################
84  ##############################################################################
85  ########                                                          ############
86  ########                PHYLOGENETIC NETWORK CLASS                ############
87  ########                                                          ############
88  ##############################################################################
89  ##############################################################################
90  ##############################################################################
91
92
93
94  #A class for phylogenetic networks
95  class PhN:
96      def __init__(self, seq = None, newick = None):
97          #the actual graph
98          self.nw = nx.DiGraph()
99          #the set of leaf labels of the network
100         self.leaves = set()
101         #a dictionary giving the node for a given leaf label
102         self.labels = dict()
103         #the number of nodes in the graph
104         self.no_nodes = 0
105         self.leaf_nodes = dict()
106         self.TCS=seq
107         self.CPS=seq
108         self.newick=newick
109         self.reducible_pairs=set()
110         self.reticulated_cherries=set()
111         self.cherries=set()
112         if seq:
113             #Creates a phylogenetic network from a cherry picking sequence:
114             for pair in reversed(seq):
115                 self.add_pair(*pair)
116         if newick:
117             self.newick = newick
118             network, self.leaves, self.labels = Newick_To_Network(newick)
119             self.nw = network
120             self.no_nodes = len(list(self.nw))
121             self.Compute_Leaf_Nodes()
122
123     def Compute_Leaf_Nodes(self):
124         self.leaf_nodes = dict()
125         for v in self.labels:
126             self.leaf_nodes[self.labels[v]]=v
```

```
127
128
129         #A method for adding a pair, using the construction from a sequence
130         def add_pair(self,x,y):
131             if len(self.leaves)==0:
132                 self.nw.add_edges_from([(0,1),(1,2),(1,3)])
133                 self.leaves = set([x,y])
134                 self.labels[x]=2
135                 self.labels[y]=3
136                 self.leaf_nodes[2]=x
137                 self.leaf_nodes[3]=y
138                 self.no_nodes=4
139                 return True
140             if y not in self.leaves:
141                 return False
142             node_y=self.labels[y]
143             if x not in self.leaves:
144                 self.nw.add_edges_from([(node_y,self.no_nodes),(node_y,self.no_nodes+1)])
145                 self.leaves.add(x)
146                 self.leaf_nodes.pop(self.labels[y],False)
147                 self.labels[y]=self.no_nodes
148                 self.labels[x]=self.no_nodes+1
149                 self.leaf_nodes[self.no_nodes]=y
150                 self.leaf_nodes[self.no_nodes+1]=x
151                 self.no_nodes+=2
152             else:
153                 node_x=self.labels[x]
154                 for parent in self.nw.predecessors(node_x):
155                     px = parent
156                 if self.nw.in_degree(px)>1:
157                     self.nw.add_edges_from([(node_y,px),(node_y,self.no_nodes)])
158                     self.leaf_nodes.pop(self.labels[y],False)
159                     self.labels[y]=self.no_nodes
160                     self.leaf_nodes[self.no_nodes]=y
161                     self.no_nodes+=1
162                 else:
163                     self.nw.add_edges_from([(node_y,node_x),(node_y,self.no_nodes),(node_x,self.no_nodes
164                     self.leaf_nodes.pop(self.labels[x],False)
165                     self.leaf_nodes.pop(self.labels[y],False)
166                     self.labels[y]=self.no_nodes
167                     self.labels[x]=self.no_nodes+1
168                     self.leaf_nodes[self.no_nodes]=y
169                     self.leaf_nodes[self.no_nodes+1]=x
170                     self.no_nodes+=2
171             return True
172
173
174         #A method which plots the network
175         def show_network(self):
176             #Change colours (and shapes, does not work currently) of nodes
177             #green: root; red: leaf; blue: tree node; black: reticulation
178             color_map=[]
179             shape_map=[]
180             for v in list(self.nw):
181                 if self.nw.out_degree(v)==0:
182                     color_map.append('red')
183                 elif self.nw.in_degree(v)==0:
184                     color_map.append('green')
185                 elif self.nw.in_degree(v)>1:
186                     color_map.append('black')
187                     shape_map.append("s")
188                 else:
```

