Linear Time Algorithm for Tree-Child Network Containment

Janssen, Remie; Murakami, Yukihiro

DOI
10.1007/978-3-030-42266-0_8

Publication date
2020

Document Version
Accepted author manuscript

Published in
Algorithms for Computational Biology

Linear time algorithm for Tree-child Network Containment*

Remie Janssen[0000−0002−5192−1470] and Yukihiro Murakami[0000−0003−1355−5884]

Delft Institute of Applied Mathematics, Delft University of Technology, Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands {R.Janssen-2, Y.Murakami}@tudelft.nl

Abstract. Phylogenetic networks are used to represent evolutionary scenarios in biology and linguistics. To find the most probable scenario, it may be necessary to compare candidate networks, to distinguish different networks, and to see when one network is embedded in another. Here, we consider the Network Containment problem, which asks whether a given network is contained in another network. We give a linear-time algorithm to this problem for the class of tree-child networks using the recently introduced tree-child sequences by Linz and Semple. We implement this algorithm in Python and show that the linear-time theoretical bound on the input size is achievable in practice.

Keywords: Phylogenetics, Tree-child networks, Network containment, Tree-child sequences

1 Introduction

Phylogenetic networks are gaining popularity in the study of the evolutionary history of taxa [12, 1]. However, small stretches of DNA (e.g., pieces of DNA coding for protein domains) evolve tree-like. Therefore, the network representing the species' evolution must contain the trees for such pieces of DNA. This leads to the following mathematical problem. For a given network N and a tree T on the same set of taxa, decide whether N contains T.

This problem, called Tree Containment, is NP-complete for general rooted phylogenetic networks [10]. The problem remains NP-complete for certain network classes (networks with particular topological restrictions), such as tree-sibling, time-consistent, and regular networks [8]. However, for other network classes, the problem becomes easier. For example, it is known that Tree Containment can be solved in polynomial time for normal networks, tree-child networks, and level-k networks [8].

There are even stronger results for some network classes: deciding whether a tree is contained in a genetically stable network can be done in quadratic time [5], and making this decision for a binary nearly-stable network takes linear time [6]. For the class of tree-child networks, Tree Containment is known to be solvable in linear time [6, 7].

From a biological and a computational perspective, there is no reason why we should restrict ourselves to inputs of a tree and a network. Indeed, while small stretches of DNA may evolve tree-like, it is possible for another part of the genome to evolve as a network. In such instances, it is of great interest to consider a more general version of Tree Containment, which we call Network Containment: For given networks N and N' on the same set of taxa, decide whether N contains N'. By extension, the problem remains NP-complete for inputs of general rooted phylogenetic networks. Computationally, it is natural to wonder whether network classes that can solve Tree Containment efficiently can also solve Network Containment in a similar fashion. To date, no study has ever considered this problem, and we take the first steps in this endeavour.

We solve the Network Containment problem for tree-child networks (defined formally in Section 2) by considering *tree-child sequences*. These sequences were developed to tackle the problem of finding a "simple" network that contains a given set of trees [4, 11]. Two leaves of a tree form a *cherry* if they share a common parent—by successively *picking* cherries (removing one of the leaves in a cherry) from the set of input trees, we obtain a sequence of cherries that ultimately reduce each input tree to a tree on a single leaf. This sequence of cherries then corresponds to some network that contains the set of all input trees.

* Research funded by the Netherlands Organization for Scientific Research (NWO), with the Vidi grant 639.072.602.

Previously, these reductions were only defined on trees, and not on networks. In this paper, we start by defining tree-child sequences and their actions on tree-child networks. We show that for every tree-child network, there exists a sequence of ordered pairs of leaves that reduces it to a network on a single leaf. The order in which these pairs are picked does not matter. We also show that a tree-child network is contained in another tree-child network if and only if a sequence that reduces the first network also reduces the second one. Combining these results culminates in a linear-time algorithm for solving the Network Containment problem for tree-child networks.

Structure of the paper. We start by giving all relevant definitions and outlining how to construct networks from tree-child sequences (Section 2). In Section 3, we investigate properties of tree-child sequences, and their relation to tree-child networks. In particular, we focus on the relation between tree-child subsequences, and subnetworks of tree-child networks. This section also includes an algorithm to solve Tree-child Network Containment. Then, in Section 4, we present an efficient implementation of this algorithm in Python, and show that the theoretical running time is achievable in practice. We test our implementation on simulated data, and show that even for large data sets (1000 leaves and 1000 reticulations), the software outputs the solution within a tenth of a second. Lastly, in Section 5, we conclude with open problems and future directions for the use of cherry-picking strategies.

