

RESEARCH

Open Access



Constructing phylogenetic networks via cherry picking and machine learning

Giulia Bernardini¹, Leo van Iersel², Esther Julien² and Leen Stougie^{3,4,5*}

Abstract

Background Combining a set of phylogenetic trees into a single phylogenetic network that explains all of them is a fundamental challenge in evolutionary studies. Existing methods are computationally expensive and can either handle only small numbers of phylogenetic trees or are limited to severely restricted classes of networks.

Results In this paper, we apply the recently-introduced theoretical framework of cherry picking to design a class of efficient heuristics that are guaranteed to produce a network containing each of the input trees, for practical-size datasets consisting of binary trees. Some of the heuristics in this framework are based on the design and training of a machine learning model that captures essential information on the structure of the input trees and guides the algorithms towards better solutions. We also propose simple and fast randomised heuristics that prove to be very effective when run multiple times.

Conclusions Unlike the existing exact methods, our heuristics are applicable to datasets of practical size, and the experimental study we conducted on both simulated and real data shows that these solutions are qualitatively good, always within some small constant factor from the optimum. Moreover, our machine-learned heuristics are one of the first applications of machine learning to phylogenetics and show its promise.

Keywords Phylogenetics, Hybridization, Cherry picking, Machine learning, Heuristic

Background

Phylogenetic networks describe the evolutionary relationships between different objects: for example, genes, genomes, or species. One of the first and most natural approaches to constructing phylogenetic networks is to build a network from a set of gene trees. In the absence of incomplete lineage sorting, the constructed network is naturally required to “display”, or embed, each of the gene trees. In addition, following the parsimony principle,

a network assuming a minimum number of reticulate evolutionary events (like hybridization or lateral gene transfer) is often sought. Unfortunately, the associated computational problem, called HYBRIDIZATION, is NP-hard even for two binary input trees [1], and indeed existing solution methods do not scale well with problem size.

For a long time, research on this topic was mostly restricted to inputs consisting of two trees. Proposed algorithms for multiple trees were either completely impractical or ran in reasonable time only for very small numbers of input trees. This situation changed drastically with the introduction of so-called cherry-picking sequences [2]. This theoretical setup opened the door to solving instances consisting of many input trees like most practical datasets have. Indeed, a recent paper showed that this technique can be used to solve instances with up to 100 input trees to optimality [3], although it was restricted to binary trees all having the same leaf set and

*Correspondence:

Leen Stougie
leen.stougie@cw.nl

¹ University of Trieste, Trieste, Italy

² Delft Institute of Applied Mathematics, Delft, The Netherlands

³ CWI, Amsterdam, The Netherlands

⁴ Vrije Universiteit, Amsterdam, The Netherlands

⁵ INRIA-ERABLE, Lyon, France



© The Author(s) 2023. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>. The Creative Commons Public Domain Dedication waiver (<http://creativecommons.org/publicdomain/zero/1.0/>) applies to the data made available in this article, unless otherwise stated in a credit line to the data.

to so-called “tree-child” networks. Moreover, its running time has a (strong) exponential dependence on the number of reticulate events.

In this paper, we show significant progress towards a fully practical method by developing a heuristic framework based on cherry picking comprising very fast randomised heuristics and other slower but more accurate heuristics guided by machine learning. Admittedly, our methods are not yet widely applicable since they are still restricted to binary trees. However, our set-up is made in such a way that it may be extendable to general trees.

Despite their limitations, we see our current methods already as a breakthrough as they are not restricted to tree-child networks and scale well with the number of trees, the number of taxa and the number of reticulations. In fact, we experimentally show that our heuristics can easily handle sets of 100 trees in a reasonable time: the slowest machine-learned method takes 4 min on average for sets consisting of 100 trees with 100 leaves each, while the faster, randomised heuristics already find feasible solutions in 2 s for the same instances. As the running time of the fastest heuristic depends at most quadratically on the number of input trees, linearly on the number of taxa, and linearly on the output number of reticulations, we expect it to be able to solve much larger instances still in a reasonable amount of time.

In addition, in contrast with the existing algorithms, our methods can be applied to trees with different leaf sets, although they have not been specifically optimized for this kind of input. Indeed, we experimentally assessed that our methods give qualitatively good results only when the leaf sets of the input trees have small differences in percentage (up to 5–15%); when the differences are larger, they return feasible solutions that are far from the optimum.

Some of the heuristics we present are among the first applications of machine learning in phylogenetics and show its promise. In particular, we show that crucial features of the networks generated in our simulation study can be identified with very high test accuracy (99.8%) purely based on the trees displayed by the networks.

It is important to note at this point that no method is able to reconstruct any specific network from displayed trees as networks are, in general, not uniquely determined by the trees they display [4]. In addition, in some applications, a phenomenon called “incomplete lineage sorting” can cause gene trees that are not displayed by the species network [5], and hence our methods, and other methods based on the HYBRIDIZATION problem, are not (directly) applicable to such data.

We focus on orchard networks (also called cherry picking networks), which are precisely those networks that

can be drawn as a tree with additional horizontal arcs [6]. Such horizontal arcs can for example correspond to lateral gene transfer (LGT), hybridization and recombination events. Orchard networks are broadly applicable: in particular, the orchard network class is much bigger than the class of tree-child networks, to which the most efficient existing methods are limited [7].

Related work. Previous practical algorithms for HYBRIDIZATION include PIRN [8], PIRNs [9] and Hybroscale [7], exact methods that are only applicable to (very) small numbers of trees and/or to trees that can be combined into a network with a (very) small reticulation number. Other methods such as *phylonet* [10] and *phylonetworks* [11] also construct networks from trees but have different premises and use completely different models.

The theoretical framework of cherry picking was introduced in [12] (for the restricted class of temporal networks) and [2] (for the class of tree-child networks) and was later turned into algorithms for reconstructing tree-child [3] and temporal [13] networks. These methods can handle instances containing many trees but do not scale well with the number of reticulations, due to an exponential dependence. The class of orchard networks, which is based on cherry picking, was introduced in [14] and independently (as cherry-picking networks) in [15], although their practical relevance as trees with added horizontal edges was only discovered later [6].

The applicability of machine-learning techniques to phylogenetic problems has not yet been fully explored, and to the best of our knowledge existing work is mainly limited to phylogenetic tree inference [16, 17] and to testing evolutionary hypotheses [18].

Our contributions. We introduce CHERRY PICKING HEURISTICS (CPH), a class of heuristics to combine a set of binary phylogenetic trees into a single binary phylogenetic network based on cherry picking. We define and analyse several heuristics in the CPH class, all of which are guaranteed to produce feasible solutions to HYBRIDIZATION and all of which can handle instances of practical size (we run experiments on tree sets of up to 100 trees with up to 100 leaves which were processed in on average 4 minutes by our slowest heuristic).

Two of the methods we propose are simple but effective randomised heuristics that proved to be extremely fast and to produce good solutions when run multiple times. The main contribution of this paper consists in a machine-learning model that potentially captures essential information about the structure of the input set of trees. We trained the model on different extensive sets of synthetically generated data and applied it to guide our algorithms towards better solutions. Experimentally, we show that the two machine-learned heuristics we design

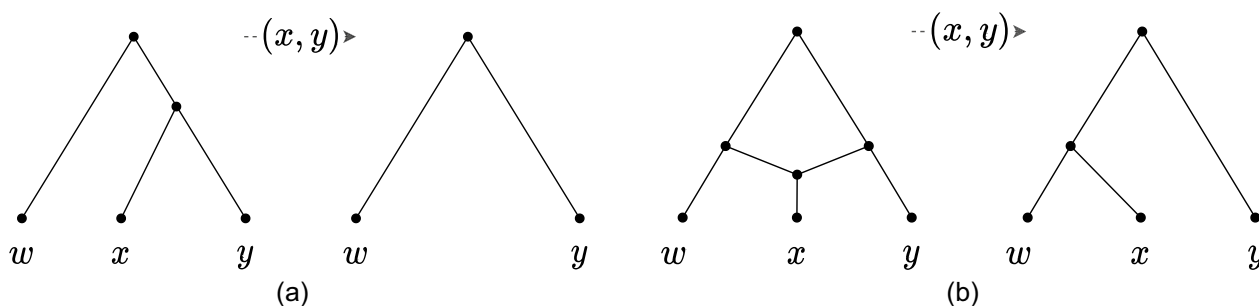


Fig. 1 (x, y) is picked in two different networks. In (a) (x, y) is a cherry, and in (b) (x, y) is a reticulated cherry. After picking, degree-two nodes are replaced by a single edge

yield good results when applied to both synthetically generated and real data.

We also analyse our machine-learning model to identify the most relevant features and design a non-learned heuristic that is guided by those features only. Our experiments show that this heuristic leads to reasonably good results without the need to train a model. This result is interesting per se as it is an example of how machine learning can be used to guide the design of classical algorithms, which are not biased towards certain training data.

A preliminary version of this work appeared in [19]. Compared to the preliminary version, we have added the following material: (i), we defined a new non-learned heuristic based on important features and experimentally tested it (Sect. "A non-learned heuristic based on important features"); (ii), we extended the experimental study to data generated from non-orchard networks (Sect. "Experiments on ZODS data"), data generated from a class of networks for which the optimum number of reticulations is known (Sect. "Experiments on normal data") and to input trees with different leaf sets (Sect. "Experiments on non-exhaustive input trees"); and (iii), we provided a formal analysis of the time complexity of all our methods (Sect. "Time complexity") and conducted experiments on their scalability (Sect. "Experiments on scalability").

Preliminaries

A phylogenetic network $N = (V, E, X)$ on a set of taxa X is a directed acyclic graph (V, E) with a single root with in-degree 0 and out-degree 1, and the other nodes with either (i) in-degree 1 and out-degree $k > 1$ (tree nodes); (ii) in-degree $k > 1$ and out-degree 1 (reticulations); or (iii) in-degree 1 and out-degree 0 (leaves). The leaves of N are biunivocally labelled by X . A surjective map $\ell : E \rightarrow \mathbb{R}^{\geq 0}$ may assign a nonnegative branch length to each edge of N . We will denote by $[1, n]$ the set of integers $\{1, 2, \dots, n\}$. Throughout this paper, we will only

consider binary networks (with $k = 2$), and we will identify the leaves with their labels. We will also often drop the term "phylogenetic", as all the networks considered in this paper are phylogenetic networks. The reticulation number $r(N)$ of a network N is $\sum_{v \in V} \max(0, d^-(v) - 1)$, where $d^-(v)$ is the in-degree of v . A network T with $r(T) = 0$ is a phylogenetic tree. It is easy to verify that binary networks with $r(N)$ reticulations have $|X| + r(N) - 1$ tree nodes.

Cherry-picking. We denote by \mathcal{N} a set of networks and by \mathcal{T} a set of trees. An ordered pair of leaves (x, y) , $x \neq y$, is a cherry in a network if x and y have the same parent; (x, y) is a reticulated cherry if the parent $p(x)$ of x is a reticulation, and $p(y)$ is a tree node and a parent of $p(x)$ (see Fig. 1). A pair is reducible if it is either a cherry or a reticulated cherry. Notice that trees have cherries but no reticulated cherries.

Reducing (or picking) a cherry (x, y) in a network N (or in a tree) is the action of deleting x and replacing the two edges $(p(p(x)), p(x))$ and $(p(x), y)$ with a single edge $(p(p(x)), y)$ (see Figure 1a). If N has branch lengths, the length of the new edge is $\ell(p(p(x)), y) = \ell(p(p(x)), p(x)) + \ell(p(x), y)$. A reticulated cherry (x, y) is reduced (picked) by deleting the edge $(p(y), p(x))$ and replacing the other edge $(z, p(x))$ incoming to $p(x)$, and the consecutive edge $(p(x), x)$, with a single edge (z, x) . The length of the new edge is $\ell(z, x) = \ell(z, p(x)) + \ell(p(x), x)$ (if N has branch lengths). Reducing a non-reducible pair has no effect on N . In all cases, the resulting network is denoted by $N_{(x,y)}$: we say that (x, y) affects N if $N \neq N_{(x,y)}$.

Any sequence $S = (x_1, y_1), \dots, (x_n, y_n)$ of ordered leaf pairs, with $x_i \neq y_i$ for all i , is a partial cherry-picking sequence; S is a cherry-picking sequence (CPS) if, for each $i < n$, $y_i \in \{x_{i+1}, \dots, x_n, y_n\}$. Given a network N and a (partial) CPS S , we denote by N_S the network obtained by reducing in N each element of S , in order. We denote $S \circ (x, y)$ the sequence obtained by appending pair (x, y) at the end of S . We say that S fully reduces N if N_S

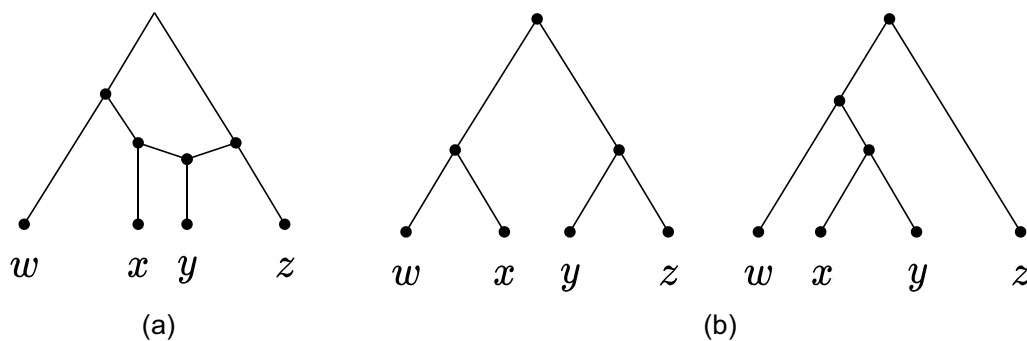


Fig. 2 The two trees in (b) are displayed in the network (a)

consists of the root with a single leaf. N is an orchard network (ON) if there exists a CPS that fully reduces it, and it is tree-child if every non-leaf node has at least one child that is a tree node or a leaf. A normal network is a tree-child network such that, in addition, the two parents of a reticulation are always incomparable, i.e., one is not a descendant of the other. If S fully reduces all $N \in \mathcal{N}$, we say that S fully reduces \mathcal{N} . In particular, in this paper we will be interested in CPS which fully reduce a set of trees \mathcal{T} consisting of $|\mathcal{T}|$ trees of total size $||\mathcal{T}||$.

Hybridization. The Hybridization problem can be thought of as the computational problem of combining a set of phylogenetic trees into a network with the smallest possible reticulation number, that is, to find a network that displays each of the input trees in the sense specified by Definition 1, below. See Fig. 2 for an example. The definition describes not only what it means to display a tree but also to display another network, which will be useful later.

Definition 1 Let $N = (V, E, X)$ and $N' = (V', E', X')$ be networks on the sets of taxa X and $X' \subseteq X$, respectively. The network N' is displayed in N if there is an embedding of N' in N : an injective map of the nodes of N' to the nodes of N , and of the edges of N' to edge-disjoint paths of N , such that the mapping of the edges respects the mapping of the nodes, and the mapping of the nodes respects the labelling of the leaves.

We call exhaustive a tree displayed in $N = (V, E, X)$ with the whole X as a leaf set. Note that Definition 1 only involves the topologies of the networks, disregarding possible branch lengths. In the following problem definition, the input trees may or may not have branch lengths, and the output is a network without branch lengths. We allow branch lengths for the input because they will be

useful for the machine-learned heuristics of Sect. "Predicting good cherries via machine learning".

HYBRIDIZATION

Input: A set of phylogenetic trees \mathcal{T} on a set of taxa X .

Output: A network displaying \mathcal{T} with minimum possible reticulation number.

Solving the hybridization problem via cherry-picking sequences

We will develop heuristics for the Hybridization problem using cherry-picking sequences that fully reduce the input trees, leveraging the following result by Janssen and Murakami.

Theorem 1 ([15]), *Theorem 3* Let N be a binary orchard network, and N' a (not necessarily binary) orchard network on sets of taxa X and $X' \subseteq X$, respectively. If a minimum-length CPS S that fully reduces N also fully reduces N' , then N' is displayed in N .

