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

Philippe Gambette, Andreas D.M. Gunawan, Anthony Labarre, Stéphane Vialette, Louxin Zhang

▶ To cite this version:

Philippe Gambette, Andreas D.M. Gunawan, Anthony Labarre, Stéphane Vialette, Louxin Zhang. Solving the tree containment problem in linear time for nearly stable phylogenetic networks. Discrete Applied Mathematics, 2018, 246, pp.62-79. 10.1016/j.dam.2017.07.015 . hal-01575001

HAL Id: hal-01575001
https://hal.science/hal-01575001

Submitted on 26 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Solving the Tree Containment Problem in Linear Time for Nearly Stable Phylogenetic Networks☆

Philippe Gambette[a,1,], 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*

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.

Keywords: phylogenetic trees, phylogenetic networks, tree containment, reticulation visibility, nearly stable networks, genetically stable networks

1 ## 1. Introduction

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

6 A (*phylogenetic*) *network* over a set X of taxa is a rooted acyclic digraph with a set
7 of leaves (*i.e.*, vertices of outdegree zero) that are uniquely labeled with the taxa. Such a
8 network represents the evolutionary history of the taxa in X. In a network, the *tree vertices*
9 (*i.e.*, vertices of indegree one) represent speciation events, whereas the vertices of indegree

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

Email addresses: philippe.gambette@u-pem.fr (Philippe Gambette), a0054645@u.nus.edu (Andreas D. M. Gunawan), anthony.labarre@u-pem.fr (Anthony Labarre), stephane.vialette@u-pem.fr (Stéphane Vialette), matzlx@nus.edu.sg (Louxin Zhang)

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 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

51 the root and leaves. For a given network N, $\mathcal{L}(N)$ denotes its leaf set, $\mathcal{R}(N)$ its set of
52 reticulation vertices, $\mathcal{T}(N)$ its set of tree vertices (including leaves), $\mathcal{V}(N)$ the entire set of
53 vertices, and $\mathcal{E}(N)$ the entire set of edges. An edge is a *reticulation edge* if it is an edge
54 entering a reticulation vertex.

55 Let N be a network. For an edge set $E \subseteq \mathcal{E}(N)$, $N - E$ denotes the subnetwork with
56 vertex set $\mathcal{V}(N)$ and edge set $\mathcal{E}(N)\backslash E$. Similarly, for a vertex subset $S \subseteq \mathcal{V}(N)$, $N - S$
57 denotes the subnetwork with the vertex set $\mathcal{V}(N)\backslash S$ and the edge set $\{(u, v) \in \mathcal{E}(N) \mid u \notin$
58 $S, v \notin S\}$. When E or S contains only one element x, we simply write $N - x$.

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

61 We say that a network obtained by removing all but one incoming edge from each retic-
62 ulation in a network N is a *spanning tree* of N.

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

66 *2.2. Reticulation stability*

67 Consider a network N. Let x and y be two vertices of N. We say that x is a *parent* of
68 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
69 *ancestor* of y (or *above* y) and equivalently y is a *descendant* of x (or *below* x) if there is
70 a directed path from x to y in N. Two vertices are *siblings* if they are the children of the
71 same vertex.

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

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

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

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

83 *(1) If v has two reticulation children, then v is unstable.*

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

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

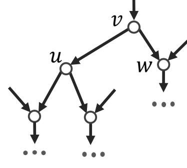

Figure 1: Two types of unstable tree vertices in a network.

(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.

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

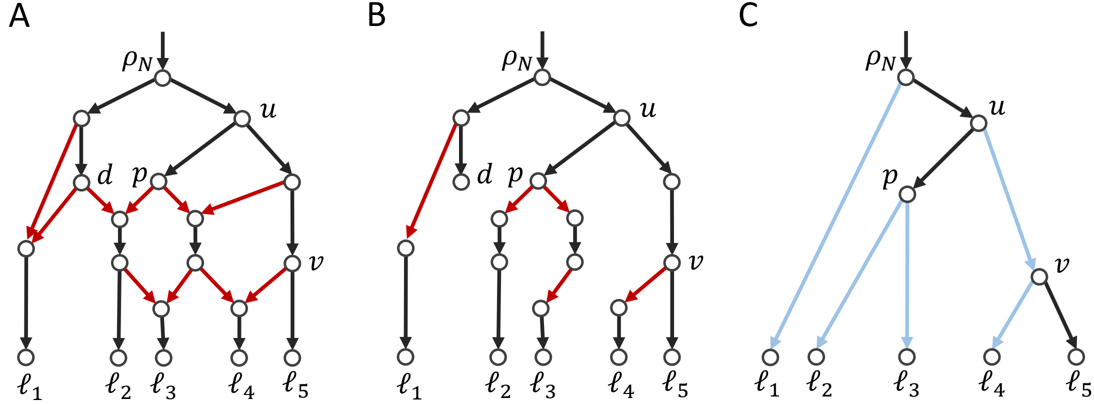

Figure 2: (**A**) A network N with five leaves labeled with ℓ_i ($1 \leq i \leq 5$). Reticulation edges are colored dark red. (**B**) A spanning tree T obtained by removing a reticulation edge from each reticulation in the network, in which there is a dummy leaf d. (**C**) A tree that is displayed in 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.

119 applying Hall's Theorem to a bipartite graph with one part consisting of tree vertices and
120 another consisting of reticulation vertices, and the edges being reticulation edges in N. Since
121 each reticulation vertex has two incoming reticulation edges and each tree vertex has at most
122 two outgoing reticulation edges, the existence of such a matching E follows from a theorem
123 of Alon [2, page 429]. □

124 *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 $\mathcal{V}(N)\backslash\{v\}$ and the following edge set:

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

125 Given a phylogenetic tree G and a network N, we say that N *displays* G if there is a
126 spanning tree T of N such that G is a *contraction* of T, i.e. T has the same vertex set as N
127 and G can be obtained from T by repeatedly applying contraction to all edges entering either
128 the "dummy" leaves or the vertices of both outdegree and indegree one, where a dummy
129 leaf is a leaf in T but not a leaf in N. Figure 2 shows an example of a network N and a
130 phylogenetic tree that is displayed by N.
131 The *tree containment* (TC) problem is to determine whether a given phylogenetic network
132 displays a given phylogenetic tree. We shall discuss the TC problem for network class defined
133 using vertex stability, namely the nearly stable networks.