2 Preliminaries

Definition 1. *A* phylogenetic semi-binary network N *is a DAG with one outdegree-1 source* (the root)*, a set* $L(N)$ *of indegree-1 sinks* (leaves) *bijectively labelled with a set* X*, and all other nodes are either of indegree-1 and outdegree-2* (tree nodes) *or of indegree at least* 2 *and outdegree-*1 (reticulations).

A network is *binary* if all reticulations have indegree-2. In this paper, all networks we consider are phylogenetic semi-binary networks unless stated otherwise, so we call these networks for short. Furthermore, all networks have the leaf set $X = \{1, 2, \ldots, n\}$, unless stated otherwise.

An edge feeding into reticulations is called a *reticulation edge*. Given an edge uv in N, we say that u is a *parent* of v and that v is a *child* of u. The node u is *above* v if there is a directed path from u to v in N. The network N is *tree-child* if every non-leaf node in N is a parent of a tree node or a leaf. The *reticulation number* is the total number of reticulation edges minus the total number of reticulations.

Let N and N' be tree-child networks on the same set of taxa X. Then N *contains* N' if N' can be obtained from N by deleting reticulation edges and suppressing degree-2 nodes. We now formally define the Tree-child Network Containment problem.

Tree-child Network Containment
Instance: Two tree-child networks N and N' on the same leaf-set.
Question: Does N contain N'?

2.1 Reducible pairs

Let (x, y) be an ordered pair of leaves in a network N, and let p_x, p_y denote the parents of x, y respectively. We call (x, y) a *cherry* if $p_x = p_y$, if x and y share a common parent. Observe that if (x, y) is a cherry, then (y, x) must also be a cherry. We call (x, y) a *reticulated cherry* if p_x is a reticulation and p_y is a parent of p_x. If (x, y) is a cherry or a reticulated cherry in N, we call this a *reducible pair*. The following algorithms show that finding reducible pairs of a network can be done quickly. Observe that since tree nodes are of outdegree-2, each leaf appears as a second coordinate in at most one reducible pair in a network; Algorithm 1 finds such a reducible pair, if it exists, for a given leaf in constant time. Algorithm 2 on the other hand finds all reticulated cherries that contain a given leaf as the first coordinate of the reducible pair. The running time for this algorithm depends on the indegree of the parent of the given leaf, as this gives the maximum possible number of such reticulated cherries.

Algorithm 1: FindRP2nd(N, x)

Data: A network N and a leaf x
Result: The set containing exactly one reducible pair of N that has x as the second coordinate if it exists; $\emptyset$ otherwise.

Let p be the parent of x;
if *p is a tree node* **then**
  let $c(p)$ be the child of p that is not x;
  if *$c(p)$ is a leaf* **then**
    return $\{(c(p), x)\}$;
  if *$c(p)$ is a reticulation and the child $c(c(p))$ of $c(p)$ is a leaf* **then**
    return $\{(c(c(p)), x)\}$;
  end
end
return $\emptyset$;

Lemma 1. *Let x be a leaf in a network N. If a reducible pair with x as the second element of the pair exists, then Algorithm 1 finds this pair in constant time. Otherwise it returns the empty set in constant time.*

Algorithm 2: FindRC1st(N, x)

Data: A network N and a leaf x
Result: The set of all reticulated cherries in N that has x as the first coordinate

Let p be the parent of x;
Set $\mathcal{C}_r = \emptyset$;
if *p is a reticulation* **then**
  for *every parent g of p* **do**
    let $c(g)$ be the child of g that is not p;
    if *$c(g)$ is a leaf* **then**
      $\mathcal{C}_r = \mathcal{C}_r \cup \{(x, c(g))\}$
  end
end
return $\mathcal{C}_r$;

Lemma 2. *Let x be a leaf in a network N, and let p_x denote the parent of x. Let I denote the indegree of p_x. Algorithm 2 finds the set of all reticulated cherries that has x as the first coordinate in $O(I)$ time.*

2.2 Reducing pairs from networks

Given a cherry or a reticulated cherry, we may *reduce* them from a network to obtain a network of smaller size.

Definition 2. *Let N be a network and let (x, y) be an ordered pair of leaves.* Reducing *(x, y) in N is the action of*

- *deleting x and suppressing its parent node in N if (x, y) is a cherry in N;*
- *deleting the reticulation edge between the parents of x and y and subsequently suppressing degree-2 nodes, if (x, y) is a reticulated cherry;*
- *doing nothing to N otherwise.*

In all cases, the resulting network is denoted $N(x, y)$.

