Discrete Applied Mathematics 246 (2018) 62–79

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Solving the tree containment problem in linear time for nearly stable phylogenetic networks☆

Philippe Gambette [a,*], Andreas D.M. Gunawan [b], Anthony Labarre [a], Stéphane Vialette [a], Louxin Zhang [b]

[a] Université Paris-Est, LIGM (UMR 8049), UPEM, CNRS, ESIEE, ENPC, F-77454, Marne-la-Vallée, France
[b] Department of Mathematics, National University of Singapore, Singapore 119076, Singapore

ARTICLE INFO

Article history:
Received 15 February 2016
Received in revised form 1 June 2017
Accepted 17 July 2017
Available online 16 August 2017

Keywords:
Phylogenetic trees
Phylogenetic networks
Tree containment
Reticulation visibility
Nearly stable networks
Genetically stable networks

ABSTRACT

A phylogenetic network is a rooted acyclic digraph whose leaves are uniquely labeled with a set of taxa. The *tree containment problem* asks whether or not a phylogenetic network *displays* a phylogenetic tree over the same set of labeled leaves. It is a fundamental problem arising from validation of phylogenetic network models. The tree containment problem is NP-complete in general. To identify network classes on which the problem is polynomial time solvable, we introduce two classes of networks by generalizations of *tree-child networks* through vertex stability, namely nearly stable networks and genetically stable networks. Here, we study the combinatorial properties of these two classes of phylogenetic networks. We also develop a linear-time algorithm for solving the tree containment problem on binary nearly stable networks.

1. Introduction

With thousands of genomes being fully sequenced, phylogenetic networks have been adopted to study "horizontal" processes that transfer genetic material from a living organism to another without descendant relation. These processes are believed to be a driving force that shapes the genome of a species in evolution [6,25].

A (*phylogenetic*) *network* over a set X of taxa is a rooted acyclic digraph with a set of leaves (*i.e.*, vertices of outdegree zero) that are uniquely labeled with the taxa. Such a network represents the evolutionary history of the taxa in X. In a network, the *tree vertices* (*i.e.*, vertices of indegree one) represent speciation events, whereas the vertices of indegree at least two (called *reticulation vertices*, or simply *reticulations*) represent genetic material flow from several ancestral species into an "unrelated" species. A plethora of methods for reconstructing networks and related algorithmic issues have been extensively studied over the past two decades [16,17,23,24,27].

One approach to assessing the quality of a network is to verify whether or not it is consistent with previous biological knowledge about the species. Biologists therefore demand that the network displays existing gene trees, which corresponds to the *tree containment* (TC) problem [17]. This problem is well-known to be NP-complete [19,26], and great efforts have therefore been devoted to identifying "tractable" subclasses of binary networks, such as *galled trees* [19], *normal networks*, *tree-child networks* and *level-k networks* [26]. Recently, Gunawan et al. [12] and Bordewich and Semple [3] independently proved that the TC problem can be solved in cubic time for binary reticulation-visible networks, thereby settling an

☆ Parts of this work were presented in conferences RECOMB'2015 and IWOCA'2015.

* Corresponding author.
E-mail addresses: philippe.gambette@u-pem.fr (P. Gambette), a0054645@u.nus.edu (A.D.M. Gunawan), anthony.labarre@u-pem.fr (A. Labarre), stephane.vialette@u-pem.fr (S. Vialette), matzlx@nus.edu.sg (L. Zhang).

http://dx.doi.org/10.1016/j.dam.2017.07.015
0166-218X/

open problem [17,26]. The time complexity was further improved to quadratic time even for arbitrary (i.e. non-binary) reticulation-visible networks in [13].

To tackle the TC problem for reticulation-visible networks, we introduced *nearly stable networks* and *genetically stable networks*, which both generalize tree-child networks, and gave quadratic time algorithms for solving the TC problem on both classes [10,11]. These results gave an insight on the topological structure of a reticulation-visible network, and eventually led to a solution to the open problem in [12]. In this paper, we establish the tight upper bounds on the numbers of vertices in a nearly stable network and in a genetically stable network. We further show on simulated data that these two new classes cover a significant proportion of phylogenetic networks, compared with binary reticulation-visible networks.

In this paper, we also revisit the TC problem for nearly stable networks. In an earlier version of this work [10], we developed a quadratic-time algorithm for the TC problem on that class. The time complexity of the algorithm was further improved to $O(n \log n)$ in [8] using the same approach but with a more efficient data structure. Here, we develop a linear-time TC algorithm by combining the structure analysis in [10] and a decomposition technique introduced in [12]. This technique plays a vital role in the designs of a quadratic-time TC algorithm for reticulation-visible networks and of a fast exponential-time algorithm for arbitrary networks [14].

2. Concepts and notions

2.1. Phylogenetic trees and networks

A (*phylogenetic*) *network* on a set X of taxa is an acyclic digraph N with a single root ρ_N (a unique vertex with indegree 0) whose leaves (vertices with outdegree 0) are in one-to-one correspondence with the taxa in X. The fact that the root is the unique vertex with indegree 0 implies that there is a (directed) path from the root to every other vertex. For convenience, we attach an incoming edge to ρ_N (so the indegree becomes one) with one open-end. We identify each leaf with the taxon corresponding to it.

In a network, *reticulation vertices* (or simply *reticulations*) are vertices with indegree at least two and outdegree one; *tree vertices* are vertices with indegree one, which includes the root and leaves. For a given network N, $\mathcal{L}(N)$ denotes its leaf set, $\mathcal{R}(N)$ its set of reticulation vertices, $\mathcal{T}(N)$ its set of tree vertices (including leaves), $\mathcal{V}(N)$ the entire set of vertices, and $\mathcal{E}(N)$ the entire set of edges. An edge is a *reticulation edge* if it is an edge entering a reticulation vertex.

Let N be a network. For an edge set $E \subseteq \mathcal{E}(N)$, $N - E$ denotes the subnetwork with vertex set $\mathcal{V}(N)$ and edge set $\mathcal{E}(N) \setminus E$. Similarly, for a vertex subset $S \subseteq \mathcal{V}(N)$, $N - S$ denotes the subnetwork with the vertex set $\mathcal{V}(N) \setminus S$ and the edge set $\{(u, v) \in \mathcal{E}(N) \mid u \notin S, v \notin S\}$. When E or S contains only one element x, we simply write $N - x$.

Let $v \in \mathcal{V}(N)$. The *subnetwork of N induced by v* consists of v and all its descendants and the edges between them. It is rooted at v and is denoted by $N[v]$.

We say that a network obtained by removing all but one incoming edge from each reticulation in a network N is a *spanning tree* of N.

A network is *binary* if its leaves are of degree one, and all other vertices are of degree three. A *phylogenetic tree* is simply a binary network without reticulation vertices. All networks we shall consider in this paper are binary unless stated otherwise.

2.2. Reticulation stability

Consider a network N. Let x and y be two vertices of N. We say that x is a *parent* of y and y is a *child* of x if (x, y) is an edge of N. More generally, we say that $x \neq y$ is an *ancestor* of y (or *above* y) and equivalently y is a *descendant* of x (or *below* x) if there is a directed path from x to y in N. Two vertices are *siblings* if they are the children of the same vertex.

A vertex x is a *stable ancestor* of a vertex v (or x is *stable on* v) if it belongs to all directed paths from ρ_N to v. We say that x is *stable* (or *visible*) if there exists a leaf ℓ such that x is a stable ancestor of ℓ. A vertex is *unstable* if it is not a stable ancestor of any leaf. A network is *reticulation-visible* [17] if every reticulation vertex it contains is visible.

Proposition 2.1 (*Lemma 2.3, [15]*)**.** *Let N be a network and $u \in \mathcal{V}(N)$. Let R be a set of reticulations below u such that for each $r \in R$, either* (i) *r is a descendant of another reticulation $r' \in R$, or* (ii) *there is a path from ρ_N to r that avoids u. Then, u is not stable on any leaf ℓ below a reticulation in R.*

By Proposition 2.1, we have the following simple criteria to determine whether a vertex is stable or not in a network.

Corollary 2.1. *Let v be a tree vertex in a network N.*

(1) *If v has two reticulation children, then v is unstable.*
(2) *Assume v has two children u and w such that u is a tree vertex with two reticulation children and w is a reticulation vertex (Fig. 1). If w is different from the children of u, then v is unstable.*

Proposition 2.2. *The following facts hold for a network.*

(1) *A vertex u is stable if it has a stable tree vertex as a child.*
(2) *A reticulation u is stable if and only if its unique child is a stable tree vertex.*
(3) *If u and v are stable ancestors of w, then either u is an ancestor of v or vice versa.*

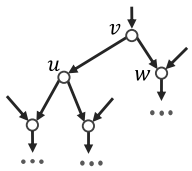

Fig. 1. Two types of unstable tree vertices in a network.

Proof. (1). Let v be a child of u. Assume it is a tree vertex and stable. Since v is stable, then it is a stable ancestor of some network leaf ℓ. Since v is a tree vertex, u is the unique parent of v. Taken together, both facts imply that all paths from ρ_N to ℓ must contain v and hence u. Therefore, u is also stable on ℓ.

(2). The sufficiency follows from (1). To prove the necessity, we assume that u is a reticulation stable on a leaf ℓ and let x denote its unique child. Since every path from ρ_N to ℓ goes through u, each such path must also go through x, which is therefore stable on ℓ. We now show by contradiction that x is a tree vertex: if x is a reticulation, it has another parent v different from u. For a path from ρ_N to ℓ passing u, it also passes x and so joining a path from ρ_N to v, the edge (v, x) and a path from x to ℓ gives a path from ρ_N to ℓ avoiding u, contradicting the fact that u is stable on ℓ. Therefore, x must be a tree vertex.

(3) The fact follows from the fact that both u and v must be within every path from ρ_N to w. □

We refer to leaves resulting from the removal of edges or vertices as *dummy leaves* (i.e. leaves that were not leaves in the original network). A network is *tree-based* if there is an edge set E that contains an incoming edge for each reticulation vertex such that $N - E$ is a spanning tree of N without dummy leaves [9]. In [10], we used the following result to establish the first upper bound on the size of reticulation-visible network. Although a better bound can be obtained using a different approach, this result is interesting in its own right and therefore presented below.