134 3. Nearly stable and genetically stable networks

135 *3.1. Inclusion Relationship*

136 A network is *tree-child* if each of its internal vertices has a child that is a tree vertex [5].
137 A binary network is *tree-sibling* if every reticulation vertex has a sibling that is a tree

138 vertex [4, 22]. Cordue et al. [7] investigated reticulation-visible networks in which each
139 reticulation has at least a parent p that is connected to some leaf by a path consisting of
140 only tree vertices. Here, such networks are defined to be *nearly tree-child* networks.
141 Huson et al. [17, page 164] noted that if a network is tree-child, then all its vertices are
142 stable. We strengthen their result by proving the other direction.

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

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

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

149 **Proposition 3.2.**

150 *(1) Every genetically stable network is tree-sibling.*

151 *(2) Every tree-child network is nearly stable.*

152 *(3) Every nearly tree-child network is genetically stable.*

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

157 Based on Proposition 3.2, we summarize the relationships between the classes we study
158 and the other network classes for which the complexity of the TC problem is known in
159 Figure 3.
160 A reticulation-visible and tree-sibling network is not necessarily genetically stable (Fig-
161 ure 4.A). A genetically stable network is not necessarily nearly tree-child (Figure 4.B). A
162 nearly stable and nearly tree-child network is not necessarily tree-child (Figure 4.C). A
163 nearly tree-child network is not necessarily nearly stable (Figure 4.D).

164 *3.2. Class sizes*

165 Recombination histories of viruses, hybridization histories of plants, and histories of
166 horizontal gene transfers reported in the literature are often found to be nearly stable or
167 reticulation-visible (see e.g. the networks given in [18, 21] which are available at `http:
168 //phylnet.info/recophync/networkDraw.php`).
169 In order to evaluate whether the class of nearly stable networks is relevant in practice,
170 especially combined with the class of reticulation-visible networks for which there also exists
171 a polynomial-time algorithm solving the TC problem, we used a set of phylogenetic networks
172 randomly generated using a simulation program [1] and calculated the proportion of those
173 networks belonging to the classes[1]. Figure 5 summarizes the results of our simulation study.

[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/`.

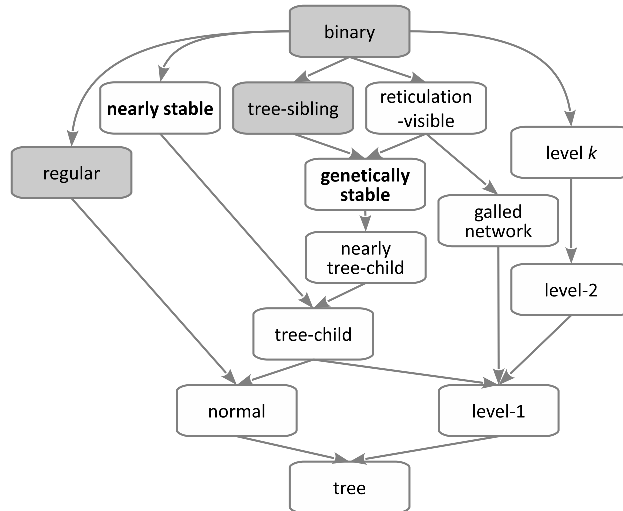

Figure 3: Inclusion relationships between classes of binary phylogenetic networks: an arrow from class A to class B means that A contains B. Class inclusions involving nearly stable and genetically stable are justified in this article (Proposition 3.2); for the other ones, references are available at `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.

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

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

180 A network with n leaves may contain an arbitrary large number of non-leaf vertices and
181 hence, unlike phylogenetic trees, its size is not bounded from above by a function of the
182 number of leaves. In [10], we proved that a reticulation-visible network with n leaves has
183 at most $10n - 9$ vertices (including leaves). Later, the tight size bound $8n - 7$ was proved

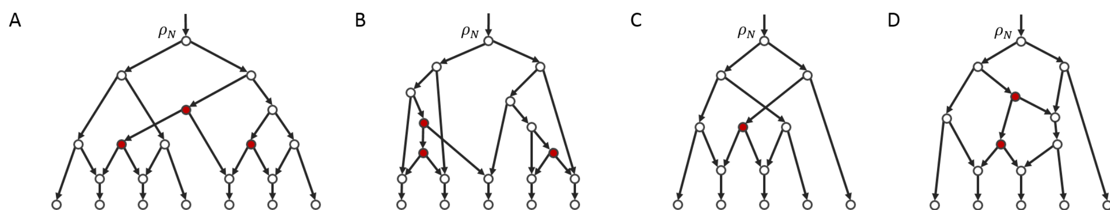

Figure 4: **(A)** A network which is reticulation-visible and tree-sibling, but not genetically stable. **(B)** A network which is genetically stable but neither nearly stable nor nearly tree-child. **(C)** A network which is nearly tree-child and nearly stable but not tree-child. **(D)** A network which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

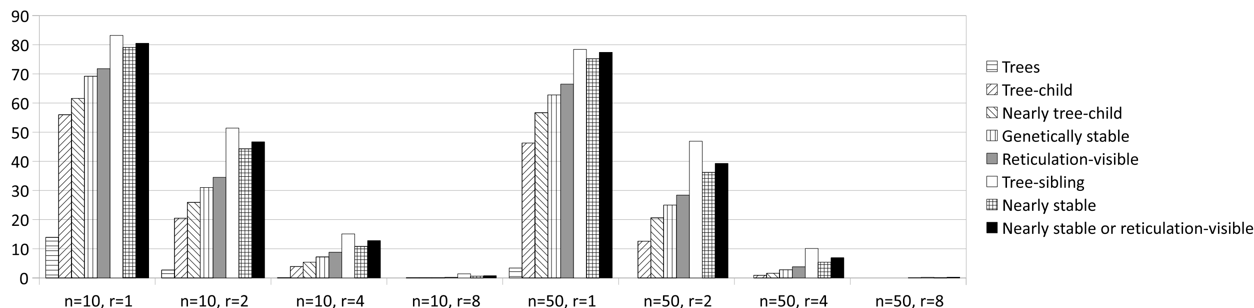

Figure 5: Percentage of binary phylogenetic networks on n leaves generated with the coalescent with recombination model (recombination rate r) in each class of phylogenetic networks. The corresponding data is available at `http://phylnet.info/recophync/`.