We refer to this as *picking a reducible pair (x, y) from N*. We transform this definition into an algorithm, and show that a reduction of a pair from a network can be done in constant time.

Lemma 3. *Algorithm 3 correctly reduces a given reducible pair in a network N in constant time.*

Algorithm 3: REDUCEPAIR$(N, (x, y))$

Data: A network N and a pair of leaves (x, y)
Result: The network $N(x, y)$
if (x, y) *is a cherry in* N **then**
  Let p be the parent of x and y;
  Remove edge px from N;
  Suppress p (if it is a node of degree-2) and remove x in N;
if (x, y) *is a reticulated cherry in* N **then**
  Let p_x be the parent of x and p_y the parent of y;
  Remove edge $p_y p_x$ from N;
  Suppress p_y and p_x (if they are nodes of degree-2) in N;
end
return N;

3 Tree-child sequence

In this section, we formally define *tree-child sequences* and how they correspond to tree-child networks. These are sequences of ordered pairs with additional properties; to illustrate the intuition behind these properties, we start by showing how to construct networks from sequences of ordered pairs.

Definition 3. *Let N be a network and let (x, y) be reducible pair. Then we may* construct *N from $N(x, y)$—also called* add *(x, y) to $N(x, y)$—by applying the following.*

1. *If x is a leaf in $N(x, y)$ (i.e., if (x, y) is a reticulated cherry in N), and*
   (a) *if p, the parent of x in $N(x, y)$, is a reticulation then add a node q directly above y, and add an edge qp.*
   (b) *otherwise, add nodes p and q directly above x and y respectively, and add an edge qp.*
2. *If x is not a leaf in $N(x, y)$ (i.e., if (x, y) is a cherry in N) then add a labelled node x, insert a node q directly above y, and add an edge qx.*

Observe that when adding (x, y) to $N(x, y)$, we assume that y is a leaf in the network $N(x, y)$. Otherwise, adding (x, y) to $N(x, y)$ is not well-defined.

Now let $S = S_1 S_2 \cdots S_{|S|} = (x_1, y_1)(x_2, y_2) \cdots (x_{|S|}, y_{|S|})$ be a sequence of ordered pairs with the condition that the second coordinate of each pair occurs as a first coordinate in the rest of the sequence, or as the second coordinate of the last pair. Starting with a network on a single leaf $y_{|S|}$, we may iteratively add S_i to the network for $i = |S|, |S| - 1, \ldots, 1$ (i.e., backwards through the sequence S) to obtain some network. We call this *the network obtained from S*. This condition ensures that when adding (x_i, y_i) to the network, y_i is already a leaf in the network so that the operation is well-defined.

Now suppose that we add a second condition on S that the first coordinate of each pair does not appear as a second coordinate of another pair in the remainder of the sequence. We will sometimes refer to this condition as the *tree-child condition*. Then, we claim that the network obtained from S is tree-child. By construction, we never obtain reticulation nodes that are adjacent to one another. In particular, every reticulation edge is inserted to existing reticulation nodes whenever possible (Definition 3.1a). Hence, we may only violate the tree-child property from a tree node having two reticulation children. So say that we have just added a reticulated cherry (x, y) to a network N. In N, the tree node parent p_y of y currently has one reticulation child and one leaf child y. For p_y to have two reticulation children, we require some reticulation node to be inserted between p_y and y, which can only happen if we add some ordered pair (y, z) to N. However, this would mean that y appears as a first coordinate of some pair and also as a second coordinate of some pair later on in the sequence, which contradicts our second condition. If, on the other hand, we have just added a cherry (x, y) to N, then the parent p of x cannot be a parent of two reticulations after adding more reducible pairs. Indeed, this would imply that we have added some reducible pair (y, z) later on to the network (and hence it would appear earlier in the sequence), which again contradicts our second condition.

This brings us to the following definition.

Definition 4. *A* tree-child sequence (TCS) *is a sequence of ordered pairs of two leaves such that*

- *the second coordinate of each pair occurs as a first coordinate in the rest of the sequence, or as the second coordinate of the last pair; and*
- *no first coordinate leaf is used as a second coordinate in the remainder of the sequence.*

Let N be a network and let S be a TCS. Denote by NS the network obtained by repeatedly reducing N with each element of S in order. We say that S *reduces* N if NS is a network with a single leaf (for any leaf in N), a root, and no other nodes. We call a TCS S *minimal* for a tree-child network N if S reduces N and if $NS_1 \cdots S_{i-1} \neq NS_1 \cdots S_i$ for all $i \in [|S|]$. Suppose that N contains n leaves and has reticulation number r. Then any minimal TCS for N is of length $n+r-1$.