Theorem 2.1. *Every reticulation-visible network is tree-based.*

Proof. Let N be a reticulation-visible network and $E \subset \mathcal{E}(N)$. If E contains two edges coming out of the same tree vertex, there will be a dummy leaf in $N - E$. Similarly, if E contains two edges entering the same reticulation vertex, this reticulation vertex will become a vertex of indegree 0 in $N - E$. Therefore, $N - E$ is a spanning tree of N without dummy leaves if and only if E is a matching covering every reticulation vertex in N.

Since N is reticulation-visible, by the part (2) of Proposition 2.2, the parents of each reticulation vertex are tree vertices. The existence of such a matching can be found by applying Hall's Theorem to a bipartite graph with one part consisting of tree vertices and another consisting of reticulation vertices, and the edges being reticulation edges in N. Since each reticulation vertex has two incoming reticulation edges and each tree vertex has at most two outgoing reticulation edges, the existence of such a matching E follows from a theorem of Alon [2, page 429]. □

2.3. The tree containment problem

Let $(u, v) \in \mathcal{E}(N)$. The *contraction* of (u, v) in N transforms N into the network with the vertex set $\mathcal{V}(N) \setminus \{v\}$ and the following edge set:

$$\big\{(x, y) \in \mathcal{E}(N) \mid x \neq v \neq y\big\} \cup \big\{(x, u) \mid (x, v) \in \mathcal{E}(N) \setminus \{(u, v)\}\big\} \cup \big\{(u, y) \mid (v, y) \in \mathcal{E}(N)\big\}.$$

Given a phylogenetic tree G and a network N, we say that N *displays* G if there is a spanning tree T of N such that G is a *contraction* of T, i.e. T has the same vertex set as N and G can be obtained from T by repeatedly applying contraction to all edges entering either the "dummy" leaves or the vertices of both outdegree and indegree one, where a dummy leaf is a leaf in T but not a leaf in N. Fig. 2 shows an example of a network N and a phylogenetic tree that is displayed by N.

The *tree containment* (TC) problem is to determine whether a given phylogenetic network displays a given phylogenetic tree. We shall discuss the TC problem for network class defined using vertex stability, namely the nearly stable networks.

3. Nearly stable and genetically stable networks

3.1. Inclusion relationship

A network is *tree-child* if each of its internal vertices has a child that is a tree vertex [5]. A binary network is *tree-sibling* if every reticulation vertex has a sibling that is a tree vertex [4,22]. Cordue et al. [7] investigated reticulation-visible networks in which each reticulation has at least a parent p that is connected to some leaf by a path consisting of only tree vertices. Here, such networks are defined to be *nearly tree-child* networks.

Huson et al. [17, page 164] noted that if a network is tree-child, then all its vertices are stable. We strengthen their result by proving the other direction.

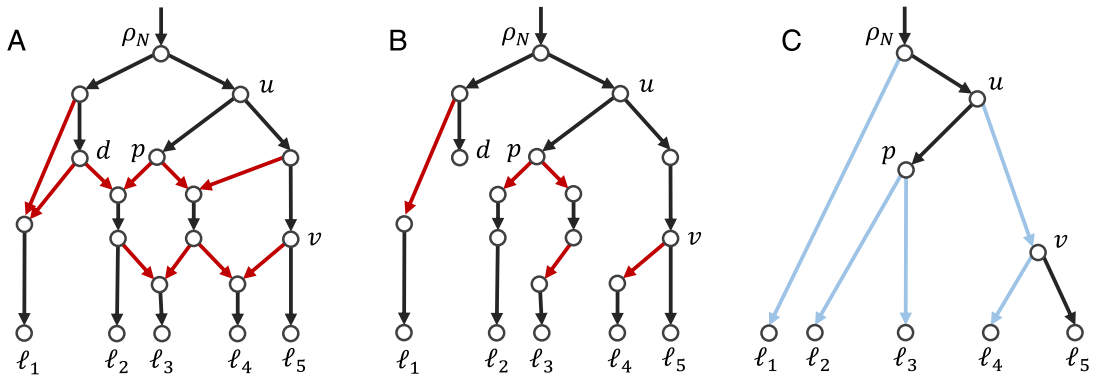

Fig. 2. (A) A network N with five leaves labeled with ℓ_i ($1 \leq i \leq 5$). Reticulation edges are colored dark red. (B) A spanning tree T obtained by removing a reticulation edge from each reticulation in the network, in which there is a dummy leaf d. (C) A tree that is displayed in N, which can be obtained from T by contraction. Each shallow blue edge corresponds to a path consisting of two or more edges in T. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

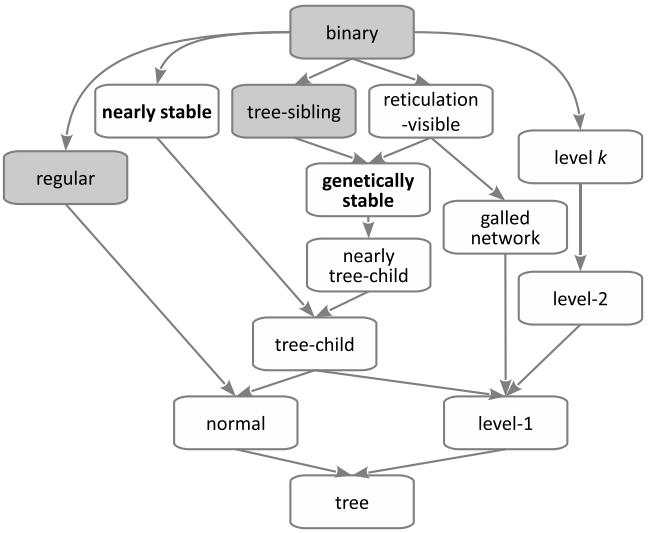

Fig. 3. Inclusion relationships between classes of binary phylogenetic networks: an arrow from class A to class B means that A contains B. Class inclusions involving nearly stable and genetically stable are justified in this article (Proposition 3.2); for the other ones, references are available at http://phylnet.info/isiphync/. The boxes of classes where the TC problem is NP-complete are colored gray, the ones where the TC problem is solvable in polynomial time are colored white.

Proposition 3.1. *A network is tree-child if and only if every vertex is stable in the network.*

Proof. The sufficiency follows from Proposition 2.2(2) and Corollary 2.1. □

To extend tree-child networks, we introduced two subclasses of phylogenetic networks. A network is *nearly stable* if for every vertex, either the vertex or its parents are stable [10]. It is *genetically stable* if every reticulation vertex is stable and has at least one stable parent [11].

Proposition 3.2.

(1) *Every genetically stable network is tree-sibling.*
(2) *Every tree-child network is nearly stable.*
(3) *Every nearly tree-child network is genetically stable.*

Proof. (2) and (3) follow from the definitions. For (1), let N be a genetically stable network and let $p \in \mathcal{R}(N)$. Since N is genetically stable, p has a stable parent p'. By Proposition 2.2(2), p' is a tree vertex, so it must have another child c. By Corollary 2.1, c is a tree vertex, and N is therefore tree-sibling. □

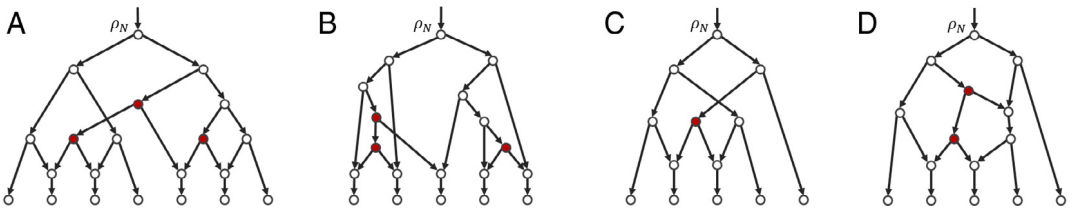

Fig. 4. (A) A network which is reticulation-visible and tree-sibling, but not genetically stable. (B) A network which is genetically stable but neither nearly stable nor nearly tree-child. (C) A network which is nearly tree-child and nearly stable but not tree-child. (D) A network which is nearly tree-child but not nearly stable. Here, the filled vertices are unstable.

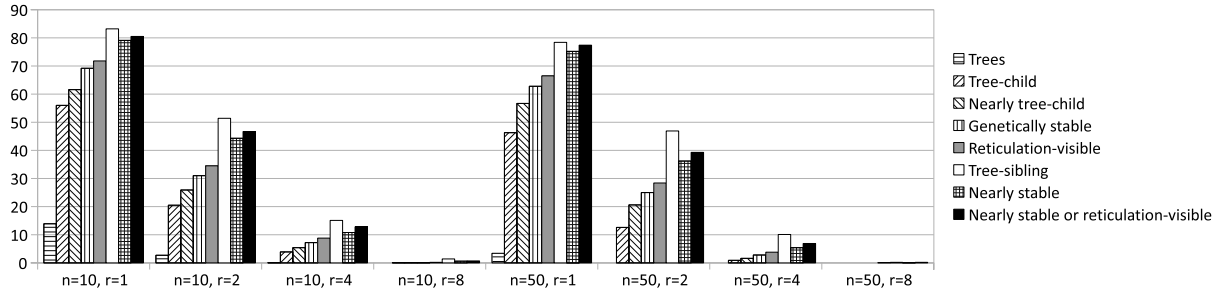

Fig. 5. Percentage of binary phylogenetic networks on n leaves generated with the coalescent with recombination model (recombination rate r) in each class of phylogenetic networks. The corresponding data is available at http://phylnet.info/recophync/.

Based on Proposition 3.2, we summarize the relationships between the classes we study and the other network classes for which the complexity of the TC problem is known in Fig. 3.

A reticulation-visible and tree-sibling network is not necessarily genetically stable (Fig. 4A). A genetically stable network is not necessarily nearly tree-child (Fig. 4B). A nearly stable and nearly tree-child network is not necessarily tree-child (Fig. 4C). A nearly tree-child network is not necessarily nearly stable (Fig. 4D).

3.2. Class sizes

Recombination histories of viruses, hybridization histories of plants, and histories of horizontal gene transfers reported in the literature are often found to be nearly stable or reticulation-visible (see e.g. the networks given in [18,21] which are available at http://phylnet.info/recophync/networkDraw.php).