Notice that HYBRIDIZATION remains NP-hard for binary orchard networks. For binary networks we have the following lemma, a special case of [15, Lemma 1].

Lemma 1 Let N be a binary network, and let (x, y) be a reducible pair of N . Then reducing (x, y) and then adding it back to $N_{(x,y)}$ results in N .

Note that Lemma 1 only holds for binary networks: in fact, there are different ways to add a pair to a non-binary network, thus the lemma does not hold unless a specific rule for adding pairs is specified (inspect [15] for details). Theorem 1 and Lemma 1 provide the following approach for finding a feasible solution to HYBRIDIZATION: find

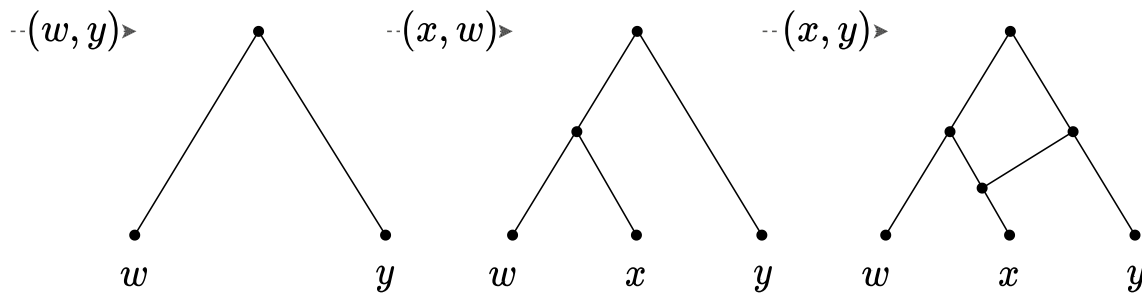


Fig. 3 The ON reconstructed from the sequence $S = (x, y), (x, w), (w, y)$. The pairs are added to the network in reverse order: if the first element of a pair is not yet in the network, it is added as a cherry with the second element (see the pair (x, w)). Otherwise, a reticulation is added above the first element with an incoming edge from a new parent of the second element (see the pair (x, y))

a CPS S that fully reduces all the input trees, and then uniquely reconstruct the binary orchard network N for which S is a minimum-length CPS, by processing S in the reverse order. N can be reconstructed from S using one of the methods underlying Lemma 1 proposed in the literature, e.g., in [15] (illustrated in Fig. 3) or in [3]. The following lemma relates the length of a CPS S and the number of reticulations of the network constructed from S .

Lemma 2 ([20]) *Let S be a CPS on a set of taxa X . The number of reticulations of the network N reconstructed from S is $r(N) = |S| - |X| + 1$.*

In the next section we focus on the first part of the heuristic: producing a CPS that fully reduces a given set of phylogenetic trees.

Randomised heuristics

We define a class of randomised heuristics that construct a CPS by picking one reducible pair of the input set \mathcal{T} at a time and by appending this pair to a growing partial sequence, as described in Algorithm 1 (the two subroutines *PickNext* and *CompleteSeq* will be later described in details). We call this class CPH (for Cherry-Picking Heuristics). Recall that \mathcal{T}_S denotes the set of trees \mathcal{T} after reducing all trees with a (partial) CPS S . The while loop at lines 2–5 produces, in general, a partial CPS S , as shown in Example 1. To make it into a CPS, the subroutine *CompleteSeq* at line 6 appends at the end of S

a sequence S' of pairs such that each second element in a pair of $S \circ S'$ is a first element in a later pair (except for the last one), as required by the definition of CPS. These additional pairs do not affect the trees in \mathcal{T} , which are already fully reduced by S . Algorithm 2 describes a procedure *CompleteSeq* that runs in time linear in the length of S .

Algorithm 1 CPH

INPUT: A set \mathcal{T} of phylogenetic trees

OUTPUT: A CPS reducing \mathcal{T} .

```

1:  $S \leftarrow \emptyset$ ;
2: while there is a reducible pair in  $\mathcal{T}_S$  do
3:    $(x, y) \leftarrow \text{PickNext}(\mathcal{T}_S)$ ;
4:    $S \leftarrow S \circ (x, y)$ ;
5:   Reduce  $(x, y)$  in all trees of  $\mathcal{T}_S$ ;
6:  $S \leftarrow \text{CompleteSeq}(S)$ ;
7: return  $S$ ;
```

Example 1 Let \mathcal{T} consist of the 2-leaf trees (x, y) and (w, z) . A partial CPS at the end of the while loop in Algorithm 1 could be, e.g., $S = (x, y), (w, z)$. The trees are both reduced to one leaf, so there are no more reducible pairs, but S is not a CPS. To make it into a CPS either pair (y, z) or pair (z, y) can be appended: e.g., $S \circ (y, z) = (x, y), (w, z), (y, z)$ is a CPS, and it still fully reduces the two input trees.

Algorithm 2 CompleteSeq

INPUT: A partial CPS $S = (x_1, y_1), \dots, (x_n, y_n)$ that reduces \mathcal{T}
OUTPUT: A CPS S' for \mathcal{T} .

$C \leftarrow \emptyset; P \leftarrow \emptyset;$
for $i = n, \dots, 1$ **do**
 if $y_i \notin C$ **then**
 $P \leftarrow P \cup \{y_i\};$
 $C \leftarrow C \cup \{x_i, y_i\};$
 $S' \leftarrow S;$
while $|P| > 1$ **do**
 Let r_1 and r_2 be two arbitrary elements of P ;
 $S' \leftarrow S' \circ (r_1, r_2);$
 $P \leftarrow P \setminus \{r_1\};$
return S'

The class of heuristics given by Algorithm 1 is concretised in different heuristics depending on the function `PickNext` at line 3 used to choose a reducible pair at each iteration. To formulate them we need to introduce the following notions of height pair and trivial pair. Let N be a network with branch lengths and let (x, y) be a reducible pair in N . The height pair of (x, y) in N is a pair $(h_x^N, h_y^N) \in \mathbb{R}_{\geq 0}^2$, where $h_x^N = \ell(p(x), x)$ and $h_y^N = \ell(p(y), y)$ if (x, y) is a cherry (indeed, in this case, $p(x) = p(y)$); $h_x^N = \ell(p(y), p(x)) + \ell(p(x), x)$ and $h_y^N = \ell(p(y), y)$ if (x, y) is a reticulated cherry. The *height* $h_{(x,y)}^N$ of (x, y) is the average $(h_x^N + h_y^N)/2$ of h_x^N and h_y^N . Let \mathcal{T} be a set of trees whose leaf sets are subsets of a set of taxa X . An ordered leaf pair (x, y) is a trivial pair of \mathcal{T} if it is reducible in all $T \in \mathcal{T}$ that contain both x and y , and there is at least one tree in which it is reducible. We define the following three heuristics in the CPH class, resulting from as many possible implementations of `PickNext`.

Rand	Function <code>PickNext</code> picks uniformly at random a reducible pair of \mathcal{T}_S
LowPair	Function <code>PickNext</code> picks a reducible pair (x, y) with the lowest average of values $h_{(x,y)}^T$ over all $T \in \mathcal{T}_S$ in which (x, y) is reducible (ties are broken randomly)
TrivialRand	Function <code>PickNext</code> picks a trivial pair if there exists one and otherwise picks a reducible pair of \mathcal{T}_S uniformly at random

Theorem 2 *Algorithm 1 computes a CPS that fully reduces \mathcal{T} , for any function `PickNext` that picks, in each iteration, a reducible pair of \mathcal{T}_S .*

Proof The sequence S is initiated as an empty sequence. Then, each iteration of the while loop (lines 2–5) of Algorithm 1 appends one pair to S that is reducible in at least one of the trees in \mathcal{T} , and reduces it in all trees. Hence, in each iteration, the total size of \mathcal{T}_S is reduced, so the algorithm finishes in finite time. Moreover, at the end of the while loop, each tree in \mathcal{T}_S is reduced, thus the partial CPS S reduces \mathcal{T}_S . As `CompleteSeq` only appends pairs at the end of S , the result of this subroutine still reduces all trees in \mathcal{T}_S . \square

In Sect. “Experiments” we experimentally show that `TrivialRand` produces the best results among the proposed randomised heuristics. In the next section, we introduce a further heuristic step for `TrivialRand` which improves the output quality.

Improving heuristic `TrivialRand` via tree expansion

Let \mathcal{T} be a set of trees whose leaf sets are subsets of a set of taxa X , let S be a partial CPS for \mathcal{T} and let \mathcal{T}_S be the tree set obtained by reducing in order the pairs of S in \mathcal{T} . With respect to a trivial pair (x, y) , each tree $T \in \mathcal{T}_S$ is of one of the following types: (i) (x, y) is reducible in T ; or (ii) neither x nor y are leaves of T ; or (iii) y is a leaf of T but x is not; or (iv) x is a leaf of T but y is not.

Suppose that at some iteration of `TrivialRand`, the subroutine `PickNext` returns the trivial pair (x, y) . Then, before reducing (x, y) in all trees, we do the following extra step: for each tree of type (iv), replace leaf x with cherry (x, y) . We call this operation the tree expansion: see Fig. 4c. The effect of this step is that, after reducing (x, y) , leaf x disappears from the set of trees, which would have not necessarily been the case before, because of trees of type (iv). Tree expansion followed by the reduction of (x, y) can, alternatively, be seen as relabelling leaf x in any tree of type (iv) by y . The choice of describing this relabelling as tree expansion is just for the purpose of proving Lemma 3.

To guarantee that a CPS S produced with tree expansion implies a feasible solution for `HYBRIDIZATION`, we must show that the network N reconstructed from S displays all the trees in the input set \mathcal{T} . We prove that indeed this is the case with the following steps: (1), we consider the networks N_T obtained by “reverting” a

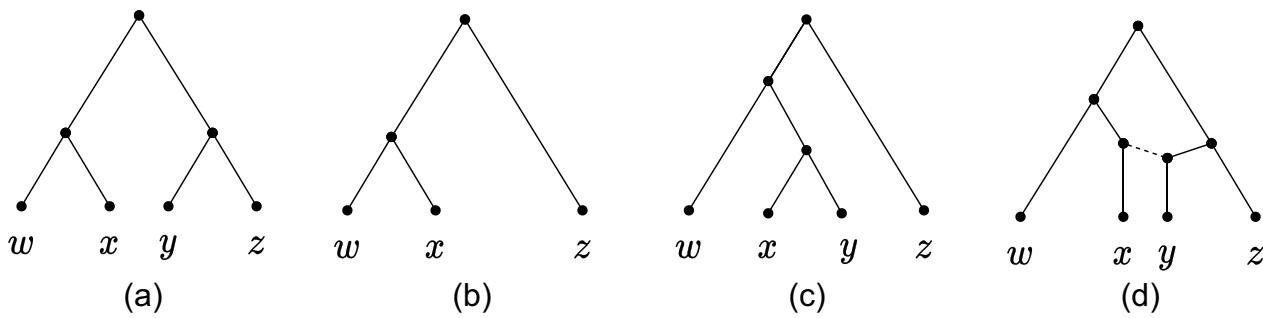


Fig. 4 Tree expansion of T (a) with the trivial cherry (x, y) of $T_{(y,z)}$. (b) After picking cherry (y, z) , leaf y is missing in $T^{(1)}$. (c) Leaf x is replaced by the cherry (x, y) . After completion of the heuristic, we have $S_T = (y, z), (x, y), (y, w), (w, z)$. (d) The network N_T reconstructed from $S^{(1)} \circ (x, y)$. Note that the input tree T is displayed in N_T (solid edges)

partial CPS S obtained right after applying tree expansion to a tree T_S : in other words, to obtain N_T we add to the partially reduced tree T_S the trivial pair (x, y) and then all the pairs previously reduced by S in the sense of Lemma 1. We show that N_T always displays T , the original tree; (2), we prove that this holds for an arbitrary sequence of tree expansion operations; and (3), since the CPS obtained using tree expansions fully reduces the networks of point (2), and since these networks display the trees in the original set \mathcal{T} , we have the desired property by Theorem 1. We prove this more formally with the following lemma.

Lemma 3 *Let S be the CPS produced by TrivialRand using tree expansion with input T . Then the network reconstructed from S displays all the trees in \mathcal{T} .*

Proof Let us start with the case where only 1 tree expansion occurs. Let $S^{(i-1)}$ be the partial CPS constructed in the first $i - 1$ steps of TrivialRand, and let i be the step in which we pick a trivial pair (x, y) . For each $T \in \mathcal{T}_{S^{(i-1)}}$ that is reduced by $S^{(i-1)}$ to a tree $T^{(i-1)}$ of type (iv) for (x, y) , let $S_T^{(i-1)}$ be the subsequence of $S^{(i-1)}$ consisting only of the pairs that subsequently affect T . We use the partial CPS $S_T^i = S_T^{(i-1)} \circ (x, y)$ to reconstruct a network N_T with a method underlying Lemma 1, starting from $T^{(i-1)}$: see Fig. 4d.

For trees of type (i)–(iii), $N_T = T$. We call the set \mathcal{N}_T , consisting of the networks N_T for all $T \in \mathcal{T}$, the expanded reconstruction of \mathcal{T} . Note that, by construction and Lemma 1, all the elements of \mathcal{N}_T after reducing, in order, the pairs of $S^{(i-1)} \circ (x, y)$, are trees: in particular, they are equal to the trees of $\mathcal{T}_{S^{(i-1)} \circ (x, y)}$ in which all the labels y have been replaced by x . We denote this set of trees $(\mathcal{N}_T)_{S^{(i-1)} \circ (x, y)}$.

We can generalise this notion to multiple trivial pairs: we denote by $\mathcal{N}_T^{(j)}$ the expanded reconstruction of \mathcal{T} with the first j trivial pairs, and suppose we added the j -th pair (w, z) to the partial CPS S at the k -th step. Consider a tree $T' \in (\mathcal{N}_T^{(j-1)})_{S^{(k-1)}}$ of type (iv) for (w, z) , and let $N_T^{(j-1)} \in \mathcal{N}_T^{(j-1)}$ be the network it originated from. Let $S_T^{(k-1)}$ be the subsequence of $S^{(k-1)}$ consisting only of the pairs that subsequently affected $N_T^{(j-1)}$. Then $N_T^{(j)}$ is the network reconstructed from $S_T^{(k-1)} \circ (w, z)$, starting from T' . For trees of $(\mathcal{N}_T^{(j-1)})_{S^{(k-1)}}$ that are of type (i)–(iii) for (w, z) , we have $N_T^{(j)} = N_T^{(j-1)}$. The elements of $\mathcal{N}_T^{(j)}$ are all networks $N_T^{(j)}$. For completeness, we define $\mathcal{N}_T^{(0)} = \mathcal{T}$ and $\mathcal{N}_T^{(1)} = \mathcal{N}_T$.

By construction, S fully reduces all the networks in $\mathcal{N}_T^{(j)}$, thus the network N reconstructed from S displays all of them by Theorem 1. We prove that $N_T^{(j)}$ displays T for all $T \in \mathcal{T}$, and thus N displays the original tree set \mathcal{T} too, by induction on j .

In the base case, we pick $j = 0$ trivial pairs, so the statement is true by Theorem 1. Now let $j > 0$. The induction hypothesis is that each network $N_T^{(j-1)} \in \mathcal{N}_T^{(j-1)}$ displays the tree $T \in \mathcal{T}$ it originated from. Let (w, z) be the j -th trivial pair, added to the sequence at position k . Let $T' \in (\mathcal{N}_T^{(j-1)})_{S^{(k-1)}}$ be a tree of type (iv) for (w, z) , and let $N_T^{(j-1)}$ be the network it originates from. Then there are two possibilities: either z is a leaf of $N_T^{(j-1)}$ or it is not. In case it is not, then adding (w, z) to $N_T^{(j-1)}$ does not create any new reticulation, and clearly $N_T^{(j)}$ keeps displaying T . If z does appear in $N_T^{(j-1)}$, then it must have been reduced by a pair (z, v) of $S^{(k-1)}$ (otherwise T' would not

be of type (iv)). Then the network $N_T^{(j)}$ has an extra reticulation, created with the insertion of (z, v) at some point after (w, z) during the backwards reconstruction. In both cases, by [15, Lemma 10] $N_T^{(j-1)}$ is displayed in $N_T^{(j)}$, and thus by the induction hypothesis T is displayed too. \square

Good cherries in theory

By Lemma 1 the binary network N reconstructed from a CPS S is such that S is of minimum length for N , that is, there exists no shorter CPS that fully reduces N . By Theorem 1 if S , in turn, fully reduces T , then N displays all the trees in T . Depending on S , though, N is not necessarily an optimal network (i.e., with minimum reticulation number) among the ones displaying T : see Example 2.