Using the operations outlined in Definition 3, one may obtain a tree-child network N from a given TCS S. As each addition of an ordered pair creates either a cherry or a reticulated cherry in the network, we may simply reverse the operations to see that S reduces N. This brings us to the following correspondence.

Theorem 1. *Let N be a tree-child network. Then there exists a minimal TCS S that reduces it. The network obtained from S is isomorphic to N.*

Let S be a TCS. Then the network obtained from S is unique and is tree-child. Furthermore, S is a minimal TCS for this network.

While each TCS gives rise to a unique tree-child network, there can be many TCSs that reduce the same tree-child network. In particular, given a tree-child network, we may pick the reducible pairs in any order.

Theorem 2. *Let N be a tree-child network and let (x, y) be a reducible pair of N. Then there exists a minimal TCS of N whose first element is (x, y).*

In the setting of Theorem 2, we have that $N(x, y)$ is a tree-child network. Then, by iteratively applying the theorem to the reduced network each time, it is indeed the case that we may pick the reducible pairs in any order—making sure the second property of a TCS is not violated. The following algorithm then shows how we may obtain a minimal TCS for a tree-child network by picking reducible pairs in any order, and maintaining a list of all reducible pairs in the network.

Algorithm 4: FindTCS(N)

Data: A tree-child network N
Result: A minimal TCS S for N

Set $\mathcal{C} = \emptyset$;
for $x \in L(N)$ **do**
  $\mathcal{C} \cup$FindRP2nd(N, x);
end
Let S be an empty sequence;
while $\mathcal{C} \neq \emptyset$ **do**
  Choose $(x, y) \in \mathcal{C}$;
  Set $S = S(x, y)$;
  $N' =$ ReducePair$(N, (x, y))$;
  if (x, y) *is a cherry in* N **then**
    $\mathcal{C} = \mathcal{C} \setminus \{(x, y), (y, x)\} \cup$FindRP2nd$(N', y) \cup$FindRC1st$(N', y)$;
  if (x, y) *is a reticulated cherry in* N **then**
    $\mathcal{C} = \mathcal{C} \setminus \{(x, y)\} \cup$FindRP2nd$(N', y) \cup$FindRC1st$(N', y)$;
  end
  $N = N'$;
end
return S;

Lemma 4. *Let N be a tree-child network on X with reticulation number r. Algorithm 4 finds a minimal TCS for N in $O(n + r)$ time.*

3.1 Putting it all together

The following theorem characterizes when a tree-child network is contained in another tree-child network, using TCSs.

Theorem 3. *Let N and N' be two tree-child networks on the same leaf-sets. N contains N' if and only if any minimal TCS of N reduces N'.*

Therefore, using the subroutines that we have introduced previously (Algorithms 1 - 4), we obtain the following algorithm that solves the Tree-child Network Containment problem. Let N and N' be two tree-child networks on the same leaf-sets. Using Theorem 2, we first obtain some minimal sequence S that reduces N by picking reducible pairs in any order (Algorithm 4). By Theorem 3, if S reduces N', then N' is contained in N; otherwise, N' is not contained in N.

Algorithm 5: TCNContains(N, N')

```
Data: Two tree-child networks N and N' on the same set of taxa
Result: Yes if N contains N', No otherwise.
Set S =FindTCS(N);
for i = 1, ..., |S| do
    N' = ReducePair(N', S_i);
end
if N' is a network on a single leaf then
    return Yes;
end
return No;
```

Theorem 4. *Given two tree-child networks N and N' on the same taxa set X where the reticulation number of N is r, it can be decided in time $O(n + r)$ whether N' is contained in N.*

The theorem has the following corollary regarding the network Isomorphism problem, which asks whether two given networks are isomorphic. Indeed, we can solve this problem by running Algorithm 5 twice, since two networks are isomorphic if and only if they are contained in one another. The problem for tree-child networks was previously shown to be solvable in $O(n^2)$ time [3]. Therefore, we present the first linear-time algorithm for checking whether two tree-child networks are isomorphic.

Corollary 1. *Given two tree-child networks N and N' on taxa set X where the reticulation number of N is r, it can be decided in $O(n + r)$ time whether N is isomorphic to N'.*

4 Implementation

Algorithm 5, which checks whether a given tree-child network is a subnetwork of another, was implemented in Python to test the theoretical linear bound in practice. In this section, we present running time results of the implementation on a large randomly generated data set. We show that the theoretically proven linear running time is indeed achievable in practice. The tests were run on a Linux system with a quad-core Intel Xeon W3570 running at 1.7GHz and 24GB of DDR3 RAM clocked at 1333MHz. The operating system was Debian GNU/Linux 9 with a 4.19.46-64 Linux kernel. The software was written in Python version 3.7.3.