In order to evaluate whether the class of nearly stable networks is relevant in practice, especially combined with the class of reticulation-visible networks for which there also exists a polynomial-time algorithm solving the TC problem, we used a set of phylogenetic networks randomly generated using a simulation program [1] and calculated the proportion of those networks belonging to the classes.[1] Fig. 5 summarizes the results of our simulation study.

This experiment shows that among networks generated with the coalescent with recombination model, the proportions of reticulation-visible and especially nearly stable networks are significantly larger than that of tree-child networks. Furthermore, the proportion of networks which are reticulation-visible or nearly stable is also significantly larger than the proportion of reticulation-visible networks.

4. How large can nearly stable and genetically stable networks be?

A network with n leaves may contain an arbitrary large number of non-leaf vertices and hence, unlike phylogenetic trees, its size is not bounded from above by a function of the number of leaves. In [10], we proved that a reticulation-visible network with n leaves has at most $10n-9$ vertices (including leaves). Later, the tight size bound $8n-7$ was proved [3]. In this section, using the decomposition theorem introduced below, we shall first give a short proof of this tight bound. We also show that nearly stable networks have the same tight size bound as reticulation-visible networks, whereas genetically stable networks with n leaves have the tight size bound $6n-5$.

[1] A Python script for class recognition as well as the data and the obtained results are available at http://phylnet.info/recophync/, and an online demo is provided at http://phylnet.info/tools/.

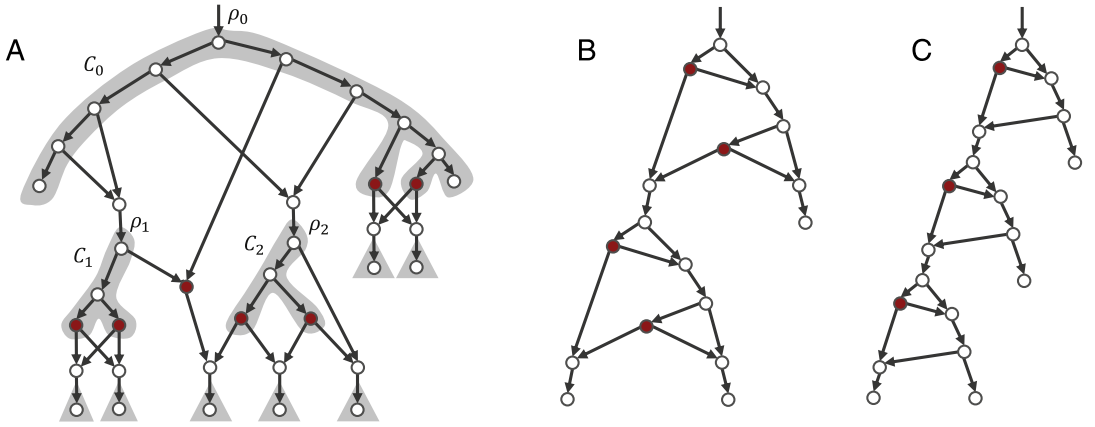

Fig. 6. (A) The decomposition of a nearly stable network into ten tree vertex components. Seven components consist of a single leaf. The remaining three "big" components are C_0, C_1, and C_2, where ρ_i denotes the root of C_i for each $i = 1, 2, 3$. (B) A network with 3 leaves and 6 reticulations that is both nearly stable and reticulation-visible. (C) A genetically stable network with 4 leaves and 6 reticulations. Here the filled vertices are unstable ones.

4.1. A decomposition theorem

Consider a network N. After the removal of all reticulations, N becomes a forest $N-\mathcal{R}(N)$. One connected component of the forest is rooted at ρ_N, whereas the other components are each rooted at the child of a reticulation (Fig. 6). Since components consist of tree vertices of N, they are called *tree vertex components* of N. We call them *big* tree vertex components if they contain more than one vertex.

A reticulation is said to be *inner* if its two parents belong to the same tree vertex components. Otherwise, it is said to be *cross*.

We now prove the following decomposition theorem for nearly stable networks, which is similar to Theorem 1 of [13] for reticulation-visible networks.

Theorem 4.1. *Let N be a nearly stable or reticulation-visible network with tree vertex components $C_0, C_1, C_2, \ldots, C_r$. Then,*

(1) *Each component C_j is rooted at a stable tree vertex. Additionally, a vertex is a component root if and only if it is either the network root or the child of a stable reticulation.*
(2) *Each component C_j contains either a network leaf or the two parents of an inner reticulation.*
(3) *Each component C_j contains at least two tree vertices if $C_j \neq \{\ell\}$ for any leaf $\ell \in \mathcal{L}(N)$.*

Proof. (1) By definition of a tree vertex component, its root r must be a tree vertex. If r is the root of N, then it is obviously stable. Otherwise, as r is a tree vertex, then its parent p is a reticulation (otherwise r is not a component root). If p is unstable, then r must be stable as N is a nearly stable network, but this contradicts Proposition 2.2(1). Therefore, p is a stable reticulation vertex and r is a stable tree vertex according to the part (2) of Proposition 2.2. Clearly, every child of a stable reticulation is a tree vertex, and thus is a component root as well.

(2) For a component C_j that does not contain a network leaf, each leaf of C_j is a parent of some reticulation below C_j. If for any reticulation below C_j not all parents are in C_j, then, the set of the reticulations satisfies the conditions in Proposition 2.1 and hence the root of C_j is not stable, contradicting (1).

(3) Suppose for contradiction that C_j contains exactly one tree vertex v and v is not a leaf. Then both children of v are reticulation vertices, and so v is unstable according to Corollary 2.1(1), contradicting the condition (1) above (i.e. v is stable). □

For a tree vertex component C of N, we denote its root by ρ_C. A tree vertex component C is *below* another component D, if there is a path from ρ_D to ρ_C in N. A *lowest* big tree vertex component is a component such that every other component below it contains exactly a single leaf. Such component is always guaranteed to exist, see [12].

4.2. Three size bounds

Let N be a network. In this subsection, we use r and t to denote the number of reticulations and non-leaf tree vertices in N.

Theorem 4.2. (i) [3] *If N is reticulation-visible, then $r \leq 3(n-1)$.*
(ii) *If N is reticulation-visible and tree-sibling, $r \leq 2(n-1)$. In particular, the bound holds for genetically stable networks.*
(iii.) *If N is nearly stable, then $r \leq 3(n-1)$.*

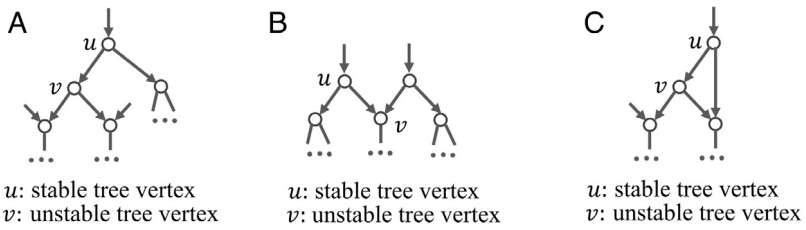

Fig. 7. Illustration of three types of tree vertices in a nearly stable network. (A) An unstable tree vertex v with two reticulation children and a stable tree vertex u with two tree children. (B) A stable tree vertex u for which a child is an unstable reticulation vertex. (C) A stable tree vertex u that has a common reticulation child with its tree vertex child.

Proof. (i) Ignoring the open-edge attached to the root, the network root is of indegree 0 and outdegree 2, the other tree vertices are of indegree 1 and outdegree 2 if they are not leaves, and each reticulation vertex is of indegree 2 and outdegree 1. By the handshaking lemma, $2t + r = t - 1 + 2r + n$, which is further simplified into:

$$t = r + n - 1. \tag{1}$$

Additionally, we let c be the number of tree vertex components of N for the rest of the proof. By Theorem 4.1(1), $c = r+1$.

Consider a component C. Since N is reticulation-visible, by Theorem 4.1(3), C contains two distinct parents of an inner reticulation if it does not contain a network leaf. Hence,

$$r + 1 = c \leq n + t/2. \tag{2}$$

Replacing t with $r + n - 1$ in Eq. (2), we obtain that $r \leq 3(n - 1)$.

(ii) Assume N is reticulation-visible and tree-sibling. We distinguish three types of tree vertices of N by using T_i to denote the set of tree vertices with exactly i children being also tree vertices for $i = 0, 1, 2$, respectively.

Since N is tree-sibling, each reticulation vertex x has a parent v_x such that v_x is a tree vertex and has x and another tree vertex as its children. Therefore, mapping x to v_x is an injective map from $\mathcal{R}(N)$ to T_1 and thus:

$$r \leq |T_1|. \tag{3}$$

Consider a tree vertex component C of N that does not contain any network leaf. By fact (3) of Theorem 4.1, C contains a leaf v that differs from ρ_C. The children of v must be reticulation vertices, and hence $v \in T_0$ (Fig. 7A).

Since there are at most n tree vertex components that contain one or more network leaves,

$$r + 1 = c \leq |T_0| + n. \tag{4}$$

Combining Inequalities (3) and (4) with Eq. (1), we have:

$$2r + 1 \leq |T_1| + |T_0| + n \leq t + n = r + 2n - 1,$$

and thus $r \leq 2(n - 1)$.

(iii) Let N be a nearly stable network. Given that each reticulation vertex may or may not be stable, we use r_s and r_u to denote the numbers of stable and unstable reticulation vertices of N, respectively.

First, for an unstable reticulation vertex, its parents are both stable tree vertices, as N is nearly stable. By Corollary 2.1, a tree vertex with two reticulation children is unstable. This implies that any two different unstable reticulation vertices have distinct parents. Therefore, there are at least $2r_u$ stable tree vertices that have an unstable reticulation child (Fig. 7B).

Secondly, for a tree vertex component C not containing any network leaf, its root is a stable tree vertex. By the fact (3) of Theorem 4.1, C contains a leaf v that differs from the root. The children of v are both reticulation vertices and thus, by Corollary 2.1, v is unstable. This implies that:

$$c - n \leq t_u, \tag{5}$$

as there are at most n components that contain at least a network leaf. Additionally, since N is nearly stable, the parent u of v has to be a stable tree vertex. Since v has two reticulation children, by Corollary 2.1, the stability of u implies that the other child of u has to be either a tree vertex (Fig. 7A) or a child of v (Fig. 7C). Since there are at least $c - n$ tree vertex components that do not contain any network leaf, there are at least $c - n$ stable tree vertices such that their children are either two tree vertices or a tree vertex and one (stable) reticulation vertex.