Let $\text{OPT}(T)$ denote the set of networks that display T with the minimum possible number of reticulations (in general, this set contains more than one network). Ideally, we would like to produce a CPS fully reducing T that is also a minimum-length CPS fully reducing some network of $\text{OPT}(T)$. In other words, we aim to find a CPS $\tilde{S} = (x_1, y_1), \dots, (x_n, y_n)$ such that, for any $i \in [1, n]$, (x_i, y_i) is a reducible pair of $\tilde{N}_{\tilde{S}^{(i-1)}}$, where $\tilde{S}^{(0)} = \emptyset$, $\tilde{S}^{(k)} = (x_1, y_1), \dots, (x_k, y_k)$ for all $k \in [1, n]$, and $\tilde{N} \in \text{OPT}(T)$. Let $S = (x_1, y_1), \dots, (x_n, y_n)$ be a CPS fully reducing T and let $\text{OPT}^{(k)}(T)$ consist of all networks $N \in \text{OPT}(T)$ such that each pair (x_i, y_i) , $i \in [1, k]$, is reducible in $N_{\tilde{S}^{(i-1)}}$.

Lemma 4 *A CPS S reducing T reconstructs an optimal network \tilde{N} if and only if each pair (x_i, y_i) of S is reducible in $\tilde{N}_{\tilde{S}^{(i-1)}}$, for all $i \in [1, n]$.*

Proof (\Rightarrow) By Lemma 1, S is a minimum-length CPS for the network \tilde{N} that is reconstructed from it; and a CPS $C = (w_1, z_1), \dots, (w_n, z_n)$ reducing a network N is of minimum length precisely if, for all $j \in [1, n]$, (w_j, z_j) is a reducible pair of $N_{C^{(j-1)}}$ (otherwise the pair (w_j, z_j) could be removed from C and the new sequence would still reduce N).

(\Leftarrow) If all pairs of S affect some optimal network \tilde{N} , then S is a minimum-length CPS for \tilde{N} , thus \tilde{N} is reconstructed from S (and it displays T by Theorem 1). \square

Lemma 4 implies that if some pair (x_i, y_i) of S does not reduce any network in $\text{OPT}^{(i-1)}(T)$, then the network reconstructed from S is not optimal: see Example 2.

Example 2 Consider the set T of Fig. 2b: $S = (y, x), (y, z), (w, x), (x, z)$ is a CPS that fully reduces

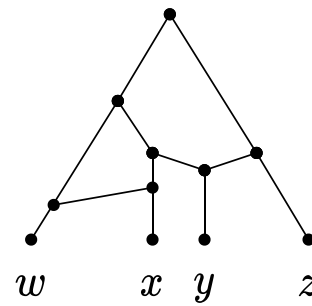


Fig. 5 Network N' of Example 2

T and consists only of pairs successively reducible in the network N of Fig. 2a, thus it reconstructs it by Lemma 1. Now consider (w, x) , which is reducible in T but not in N , and pick it as first pair, to obtain e.g. $S' = (w, x), (y, z), (y, x), (w, x), (x, z)$. The network N' reconstructed from S' , depicted in Fig. 5, has $r(N') = 2$, whereas $r(N) = 1$.

Suppose we are incrementally constructing a CPS $S = (x_1, y_1), \dots, (x_n, y_n)$ for T with some heuristic in the CPH class. If we had an oracle that at each iteration i told us if a reducible pair (x, y) of $T^{(i-1)}$ were a reducible pair in some $N \in \text{OPT}^{(i-1)}(T)$, then, by Lemma 4, we could solve HYBRIDIZATION optimally. Unfortunately no such exact oracle can exist (unless $P = NP$). However, in the next section we exploit this idea to design machine-learned heuristics in the CPH framework.

Predicting good cherries via machine learning

In this section, we present a supervised machine-learning classifier that (imperfectly) simulates the ideal oracle described at the end of Sect. "Good cherries in theory". The goal is to predict, based on T , whether a given cherry of T is a cherry or a reticulated cherry in a network N displaying T with a close-to-optimal number of reticulations, without knowing N . Based on Lemma 4, we then exploit the output of the classifier to define new functions `PickNext`, that in turn define new machine-learned heuristics in the class of CPH (Algorithm 1).

Specifically, we train a random forest classifier on data that encapsulates information on the cherries in the tree set. Given a partial CPS, each reducible pair in T_S is represented by one data point. Each data point is a pair (\mathbf{F}, \mathbf{c}) , where \mathbf{F} is an array containing the features of a cherry (x, y) and \mathbf{c} is an array containing the probability that the cherry belongs to each of the possible classes described below. Recall that cherries are ordered pairs, so (x, y) and (y, x) give rise to two distinct data points. The classification model learns the association between \mathbf{F} and \mathbf{c} .

Table 1 Features of a cherry (x, y)

Num	Feature name	Description
1	Cherry in tree	Ratio of trees that contain cherry (x, y)
2	New cherries	Number of new cherries of \mathcal{T} after picking cherry (x, y)
3	Before/after	Ratio of the number of cherries of \mathcal{T} before/after picking cherry (x, y)
4	Trivial	Ratio of trees with both leaves x and y that contain cherry (x, y)
5	Leaves in tree	Ratio of trees that contain both leaves x and y
<i>Features measured by distance (d) and topology (t)</i>		
$6_{d,t}$	Tree depth	Avg over trees with (x, y) of ratios “depth of the tree/max depth over all trees”
$7_{d,t}$	Cherry depth	Avg over trees with (x, y) of ratios “depth of (x, y) in the tree/depth of the tree”
$8_{d,t}$	Leaf distance	Avg over trees with x and y of ratios “ x - y leaf distance/depth of the tree”
$9_{d,t}$	Leaf depth x	Avg over trees with x and y of ratios “root- x distance/depth of the tree”
$10_{d,t}$	Leaf depth y	Avg over trees with x and y of ratios “root- y distance/depth of the tree”
$11_{d,t}$	LCA distance	Avg over trees with x and y of ratios “ x -LCA(x, y) distance/ y -LCA(x, y) distance”
$12_{d,t}$	Depth x/y	Avg over trees with x and y of ratios “root- x distance/root- y distance”

Features 6–12 can be computed for both branch lengths and unweighted branches. We refer to these two options as distance and topological distance, respectively

The true class of a cherry (x, y) of \mathcal{T} depends on whether, for the (unknown) network N that we aim to reconstruct: (class 1) (x, y) is a cherry of N ; (class 2) (x, y) is a reticulated cherry of N ; (class 3) (x, y) is not reducible in N , but (y, x) is a reticulated cherry; or (class 4) neither (x, y) nor (y, x) are reducible in N . Thus, for the data point of a cherry (x, y) , $\mathbf{c}[i]$ contains the probability that (x, y) is in class i , and $\mathbf{c}[1] + \mathbf{c}[2]$ gives the predicted probability that (x, y) is reducible in N . We define the following two heuristics in the CPH framework.

ML	Given a threshold $\tau \in [0, 1)$, function <code>PickNext</code> picks the cherry with the highest predicted probability of being reducible in N if this probability is at least τ ; or a random cherry if none has a probability of being reducible above τ
TrivialML	Function <code>PickNext</code> picks a random trivial pair, if there exists one; otherwise it uses the same rules as ML

In both cases, whenever a trivial pair is picked, we do tree expansion, as described in Sect. “[Improving heuristic TrivialRand via tree expansion](#)”. Note that if $\tau = 0$, since the predicted probabilities are never exactly 0, ML is fully deterministic. In Sect. “[Effect of the threshold on ML](#)” we show how the performance of ML is impacted by the choice of different thresholds.

To assign a class to each cherry, we define 19 features, summarised in Table 1, that may capture essential information about the structure of the set of trees, and that can be efficiently computed and updated at every iteration of the heuristics.

The depth (resp. topological depth) of a node u in a tree T is the total branch length (resp. the total number of edges) on the root-to- u path; the depth of a cherry (x, y) is the depth of the common parent of x and y ; the depth of T is the maximum depth of any cherry of T . The (topological) leaf distance between x and y is the total branch length of the path from the parent of x to the lowest common ancestor of x and y , denoted by $\text{LCA}(x, y)$, plus the total length of the path from the parent of y to $\text{LCA}(x, y)$ (resp. the total number of edges on both paths). In particular, the leaf distance between the leaves of a cherry is zero.

Time complexity

Designing algorithms with the best possible time complexity was not the main objective of this work. However, for completeness, we provide worst-case upper bounds on the running time of our heuristics. The omitted proofs can be found in Appendix A. We start by stating a general upper bound for the whole CPH framework in the function of the time required by the `PickNext` routine.

Lemma 5 *The running time of the heuristics in the CPH framework is $\mathcal{O}(|\mathcal{T}|^2|X| + \text{cost}(\text{PickNext}))$, where $\text{cost}(\text{PickNext})$ is the total time required to choose reducible pairs over all iterations. In particular, `Rand` takes $\mathcal{O}(|\mathcal{T}|^2|X|)$ time.*

Proof An upper bound for the sequence length is $(|X| - 1)|\mathcal{T}|$ as each tree can individually be fully reduced using at most $|X| - 1$ pairs. Hence, the while loop of Algorithm 1 is executed at most $(|X| - 1)|\mathcal{T}|$ times. Moreover, reducing the pair and updating the set of reducible pairs after one iteration takes $\mathcal{O}(1)$ time per

tree. Combining this with the fact that `CompleteSeq` takes $\mathcal{O}(|S|) = \mathcal{O}(|X||T|)$ time, we obtain the stated time complexity. Since choosing a random reducible pair takes $\mathcal{O}(1)$ time at each iteration, `Rand` takes trivially $\mathcal{O}(|T|^2|X|)$ time. \square

Note that by Lemma 2 the number of reticulations $r(N)$ of the network reconstructed from the output CPS is bounded by $(|X| - 1)|T| - |X| + 1 = \mathcal{O}(|T| \cdot |X|)$, and thus the time complexity of `Rand` is also $\mathcal{O}(r(N)|T|)$.

Let us now focus on the time complexity of the machine-learned heuristics `ML` and `TrivialML`. At any moment during the execution of the heuristics, we maintain a data structure that stores all the current cherries in \mathcal{T} and allows constant-time insertions, deletions, and access to the cherries and their features. A possible implementation of this data structure consists of a hashtable `cherryfeatures` paired with a list `cherrylist` of the pairs currently stored in `cherryfeatures`. We will use `cherrylist` to iterate over the current cherries of \mathcal{T} , and `cherryfeatures` to check whether a certain pair is currently a cherry of \mathcal{T} and to access its features.

Note that the total number of cherries inserted in `cherryfeatures` over all the iterations is bounded by the total size of the trees $||T||$ because up to two cherries can be created for each internal node over the whole execution. We will assume that we have constant-time access to the leaves of each tree: specifically, given $T \in \mathcal{T}$ and $x \in X$, we can check in constant time whether x is currently a leaf of T ¹.

Initialisation The cherries of \mathcal{T} can be identified and features 1–3 can be initially computed in $\mathcal{O}(|T|)$ time by traversing all trees bottom-up. Features 4–5 can be computed in $\mathcal{O}(\min\{|T| \cdot ||T||, |T| \cdot |X|^2\})$ time by checking, for each $T \in \mathcal{T}$ and each cherry (x, y) of \mathcal{T} , whether both x and y appear in T . Features $6_{d,t}$ to $12_{d,t}$ can also be initially computed with a traversal of \mathcal{T} made efficient by preprocessing each tree in linear time to allow constant-time LCA queries [21] and by storing the depth (both topological and with the branch lengths) of each node. We also store the topological and branch length depth of each tree and their maximum value over \mathcal{T} . Altogether this gives the following lemma.

Lemma 6 *Initialising all features for a tree set \mathcal{T} of total size $||T||$ over a set of taxa X requires $\mathcal{O}(\min\{|T| \cdot ||T||, |T| \cdot |X|^2\})$ time and $\mathcal{O}(|T|)$ space.*

¹ This can be obtained maintaining a list of leaves of each tree and a hashtable with the leaves as keys: the value of a key x is a pointer to the position of x in the list.

The next lemma provides an upper bound on the time complexity of updating the distance-independent features.

Lemma 7 *Updating features 1–5 for a set \mathcal{T} of $|T|$ trees of total size $||T||$ over a set of taxa X requires $\mathcal{O}(|T|(|T|| + |X|^2))$ total time and $\mathcal{O}(|T|)$ space.*

Since searching for trivial cherries at each iteration of the randomised heuristic `TrivialRand` can be done with the same procedure we use for updating feature 4 in the machine-learned heuristics, which in particular requires $\mathcal{O}(|T| \cdot ||T||)$ time, we have the following corollary.

Corollary 1 *The time complexity of `TrivialRand` is $\mathcal{O}(|T| \cdot ||T||) = \mathcal{O}(|T|^2 \cdot |X|)$.*

The total time required for updating the distance-dependent features raises the time complexity of `ML` and `TrivialML` to quadratic in the input size. However, the extensive analysis reported in Appendix A shows that this is only due to the single feature 6_d , and without such a feature, the machine-learned heuristics would be asymptotically as fast as the randomised ones. Since Table 4 in Appendix B shows that this feature is not particularly important, in future work it could be worth investigating whether disregarding it leads to equally good results in shorter time.

Lemma 8 *The time complexity of `ML` and `TrivialML` is $\mathcal{O}(|T|^2)$.*

Obtaining training data

The high-level idea to obtain training data is to first generate a phylogenetic network N ; then to extract the set \mathcal{T} of all the exhaustive trees displayed in N ; and finally, to iteratively choose a random reducible pair (x, y) of N , to reduce it in \mathcal{T} as well as in N , and to label the remaining cherries of \mathcal{T} with one of the four classes defined in Sect. "Predicting good cherries via machine learning" until the network is fully reduced.

We generate two different kinds of binary orchard networks, normal and not normal, with branch lengths and up to 9 reticulations using the LGT (lateral gene transfer) network generator of [22], imposing normality constraints when generating the normal networks. For each such network N , we then generate the set \mathcal{T} consisting of all the exhaustive trees displayed in N .

If N is normal, N is an optimal network for \mathcal{T} [23, Theorem 3.1]. This is not necessarily true for any LGT-generated network, but even in this case, we expect N to be

reasonably close to optimal, because we remove redundant reticulations when we generate it and because the trees in \mathcal{T} cover all the edges of N . In particular, for LGT networks $r(N)$ provides an upper bound estimate on the minimum possible number of reticulations of any network displaying \mathcal{T} , and we will use it as a reference value for assessing the quality of our results on synthetic LGT-generated data.

Experiments

The code of all our heuristics and for generating data is written in Python and is available at <https://github.com/estherjulien/learn2cherry-pick>. All experiments ran on an Intel Xeon Gold 6130 CPU @ 2.1 GHz with 96 GB RAM. We conducted experiments on both synthetic and real data, comparing the performance of Rand, TrivialRand, ML and TrivialML, using threshold $\tau = 0$. Similar to the training data, we generated two synthetic datasets by first growing a binary orchard network N using [22], and then extracting \mathcal{T} as a subset of the exhaustive trees displayed in N . We provide details on each dataset in Sect. "Experimental results".

We start by analysing the usefulness of tree expansion, the heuristic rule described in Sect. "Improving heuristic TrivialRand via tree expansion". We synthetically generated 112 instances for each tree set size $|\mathcal{T}| \in \{5, 10, 20, 50, 100\}$ (560 in total), all consisting of trees with 20 leaves each, and grouped them by $|\mathcal{T}|$; we then ran TrivialRand 200 times (both with and without tree expansion) on each instance, selected the best output for each of them, and finally took the average of these results over each group of instances. The results are in Fig. 6, showing that the use of tree expansion brought the output reticulation number down by at least 16% (for small instances) and up to 40% for the larger instances. We consistently chose to use this rule in all the heuristics that detect trivial cherries, namely, TrivialRand, TrivialML, ML (although ML does not explicitly favour trivial cherries, it does check whether a selected cherry is trivial using feature number 2), and the non-learned heuristic that will be introduced in Sect. "A non-learned heuristic based on important features".