4.1 Generating the datasets

For the test data, we generated 131200 instances of the Tree-child Network Containment problem: two yes-instances and two no-instances for all $n, r, r' \in \{25, 50, \ldots, 975, 1000\}$ with $r' \leq r$, where n is the number of leaves of both networks, r is the reticulation number in the first network,

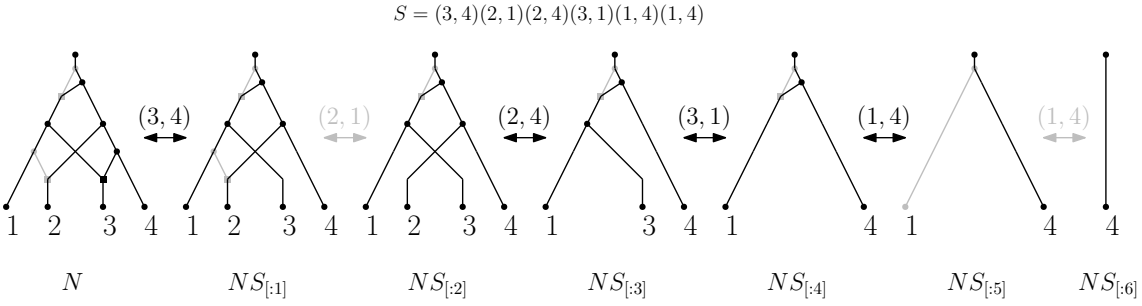

Fig. 1. A binary tree-child network N (grey and black) reduced to a leaf 4 by a tree-child sequence S. The reduction is shown as a sequence of networks $NS_{[:i]}$ for $i = 0, 1, \ldots, 6$ from left to right, in which an element of S is applied to the network successively. This sequence is minimal for the network, as every element of the sequence reduces either a cherry or a reticulated cherry of the network. An example of a cherry $(3, 1)$ can be seen in the network $NS_{[:3]}$, and a reticulated cherry $(3, 4)$ can be seen in the network N. The reduction of both reducible pairs is carried out as in Subsection 2.1. Observe that this sequence is a tree-child sequence. The black subnetwork is also reduced by S, and the embedding can be constructed by building both networks simultaneously and keeping track of the edges added by the pairs that change the subnetwork (black pairs and arrows).

and r' the reticulation number in the second network. Each instance consists of two semi-binary tree-child networks on the same leaf-set, for which we asked whether the first network contained the second network.

For each instance, we generated the first network with n leaves and reticulation number r using Algorithm 6. The second network was generated depending on whether it was a yes- or a no-instance. If it was a yes-instance, a subnetwork with reticulation number r' was obtained using Algorithm 7; for a no-instance, a network on the same number of leaves and reticulation number r' was randomly generated with the same process as the first network (using Algorithm 6).

This way, each generated yes-instance is always a yes-instance for the Tree-child Network Containment problem. For the no-instances, however, the random generation of the second network could also give a subnetwork of the first network, but the probability of that happening is very small, as the number of tree-child networks grows very quickly with the number of leaves and reticulations [2].

The dataset used for the experiment along with the code for generating random datasets, and the actual implementation of Algorithm 5 can be found on `https://github.com/RemieJanssen/Cherry-picking_TC_Network_Containment`.

Generating random networks The tree-child networks were randomly generated as TCSs using Algorithm 6. This algorithm takes two positive integers n and r, and outputs a tree-child network with n leaves and reticulation number r. It starts with the cherry $(1, 2)$, and successively adds leaves as cherries, and reticulated cherries between two leaves that already exist in the network (respecting the tree-child condition).

In the algorithm, this is achieved by building a tree-child sequence backwards. It chooses to add a reducible pair corresponding to a cherry or reticulated cherry uniformly at random until we have added the required number of leaves and reticulation number. To make sure the sequence is a tree-child sequence, we keep a list NF of taxa that are 'non-forbidden', which, in this case, means that the taxon is not currently the child of a tree node that has a reticulation as the other child (i.e., the leaf has not appeared as a second coordinate element of a reducible pair). If a taxon is in NF, it is possible to take this taxon as the first element of a pair appended at the start of the sequence. As a tree-child network always has a cherry or a reticulated cherry, NF is never empty. This implies that the algorithm should never output False, but lines 15 and 16 are kept so that the algorithm can easily be adapted to return only binary tree-child networks. To achieve this, one only has to add the line "$NF = NF \setminus \{\text{first\_element}\}$" between lines 21 and 22 in the algorithm. Finally, the algorithm outputs a TCS, from which we can uniquely construct a TCN.