184 [3]. In this section, using the decomposition theorem introduced below, we shall first give
185 a short proof of this tight bound. We also show that nearly stable networks have the same
186 tight size bound as reticulation-visible networks, whereas genetically stable networks with
187 n leaves have the tight size bound $6n - 5$.

188 *4.1. A decomposition theorem*

189 Consider a network N. After the removal of all reticulations, N becomes a forest $N -$
190 $\mathcal{R}(N)$. One connected component of the forest is rooted at ρ_N, whereas the other components
191 are each rooted at the child of a reticulation (Figure 6). Since components consist of tree
192 vertices of N, they are called *tree vertex components* of N. We call them *big* tree vertex
193 components if they contain more than one vertex.

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

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

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

200 *(1) Each component C_j is rooted at a stable tree vertex. Additionally, a vertex is a compo-*
201 *nent root if and only if it is either the network root or the child of a stable reticulation.*

202 *(2) Each component C_j contains either a network leaf or the two parents of an inner*
203 *reticulation.*

204 *(3) Each component C_j contains at least two tree vertices if $C_j \neq \{\ell\}$ for any leaf $\ell \in \mathcal{L}(N)$.*

205 *Proof.* (1) By definition of a tree vertex component, its root r must be a tree vertex. If r is
206 the root of N, then it is obviously stable. Otherwise, as r is a tree vertex, then its parent
207 p is a reticulation (otherwise r is not a component root). If p is unstable, then r must be
208 stable as N is a nearly stable network, but this contradicts Proposition 2.2(1). Therefore,

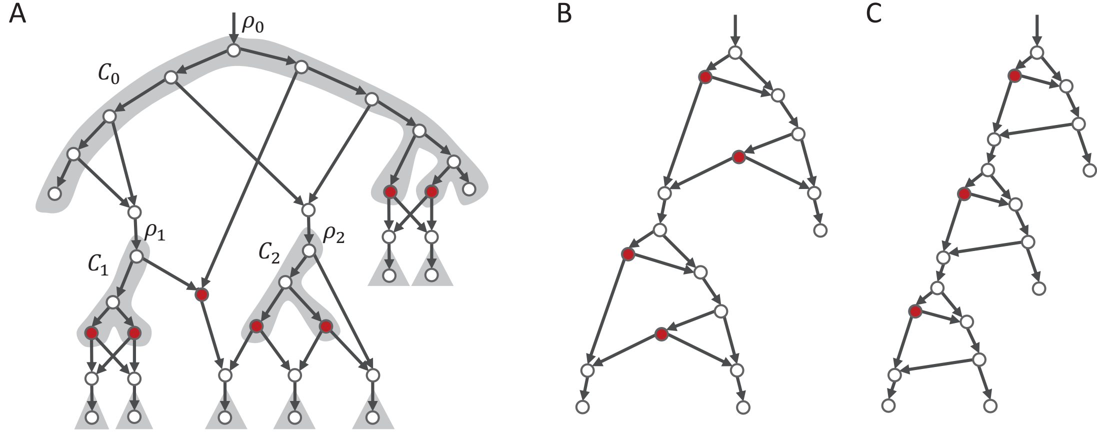

Figure 6: **(A)** The decomposition of a nearly stable network into ten tree vertex components. Seven components consist of a single leaf. The remaining three "big" components are C_0, C_1, and C_2, where ρ_i denotes the root of C_i for each $i = 1, 2, 3$. **(B)** A network with 3 leaves and 6 reticulations 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.

209 p is a stable reticulation vertex and r is a stable tree vertex according to the part (2) of
210 Proposition 2.2. Clearly, every child of a stable reticulation is a tree vertex, and thus is a
211 component root as well.

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

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

219 □

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

224 *4.2. Three size bounds*

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

227 **Theorem 4.2.** (i.) ([3]) *If N is reticulation-visible, then $r \leq 3(n-1)$.*

228 (ii.) *If N is reticulation-visible and tree-sibling, $r \leq 2(n-1)$. In particular, the bound*
229 *holds for genetically stable networks.*

230 (iii.) *If N is nearly stable, then $r \leq 3(n-1)$.*

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

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

235 Additionally, we let c be the number of tree vertex components of N for the rest of the
236 proof. By Theorem 4.1(1), $c = r + 1$.
237 Consider a component C. Since N is reticulation-visible, by Theorem 4.1(3), C contains
238 two distinct parents of an inner reticulation if it does not contain a network leaf. Hence,

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

239 Replacing t with $r + n - 1$ in Equation (2), we obtain that $r \leq 3(n - 1)$.
240
241 (ii.) Assume N is reticulation-visible and tree-sibling. We distinguish three types of tree
242 vertices of N by using T_i to denote the set of tree vertices with exactly i children being also
243 tree vertices for $i = 0, 1, 2$, respectively.
244 Since N is tree-sibling, each reticulation vertex x has a parent v_x such that v_x is a tree
245 vertex and has x and another tree vertex as its children. Therefore, mapping x to v_x is an
246 injective map from $\mathcal{R}(N)$ to T_1 and thus:

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

247 Consider a tree vertex component C of N that does not contain any network leaf. By
248 the fact (3) of Theorem 4.1, C contains a leaf v that differs from ρ_C. The children of v must
249 be reticulation vertices, and hence $v \in T_0$ (Fig 7.A). Since there are at most n tree vertex
250 components that contain one or more network leaves,

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

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

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

251 and thus $r \leq 2(n - 1)$.
252
253 (iii.) Let N be a nearly stable network. Given that each reticulation vertex may or may
254 not be stable, we use r_s and r_u to denote the numbers of stable and unstable reticulation
255 vertices of N, respectively.
256 First, for an unstable reticulation vertex, its parents are both stable tree vertices, as N is
257 nearly stable. By Corollary 2.1, a tree vertex with two reticulation children is unstable. This
258 implies that any two different unstable reticulation vertices have distinct parents. Therefore,
259 there are at least $2r_u$ stable tree vertices that have an unstable reticulation child (Fig 7.B).