In summary, we have shown that (i) N contains at least $2r_u$ stable tree vertices that have an unstable reticulation child and (ii) there are at least $c - n$ stable tree vertices that have no unstable reticulation child. Hence,

$$t_s \geq (c - n) + 2r_u.$$

By Inequality (5),

$$t = t_u + t_s \geq 2(c - n) + 2r_u.$$

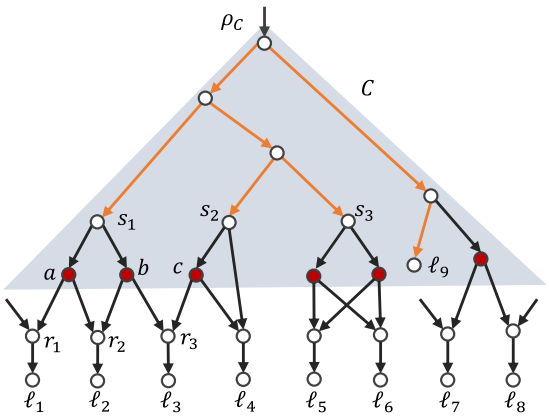

Fig. 8. Illustration of the idea of our algorithm. A tree-vertex component C of a nearly stable network, where filled vertices are unstable ones and open edges have their end in other components. The orange subtree is the subtree spanned by the stable vertices with four leaves s_1, s_2, s_3, and ℓ_9. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Since each tree vertex component is rooted at either the network root or the child of a stable reticulation vertex, then, $c = r_s + 1$. Replacing c with $r_s + 1$ and $r_s + r_u$ with r in the last inequality, we have:

$$t \geq 2r + 2 - 2n.$$

This inequality and Eq. (1) imply that $3(n-1) \geq r$. This completes the proof. □

Fig. 6B gives a nearly stable and reticulation-visible network with 3 leaves and 6 reticulation vertices. Fig. 6C gives a genetically stable network with 4 leaves and 6 reticulations. Hence, the second and third bounds in Theorem 4.2 are also tight.

Finally, it is not hard to see that in nearly tree-child network, each tree-vertex component contains at least a network leaf and thus we have $r = c - 1 \leq n - 1$.

5. A linear-time TC algorithm for nearly stable networks

In this section, we present a recursive linear-time algorithm for TC on nearly stable networks.

5.1. Stable tree vertices

Let N be a nearly stable network. Consider a lowest big tree vertex component C of N, and let ρ_C denote the root of C. Let u be a vertex in C. If u is stable, then all vertices in the path from ρ_C to u are stable by Proposition 2.2(1). Therefore, the stable vertices in C span a subtree of C with the same root ρ_C (Fig. 8A), called the *stable subtree* of C. The following proposition characterizes vertices outside the stable subtree.

Proposition 5.1. *Let N be a nearly stable network, and C be a lowest big tree vertex component of N. Then every unstable tree vertex in C has two reticulation children.*

Proof. Let x be an unstable tree vertex and y be a child of x in C. Assume y is a tree vertex. If y was stable, then x would also be stable (Proposition 2.2(1)), a contradiction. Therefore both y and x should be unstable, which contradicts the fact that N is nearly stable, so y cannot be a tree vertex. □

In the example given in Fig. 8, the stable subtree has four leaves ℓ_9, s_1, s_2, s_3. Clearly, every network leaf in C is a leaf of the stable subtree. We let $S(C)$ denote the set of leaves of the stable subtree that are not network leaves, i.e.:

$$S(C) = \{s \in \mathcal{V}(C) \setminus \mathcal{L}(N) : s \text{ is stable but every tree vertex child of } s \text{ is unstable}\}. \quad (6)$$

Proposition 5.2. *Let x be a vertex in C. Then, $x \in S(C)$ if and only if each child of x in C is a leaf of C but not a network leaf.*

Proof. Let $x \in S(C)$.

Suppose by contradiction that x has no child in C. Then x has two reticulation children r_1 and r_2. Thus, by Corollary 2.1, x is unstable: contradiction. Then let y be a child of x in C. By definition, y is unstable and thus is not a network leaf. If y is not a leaf of C, there is an unstable tree vertex below y, contradicting that N is nearly stable.

Conversely, let y be a leaf of C but not a network leaf. Then, y has two reticulation children. Thus, by Corollary 2.1, y is unstable. Since N is nearly stable, the parent of y (i.e. the vertex x) must be stable. If the sibling of y is either a reticulation or a leaf of C but not a network leaf, its parent is then a lowest stable vertex in C and hence is in $S(C)$. □

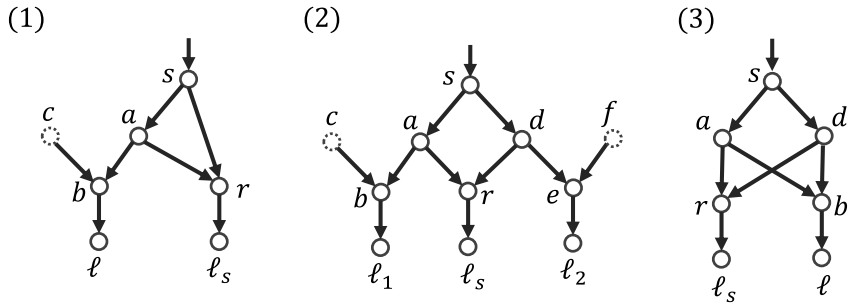

Fig. 9. Three possible mini-structures below a vertex $s \in S(C)$ in a lowest component C. Here, the vertices c and f have undetermined vertex type, each of them can be a tree or reticulation vertex.

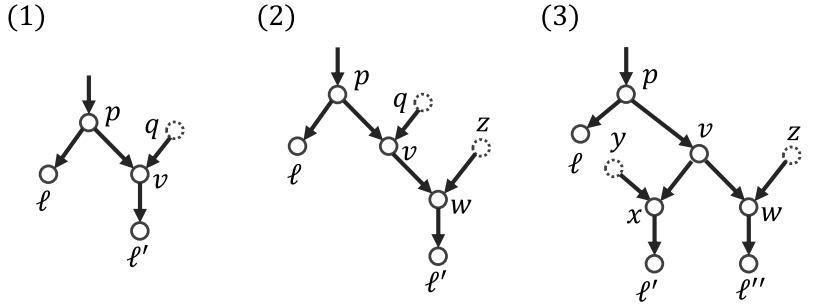

Fig. 10. Three possible mini-structures below a tree vertex for which one child is a network leaf and the other is not a stable tree vertex in a lowest component. The vertices q, y and x have undetermined vertex type.

5.2. Two key lemmas

Let N and G be the given network and phylogenetic tree. We assume that N does not contain any parallel edge nor a vertex with indegree and outdegree one.

The following lemma limits the possible mini-structures below a vertex $s \in S(C)$ for a lowest tree vertex component C. We use $N[s]$ to denote the subnetwork consisting of all the vertices below s (including reticulation vertices below C).

Lemma 5.1. *Let C be a lowest tree vertex component of a network N. For $s \in S(C)$, $N[s]$ has only three possible mini-structures given in Fig. 9.*

Proof. Assume s is stable on a leaf ℓ_s. The leaf ℓ_s cannot be in C. Otherwise all the vertices in the path from s to ℓ_s are stable, contradicting the fact that $s \in S(C)$. Thus, ℓ_s is the unique child of an inner reticulation r below s. Let u be a parent of r. If u is a reticulation, it is then unstable and its parents are two tree vertices equal to or below s in C. Since $s \in S(C)$, u and each of its parents below s are unstable, contradicting that N is nearly stable. Hence the parents of r are two tree vertices below s in C.

Since N does not contain parallel edges, at least a parent of r is not s. Let a be such a parent of r. Note that a is a child of s, otherwise, since $s \in S(C)$, any parent of the tree vertex a below s in C would not be a leaf of C, therefore contradicting Proposition 5.2. The fact that $s \in S(C)$ also implies that a is unstable. Since N is nearly stable, there are no two unstable vertices that appear consecutively in a path. This implies that a must also be a child of s and the other child b of a is a stable reticulation for which the child is a network leaf ℓ, as C is a lowest component of N.

Let d be the other parent of r. If $d = s$, we obtain the mini-structure (1) in Fig. 9.

If $d \neq s$, the other child of d is also a stable reticulation with a network leaf as its child, as N is nearly stable. If a and d have distinct child other than r, we obtain the mini-structure (2) in Fig. 9. If a and d have the same children, we obtain the mini-structure (3) in Fig. 9. This completes the proof. □

Now, we consider network leaves in C. For a network leaf ℓ in C, let p be the parent of ℓ. The following lemma gives all possible mini-structures of $N[p]$ if the sibling of ℓ is not a stable tree vertex.

Lemma 5.2. *If the sibling of ℓ is not a stable tree vertex in C, $N[p]$ has only three possible mini-structures given in Fig. 10.*

Proof. Let v be the sibling of ℓ. It is either a tree vertex or a reticulation vertex.

If v is a stable reticulation, the child of v must be a network leaf. Then, we obtain the mini-structure (1) in Fig. 10.

If v is an unstable reticulation, then its child w is a stable reticulation. Clearly, the child of w is a leaf. This gives the mini-structure (2) in Fig. 10.

Finally, if v is an unstable tree vertex, the children of v must be stable, as N is nearly stable. By the part (1) of Proposition 2.2, the children of v are stable reticulation vertices, for which the unique child is a network leaf. Otherwise, v is stable. Thus, we obtain the mini-structure (3) in Fig. 10. □

5.3. Dissolving the lowest components

Now, we show how to dissolve a lowest big tree vertex component C by working one by one on the subnetworks below the vertices in $S(C)$ and then the parents of network leaves in C.

In the rest of this discussion, we use $x \wedge_G y$ (resp. $x \not\wedge_G y$) to denote that x and y are (resp. not) siblings in G. We also use $\mathrm{par}_G(v)$ to denote the parent of v in G.

First, we consider the mini-structure (1) in Fig. 10, which is named the "uncle–nephew structure" in [10], where two leaves ℓ and ℓ' have the uncle and nephew relationship and q is either a tree or reticulation vertex.

Lemma 5.3. *If N contains the mini-structure (1) in Fig. 10, define*

$$E = \begin{cases} (q, v) \text{ if } \ell \wedge_G \ell'; \\ (p, v) \text{ otherwise.} \end{cases}$$

Then N displays G if and only if $N - E$ displays G.

