On normal networks

by

Devin Robert Bickner

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Mathematics

Program of Study Committee:

Stephen Willson, Major Professor

David Fernández-Baca

Clifford Bergman

Ryan Martin

Zhijun Wu

Iowa State University

Ames, Iowa

2012

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Phylogenetic trees have long been the standard object used in evolutionary biology to illustrate how a given set of species are related. Evidence is mounting to suggest that *hybridization*, historical events when multiple species merge to form new species, are prevalent enough to warrant inclusion into the field. Phylogenetic *networks* allow for this possibility.

In this paper, we discuss *normal* networks, a specific type of network with desirable tree-like properties. We find tight upper and lower bounds for certain aspects of the networks, including the number of edges, normal edges, hybrid vertices, parents of a vertex, and children of a vertex. We also find tight upper and lower bounds on the number of vertices and edges of specific cases of normal networks, as well as various interesting, related results that lead to these counts.

We discuss the *tree containment problem*, which asks whether a given network contains the information contained within a given tree. We give an algorithm and prove that the tree containment problem for normal networks is solvable in polynomial time.

We also discuss new operations on normal networks that are based off of the subtree-pruning and regrafting operation, a standard phylogenetic tree operation. These new operations allow for us to navigate through normal network *space*, a graph that represents all normal networks with a given set of leaves in which an edge connecting two networks is present if one network can be obtained from the other using exactly one of the operations discussed. We show that these operations connect binary normal network space, the normal network space in which the normal networks have no more than two edges going into or out of each of vertex. These operations on this network space can be used to give better upper bounds on the number of binary normal networks. We show a few of these upper bounds, as well as compare them to upper bounds of trees and *regular* networks, a type of network that contains normal networks.

Finally, we discuss some work that might be pursued based off of the results in this paper.

CHAPTER 1. INTRODUCTION

1.1 Background

Much work has been done in the study of the history of species. One of the more common assumptions that is made is that history is in the form of a tree. The "tree of life" is an attempt to relate all species through an evolutionary tree. Many scientists use trees to explain evidence that they find in the field. In a tree, it as assumed that each species in history evolved from exactly one parent species, and that any other ancestors of the first species are also ancestors of the parent species. Each day, however, more evidence is mounting that suggests that history is not quite so simple. It might not be the case that each new species is created from a single parent species (11). It might be that different genes suggest different trees relating a set of species. For example, in (14), trees are created from different genes common to a set of species, with each gene resulting in a different phylogenetic gene tree explaining the data. If we wish to represent the history of the species in a single species tree, we will certainly contradict information from at least one of the trees. Perhaps, instead, evolutionary history can be explained with a *network*. Networks, which are a generalization of trees that allow vertices with multiple parents, allow a graph structure to contain information found in many different trees by allowing *hybridization*, where species inherit genes from multiple parent species, *lateral gene transfer*, where organisms obtain genetic material from other organisms without actually being their offspring (10), or other such events, while still maintaining a flow of time.

Of course, if we remove an assumption as strong as assuming that history is explainable through trees, we expect to lose some nice properties. Whereas a tree is very structured, with a predictable number of vertices and edges, along with easily obtainable information about descendants, a network, in general, is not nearly as well-behaved, and requires some restrictions

to keep the networks from getting unreasonably large. There are several theories and methods that attempt to explain the possible differences between a species tree and gene trees that do not use networks. A *consensus tree* method usually assumes that all of the input trees are equally important and tries to find a new tree that is similar to the largest number of the input trees, usually looking at clusters in the input trees. Some details and inherent flaws are discussed in (17). *Reconciled trees*, discussed in (12), explain differences in gene trees primarily through gene duplication and deletion; a species tree is found such that each gene tree can be transformed into the species tree through a number of gene duplications and deletions, with fewer such events being more desirable. On the other hand, some believe that using gene trees to determine species trees is flawed. (4) and (15) discuss an *anomalous zone* on trees with more than five taxa, arguing that any method that constructs a species tree from gene trees with five or more taxa will lead to species trees that are different from the most likely gene tree. All of this suggests that networks seem like a very important and powerful tool in genetics. We will explore a special class of networks in detail, with the hope that future work will focus on both networks and trees, rather than just the latter.

There are several types of networks currently being studied. Some examples include *regular* networks, *level-k* networks, *galled* networks, *tree-child*, and *normal* networks. We will discuss these in greater detail later. The focus of this paper is mostly upon normal networks. Normal networks, just like all of the types of networks, are relatively new in the field of phylogenetics, when compared to trees. In the next chapter, we discuss some basic properties of normal networks, including bounds on the numbers of vertices and edges, as well as special cases of normal networks. In chapter three, we discuss the *tree containment problem*, which asks whether or not a given tree is "contained" in a given normal network. In chapter four, we build upon a well-known tree operation, rSPR (rooted subtree pruning and regrafting), which allows us to transform trees into other trees with the same set of species being studied. We expand this operation to include two other operations that allow a similar discussion about normal networks. We discuss normal network *space*, which is a way to visualize the set of all possible normal networks on the same set of species. We close with some discussion, as well as some future possibilities for study.

1.2 Preliminaries

In this section, we discuss the plethora of definitions that are needed in the main sections of the paper.

A *network* N is a pair $N = (V, E)$ consisting of a set of vertices V, along with directed edges E of the form (u, v) representing an edge from vertex u pointing to vertex v. A *leaf* is a vertex in V such that no edges exist going out from the vertex into any other vertex. Vertices that are not leaves are called *internal* vertices. The leaves will represent the species being studied. A *phylogenetic* X-network is a pair (N, X) where N is a network and X is set of distinct labels for the leaves of N for which there is a bijection between the elements of X and the leaves of the network. We will often abuse this definition and refer to phylogenetic X-networks simply as networks.

A *path* is a sequence of vertices $u = x_1, x_2, ..., x_n = v$ such that $(x_i, x_{i+1}) \in E$ for each i where $1 \leq i \leq n-1$. A (u, v)-*path* is a path beginning at u and ending at v. If such a path exists, u is called an *ancestor* of v and v is called a *descendant* of u. We write $u \leq_N v$ in both cases, omitting the N if the context is clear and use $u <_N v$ if $u \neq v$. Note that we allow the trivial path from u to itself. If a (u, v)-path consists of only two vertices, (i.e. $(u, v) \in E$ is the only edge in the path), then u is called a *parent* of v, and v is called a *child* of u. The edge (u, v) is called a *parent edge* of v. Two vertices u and v are *related* if there exists a path from one vertex to the other, that is either $u \leq v$ or $v \leq u$. Vertices are *unrelated* if no path exists between the vertices. A path has *length* t if if there are $t + 1$ vertices in the path (thus spanning t edges). A network whose vertices all have at most two children and at most two parents is called *binary*. By *closest* ancestor or descendant with a given property, we mean those with minimal path distances that satisfy the given conditions. A vertex m is called a *common ancestor* of two other vertices u and v if m is an ancestor of both u and v. Furthermore, m is called a *most recent* common ancestor (MRCA) of u and v if it is not an ancestor of any other common ancestor of u and v.

A *cycle* is a path that passes through at least two distinct vertices and whose first and last vertices are the same. A graph is *acyclic* if there are no cycles. A *rooted*, acyclic network is a

network with a vertex r such that, for each vertex v, there is an (r, v)-path. In such a network, the *root* r is the unique vertex that has no edges going into it. A *simple* network is one whose edge set E does not contain any trivial edges (v, v). We will assume that all of our networks are rooted, acyclic, and simple.

A vertex v is called *hybrid* if there are at least two vertices u and w such that (u, v) and (w, v) are both edges. The vertices u and v are called *hybrid parents* of u, and u is a *hybrid child* of u and of v. A vertex that is not hybrid is called a *normal* vertex. If (u, v) is an edge from u to a normal vertex v, then v is called a *normal child* of u. The word *normal*, sometimes replaced with *tree*, is used to suggest that the object in question is similar to what you might see in a tree, whereas *hybrid* objects are those that do not appear in trees and are special to networks.

The *degree* of a vertex is the total number of edges going into and out of the vertex. The *in-degree* of a vertex is the total number of edges going into the vertex. The *out-degree* is the number of edges going out of the vertex. By *total* in-degree for a set of vertices, we will mean the sum of the in-degrees of the vertices in the set. Similarly, the total out-degree for a set of vertices is the sum of the out-degrees of the vertices in the set. A vertex with precisely one parent and one child, which is a vertex of in- and out-degree 1, is called a 1-*vertex*. We will often require that networks not have any 1-vertices, since they offer little biological information, and cause a lot of troubles with counting.

A *normal* edge is an edge going into a normal vertex. A (u, v)-path is called *normal* if all vertices on the path from u to v, but not necessarily u itself, are normal vertices. If v is a leaf and there is a normal (u, v)-path, then we call v a *normal-path leaf descendant* (NPLD) of u. In this paper, we will say that a leaf is trivially a NPLD of itself, even if the leaf is hybrid. A *hybrid* edge is an edge going into a hybrid vertex. An edge (u, v) is called *redundant* if there is a (u, v)-path that does not contain the edge (u, v). A redundant edge also causes many counting problems, and will be avoided when possible. And, since the (u, v)-path already implies that u is an ancestor of v, the information in the edge (u, v) is somewhat redundant, as the name suggests.

The *cluster* of a vertex v is the set $\mathrm{cl}(v, N) = \{x \in X : \text{a path exists in } N \text{ from } v \text{ to } x\}$.

Again, we will write $\mathrm{cl}(v)$ if the context is clear. It is easy to see that, if $u \leq v$, then $\mathrm{cl}(u) \supseteq \mathrm{cl}(v)$. We call a cluster containment $\mathrm{cl}(u) \subset \mathrm{cl}(v)$ *maximal* if it is maximal in the set theoretic sense (i.e., the sets are distinct and no other clusters $\mathrm{cl}(w)$ for any vertex w in the same network fit in $\mathrm{cl}(u) \subset \mathrm{cl}(w) \subset (v)$). The *set* of clusters $\mathrm{Cl}(N)$ of a network N is the set of clusters of each of the vertices in N. Note that it is possible for a network to have multiple vertices with the same cluster. For our discussions, we do not allow $\mathrm{Cl}(N)$ to be a multiset, and thus include each cluster only once, even if it appears as a cluster for several vertices.

We will frequently compare two networks, and often ask if they are "the same" network as each other. We define this through an *isomorphism* definition. Given two networks N and M on the same leaf label set X, we say that N is isomorphic to M ($N \cong M$) if there exists a bijection ϕ between the set of vertices of the networks $V(N)$ and $V(M)$ such that the edge $(u, v) \in E(N)$ exists if and only if the edge $(\phi(u), \phi(v))$ exists in $E(M)$, $\phi(r_N) = r_M$, and $\phi(x) = x$ for all $x \in X$, where r_N and r_M are the roots of N and M, respectively.

1.3 Types of networks

We now give definitions and examples of some of the types of networks currently being studied. Particular attention will be paid to trees and normal networks, since the results in this paper focus on those classes of networks.

1.3.1 Trees

Since we are studying networks as a generalization of trees, it makes sense that we can define trees as a special case of a network. A (phylogenetic) tree T is simply a (phylogenetic) network where all vertices are normal vertices. Figure 1.1 is an example of a tree.

1.3.2 Regular networks

The *cover digraph* of a network R is a special construction based on the set of clusters of R. Given the set $\mathrm{Cl}(R)$ of all clusters of R (recall that $\mathrm{Cl}(R)$ is not a multiset), construct a new network R' by doing the following:

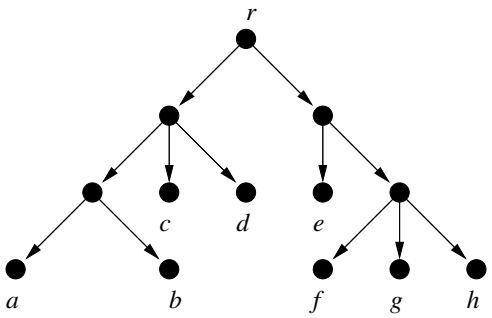


Figure 1.1 A tree.

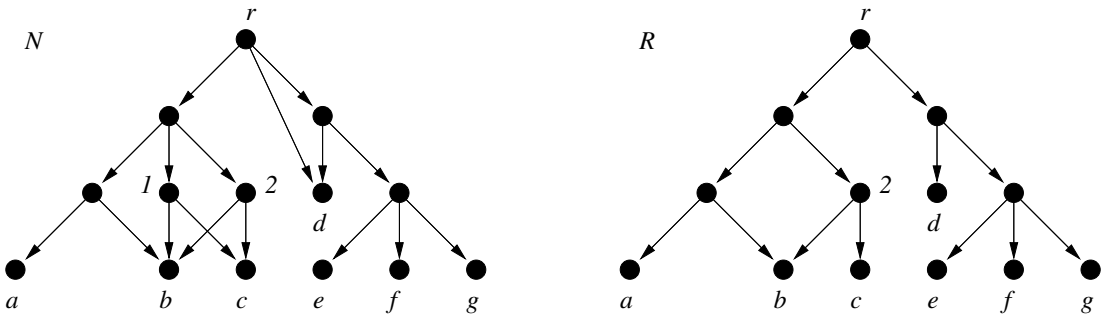


Figure 1.2 A network N and its cover digraph R.

1. For each cluster in $\mathrm{Cl}(R)$, create a vertex in $V(R')$.

2. For each pair of vertices u and v in R, if $\mathrm{cl}_R(u) \subset \mathrm{cl}_R(v)$ and the containment is maximal, then include (v, u) in $E(R')$.

This new network creates a "regularization" of the original network R. That is, R' contains the same clusters as R, but eliminates repeated clusters. Note that many networks have the same cover digraph. We call a network *regular* if it is isomorphic to its cover digraph. In other words, regularizing a regular network does not change the network. Figure 1.2 includes a network N that is not regular, since the vertices labeled 1 and 2 have the same cluster $\{1, 2\}$ and the edge (r, d) is redundant. The cover digraph R of N is a regular network, and is the regularization of N. More information can be found in (1).

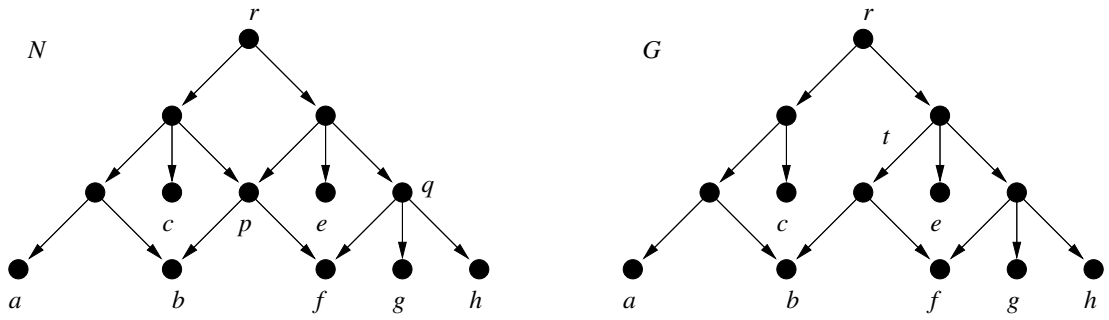


Figure 1.3 A network that is not galled (left) and a network that is galled (right).

1.3.3 Galled networks

Consider a hybrid vertex h and an ancestor a of h. Assume that a is not a parent of h, as well. Assume that there are two (a, h)-paths that share no common vertices, other than a and h. One path will pass through one parent p of h, and the other path will pass through a distinct parent q of h. These paths, when considered in the undirected sense, form a cycle passing through a, p, h and q. Such a cycle is called a *reticulation cycle*. If, for each choice of a and h, the edges in all reticulations cycles are normal, with the exception of the parent edges of h, then the network is called a *galled* network. In essence, galled networks only allow localized hybridization events. In figure 1.3, the network N on the left is not galled, since there is a reticulation cycle beginning at the root r with paths passing through p and q that are vertex disjoint, except at r and f, that contain at least one other hybrid vertex p. The network in 1.3 on the right is galled, since the only two reticulation cycles do not contain more than one hybrid vertex in them. These networks are discussed more in (5) and (9).

1.3.4 Level-k networks

A subnetwork M of a network N is a network such that $V(M) \subseteq V(N)$ and $E(M) \subseteq E(N)$. An *induced* subnetwork is a subnetwork such that $V(M) \subseteq V(N)$ and, for each edge $(u, v) \in E(N)$ with $u, v \in V(M)$, (u, v) is also in $E(M)$. A network is *biconnected* if it consists of one vertex, two vertices with a single edge connecting them, or more than two vertices such that, for any pair of vertices $u < v$, there are at least two vertex-disjoint (with the exception of u and v) (u, v)-paths. A *biconnected component* of a network N is a maximal induced subnetwork M,

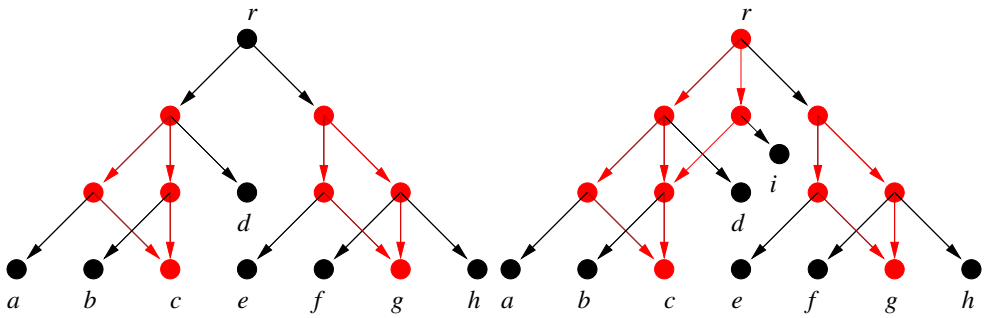


Figure 1.4 A level-1 network (left) and a level-2 network (right).

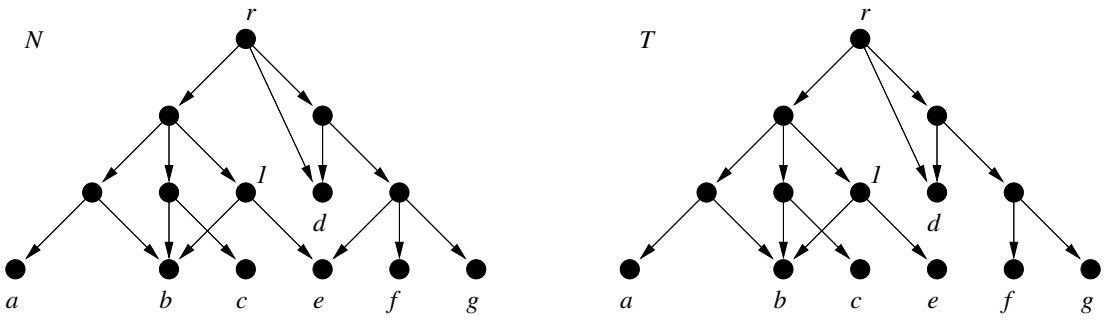


Figure 1.5 A network N that is not tree-child (left), and a tree-child network T (right).

obtained from N by removing a set of edges, in which M is biconnected. If, in each biconnected component, the number of hybrid edges is at most $k+1$, the network is called a *level-k* network. An example of level-1 (left) and level-2 (right) networks are given in figure 1.4 with the edges and vertices in the two non-trivial connected components colored red. Level-k networks are discussed in (7).