Algorithm 6: RANDOMTCS(X, r)

Data: A set of taxa $X = \{1, \ldots, n\}$, and a reticulation number r.
Result: A TCS S on X of length $n + r - 1$.

Initialize $Y = \{1, 2\}$ the current set of taxa;
Initialize $S = (2, 1)$ the current sequence;
Initialize $L = n - 2$;
Initialize $R = r$;
Initialize $NF = \{2\}$;
while $L > 0$ *or* $R > 0$ **do**
    type_added = None;
    if $|NF| > 0$ *and* $L > 0$ *and* $R > 0$ **then**
        With probability $\frac{L}{L+R}$, type_added = L;
        Otherwise, type_added = R;
    else if $|NF| > 0$ *and* $R > 0$ **then**
        type_added = R;
    else if $L > 0$ **then**
        type_added = L;
    else
        return False;
    end
    first_element = None;
    second_element = None;
    if *type_added* $= R$ **then**
        Set first_element to an element of NF chosen uniformly at random;
        Set $R = R - 1$;
    else
        Set first_element to the first element of $X \setminus Y$;
        Set $L = L - 1$;
        Set $Y = Y \cup \{\text{first\_element}\}$;
        Set $NF = NF \cup \{\text{first\_element}\}$;
    end
    Set second_element to an element of $Y \setminus \{\text{first\_element}\}$ chosen uniformly at random;
    $NF = NF \setminus \{\text{second\_element}\}$;
    $S = (\text{first\_element}, \text{second\_element})S$;
end
return S;

Note that each tree-child network has positive probability of appearing for this process. In fact, each tree-child sequence ending with $(2, 1)$ has positive probability.

Let us now turn to the procedure to generate a tree-child subnetwork (i.e., generating the second network in a yes-instance). For this purpose, we again work with the representation of the networks as tree-child sequences.

We first select ordered pairs from the sequence of the first network, such that the resulting subsequence corresponds to a tree. This is simply done by selecting a pair with first element x for all $x \in X$ uniformly at random. Because the sequence we started with is a tree-child sequence, the subsequence consisting of the chosen pairs is a tree-child sequence as well: suppose (x, y) and (y, z) are selected. Then (y, z) must appear after (x, y), because otherwise y appears as a first element after it has appeared as a second element in the original sequence.

After selecting the pairs that form a *base tree* of the network (a spanning tree contained by the network), we select r' additional pairs that will form the r' reticulations of the subnetwork. By a similar argument as for the base tree, this subsequence is a tree-child sequence. And as it is reduced by the subsequence, it is also reduced by the sequence of the original network. Hence, the network corresponding to the chosen pairs is a tree-child subnetwork of the original network.

Algorithm 7: RANDOMSUBTCS(S, r')

Data: A TCS S on X of length $n + r - 1$, and a number $r' \leq r$.
Result: A sub-TCS S' of S on X of length $n + r' - 1$

Let S be indexed by $\{1, \ldots, |S|\}$;
Set $I_{S'} = \emptyset$;
for $x \in X$ **do**
    Let I_x be the set of indices of pairs of S with x as first element;
    Pick i_x uniformly at random from I_x;
    Set $I_{S'} = I_{S'} \cup \{i_x\}$;
end
Randomly add r' elements from $\{1, \ldots, |S|\} \setminus I_{S'}$ to $I_{S'}$;
Let S' be the subsequence of S consisting of the elements indexed by $I_{S'}$;
return S';

4.2 Results

For all yes-instance tests in which the second network was a subnetwork of the first (i.e., the ones generated by Algorithm 7), Algorithm 5 correctly returned a true value. Similarly, for all no-instance tests in which the second network was generated randomly and independently from the first network, Algorithm 5 correctly found that the second network was not a subnetwork of the first. This means that, even though there was a non-zero probability that the second network was a subnetwork, this did not happen in any of the instances. We expected this, as the probability of this happening is extremely small.

Note that the largest test instances (1000 leaves, 1000 reticulations) had a running time of approximately 0.1s. This is expected to scale well for even larger instances, as the linear fit of the data is very good. The R^2 values for the fits and the linear dependence of the running time on the number of leaves and reticulations can be found in Table 1. For this fit, we performed a standard linear regression with an intercept of 0 (i.e., forced through the origin), which makes sense because the running time should be zero for an empty instance.