Proof. Let $N' = N - E$. If N' displays G, then N also displays G, as N' is a subnetwork of N. To prove the necessity, we assume that N displays G with a spanning tree T such that G is a contraction of T. There are two possible cases.

(1) $\ell \wedge_G \ell'$: note that T contains either (p, v) or (q, v) exclusively. If T contains (p, v), N' displays G, as T is also a spanning tree of N'.
If T contains (q, v), let h be the lowest common ancestor of ℓ and ℓ' in T. Since any path to ℓ must contain p, p must be in the path from h to p in T and $T[h]$ only has two labeled leaves, namely ℓ and ℓ'.
Define $T' = T - (q, v) + (p, v)$. Then, the vertex p becomes the lowest common ancestor of ℓ and ℓ' in T', so $T[h]$ and $T'[h]$ do not contain any network leaves other than ℓ and ℓ'. Any edge of T that is not below h is also an edge in T' and conversely, so G is a contraction of both T and T'. The tree T' is a spanning tree of $N - (q, v)$, and thus is the evidence that $N - (q, v)$ displays G.

(2) $\ell \not\wedge_G \ell'$: if (p, v) is an edge in T, then ℓ and ℓ' has p as their lowest common ancestor in T, contradicting that ℓ and ℓ' are not sibling in G. Thus, T does not contain (p, v) and T is also a spanning tree of $N - (p, v)$, implying that N' also displays G. □

Using the above lemma, we can prove the following facts. These facts suggest that we can dissolve the subnetwork below each $s \in S(C)$ by using the structural information on G.

Lemma 5.4. *Let $s \in S(C)$. Define N' as follows.*

(i) *If $N[s]$ has the mini-structure (1) in Fig. 9,*

$$N' = N - \begin{cases} \{(a, r), (c, b)\} & \text{if } \ell \wedge_G \ell_s, \\ \{(a, r), (a, b)\} & \text{otherwise.} \end{cases} \tag{7}$$

(ii) *If $N[s]$ has the mini-structure (2) in Fig. 9,*

$$N' = N - \begin{cases} \{(a, b), (a, r), (d, e)\} & \text{if neither } \ell_1 \wedge_G \ell_s \text{ nor } \ell_2 \wedge_G \ell_s, \\ \{(c, b), (d, r), (f, e)\} & \text{if } \ell_1 \wedge_G \ell_s \text{ and } \mathrm{par}_G(\ell_s) \wedge_G \ell_2, \\ \{(c, b), (d, r), (d, e)\} & \text{if } \ell_1 \wedge_G \ell_s \text{ but } \mathrm{par}_G(\ell_s) \not\wedge_G \ell_2, \\ \{(c, b), (a, r), (f, e)\} & \text{if } \ell_2 \wedge_G \ell_s \text{ and } \mathrm{par}_G(\ell_s) \wedge_G \ell_1, \\ \{(a, b), (a, r), (f, e)\} & \text{if } \ell_2 \wedge_G \ell_s \text{ but } \mathrm{par}_G(\ell_s) \not\wedge_G \ell_1. \end{cases} \tag{8}$$

(iii) *If $N[s]$ has the mini-structure (3) in Fig. 9, $N' = N - \{(a, b), (d, r)\}$.*

Then N displays G if and only if N' displays G.

Proof. Since N' is a subnetwork of N in each case, N displays G if N' displays G. To prove the other direction, assume that N displays G. There is a spanning tree T of N such that G is a contraction of T.

(i) The reticulation edge (a, r) is redundant. $N - (a, r)$ is essentially an uncle–nephew structure. Hence by Lemma 5.3, we have N displays G only if N' defined in Eq. (7) displays G.

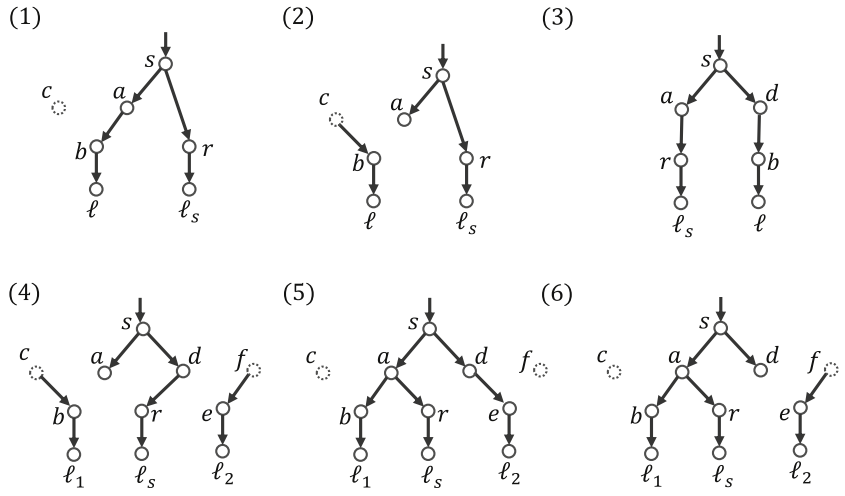

Fig. 11. The resulting networks after $N[s]$ is simplified. (1) and (2) for the two cases of the mini-structure (1). (3) for the mini-structure (3). (4)–(6) for the first three cases of the mini-structure (2). The cases 4 and 5 of the mini-structure (2) are symmetric to the cases 2 and 3, respectively.

(ii) In Eq. (8), the fourth and fifth cases are symmetric to the second and third cases, respectively. Hence, we only consider the first three cases.

CASE 1. Neither $\ell_1 \wedge_G \ell_s$ nor $\ell_2 \wedge_G \ell_s$.

Clearly, neither (a, b) nor (d, e) are in T. Otherwise, either ℓ_1 or ℓ_2 is the sibling of ℓ_s, contradicting the assumption. So T is a spanning tree of $N - \{(a, b), (d, e)\}$. If (d, r) is in T, then, T is also a spanning tree of N'. If (a, r) is in T, then $T - \{(a, r)\} + \{(d, r)\}$ also displays G and is a spanning tree of N'. Thus, N displays G only if N' displays G.

CASES 2 and 3. $\ell_1 \wedge_G \ell_s$.

We first claim that $N - (d, r)$ displays G. If (d, r) is not in T, then the claim is true. If (d, r) is in T, then (d, e) is not in T. Otherwise, ℓ_s and ℓ_2 are siblings in T. Hence, G is also a contraction of $T - (d, r) + (a, r)$, and $N - (d, r)$ displays G.

Next, note that there is essentially an uncle–nephew structure on the leaves ℓ_1 and ℓ_s in $N - (d, r)$. Hence, the assumption that $\ell_1 \wedge_G \ell_s$ implies that $N - \{(d, r), (c, b)\}$ displays G.

Finally, in $N - \{(d, r), (c, b)\}$, we have essentially an uncle–nephew structure after contraction of the subtree below a. Therefore, if ℓ_2 is the sibling of the parent of ℓ_1 and ℓ_s in G, $N - \{(d, r), (c, b), (f, e)\}$ displays G. Otherwise, $N - \{(d, r), (c, b), (d, e)\}$ displays G.

(iii) In this case, deleting which edge entering at r and b makes no difference. Therefore, N displays G if and only if N' displays G. □

After we modify $N[s]$ according to the rules suggested in Lemma 5.4 for every $s \in S(C)$, the subtree below each s consists of network leaves, vertices of degree 2 and/or dummy vertices in the resulting network N' (Fig. 11). We can further replace $N[s]$ and the corresponding subtree in G by the same leaf ℓ_s, if they are compatible. Otherwise, we conclude that N does not display G and stop the algorithm.

In summary, the procedure for simplifying the subnetwork below a lowest stable vertex in C is given in Algorithm 1.

Note that no stable reticulation vertex becomes unstable after removing reticulation edges in a network if at least one reticulation edge is kept for each reticulation vertex, which is the case for all the rules of this simplification process. Therefore, after the simplification process terminates, N' is still nearly stable. Each of the components different from C may be simplified into a smaller one in N'. Additionally, two big tree vertex components may even be merged into one component in N'.

When working on $N[s']$ for some $s' \in S(C)$, we may transform another $N[s]$ into a subtree or a subnetwork of different mini-structure through the elimination of the reticulation vertices that have a parent in both $N[s]$ and $N[s']$. The following result describes the possible modifications on the subnetwork $N[s]$ below $s \in S(C)$.

Proposition 5.3. *Let $s, s' \in S(C)$ such that $\mathcal{V}(N[s]) \cap \mathcal{V}(N[s']) \neq \emptyset$. Calling Dissolve_Lowest_Stable_Vertices on s' first will change $N[s]$ into (i) a subnetwork of the mini-structure in Fig. 9.1,* (ii) *a three-vertex tree with two network leaves*, (iii) *a length-2 path from s to a network leaf on which s is stable, or* (iv) *a subnetwork in which at least a child of s is a stable tree vertex.*

Proof. To distinguish $N[s]$ before and after the procedure Dissolve_Lowest_Stable_Vertices is called on s', we use $\bar{N}[s]$ to denote the subnetwork obtained when the procedure terminates. The existence of s' such that $\mathcal{V}(N[s]) \cap \mathcal{V}(N[s']) \neq \emptyset$ implies that $N[s]$ has the mini-structure either (1) or (2) in Fig. 9.