1.3.5 Tree-child networks

A network (N, X) is called *tree-child* if every internal vertex $v \in V$ has at least one normal child. Tree-child networks are networks that have a certain tree-like structure (3) in that each vertex has a child that appears as it would in a tree. The network N on the left in figure 1.5 is not tree-child since there is a vertex 1 that has no normal children. The network T on the right in figure 1.5 is tree-child since each internal vertex has at least one normal child.

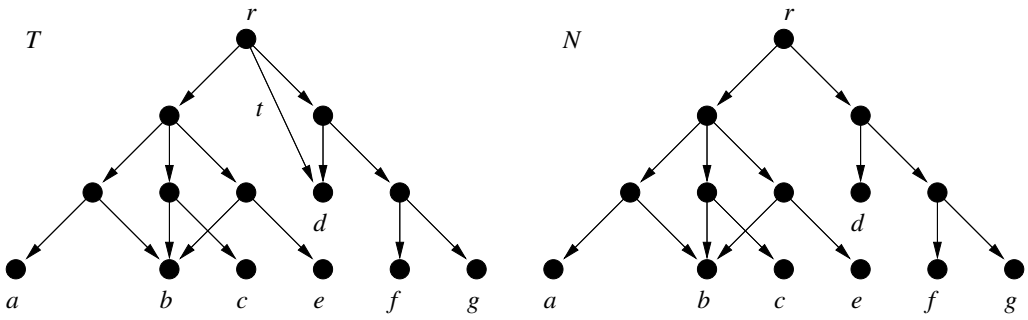


Figure 1.6 A network T that is not normal (left), and a normal network N (right).

1.3.6 Normal networks

Finally, a network is *normal* if it is tree-child, has no redundant edges, and no vertices with out-degree 1. These networks are the focus of our results. They are a generalization of trees, and offer a nice increase in complexity, while still maintaining some of the structure inherent in trees. The results in this paper should make this more apparent. The network T on the left in figure 1.6 is tree-child, but not normal since it contains a redundant edge t. The network N on the right in figure 1.6 is a normal network. Normal networks are discussed in detail in (18), (19), (20), and (21).

CHAPTER 2. COUNTING

We begin with some general results about normal networks. Since normal networks are relatively new, some basic results, such as bounds on the number of vertices and edges in normal networks, will help with future work on this special class of networks. We begin with a couple of lemmas that will be used frequently throughout the paper. In the figures given in this chapter, solid lines represent edges and dashed lines represent paths of unspecified length in order to encompass more cases.

Lemma 1. *In a normal network, the path connecting an internal vertex to a NPLD is unique.*

Proof. Let v be an internal vertex in a normal network N. Let x be a NPLD of v. This means that there must exist at least one path from v to x. Furthermore, this path must be normal. Suppose that there are two distinct paths P_1 and P_2 connecting v to x. Let P_1 be the normal path guaranteed to exist by definition of NPLD. Let $v = v_1, v_2, ..., v_s = x$ be the vertices in P_1 such that $(v_i, v_{i+1}) \in E(N)$ for $1 \leq i \leq s-1$. If $V(P_1) \subset V(P_2)$, then an edge in P_1 will be redundant. Let i be the smallest integer such that v_i is not in P_2. This must exist, since the paths are distinct and $V(P_1) \not\subset V(P_2)$. Let $j > i$ be the smallest integer greater than i such that v_j is in P_2. This must exist, since the paths share the vertex x. Clearly, $v_i \neq v_j$. We have that v_{j-1} is in P_1 and not P_2. Thus, v_j must have another parent, p, in P_2 and not in P_1. See the left half of figure 2.1 for a sketch of this. We get that v_j is actually a hybrid vertex. This contradicts the fact that P_1 is a normal path. Thus, there can be only one path connecting v to x. □

Lemma 2. *If two internal vertices in a normal network have a common NPLD, then they are related.*

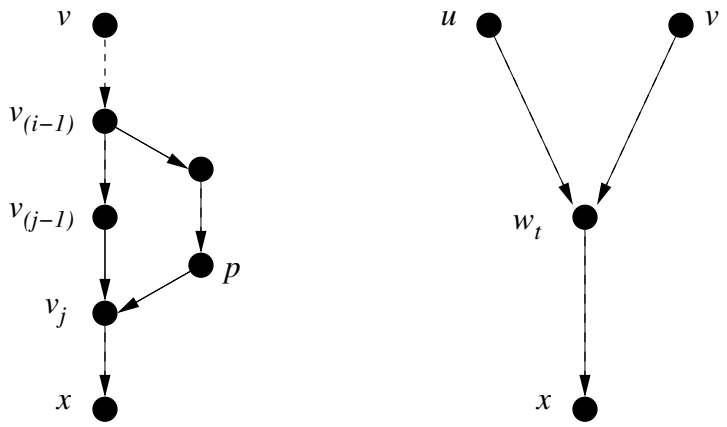


Figure 2.1 Sketches of what happens when there are two paths from a vertex to a leaf (left) and when two internal vertices share a NPLD (right).

Proof. Let u and v be two internal vertices in a normal network N, and let x be the common NPLD. By definition, there exists a (u, x)-path P_u and a (v, x)-path P_v, both of which are normal. By lemma 1, these are the unique normal paths connecting u and v to x. We will obtain a contradiction. Suppose that u and v are unrelated. This means that neither $u < v$ nor $v < u$. The two paths in question share at least one vertex, since they both end at x. This means that $V(P_u) \cap V(P_v)$ is non-empty. Let $u = w_1, w_2, ..., w_s = x$ be the vertices in P_u where $(w_i, w_{i+1}) \in E(N)$. Let t be the smallest integer such that $w_t \in V(P_v)$. This is the first vertex, beginning from u, where the paths intersect. If $t = 1$, then P_v contains u. This means $v < u$, which is a contradiction. Suppose $t > 1$. Since w_t is the first place that the paths meet, we know that w_{t-1} is not in P_v. Thus, w_t must have been hybrid, with parents w_{t-1} and another vertex from P_v not in P_u. See the right half of figure 2.1 for a sketch of this. We have a contradiction to the assumption that P_u and P_v were normal. Since these are the unique normal paths connecting u and v to x, we must have that x is not a NPLD of either vertex, since no other normal paths can exist that connect u or v to x. By contradiction, we must have that u and v are related. That is, either $u < v$ or $v < u$. □

The first basic proposition gives a bound on the number of edges that can go into or out of a vertex in a normal network.

Proposition 1. *The in-degree and out-degree of any vertex in a normal network cannot exceed*

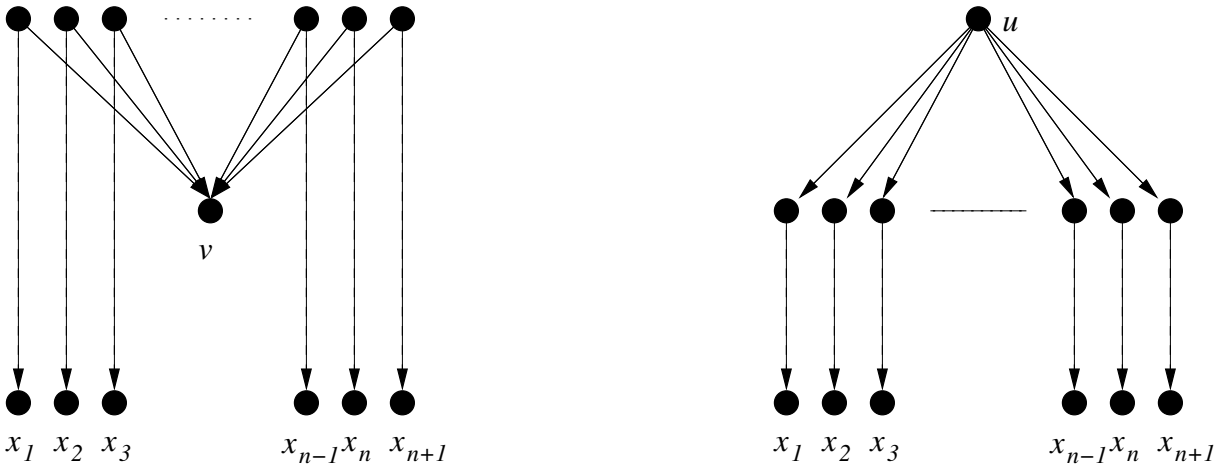


Figure 2.2 Sketches of the cases when vertex has out-degree greater than n (left) and in-degree greater than n (right).

the number of leaves.

Proof. Let n be the number of leaves in N. We start by proving that the in-degree of a vertex cannot exceed n. Let v be a vertex of in-degree at least $n+1$. Then, each of the parents, which cannot be related to any other parent of v due to the fact that there are no redundant edges, must have a NPLD. Each of these NPLDs must be on a path from its given parent of v that does not pass through v. By lemma 2, each pair of parents cannot share a common NPLD. Thus, there must be $n+1$ distinct leaves, which contradicts the fact that there are only n leaves. Thus, the in-degree of a vertex is, at most, n.

We now show that the out-degree of a vertex cannot exceed n. Let u be a vertex of out-degree at least $n+1$. No pair of the $n+1$ children can be related, since there are no redundant edges. Each of these $n+1$ children of u must have a NPLD. By lemma 2, none of these children can share a NPLD. Thus, there must be $n+1$ distinct leaves, which contradicts the fact that the network has only n leaves.

Figure 2.2 contains sketches of both cases.

□

This shows that any vertex in a normal network can have no more than $2n$ edges incident to it. In other words, a single vertex cannot have a very large neighborhood. We will be more

precise and get better bounds below.

The following lemma will be used several times in later results to help with counting vertices and edges.

Lemma 3. *In a normal network, no two hybrid vertices can share a NPLD. That is, a NPLD of a hybrid vertex cannot be a NPLD of any other hybrid vertex.*

Proof. Let h_1, h_2 be distinct hybrid vertices in a normal network. Suppose that they share a NPLD a. Lemma 2 tells us that h_1 and h_2 must be related. Without loss of generality, let $h_1 < h_2$. That is, let a path exist from h_1 to h_2.

By lemma 1, there is exactly one (h_1, a)-path and exactly one (h_2, a)-path in the network. If h_2 is on the (h_1, a)-path, then h_2 being hybrid contradicts the assumption that a is a NPLD of h_1. If h_2 is not on the (h_1, a)-path, then the (h_1, a)-path and (h_2, a) path must meet at some hybrid vertex h. This also contradicts the fact that the (h_1, a)-path is normal. Since these are the only options for h_2, and both end in contradictions, we can conclude that the assumption that h_1 and h_2 share a NPLD was false.

Thus, two hybrid vertices, whether related or not, cannot share a NPLD. That is, a NPLD of h_1 cannot be a NPLD of h_2, and vice versa. □

The following corollary to this lemma gives a nice result on the number of hybrids that can exist in a normal network. We immediately improve it slightly in the following proposition.

Corollary 1. *If a normal network has n leaves, then the number of hybrid vertices cannot exceed n.*

Proof. By the preceding lemma, each hybrid vertex has a NPLD that it does not share with any of the other hybrid vertices. Thus, if there were to exist more than n hybrid vertices, we would have more than n leaves, which is a contradiction. □

Proposition 2. *If a normal network has n leaves, then the number of hybrid vertices is at most $n - k$, where k is the out-degree of the root.*

Proof. Let the number of hybrid vertices be h. By lemma 3, we know that there must exist a NPLD for each hybrid vertex in the network that is not shared with any other hybrid vertex.

Consider, as well, the children of the root r_n. Label them $c_1, ..., c_k$. Note that none of these children can be hybrid and that none of them are related to each other (i.e. $c_i \not\leq c_j$ for any $1 \leq i, j \leq k$) because there are no redundant edges. This means that each of these k children must have a NPLD that is not shared with any of the other children. These cannot be shared with any of the hybrid vertices, either, for similar reasons. Thus, we have at least $h+k$ distinct normal leaves. And, since the number of leaves is fixed at n, we need $h + k \leq n$. In other words, $h \leq n - k$, as desired.

Note that, since we require internal vertices in a normal network to have at least two children, we can say that $k \geq 2$ for all normal networks, and the number of hybrid vertices in a normal network is at most $n - 2$, regardless of the root's out-degree.

□

We already have a nice bound on the number of hybrid vertices that a normal network can have. Next, we give a tight bound on the number of vertices that a normal network can have. This proposition is given and proved in (18), but we show it here for completeness' sake.

Proposition 3. *If a normal network has n leaves, then $|V| \leq (n^2 + n)/2$.*

Proof. Consider an internal vertex u. It must have a NPLD x, by normality, passing through one child c. It must also have at least one other child v. This child must also have a NPLD y that is different from x. In (18), it is shown that u is the unique most recent common ancestor for x and y. Since there are n leaves, there can be only $\binom{n}{2}$ pairs of leaves. Thus, there can be only $\binom{n}{2}$ internal vertices. Thus, including the leaves, the maximum number of vertices is $\binom{n}{2} + n = (n^2 + n)/2$.

This bound is shown to be tight in a construction given below in the proof of theorem 1. □

The following proposition gives an upper bound on the in-degree of a hybrid vertex. This will be used later in the section when we count the number of edges in a normal network.

Lemma 4. *If a normal network has n leaves, then a hybrid vertex in that network can have in-degree $n - 1$ at most.*

Proof. Suppose a hybrid vertex has in-degree n. Since no two parents can be related, due to the fact that we cannot have redundant edges, each parent must have a NPLD that cannot be a NPLD of any other parent of the hybrid vertex. That forces the existence of n leaves. Note, also, that the hybrid vertex itself must have a NPLD. If the hybrid vertex happens to be a leaf, then we have $n+1$ leaves, which is a contradiction. If the hybrid vertex is not a leaf, then it still needs to have a NPLD that is different from the NPLDs of its parents, which still forces the number of leaves to be $n+1$, which is a contradiction. Thus, a hybrid vertex cannot have in-degree greater than $n-1$. □

The following proposition considers what happens when a hybrid vertex has extremely large in-degree. It shows that there are limitations to normal networks, since we should not need to assume that a species with a very high number of hybrid parents in extant. However, this is a very specific case that probably will not arise often.

Proposition 4. *If a hybrid vertex in a normal network with n leaves has $n-1$ parents, then it must be a leaf.*

Proof. Suppose that a hybrid vertex h has $n-1$ parents. Suppose that h is not a leaf for a contradiction. Since we are in a normal network, h must have at least two children. The parents of h cannot be related to avoid redundant edges. The children of h also cannot be related to avoid redundant edges. Lemma 1 gives us that each parent of h must have its own NPLD that is not shared by any of the other parents. Similarly, each child of h must have its own NPLD that is not shared by any of the other children. Suppose that a parent p and a child c of h share a NPLD x. Lemma 2 says that p and c must be related. They certainly are. However, the path from p to x must pass through h, which is hybrid, contradicting the assumption that the path connecting p to x was normal. Thus, p and c cannot share a NPLD. We have a total of $(n-1)+2 = n+1$ distinct NPLD. This contradicts the assumption that there were only n leaves. Thus, h must have had fewer than two children. This forces h to be a leaf, as desired. □

The next lemma shows that we cannot have too many vertices of very high in-degree. This is a nice result on its own, but will be used in proof of theorem 1, as well.

Lemma 5. *Consider a normal network with n leaves. There can be, at most, k vertices of in-degree $n-k$ or greater for $1 \leq k \leq n-2$.*

Proof. Suppose that there are $k+1$ vertices of in-degree $n-k$. Let $H = \{h_1, h_2, ..., h_{k+1}\}$ be these hybrids. None of the elements of H can share a NPLD by lemma 3. Thus, there must be at least $k+1$ distinct leaves as NPLDs of vertices in H. Since networks are acyclic, there must be a vertex in H, say h_i, that is not a descendant of the other k hybrids in H. This vertex has $n-k$ parents, by assumption. None of these parents can share a NPLD with any of the hybrids in H. This is due to the fact that H contains hybrids, and a normal leaf being shared with one of these hybrids would imply that the leaf is not actually a NPLD, which would be a contradiction. None of the $n-k$ parents of h_i can share a NPLD with a different parent of h_i, either, since that would contradict the normality of those descendants. Thus, there must be $n-k$ distinct leaves from the parents of h_i. Since these are all distinct, we must actually have $(n-k)+(k+1) = n+1$ distinct leaves, which is a contradiction to the fact that the network only has n leaves. Thus, there can be, at most, k vertices of in-degree $n-k$ or greater for any $1 \leq k \leq n-2$. □

Proposition 5. *The maximum total in-degree for all hybrid vertices in a normal network on n leaves is $(n-1)(n)/2 - 1 = (n^2-n-2)/2$.*

Proof. We know that we can have, at most, $n-2$ hybrid vertices by proposition 2. We also know that each hybrid vertex has in-degree at most $n-1$ by lemma 4. As a crude upper bound, we know that the maximum in-degree from the hybrid vertices is $(n-2)(n-1) = n^2-3n+2$. This assumes that each of the hybrid vertices has maximum in-degree. Let $H = \{h_1, h_2, ..., h_{n-2}\}$ be the hybrid vertices. Suppose $d_i(h_j) \leq n-1$ for each $1 \leq j \leq n-2$. We now prove, by induction on k, that, for each k, the total in-degree of the hybrid vertices is $\sum_{i=1}^{k}(n-i)+(n-k-1)(n-2-k)$.

Let $k = 1$ for our base case. According to lemma 5, we can have at most $k = 1$ vertex of in-degree $n-k = n-1$. Suppose h_1 is this hybrid vertex. Then, each of the $n-2$ other hybrid vertices can have in-degree at most $n-2$. We now have that the total in-degree of the hybrid vertices is at most $(n-1)+(n-2)(n-2-1) = \sum_{i=1}^{k=1}(n-i)+(n-k-1)(n-2-k)$ for $k=1$, which is what we wanted.