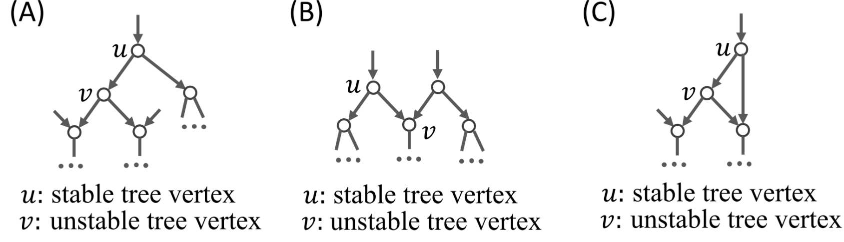

Figure 7: Illustration of three types of tree vertices in a nearly stable network. **(A)** An unstable tree vertex v with two reticulation children and a stable tree vertex u with two tree children. **(B)** A stable tree vertex u for which a child is an unstable reticulation vertex. (**C**) A stable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

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

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

264 as there are at most n components that contain at least a network leaf. Additionally, since N
265 is nearly stable, the parent u of v has to be a stable tree vertex. Since v has two reticulation
266 children, by Corollary 2.1, the stability of u implies that the other child of u has to be
267 either a tree vertex (Fig. 7.A) or a child of v (Fig. 7.C). Since there are at least $c - n$ tree
268 vertex components that do not contain any network leaf, there are at least $c - n$ stable tree
269 vertices such that their children are either two tree vertices or a tree vertex and one (stable)
270 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),

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

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:

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

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

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

275 Finally, it is not hard to see that in nearly tree-child network, each tree-vertex component
276 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 (Figure 8.A), 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 Figure 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.:

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

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.*

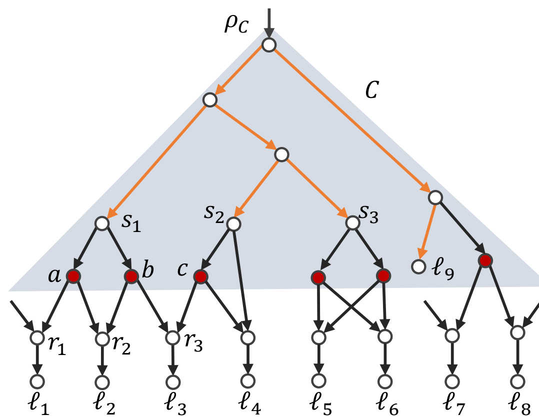

Figure 8: Illustration of the idea of our algorithm. A tree-vertex component C of a nearly stable network, where filled vertices are unstable ones and open edges have their end in other components. The orange subtree is the subtree spanned by the stable vertices with four leaves s_1, s_2, s_3, and ℓ_9.

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

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

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

306 *5.2. Two key lemmas*

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

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

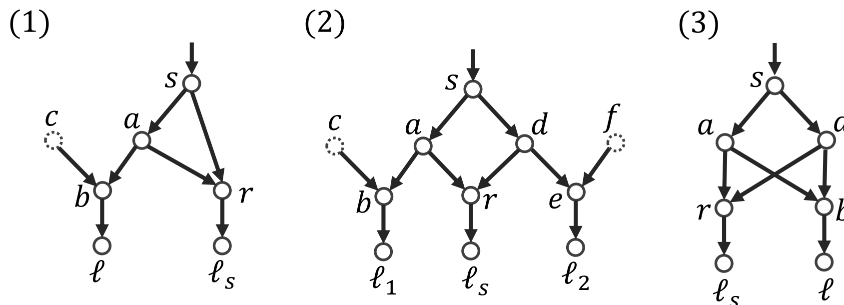

Figure 9: Three possible mini-structures below a vertex $s \in S(C)$ in a lowest component C. Here, the vertices c and f have undetermined vertex type, each of them can be a tree or reticulation vertex.

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

314 *Proof.* Assume s is stable on a leaf ℓ_s. The leaf ℓ_s cannot be in C. Otherwise all the vertices
315 in the path from s to ℓ_s are stable, contradicting the fact that $s \in S(C)$. Thus, ℓ_s is the
316 unique child of an inner reticulation r below s. Let u be a parent of r. If u is a reticulation,
317 it is then unstable and its parents are two tree vertices equal to or below s in C. Since
318 $s \in S(C)$, u and each of its parents below s are unstable, contradicting that N is nearly
319 stable. Hence the parents of r are two tree vertices below s in C.

320 Since N does not contain parallel edges, at least a parent of r is not s. Let a be such a
321 parent of r. Note that a is a child of s, otherwise, since $s \in S(C)$, any parent of the tree
322 vertex a below s in C would not be a leaf of C, therefore contradicting Proposition 5.2. The
323 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 Figure 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 Figure 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 Figure 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 Figure 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.