If $N[s]$ has the mini-structure (1) in Fig. 9, calling the procedure first on s' affects $N[s]$ only if $\mathcal{V}(N[s]) \cap \mathcal{V}(N[s']) = \{b, \ell\}$. Therefore, calling the procedure on s' may alter $N[s]$ in two possible ways. One possibility is that (a, b) is deleted and then

Algorithm 1: Dissolving lowest stable vertices

Procedure *Dissolve_Lowest_Stable_Vertices(N, C, s)*
  Input: a network N, a component C, a vertex s
  Output: simplified N with reticulations below s being eliminated
1 **if** *Case 1 holds (Fig. 9.1)* **then**
2 **if** $\ell \wedge_G \ell_s$ **then**
3 delete (a, r) and (c, b) (Fig. 11.1); contract the edge(s) entering c;
4 **else**
5 delete (a, r) and (a, b) (Fig. 11.2); contract (c, b);
6 **if** *Case 2 holds (Fig. 9.2)* **then**
7 **if** *neither* $\ell_1 \wedge_G \ell_s$ *nor* $\ell_2 \wedge_G \ell_s$ **then**
8 delete (a, b), (a, r), and (d, e) (Fig. 11.4);
9 contract the edges (c, b) and (f, e);
10 **else if** $\ell_1 \wedge_G \ell_s$ *and* $par_G(\ell_s) \wedge_G \ell_2$ **then**
11 delete (c, b), (d, r), and (f, e) (Fig. 11.5);
12 contract the edge(s) entering c or f;
13 **else if** $\ell_1 \wedge_G \ell_s$ *but* $par_G(\ell_s) \not\wedge_G \ell_2$ **then**
14 delete (c, b), (d, r), and (f, e);
15 contract the edge(s) entering f and the edge (c, b);
16 **else if** $\ell_2 \wedge_G \ell_s$ *and* $par_G(\ell_s) \wedge_G \ell_1$ **then**
17 delete (c, b), (a, r), and (f, e);
18 contract the edge(s) entering c or f;
19 **else if** $\ell_2 \wedge_G \ell_s$ *but* $par_G(\ell_s) \not\wedge_G \ell_1$ **then**
20 delete (a, b), (a, r), and (f, e);
21 contract the edge(s) entering f and the edge (c, b);
22 **if** *Case 3 holds (Fig. 9.3)* **then**
23 **if** $\ell \wedge_G \ell_s$ **then**
24 skip;
25 **else**
26 output "G is not displayed" and exit;
27 contract $N[s]$ into ℓ_s;
28 contract $G[\mathrm{par}_G(\mathrm{par}_G(\ell_s))]$ or $G[\mathrm{par}_G(\ell_s)]$ into ℓ_s if necessary;

(s, a) is contracted, resulting in $\bar{N}[s]$ equivalent to a path: $s \to r \to \ell_s$. The second possibility is that (c, b) is deleted and (a, b) is then contracted, changing the tree vertex a into a tree vertex stable on ℓ in $\bar{N}[s]$ and hence making s not in $S(C)$.

If $N[s]$ has the mini-structure (2) in Fig. 9, calling the procedure first on s' affects $N[s]$ only if $\mathcal{V}(N[s]) \cap \mathcal{V}(N[s'])$ is equal to $\{b, \ell_1\}$, $\{e, \ell_2\}$, or $\{b, e, \ell_1, \ell_2\}$. The first two cases are symmetric and thus we consider the following two cases.

CASE 1. $\mathcal{V}(N[s]) \cap \mathcal{V}(N[s']) = \{b, \ell_1\}$ (resp. $\{e, \ell_2\}$).

Running the procedure on s' may alter $N[s]$ in two possible ways. One possibility is that (a, b) (resp. (d, e)) is deleted and then (s, a) (resp. (s, d)) is contracted, simplifying $\bar{N}[s]$ into a subnetwork of the mini-structure (1) in Fig. 9.

The second possibility is that (c, b) (resp. (f, e)) is deleted and (a, b) (resp. (d, e)) is then contracted. This changes the vertex a from an unstable tree vertex into a stable tree vertex. As such, $\bar{N}[s]$ contains an uncle–nephew structure and s is no longer in $S(C)$.

CASE 2. $\mathcal{V}(N[s]) \cap \mathcal{V}(N[s']) = \{b, e, \ell_1, \ell_2\}$.

In this case, $N[s']$ has the same mini-structure (2) in Fig. 9. Running the procedure on s' may alter $N[s]$ in three possible ways. One possibility is that both (a, b) and (d, e) are deleted and thus both (s, a) and (s, d) are contracted. This makes $\bar{N}[s]$ equivalent to the path $s \to r \to \ell_s$.

Another possibility is that (a, b) and (f, e) (resp. (c, b) and (d, e)) are deleted and thus (s, a) and (d, e) (resp. (a, b) and (s, d)) are contracted. This makes $\bar{N}[s]$ essentially equivalent to a 3-vertex tree $(s, (\ell_1, \ell_s))$ (resp. $(s, (\ell_2, \ell_s))$).

The third possibility is that both (c, b) and (f, e) are deleted and thus both (a, b) and (d, e) are contracted. This changes the vertices a and d from unstable tree vertices into stable tree vertices, making s not in $S(C)$. □

In N', each vertex of $S(C)$ becomes a new network leaf. Importantly, the following fact is true for the simplified component, which is still called C for convenience. It says that the simplification process will not create new elements for $S(C)$.

Proposition 5.4. *$S(C) = \emptyset$ in N'.*

Proof. Assume the resulting stable subtree of the component contains a leaf u that is not a network leaf of N'. Then, u must be an unstable tree vertex in N. Otherwise, either u has a stable tree child in N (and thus in N'), or u is in $S(C)$ and thus becomes a leaf in N', a contradiction. By Proposition 5.1, u has two reticulation children x and y. Since C is a lowest component of N, the unique child for x and y must be a network leaf. If the reticulation edges entering x and y have never been removed, u could not become stable. If some edge entering x or y is deleted when $N[s]$ is modified for some $s \in S(C)$, then the involved child of u is contracted and thus u becomes either a dummy leaf or the parent of a network leaf. By Proposition 5.2, u is not in $S(C)$, a contradiction. □

Now $S(C) = \emptyset$. However, there may be reticulation vertices below C. We further simplify C using each of the rules in the following lemma.

Lemma 5.5. *Let ℓ be a network leaf in C and p be the parent of ℓ such that the sibling of ℓ is not a stable tree vertex in C. Define N' as follows.*

(i) *If $N[p]$ has the mini-structure (1) in Fig. 10,*

$$N' = N - \begin{cases} (q, v) & \text{if } \ell' \wedge_G \ell, \\ (p, v) & \text{otherwise.} \end{cases} \tag{9}$$

(ii) *If $N[p]$ has the mini-structure (2) in Fig. 10,*

$$N' = N - \begin{cases} \{(q, v), (z, w)\} & \text{if } \ell' \wedge_G \ell, \\ \{(p, v)\} & \text{otherwise.} \end{cases} \tag{10}$$

(iii) *If $N[p]$ has the mini-structure (3) in Fig. 10,*

$$N' = N - \begin{cases} \{(y, x), (v, w)\} & \text{if } \ell' \wedge_G \ell, \\ \{(v, x), (z, w)\} & \text{if } \ell'' \wedge_G \ell, \\ \{(y, x), (z, w)\} & \text{if } \ell' \wedge_G \ell'' \text{ and } par_G(\ell') \wedge_G \ell, \\ \{(v, x), (v, w)\} & \text{otherwise.} \end{cases} \tag{11}$$

Then, N displays G if and only if N' displays G.

Proof. N' is a subnetwork of N, so clearly N displays G if N' displays G.

Assume that N displays G. There exists a spanning tree T of N such that G is a contraction of T.

By Lemma 5.2, there are three possible mini-structures for $N[p]$ shown in Fig. 10. The mini-structure (1) is an uncle–nephew structure and thus the sufficiency follows from Lemma 5.3. The sufficiency for the mini-structure (2) in Fig. 10 can be proven similarly.

Suppose $N[p]$ has the mini-structure (3) in Fig. 10, the first and second case in the definition of N' are symmetric, so we only need to consider the cases 1, 3 and 4.

1: $\ell' \wedge_G \ell$: since T must contain p, T does not contain (v, w). Otherwise, ℓ'' is the sibling of either ℓ' or ℓ depending whether or not (v, x) is in T, contradicting the assumption in this case. This implies that T is a spanning tree of $N - (v, w)$.
In $N - (v, w)$, $(N - (v, w))[p]$ is essentially an uncle–nephew structure. By Lemma 5.3, $N' = (N - (v, w)) - (y, x)$ displays G.

3: $\ell' \wedge_G \ell''$ and $\text{par}_G(\ell') \wedge_G \ell$.
Notice that T contains p. If T contains exactly one of (v, x) and (v, w), ℓ is a sibling of either ℓ' or ℓ'' in G, contradicting the assumption in this case.

Therefore, either both (v, x) and (v, w) are in T or none of them is in T. If the former holds, then T is a spanning tree of N' and N' also displays G. If the latter holds, let t_1 denote the lowest common ancestor of ℓ' and ℓ'' and let t_2 be the lowest common ancestor of t_1 and ℓ in T. Then, $T[t_2]$ contains only three labeled leaves: ℓ, ℓ', and ℓ''. Let $T' = T - \{(y, x), (z, w)\} + \{(v, x), (v, w)\}$. The tree in which ℓ' and ℓ'' are siblings and ℓ is their uncle is a contraction of $T'[t_2]$ and $T[t_2]$, and T' is a spanning tree of $N - \{(y, x), (z, w)\}$. Therefore, $N - \{(y, x), (z, w)\}$ displays G.

4: All the first three cases are not true.
As shown in the third case, either both (v, x) and (v, w) are both in T or none of them is in T. If (v, x) and (v, w) are both in T, then ℓ' and ℓ'' are sibling and their parent is a sibling of ℓ in G, contradiction. Therefore, T is a spanning tree of N'. Hence, N' also display G. □

By Lemma 5.5, we can simplify the subnetwork below the parent of each network leaf in C using the following procedure called Dissolve_SubntkNearNtkLeaf, which is illustrated in Fig. 12. When the procedure is called on a network leaf, it may alter the subnetwork below the parent of another network leaf. Because of this, when a network leaf is examined, the subnetwork below its parent may have one of the degenerated structures listed in Fig. 13. For this case, Dissolve_DegeneratedCases is called.

Algorithm 2: Dissolving subnetwork near network leaf

Procedure *Dissolve_SubntkNearNtkLeaf(N, C, ℓ)*
Input: a network N, a component C, a leaf ℓ
Output: N with reticulations below the parent p of ℓ being eliminated
1 **if** *Case 1 holds (Fig. 10.1)* **then**
2 **if** $\ell \wedge_G \ell'$ **then**
3 delete (q, v);
4 **else**
5 delete (p, v);
6 **if** *Case 2 holds (Fig. 10.2)* **then**
7 **if** $\ell \wedge_G \ell'$ **then**
8 delete (q, v) and (z, w) (Fig. 12.1);
9 **else**
10 delete (p, v) (Fig. 12.2);
11 **if** *Case 3 holds (Fig. 10.3)* **then**
12 **if** $\ell \wedge_G \ell'$ **then**
13 delete (y, x) and (v, w) (Fig. 12.3);
14 **else if** $\ell \wedge_G \ell''$ **then**
15 delete (v, x) and (z, w) (Fig. 12.4);
16 **else if** *($\ell' \wedge_G \ell''$) and ($par_G(\ell') \wedge_G \ell$)* **then**
17 delete (y, x) and (z, w) (Fig. 12.5);
18 **else**
19 delete (v, x) and (v, w) (Fig. 12.6);
20 contract $N[p]$ into ℓ; contract $G[\mathrm{par}(\ell)]$ into ℓ if necessary (Fig. 12).