```
189                 color_map.append('blue')
190                 shape_map.append("o")
191         #Draw with algorithm that makes nice layout
192         nx.draw_kamada_kawai(self.nw, node_color=color_map)
193         plt.show()
194
195     def find_cps_method(self):
196         self.CPS = find_cps(self)
197
198     def Newick(self):
199         if not self.newick:
200             self.find_cps_method()
201             self.newick = SeqToNewick(self.CPS)
202         return self.newick
203
204
205
206
207
208 ##############################################################################
209 ##############################################################################
210 ##############################################################################
211 ########                                                          ############
212 ########                        RANDOM NETWORKS                   ############
213 ########                                                          ############
214 ##############################################################################
215 ##############################################################################
216 ##############################################################################
217
218
219
220
221
222
223
224 #A function that returns a tree-child sequence with a given number of leaves and reticulations
225 def random_TC_sequence(leaves, retics):
226     current_leaves = set([1,2])
227     seq = [(2,1)]
228     not_forbidden = set([2])
229     leaves_left = leaves-2
230     retics_left = retics
231
232     #Continue untill we added enough leaves and reticulations
233     while leaves_left > 0 or retics_left > 0:
234         #Decide if we add a leaf, or a reticulation
235         type_added='L'
236         if len(not_forbidden)>0 and leaves_left>0 and retics_left>0:
237             if random.randint(0 , leaves_left+retics_left-1)<retics_left:       #probability of retic
238 #               if random.randint(0 , 1)<1:                                      #probability of reti
239                 type_added='R'
240         elif len(not_forbidden)>0 and retics_left>0:
241             type_added='R'
242         elif leaves_left>0:
243             type_added='L'
244         else:
245             return(False)
246
247
248         #Actually add the pair
249         if type_added=='R':
250             first_element = random.choice(list(not_forbidden))
```

```
251             retics_left -=1
252         if type_added=='L':
253             first_element = len(current_leaves)+1
254             leaves_left -=1
255             current_leaves.add(first_element)
256             not_forbidden.add(first_element)
257 
258         second_element = random.choice(list(current_leaves-set([first_element])))
259         not_forbidden.discard(second_element)
260         seq.append((first_element,second_element))
261 
262     #reverse the sequence, as it was built in reverse order
263     seq=[pair for pair in reversed(seq)]
264     return(seq)
265 
266 
267 #A function that returns a tree-child subsequence with a given number of reticulations
268 def random_TC_subsequence(seq,r):
269     #First 'uniformly at random' choose one pair per leaf, with that leaf as first element
270     leaves  =  dict()
271     indices =  set()
272     for i, pair in enumerate(seq):
273         x=pair[0]
274         if x not in leaves:
275             indices.add(i)
276             leaves[x]=(1,i)
277         else:
278             if random.randint(0,leaves[x][0])<1:
279                 indices.remove(leaves[x][1])
280                 indices.add(i)
281                 leaves[x]=(leaves[x][0]+1,i)
282             else:
283                 leaves[x]=(leaves[x][0]+1,leaves[x][1])
284     #Add r reticulations with a max of the whole sequence
285     unused = set(range(len(seq)))-indices
286     for j in range(r):
287         new = random.choice(list(unused))
288         unused = unused-set([new])
289         indices.add(new)
290     newSeq=[]
291     for i, pair in enumerate(seq):
292         if i in indices:
293             newSeq.append(pair)
294     return newSeq
295 
296 
297 def make_random_files(repeats, failures, leaves, reticulations, folder_name, edges = False):
298     try:
299         os.mkdir("./"+folder_name+"/")
300     except:
301         pass
302     f= open("./"+folder_name+"/hits_and_misses.txt","w+")
303     f.write("index; subnetwork? \n")
304     f.close()
305 
306     #Make a list of indices we need to do
307     todo=list(range(1,repeats+1))
308     todo.reverse()
309     for i in range(repeats):
310         #pick a random index to fill with an instance
311         index = todo.pop()
312         #Add the right number of zeros to the index.
```