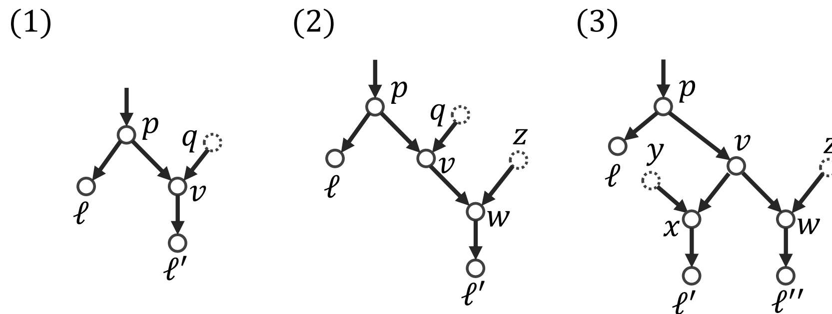

Figure 10: Three possible mini-structures below a tree vertex for which one child is a network leaf and the other is not a stable tree vertex in a lowest component. The vertices q, y and x have undetermined vertex type.

350 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
351 (resp. not) siblings in G. We also use $\text{par}_G(v)$ to denote the parent of v in G.
352 First, we consider the mini-structure (1) in Figure 10, which is named the "uncle-nephew
353 structure" in [10], where two leaves ℓ and ℓ' have the uncle and nephew relationship and q
354 is either a tree or reticulation vertex.

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

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

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

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

359 1. $\ell \wedge_G \ell'$: note that T contains either (p, v) or (q, v) exclusively. If T contains (p, v), N'
360 displays G, as T is also a spanning tree of N'.

361 If T contains (q, v), let h be the lowest common ancestor of ℓ and ℓ' in T. Since any
362 path to ℓ must contain p, p must be in the path from h to p in T and $T[h]$ only has
363 two labeled leaves, namely ℓ and ℓ'.

364 Define $T' = T - (q, v) + (p, v)$. Then, the vertex p becomes the lowest common ancestor
365 of ℓ and ℓ' in T', so $T[h]$ and $T'[h]$ do not contain any network leaves other than ℓ and
366 ℓ'. Any edge of T that is not below h is also an edge in T' and conversely, so G is a
367 contraction of both T and T'. The tree T' is a spanning tree of $N - (q, v)$, and thus
368 is the evidence that $N - (q, v)$ displays G.

369 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
370 in T, contradicting that ℓ and ℓ' are not sibling in G. Thus, T does not contain (p, v)
371 and T is also a spanning tree of $N - (p, v)$, implying that N' also displays G.

372 □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

394 We first claim that $N-(d,r)$ displays G. If (d,r) is not in T, then the claim is true. If
395 (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
396 a contraction of $T-(d,r)+(a,r)$, and $N-(d,r)$ displays G.

397 Next, note that there is essentially an uncle-nephew structure on the leaves ℓ_1 and ℓ_s in
398 $N-(d,r)$. Hence, the assumption that $\ell_1 \wedge_G \ell_s$ implies that $N - \{(d,r),(c,b)\}$ displays G.