Prediction model

The random forest is implemented with Python's `scikit-learn` [24] package using default settings. We evaluated the performance of our trained random forest models on different datasets in a holdout procedure: namely, we removed 10% of the data from each training dataset, trained the models on the remaining 90% and used the holdout 10% for testing. The accuracy was assessed by assigning to each test data point the class

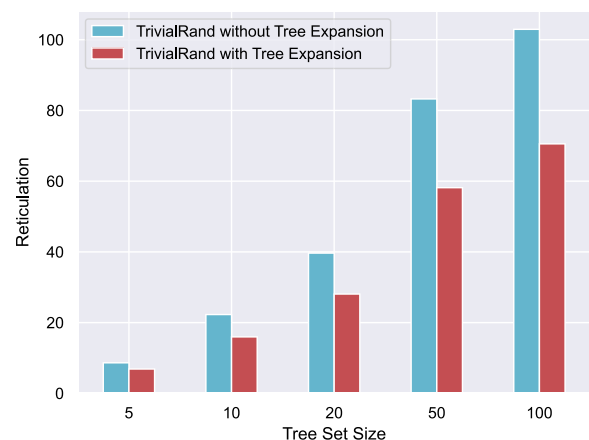


Fig. 6 Number of reticulations output by TrivialRand with and without using tree expansion. The height of the bars is the average reticulation number over each group, obtained by selecting the best of 200 runs for each instance

with the highest predicted probability and comparing it with the true class. Before training the models, we balanced each dataset so that each class had the same number of representatives.

Each training dataset differed in terms of the number M of networks used for generating it and the number of leaves of the networks. For each dataset, the number L of leaves of each generated network was uniformly sampled from $[2, \max L]$, where $\max L$ is the maximum number of leaves per network. We constructed LGT networks using the LGT generator of [22]. This generator has three parameters: n for the number of steps, α for the probability of lateral gene transfer events, and β for regulating the size of the biconnected components of the network (called blobs). The combination of these parameters determines the level (maximum number of reticulations per blob), the number of reticulations, and the number of leaves of the output network. In our experiments, α was uniformly sampled from $[0.1, 0.5]$ and $\beta = 1$ (see [22] for more details).

To generate normal networks we used the same generator with the same parameters, but before adding a reticulation we check if it respects the normality constraints and only add it if it does. Each generated network gave rise to a number of data points: the total number of data points per dataset is shown in Table 3 in Appendix B. Each row of Table 3 corresponds to a dataset on which the random forest can be trained, obtaining as many ML models. We tested all the models on all the synthetically generated instances: we show these results in Figs. 18, 19 and 20 in Appendix C. In Sect. "Experimental results" we will report the results obtained for the best-performing model for each type of instance.

Among the advantages of using a random forest as a prediction model, there is the ability of computing feature importance, shown in Table 4 in Appendix B. Some of the most useful features for a cherry (x, y) appear to be ‘Trivial’ (the ratio of the trees containing both leaves x and y in which (x, y) is a cherry) and ‘Cherry in tree’ (the ratio of trees that contain (x, y)). This was not unexpected, as these features are well-suited to identify trivial cherries.

‘Leaf distance’ (t, d), ‘LCA distance’ (t) and ‘Depth x/y ’ (t) are also important features. The rationale behind these features was to try to identify reticulated cherries. This was also the idea for the feature ‘Before/after’, but this has, surprisingly, a very low importance score. In future work, we plan to conduct a thorough analysis of whether some of the seemingly least important features can be removed without affecting the quality of the results.

Experimental results

We assessed the performance of our heuristics on instances of four types: normal, LGT, ZODS (binary non-orchard networks), and real data. Normal, LGT and ZODS data are synthetically generated. We generated the normal instances much as we did for the training data: we first grew a normal network using the LGT generator and then extracted all the exhaustive trees displayed in the network. We generated normal data for different combinations of the following parameters: $L \in \{20, 50, 100\}$ (number of leaves per tree) and $R \in \{5, 6, 7\}$ (reticulation number of the original network). Note that, for normal instances, $|\mathcal{T}| = 2^R$. For every combination of the parameters L and R we generated 48 instances: by instance group we indicate the set of instances generated for one specific parameter pair.

For the LGT instances, we grew the networks using the LGT generator, but unlike for the normal instances we then extracted only a subset of the exhaustive trees from each of them, up to a certain amount $|\mathcal{T}| \in \{20, 50, 100\}$. The other parameters for LGT instances are the number of leaves $L \in \{20, 50, 100\}$ and the number of reticulations $R \in \{10, 20, 30\}$. For a fixed pair $(L, |\mathcal{T}|)$, we generated 16 instances for each possible value of R , and analogously, for a fixed pair (L, R) we generated 16 instances for each value of $|\mathcal{T}|$. The 48 instances generated for a fixed pair of values constitute a LGT instance group.

We generated non-orchard binary networks using the ZODS generator [25]. This generator has two user-defined parameters: λ , which regulates the speciation rate, and ν , which regulates the hybridization rate. Following [26] we set $\lambda = 1$ and we sampled $\nu \in [0.0001, 0.4]$ uniformly at random. Like for the LGT instances, we generated an instance group of size 48 for each pair of values $(L, |\mathcal{T}|)$ and (L, R) , with $L \in \{20, 50, 100\}$, $|\mathcal{T}| \in \{20, 50, 100\}$, $R \in \{10, 20, 30\}$.

Table 2 Number of real data instances for each group (combination of parameters L and $|\mathcal{T}|$)

L	$ \mathcal{T} $				Tot. Trees
	10	20	50	100	
20	50	50	50	50	1684
50	20	20	20	20	290
100	5	5	1	0	53

Finally, the real-world dataset consists of gene trees on homologous gene sets found in bacterial and archaeal genomes, was originally constructed in [27] and made binary in [3]. We extracted a subset of instances (Table 2) from the binary dataset, for every combination of parameters $L \in \{20, 50, 100\}$ and $|\mathcal{T}| \in \{10, 20, 50, 100\}$.

For the synthetically generated datasets, we evaluated the performance of each heuristic in terms of the output number of reticulations, comparing it with the number of reticulations of the network N from which we extracted \mathcal{T} . For the normal instances, N is the optimal network [23, Theorem 3.1]; this is not true, in general, for the LGT and ZODS datasets, but even in these cases, $r(N)$ clearly provides an estimate (from above) of the optimal value, and thus we used it as a reference value for our experimental evaluation.

For real data, in the absence of the natural estimate on the optimal number of reticulations provided by the starting network, we evaluated the performance of the heuristics comparing our results with the ones given by the exact algorithms from [3] (TreeChild) and from [7] (Hybroscale), using the same datasets that were used to test the two methods in [3]. These datasets consist of rather small instances ($|\mathcal{T}| \leq 8$); for larger instances, we run TrivialRand 1000 times for each instance group, selected the best result for each group, and used it as a reference value (Fig. 10).

We now describe in detail the results we obtained for each type of data and each of the algorithms we tested.

Experiments on normal data

For the experiments in this section we used the ML model trained on 1000 normal networks with at most 100 leaves per network (see Fig. 18 in Appendix C). We ran the machine-learned heuristics once for each instance and then averaged the results within each instance group (recall that one instance group consists of the sets of all the exhaustive trees of 48 normal networks having the same fixed number of leaves and reticulations). The randomised heuristics Rand and TrivialRand were run $\min\{x(I), 1000\}$ times for each instance I , where $x(I)$ is the number of runs that can be executed in the same

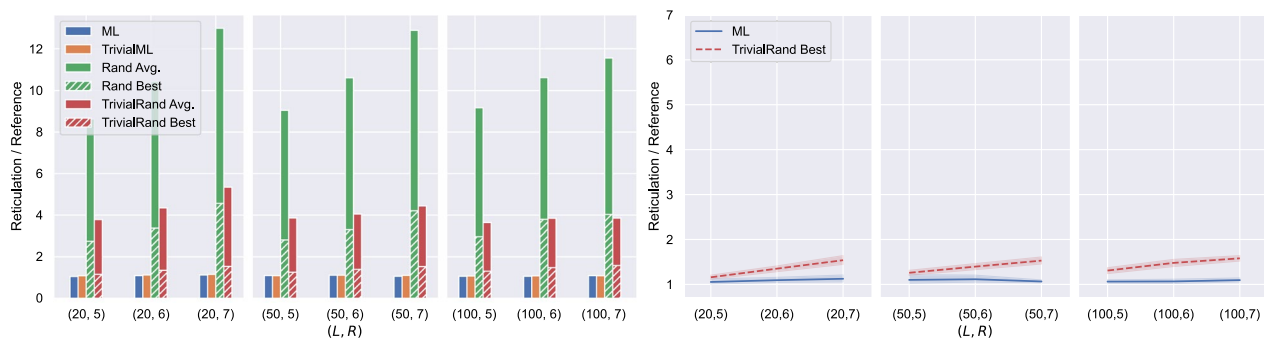


Fig. 7 Experimental results for normal data. Each point on the horizontal axis corresponds to one instance group. In the left graph, the height of each bar gives the average of the results over all instances of the group, scaled by the optimum value for the group. The right graph compares the average output of ML within each instance group and the average of the best output given by *TrivialRand* for each instance of a group. The shaded areas represent 95% confidence intervals

time as one run of ML on the same instance. We omitted the results for *LowPair* because they were at least 44% worse on average than the worst-performing heuristic we report.

In Fig. 7 we summarise the results. Solid bars represent the ratio between the average reported reticulation number and the optimal value, for each instance group and for each of the four heuristics. Dashed bars represent the ratio between the average (over the instances within each group) of the best result among the $\min\{x(I), 1000\}$ runs for each instance I and the optimum.

The machine-learned heuristics ML and *TrivialML* seem to perform very similarly, both leading to solutions close to optimum. The average performance of *TrivialRand* is around 4 times worse than the machine-learned heuristics; in contrast, if we only consider the best solution among the multiple runs for each instance, they are quite good, having only up to 49% more reticulations than the optimal solution, but they are still at least 4% worse (29% worse on average) than the machine-learned heuristics' solutions: see the right graph of Fig. 7.

The left graph of Fig. 7 shows that the performance of the randomised heuristics seems to be negatively impacted by the number of reticulations of the optimal solution, while we do not observe a clear trend for the machine-learned heuristics, whose performance is very close to optimum for all the considered instance groups. Indeed, the number of existing phylogenetic networks with a certain number of leaves grows exponentially in the number of reticulations, thus making it less probable to reconstruct a “good” network with random choices. This is consistent with the existing exact methods being FPT in the number of reticulations [3, 28].

The fully randomised heuristic *Rand* always performed much worse than all the others, indicating that identifying the trivial cherries has a great impact on the

effectiveness of the algorithms (recall that ML implicitly identifies trivial cherries).

Experiments on LGT data

For the experiments on LGT data we used the ML model trained on 1000 LGT networks with at most 100 leaves per network (see Fig. 19 in Appendix C). The setting of the experiments is the same as for the normal data (we run the randomised heuristics multiple times and the machine-learned heuristics only once for each instance), with two important differences.

First, for LGT data we only take proper subsets of the exhaustive trees displayed by the generating networks, and thus we have two kinds of instance groups: one where in each group the number of trees extracted from a network and the number of leaves of the networks are fixed, but the trees come from networks with different numbers of reticulations; and one where the number of reticulations of the generating networks and their number of leaves are fixed, but the number of trees extracted from a network varies.

The second important difference is that the reference value we use for LGT networks is not necessarily the optimum, but it is just an upper bound given by the number of reticulations of the generating networks which we expect to be reasonably close to the optimum (see Sect. “Obtaining training data”).

The results for the LGT datasets are shown in Fig. 8. Comparing these results with those of Fig. 7, it is evident that the LGT instances were more difficult than the normal ones for all the tested heuristics: this could be due to the fact that the normal instances consisted of all the exhaustive trees of the generating networks, while the LGT instances only have a subset of them and thus carry less information.

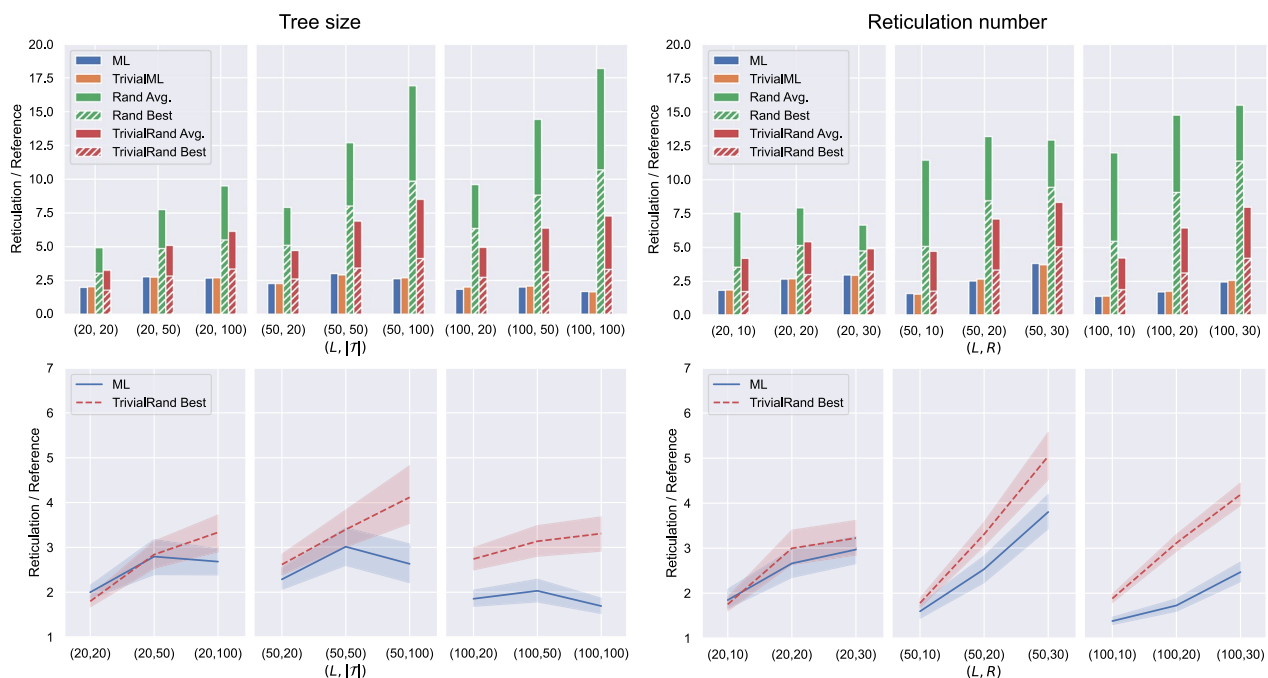


Fig. 8 Experimental results for LGT data. Each point on the horizontal axis corresponds to one instance group. For the graphs on the left, there is one group for each fixed pair $(L, |\mathcal{T}|)$ consisting of 16 instances coming from LGT networks for each value of $R \in \{10, 20, 30\}$. For the graphs on the right, there is one group for each fixed pair (L, R) consisting of 16 instances coming from LGT networks for each value of $|\mathcal{T}| \in \{20, 50, 100\}$. In the top graphs, the height of each bar gives the average of the results over all instances of the group, each scaled by the number of reticulations of the generating network. The bottom graphs compare the average output of ML within each instance group and the average of the best output given by TrivialRand for each instance group. The shaded areas represent 95% confidence intervals

The machine-learned heuristics performed substantially better (up to 80% on average) than the best randomised heuristic TrivialRand in all instance groups but the ones with the smallest values for parameters $R, |\mathcal{T}|$ and L , for which the performances are essentially overlapping. On the contrary, the advantage of the machine-learned methods is more pronounced when the parameters are set to the highest values. This is because the larger the parameters, the more the possible different networks that embed \mathcal{T} , thus the less likely for the randomised methods to find a good solution.