For the inductive step, we suppose that the equation above holds for arbitrary $1 \leq k \leq n-3$. We wish to show that the equation is true for $k+1$. We can pick the hybrid labels so that $d_i(h_j) = n - j$ for $1 \leq j \leq k$ since, if they are not in the desired order, we can simply relabel them to meet our needs without changing the proof. We know, by assumption, that the total in-degree of the hybrid vertices is at most $\sum_{i=1}^{k}(n-i) + (n-k-1)(n-2-k)$. This bound assumes that the remaining $n-2-k$ hybrid vertices all have in-degree at most $n-k-1$. According to lemma 5, we can have only $k+1$ vertices of in-degree $n-k-1$. We know that $h_1, h_2, ..., h_k$ all have in-degree greater than $n-k-1$. Thus, of the remaining $n-2-k$ vertices, only one can have in-degree $n-k-1$. Let h_{k+1} have in-degree $n-k-1$. We now assume that the remaining $n-2-k-1$ hybrid vertices have in-degree $n-k-1-1$. Now, the total in-degree of all hybrid vertices is at most $\sum_{i=1}^{k+1}(n-i) + (n-2-(k+1))(n-(k+1)-1)$. This is the total that we wanted.

We continue this process until we get to $k = n-3$. Apply the induction one more time. We know that h_{n-2} has in-degree $n-(n-2) = 2$. This accounts for all of the vertices in H. We have that the maximum total in-degree for all hybrid vertices is $\sum_{i=1}^{k+1}(n-i) + (n-(k+1)-1)(n-2-(k+1))$. For $k = n-3$, we get that this maximum is actually $\sum_{i=1}^{n-2}(n-i) + (n-(n-2)-1)(n-2-(n-2)) = \sum_{i=1}^{n-1}(i) - 1 = (n-1)(n)/2 - 1 = (n^2-n-2)/2$, as desired. □

We finally have enough to prove the main theorem of the section, which tightly bounds the number of edges that a normal network can have.

Theorem 1. *If a normal network has n leaves, then $|E| \leq n^2 - n$.*

Proof. We know that the maximum number of vertices is $(n^2+n)/2$ by proposition 3. The maximum number of hybrids is $n-2$ by proposition 2. Then, each of the normal vertices has in-degree one, the root has in-degree 0, and the total in-degree of all of the hybrid vertices, as shown above in proposition 5, is, at most $(n^2-n-2)/2$. The total in-degree can be at most the sum of the in-degrees of all of the vertices. We get that this maximum is $[((n^2+n)/2 - (n-2) - 1)] + [(1)(0)] + [(n^2-n-2)/2] = n^2 - n$, as desired.

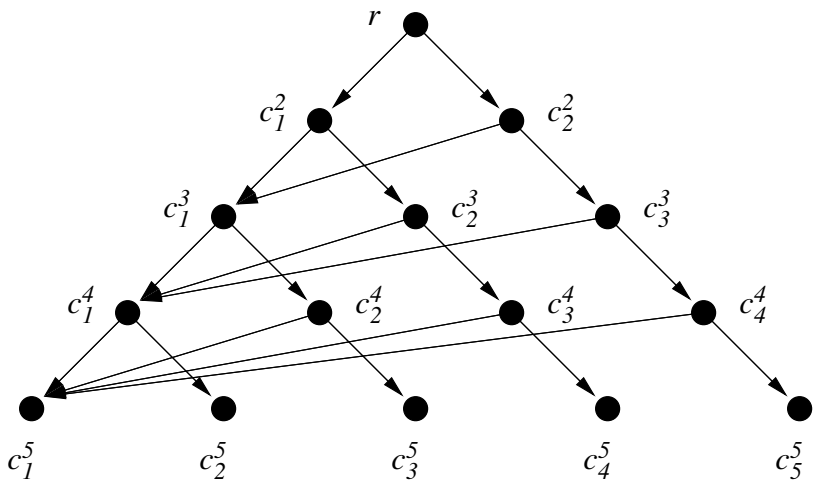


Figure 2.3 A normal network with 5 leaves with the maximum possible number of vertices and edges.

To see that this bound, along with the bound given in proposition 3 that $|V| \leq (n^2+n)/2$, are tight, we construct a normal network that attains both of these values. Begin with the root r. Let c_1^2, c_2^2 be the children of r. For $3 \leq i \leq n$, create a set of i vertices $c_1^i, c_2^i, ..., c_i^i$. Let c_1^i be a child of each of c_j^{i-1} for $1 \leq j \leq i-1$. Let c_k^i be a child of c_{k-1}^{i-1} for $2 \leq k \leq i$. There will be $\sum_{i=1}^{n} i = (n^2+2)/2$ vertices, as desired. There will also be $\sum_{i=1}^{n-1} 2i = n^2 - n$ edges, as desired. An example of such a network when $n = 5$ is given in figure 2.3. □

Next, we gives bounds on the number of edges and vertices for specific cases of networks. First we give tight bounds on the number of vertices and edges of a *binary* normal network (i.e. the in- and out-degree of any vertex is at most 2), then do the same for normal networks where the out-degree is bounded by 2, and the in-degree is bounded by some constant t. The binary normal network counts will be used in chapter four in the discussion of network space.

Proposition 6. *If a normal network has n leaves and is binary, then $2n-1 \leq |V| \leq 3n-3$ and $2n-2 \leq |E| \leq 4n-6$.*

Proof. We will count the number of edges twice. We begin by counting the edges that are going into vertices of the network. Each vertex has in-degree at most 2. The root has in-degree 0. There can be, by proposition 2, at most $n-2$ hybrid vertices. The rest of the vertices will have in-degree exactly 1. Let us assume that we have $n-k$ hybrid vertices, where $2 \leq k \leq n$.

Thus, we have that $|E| = [0 \cdot 1] + [1 \cdot (|V| - (n-k) - 1)] + [2 \cdot (n-k)]$. This gives us that $|E| = |V| + n - k - 1$. Using our bounds for k, we get that $|V| - 1 \leq |E| \leq |V| + n - 3$.

We now count the edges going out of vertices in the network. Each vertex, except for the n leaves, must have out-degree at least 2 by definition of the network. Since the network is binary, we have that each vertex must have out-degree at most 2. Thus, each vertex, except for the leaves, has out-degree exactly 2. The leaves have out-degree 0. Thus, we have that $|E| = 2(|V| - n) = 2|V| - 2n$.

Since the total out-degree must equal the total in-degree, the out-degree count of $|E| = 2|V| - 2n$ must fall in the interval found for the in-degree count. That is, $|V| - 1 \leq 2|V| - 2n \leq |V| + n - 3$. Simplifying this, we get $2n - 1 \leq |V| \leq 3n - 3$. And, since $|E| = 2|V| - 2n$, we can use this inequality to get that $2n - 2 \leq |E| \leq 4n - 6$, as desired.

We now show that these bounds are tight. To see that the lower bounds are tight, simply observe that the counts of $|V| = 2n - 1$ and $|E| = 2n - 2$ are those of a rooted tree on n leaves. That is, the network has no hybrid vertices. These are well-known results.

To see that the upper bounds are tight, we construct a normal network that attains both of these values. Begin with the root r. Let c_1^2, c_2^2 be the children of r. For $3 \leq i \leq n$, create a set of 3 vertices c_1^i, c_2^i, c_3^i. Let c_1^i be a child of both c_1^{i-1} and c_2^{i-1}. Let c_3^i be a child of c_2^{i-1}. There will be $1 + 2 + 3(n-2) = 3n - 3$ vertices, as desired. There will be $2 + 4(n-2) = 4n - 6$ edges, as desired. An example of such a network when $n = 5$ is given in figure 2.4. □

Proposition 7. *If a normal network has n leaves, the maximum out-degree is 2, and the maximum in-degree is t where $1 \leq t \leq n - 1$, then $2n - 1 \leq |V| \leq n + nt - t^2/2 - t/2$ and $2n - 2 \leq |E| \leq (2n - t - 1)(t)$. Furthermore, these bounds are tight.*

Proof. We will count the number of edges twice. We begin by counting the edges that are going into vertices of the network. Each vertex has in-degree at most t. The root has in-degree 0. There can be at most $n - 2$ hybrid vertices by proposition 2. The rest of the vertices will have in-degree exactly 1. Let us assume that we have $n - 2$ hybrid vertices. By lemma 5, we can have no more than k vertices of in-degree $n - k$. Applying this, as we did in the proof above, we get that the maximum number of edges going into hybrid vertices cannot exceed the case

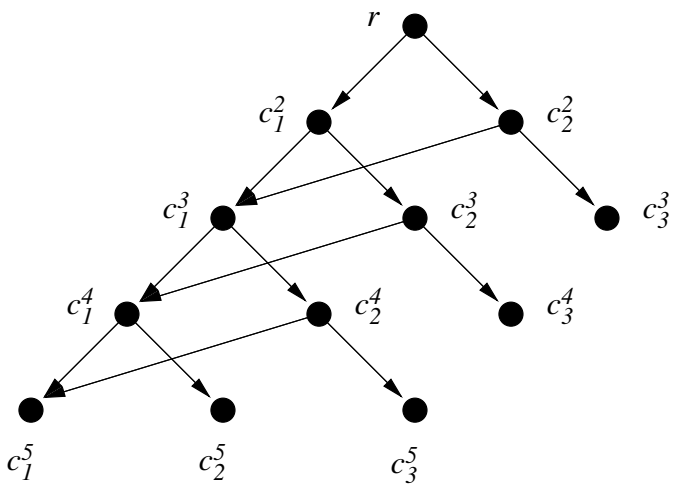


Figure 2.4 A binary normal network with 5 leaves with the maximum possible number of vertices and edges.

when there is one vertex of in-degree k for $2 \leq k \leq t-1$ and the rest have in-degree exactly t. Thus, we have that $|E| = [0 \cdot 1] + [1 \cdot (|V| - (n-2) - 1)] + \sum_{i=2}^{t-1}(i) + (t)(n-t)$. Certainly, $|V| - 1 \leq |E|$, which is the case when our network is a star tree. Combining these inequalities, we have that $|V| - 1 \leq |E| \leq |V| + n(t-1) - t^2/2 - t/2$.

We now count the edges going out of vertices in the network. Each vertex, except for the n leaves, must have out-degree at least 2 by definition of the network. We assume that the maximum out-degree is 2. Thus, each vertex, except for the leaves, has out-degree exactly 2. The leaves have out-degree 0. Thus, we have that $|E| = 2(|V| - n) = 2|V| - 2n$. We can insert this into the inequality found in the paragraph above and simplify to get that $2n - 1 \leq |V| \leq n + nt - t^2/2 - t/2$, as desired. Also, since $|E| = 2|V| - 2n$, we can use this bound on the number of vertices to get that $2n - 2 \leq |E| \leq (2n - t - 1)t$, as desired.

We now show that these bounds are tight. To see that the lower bounds are tight, simply observe that the counts of $|V| = 2n-1$ and $|E| = 2n-2$ are those of a rooted tree on n leaves. That is, the network has no hybrid vertices. These are well-known results.

To see that these upper bounds are tight, we construct a normal network that attains both of these values. Begin with the root r. Let c_1^2, c_2^2 be the children of r. For $3 \leq i \leq t$, create a set of i vertices $c_1^i, c_2^i, ..., c_i^i$. Let c_1^i be a child of each of c_j^{i-1} for $1 \leq j \leq i-1$. Let c_k^i be a child of c_{k-1}^{i-1} for $2 \leq k \leq i$. For $t+1 \leq i \leq n$, create a set of $t+1$ vertices $c_1^i, c_2^i, ..., c_{t+1}^i$.

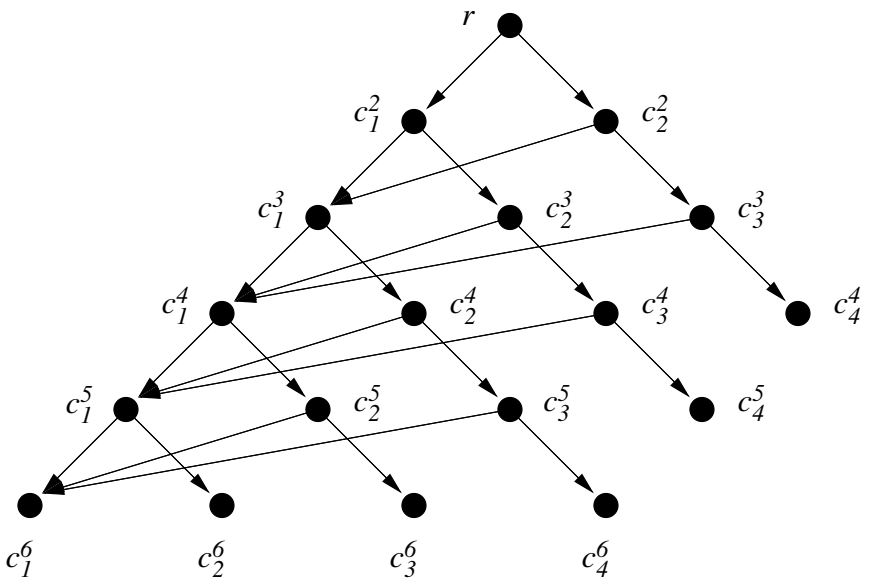


Figure 2.5 A normal network whose vertices have in-degree at most t and out-degree at most 2 with the maximum possible number of vertices and edges where $t = 3$ and $n = 6$.

Let c_1^i be a child of each of c_j^{i-1} for $1 \leq j \leq i-2$. Let c_{t+1} be a child of c_t^{i-1}. There will be $\sum_{i=1}^{t} i + (n-t) * (t+1) = n + nt - (t^2+t)/2$ vertices, as desired. There will also be $\sum_{i=1}^{t-1} 2i + (n-t)(2t) = (2n-t-1) * t$ edges, as desired. An example of such a network with $t = 3$ and $n = 6$ is given in figure 2.5.

□

Finally, we give a bound on the number of normal edges that a normal network can have. Certainly, we know that it cannot exceed the total number of edges. However, we would like a better bound specifically on how many normal edges a normal network can have.

Proposition 8. *A normal network with n leaves has no more than $\binom{n}{2} + 1$ normal edges.*

Proof. A normal network can be constructed from four basic operations on a tree that is displayed by the network. They are listed below. We can only apply these so many times before we exceed the number of allowable vertices, edges, hybrid vertices, or in-degree on hybrid vertices. Assume that the operations will only be applied in a way that will not violate any parts of the definition of a normal network. Since we can actually construct a network with the bounds that are discussed below, we need only show that these are plausible operations on

Op.	Vert.	HE	NE	NV to HV	NE to HE	Net HE	Net HV	Net NE	Net NV
1	1	1	1	1	1	+2	+1	0	0
2	0	1	0	1	1	+2	+1	-1	-1
3	1	1	1	0	0	+1	0	+1	+1
4	0	1	0	0	0	+1	0	0	0

Table 2.1 The operations and their contributions to a network. The middle section gives the number of vertices, hybrid edges, and normal edges added, how many normal vertices become hybrid, and how many normal edges become hybrid. The right section show the net change in the number of hybrid edges, hybrid vertices, normal edges, and normal vertices.

a normal network.

Operation 1: Let $e = (a, b)$ be an edge. Remove the edge. Add a vertex c. Add edges $(a, c), (c, b)$ and (c, x), where x is a normal vertex. The edge e can be normal or hybrid.

Operation 2: Let u and v be vertices in the network, with u not a leaf, and v normal. Add edge (u, v).

Operation 3: Let $e = (a, b)$ be an edge. Remove the edge. Add a vertex c. Add edges $(a, c), (c, b)$ and (c, h), where h is a hybrid vertex. The edge e can be normal or hybrid.

Operation 4: Let u and v be vertices in the network, with u not a leaf, and v hybrid. Add edge (u, v).

We begin with a tree. To maximize the number of edges, we use a binary tree. This begins with $2n - 1$ vertices and $2n - 2$ edges. We will try to add normal edges. Note that the only operation that yields a positive net increase in the number of normal edges is operation 3. Thus, we want to maximize the number of times we use that operation. Operation 4 adds a hybrid edge, but does not add normal edges. The only way this could help us is by adding an edge that could possibly be split to add a normal edge. We will see, however, that we need not use this operation because there will be a sufficient number of edges produced by the other operations. Operations 1 and 2 introduce new hybrid vertices. We actually gain normal edges by using operation 3. However, we must have hybrid vertices in order to use 3. The difference between Operations 1 and 2 is the fact that we add a normal edge and normal vertex in 1, and not in 2. We would only use operation 1 if we had no edges left to split to use operation 1. Again, it will be shown that we need not use operation 2 because edges will always exist for us

to use operation 1.

To begin, we must use either operation 1 or 2 in order to get a hybrid vertex. Operation 1 is the best choice, however, since we lose normal edges with operation 2. We then apply operation 3 enough times to maximize the in-degree of the hybrid vertex. The first time, the maximum degree is $(n-1)$. Repeat this process. Lemma 5 above tells us that we can have only one vertex of degree $(n-1)$. Thus, this new hybrid vertex will have maximum degree $n-2$. Repeat the process, each time adding a new hybrid vertex with operation A, and applying operation 3 enough times to maximize the in-degree of the new hybrid vertex, while not violating lemma 5. We do this until there are $(n-2)$ hybrid vertices, since more than that would exceed the limit found in proposition 2.

We prove more rigorously that operation 3 is more desirable than operation 4. In fact, we can avoid using operation 4 entirely and still be guaranteed the maximum number of normal edges. Suppose that we have a hybrid vertex h and that we have decided to use operation 4 to add an edge (v,h) from some other vertex v in N. There cannot be any parents of h that are related to v. This is due to the fact that, if a descendant d of v were a parent of h, then (v,h) would be redundant, and if some ancestor a of v were a parent of v, then (a,h) would be redundant. The same would be true after we added the edge (v,h). Now, consider a parent p of v, and what would happen if we were to apply operation 3 and split the edge (p,v) into two edges with a new vertex u. Before adding any edges to h, observe that the set of descendants of u is $\mathrm{des}(u) = \mathrm{des}(v) + v$ and that the set of ancestors of u is $\mathrm{anc}(u) \subseteq \mathrm{anc}(v)$. In fact, $\mathrm{des}(u) \cup \mathrm{anc}(u) \subseteq \mathrm{des}(v) \cup \mathrm{anc}(v)$. If we apply operation 4, then we cannot use any other vertex in $\mathrm{des}(v) \cup \mathrm{anc}(v)$ as a parent of h. If we apply operation 3, however, we cannot use any other vertex in $\mathrm{des}(u) \cup \mathrm{anc}(u)$, which is contained in $\mathrm{des}(v) \cup \mathrm{anc}(v)$. We also do not preclude the possibility of using v or any parent of v in operations for other hybrid vertices. Thus, using operation 3 not only immediately adds more normal edges than operation 4, it also allows for the possibility of adding more normal edges than operation 4 in the future.