By repeatedly simplifying the subnetwork below the parent of each network leaf through the removal of edges entering reticulations and contraction of edges, we will further transform the component into a single vertex or else we will discover that the input network does not display the tree G.

5.4. The algorithm

We are now ready to describe a linear-time TC algorithm (Algorithm 4) for nearly stable networks. It is divided into the pre-processing and simplification parts. Roughly speaking, the algorithm first topologically sorts the big tree vertex components and then dissolves these components one by one in a bottom-up manner. Each component is dissolved in two stages. In the first stage , Dissolve_Lowest_Stable_Vertices is called for each $s \in S(C)$. In the second stage, Dissolve_SubntkNearNtkLeaf and Dissolve_DegenerateCases are repeatedly called. In the rest of this section, we discuss the correctness and running time of the algorithm step by step.

Let N and G be the input nearly stable network and phylogenetic tree with the same set of n labeled leaves, respectively. Then, $|\mathcal{V}(N)| = O(n)$, which is proved in Section 4, and hence $|\mathcal{E}(N)| = O(n)$, as N is degree-bounded. If G is displayed in N, $|\mathcal{V}(G)| \leq |\mathcal{V}(N)|$. We shall show that the proposed TC algorithm takes $O(n)$ operations. Here, the operations include (i) edge contraction, (ii) edge deletion, (iii) membership check for $\mathcal{T}(N)$, $\mathcal{R}(N)$ and $\mathcal{L}(N)$, (iv) verification of sibling or parent–son relationship and (v) traverse to a vertex in N or G.

First, in the pre-processing (line 1), we first identify and topologically sort the roots of the big tree vertex components in $O(|\mathcal{E}(N)| + |\mathcal{V}(N)|)$ [20, page 103]. Note that the root of each component is the unique child of a stable reticulation vertex. We then use breadth-first search to compute each big component by starting from its root and order these tree vertex components in the same way as their roots are ordered.

After the pre-processing, the components are simplified one by one in a bottom-up manner. In line 3, the number of operations taken is linear in the number of contracted edges. Because each edge is contracted at most once, in a bottom-up manner, this step takes at most $O(|\mathcal{E}(N)|)$ operations.

For a component C, by Proposition 5.2, a vertex x in $S(C)$ if and only if each child of x in C is a leaf of C but not a network leaf. Hence, when traversing C in pre-order, a vertex x is in $S(C)$ if and only if (a) the next vertex is in $\mathcal{L}(C) \setminus \mathcal{L}(N)$ and (b) the next vertex is either not a child of x or also in $\mathcal{L}(C) \setminus \mathcal{L}(N)$, each of which can be checked in constant time. Hence, the total number of operations taken on line 3 is linear in $\sum_{0 \leq i \leq k} |\mathcal{V}(C_i)|$, for all $k + 1$ big tree-vertex components of N, which is at most $|\mathcal{V}(N)|$.

Algorithm 3: Dissolving degenerated cases

Procedure *Dissolve_DegeneratedCases(N, C, ℓ)*
Input: a network N, a component C, a leaf ℓ
Output: N with reticulations below the parent p of ℓ being eliminated
if *Case 1 or 2 holds (Fig. 13.1 or Fig. 13.2)* **then**
  skip;
if *Case 3 or 4 holds (Fig. 13.3 or Fig. 13.4)* **then**
  if $\ell \wedge_G \ell'$ **then**
    skip;
  else
    output "No" and exit;
if *Case 5 holds (Fig. 13.5)* **then**
  if *($\ell' \wedge_G \ell''$) and ($par_G(\ell') \wedge_G \ell$)* **then**
    skip;
  else
    output "No" and exit;
if *Case 6 holds (Fig. 13.6)* **then**
  if $\ell \wedge_G \ell'$ **then**
    delete (z, w);
  else
    delete (v, w);
if *Case 7 holds (Fig. 13.7)* **then**
  if *($\ell' \wedge_G \ell''$) and ($par_G(\ell') \wedge_G \ell$)* **then**
    delete (z, w);
  else if $\ell' \wedge_G \ell$ **then**
    delete (v, w);
  else
    output "No" and exit;
contract $N[p]$ into ℓ; contract $G[\mathrm{par}_G(\ell)]$ into ℓ if necessary.

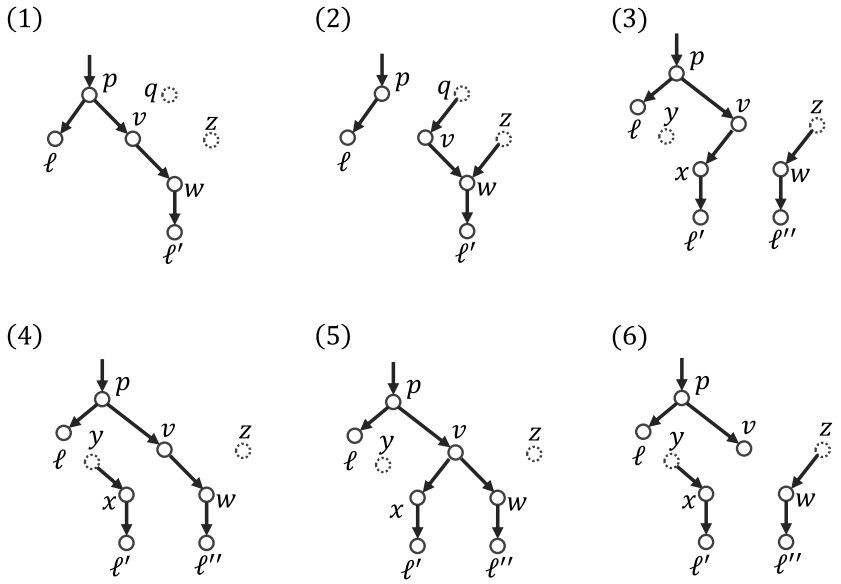

Fig. 12. The resulting networks after $N[p]$ is simplified using each of the rule in Lemma 5.5. (1) and (2) for the two cases of the mini-structure (2) in Fig. 10. (3)–(6) for the four cases of the mini-structure (3) in Fig. 10.

Algorithm 4: A linear-time tree containment algorithm for nearly stable networks

Procedure *Tree_containment(N, G)*
Input: a network N, a tree G
Output: true if N contains G, false otherwise
`// 1. Pre-process N`
1 Compute and topologically sort the big tree-vertex components C_i $(0 \leq i \leq k)$ so that $\rho(C_j)$ is a descendant of $\rho(C_i)$ only if $i < j$;
`/* Some big components may contain only leaves that are in` $\mathcal{L}(N)$ `(i.e., below which there are no reticulation vertices) */`
2 **for** $i = k$ **to** 0 **do**
3 simplify $C = C_i$ by repeatedly contracting degree-2 vertices and dummy leaves so that it becomes binary;
4 compute $S(C)$ by traversing C in pre-order (see Proposition 5.2);
5 **foreach** $s \in S(C)$ **do**
6 **if** $N[s]$ *has a mini-structure in Fig. 9* **then**
7 Dissolve_Lowest_Stable_Vertices(N,C,s);
8 **else**
9 simplify $N[s]$;
10 Resimplify C by repeatedly contracting degree-2 vertices and dummy leaves;
`/* Assume the left child of a vertex is always a stable tree vertex, if it has any stable tree vertex child. Exchange the left and right children of a vertex if needed. */`
11 (Traversing C in post-order: $u_1, u_2, \ldots, u_t = \rho(C)$)
12 **foreach** $\ell = u_i$ **do**
13 **if** ℓ *does not have a sibling* **then**
14 replace its parent u_{i+1} with ℓ;
15 **else if** *its sibling* v *is also a network leaf* **then** `// Case C1`
16 **if** $\ell \not\prec_G v$ **then**
17 **return false**;
18 **else if** v *has been visited* **then**
19 contract $N[\mathrm{par}(\ell)]$ and $G[\mathrm{par}(\ell)]$ into ℓ;
20 **else** skip ;
21 **else if** *its sibling is not a stable tree vertex* **then** `// Cases C2 & C3`
22 **if** *the subnetwork below the parent of* ℓ *has a mini-structure in Fig. 10* **then**
23 Dissolve_SubntkNearNtkLeaf(N, C, ℓ);
24 **else**
25 Dissolve_DegenerateCases(N, C, ℓ);
26 **else** skip ; `// Case C4: the sibling of` ℓ `is a stable tree vertex`
27 **return true**;

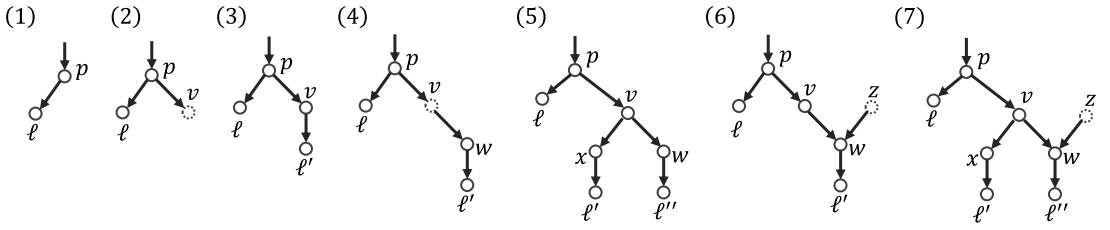

Fig. 13. The possible degenerated structures of the subnetwork below the parent of a network leaf in a lowest tree component C. In (2) and (4), v can be either an unstable tree vertex or a reticulation vertex.