From the graphs on the right of Fig. 8, it seems that the number of reticulations has a negative impact on both machine-learned and randomised heuristics, the effect being more pronounced for the randomised ones. The effect of the number of trees $|\mathcal{T}|$ on the quality of the solutions is not as clear (Fig. 8, left). However, we can still see that the trend of ML and TrivialRand is the same: the “difficult” instance groups are so for both heuristics, even if the degradation in the quality of the solutions for such instance groups is less marked for ML than for TrivialRand.

Experiments on ZODS data

For the experiments on ZODS data we used the ML model trained on 1000 LGT networks with at most 100 leaves per network (see Fig. 20 in Appendix C). The setting of the experiments is the same as for the LGT data, and the results are shown in Fig. 9.

At first glance, the performance of the randomised heuristics seems to be better for ZODS data than for LGT data (compare Figs. 8 and 9), which sounds counterintuitive. Recall, however, that all the graphs show the ratio between the number of reticulations returned by our methods and a reference value, i.e., the number of reticulations of the generating network: while we expect this reference to be reasonably close to the optimum for LGT networks, this is not the case for ZODS networks. In fact, a closer look to ZODS networks shows that they have a large number of redundant reticulations which could be removed without changing the set of trees they display, and thus their reticulation number is in general quite larger than the optimum. This is an inherent effect of the ZODS generator not having any constraints on the reticulations that can be introduced, and it is more marked on networks with a small number of leaves.

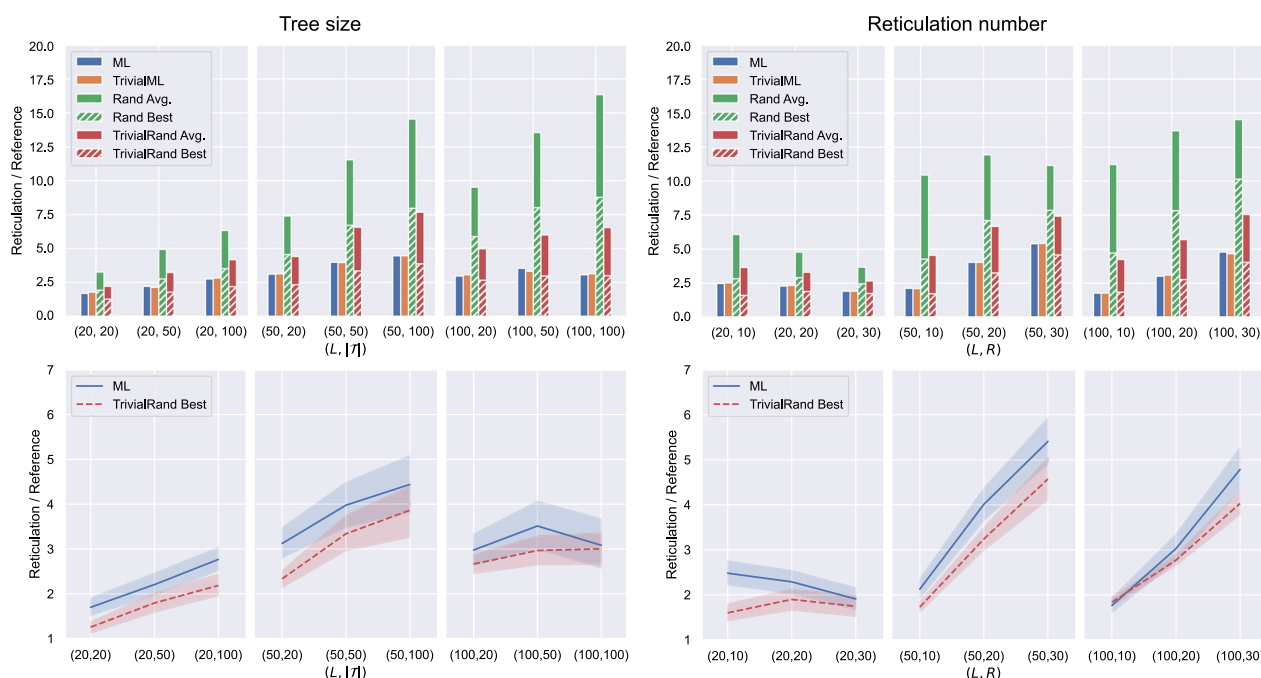


Fig. 9 Experimental results for ZODS data. Each point on the horizontal axis corresponds to one instance group. For the graphs on the left, there is one group for each fixed pair $(L, |T|)$ consisting of 16 instances coming from ZODS networks for each value of $R \in \{10, 20, 30\}$. For the graphs on the right, there is one group for each fixed pair (L, R) consisting of 16 instances coming from ZODS networks for each value of $|T| \in \{20, 50, 100\}$. In the top graphs, the height of each bar gives the average of the results it represents over all instances of the group, each scaled by the number of reticulations of the network the instance originated from. The bottom graphs compare the average output of ML within each instance group and the average of the best output given by TrivialRand for each group instance. The shaded areas represent 95% confidence intervals

Having a reference value significantly larger than the optimum makes the ratios in Fig. 9 small (close to 1, especially for TrivialRand on small instances) without implying that the results for the ZODS data are better than the ones for the LGT data. The graphs of Figs. 8 and 9 are thus not directly comparable.

The reference value for the experiments on ZODS data not being realistically close to the optimum, however, does not invalidate their significance. Indeed, the scope of such experiments was just to compare the performance of the machine-learned heuristics on data entirely different from those they were trained on with the performance of the randomised heuristics, which should not depend on the type of network that was used to generate the input.

As expected and in contrast with normal and LGT data, the results show that the machine-learned heuristics perform worse than the randomised ones on ZODS data, consistent with the ML methods being trained on a completely different class of networks.

Experiments on real data

We conducted two sets of experiments on real data, using the ML model trained on the dataset trained on 1000 LGT networks with at most 100 leaves each. For sufficiently

small instances, we compared the results of our heuristics with the results of two existing tools for reconstructing networks from binary trees: TreeChild[3] and Hybroscale[7]. Hybroscale is an exact method performing an exhaustive search on the networks displaying the input trees, therefore it can only handle reasonably small instances in terms of the number of input trees. TreeChild is a fixed-parameter (in the number of reticulations of the output) exact algorithm that reconstructs the best tree-child network, a restricted class of phylogenetic networks, and due to its fast-growing computation time cannot handle large instances either.

We tested ML and TrivialRand against Hybroscale and TreeChild using the same dataset used in [3], in turn taken from [27]. The dataset consists of ten instances for each possible combination of the parameters $|T| \in [2, 8]$ and $L \in \{10, 20, 30, 40, 50, 60, 80, 100, 150\}$. In Fig. 10 we show results only for the instance groups for which Hybroscale or TreeChild could output a solution within 1 h, consistent with the experiments in [3]. As a consequence of Hybroscale and TreeChild being exact methods (TreeChild only for a restricted class of networks), they performed better than both ML and TrivialRand on all instances they could solve, although the best results of TrivialRand are often

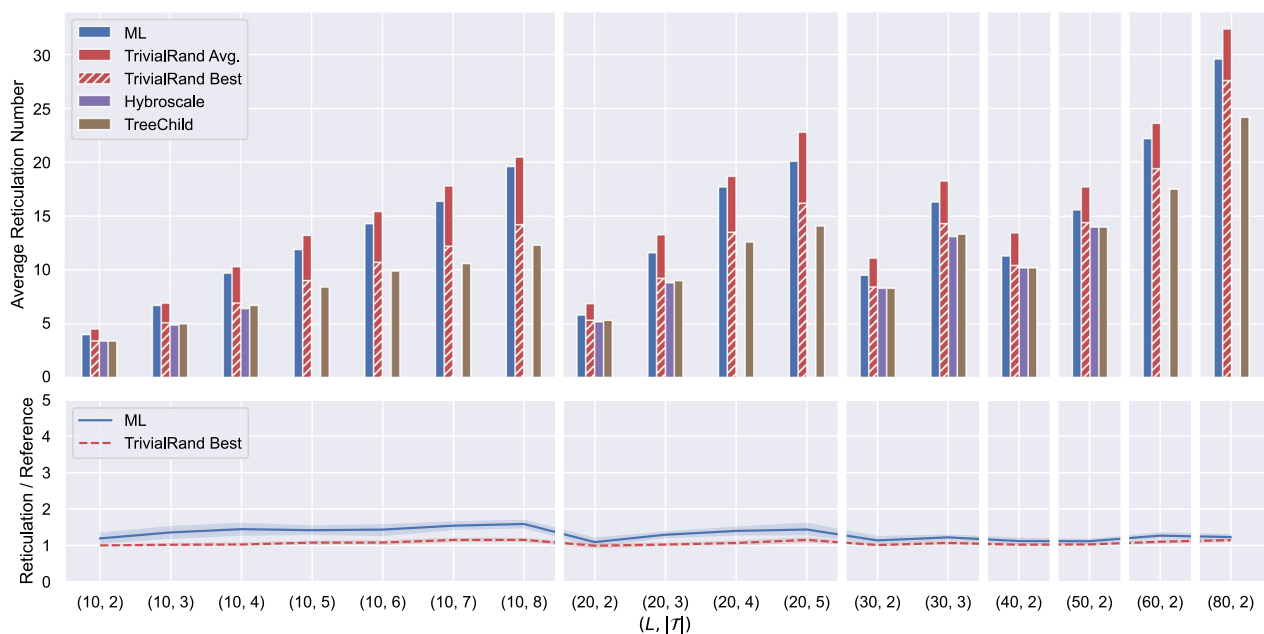


Fig. 10 Comparison of ML, TrivialRand, Hybroscale, and TreeChild on real data. Each point on the horizontal axis corresponds to one instance group, consisting of 10 instances for a fixed pair $(L, |T|)$. In the top graph, the height of each bar gives the average, over all instances of the group, of the number of reticulations returned by the method. The bottom graphs compare the average output of ML within each instance group and the average of the best output given by TrivialRand within the group. The shaded areas represent 95% confidence intervals

close (no worse than 15%) and sometimes match the optimal value.

The main advantage of our heuristics is that they can handle much larger instances than the exact methods. In the conference version of this paper [19] we showed the results of our heuristics on large real instances, using a ML model trained on 10 networks with at most 100 leaves each. These results demonstrated that consistently with the simulated data, the machine-learned heuristics gave significantly better results than the randomised ones for the largest instances. When we first repeated the experiments with the new models trained on 1000 networks with $\max L = 100$, however, we did not obtain similar results: instead, the results of the randomised heuristics were better or only marginally worse than the machine-learned ones on almost all the instance groups, including the largest.

Puzzled by these results, we conducted an experiment on the impact of the training set on real data. The results are reported in Fig. 11, and show that the choice of the networks on which we train our model has a big impact on the quality of the results for the real datasets. This is in contrast with what we observed for the synthetic datasets, for which only the class of the training networks was important, not the specific instances of the networks themselves. According to what was noted in [3], this is

most likely due to the fact that the real phylogenetic data have substantially more structure than random synthetic datasets, and the randomly generated training networks do not always reflect this structure. By chance, the networks we used for training the model we used in [19] were similar to real phylogenetic networks, unlike the 1000 networks in the training set of this paper.

Experiments on scalability

We conducted experiments to study how the running time of our heuristics scales with increasing instance size for all datasets. In Fig. 12 we report the average of the running times of ML for the instances within each instance group with a 95% confidence interval, for an increasing number of reticulations (synthetic datasets) or number of trees (real dataset). The datasets and the instance groups are those described in the previous sections. Note that we did not report the running times of the randomised heuristics because they are meant to be executed multiple times on each instance, and in all the experiments we bounded the number of executions precisely using the time required for one run of ML.

We also compared the running time of our heuristics with the running times of the exact methods TreeChild and Hybroscale. The results are shown in Fig. 13 and are consistent with the execution times of the

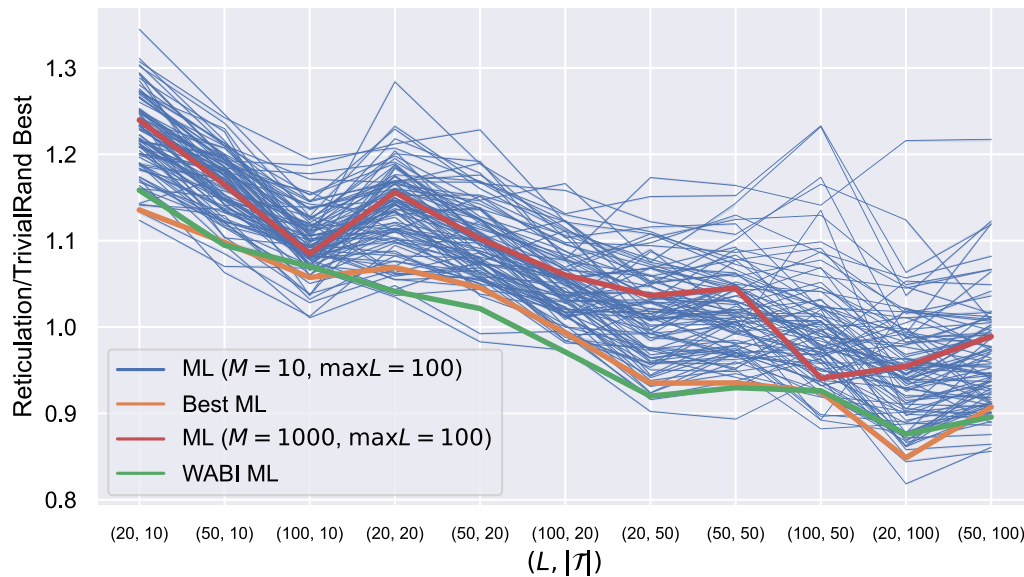


Fig. 11 Ratio between the performance of ML and the best value output by `TrivialRand` for different instance groups and different training sets. `TrivialRand` is executed $\min\{x(l), 1000\}$ times for each instance l , $x(l)$ being the number of runs that could be completed in the same time as one run of ML on l . The results are then averaged within each group. Each blue line represents the results obtained training the model with a different set of 10 randomly generated LGT networks with at most 100 leaves each. The green line corresponds to the training set used in [19]; the orange line represents one of the best-performing sets; the red line corresponds to the training set we used for the experiments on LGT and ZODS data in this paper, consisting of 1000 randomly generated LGT networks

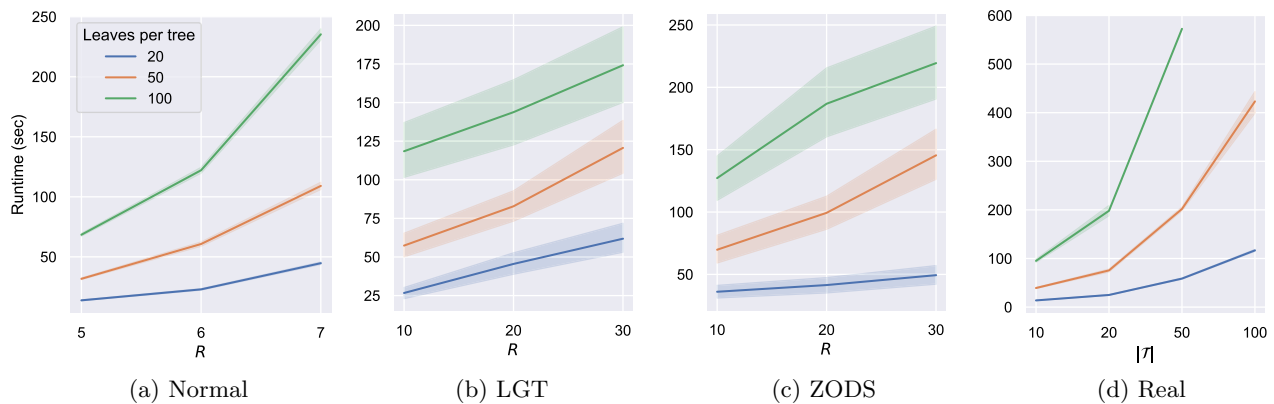


Fig. 12 The running time (in seconds) of ML for the instance groups described in Sects. "Experiments on normal data", "Experiments on LGT data", "Experiments on ZODS data", "Experiments on real data". The solid lines represent the average of the running times for the instances within each instance group. The shaded areas represent 95% confidence intervals

exact methods growing exponentially, while the running time of our heuristics grows polynomially. Note that networks with more reticulations are reduced by longer CPS and thus the running time increases with the number of reticulations.