Note that the fits become much better when we split the data in instances where the second network is or is not a subnetwork of the first (i.e., between the yes- and the no- instances), even though the dependence of the running time on the parameters does not change much after this split. The most striking difference we can see in this analysis, is the dependence on the reticulation number of the second network.

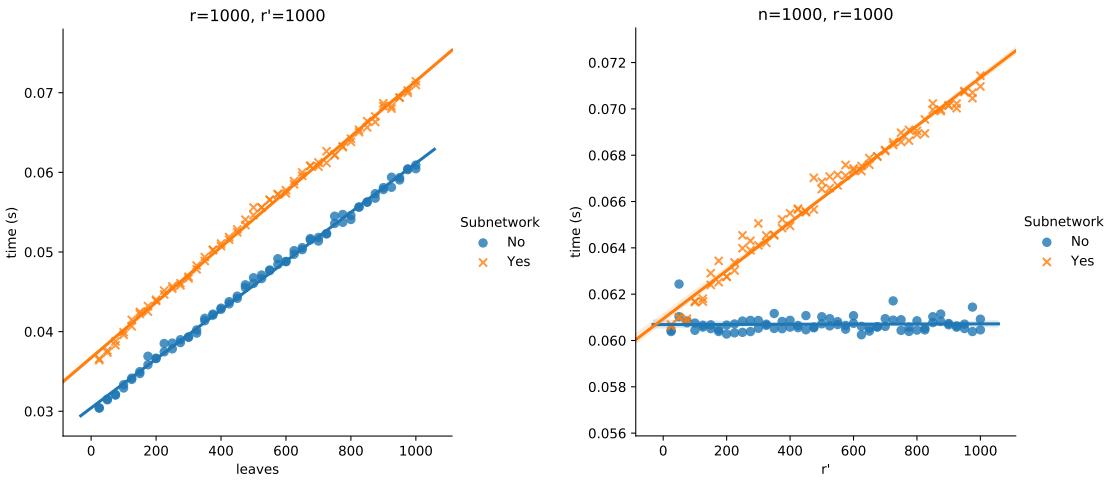

Fig. 2. The dependence of the running time on the number of leaves n (left) and the number of reticulations in the second network r' (right). This was visualized by fixing the other parameters to a set value of 1000 in both plots. Fitted lines are independent of Table 1.

As shown in Figure 2, the no-instances were consistently, and marginally, faster than the yes-instances. For varying leaf numbers, the instances where the second network was not a subnetwork (no-instances), were consistently, but marginally, faster than when the second network was a subnetwork (yes-instances) (Figure 2, Left). This was similarly true for when we varied the reticulation number r' of the second network. The effect of varying r' on instances for when the second network was not a subnetwork (no-instances) was negligible. This can be seen in the right figure of Figure 2, but also in Table 1, where the order of the slope of r' for the no-instances is far smaller than all other slopes in all the instances. For the yes-instances, the running time of the algorithm displayed a linear dependence on r', which was in the same order as the other parameters. This can be explained as follows. When the second network is not a subnetwork, Algorithm 5 will seldom need to reduce a pair in Line 3: it will check whether the pair in the sequence is reducible in the second network. As the second network is randomly generated independently of the first network, it will not have many pairs in common with the first network, which means it will not have to reduce pairs often.

Table 1. Linear regression analysis for tree-child network containment on 131200 instances, for which half were yes-instances and the other half no-instances. The high R^2 value indicates that the fit of the curve is essentially linear (where an R^2 value of 1 indicates a perfect linear fit) and the slopes indicate the change in running time for every increase in the number of leaves, reticulation number r, and reticulation number r'.

	R^2	slope		
		leaves (s/leaf)	r (s/reticulation)	r' (s/reticulation)
all data	0.9725659	$3.03328079 \cdot 10^{-5}$	$2.99713310 \cdot 10^{-5}$	$4.75681146 \cdot 10^{-6}$
subnetwork: YES	0.9966596	$3.14405850 \cdot 10^{-5}$	$2.89850496 \cdot 10^{-5}$	$9.54505907 \cdot 10^{-6}$
subnetwork: NO	0.9976078	$2.92250310 \cdot 10^{-5}$	$3.09576119 \cdot 10^{-5}$	$-3.14361016 \cdot 10^{-8}$

5 Discussion

In this paper, we have looked at tree-child sequences and how they can be used to solve the Network Containment problem for tree-child networks. A theoretical linear-time algorithm was given for this, and we have shown that our Python implementation also runs in linear time, in the number of leaves and the reticulation number.