Using table 2.1, we end up with a total count of $(n^2+n)/2$ vertices, n^2-n edges, $(n^2-n-2)/2$ hybrid edges, $(n^2-n+2)/2 = \binom{n}{2}+1$ normal edges, $(n-2)$ hybrid vertices, and $(n^2-n+4)/2$ normal vertices. These do not violate the lemmas above, and the totals add up to be the same

number as in the maximal network example given in the proof of theorem 1.

We are assuming that X is the set of leaf labels and that the root is not a leaf, and thus not in X. We also assume that no vertices have out-degree 1. Some people use a slightly different definition. If, instead, we include the root as a leaf, call the new label set X', and require that all hybrid vertices have out-degree 1, the count will change slightly. Let $m = |X| + 1 = |X'|$, where $n = |X|$. According to our previous count, we will have a bound of $\binom{n}{2}$. In the new definition, we add $n - 2$ (or $(m - 1)$) normal edges and normal vertices as children of hybrid vertices, include one more vertex as the new leaf, and include one more normal edge to connect the leaf to the network. This works out to be $\binom{m-1}{2} + 1 + ((m - 1) - 2) + 1 = \binom{m}{2}$. Thus, according to this new definition, the maximum number of normal edges in a normal network is $\binom{m}{2}$.

□

CHAPTER 3. TREES DISPLAYED BY NETWORKS

3.1 Introduction

Since trees are also networks, we consider the idea that information about a network can be obtained by studying the trees which a network *displays*, that is, trees that can be obtained from a network through a sequence of basic operations on the network. However, in general, the number of trees a network displays can be exponential with respect to the number of hybridization events. We seek to find a fast algorithm to determine whether a given tree is displayed by a network.

In (9), it is shown that the tree containment problem, which determines whether a given tree is displayed by a given network, is NP-complete. The next step should be to see if this is still true for certain classes of networks. Galled networks are very nice in that tree containment is determinable in polynomial time, as shown in (9). In fact, the cluster containment problem, determining whether a given network displays a tree which has a given cluster, is even polynomial for galled networks (6). (8) shows that the tree containment problem for a specific type of regular network is NP-complete. This implies that the tree containment problem for all regular networks is NP-complete, as well.

In this section, we consider the class of normal networks. Each vertex in a normal network has a NPLD. That is, from each vertex in a normal network, there is a path to a leaf that passes through only normal vertices. This path then resembles a path that one would see in a tree, and the leaf and the vertex in question will maintain this relationship, since we will only remove hybrid edges during reductions from networks to trees. Because of this structure, we can show that the normal network and any tree it does display share enough structure to make it relatively simple to determine whether or not that tree is displayed by that normal

network. We solve the tree containment problem for normal networks. That is, we show that, given a normal network and a tree, it can be determined in polynomial time whether or not the network displays the tree.

We will provide a pseudocode version of an algorithm that will determine whether a given normal network displays a given tree. The algorithm is relatively simple and uses a somewhat greedy approach. It begins with the root of the network, checking if the children in the tree are similar enough to the children in the network, and continues until it has exhaustively checked all vertices. Furthermore, we will show that this algorithm is polynomial in time, thus making it much more practical than brute-force, exhaustive searches.

3.2 Preliminaries

Suppose a vertex v has out-degree 1, parents $P = \{p_1, p_2, ..., p_k\}$ and child c. We *suppress* v by deleting edges (p_i, v) and vertex v, and adding the edges (p_i, c) for each i such that $1 \leq i \leq k$. A *parent map* is a choice of one parent edge for each of the hybrid vertices in a network N such that removing all other hybrid edges and suppressing out-degree 1 vertices results in a tree T. Such a tree is said to be *displayed* by N. Figure 3.1 gives an example of a network, along with a tree that is displayed by the network, and a tree that is not. When we find a parent map, we are choosing a single parent for each hybrid vertex, which clearly results in a tree, since each vertex other than the root now has exactly one parent. To show that the tree T_1 is displayed by N in figure 3.1, we pick parents b, d, and g for vertices c, g, and i, respectively. We will find connections between parent maps and the algorithm that we create to test if a tree is displayed by a network.

Let N be a normal network with leaf set X. Let T be a tree on the same leaf set X with root r_t. For convenience's sake, we do not allow vertices with out-degree 1 to be initially present. To show that the normal network N displays T, we need to show that we can remove hybrid edges and suppress out-degree 1 vertices in N until we are left with T. To do this, we will find an injective mapping ϕ from the vertices of the tree to the vertices of the network that satisfy certain conditions, which will be listed later. We will show that this mapping, if it exists, will imply that N displays T. It should be noted that, without the normality assumption, new

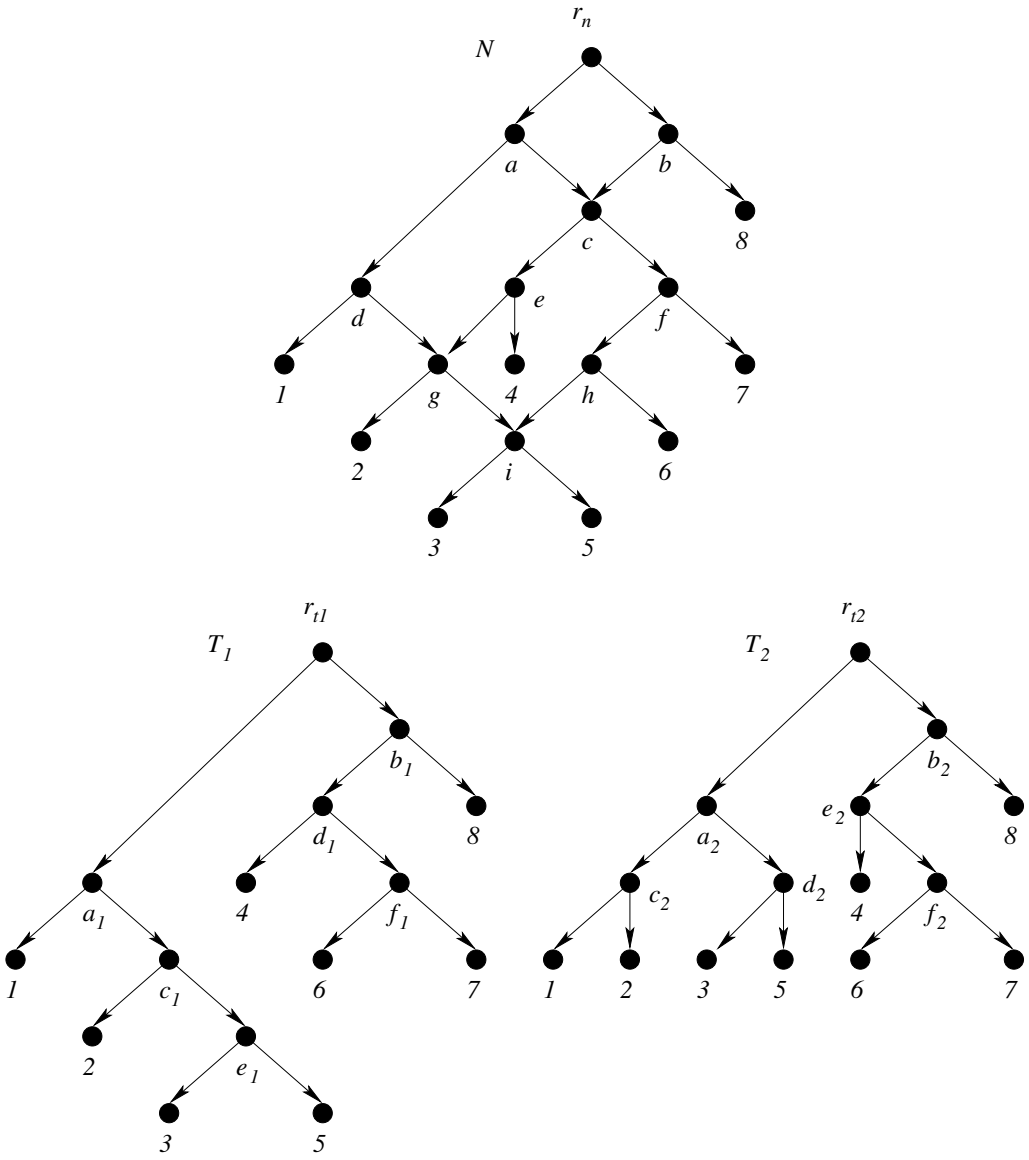


Figure 3.1 A normal network N and two trees T_1 and T_2. T_1 is displayed by N, while T_2 is not.

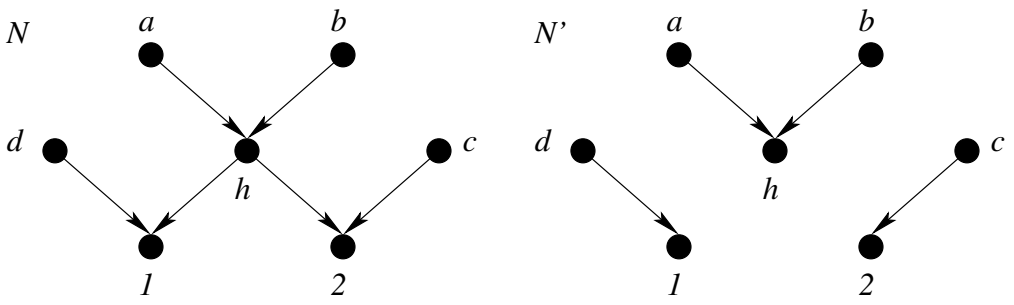


Figure 3.2 A sketch of what happens when we remove two hybrid edges from a network that is not normal, resulting in a new leaf h that does not have a label from the original label set of the network.

leaves might be created during this process. In figure 3.2, h becomes a leaf when we remove hybrid edges $(h, 1)$ and $(h, 2)$. This is undesirable, since we want the leaves to remain the same, regardless of any operations done to the network. If the network is normal, however, this cannot happen since each vertex has a normal path from itself to a leaf in X, and none of the normal edges will be removed. Thus, for a normal network, we can ignore this possibility.

Let N be a normal network with root r_n and leaf set X. For each vertex v, we make note of its set of NPLDs, $\mathbf{t}(v, N) = \{x \in X : \text{a normal path exists in } N \text{ from } v \text{ to } x\}$. Call this the *normal set* of v. If the choice of N is clear, we will use simply $\mathbf{t}(v)$. We can determine if a network displays a tree by exhaustively removing hybrid edges without disconnecting the network, then suppressing new leaves and out-degree 1 vertices until a tree is created. When we do this, since all the edges from a vertex to its normal leaves are normal edges, none of the edges removed will be on a path from a vertex to any of its NPLDs. Thus, in any tree displayed by N, the cluster of a vertex will contain the set $\mathbf{t}(v, N)$. These sets will be crucial in determining whether an arbitrary tree is displayed by the network.

Since N is normal, each vertex has at least one leaf such that a normal path exists from the vertex to the leaf. Therefore, the set $\mathbf{t}(v)$ is nonempty for all vertices v in the network, including the leaves themselves. Also, consider another distinct vertex u such that u and v are unrelated (i.e. neither $u < v$ nor $v < u$). Suppose $\mathbf{t}(u) \cap \mathbf{t}(v) \neq \emptyset$. Then there is a vertex w such that normal paths exist from u to w and from v to w. These paths must meet somewhere, since w exists only once. Thus, there must be a hybrid vertex somewhere in both paths, making

them not normal paths. This is a contradiction to the fact that w was a NPLD of u and v. Thus, $\mathbf{t}(u) \cap \mathbf{t}(v) = \emptyset$ for unrelated vertices. This does not mean, however, that related vertices will necessarily share NPLDs.

3.3 The main theorem

We begin with some basic properties that a mapping ϕ from $V(T)$ to $V(N)$ must have to show that N displays T. Clearly, the roots must correspond, and the leaves must correspond. That is, $\phi(r_t) = r_n$ and $\phi(x) = x$ for each $x \in X$. We also need the condition that, if an edge (u, v) exists in T, then a path exists from $\phi(u)$ to $\phi(v)$ in N. We do not require an edge between $\phi(u)$ and $\phi(v)$ because there might be some vertices that are suppressed in N when we remove edges to create a tree. We will keep track of edges from such paths in N that correspond to edges in T with a set of edges $E'(N)$ of N. Observe that, in a tree, all leaves that are descendants of a given vertex are NPLDs. That is, $\mathbf{t}(u, T)$ is just the cluster of the vertex $\mathrm{cl}(u, T)$. We will compare the clusters in T to the normal sets of vertices in N.

We must have that $\mathbf{t}(\phi(u), N) \subseteq \mathrm{cl}(u, T)$. If this were not the case, then, no matter how we had chosen the set of edges to remove in N to make a tree N', we would have $\mathrm{cl}(\phi(u), N') \neq \mathrm{cl}(u, T)$, which would give us that T is not displayed by N. We also will need that $\mathrm{cl}(u, T) \subseteq \mathrm{cl}(\phi(u), N)$. If this were not the case, then, after removing edges from N to form the tree N', there is no way that $\phi(u)$ would have all of $\mathrm{cl}(u, N)$ as descendants. Thus, the tree would not be displayed. We also need to make sure that the mapping ϕ is not changed during the reduction from a network to a tree after the mapping has been found. To avoid this, we need that no children c of $\phi(u)$ exist such that $\mathbf{t}(\phi(u)) = \mathbf{t}(c)$ and $\mathrm{cl}(u, T) \subseteq \mathrm{cl}(c, N)$. Combining all of this, we have the following requirements:

$$\text{For all } u \in V(T), \mathbf{t}(\phi(u), N) \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(\phi(u), N) \quad (3.1)$$

$$\text{If } (u, v) \in E(T), \text{ then there exists a } (\phi(u), \phi(v)) \text{ path in } N \text{ whose edges are in } E'(N) \quad (3.2)$$

$$\text{No children } c \text{ of } \phi(u) \in V(N) \text{ exist in } N \text{ such that } \mathbf{t}(\phi(u)) = \mathbf{t}(c) \text{ and } \mathrm{cl}(u, T) \subseteq \mathrm{cl}(c, N) \quad (3.3)$$

$$\phi \text{ is injective, } \phi(r_t) = r_n \text{ and for each } x \in X, \phi(x) = x \quad (3.4)$$

We will say that a vertex $v \in N$ *satisfies* (3.1) for a vertex $u \in T$ if $\mathbf{t}(v, N) \subseteq \text{cl}(u, T) \subseteq \text{cl}(v, N)$.

We will construct this mapping somewhat greedily. That is, we begin by mapping r_t to r_n, then assigning mappings to each vertex in order, picking the vertex in N that will work best for the vertex in question. Below, we outline **Map** and **CheckTree**. Given a vertex $u \in T$, **Map**$(N, T, u \in T, \phi)$ maps children of u to children of $\phi(u)$ in N. **CheckTree**(N, T) organizes the vertices in T and simply calls **Map** on each of the vertices in T. We will keep track of the *level* of a vertex in T, which will measure how far from the root a given vertex is. That is, a vertex is in level L_n if a shortest path from the root r_t to the vertex has length n. Trivially, $r_t \in L_0$.

Map$(N, T, u \in T, \phi)$ Consider the vertex u in T. If u has children, then order them $c_1, c_2, ..., c_s$. Assume that ϕ is defined for all ancestors of u, $\phi(u) = v$ in N, and, if v has children, order them $d_1, d_2, ..., d_t$.

1. If there exists a child d_j such that $\mathbf{t}(d_j, N) = \mathbf{t}(v, N)$ and $\text{cl}(u, T) \subseteq \text{cl}(d_j, N)$, then(

   Redefine $\phi(u) = d_j$, add (v, d_j) to $E'(N)$, AND $\phi = \mathbf{Map}(N, T, u, \phi)$).

2. Else, If u is not a leaf, then For $i = 1, 2, ..., s$,

   a. If there exists unique child d_k of v such that $\mathbf{t}(d_k, N) \subseteq \text{cl}(c_i, T) \subseteq \text{cl}(d_k, N)$, AND there is not already another child c_l such that $\phi(c_l) = d_k$, then let $\phi(c_i) = d_k$ and add (v, d_k) to $E'(N)$.

   b. Else, stop. Print "N does not display T."

Return ϕ.

CheckTree(N, T) Let ϕ be a partially defined mapping from the vertices of T to the vertices of N. Let $\phi(r_t) = r_n$. Let $E'(N)$ be empty.

1. Place all vertices in T in their appropriate levels. Let P be the largest integer such that L_P is nonempty.

2. $\phi = \mathbf{Map}(N, T, r_t, \phi)$.

3. For $i = 1, 2, ..., P$,

   Let $M = |L_i|$. Order the vertices in L_i as $\{u_{ij}\}$ for $j = 1, 2, ..., M$. For $j = 1, 2, ..., M$,

   If ϕ has not yet been defined for u_{ij}, then $\phi = \mathbf{Map}(N, T, u_{ij}, \phi)$.

4. Print "N displays T."

It should be noted here that by "partially defined mapping," we mean a mapping whose domain is a subset of the vertex set of T. Then, as the algorithm runs, the domain is expanded by defining the map on a new vertex of T that was not in the previous domain of ϕ.

We now state the main theorem.

Theorem 2. *Let N be a normal network and T be a tree on the same leaf set X. N displays T if and only if **CheckTree**(N, T) returns a mapping ϕ without returning "N does not display T."*

Proof. We first show that this algorithm will terminate. Note that $|V(N)|$ and $|V(T)|$ are both finite. Once a vertex u in $V(T)$ is mapped to a vertex v in $V(N)$ in **Map**, it can only be remapped to descendants of v. Since $|V(N)|$ is finite, this can only happen a finite number of times. Step 3 in **Checktree** will only happen a finite number of times, since it is only executed once per vertex in $|V(T)|$. All other steps will happen only once per vertex in $|V(T)|$, which, again, is a finite number of times. Thus, the algorithm will terminate, resulting in either a mapping from $V(T)$ to $V(N)$, or a returned message telling us that the tree is not displayed by the network.

We now show the reverse implication of the theorem. That is, suppose **CheckTree**(N, T) runs and returns "N displays T." Then, we were able to find an injective mapping from the entire vertex set of T to some subset of the vertex set of N. Let N' be a network obtained from N by removing exactly one hybrid edge and suppressing resulting 1-vertices. Note that $\mathbf{t}(v, N) \subseteq \mathbf{t}(v, N')$ and $\mathrm{cl}(v, N') \subseteq \mathrm{cl}(v, N)$. Consider a hybrid vertex h in N. We need to remove all hybrid edges going into h, except one, in order to make N into a tree.