Experiments on non-exhaustive input trees

The instances on which we tested our methods so far all consisted of a set of exhaustive trees, that is, each input

tree had the same set of leaves which coincided with the set of leaves of the network. However, this is not a requirement of our heuristics, which are able to produce feasible solutions also when the leaf sets of the input trees are different, that is when their leaves are proper subsets of the leaves of the optimal networks that display them.

To test their performance on this kind of data, we generated 18 LGT instance groups starting from the instances we used in Sect. "Experiments on LGT data"

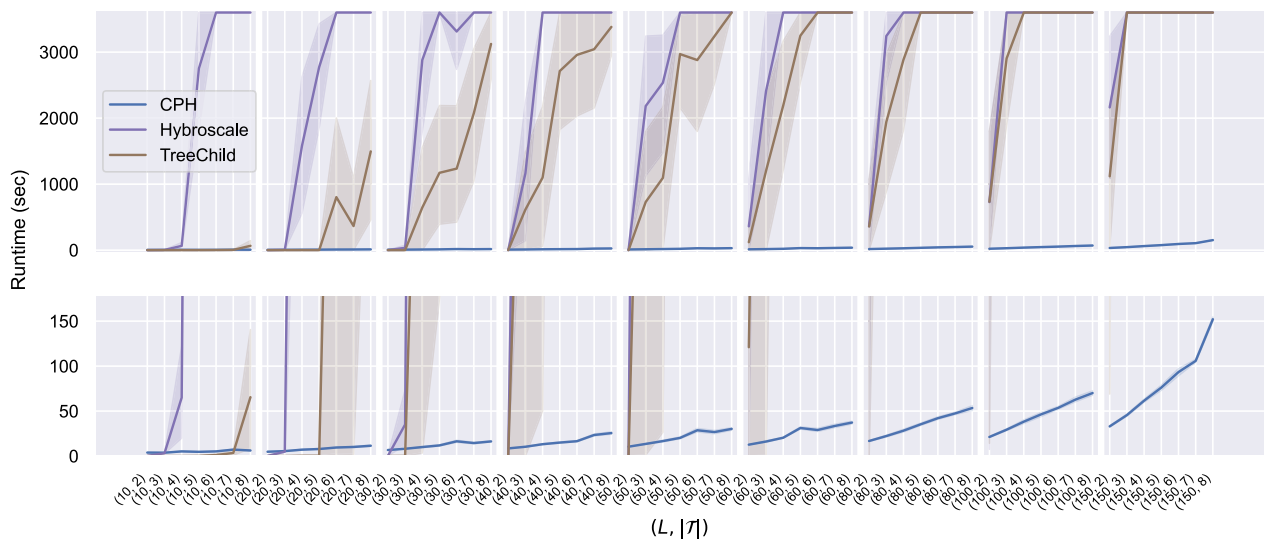


Fig. 13 The running time of ML on the real dataset described in Sect. "Experiments on real data" compared with the running time of the exact methods Hybroscale and TreeChild on the same dataset. The solid lines represent the average running times within each instance group. The shaded areas represent 95% confidence intervals

and removing a certain percentage p of leaves from each tree in each instance uniformly at random. Specifically, we generated an instance group for each value of $p \in \{5, 10, 15, 20, 25, 50\}$ starting from the LGT instance groups with $L = 100$ leaves and $R \in \{10, 20, 30\}$ reticulations. Since the performances of the two machine-learned heuristics were essentially overlapping for all of the other experiments, and since TrivialRand performed consistently better than the other randomised

heuristics, we limited this test to ML and TrivialRand. The results are shown in Fig. 14.

In accordance with intuition, the performance of both methods decreases with an increasing percentage of removed leaves, as the trees become progressively less informative. However, the degradation in the quality of the solutions is faster for ML than for TrivialRand, consistent with the fact that ML was trained on exhaustive trees only: when the difference between the training data and the input data becomes too large, the behaviour

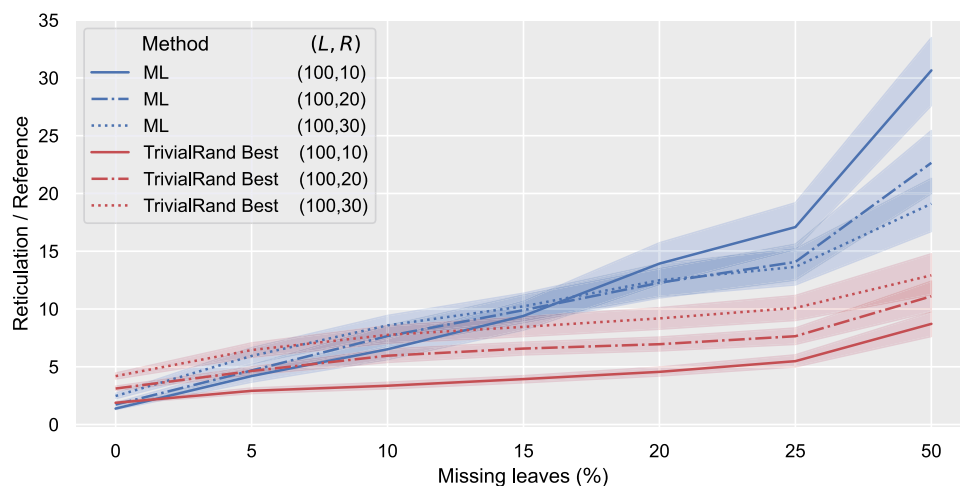


Fig. 14 Ratio between the number of reticulations outputted by ML and TrivialRand Best and the reference value for an increasing percentage of removed leaves on LGT data. Each point on the horizontal axis corresponds to a certain percentage of leaves removed from each tree; each line represents the average, within the instances of a group (L, R) with a certain percentage of removed leaves, of the output reticulation number divided by the reference value. The shaded areas represent 95% confidence intervals

of the machine-learned heuristic becomes unpredictable. We demand the design of algorithms better suited for trees with missing leaves for future work.

Effect of the threshold on ML

We tested the effectiveness of adding a threshold $\tau > 0$ to ML on the same datasets of Sects. "Experiments on normal data", "Experiments on LGT data" and "Experiments on ZODS data" (normal, LGT and ZODS). Recall that each instance group consists of 48 instances. We ran ML ten times for each threshold $\tau \in \{0, 0.1, 0.3, 0.5, 0.7\}$ on each instance, took the lowest output reticulation number and averaged these results within each instance group.

The results are shown in Fig. 15. For all types of data, a threshold $\tau \leq 0.3$ is beneficial, intuitively indicating that

when the probability of a pair being reducible is small it gives no meaningful indication, and thus random choices among these pairs are more suited. The seemingly best value for the threshold, though, is different for different types of instances. The normal instances seem to benefit from quite high values of τ , the best among the tested values being $\tau = 0.7$. While the optimal τ value for normal instances could be even higher, we know from Figure 7 that it must be $\tau < 1$, as the random strategies are less effective than the one based on machine learning for normal data. For the LGT and the ZODS instances, the best threshold seems to be around $\tau = 0.3$, while very high values ($\tau = 0.7$) are counterproductive. This is especially true for the LGT instances, consistent with the randomised heuristics being less effective for them than for the other types of data (see Fig. 8).

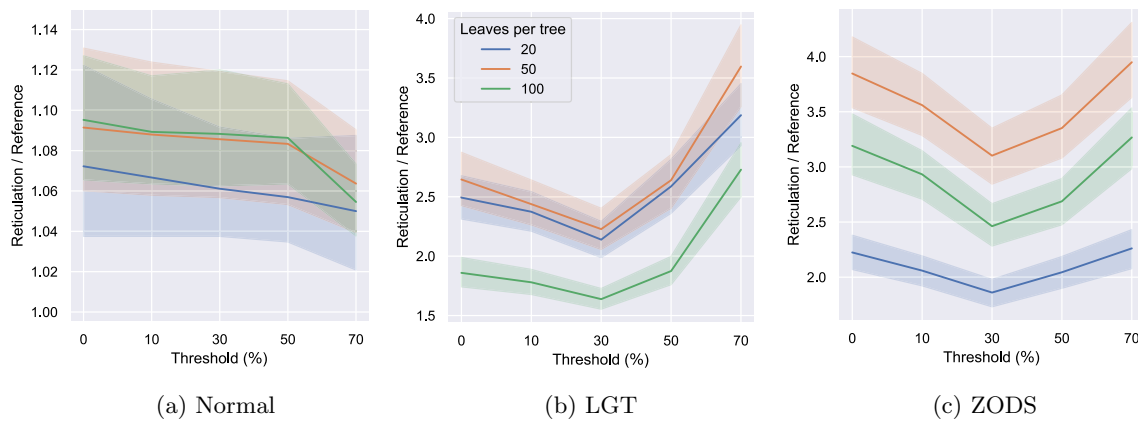


Fig. 15 The reticulation number when running ML with different thresholds on the instance groups of Sects. "Experiments on normal data", "Experiments on LGT data" and "Experiments on ZODS data". Each instance was run 10 times, and the lowest reticulation value of these runs was selected. The shaded areas represent 95% confidence intervals

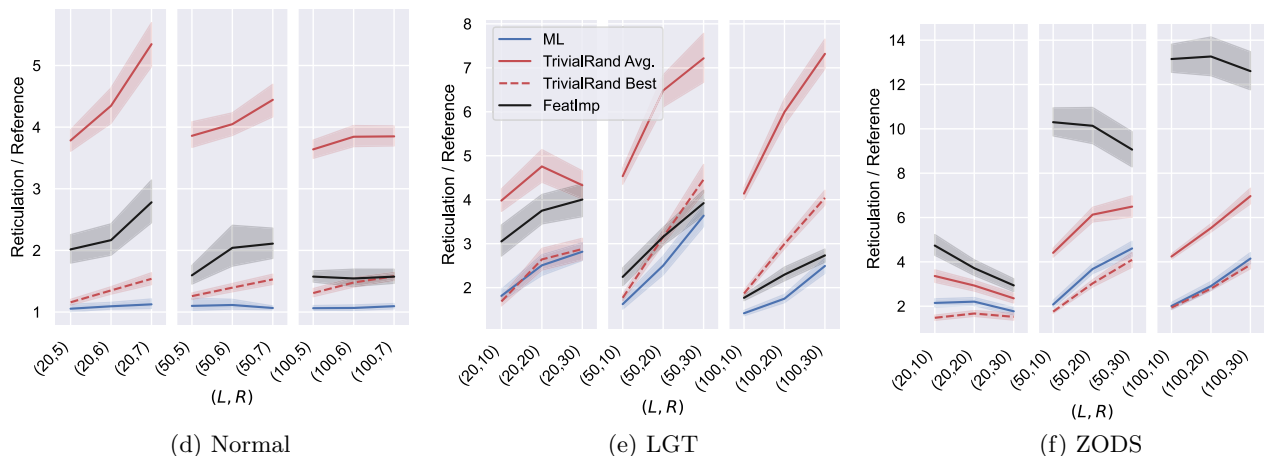


Fig. 16 Comparison of the results of FeatImp, ML and TrivialRand on the instance groups described in Sects. "Experiments on normal data", "Experiments on LGT data" and "Experiments on real data". Each point on the horizontal axis corresponds to an instance group; each line represents the average, within the instance group, of the output reticulation number divided by the reference value. The shaded areas represent 95% confidence intervals

These experiments should be seen as an indication that introducing some randomness may improve the performance of the ML heuristics, at the price of running them multiple times. We defer a more thorough analysis to future work.

A non-learned heuristic based on important features

In this section we propose *FeatImp*, yet another heuristic in the CPH framework. Although *FeatImp* does not rely on a machine learning model, we defined the rules to choose a cherry on the basis of the features that were found to be the most relevant according to the model we used for ML and *TrivialML*.

To identify the most suitable rules, we trained a classification tree using the same features and training data as the ones used for the ML heuristic (see Fig. 17 in Appendix A). We then selected the most relevant features used in such tree and used them to define the function *PickNext* listed by Algorithm 3: namely, the features 4 , 8_t , 11_d and 12_t of Table 1 (the ratio of trees having both leaves x and y in which (x, y) is reducible, the average of the topological leaf distance between x and y scaled by the depth of the trees, the average of the ratios $d(x, \text{LCA}(x, y))/d(y, \text{LCA}(x, y))$ and the average of the topological distance from x to the root over the topological distance from y to the root, respectively).

To compute and update these quantities we proceed as described in Sect. "Time complexity" and Appendix A. The general idea of the function *PickNext* used in *FeatImp* is to mimic the first splits of the classification tree by progressively discarding the candidate reducible pairs that are not among the top $\alpha\%$ scoring for each of the considered features, for some input parameter α .

Algorithm 3 Function *PickNext* used in *FeatImp*

INPUT: A set \mathcal{T} of phylogenetic trees and a parameter $\alpha \in (0, 100)$.

OUTPUT: Next cherry to pick (x, y)

```

1: if there exists a trivial cherry then
2:   Select a trivial cherry  $(x, y)$  uniformly at random;
3: else
4:    $C \leftarrow$  all reducible pairs of  $\mathcal{T}$ ;
5:    $C \leftarrow$  the  $\alpha\%$  cherries of  $C$  with the highest value for feature 4;
6:    $C \leftarrow$  the  $\alpha\%$  cherries of  $C$  with the highest value for feature  $8_t$ ;
7:    $C \leftarrow$  the  $\alpha\%$  cherries of  $C$  with the highest value for feature  $11_d$ ;
8:    $(x, y) \leftarrow$  the pair of  $C$  with the highest value for feature  $12_t$ ;
9: return  $(x, y)$ ;

```

We implemented *FeatImp* and test it on the same instances as Sects. "Experiments on normal data", "Experiments on LGT data" and "Experiments on ZODS

data" with $\alpha = 20$. The results are shown in Figure 16. As expected, *FeatImp* works consistently worse than ML on all the tested datasets, and it also performs worse than *TrivialRand* on most instance groups. However, it is on average 12% better than *TrivialRand* on the LGT instance group having 50 leaves and 30 reticulations and on all the LGT instance groups with 100 leaves, which are the most difficult for the randomised heuristics, as already noticed in Sect. "Experiments on LGT data". The results it provides for such difficult instances are only on average 20% worse than those of ML, with the advantage of not having to train a model to apply the heuristic.

These experiments are not intended to be exhaustive, but should rather be seen as an indication that machine learning can be used as a guide to design smarter non-learned heuristics. Possible improvements of *FeatImp* include using different values of α for different features, introducing some randomness in Line 8, that is, instead of choosing the single top scoring pair to choose one among the top $\alpha\%$ at random, or to use fewer/more features.

Conclusions

Our contributions are twofold: first, we presented the first methods that allow reconstructing a phylogenetic network from a large set of large binary phylogenetic trees. Second, we show the promise and the limitation of the use of machine learning in this context. Our experimental studies indicate that machine-learned strategies, consistent with intuition, are very effective when the training data have a structure similar enough to the test data. In this case, the results we obtained with machine learning were the best among all the tested methods, and the advantage is particularly evident in the most difficult instances. Furthermore, preliminary experiments indicate that the performance of the machine-learned methods can even be improved by introducing appropriate thresholds, in fact mediating between random choices and predictions. However, when the training data do not sufficiently reflect the structure of the test data, repeated runs of the fast randomised heuristics lead to better results. The non-learned cherry-picking heuristic we designed based on the most relevant features of the input (identified using machine learning) shows yet another interesting direction.