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

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

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

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

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 ℓ ∧_G ℓ_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 ℓ_1 ∧_G ℓ_s nor ℓ_2 ∧_G ℓ_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 ℓ_1 ∧_G ℓ_s and par_G(ℓ_s) ∧_G ℓ_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 ℓ_1 ∧_G ℓ_s but par_G(ℓ_s) ⋀̸_G ℓ_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 ℓ_2 ∧_G ℓ_s and par_G(ℓ_s) ∧_G ℓ_1 then
17              delete (c, b), (a, r), and (f, e);
18              contract the edge(s) entering c or f;
19          else if ℓ_2 ∧_G ℓ_s but par_G(ℓ_s) ⋀̸_G ℓ_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 ℓ ∧_G ℓ_s then
24              skip;
25          else
26              output "G is not displayed" and exit;
27      contract N[s] into ℓ_s;
28      contract G[par_G(par_G(ℓ_s))] or G[par_G(ℓ_s)] into ℓ_s if necessary;
```

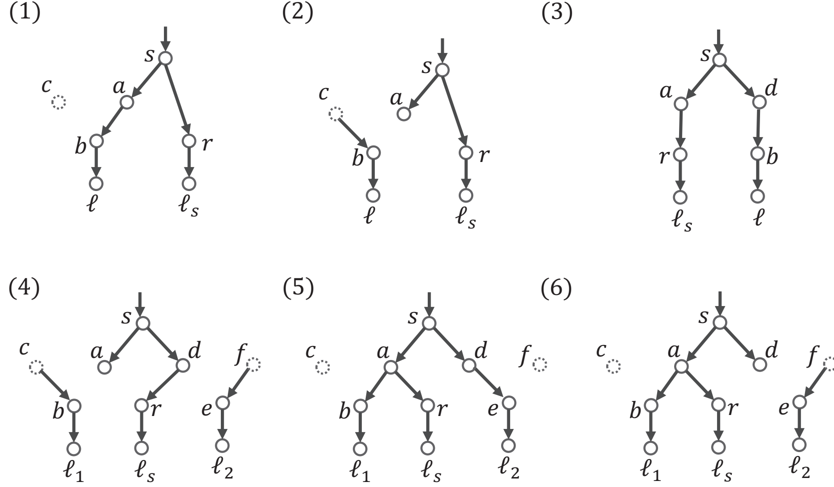

Figure 11: The resulting networks after $N[s]$ is simplified. (**1**) and (**2**) for the two cases of the mini-structure (1). (**3**) for the mini-structure (3). (**4**)-(**6**) for the first three cases of the mini-structure (2). The cases 4 and 5 of the mini-structure (2) are symmetric to the cases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

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 Figure 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 Figure 9.

If $N[s]$ has the mini-structure (1) in Figure 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 (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 Figure 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 Figure 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 Figure 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.*

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

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

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

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

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

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

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

478 *Proof.* N' is a subnetwork of N, so clearly N displays G if N' displays G.
479 Assume that N displays G. There exists a spanning tree T of N such that G is a
480 contraction of T.
481 By Lemma 5.2, there are three possible mini-structures for $N[p]$ shown in Fig. 10. The
482 mini-structure (1) is an uncle-nephew structure and thus the sufficiency follows from
483 Lemma 5.3. The sufficiency for the mini-structure (2) in Fig. 10 can be proven similarly.
484 Suppose $N[p]$ has the mini-structure (3) in Fig. 10, the first and second case in the
485 definition of N' are symmetric, so we only need to consider the cases 1, 3 and 4.

486 1: $\ell' \wedge_G \ell$: since T must contain p, T does not contain (v, w). Otherwise, ℓ'' is the sibling
487 of either ℓ' or ℓ depending whether or not (v, x) is in T, contradicting the assumption
488 in this case. This implies that T is a spanning tree of $N - (v, w)$.

489 In $N - (v, w)$, $(N - (v, w))[p]$ is essentially an uncle-nephew structure. By Lemma 5.3,
490 $N' = (N - (v, w)) - (y, x)$ displays G.

491 3: $\ell' \wedge_G \ell''$ and $\mathrm{par}_G(\ell') \wedge_G \ell$.

492 Notice that T contains p. If T contains exactly one of (v, x) and (v, w), ℓ is a sibling
493 of either ℓ' or ℓ'' in G, contradicting the assumption in this case.

494 Therefore, either both (v, x) and (v, w) are in T or none of them is in T. If the former
495 holds, then T is a spanning tree of N' and N' also displays G. If the latter holds, let
496 t_1 denote the lowest common ancestor of ℓ' and ℓ'' and let t_2 be the lowest common
497 ancestor of t_1 and ℓ in T. Then, $T[t_2]$ contains only three labeled leaves: ℓ, ℓ', and ℓ''.
498 Let $T' = T - \{(y, x), (z, w)\} + \{(v, x), (v, w)\}$. The tree in which ℓ' and ℓ'' are siblings
499 and ℓ is their uncle is a contraction of $T'[t_2]$ and $T[t_2]$, and T' is a spanning tree of
500 $N - \{(y, x), (z, w)\}$. Therefore, $N - \{(y, x), (z, w)\}$ displays G.

501 4: All the first three cases are not true.

502 As shown in the third case, either both (v, x) and (v, w) are both in T or none of them
503 is in T. If (v, x) and (v, w) are both in T, then ℓ' and ℓ'' are sibling and their parent
504 is a sibling of ℓ in G, contradiction. Therefore, T is a spanning tree of N'. Hence, N'
505 also display G.

506 □

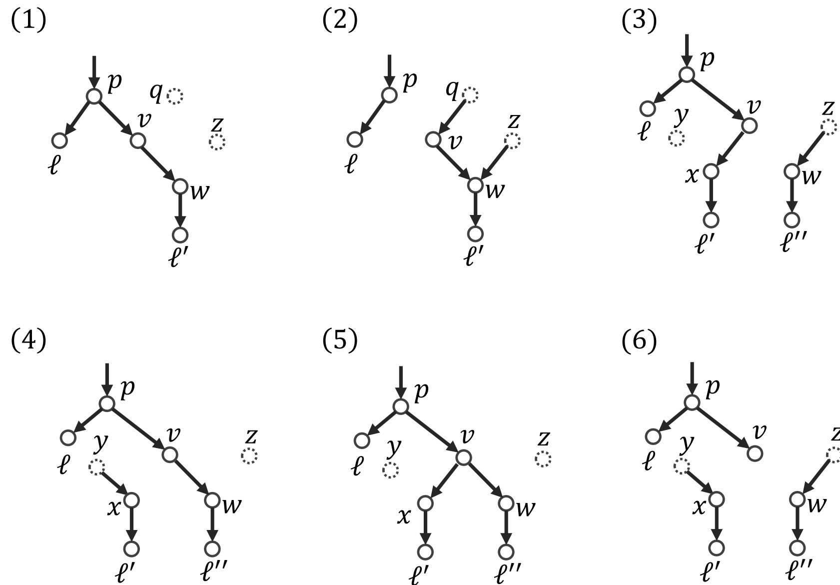

Figure 12: The resulting networks after $N[p]$ is simplified using each of the rule in Lemma 5.5. (**1**) and (**2**) for the two cases of the mini-structure (2) in Fig. 10. (**3**)-(**6**) for the four cases of the mini-structure (3) in Fig. 10.

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

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

517 *5.4. The algorithm*

518 We are now ready to describe a linear-time TC algorithm (Algorithm 4) for nearly stable
519 networks. It is divided into the pre-processing and simplification parts. Roughly speaking,

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 $G[\text{par}(\ell)]$ into ℓ if necessary (Fig. 12).

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
1   if Case 1 or 2 holds (Fig. 13.1 or Fig. 13.2) then
2       skip;
3   if Case 3 or 4 holds (Fig. 13.3 or Fig. 13.4) then
4       if ℓ ∧_G ℓ' then
5           skip;
6       else
7           output "No" and exit;
8   if Case 5 holds (Fig. 13.5) then
9       if (ℓ' ∧_G ℓ'') and (par_G(ℓ') ∧_G ℓ) then
10          skip;
11      else
12          output "No" and exit;
13  if Case 6 holds (Fig. 13.6) then
14      if ℓ ∧_G ℓ' then
15          delete (z, w);
16      else
17          delete (v, w);
18  if Case 7 holds (Fig. 13.7) then
19      if (ℓ' ∧_G ℓ'') and (par_G(ℓ') ∧_G ℓ) then
20          delete (z, w);
21      else if ℓ' ∧_G ℓ then
22          delete (v, w);
23      else
24          output "No" and exit;
25  contract N[p] into ℓ; contract G[par_G(ℓ)] into ℓ if necessary.
```