Lemma 6. *Suppose **CheckTree**(N, T) runs and returns "N displays T." Let h be a hybrid vertex in N. Let p and q be distinct parents of h. Let a and b be ancestors of p and q, respectively. Then exactly one of (p, h) and (q, h) will be in $E'(N)$.*

Proof of Lemma. Note that a and p, and b and q are not necessarily distinct. Suppose that there is an (a, x)-path containing the edge (p, h) and a (b, y)-path containing the edge (q, h), both of which pass through h, and both of whose edges are in $E'(N)$. Let s, t, m and n be the

vertices in T such that $\phi(s) = a, \phi(t) = b, \phi(m) = x$ and $\phi(n) = y$. Then, the edges (s, m) and (t, n) correspond, respectively, to the (a, x)-path and (b, y)-paths. During the construction of ϕ, since we have $\phi(s) = a$ and $\phi(t) = b$, we must have selected to map both m and n to h. Note that we might have then mapped both m and n to some descendant of h. From (3.1), we have that $\mathbf{t}(h, N) \subseteq \text{cl}(m, T) \subseteq \text{cl}(h, N)$ and $\mathbf{t}(h, N) \subseteq \text{cl}(n, T) \subseteq \text{cl}(h, N)$. Thus, $\mathbf{t}(h, N)$, which is nonempty since N is normal, is a subset of both $\text{cl}(m, T)$ and $\text{cl}(n, T)$. Thus, there is at least one leaf that is a NPLD of both m and n in T. Since T is a tree, this means m and n must be related. We have several cases.

As the first case, suppose that $x = y$. That is, the paths in question in N actually end at the same vertex. Note that $m = n$ immediately. If $s = t$, then $(s, m) = (t, n)$ in T. This means the edge (s, m) corresponds to two distinct (a, m)-paths in N with edges in $E'(N)$, which is a contradiction, since we only map vertices once in the algorithm. If $s \neq t$, then we have a vertex m in T that has two distinct parents s and t, which clearly violates the fact that T is a tree. Thus, we cannot have $x = y$.

Next, we consider the case when one of the paths ends at h, and the other ends at a vertex y that is a descendant of h. Without loss of generality, let $x = h$, but $y \neq x$. This means $\phi(m) = h$ and $x < y$. This means, however, that $m \leq n$ in T. Since T is a tree and both (s, m) and (t, n) are present in $E(T)$, we must have $s < m \leq t < n$. If $m < t$, then $h < b$, which is clearly a contradiction to the fact that b is an ancestor of h. If $s < m = t < n$, then $b = h$, which forces $b = q = h$, which contradicts the assumption that $q \neq h$. This case cannot occur.

Finally, consider the case when both paths end at descendants of h that are distinct from h and each other. That is, $x \neq y$ and $x \neq h \neq y$. We already have $\phi(s) = a, \phi(t) = b, \phi(m) = x$ and $\phi(n) = y$. During the construction, we must have executed Step 1 of **Map** on both m and n. That is, we must have had that $\mathbf{t}(h, N) = \mathbf{t}(x, N)$ and $\mathbf{t}(h, N) = \mathbf{t}(y, N)$. Suppose $z \in \mathbf{t}(h, N)$. Then, there are (x, z)- and (y, z)-paths which are normal paths in N. This is a contradiction since x and y were distinct, making the paths not normal paths.

All cases ended in contradiction. Thus, for each hybrid vertex h, exactly one parent edge of h will be in $E'(N)$.

□

Next, we wish to show that (3.1) is maintained when we remove edges from $E(N)$ that are not in $E'(N)$. First, a lemma that will be used to show this is given.

Lemma 7. *Let N be a normal network and T a tree such that* ***CheckTree****(N, T) returns "N displays T." Each normal edge in N must be in $E'(N)$.*

Proof. Consider a normal edge (v, d) in N. Let x be a leaf. Suppose that a (d, x)-path exists and has edges only in $E'(N)$. Since the edges in the (d, x)-path must correspond to a path in T, there must exist a vertex c in T such that a (c, x)-path exists in T. Let u be the single parent of c that must exist in T, since T is a tree. By our construction, there must be a path in N with edges in $E'(N)$ that correspond to the edge (u, c) in T. If we had mapped $\phi(u) = v$, then we would have included (v, d) in $E'(N)$. Suppose $\phi(u) = w \neq v$. Then, since there is an edge (u, c), there must be a (w, d)-path in N with edges only in $E'(N)$. If the (w, d)-path does not contain v, then it must contain some other edge (s, d). This contradicts the fact that (v, d) was normal. Thus, (v, d) must be in the (w, d)-path, and (v, d) must be in $E'(N)$.

Note that, if the (d, x)-path has edges not in $E'(N)$, then we can consider the edge not in $E'(N)$ that is closest to the leaf x, apply the argument above, and repeat when we have that the (d, x)-path has edges only in $E'(N)$. □

The next lemma shows that (3.1) still holds, even when we remove edges in $E(N) \setminus E'(N)$. This is to show that the paths in T will have isomorphic images in N from the mapping ϕ that correspond to edges in $E'(N)$.

Lemma 8. *Let N be a normal network and T a tree such that* ***CheckTree****(N, T) returns "N displays T." Then, for any network M that is obtained from N by removing hybrid edges that are not in $E'(N)$, (3.1) still holds.*

Proof of Lemma. Lemma 6 says that, for each hybrid vertex, there will be only one parent edge in $E'(N)$. Thus, we can remove the rest of the hybrid edges going into h without removing any edges from $E'(N)$.

As a base case, we note that, without removing any edges, (3.1) is satisfied by the mapping ϕ that is constructed in **CheckTree**. This trivially shows the result.

Consider the normal network N' with at least one hybrid vertex that is obtained by removing an arbitrary number of hybrid edges not in $E'(N)$ from N. Let M be obtained from N' by removing precisely one hybrid edge $(p, h) \notin E'(N)$ from N'. Note that, each time we remove a parent of a hybrid vertex, we get a new network, and the normal sets and clusters will change. Consider a vertex u in T such that $\phi(u) = v$. The induction hypothesis says that (3.1) is satisfied. That is, $\mathbf{t}(v, N') \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(v, N')$. Since we do not suppress vertices of out-degree 1, the vertex sets of N, M and N' are the all the same. Thus, if $\phi(u) = v$ in N, then $\phi(u) = v$ in M and N', as well. Consider what will happen in M after we remove (p, h) from N'. Let $a \in X$ be a NPLD of v in N'. Then, a is in $\mathbf{t}(v, N'), \mathrm{cl}(u, T)$ and $\mathrm{cl}(v, N')$. In M, since we only remove hybrid edges that are not in $E'(N)$, we will not disconnect the normal path from v to a. Thus, a is in both $\mathbf{t}(v, M)$ and $\mathrm{cl}(v, M)$.

We will show that $\mathbf{t}(v, M) \subseteq \mathrm{cl}(u, T)$ by contradiction. That is, assume that $\mathbf{t}(v, M) \not\subseteq \mathrm{cl}(u, T)$. Then, there is a leaf $b \in \mathbf{t}(v, M)$ that is not in $\mathrm{cl}(u, T)$. However, if b is a NPLD of v, then there must have already been a (v, b)-path in N'. We have two cases. In the first case, the (v, b)-path in N' was already a normal path. Then the edge that was removed to obtain M must not have been into any vertex on the (v, b)-path, except, perhaps, v itself. We can ignore when the edge is going into v since it will not affect any normal sets or clusters of v or its descendants. We must have had that $b \in \mathbf{t}(v, N') \subseteq \mathrm{cl}(u, T)$. This is a contradiction to the beginning assumption. As a second case, the (v, b)-path in N' was not a normal path. This means that, when we removed the hybrid edge to get M, we must have removed an edge into a vertex that was on the (v, b)-path in N'. Let p be the parent of the vertex x such that the edge (p, x) is maintained and where (q, x) was removed. Then, the (v, b)-path must pass through p. Every other edge in the path must have been a normal edge in N'. We now apply lemma 7. We get that the (v, b)-path must actually correspond to an entire path in T through ϕ. But, since $\phi(b) = b$ because b is a leaf, we have that the corresponding path in T is a (u, b)-path. This means $b \in \mathrm{cl}(u, T)$, which is a contradiction to the assumption that b was not in $\mathrm{cl}(u, T)$. Thus, we must have actually had that $\mathbf{t}(v, M) \subseteq \mathrm{cl}(u, T)$.

Finally, we must show that $\mathrm{cl}(u, T) \subseteq \mathrm{cl}(v, M)$. Suppose that this is false. That is, suppose that $\mathrm{cl}(u, T) \not\subseteq \mathrm{cl}(v, M)$ for a contradiction. Let $c \in \mathrm{cl}(u, T)$ but $c \notin \mathrm{cl}(v, M)$. We know that

the (u, c)-path in T corresponds to a path in N' with edges in $E'(N)$, since each edge in T corresponds to a path in N, and we only removed edges that were not in $E'(N)$. Thus, there should be a (v, c)-path in M. But, that would mean $c \in \mathrm{cl}(v, M)$, which is a contradiction to the assumption that $\mathrm{cl}(u, T) \not\subseteq \mathrm{cl}(v, M)$.

Therefore, we have that $\mathbf{t}(v, M) \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(v, M)$ for any vertex $u \in T$ such that $\phi(u) = v$. This is precisely condition (3.1).

□

We can now conclude the reverse implication of Theorem 1.

Lemma 8 shows that we can remove as many hybrid edges not in $E'(N)$ as we need to remove in order to create a tree, and (3.1) will hold throughout the process. Recall that, in a tree, the set of NPLDs is the same as the cluster. If M is the tree obtained from N by removing hybrid edges that are not in $E'(N)$, then we know $\mathbf{t}(v, M) \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(v, M)$ for any $u \in T$ with $\phi(u) = v$. But, since $\mathbf{t}(v, M) = \mathrm{cl}(v, M)$, we actually have that $\mathrm{cl}(u, T) = \mathrm{cl}(v, M)$. That is, each cluster in T appears as a cluster in M, as well.

On the other hand, if there is a cluster of M that is not a cluster of T, then it must occur at a vertex that is not in the image of ϕ. Let $w \in M$ be such a vertex. Clearly, w cannot be the root, nor can it be a leaf because of (3.4). Let w have closest ancestor $a \neq w$ and closest descendant $d \neq w$ such that a and d are in the range of ϕ. These must exist since the root and all leaves of M are in the range of ϕ. Since M is a tree, we must have that $\mathrm{cl}(d, M) \subseteq \mathrm{cl}(w, M) \subseteq \mathrm{cl}(a, M)$. Also, we have that $\mathrm{cl}(d, M)$ and $\mathrm{cl}(a, M)$ appear as clusters of T, as well. In fact, we must have that (a', d') is an edge in T, where $\phi(a') = a$ and $\phi(d') = d$, since a and d were the closest vertices to w in the range of ϕ. If $\mathrm{cl}(w, M)$ does not appear as a cluster of T, then there must be a leaf l such that $l \in \mathrm{cl}(w, M)$ but $l \notin \mathrm{cl}(d, M)$. This means that there is a vertex z such that $a \leq z \leq d$ with two children z_1 and z_2 such that both (z, z_1) and (z, z_2) are in $E'(N)$ where z_1 is on the (a, d)-path. This would imply that (a', d') was not actually an edge, but a path with another vertex in between, which contradicts the assumption.

Therefore, M and T have exactly the same set of clusters. Finally, because of condition (3.3), we can suppress vertices of out-degree 1 and not change ϕ. It is known (and shown in

(16)) that two trees with the same set of clusters are equivalent up to isomorphism. And, since M is displayed by N, we get that T is displayed by N.

We have shown the reverse implication of the theorem (i.e. If **CheckTree** runs and returns "N displays T", then N indeed displays T).

We now show the forward implication. That is, assume that N displays T. We wish to show that the algorithm **CheckTree** will, in fact, return a mapping ϕ that satisfies all of the conditions needed and return "N displays T." We begin by addressing the fact that we need to have an injective mapping that satisfies the conditions throughout the process.

Lemma 9. *After each call of* ***Map***(N, T, u, ϕ)*, ignoring descendants of both the domain of* ϕ *and the image of* ϕ*,* ϕ *is an injective mapping that satisfies conditions (3.1), (3.2) and (3.3), provided that the algorithm does not print "*N *does not display* T*."*

Proof. We begin with domain $\emptyset$. When we start the algorithm, we immediately define $\phi(r_t) = r_n$. Then, after each call to **Map**(N, T, u, ϕ), we increase the domain by the number of children of u. By "injective mapping," we mean only on this domain at each step.

We show, by induction, that ϕ is a mapping after each step. As a base case, just consider the roots. We clearly need the root of T to map to the root of N. This is well-defined and clearly injective. Thus, ϕ is an injective mapping.

Next, we assume that ϕ is an injective mapping on a given domain. Then, when we call **Map**(N, T, u, ϕ), we wish to map the children of u to vertices in N. The only changes we make to the map ϕ are either to change the mapping from $\phi(u) = v$ to $\phi(u) = d_j$, or to define $\phi(c_i) = d_k$. Both of these operations are well-defined. Each vertex in the new domain corresponds to exactly one vertex in N. And, as long as no two vertices in T map to the same vertex in N, we would still have an injective mapping that meets the requirements listed above. We show this below.

Notice that, for any given vertex in T, ϕ is defined for all ancestors of the vertex by construction. Furthermore, no two children of the vertex in T can map to the same vertex in N. Suppose that there is a vertex u in T with child c and a vertex v in N with a child d such that $\phi(u) = v$ and $\phi(c) = d$. Suppose that d satisfies condition (3.1) for c. Suppose,

furthermore, that we already have a distinct vertex w in T that is not a child of u such that $\phi(w) = d$. Then, by (3.1), $\mathbf{t}(d) \subseteq \mathrm{cl}(c, T) \subseteq \mathrm{cl}(d, N)$ and $\mathbf{t}(d) \subseteq \mathrm{cl}(w, T) \subseteq \mathrm{cl}(d, N)$. Thus, $\mathbf{t}(d, N) \subseteq \mathrm{cl}(c, T) \cap \mathrm{cl}(w, T)$. This shows that w and c have a nonempty set of shared descendants. In a tree, this means that w and c are related. That is, either $w \leq c$ or $c \leq w$. We assumed that $w \neq c$. By the construction of the algorithm, w, since it is a descendant of u and not a child of u, would not have been assigned a vertex in N yet. This is a contradiction. If $w \leq c$, then we mapped u, a descendant of w, to an ancestor of $\phi(w) = d$. By construction of the algorithm, we only map descendants of w to descendants of d. Thus, we get another contradiction. Therefore, each vertex in N can have only one vertex in T mapping to it. That is, ϕ is an injective mapping.

Clearly, by construction, (3.1) and (3.2) are satisfied. Note that, if we consider ϕ whose range is the entire vertex set of N, then (3.3) is not necessarily satisfied. However, if we restrict our attention only to those vertices in N that are in the image of ϕ, then (3.3) will be satisfied because of step 1 in **Map**.

Thus, after each call of **Map**, the mapping ϕ is an injective mapping that satisfies the conditions given above, provided a mapping is actually returned.

□

In the previous lemma, we ignored descendants of the domain and image of ϕ because (3.3) might not be satisfied. However, once the algorithm has been executed, since the domain of ϕ must include all of the leaves in T, and the leaves in T must map to the leaves in N by construction, there are no descendants of the domain or image of ϕ to ignore, since leaves have no nontrivial ancestors. Thus, after the algorithm has finished, ϕ is an injective mapping that satisfies all of the conditions on all of the vertices. Furthermore, (3.4) is satisfied.

Next, we consider the cases when the algorithm might fail, (that is, when the algorithm returns "N does not display T.")

Lemma 10. *Let N be a normal network, and T be a tree that is displayed by N. In **CheckTree** (N, T), during step 2 of **Map**(N, T, u, ϕ), for each child c of u, there must always exist at least one child d of $\phi(u)$ that satisfies (3.1) for c.*

Proof. Without loss of generality, we assume that the mapping ϕ on all ancestors of u is uniquely determined. If not, then we can simply look at vertices closer to the root where the mapping on all ancestors of u is uniquely determined. This must occur at some point since the roots must map to each other. Let c have parent u in T and let $\phi(u) = v$ in N.

Suppose no child d of v exists in N that satisfies (3.1) for c. Then, we must map c to a vertex in N that is not a child of v. We already mapped vertices to all ancestors of v. Thus, we must either map c to a descendant of v that is not a child of v, or to a vertex that is completely unrelated to v. The latter option would clearly violate our requirements for the mapping ϕ. Thus, we only consider the former option.

First, we note that, if v has no children, then v has no nontrivial descendants. Then, we trivially cannot map any children of u to any descendants of v, and would be forced to map them to unrelated vertices. In this case, the tree is not displayed by the network, which is a contradiction.

We have, from (3.1), that $\mathbf{t}(v, N) \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(v, N)$. Suppose that, for every child of v, mapping c to that child would violate (3.1) for c and that child. Suppose, furthermore, that some descendant w of v that is not a child of v does satisfy (3.1) for c. That is, $\mathbf{t}(w, N) \subseteq \mathrm{cl}(c, T) \subseteq \mathrm{cl}(w, N)$. Furthermore, suppose that d is a child of v through which a path from v to w passes. We know that $\mathrm{cl}(c, T) \subseteq \mathrm{cl}(u, T)$ because a (c, x)-path from c to some $x \in \mathrm{cl}(c, T)$ can be made into an (u, x)-path simply by adding u to the beginning of the path. Also, $\mathrm{cl}(w, N) \subseteq \mathrm{cl}(d, N) \subseteq \mathrm{cl}(v, N)$ by a similar argument. We get that $\mathrm{cl}(c, T) \subseteq \mathrm{cl}(d, N)$. Thus, since (3.1) was violated for c and d, we must not have been able to choose $\phi(c) = d$ because $\mathbf{t}(d, N) \subseteq \mathrm{cl}(c, T)$ fails. Thus, there exists some vertex $a \in \mathbf{t}(d, N)$ such that $a \notin \mathrm{cl}(c, T)$. We have a few cases to consider.

We begin by considering the case when the normal path from d to a passes through w. In this case, since $d \leq w$ and there is a normal path from d to a, the path must also pass through w. This gives us that the path from w to a is also a normal path, and thus $a \in \mathbf{t}(w, N)$. Then, since $\mathbf{t}(w, N) \subseteq \mathrm{cl}(c, T)$, we must have $a \in \mathrm{cl}(c, T)$. This is a contradiction to the statement above. Thus, we cannot have the normal path from d to a passing through w.

The other case to consider is when the normal path from d to a does not pass through w.