```
313             index = "0000000"+str(index)
314             index = index[-4:]
315             print(i)
316             print(index)
317             #Find random tree-child sequence
318             sequence = random_TC_sequence(leaves,reticulations)
319             #Decide if it fails or not
320             snw_or_not = 'yes'
321             if random.randint(0,repeats-i)<failures :
322                 failures-=1
323                 #Find some other random network, unrelated to the first
324                 subsequence = random_TC_sequence(leaves,reticulations)
325                 snw_or_not = 'no'
326             else:
327                 #Finds sebsequence corresponding to a network wit the same leaf set
328                 subsequence = random_TC_subsequence(sequence,reticulations)
329             f= open("./"+folder_name+"/input"+index+".txt","w+")
330             if edges:
331                 #build network from sequences
332                 network = PhN(seq = sequence)
333                 subnetwork = PhN(seq = subsequence)
334                 #now change the labels
335                 for node in network.leaf_nodes:
336                     network.leaf_nodes[node] = "L"+str(network.leaf_nodes[node])
337                 for node in subnetwork.leaf_nodes:
338                     subnetwork.leaf_nodes[node] = "L"+str(subnetwork.leaf_nodes[node])
339                 network.nw = nx.relabel_nodes(network.nw,network.leaf_nodes)
340                 subnetwork.nw = nx.relabel_nodes(subnetwork.nw,subnetwork.leaf_nodes)
341                 #Write edges to file
342                 #f.write(str(network.nw.edges)+"\r\n")
343                 f.write(str(network.nw.edges())+"\r\n")
344                 f.write(str(subnetwork.nw.edges()))
345             else:
346                 #Write newick to file
347                 f.write(SeqToNewick(sequence)+"\r\n")
348                 f.write(SeqToNewick(subsequence))
349             f.close()
350             #write answer to seperate file
351             f= open("./"+folder_name+"/hits_and_misses.txt","a+")
352             f.write(index+" ; "+snw_or_not+"\n")
353             f.close()
354
355
356     def make_a_lot_of_random_files(folder_name, edges = False):
357         try:
358             os.mkdir("./"+folder_name+"/")
359         except:
360             pass
361         f= open("./"+folder_name+"/hits_and_misses.txt","w+")
362         f.write("index; subnetwork? \n")
363         f.close()
364         start = time.time()
365
366         for leaves in range(1, 101):
367             for reticulations in range(1, 101):
368                 index1 = "0000000"+str(leaves)
369                 index1 = index1[-4:]
370                 index2 = "0000000"+str(reticulations)
371                 index2 = index2[-4:]
372                 index = index1 + index2
373
374                 print(index)
```

```
375             sequence = random_TC_sequence(10*leaves,10*reticulations)
376             snw_or_not = 'yes'
377             if random.randint(0,1) == 1:
378                 subsequence = random_TC_sequence(10*leaves,10*reticulations)
379                 snw_or_not = 'no'
380             else:
381                 subsequence = random_TC_subsequence(sequence,5*reticulations)
382             f= open("./"+folder_name+"/input"+index+".txt","w+")
383             if edges:
384                 #build network from sequences
385                 network = PhN(seq = sequence)
386                 subnetwork = PhN(seq = subsequence)
387                 #now change the labels
388                 for node in network.leaf_nodes:
389                     network.leaf_nodes[node] = "L"+str(network.leaf_nodes[node])
390                 for node in subnetwork.leaf_nodes:
391                     subnetwork.leaf_nodes[node] = "L"+str(subnetwork.leaf_nodes[node])
392                 network.nw = nx.relabel_nodes(network.nw,network.leaf_nodes)
393                 subnetwork.nw = nx.relabel_nodes(subnetwork.nw,subnetwork.leaf_nodes)
394                 #Write edges to file
395                 #f.write(str(network.nw.edges)+"\r\n")
396                 f.write(str(network.nw.edges())+"\r\n")
397                 f.write(str(subnetwork.nw.edges()))
398             else:
399                 #Write newick to file
400                 f.write(SeqToNewick(sequence)+"\r\n")
401                 f.write(SeqToNewick(subsequence))
402             f.close()
403             #write answer to seperate file
404             f= open("./"+folder_name+"/hits_and_misses.txt","a+")
405             f.write(index+" ; "+snw_or_not+"\n")
406             f.close()
407     end = time.time()
408     print "time elapsed:", end - start, "seconds"
```