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 ≤ i ≤ k) so
    that ρ(C_j) is a descendant of ρ(C_i) only if i < j;
    /* Some big components may contain only leaves that are in 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 ∈ 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, ..., u_t = ρ(C))
12      foreach ℓ = 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 ℓ ⋪_G v then
17                  return false;
18              else if v has been visited then
19                  contract N[par(ℓ)] and G[par(ℓ)] 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;
```

520 the algorithm first topologically sorts the big tree vertex components and then dissolves these
521 components one by one in a bottom-up manner. Each component is dissolved in two stages.
522 In the first stage , Dissolve_Lowest_Stable_Vertices is called for each $s \in S(C)$. In the second
523 stage, Dissolve_SubntkNearNtkLeaf and Dissolve_DegenerateCases are repeatedly called. In
524 the rest of this section, we discuss the correctness and running time of the algorithm step
525 by step.

526 Let N and G be the input nearly stable network and phylogenetic tree with the same set
527 of n labeled leaves, respectively. Then, $|\mathcal{V}(N)| = O(n)$, which is proved in Section 4, and
528 hence $|\mathcal{E}(N)| = O(n)$, as N is degree-bounded. If G is displayed in N, $|\mathcal{V}(G)| \leq |\mathcal{V}(N)|$.
529 We shall show that the proposed TC algorithm takes $O(n)$ operations. Here, the operations
530 include (i) edge contraction, (ii) edge deletion, (iii) membership check for $\mathcal{T}(N)$, $\mathcal{R}(N)$ and
531 $\mathcal{L}(N)$, (iv) verification of sibling or parent-son relationship and (v) traverse to a vertex in
532 N or G.

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

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

542 For a component C, by Proposition 5.2, a vertex x in $S(C)$ if and only if each child of x
543 in C is a leaf of C but not a network leaf. Hence, when traversing C in pre-order, a vertex
544 x is in $S(C)$ if and only if (a) the next vertex is in $\mathcal{L}(C) \backslash \mathcal{L}(N)$ and (b) the next vertex
545 is either not a child of x or also in $\mathcal{L}(C) \backslash \mathcal{L}(N)$, each of which can be checked in constant
546 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
547 all $k+1$ big tree-vertex components of N, which is at most $|\mathcal{V}(N)|$.

548 For each component C, $|S(C)| \leq |\mathcal{L}(C)|$ and thus $|S(C)| \leq |\mathcal{V}(C)|/2$. By the **foreach**
549 loop on line 5–9, we simplify the networks $N[s]$ below $s \in S(C)$ one-by-one. Since there
550 are only three possible mini-structures in Figure 9, each containing 4 to 6 vertices, the

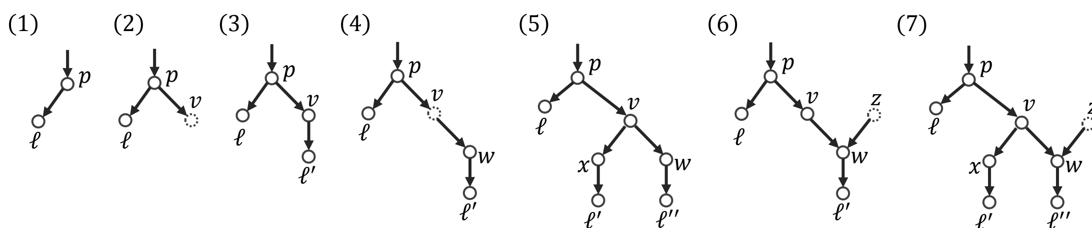

Figure 13: The possible degenerated structures of the subnetwork below the parent of a network leaf in a lowest tree component C. In (2) and (4), v can be either an unstable tree vertex or a reticulation vertex.

551 conditional statement can be verified in constant time. If the conditional statement is
552 true, calling the procedure Dissolve_Lowest_Stable_Vertices on s takes constant time. If the
553 conditional statement is false, $N[s]$ has been modified when working on some $s' \in S(C)$
554 earlier. By Proposition 5.3, s is no longer in $S(C)$ or $N[s]$ is equivalent to a length-2 path
555 from s to ℓ_s on which s is stable. Hence, the simplification of $N[s]$ also takes constant
556 time. Overall, the first **foreach** loop takes $O(|S(C)|)$ operations for each C. Hence the
557 total number of the operations taken by the algorithm is $O(|\mathcal{V}(N)|)$ for line 5–9. Note that
558 when the leaves of C are all network leaves, $S(C) = \emptyset$ and the **foreach** loop will stop.

559 $S(C)$ is empty when line 10 starts. Then, either an internal tree vertex or its parent is
560 in the path from $\rho(C)$ to a network leaf. Moreover, if an internal tree vertex is stable, as
561 assumed in the beginning of this step, its left child is also a stable tree vertex.

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

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

568 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
569 siblings in G, we have the evidence that G is not displayed and hence the algorithm
570 terminates. If ℓ and v are siblings in G, then, the algorithm simply moves on to process
571 $v = u_{i+1}$. At v, the algorithm will replace p with v and also replace the parent of ℓ and
572 v with v in G. As such, when the algorithm processes u_{i+2}, it has become a network
573 leaf now and $\{u_{i+2}, \ldots, u_t\}$ is still the vertex set of the modified component, listed in
574 the post-order. This explain why the algorithm simply does not act when visiting ℓ in
575 this case.

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

581 C3. The vertex v is a tree vertex but unstable. Then, $v = u_{i+1}$ and $N[p]$ has the mini-
582 structure (3) in Fig. 10 or a degenerated structure in Fig. 13. Similar to the case C2,
583 after Dissolve_SubntkNearNtkLeaf or Dissolve_DegenerateCases is called on ℓ, $\{p =$
584 $u_{i+2}, \ldots, u_t\}$ is the vertex set of the modified component listed in the post-order.

585 C4. The vertex v is an internal, stable tree vertex in C. This indicates that the subnetwork
586 $N[v]$ contains at least one stable tree vertex and possibly some reticulation vertices.
587 In this case, u_{i+1} is the leftmost network leaf in $C[v]$. The algorithm will move on to
588 simplify $C[v]$ into a network leaf and then contracting the parent of both ℓ and v into