Requirement (3.2) above states that if (u, c) is an edge in T, then there exists a $(\phi(u), \phi(c))$-path in N. In this case, $\phi(u) = v$ and $\phi(c) = w$. That is, the edge (u, c) in T corresponds to the path from v to w. If N were actually to display T, we would need to suppress the extra vertices in this path. Eventually, d must either be suppressed into w or into v. If we contract d and w into one vertex, then there is now a normal path from w to a. This possibility is eliminated in the case above. If we contract d and v into one vertex, then we must actually have been able to map u to d. However, this violates the assumption that we have already executed step 1 in **Map**. Thus, this case cannot occur, either.

Thus, by contradiction, during step 2 of **Map**, if there are no vertices in N that satisfy (3.1) for the given vertex in T, the tree is not displayed by N.

□

This lemma shows that each time we call **Map**, there must be an option for mapping each child of the vertex in question. Next, we show that there can be only one such vertex.

Lemma 11. *Let N be a normal network, and T be a tree that is displayed by N. In **CheckTree** (N, T), during step 2 of **Map**(N, T, u, ϕ), for each child c of u, there must be exactly one child d of $\phi(u)$ in N that satisfies (3.1) for c.*

Proof. Again, without loss of generality, we assume that the mapping on all ancestors of u (including u itself) is uniquely determined. If they are not, then we can simply look at vertices closer to the root, and eventually find one which does have this property.

Consider $c \in T$ with parent u such that $\phi(u) = v \in N$ with children d and e. Note that v could have more children. By lemma 10, there must exist at least one child of v that satisfies (3.1) for c. Suppose that both d and e fit the criterion for the mapping. That is, suppose $\mathbf{t}(d, N) \subseteq \mathrm{cl}(c, T) \subseteq \mathrm{cl}(d, N)$ and $\mathbf{t}(e, N) \subseteq \mathrm{cl}(c, T) \subseteq \mathrm{cl}(e, N)$. We strive for a contradiction. By lemma 1, we know that, for some $x \in \mathbf{t}(d, N)$, there is a unique path connecting d to x. However, the inequalities show us that $x \in \mathrm{cl}(e, N)$. Thus, there is a path from e to x, as well. If the (e, x)-path contains d, then $e < d$. If the (d, x)-path contains e, then $d < e$. However, if $e < d$, then (v, d) is redundant. If $d < e$, then (v, e) is redundant. Either way, we get a

redundant edge that contradicts the fact that N is a normal network. Thus, there can be at most one vertex that d that satisfies condition 3.1 for c.

□

These lemmas show that, given each child of a vertex u in a call of $\mathbf{Map}(N, T, u, \phi)$ has exactly one option to map to a child of $v = \phi(u)$, assuming that we do not remap u to a descendant of v. We now show that each child of c of u such that d satisfies condition 3.1 for c where d is a child of v cannot satisfy condition 3.1 for a different child of u.

Lemma 12. *Let N be a normal network, and T be a tree that is displayed by N. In **CheckTree** (N, T), during step 2 of **Map**(N, T, u, ϕ), for each child c of u, the unique child d of $\phi(u)$ that satisfies (3.1) for c cannot also satisfy (3.1) for another child of u.*

Proof. Again, without loss of generality, assume that ϕ is uniquely determined for all ancestors of u, including u itself.

We consider when two vertices in T might map to the same vertex in N. Consider a vertex $u \in T$ with children b and c such that $\phi(u) = v \in N$ with child d. Suppose that d satisfies the criteria for ϕ for both b and c. That is, suppose that $\mathbf{t}(d, N) \subseteq \mathrm{cl}(b, T) \subseteq \mathrm{cl}(d, N)$ and $\mathbf{t}(d, N) \subseteq \mathrm{cl}(c, T) \subseteq \mathrm{cl}(d, N)$. Since T is a tree, $\mathrm{cl}(b, T) \cap \mathrm{cl}(c, T) = \emptyset$. And, since $\mathbf{t}(d, N) \subseteq \mathrm{cl}(b, T)$ and $\mathbf{t}(d, N) \subseteq (c, T)$, we have that $\mathbf{t}(d, N) \subseteq (\mathrm{cl}(u, T) \cap \mathrm{cl}(v, T)) = \emptyset$. Thus, $\mathbf{t}(d, N) = \emptyset$, which is a contradiction to the fact that N is normal. Thus, d can satisfy condition 3.1 for only one child of u, as desired. □

We have shown that, in step 2a of $\mathbf{Map}(N, T, u, \phi)$, there will always be a unique vertex in N that satisfies that conditions. Thus, we will never print "N does not display T," and will map each vertex in T to a vertex in N. We will have a mapping ϕ_C that satisfies all of the conditions. Thus, we will return "N displays T," the forward direction of the proof is complete, and theorem 2 is proved.

□

We continue with a related theorem about the mapping that is found in the previous theorem.

Theorem 3. *The map found in* ***CheckTree***(N, T) *is unique. That is, there is only one map from* $V(T)$ *to* $V(N)$ *that satisfies the conditions stated above.*

Proof. To see this, consider that, since N displays T, there is a parent map P of edges such that removing the hybrid edges not in P and suppressing vertices of out-degree 1 will yield the tree T. Consider the network N' that is obtained by removing the hybrid edges not in P without suppressing out-degree 1 vertices. A vertex that has out-degree 1 will have the same cluster as its only child. Furthermore, N' is actually a tree without the condition that each internal vertex have out-degree at least 2. Thus, all vertices with the same cluster in T will actually be consecutive vertices in a path. In essence, N' is T with extraneous out-degree 1 vertices.

We will construct a mapping $\phi_P : V(T) \to V(N)$. For each cluster C in T, there will exist at least one vertex in N' with that cluster. Let $\mathrm{cl}(u, T) = C$ for some $u \in V(T)$. If the vertex with cluster C is unique in N', then let u map to the vertex $v \in V(N')$ where $\mathrm{cl}(u, T) = \mathrm{cl}(v, N')$. If there are multiple vertices in N' with the cluster C, then let u map to $v \in V(N)$ where v is the vertex that is farthest from the root with cluster C, (that is, for each vertex $w \in V(N')$ with $\mathrm{cl}(w, N') = C$, $w \leq v$). Extend this mapping to N. This should be unambiguous, since each vertex in N' appears in N. The only difference might be that some vertices have different edges going into or out of them. Call this map ϕ_P.

We wish to show that this mapping satisfies the three conditions above. We begin with the first condition. Note that $\mathbf{t}(v, N') = \mathrm{cl}(u, T) = \mathrm{cl}(v, N')$ where $\phi_P(u) = v$. The difference between N' and N is that N has more hybrid edges. This will make the set $\mathbf{t}(v, N')$ have fewer elements, as well as make $\mathrm{cl}(v, N')$ have more elements. Thus, $\mathbf{t}(v, N) \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(v, N)$. This is condition 3.1. It is clear that condition 3.2 is met between N' and T. Adding hybrid edges will make more connections and paths. However, none of the new paths will have existed in N'. Thus, the paths in N' that correspond to edges in T will still be present in N, and satisfy condition 3.2 for N and T. Finally, condition 3.3 is met because we chose to map vertices in T to vertices in N' that were furthest from the root with the same cluster.

We now need to show that the algorithm **CheckTree**(N, T) will find the same map. We will proceed by induction. Let ϕ_C be the map that is constructed from running the algorithm.

Recall that ϕ_P is the map created from the parent mapping. Let $u \in V(T)$ be the first vertex where the two mappings differ. That is, for all ancestors a of u, let $\phi_P(a) = \phi_C(a)$, and let $\phi_P(u) = v_P$ and $\phi_C(u) = v_C$. Let p be the parent of u in T and $\phi_C(p) = \phi_P(p) = q$. It was shown in lemma 11 that exactly one child of v in N can satisfy condition 3.1 for a child of u in T. Thus, only one of v_C and v_P can be a child of q. Consider where, without loss of generality, v_C is, assuming that v_P is the child of q. We have a few cases.

Case 1: Suppose that v_C is an ancestor of q. Then, we have a path from q to v_C, as well as a path from v_C to q. This creates a cycle in N, which is a contradiction to the fact that networks are acyclic. Thus, this case cannot happen.

Case 2: Suppose that v_C is a descendant of v_P. This means that, during step 1 of $\mathbf{Map}(N, T, u, \phi_C)$, we found that $\mathbf{t}(v_C, N) = \mathbf{t}(v_P, N)$ and $\mathrm{cl}(u, T) \subseteq \mathrm{cl}(v_C, N)$, and thus were forced to change our original assignment of $\phi_C(u) = v_P$ to $\phi_C(u) = v_C$. This means that our original mapping ϕ_P was flawed, however, as condition 3.3 is no longer met for ϕ_P. This is also a contradiction. Thus, this case cannot happen.

Case 3: Suppose that $v_C$is not related to v_P. Since they are unrelated, lemma 2 tells us that they share no NPLDs. That is, $\mathbf{t}(v_C, N) \cap \mathbf{t}(v_P, N) = \emptyset$. From condition 3.1, we get that $\mathbf{t}(v_P, N) \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(v_P, N)$ and $\mathbf{t}(v_C, N) \subseteq \mathrm{cl}(u, T) \subseteq \mathrm{cl}(v_C, N)$. This means that, for some $x \in \mathbf{t}(v_P, N)$, we get that $x \in \mathrm{cl}(v_C, N)$, as well. Lemma 1 tells us that the path from v_P to x is unique. Thus, either the (v_P, x)-path contains v_C, or a (v_C, x)-path contains v_P. Either way, we get that v_C and v_P are related, which is a contradiction. Thus, this case cannot happen.

All three cases ended in contradiction. Thus, the mappings must not actually have differed at u. Since u was arbitrary, this holds for all vertices in $V(T)$. That is, $\phi_P = \phi_C$.

□

In summary, we have shown that this algorithm will determine whether or not a tree T is displayed by a network N with the same leaf set X. Furthermore, the mapping that exists when T is displayed by N is unique, and the algorithm will find this mapping. We finish the chapter with an example and discussion about the complexity of the algorithm.

3.4 An example

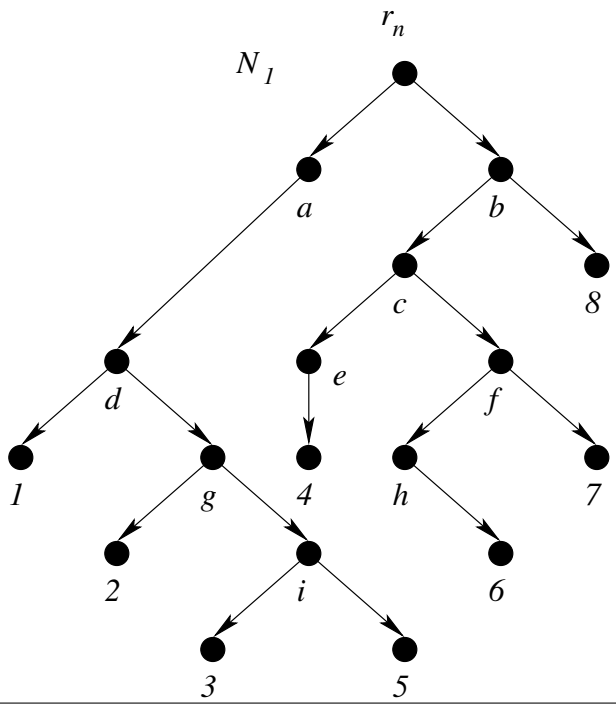
This theorem shows that the algorithm will determine whether a given tree is displayed by a normal network. Furthermore, any tree that is not displayed by the network will result in the algorithm terminating and returning "N does not display T."

We continue with a simple example. The top graph in figure 3.1 is of a network N with root r_n, leaf set $X = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and internal vertices $\{a, b, c, d, e, f, g, h, i\}$. It can be seen by simple inspection that the network is normal. This can be done by checking to see that each vertex in N has a normal path to at least one vertex in the leaf set X.

The bottom two graphs in figure 3.1 are of two trees, T_1 and T_2, on which we will run our algorithm to check to see if N displays either tree. Observe that both trees have the same leaf set X as N does, which is essential.

First, we run **CheckTree**(N, T_1). This tree is displayed by N. The reader may want to attempt to run the algorithm by hand to verify that a mapping ϕ can be found between the vertices of T_1 and the vertices of N that satisfies the conditions listed above. The graph in figure 3.4 is of a network that is obtained from N by removing hybrid edges. Observe that, after suppressing the 1-vertices (i.e. a, e, and h), we get T_1. The graph corresponds to the mapping given by the table in figure 3.4, which is the mapping that is found by **CheckTree**(N, T_1). Note that this corresponds to exactly one of the eight parent maps that exist. This mapping ϕ shows that N indeed displays T_1, as desired.

When we try to run **CheckTree**(N, T_2), however, the algorithm quickly fails. The algorithm will certainly set $\phi(r_{t_2}) = r_n$. Then, it will pick $\phi(a_2) = a$ and $\phi(b_2) = b$. During **Map**(N, T_2, a_2, ϕ), the algorithm will change ϕ so that $\phi(a_2) = d$ since $\mathbf{t}(d) = \mathbf{t}(a)$ and $\mathrm{cl}(a_2, T_2) = \{1, 2, 3, 5\} \subseteq \{1, 2, 3, 5\} = \mathrm{cl}(d, N)$. The algorithm would then call **Map**(N, T_2, a_2, ϕ) again. This time, however, no child x of d exists such that $\mathbf{t}(x) \subseteq \mathrm{cl}(c, T_2) \subseteq \mathrm{cl}(x, N)$. Thus, the algorithm would fail and return "N does not display T_2." Observe that, since there are three hybrid vertices with two parents each, that there are $2^3 = 8$ different trees that N displays. Thus, it is not terribly difficult to simply list all 8 trees that N displays and observe that none of the eight are T_2. That is, N does not display T_2.



v	r_{t_1}	a_1	b_1	c_1	d_1	e_1	f_1	1	2	3	4	5	6	7	8
$\phi(v)$	r_n	d	b	g	c	i	f	1	2	3	4	5	6	7	8

Figure 3.3 A network N_1 found from N by a parent map that yields T_1, and a table showing the mapping ϕ that is found from **CheckTree**(N, T_1).

Note that the network N given in figure 3.1 is a *level*-3 network. The definition of a level-k network is given in (7). It is not difficult to see that level-k normal networks can be constructed for arbitrarily large k. No connection is being claimed. However, it might be of some interest to note that this algorithm will still run properly on level-k networks that happen to be normal.

Finally, it should also be noted that this algorithm is only proved to work for normal networks. Trying to prove this for other classes of networks is difficult because the very definition of the problem becomes ambiguous. For example, there are networks, as in figure 3.2, such that removing hybrid edges can yield extra leaves which do not have labels from the leaf label set X. Thus, the algorithm, as well as the definition of a tree being *displayed* by a network, would have to be changed in order to ensure that this algorithm would work.

3.4.1 Complexity

We now show that this algorithm is polynomial, and hence "fast" on the size of the leaf set X. Let $|A|$ be the number of elements in the set A.

Theorem 4. ***CheckTree*** *runs in polynomial time in* n.

Proof. Say $|X| = n$. Then, $|V| \leq (n^2 + n)/2$, as shown in (18). That is, $|V| \in O(n^2)$.

Lemma 2.2 shows that each vertex in both N and T cannot have more than n children. Each time step 1 is called in **Map**, we check if $\mathbf{t}(d_j) = \mathbf{t}(u)$ and if $\mathrm{cl}(v, T) \subseteq \mathrm{cl}(d_j, N)$, we make, at most, $n + n$ comparisons. To check if $\mathbf{t}(d_j) = \mathbf{t}(u)$ and if $\mathrm{cl}(v, T) \subseteq \mathrm{cl}(d_j, N)$ is quadratic. Thus, checking for suitable d_j in line 1 takes $O(n^3)$ time.

Each time step 2 is called, we must check if $\mathbf{t}(d_k) \subseteq \mathrm{cl}(c_i, T) \subseteq \mathrm{cl}(d_k, N)$. There are $n + n$ comparisons to make. Each comparison runs in $O(n^2)$ time. Step 2 runs once for each child of v. Thus, step 2 runs at most n times, and step 2 is $O(n^4)$ time.

We call **Map** once for each vertex in $V(T)$. Since $|V(T)| \leq 2n - 1$, this step is $O(n)$. Thus, the total amount of time it takes to run **CheckTree** is at most $O(n^5)$. □

This author is not sure that the estimate on the running time is optimal. That is, it might be that the algorithm can run in a better time than $O(n^5)$. However, this does show that the algorithm is polynomial in time.

3.4.2 Discussion

This result was shown independently in (8). Though our proofs are different, the results are the same. In this paper, we give an algorithm to find a mapping between the set of vertices of T to the set of vertices of N that will show that the network displays that tree, whereas the proof in (8) directly constructs a tree from the original network, then checks to see if the original tree and new tree are the same. Both algorithms run in polynomial time. The algorithm in this paper considers more information about individual vertices and their clusters and sets of NPLDs and the connection between vertices in the tree and the vertices in the network. Just how relevant this information is to the general problem is for the reader to decide.

CHAPTER 4. NORMAL NETWORK SPACE

4.1 Introduction

In (13), some rooted binary tree operations are defined. Of particular interest to us is the rooted subtree-pruning and regrafting (rSPR) operation. This operation, as suggested in the name, detaches a subtree of a rooted tree and reattaches it somewhere else. This gives us a way to measure how different two trees are from each other. In fact, it is shown in (13) that, given arbitrary rooted binary trees on the same leaf set, there exists a sequence of rSPR operations that will change one tree into the other. If the number of operations needed to change one tree into another is small, it suggests that the trees are not very different from each other and that the trees will look similar. The smallest number of operations needed to change one tree into another tree gives us a mathematical concept of distance. Determining the smallest number of operations to change one tree into another is NP-hard, as shown in (2). However, it is easy to find a path that is linear with respect to the number of leaves that the trees have. This fact is shown in (13), and we will give an algorithm to demonstrate this result.

We wish to expand the idea of rSPR to binary normal networks. That is, we wish to find a set of operations on binary normal networks such that, given two arbitrary binary normal networks, we can find a sequence of operations to transform one into the other. The goal is to choose operations that are both mathematically simple and biologically significant. In this chapter, we give an example of such a collection of operations. We will show that *binary normal network space* (BNNS), a graph representing the set of all binary normal networks on a common leaf label set, is connected. We also discuss some related results about complexity, normal network space, and counting of normal networks.