Our results suggest many interesting directions for future work. First of all, we have seen that machine learning is an extremely promising tool for this problem since it can identify cherries and reticulated cherries of a network, from displayed trees, with very high accuracy. It would be interesting to prove a relationship between the

machine-learned models' accuracy and the produced networks' quality. In addition, do there exist algorithms that exploit the high accuracy of the machine-learned models even better? Could other machine learning methods than random forests, or more training data, lead to even better results? Our methods are applicable to trees with missing leaves but perform well only if the percentage of missing leaves is small. Can modified sets of features be defined that are more suitable for input trees with many missing leaves? Moreover, we have seen that combining randomness with machine learning can lead to better results than either individual approach. However, we considered only one strategy to achieve this. What are the best strategies for combining randomness with machine learning for this, and other, problems? From a practical point of view, it is important to investigate whether our methods can be extended to deal with nonbinary input trees and to develop efficient implementations: in fact, we point out that our current implementations are in Python and not optimised for speed. Faster implementations could make machine-learned heuristics with nonzero thresholds even more effective. Finally, can the machine-learning-based approach be adapted to other problems in the phylogenetic networks research field?

Appendix A: time complexity

Lemma 7 *Updating features 1–5 for a set \mathcal{T} of $|\mathcal{T}|$ trees of total size $||\mathcal{T}||$ over a set of taxa X requires $\mathcal{O}(|\mathcal{T}|(|\mathcal{T}| + |X|^2))$ total time and $\mathcal{O}(|\mathcal{T}|)$ space.*

Proof Let $F_{(x,y)}^i$ denote the current value of the i -th feature for a cherry (x, y) . When reducing a cherry (x, y) in a tree T (thus deleting x and $p(x) = p(y)$ and then adding a direct edge from $p(p(y))$ to y), we check whether the other child of $p(p(y))$ is a leaf z or not. If not, no new cherry is created in T , thus the features 1–4 remain unaffected for all the cherries of \mathcal{T} . Otherwise, (z, y) and (y, z) are new cherries of T and we can distinguish two cases.

- 1 (z, y) and (y, z) are already cherries of \mathcal{T} . Then, $F_{(y,z)}^1$ and $F_{(z,y)}^1$ are increased by $\frac{1}{|\mathcal{T}|}$; $F_{(y,z)}^4$ and $F_{(z,y)}^4$ are increased by $\frac{1}{|\mathcal{T}^{y,z}|}$, where $|\mathcal{T}^{y,z}|$ is the number of trees that contain both y and z and is equal to $|\mathcal{T}|F_{(y,z)}^5$. To update features 2 and 3 we use two auxiliary data structures $\text{new_cherries}_{(y,z)}$ and $\text{new_cherries}_{(z,y)}$ to collect the distinct cherries that would originate after picking (y, z) and (z, y) in each tree, respectively. These structures must allow efficient inser-

tions, membership queries, and iteration over the elements², and can be deleted before picking the next cherry in \mathcal{T} . If the other child of $p(p(z))$ is a leaf w , we add (z, w) and (w, z) to $\text{new_cherries}_{(y,z)}$ and (y, w) and (w, y) to $\text{new_cherries}_{(z,y)}$ (unless they are already present).

- 2 (z, y) and (y, z) are new cherries of \mathcal{T} . Then we insert them into cherryfeatures . We initially set $F_{(y,z)}^1 = F_{(z,y)}^1 = \frac{1}{|\mathcal{T}|}$, and for features 2–3 we create the same data structures as the previous case. To compute $F_{(y,z)}^5 = F_{(z,y)}^5$ we first compute $|\mathcal{T}^{y,z}|$ by checking whether y and z are both leaves of T for each $T \in \mathcal{T}$. Then we set $F_{(y,z)}^5 = F_{(z,y)}^5 = \frac{|\mathcal{T}^{y,z}|}{|\mathcal{T}|}$ and $F_{(y,z)}^4 = F_{(z,y)}^4 = \frac{1}{|\mathcal{T}^{y,z}|}$.

Once we have reduced (x, y) in all trees, we count the elements of each of the auxiliary data structures new_cherries and update features 2–3 of the corresponding cherries accordingly. Since picking a cherry can create up to two new cherries in each tree, and for each new cherry we add up to two elements to an auxiliary data structure, this step requires $\mathcal{O}(|\mathcal{T}|)$ time for each iteration.

Feature 5 must be updated for all the cherries corresponding to the unordered pairs $\{x, w\}$ with $w \neq y$. To do so, when we reduce (x, y) in a tree T we go over its leaves: for each leaf $w \neq y$ we decrease $F_{(x,w)}^5$ and $F_{(w,x)}^5$ by $\frac{1}{|\mathcal{T}|}$ (if (x, w) and (w, x) are currently cherries of \mathcal{T}). This requires $\mathcal{O}(|X|^2)$ total time per tree over all the iterations, because we scan the leaves of a tree only when we reduce a cherry in that tree. Computing feature 5 when new cherries of \mathcal{T} are created (case 2) requires constant time per tree per cherry. The total number of cherries created in \mathcal{T} over all the iterations cannot exceed $2|\mathcal{T}|$, thus the total time required to update feature 5 is $\mathcal{O}(|\mathcal{T}|(|\mathcal{T}| + |X|^2))$. We arrived at the following result. \square

Lemma 8 *The time complexity of ML and TrivialML is $\mathcal{O}(|\mathcal{T}|^2)$.*

Proof Recall that during the initialization phase, we store the depth of each node, both topological and with respect to the branch lengths, and we preprocess each tree to allow constant-time LCA queries. Note that reducing cherries in the trees does not affect the height of the nodes nor their ancestry relations, thus it suffices to preprocess the tree set only once at the beginning of the algorithm.

When we reduce a cherry (x, y) in a tree T , this may affect the depth of T as a consequence of the internal node $p(x)$ being deleted. We thus visit T to update its

² For example, hashables paired with lists.

depth (both topological and with the branch lengths), and after updating the depth of all trees, we update the maximum value over the whole set \mathcal{T} accordingly. In order to describe how to update the features $6_{d,t} - 12_{d,t}$ we denote by $\text{old_depth}^t(T)$ the topological depth of T before reducing (x, y) , $\text{new_depth}^t(T)$ its depth after reducing (x, y) , and use analogous notation for the distances old_dist^t and new_dist^t between two nodes of a tree and for the depth, the max depth, and distances with the branch lengths.

Whenever the value of the maximum topological depth changes, we update the value of feature 6_t for all the current cherries (z, w) as $F_{(z,w)}^{6_t} = \frac{F_{(z,w)}^{6_t} \cdot \text{old_max_depth}^t}{\text{new_max_depth}^t}$. Since the maximum topological depth can change $\mathcal{O}(|X|)$ times over all the iterations, and the total number of cherries at any moment is $\mathcal{O}(|\mathcal{T}||X|)$, these updates require $\mathcal{O}(|\mathcal{T}||X|^2)$ total time. We do the same for feature 6_d , but since the maximum branch-length depth can change once per iteration in the worst case, this requires $\mathcal{O}(|\mathcal{T}|^2)$ time overall.

Features $8_{d,t} - 12_{d,t}$ must be then updated to remove the contribution of T for the cherries (x, w) and (w, x) for each leaf $w \neq x \neq y$ of T , because x and w will no longer appear together in T . These updates require $\mathcal{O}(1)$ time per leaf and can be done as follows. We set

$$F_{(x,w)}^{8_t} = \frac{F_{(x,w)}^{8_t} \cdot |\mathcal{T}^{x,w}| - \frac{\text{old_dist}^t(x,w)}{\text{old_depth}^t(T)}}{|\mathcal{T}^{x,w}| - 1} \quad (1)$$

and use analogous formulas to update $F_{(x,w)}^{8_d}$ and features $9_{d,t} - 12_{d,t}$ for (x, w) and (w, x) .

We finally need to further update all the features $6_{d,t} - 12_{d,t}$ for all the cherries of a tree T in which (x, y) has been reduced and whose depth has changed, including the newly created ones. This can be done in $\mathcal{O}(1)$ time per cherry per tree with opportune formulas of the form of Equation 1. We have obtained the stated bound. \square

Appendix B: random forest models

See Fig. 17, Tables 3 and 4.

(See figure on next page.)

Fig. 17 Classification tree with depth 4 of (a) the normal data set and (b) the LGT data set. For each node in the trees, except for the terminal ones, the first line is the feature condition. If this condition is met by a data point, it traverses to the left child node, otherwise to the right one. In the terminal nodes this line is omitted as there is no condition given. In each node, as also indicated with labels in the root node, the second line ‘samples’ is the proportional number of samples that follow the YES/NO conditions from the root to the parent of that node during the training process. The ‘value’ list gives the proportion of data points in each class, compared to the sample of that node. The last line indicates the most dominant class of that node. If a data point reaches a terminal node, the observation will be classified as the indicated class

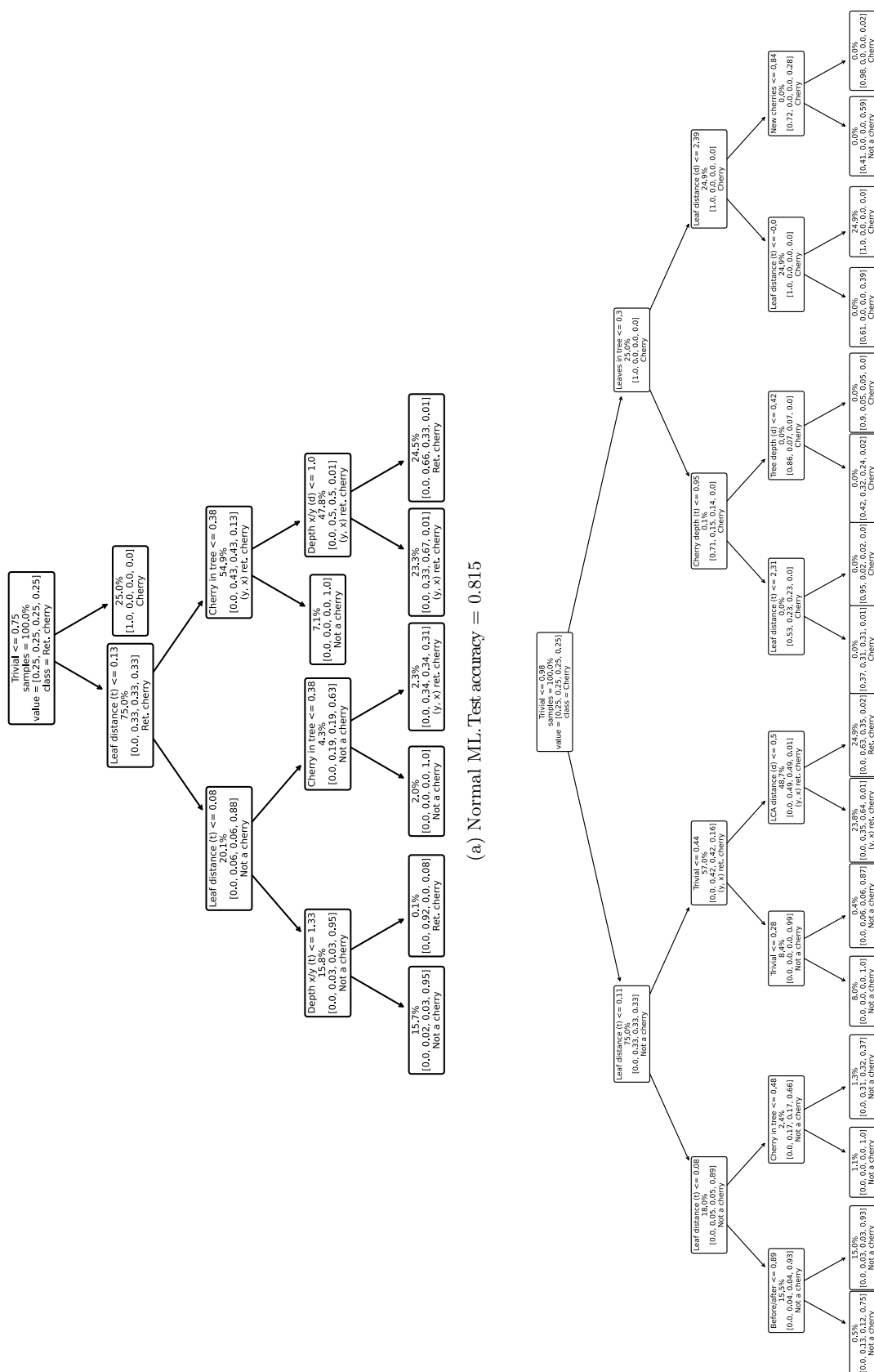


Fig. 17 (See legend on previous page.)

(b) LGTML and Test accuracy

Table 3 Trained random forest models on different datasets for different combinations of max L (maximum number of leaves per network) and M (number of networks)

max L	M	Accuracy	Num. data	Training (min)	Data gen. (hour/core)
<i>(a) Normal</i>					
20	5	1.0	840	00:00	00:00:12
	10	0.994	1804	00:00	00:00:22
	100	0.998	17,388	00:03	00:04:19
	500	0.994	73,168	00:16	00:15:18
	1000	0.993	151,308	00:42	00:29:49
50	5	0.994	3580	00:00	00:01:21
	10	0.997	7860	00:01	00:02:22
	100	0.996	53,988	00:11	00:18:07
	500	0.997	268,552	01:04	01:31:18
	1000	0.998	535,624	04:01	02:56:21
100	5	1.0	4944	00:00	00:01:13
	10	0.999	12,444	00:01	00:04:05
	100	0.999	128,824	00:25	00:41:54
	500	0.999	676,768	04:21	04:15:49
	1000	0.999	1,362,220	12:10	08:08:58
max L	M	Accuracy	Num. data	Training (min)	Data gen. (hour/core)
<i>(b) LGT</i>					
20	5	0.974	768	00:01	00:00:19
	10	0.994	1548	00:02	00:00:41
	100	0.976	12,244	00:09	00:04:20
	500	0.975	58,900	00:24	00:19:13
	1000	0.975	118,104	00:27	00:35:38
50	5	0.997	2952	00:01	00:00:43
	10	0.995	3796	00:03	00:01:01
	100	0.995	44,116	00:23	00:14:01
	500	0.994	219,472	01:39	01:06:45
	1000	0.994	421,204	02:45	02:10:45
100	5	0.996	5080	00:06	00:01:23
	10	0.996	7540	00:05	00:01:58
	100	0.998	114,900	00:31	00:34:25
	500	0.998	605,652	04:44	02:54:15
	1000	0.998	1,175,628	10:23	05:31:13

Each row in the table represents one model. For each model, the testing accuracy is given under "Accuracy", and the total number of data points retrieved from all M networks is given under "Num. data". Each dataset is split for training and testing (90 – 10%). The training duration for the random forest is given in column "Training" and the time needed to generate the training data is given in column "Data gen.", in hours per core (we used 16 cores in total)

Table 4 Feature importances of random forest trained on the biggest dataset ($M = 1000$ and $\max L = 100$) based on normal (a) and LGT (b) network data

Features	Importance	
	(a) Normal	(b) LGT
Leaf distance (t)	0.190	0.162
Trivial	0.155	0.184
Cherry in tree	0.143	0.146
Leaf distance (d)	0.122	0.114
LCA distance (t)	0.068	0.056
Depth x/y (t)	0.050	0.058
Cherry depth (t)	0.047	0.045
Depth x/y (d)	0.043	0.038
LCA distance (d)	0.028	0.032
Leaf depth x (t)	0.023	0.024
Leaf depth y (t)	0.023	0.023
Cherry depth (d)	0.020	0.023
Leaf depth x (d)	0.020	0.022
Leaf depth y (d)	0.020	0.022
Before/after	0.015	0.016
Tree depth (d)	0.012	0.013
Tree depth (t)	0.011	0.011
New cherries	0.006	0.006
Leaves in tree	0.004	0.003

Higher importance indicates that a feature has more effect on the trained model. The values sum up to one. The descriptions of the features are given in Table 1

Appendix C: heuristic performance of ML models

See Figs. 18, 19 and 20.