589 a network leaf. In order to simplify $C[v]$ first, the algorithm simply skips at ℓ.

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

591 Taken together, the above facts show the following theorem.

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

594 6. Conclusion

595 In this paper, we have studied nearly stable and genetically stable networks, two classes
596 that were introduced in the study of the tree containment problem. In particular, we have
597 proved that each nearly (resp. genetically) stable network with n leaves contains at most
598 $3(n-1)$ (resp. $2(n-1)$) reticulation vertices. We have also designed a linear-time TC
599 algorithm for binary nearly stable networks.

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

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

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

613 7. Acknowledgments

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

616 References

617 [1] Arenas, M., Valiente, G., Posada, D., 2008. Characterization of reticulate networks based on the coa-
618 lescent with recombination. Molecular Biology and Evolution 25 (12), 2517–2520.
619 [2] Bondy, J. A., Murty, U. S. R., 2008. Graph Theory. Springer.
620 [3] Bordewich, M., Semple, C., 2016. Reticulation-visible networks. Advances in Applied Mathematics 78,
621 114–141.
622 [4] Cardona, G., Llabrés, M., Rosselló, F., Valiente, G., 2008. A distance metric for a class of tree-sibling
623 phylogenetic networks. Bioinformatics 24 (13), 1481–1488.
624 [5] Cardona, G., Rosselló, F., Valiente, G., 2009. Comparison of tree-child phylogenetic networks.
625 IEEE/ACM Trans. Comput. Biol. Bioinfo. 6 (4), 552–569.
626 [6] Chan, J. M., Carlsson, G., Rabadan, R., 2013. Topology of viral evolution. PNAS 110 (46), 18566–
627 18571.
628 [7] Cordue, P., Linz, S., Semple, C., 2014. Phylogenetic networks that display a tree twice. Bull. Math.
629 Biol. 76 (10), 2664–2679.

630 [8] Fakcharoenphol, J., Kumpijit, T., Putwattana, A., 2015. A faster algorithm for the tree containment
631 problem for binary nearly stable phylogenetic networks. In: Proceedings of the The 12th International
632 Joint Conference on Computer Science and Software Engineering (JCSSE'15). IEEE, pp. 337–342.
633 [9] Francis, A. R., Steel, M., 2015. Which phylogenetic networks are merely trees with additional arcs?
634 Systematic Biology 64 (5), 768–777.
635 [10] Gambette, P., Gunawan, A. D. M., Labarre, A., Vialette, S., Zhang, L., 2015. Locating a tree in a
636 phylogenetic network in quadratic time. In: Proceedings of the 19th Annual International Conference
637 on Research in Computational Molecular Biology (RECOMB 2015). Vol. 9029 of LNCS. pp. 96–107.
638 [11] Gambette, P., Gunawan, A. D. M., Labarre, A., Vialette, S., Zhang, L., 2016. Solving the tree contain-
639 ment problem for genetically stable networks in quadratic time. In: Proceedings of the 26th Interna-
640 tional Workshop on Combinatorial Algorithms (IWOCA 2015). Vol. 9538 of LNCS. pp. 197–208.
641 [12] Gunawan, A. D. M., DasGupta, B., Zhang, L., 2016. Locating a tree in a reticulation-visible network in
642 cubic time. In: Proceedings of the 20th Annual International Conference on Research in Computational
643 Molecular Biology (RECOMB 2016). Vol. 9649 of LNBI. pp. 266–266, arXiv:1507.02119, 2015.
644 [13] Gunawan, A. D. M., DasGupta, B., Zhang, L., 2017. A decomposition theorem and two algorithms for
645 reticulation-visible networks. Information and Computation 252, 161–175.
646 [14] Gunawan, A. D. M., Lu, B., Zhang, L., 2016. A program for verification of phylogenetic network models.
647 Bioinformatics 32 (17), i503–i510.
648 [15] Gunawan, A. D. M., Zhang, L., 2015. Bounding the size of a network defined by visibility property.
649 http://arxiv.org/abs/1510.00115.
650 [16] Gusfield, D., 2014. ReCombinatorics: The Algorithmics of Ancestral Recombination Graphs and Ex-
651 plicit Phylogenetic Networks. The MIT Press.
652 [17] Huson, D. H., Rupp, R., Scornavacca, C., 2011. Phylogenetic Networks: Concepts, Algorithms and
653 Applications. Cambridge University Press.
654 [18] Jenkins, P., Song, Y., Brem, R., 2012. Genealogy-based methods for inference of historical recombina-
655 tion and gene flow and their application in *saccharomyces cerevisiae*. PLoS ONE 7 (11), e46947.
656 [19] Kanj, I. A., Nakhleh, L., Than, C., Xia, G., 2008. Seeing the trees and their branches in the network
657 is hard. Theoretical Comput. Sci. 401, 153–164.
658 [20] Kleinberg, J., Tardos, E., 2006. Algorithm Design. Pearson Education.
659 [21] Marcussen, T., Jakobsen, K. S., Danihelka, J., Ballard, H. E., Blaxland, K., Brysting, A. K., Oxelman,
660 B., 2012. Inferring species networks from gene trees in high-polyploid north american and hawaiian
661 violets (*viola*, violaceae). Systematic Biology 61, 107–126.
662 [22] Nakhleh, L., 2004. Phylogenetic networks. Ph.D. thesis, University of Texas at Austin, U.S.A.
663 [23] Nakhleh, L., 2013. Computational approaches to species phylogeny inference and gene tree reconcilia-
664 tion. Trends Ecol. Evolut. 28 (12), 719–728.
665 [24] Steel, M., 2016. Phylogeny: Discrete and Random Processes in Evolution. SIAM.
666 [25] Treangen, T. J., Rocha, E. P., 2011. Horizontal transfer, not duplication, drives the expansion of protein
667 families in prokaryotes. PLoS Genetics 7 (1), e1001284.
668 [26] van Iersel, L., Semple, C., Steel, M., 2010. Locating a tree in a phylogenetic network. Inf. Process. Lett.
669 110 (23), 1037–1043.
670 [27] Wang, L., Zhang, K., Zhang, L., 2001. Perfect phylogenetic networks with recombination. J. Comp.
671 Biol. 8 (1), 69–78.