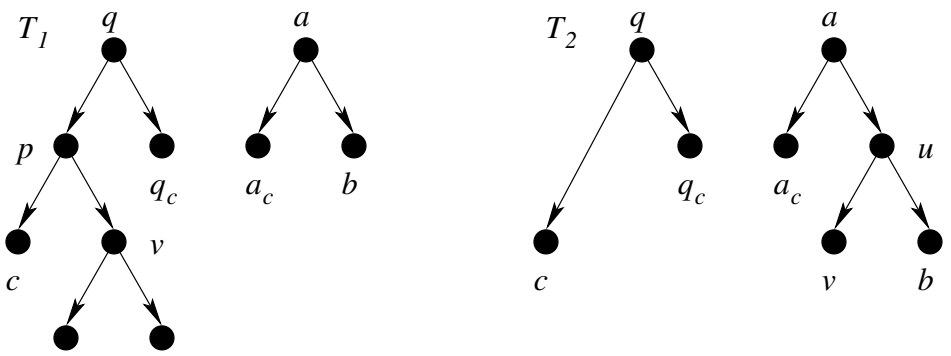


Figure 4.1 A sketch of the vertices involved in a single rSPR operation to change a binary rooted tree T_1 into a different rooted binary tree T_2 on the same leaf set.

4.2 Rooted subtree-pruning and regrafting

We begin by rigorously defining the rSPR operation and rooted binary tree space (rBTS). Instead of rSPR, we will use the letter P as the operation name in equations.

Definition: Let T be a rooted binary tree (V, E) on n leaves. Let v be a vertex in V with parent p. Let p have parent q and children c and v. Let q_c be the child of q such that $q_c \neq p$. Let $e = (a, b)$ be an edge in E. Let a_c be the child of a such that $a_c \neq b$. We define operation $P(T, v, e)$ to be the operation in which we remove the edges $(p, v), (q, p), (p, c), (a, b)$, remove vertex p, add a vertex u, and add edges $(q, c), (a, u), (u, b), (u, v)$. We are essentially detaching the subtree whose root is v and reattaching it where we split the edge (a, b) into two edges. This is what is usually referred to as the rSPR operation. A sketch of the vertices involved in rSPR is given in figure 4.1. Note that v need not be a leaf; we pruned the subtree with root v.

Summary - $P(T, v, e)$

1. Remove $(p, v), (q, p), (p, c), (a, b)$.
2. Remove p.
3. Add u.
4. Add $(q, c), (a, u), (u, b), (u, v)$.

This rSPR operation can be used to change rooted binary trees into other rooted binary trees. We can visualize the set of all rooted binary trees on the same leaf set as a very big

graph. The vertices each represent a unique rooted binary tree. Given two arbitrary rooted binary trees T_1 and T_2, the vertices representing these trees are connected with an edge if there exists a single rSPR operation that changes T_1 into T_2 or vice versa. This is what we call *rooted binary tree space* (rBTS).

Though it was shown in (13), we will show that rBTS is connected with an algorithm that we will use in showing that BNNS is connected. We begin by showing that applying P yields another rooted binary tree.

Proposition 9. *Let T be a rooted binary tree. Let T' be the tree obtained from applying operation P to T once. Then T' is still a rooted binary tree.*

Proof. The only vertices that are affected are a, b, c, q, u, v. We need to show that these each have in-degree 0 or 1 and out-degree 0 or 2. We can ignore p, since it is removed.

Let a have children b and c_a in T. Observe that a still has the same in-degree as it did in T, which must be either 0 or 1. In T', a will have children u and c_a, giving it out-degree 2.

Observe that b, even though it has u as a parent instead of a, still has in-degree exactly 1. It will have the same number of children as it did in T. Since T is a binary tree, b has out-degree 0 or 2.

Observe that c, even though it has q as a parent instead of p, still has in-degree exactly 1. It will have the same number of children as it did in T. Since T is a binary tree, c has out-degree 0 or 2.

Let q have children p and c_q in T. Observe that q still has the same in-degree as it did in T, which must be either 0 or 1. In T', q will have children c and c_q, giving it out-degree 2.

Observe that u has in-degree 1 from parent a and out-degree 2 to children v and b.

Observe that v has in-degree 1 from parent q and will have the same out-degree as it did in T, which must have been either 0 or 2, by assumption.

All other vertices, even though they might have very different ancestors and descendants, still have the same parents and children as they did in T. Thus, all vertices still have in-degree 0 or 1 and have out-degree 0 or 2. Also observe that the root in T still has in-degree 0 in T'. Furthermore, no interior vertices, with the exception of p, which was removed entirely, have

in-degree 0. Thus, r is still the unique root in T'. Finally, we have that T' is a rooted binary tree. □

This proposition shows that, given a rooted binary tree T, the tree that is obtained from applying a single rSPR operation $P(T, v, e)$ is a rooted binary tree on the same leaf set. That is, applying rSPR does not take us outside of rBTS.

Now, we show that each P operation has an inverse, as well.

Proposition 10. *Let T be a rooted binary tree. Let $T' = P(T, v, e)$ for some v and e. Then, there exists a vertex v' and edge e' such that $T = P'(T', v', e')$.*

Proof. When we apply P to T, we remove edges $(p, v), (q, p), (p, c)$ and (a, b), vertex v and add the vertex u and edges $(q, c), (a, u), (u, b)$ and (u, v). Let $v' = v, p' = u, q' = a, c' = b, a' = q$ and $b' = c$. Now, consider $T'' = P'(T', v', e')$ where $e' = (a', b') = (q, c)$. We wish to show that $T = T''$.

Consider two distinct vertices x and y in T. We want to show that $x < y$ in T if and only if $x < y$ in T''.

We start by showing that $x < y$ in T implies $x < y$ in T''. Suppose $x < y$. Since T is a tree, there is a unique path from x to y. We have three cases to consider.

For the first case, let the (x, y)-path pass through v. Then there is an (x, p)-path and a (v, y)-path. We remove the edge (p, v) as well as the vertex p in T'. We then replace the vertex p as well as the edge (p, v). In T'', then, there are (x, p)- and (v, y)-paths as well as the edge (p, v). Thus, in T'', $x < y$.

For the second case, let the (x, y)-path pass through b. Then, in T, there are (x, a)- and (b, y)-paths in T. In T', these paths are unaffected. In addition, there are edges (a, u) and (u, b). Thus, there is still an (x, y)-path in T'. In T'', we remove the edges (a, u) and (u, b) and remove the vertex u, then add back the edge (a, b). Thus, in T'', there is still an (x, y)-path.

For the third case, the (x, y)-path passes through neither v nor b. None of the vertices or edges in the path are affected. Thus, the (x, y)-path exists in T'', as well.

These three cases show that if $x < y$ in T, then $x < y$ in T''.

Now we show that if $x < y$ in T'', then $x < y$ in T. We show the contrapositive. Suppose $x \not< y$. The case when $y < x$ is the same as above. Thus, we can consider only when x and y are not related. That is, there is not a path from x to y in T. We have two cases to consider.

For the first case, suppose $x < y$ in T'. Then, we must have an (x, a)- and a (b, y)-path in T'. However, (a, b) is present in T. Thus, we must have had $x < y$ in T, which is a contradiction. Thus, $x \not< y$ in T'.

For the second case, suppose $x \not< y$ in T'. If $x < y$ in T'', then the (x, y)-path in T'' must pass through v. This implies that (x, p)- and (v, y)-paths exist in T. However, (p, v) is present in T. Thus, $x < y$ in T, which is a contradiction. Thus, $x \not< y$ in T''.

These two cases show that if $x \not< y$ in T, then $x \not< y$ in T''.

Finally, we have that $x < y$ in T if and only if $x < y$ in T''. A similar argument to the proof that A and D are inverses gives us that T and T'' have the same clusters (16). And, since two trees with the same clusters are isomorphic, we have that T and T'' are isomorphic. That is, for $T' = P(T, e, v)$, there is an inverse P' such that $T = P'(T', e', v')$.

□

This shows us that, if two trees T_1 and T_2 are adjacent in rBTS, then there are two rSPR operations P and P' that are inverses of each other such that $P(T_1, v, e) = T_2$ and $P'(T_2, v', e') = T_1$. In other words, if there is an rSPR to change T_1 into T_2, then there is an inverse rSPR operation to change T_2 into T_1.

The *caterpillar tree* on a leaf set X is the tree with clusters $\{\{x_1, x_2\}, \{x_1, x_2, x_3\}, ..., \{x_1, x_2, x_3, ..., x_n\}\}$. A sketch of the caterpillar tree with leaf set $X = \{1, 2, 3, 4, 5, 6\}$ is given in figure 4.2

Next, we show that we can transform T_1 into C, the caterpillar tree on n leaves. Then, since we know that rSPR operations are invertible, we will be able to transform T_1 into T_2 using only rSPR operations.

Lemma 13. *Given a rooted binary tree T with leaf set X, there exists a sequence of rSPR operations to change T into the caterpillar tree C.*

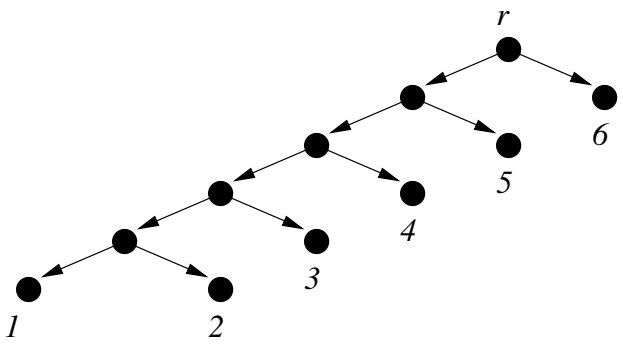


Figure 4.2 The caterpillar tree with leaf set $X = \{1, 2, 3, 4, 5, 6\}$.

Proof. Apply the following algorithm to T. Let $T^* = T$. T' will be for record-keeping purposes. T^* will be the current tree that will eventually become the caterpillar tree.

1. Let r^1 and r^2 be the children of the root r.

2. If $x_n \notin \{r^1, r^2\}$, then { let $T' = P(T^*, x_n, (r, r^2))$, then redefine $T^* = P(T', r^2, (r, r^1))$.
   }
   Let a_{n-1} be the child of r that is not x_n.

3. For$(i = n-1, n-2, ..., 3, 2)\{$
   Let a_i^1 and a_i^2 be the children of a_i.
   If $(x_i \notin \{a_i^1, a_i^2\})$, then{ let $T' = P(T^*, x_i, (a_i, a_i^1))$, then redefine $T^* = P(T', a_i^2, (a_i, a_i^2))$.
   }
   Let a_{i-1} be the child of r that is not x_i.
   }

In the final version of the tree T^*, vertex a_i will have cluster $\{x_1, x_2, ..., x_i\}$ for $2 \leq i \leq n-1$, with the root having cluster $\{x_1, x_2, ..., x_n\}$. This is the tree with clusters $\{\{x_1, x_2\}, \{x_1, x_2, x_3\}, ..., \{x_1, x_2, ..., x_{n-1}\}, \{x_1, x_2, ..., x_n\}\}$, the caterpillar on n leaves.

□

This caterpillar will act as a meeting point for two trees. That is, we can transform any tree into the caterpillar tree, then change the caterpillar tree into any other tree. This avoids the problem of having to change a single tree directly into a possibly very different tree. The

next theorem is the main result for trees. Again this has already been established in (13). We will use this algorithm later.

Theorem 5. *Rooted binary tree space is connected.*

Proof. Let T_1 and T_2 be arbitrary rooted binary trees on the same leaf set $X = \{x_1, x_2, ..., x_n\}$. Let v_1 and v_2 be the vertices in rBTS representing these trees, respectively. We wish to show that there is a path from v_1 to v_2 in rBTS.

We can apply the above algorithm to find two sequences S_1 and S_2 to change T_1 and T_2, respectively, into the caterpillar C. Let $S_1 = \{P_1^1, P_1^2, ..., P_1^s\}$ and $S_2 = \{P_2^1, P_2^2, ..., P_2^t\}$ where s and t are the number of operations needed to perform the algorithm for each tree.

From proposition 10, each operation P_2^i is invertible. That is, for each $T_b = P_2^i(T_a, v, e)$, we can find another rSPR operation Q_2^i such that $T_a = Q_2^i(T_b, v', e')$. Let S_2' be the inverses of S_2 given in reverse order. Whereas S_2 will change T_2 into C, S_2' will change C into T_2 using rSPR operations that are inverses of S_2. We can then concatenate sequences S_1 and S_2' to find a sequence of rSPR operations that will change T_1 into C into T_2. This set of operations yields a set of tree in rBTS. Each vertex representing a tree in the sequence is adjacent to the vertex representing the previous tree in the sequence. Since the set of sequences is finite, we know that a path exists from v_1 to v_2. Finally, since T_1 and T_2 were arbitrary, we have that rBTS is connected.

□

We have shown that rBTS on a common leaf set X is connected. Thus, between any two trees in the tree space, there is a path representing a sequence of rSPR operations to change one tree into the other. In the next section, we extend this to networks.

Observe that, in the algorithm of the theorem, step 2 uses at most 2 rSPR operations, and each time step 3 runs, at most 2 rSPR operations are used. Step 3 runs at most n-2 times. Thus, the algorithm will use at most $2 + 2(n-2) = 2n - 2$ rSPR operations.

4.3 Addition and deletion operations

The rSPR operation is both simple and biologically plausible. As stated, however, the rSPR operation is not sufficient to show that BNNS is connected. Thus, we need to either change rSPR slightly, or add some operations. We will do the latter.

We show that we can transform a binary normal network N_1 into a different binary normal network N_2 using only three types of network operations, two of which are inverses of each other. The first operation is a sequence of reduction moves to get to a binary tree T_1. We remove edges from the tree, identifying vertices with inappropriate in-degree or out-degree along the way. Since a normal network can have at most $n-2$ hybrids by proposition 2, we can accomplish this reduction in $n-2$ moves. We do the same thing to N_2 to obtain a binary tree T_2. This will also take at most $n-2$ moves. We then transform T_1 into T_2 using rSPR moves, as shown in the previous section. Then, using all of these operations, we can transform N_1 into N_2 using only these operations.

Before we define the operations, we give an organizational definition.

Definition: Let a vertex v be called *nice* if it falls into exactly one of the following categories:

- v is the root. That is, v has in-degree 0, out-degree 2, and has a tree child.
- v is an interior vertex. That is, v has in-degree 1 (normal) or 2 (hybrid), has out-degree 2, and has a tree child.
- v is a leaf. That is, v has in-degree 1 (normal) or 2 (hybrid) and has out-degree 0.

If all vertices in a network are nice, there is only one root, and the network has no redundant edges, then N is a binary normal network.

We now define the operations to move between binary normal networks. We begin with a deletion operation.

Definition: Let $N = (V, E)$ be a binary normal network with at least one hybrid vertex h. Let h have parents p_1 and p_2. Let p_1 have, along with h, c as a child. Since N is binary, c and h are the only children of p_1. Since N is normal, c must be normal. Let p_1 have parents P. $P = \{q\}$ if p_1 is normal, and $P = \{q_1, q_2\}$ if p_1 is hybrid. Then define operation $D(N, c, h)$

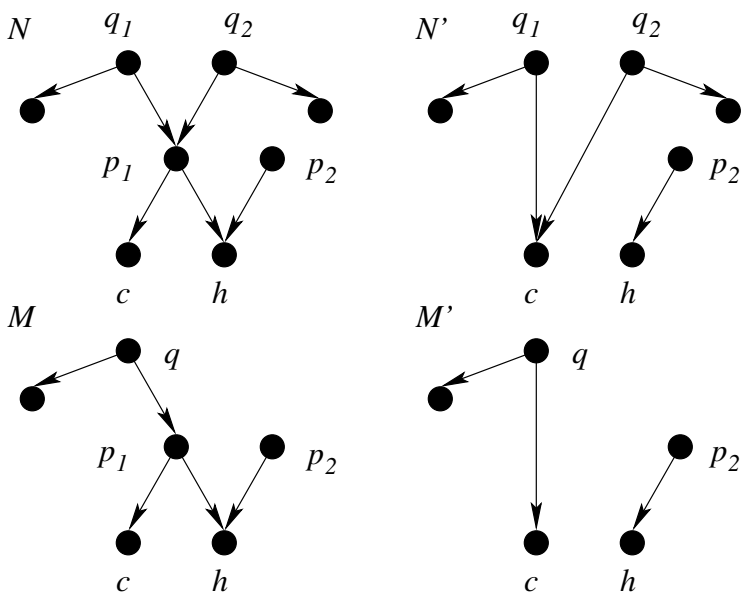


Figure 4.3 Top: $D(N, c, h)$ where p_1 has two parents. The left is N. The right is $D(N, c, h)$. Bottom: $D(M, c, h)$ where p_1 has one parents. The left is M. The right is $D(M, c, h)$.

to be the operation in which we remove (p_1, h) from E and identify p_1 and c (i.e., remove (p_1, c), remove edges going out of P into p_1, discard p_1, and add edges from P to c for each member of P and rename c to be $\{p_1, c\}$). We do this because, after removing (p_1, h), we have a vertex p_1 with out-degree 1. A sketch of this operation is given in figure 4.3, where the top two networks are the case when p_1 is hybrid, and the bottom two networks are the the case when p_1 is normal.

Summary - $D(N, c, h)$

1. Remove $(p_1, h), (p_1, c)$ and (q, p_1) for each $q \in P$.
2. Remove p_1.
3. Rename c to be $\{p, c\}$.
4. Add (q, c) for each $q \in P$.

Applying D is essentially the same as deleting a single hybrid edge (p_1, h). But, because the definition of a binary normal network is strict, we need to clean up the vertices and edges that do not fit the definition. This makes the rigorous definition of D somewhat lengthy.

As we did with P in the previous section, we wish to show that applying D to a binary normal network will yield another binary normal network. Then, in BNNS, applying D will be the same as moving from one vertex in BNNS to another vertex through a single edge.

Proposition 11. *Let N be a binary normal network. Let N' be the network obtained after applying D to N once. Then N' is a binary normal network, as well.*

Proof. We need to show that N is binary and is normal.

Let v be a vertex in N. We wish to show that v is nice in N'. We have several cases to consider.

Suppose $v = c$. Then, in N', v will have the in-edges that p_1 has in N and the out-edges that c has in N. We know that v must have parents P, and that c must either have out-degree 2 and have a tree child or be a leaf. Thus, v has in-degree 1 or 2 and has in-degree 2 and has a tree child or is a leaf. That is, v is nice in N'.

Suppose $v = h$. Then v has in-degree 1 from p_2. If h is a leaf in N, then v is a leaf in N'. If h is not a leaf in N, then it must have out-degree 2 and have a tree child, by assumption. Thus, v has out-degree 2 and has a tree child in N'. Thus, v is nice in N'.

Suppose $v \in P$. Then the parents of v have not been changed. Even though one of the children of v has changed from p_1 to c, it will still have 2 children, since it must have had another child in N by assumption. The other child could not have been c, since c is normal. Thus, v is nice.

Suppose $v \notin \{p_1, h, c\} \cup P$. Then, the operation does not affect any edges going into or out of v. Since v was nice in N, it will be nice in N'.