For each component C, $|S(C)| \leq |\mathcal{L}(C)|$ and thus $|S(C)| \leq |\mathcal{V}(C)|/2$. By the **foreach** loop on line 5–9, we simplify the networks $N[s]$ below $s \in S(C)$ one-by-one. Since there are only three possible mini-structures in Fig. 9, each containing 4–6 vertices, the conditional statement can be verified in constant time. If the conditional statement is true, calling the procedure Dissolve_Lowest_Stable_Vertices on s takes constant time. If the conditional statement is false, $N[s]$ has been modified when working on some $s' \in S(C)$ earlier. By Proposition 5.3, s is no longer in $S(C)$ or $N[s]$ is equivalent to a length-2 path from

s to ℓ_s on which s is stable. Hence, the simplification of $N[s]$ also takes constant time. Overall, the first **foreach** loop takes $O(|S(C)|)$ operations for each C. Hence the total number of the operations taken by the algorithm is $O(|\mathcal{V}(N)|)$ for line 5–9. Note that when the leaves of C are all network leaves, $S(C) = \emptyset$ and the **foreach** loop will stop.

$S(C)$ is empty when line 10 starts. Then, either an internal tree vertex or its parent is in the path from $\rho(C)$ to a network leaf. Moreover, if an internal tree vertex is stable, as assumed in the beginning of this step, its left child is also a stable tree vertex.

Let u be an internal tree vertex in the current component C. Assume at least one child of $u \in \mathcal{V}(C)$ is a network leaf. Since the vertices in C are traversed in post-order, a network leaf child of u must be visited before u. In particular, line 12 always starts with a network leaf, which is the leftmost leaf of C.

Assume the currently visited vertex $\ell = u_i$ is a network leaf and it is the left child of its parent p. Let v be the other child of p in N. There are the following four cases.

C1. The vertex v is also a network leaf. Then $v = u_{i+1}$ and $p = u_{i+2}$. If ℓ and v are not siblings in G, we have the evidence that G is not displayed and hence the algorithm terminates. If ℓ and v are siblings in G, then, the algorithm simply moves on to process $v = u_{i+1}$. At v, the algorithm will replace p with v and also replace the parent of ℓ and v with v in G. As such, when the algorithm processes u_{i+2}, it has become a network leaf now and $\{u_{i+2}, \ldots, u_t\}$ is still the vertex set of the modified component, listed in the post-order. This explain why the algorithm simply does not act when visiting ℓ in this case.

C2. The vertex v is not in C, that is, a reticulation vertex. Then, $u_{i+1} = p$. If $N[p]$ has the mini-structures (1) and (2) in Fig. 10, the procedure Dissolve_SubntkNearNtkLeaf is called on ℓ. Otherwise, Dissolve_DegenerateCases is called on ℓ. After p is replaced with a network leaf, then $\{p = u_{i+1}, \ldots, u_t\}$ is the vertex set of the modified component listed in the post-order.

C3. The vertex v is a tree vertex but unstable. Then, $v = u_{i+1}$ and $N[p]$ has the mini-structure (3) in Fig. 10 or a degenerated structure in Fig. 13. Similar to the case C2, after Dissolve_SubntkNearNtkLeaf or Dissolve_DegenerateCases is called on ℓ, $\{p = u_{i+2}, \ldots, u_t\}$ is the vertex set of the modified component listed in the post-order.

C4. The vertex v is an internal, stable tree vertex in C. This indicates that the subnetwork $N[v]$ contains at least one stable tree vertex and possibly some reticulation vertices. In this case, u_{i+1} is the leftmost network leaf in $C[v]$. The algorithm will move on to simplify $C[v]$ into a network leaf and then contracting the parent of both ℓ and v into a network leaf. In order to simplify $C[v]$ first, the algorithm simply skips at ℓ.

Clearly, the second **foreach** loop takes constant operations in each execution.

Taken together, the above facts show the following theorem.

Theorem 5.1. *Algorithm 4 uses $O(|\mathcal{L}(N)|)$ operations to solve the TC problem for nearly stable networks N.*

6. Conclusion

In this paper, we have studied nearly stable and genetically stable networks, two classes that were introduced in the study of the tree containment problem. In particular, we have proved that each nearly (resp. genetically) stable network with n leaves contains at most $3(n-1)$ (resp. $2(n-1)$) reticulation vertices. We have also designed a linear-time TC algorithm for binary nearly stable networks.

This study arises several problems for future study. The cluster containment (CC) problem asks whether or not a given cluster appears in some phylogenetic tree displayed by a given network. Gunawan et al. [12] designed a linear-time algorithm for the CC problem. Now, we have obtained a linear-time algorithm for the TC problem. Is there a linear-time TC algorithm for reticulation-visible networks? If the answer to this question is negative, is there a linear-time TC problem for genetically stable networks?

In each binary nearly stable network, there are at most $3(n-1)$ reticulation vertices. This is also true for reticulation-visible networks. Is it possible to define a superclass of networks that contains both reticulation-visible networks and nearly stable networks for which the TC and CC problems are still polynomial-time solvable?

Binary nearly stable networks have simple local structures compared with reticulation-visible networks. Hence, it is also interesting to investigate how to efficiently reconstruct nearly stable network models from gene trees or sequence data.

Acknowledgments

The project was financially supported by the 2013 Merlion Programme (grant 9.02.13). ADMG and LXZ were also supported by a Singapore MOE Tier-1 grant R-146-000-238-114.

References

[1] M. Arenas, G. Valiente, D. Posada, Characterization of reticulate networks based on the coalescent with recombination, Mol. Biol. Evol. 25 (12) (2008) 2517–2520.

[2] J.A. Bondy, U.S.R. Murty, Graph Theory, Springer, 2008.

[3] M. Bordewich, C. Semple, Reticulation-visible networks, Adv. in Appl. Math. 78 (2016) 114–141.
[4] G. Cardona, M. Llabrés, F. Rosselló, G. Valiente, A distance metric for a class of tree-sibling phylogenetic networks, Bioinformatics 24 (13) (2008) 1481–1488.
[5] G. Cardona, F. Rosselló, G. Valiente, Comparison of tree-child phylogenetic networks, IEEE/ACM Trans. Comput. Biol. Bioinfo. 6 (4) (2009) 552–569.
[6] J.M. Chan, G. Carlsson, R. Rabadan, Topology of viral evolution, Proc. Natl. Acad. Sci. USA 110 (46) (2013) 18566–18571.
[7] P. Cordue, S. Linz, C. Semple, Phylogenetic networks that display a tree twice, Bull. Math. Biol. 76 (10) (2014) 2664–2679.
[8] J. Fakcharoenphol, T. Kumpijit, A. Putwattana, A faster algorithm for the tree containment problem for binary nearly stable phylogenetic networks, in: Proceedings of the the 12th International Joint Conference on Computer Science and Software Engineering, JCSSE'15, IEEE, 2015, pp. 337–342.
[9] A.R. Francis, M. Steel, Which phylogenetic networks are merely trees with additional arcs?, Syst. Biol. 64 (5) (2015) 768–777.
[10] P. Gambette, A.D.M. Gunawan, A. Labarre, S. Vialette, L. Zhang, Locating a tree in a phylogenetic network in quadratic time, in: Proceedings of the 19th Annual International Conference on Research in Computational Molecular Biology, RECOMB 2015, in: LNCS, vol. 9029, 2015, pp. 96–107.
[11] P. Gambette, A.D.M. Gunawan, A. Labarre, S. Vialette, L. Zhang, Solving the tree containment problem for genetically stable networks in quadratic time, in: Proceedings of the 26th International Workshop on Combinatorial Algorithms, IWOCA 2015, in: LNCS, vol. 9538, 2016, pp. 197–208.
[12] A.D.M. Gunawan, B. DasGupta, L. Zhang, Locating a tree in a reticulation-visible network in cubic time, in: Proceedings of the 20th Annual International Conference on Research in Computational Molecular Biology, RECOMB 2016, in: LNBI. vol. 9649, 2016, pp. 266–266. arXiv:1507.02119.
[13] A.D.M. Gunawan, B. DasGupta, L. Zhang, A decomposition theorem and two algorithms for reticulation-visible networks, Inform. and Comput. 252 (2017) 161–175.
[14] A.D.M. Gunawan, B. Lu, L. Zhang, A program for verification of phylogenetic network models, Bioinformatics 32 (17) (2016) i503–i510.
[15] A.D.M. Gunawan, L. Zhang, Bounding the size of a network defined by visibility property, 2015, http://arxiv.org/abs/1510.00115.
[16] D. Gusfield, ReCombinatorics: The Algorithmics of Ancestral Recombination Graphs and Explicit Phylogenetic Networks, The MIT Press, 2014.
[17] D.H. Huson, R. Rupp, C. Scornavacca, Phylogenetic Networks: Concepts, Algorithms and Applications, Cambridge University Press, 2011.
[18] P. Jenkins, Y. Song, R. Brem, Genealogy-based methods for inference of historical recombination and gene flow and their application in *saccharomyces cerevisiae*, PLoS ONE 7 (11) (2012) e46947.
[19] I.A. Kanj, L. Nakhleh, C. Than, G. Xia, Seeing the trees and their branches in the network is hard, Theoret. Comput. Sci. 401 (2008) 153–164.
[20] J. Kleinberg, E. Tardos, Algorithm Design, Pearson Education, 2006.
[21] T. Marcussen, K.S. Jakobsen, J. Danihelka, H.E. Ballard, K. Blaxland, A.K. Brysting, B. Oxelman, Inferring species networks from gene trees in high-polyploid north american and hawaiian violets (*viola*, Violaceae), Syst. Biol. 61 (2012) 107–126.
[22] L. Nakhleh, Phylogenetic Networks, University of Texas at Austin, U.S.A., 2004 Ph.D. thesis.
[23] L. Nakhleh, Computational approaches to species phylogeny inference and gene tree reconciliation, Trends Ecol. Evolut. 28 (12) (2013) 719–728.
[24] M. Steel, Phylogeny: Discrete and Random Processes in Evolution, SIAM, 2016.
[25] T.J. Treangen, E.P. Rocha, Horizontal transfer, not duplication, drives the expansion of protein families in prokaryotes, PLoS Genet. 7 (1) (2011) e1001284.
[26] L. van Iersel, C. Semple, M. Steel, Locating a tree in a phylogenetic network, Inform. Process. Lett. 110 (23) (2010) 1037–1043.
[27] L. Wang, K. Zhang, L. Zhang, Perfect phylogenetic networks with recombination, J. Comput. Biol. 8 (1) (2001) 69–78.