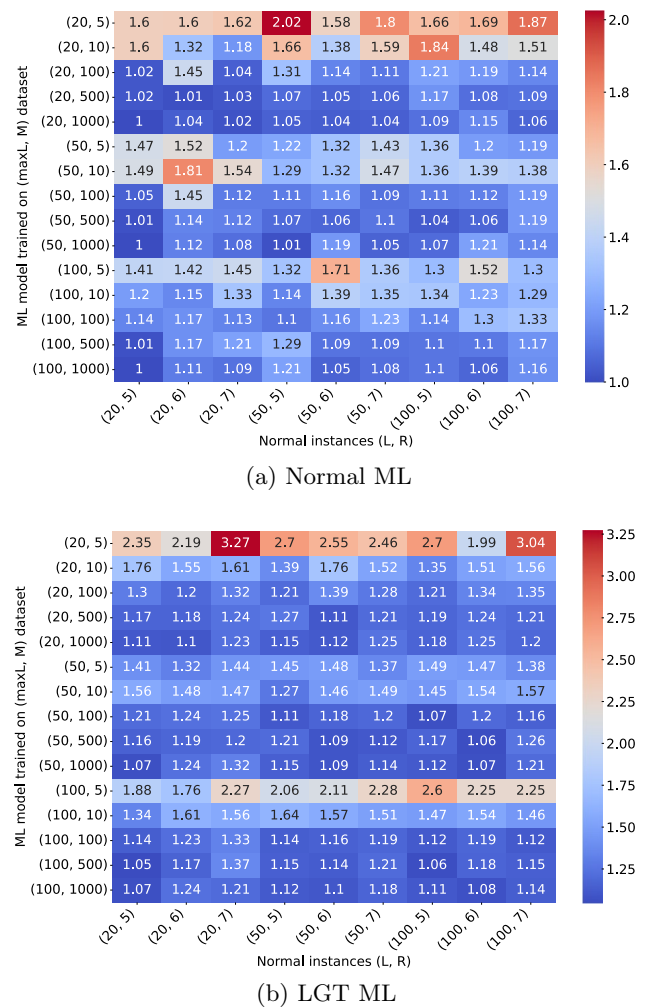


Fig. 18 Results for ML on normal instances with the random forest model trained on each of the datasets given in Table 3, where **a** gives the results when the ML model is trained on normal data, and **b** gives the results when the model is trained on LGT data. For each training dataset, identified by the parameter pair $(\max L, M)$, the value shown in the heatmap is the average, within each instance group, of the reticulation number found by ML divided by the reference value. We used a group of 16 instances for each combination of parameters $L \in \{20, 50, 100\}$ and $R \in \{5, 6, 7\}$

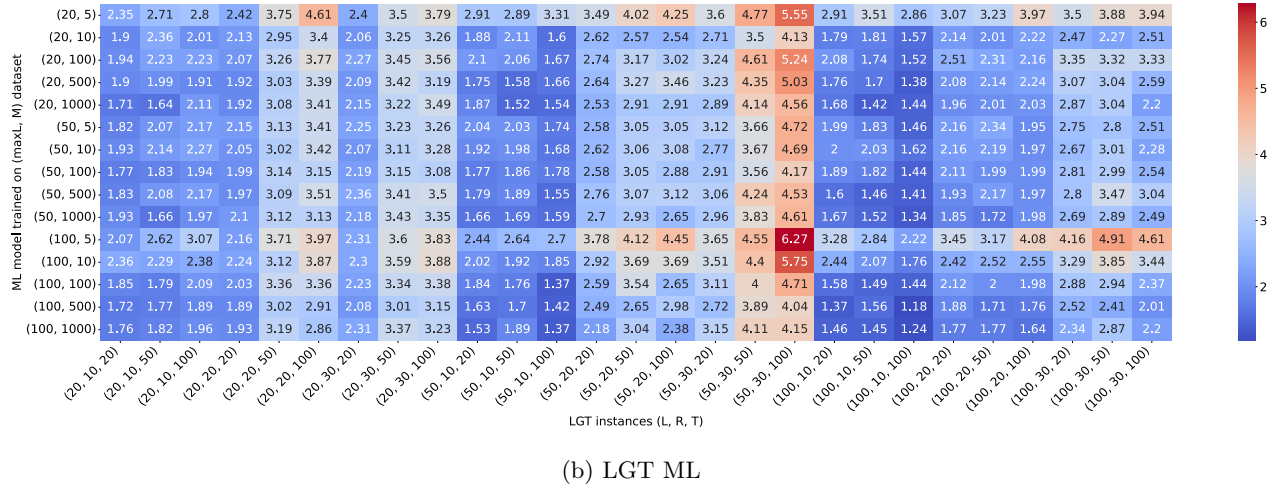
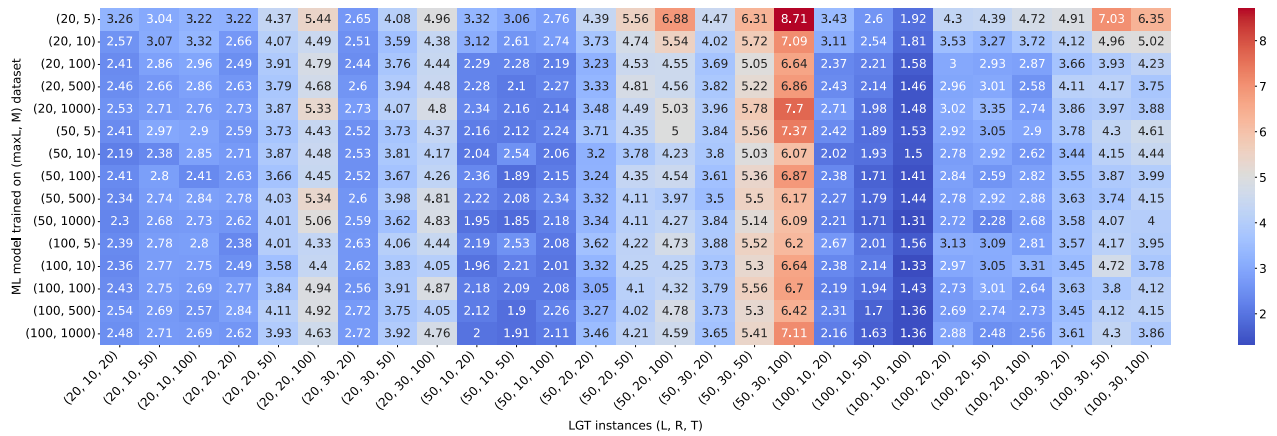
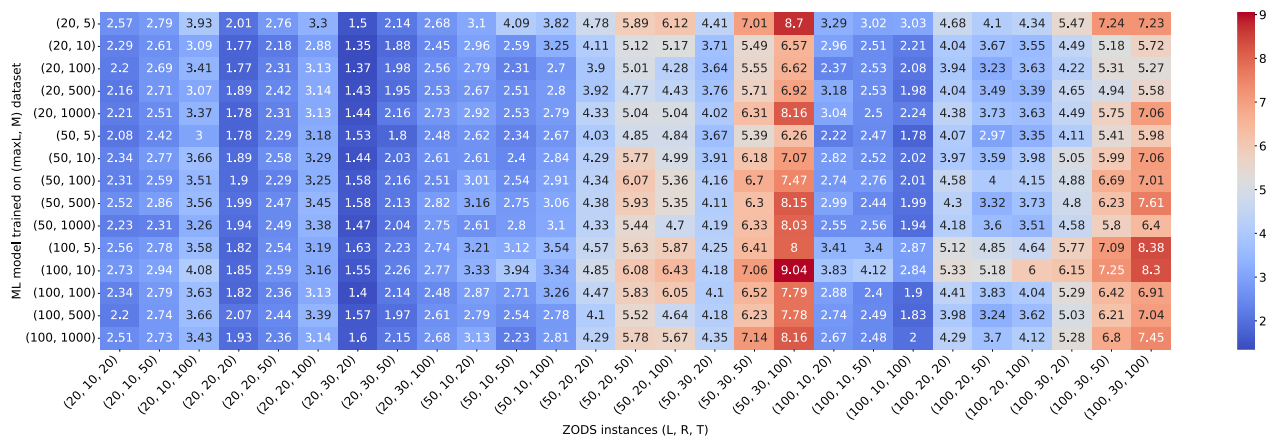
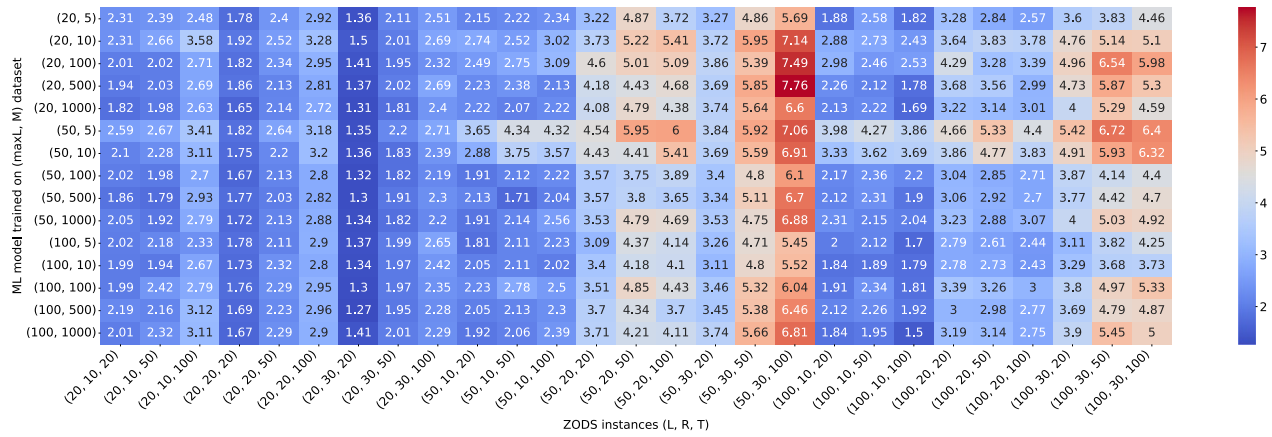


Fig. 19 Results for ML on LGT instances for different training datasets, similar to Fig. 18, with $L \in \{20, 50, 100\}$, $R \in \{10, 20, 30\}$ and $|\mathcal{T}| \in \{20, 50, 100\}$



(a) Normal ML



(b) LGT ML

Fig. 20 Results for ML on ZODS instances for different training datasets, similar to Fig. 18, with $L \in \{20, 50, 100\}$, $R \in \{10, 20, 30\}$ and $|T| \in \{20, 50, 100\}$

Acknowledgements

The authors thank Remie Janssen for providing ideas and preliminary code for the randomised heuristics, and Yukihiro Murakami for the inspiring discussions.

Funding

This paper received funding from the Netherlands Organisation for Scientific Research (NWO) under project OCENW.GROOT.2019.015 “Optimization for and with Machine Learning (OPTIMAL)”, from the MUR - FSE REACT EU - PON R & I 2014-2020 and from the PANGAIA and ALPACA projects that have received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

Availability of data and materials

The source code used in the experimental study of this article is available on <https://github.com/estherjulien/learn2cherrypick> and <https://doi.org/10.4121/c679cd3c-0815-4021-a727-bcb8b9174b27.v1>. This code is written in Python.

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 31 March 2023 Accepted: 10 June 2023

Published online: 16 September 2023

References

- Bordewich M, Semple C. Computing the minimum number of hybridization events for a consistent evolutionary history. *Discrete Appl Math*. 2007;155(8):914–28.
- Linz S, Semple C. Attaching leaves and picking cherries to characterise the hybridisation number for a set of phylogenies. *Adv Appl Math*. 2019;105:102–29.
- van Iersel L, Janssen R, Jones M, Murakami Y, Zeh N. A practical fixed-parameter algorithm for constructing tree-child networks from multiple binary trees. *Algorithmica*. 2022;84:917–60.
- Pardi F, Scornavacca C. Reconstructible phylogenetic networks: do not distinguish the indistinguishable. *PLoS Comput Biol*. 2015;11(4):1004135.
- Yu Y, Than C, Degnan JH, Nakhleh L. Coalescent histories on phylogenetic networks and detection of hybridization despite incomplete lineage sorting. *Syst Biol*. 2011;60(2):138–49.
- van Iersel L, Janssen R, Jones M, Murakami Y. Orchard networks are trees with additional horizontal arcs. *Bull Math Biol*. 2022;84(8):76.
- Albrecht B. Computing all hybridization networks for multiple binary phylogenetic input trees. *BMC Bioinform*. 2015;16(1):1–15.
- Wu Y. Close lower and upper bounds for the minimum reticulate network of multiple phylogenetic trees. *Bioinformatics*. 2010;26(12):140–8.
- Mirzaei S, Wu Y. Fast construction of near parsimonious hybridization networks for multiple phylogenetic trees. *IEEE/ACM Trans Comput Biol Bioinform*. 2015;13(3):565–70.
- Wen D, Yu Y, Zhu J, Nakhleh L. Inferring phylogenetic networks using phylonet. *Systematic biology*. 2018;67(4):735–40.
- Solis-Lemus C, Bastide P, Ané C. Phylonetworks: a package for phylogenetic networks. *Mol Biol Evol*. 2017;34(12):3292–8.
- Humphries PJ, Linz S, Semple C. Cherry picking: a characterization of the temporal hybridization number for a set of phylogenies. *Bull Math Biol*. 2013;75(10):1879–90.
- Borst S, van Iersel L, Jones M, Kelk S. New FPT algorithms for finding the temporal hybridization number for sets of phylogenetic trees. *Algorithmica*. 2022;84(7):2050–87.
- Semple C, Toft G. Trinets encode orchard phylogenetic networks. *J Math Biol*. 2021;83(3):1–20.
- Janssen R, Murakami Y. On cherry-picking and network containment. *Theor Comput Sci*. 2021;856:121–50.
- Azouri D, Abadi S, Mansour Y, Mayrose I, Pupko T. Harnessing machine learning to guide phylogenetic-tree search algorithms. *Nat Commun*. 2021;12(1):1–9.
- Zhu T, Cai Y. Applying neural network to reconstruction of phylogenetic tree. In: 2021 13th International Conference on Machine Learning and Computing. ICMCL 2021, pp. 146–152. Association for Computing Machinery, New York, NY, USA; 2021. <https://doi.org/10.1145/3457682.3457704>
- Kumar S, Sharma S. Evolutionary sparse learning for phylogenomics. *Mol Biol Evol*. 2021;38(11):4674–82.
- Bernardini G, van Iersel L, Julien E, Stougie L. Reconstructing phylogenetic networks via cherry picking and machine learning. In: 22nd International Workshop on Algorithms in Bioinformatics (WABI 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 242, pp. 16–1622. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Dagstuhl, Germany; 2022. <https://doi.org/10.4230/LIPIcs.WABI.2022.16>
- van Iersel L, Janssen R, Jones M, Murakami Y, Zeh N. A unifying characterization of tree-based networks and orchard networks using cherry covers. *Adv Appl Math*. 2021;129: 102222. <https://doi.org/10.1016/j.aam.2021.102222>.
- Harel D, Tarjan RE. Fast algorithms for finding nearest common ancestors. *SIAM J Comput*. 1984;13(2):338–55. <https://doi.org/10.1137/0213024>.
- Pons JC, Scornavacca C, Cardona G. Generation of level-k LGT networks. *IEEE/ACM Trans Comput Biol Bioinf*. 2019;17(1):158–64.
- Willson S. Regular networks can be uniquely constructed from their trees. *IEEE/ACM Trans Comput Biol Bioinf*. 2010;8(3):785–96.
- Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: Machine learning in Python. *J Mach Learn Res*. 2011;12:2825–30.
- Zhang C, Ogilvie HA, Drummond AJ, Stadler T. Bayesian inference of species networks from multilocus sequence data. *Mol Biol Evol*. 2018;35(2):504–17.
- Janssen R, Liu P. Comparing the topology of phylogenetic network generators. *J Bioinf Comput Biol*. 2021;19(06):2140012.
- Beiko RG. Telling the whole story in a 10,000-genome world. *Biol Direct*. 2011;6(1):1–36.
- Whidden C, Beiko RG, Zeh N. Fixed-parameter algorithms for maximum agreement forests. *SIAM J Comput*. 2013;42(4):1431–66. <https://doi.org/10.1137/110845045>.

Publisher’s Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ready to submit your research? Choose BMC and benefit from:

- fast, convenient online submission
- thorough peer review by experienced researchers in your field
- rapid publication on acceptance
- support for research data, including large and complex data types
- gold Open Access which fosters wider collaboration and increased citations
- maximum visibility for your research: over 100M website views per year

At BMC, research is always in progress.

Learn more biomedcentral.com/submissions