In an effort to generalize our results, a natural question would be to ask what would happen if we weakened our current notion of a tree-child sequence. In Definition 4, we stated that the tree-child sequences must satisfy two conditions. The first condition ensures that each sequence corresponds to some network; the second condition ensures that the network is tree-child. Therefore we may consider networks that are more general than tree-child networks, by removing this second condition. These new sequences (which we call *cherry-picking sequences*) may be used to construct networks (called *cherry-picking networks*), with the operations as in Definition 3. This raises two questions. Do our Network Containment results hold when we consider inputs of cherry-picking networks? And, is there a structural characterization of these networks? To partly answer these questions for cherry-picking sequences, in [9], we have shown that a network can be reduced in any order (if it can be reduced at all). Furthermore, if a network is reduced by a minimal sequence for another network, the first is contained in the second. However, for cherry-picking sequences, it is no longer true that a subnetwork is reduced by any of the minimal sequences of the original network (Figure 3 of [9]). Therefore, Network Containment cannot be solved using cherry-picking sequences; in fact, the counter-example shows that even Tree Containment cannot be solved using cherry-picking sequences. The second question, about the structural characterization, remains open.

On a similar note, one can attempt to use tree-child sequences to solve a new problem related to Hybridization, where the input is a set of tree-child networks instead of trees. The problem aims to find a tree-child network with minimal reticulation number, containing all input networks. This problem has not been studied before, but could be very important, as there is a dire need of methods for finding a consensus network for a given set of networks.

As a follow-up, it would be interesting to extend our Network Containment results to a more general framework. In this paper, we have presented a linear time algorithm for checking whether a tree-child network contains another tree-child network on the same set of taxa. What is the change in complexity (if there is one) when we consider tree-child networks on different sets of taxa? Does the problem become NP-hard, or does it remain polynomial time? Could a modified version of our algorithm be used to solve this problem?

Acknowledgements We would like to thank Leo van Iersel for providing feedback on multiple versions of our paper. We would like to thank Robbert Huijsman for implementing and testing our pseudocode in Python. The final authenticated version is available online at `https://doi.org/10.1007/978-3-030-42266-0_8`.

References

1. Eric Bapteste, Leo van Iersel, Axel Janke, Scot Kelchner, Steven Kelk, James O McInerney, David A Morrison, Luay Nakhleh, Mike Steel, Leen Stougie, and James Whitfield. Networks: expanding evolutionary thinking. *Trends in Genetics*, 29(8):439–441, 2013.
2. Gabriel Cardona, Joan Carles Pons, and Celine Scornavacca. Generation of binary tree-child phylogenetic networks. *PLoS computational biology*, 15(9):e1007347, 2019.
3. Gabriel Cardona, Francesc Rossello, and Gabriel Valiente. Comparison of tree-child phylogenetic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(4):552–569, 2009.
4. Janosch Döcker, Leo van Iersel, Steven Kelk, and Simone Linz. Deciding the existence of a cherry-picking sequence is hard on two trees. *Discrete Applied Mathematics*, 260:131–143, 2019.
5. Philippe Gambette, Andreas DM Gunawan, Anthony Labarre, Stéphane Vialette, and Louxin Zhang. Solving the tree containment problem for genetically stable networks in quadratic time. In *International Workshop on Combinatorial Algorithms*, pages 197–208. Springer, 2015.
6. Philippe Gambette, Andreas DM Gunawan, Anthony Labarre, Stéphane Vialette, and Louxin Zhang. Solving the tree containment problem in linear time for nearly stable phylogenetic networks. *Discrete Applied Mathematics*, 246:62–79, 2018.
7. Andreas Gunawan. Solving tree containment problem for reticulation-visible networks with optimal running time. *arXiv preprint arXiv:1702.04088*, 2017.
8. Leo van Iersel, Charles Semple, and Mike Steel. Locating a tree in a phylogenetic network. *Information Processing Letters*, 110(23):1037–1043, 2010.
9. Remie Janssen and Yukihiro Murakami. Solving phylogenetic network containment problems using cherry-picking sequences. *arXiv preprint arXiv:1812.08065*, 2018.
10. Iyad A Kanj, Luay Nakhleh, Cuong Than, and Ge Xia. Seeing the trees and their branches in the network is hard. *Theoretical Computer Science*, 401(1-3):153–164, 2008.
11. Simone Linz and Charles Semple. Attaching leaves and picking cherries to characterise the hybridisation number for a set of phylogenies. *Advances in Applied Mathematics*, 105:102–129, 2019.
12. David A Morrison. Networks in phylogenetic analysis: new tools for population biology. *International journal for parasitology*, 35(5):567–582, 2005.