This covers all vertices in N'. That means that all vertices are nice in N'. Note that there can be only one root since N had only one root, and no new vertices will have in-degree 0 in N'.

The only possible redundant edge we might have is an edge (a, c), where p_1 has parent $a \in P$. However, if (a, c) is indeed redundant, then there must have been a path from a to c not passing through p_1. Then, c must have been hybrid in N. Thus, p_1 must not have had any tree children, since both of its children c and h were hybrid. This contradicts the assumption

that N is normal. All other edges are unaffected by the operation, and are thus not redundant by assumption.

Finally, since N' has one root, all of its vertices are nice, and there are no redundant edges, we have that N' is a binary normal network. □

D is used to reduce the number of hybrid edges and vertices in a binary normal network. We need a similar operation that will be used to increase the number of hybrid vertices and edges. This leads to the following addition operation A.

Definition: Let $N = (V, E)$ be a binary normal network. Suppose that the number of hybrids is less than $n - 2$, where n is the number of leaves. Consider a vertex c with parents P where $P = \{q\}$ if c is normal and $P = \{q_1, q_2\}$ if c is hybrid. Consider a normal vertex h with parent p_2, whose other child c_2 is normal. Assume that $h \not\leq a$ and $p_2 \not\leq a$ for any $a \in P$, and $c \not\leq p_2$. Define the operation $A(N, c, h)$ to be the operation in which we remove any edges going into c from P, add the vertex p_1, and add the edges $(p_1, c), (p_1, h)$ and (a, p_1) for any $a \in P$. A sketch of this operation is given in figure 4.4, where the top two networks are the case when c is hybrid, and the bottom two networks are the case when c is normal.

Summary - $A(N, c, h)$

1. Remove (a, c) for each $a \in P$.
2. Add p_1.
3. Add edges $(p_1, c), (p_1, h)$ and (a, p_1) for each $a \in P$.

We show later that A and D are inverses. First, we show that applying A to a binary normal network that has fewer than $n-2$ hybrid edges will yield another binary normal network. Again, this shows that we can apply this operation and stay inside BNNS.

Proposition 12. *Let N be a binary normal network. Let N' be the network obtained after applying A to N once. Then N' is a binary normal network, as well.*

Proof. We need to show that N' is binary and is normal.

Let v be a vertex in N. We wish to show that v is nice in N'. We have several cases to consider.

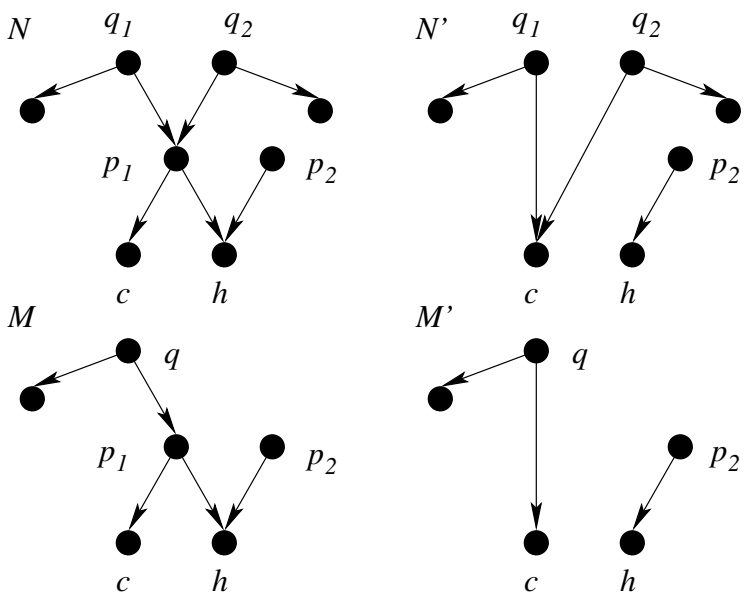


Figure 4.4 Top: $A(N', c, h)$ where c is hybrid in N'. The right is N'. The left is $A(N', c, h)$. Bottom: $A(M', c, h)$ where c is normal in M'. The right is M'. The left is $A(M', c, h)$.

Suppose $v = p_1$. Then, by construction, v has children c and h only. Also, v will have parents P, giving it in-degree 1 or 2, depending on whether c was hybrid or normal in N. Also note that c is now normal in N', giving p_1 a tree child. Thus, v is nice.

Suppose $v = c$. In N', v has only p_1 as a parent, making it normal. Furthermore, c will still have the same set of children in N' as it did in N. And, since c was nice in N, c will have out-degree 0 or 2, depending on whether c is a leaf or an interior vertex. Either way, v is nice since c was nice in N.

Suppose $v \in P$. We did not change the parents of v. Thus, it will still have proper in-degree in N'. It was also have two children, one of which must be p_1. Let c_q be the other child of v in N. Since N was normal, v is nice in N. Thus, either c or c_q was a normal child. In N', c_q is still a child of v, whereas p_1 has replaced c as a child of v in N'. However, we did not change the in-degree of the children. That is, if c_q is normal in N, then it is normal in N'. If c_q is hybrid in N, then it is hybrid in N'. If c is normal in N, then p_1 is normal in N'. If c is hybrid in N, then p_1 is hybrid in N'. In any case, one of c and c_q is normal in N, which means one of p_1 and c_q is normal in N', making v nice in N'.

Suppose $v = h$. In N', h is hybrid with parents p_1 and p_2. We did not change the out-edges, however. Thus, since h was nice in N, it will either be a leaf with 0 out-edges in N' or be a leaf with 2 out-edges, one of which must be going into a normal vertex, making h nice in N'.

Suppose $v = p_2$. We did not change the edges going in p_2. Thus, the number of in-edges is still either 1 or 2. In N', p_2 now has one hybrid child h. We assumed, however, that c_2, the other child of p_2 in N, was normal. Since we did not change the edges going into c_2, it must still be normal in N'. Thus, v is nice in N'.

Suppose $v \notin \{c, p_1, p_2, h\} \cup P$. The edges going into and out of v and its children are unaffected by A. Thus, since v was nice in N, it is still nice in N'.

This covers all vertices in N'. That means all vertices in N' are nice. The only vertex in N' with in-degree 0 is the root. Thus, the root is still unique, as needed.

We must address the possibility of redundant edges. Certainly, for $a \in P$, we know that (a, p_1) is not redundant, since that would imply that (a, c) had been redundant in N. We chose p_2 not be a descendant of c, thus avoiding the possibility of (p_1, h) being redundant. Also, (p_1, c) cannot be redundant since c is normal in N'. We also assumed that $p_2 \not\leq a$, thus avoiding the possibility of (p_2, h) being redundant.

Note that the extra assumption $h \not\leq a$ is simply to avoid creating cycles, which are forbidden in normal networks. Finally, since N' has one root, all vertices are nice, and no redundant edges exist, we have that N' is a binary normal network, as desired.

□

Suppose N_1 and N_2 are binary normal networks in BNNS such that N_1 is obtained from N_2 by applying a single deletion operation D. We certainly would like to be able to find operations to get from N_2 back to N_1. In fact, we can always find a single A operation to do just that. In other words, we can find an inverse operation to get back to N_1 from N_2. This will be true in general. The following proposition shows this.

Proposition 13. *Operations A and D are inverses.*

Proof. Consider a binary normal network N. Let $N' = A(N, c, h)$. We wish to show that there is a corresponding operation D such that $D(N', c', h') = N$. Similarly, we wish to show that if

$D(N, c, h) = N'$, then there is an operation A such that $N = A(N', c', h')$.

Part 1 -

Let $N' = A(N, c, h)$. For each parent $a \in P$ of c, we removed the edge (a, c), added a vertex p_1, and added edges $(p_1, c), (p_1, h)$ and (a, p_1). We assumed that h was normal with parent p_2 and that c_2, the other child of p_2, was normal. Finally, we assumed that $p_2 \notin P, h \not\leq a, p_2 \not\leq a$ for any $a \in P$, and $c \not\leq p_2$.

From the proposition 2 above, we know that N' is indeed a binary normal network. Let $c' = c$ and $h' = h$. Observe that all of the conditions needed to apply D are met. Consider $M = D(N', c', h')$. From proposition 1 above, M will be a binary normal network, as well. We will show that N and M are isomorphic.

Let u and v be vertices in N such that $u < v$. Consider a (u, v)-path S in N. Let $Z = \{c, h, p_1, p_2, c_2\} \cup P$ in N.

Claim: $u < v$ in N if and only if $u < v$ in M.

Proof. Let $u < v$ in N. We have two cases to consider.

Case 1: Suppose none of the vertices of S are in Z. Then, since the only edges that we change are the ones incident to Z, we will not change S. Thus, S is still a path N'. When we apply D, we change only edges that are incident to Z. Since S does not have vertices in Z, we do not change any of the vertices in S. Thus, S is still a path in M. That is, $u < v$ in M.

Case 2: Suppose that at least one vertex in S is in Z. The only edges incident to Z in N that change when we apply A are the edges (a, c) for each $a \in P$. Suppose S has (a, c). That means that there is a (u, c)-path and a (c, v)-path. When we apply A, there is still a (c, v)-path, since none of the descendants of c are changed. If $u = c$, then there is trivially a (u, c)-path. Suppose that $u < c$. Then, for some $a \in P$, there is a path from u to a to c. After applying A, since none of the ancestors of a are change, there is still a (u, a)-path. The edge (a, c) is now an (a, c)-path passing through p_1. These combined give us a (u, c)-path. Thus, there is a (u, v)-path in N'.

Thus, $u < v$ in N implies that $u < v$ in N'. Now apply D. We delete the edge (p_1, h). Note that $u \neq p_1 \neq v$. Thus, u and v are still in M. Suppose S has c. Then there is a (u, a)-path, an

edge (a, p_1), an edge (p_1, c) and a (c, v)-path for some $a \in P$. Then, after applying D, there will be a (u, a)-path, an edge (a, c) and a (c, v)-path. Thus, there is a (u, v)-path. Now, suppose that S does not have c. The only edge that is changed after applying D is the edge (p_1, h). S could not have had this edge in N since it didn't exist. S cannot have this edge in N' by construction. Thus, the path S will still exist in M, since we did not change it.

This shows that $u < v$ in N gives us $u < v$ in M.

Now, we must show that if $u \not< v$ in N, then $u \not< v$ in M. Assume that no paths exist from u to v in N. When we apply A, we change the edges incident to Z only. The changes to c and P do not affect the relationships between these vertices and their ancestors or descendants. Thus, the only way a (u, v)-path could exist in N' is if it has the edge (p_1, h). However, after we apply D, we delete this edge. Thus, in M, there is still no (u, v)-path.

Finally, we have that $u < v$ in N if and only if $u < v$ in M. □

Let u be any interior vertex and v be any leaf. The claim above shows that a leaf is a descendent of u in N if and only if it is a descendent of v in M. Thus, the cluster of u is the same in both networks. Since this vertex was arbitrary, we have that M and N have the same set of clusters. Thus, M and N are isomorphic. That is, for $A(N, c, h) = N'$, there is an inverse $D(N', c', h') = N$.

Part 2 -

Now we need to show the opposite. That is, consider a binary normal network N. Apply $D(N, c, h) = N'$. This will be a binary normal network. Let $c' = c$ and $h' = h$. All of the conditions are met to apply $A(N', c', h') = M$. Again, this is a binary normal network from above. We need to show that $M = N$.

Let u and v be vertices in N such that $u < v$. Consider a (u, v)-path S in N. Let $Z = \{c, h, p_1, p_2, c_2\} \cup P$ in N.

Claim: $u < v$ in N if and only if $u < v$ in M.

Proof. Let $u < v$ in N. The only edges that are changed are (p_1, h), (p_1, c) and (a, p_1) for each $a \in P$. If S passes through c, then there is a (u, a)-path for one $a \in P$, edges (a, p_1) and (p_1, c),

and a (c, v)-path in N. When we apply D, we will have the same (u, a)-path, an edge (a, c), and the same (c, u)-path in N. Thus, $u < v$ in N. If S does not pass through c, then the only other edge that can be affected by D is (p_1, h). If S has this edge, then S will be disconnected in N'. In other words, there is a (u, a)-path for some $a \in P$, edges (a, p_1) and (p_1, h), and an (h, v)-path in N. We delete (p_1, h) and assumed that $c \not\leq p_2$. Thus, S is not a path in N'. However, when we apply A, we reinsert p_1 and the edge (p_1, h), thus restoring the one in edge that was missing. Thus, S is a path in M, and $u < v$, as desired.

Let $u \not\leq v$ in N. That is, there does not exist a (u, v)-path in N. When we apply D, the only new edges that exist are (a, c) for each $a \in P$. However, if (a, c) creates a (u, v)-path S in N', then there is a (u, a)-path, an edge (a, c), and a (c, v)-path in N'. In N, however, there would have been the same (u, a)path, edges (a, p_1) and (p_1, c), and the same (c, v)-path. This gives us a (u, v)-path, contradicting our assumption that $u \not\leq v$ in N. Thus, $u \not\leq v$ in N'.

When we apply A, the edges (p_1, c) and (a, p_1) for each $a \in P$ cannot make a (u, v)-path in M, since this would imply that there was a (u, v)-path in N'. Suppose that (p_1, h) does create a (u, v)-path in M. That is, suppose that there is a (u, a)-path for some $a \in P$, edges (a, p_1) and (p_1, h), and an (h, v)-path in M. Note, however, that (p_1, h) was an edge in N. Thus, in N, we would have had an (u, a)-path, edges (a, p_1) and (p_1, h), and an (h, v)-path. This means a (u, v)-path exists in N, which is a contradiction. Thus, $u \not\leq v$ in M. □

Again, if we apply this claim to interior vertices and their leaf descendants, then we have shown that the set of clusters of N and M are the same. Thus, M and N are isomorphic.

Finally, we have shown that A and D are, indeed, inverses of one another.

□

This proposition shows that, for each operation $N' = D(N, c, h)$ that changes N into N', we can consider $N = A(N', c', h')$ that is the inverse operation A of D that changes N' into N. We can finally show that BNNS is connected. An example of BNNS when $n = 3$ is given in figure 4.6.

Theorem 6. *Given two binary normal networks N_1 and N_2 on the same leaf set, there is a sequence of operations to change N_1 into N_2.*

Proof. Consider two binary normal networks N_1 and N_2 on the same leaf set. We know that a normal network on n leaves has at most $n-2$ hybrid vertices. We can obtain trees T_1 and T_2 from N_1 and N_2, respectively, by applying D operations until there are no hybrid vertices left in the networks. Theorem 5 showed that rBTS is connected. Thus, there exists a sequence of rSPR operations that changes T_1 into T_2.

Let $D_1^1, D_1^2, ..., D_1^{s_1}$ be the s_1 deletion operations needed to change N_1 into a binary tree T_1. Let $D_2^1, D_2^2, ..., D_2^{s_2}$ be the s_2 deletion operations needed to change N_2 into a binary tree T_2. Let A^i be the corresponding inverse operation of D_2^i for each $1 \leq i \leq s_2$. These inverse operations are shown to exist in proposition 13. Let $P_1, P_2, ..., P_{s_3}$ be the s_3 operations needed to change T_1 into T_2. Consider the following operations performed in order.

- $D_1^1, D_1^2, ..., D_1^{s_1}$
- $P_1, P_2, ..., P_{s_3}$
- $A^{s_2}, A^{s_2-1}, ..., A^1$

The first s_1 operations will change N_1 into T_1. The next s_3 operations will change T_1 into T_2. Finally, the last s_2 operations will change T_2 into N_2. And, since each of these inverse operations were shown to exist above, each of the operations is well-defined, and we have a sequence of operations to transform N_1 into N_2.

□

We now give a very basic example. Figure 4.5 gives four networks; N_1 and N_2 are binary normal networks on the same leaf set, and thus in the same BNNS; T_1 and T_2 are the rooted binary trees that arise from applying $D(N_1, 1, b)$ and $D(N_2, 2, 3)$ operation to the networks, respectively. We can then change T_1 into T_2 by applying the rSPR operation $T_2 = P(T_1, 3, (c, 4))$. Finally, observe that $N_2 = A(P(D(N_1, 1, b), 3, (c, 4)), 2, 3)$. That is, N_2 is obtained from N_1 by applying a deletion, rSPR, and addition operation in order. Thus, there is a path of length 3 connecting N_1 to N_2 in BNNS.

We have shown that, given any binary normal networks N_1 and N_2, we can transform N_1 into N_2 using only A, D and P operations. Since each binary normal network has no more

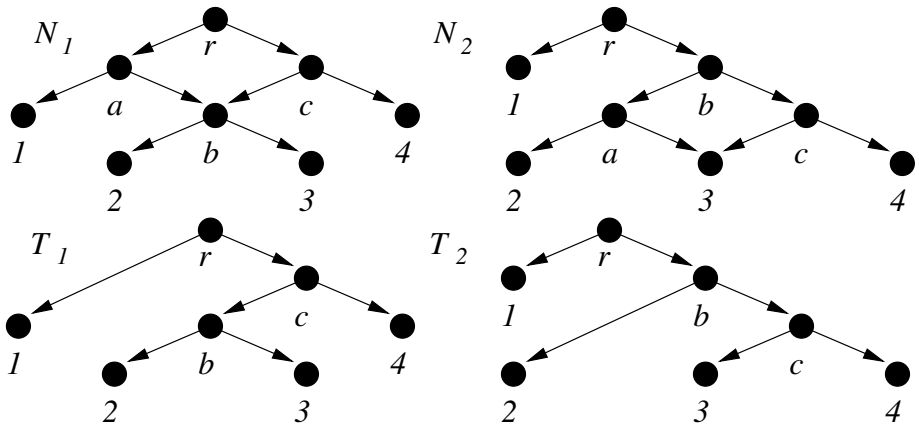


Figure 4.5 N_1 and N_2 are binary normal networks in BNNS. T_1 and T_2 are the rooted binary trees obtained from the networks by applying a single D operation. T_1 can be changed into T_2 using a single rSPR operation. N_1 can be changed into N_2 using one D, one rSPR, and one A operation, thus showing that a path exists from N_1 to N_2 in BNNS.

than $n-2$ hybrid vertices, we know that $s_1 \leq n-2$ and $s_2 \leq n-2$. We can transform T_1 into T_2 in $2n-2$ operations, as shown after theorem 5. Thus, to transform N_1 into N_2 takes no more than $4n-6$ moves. This is, however, not necessarily optimal. In fact, it is known that finding the minimum number of rSPR moves needed to transform one binary tree into another is NP-hard (2). Thus, if we are to rely on rSPR as one of our three base operations, we should expect the problem to be NP-hard for networks, as well.

4.4 Binary Normal Network Space

Just as we did for trees, we can think of all binary normal networks on n leaves as a (very big) graph, where each binary normal network is represented by a unique vertex. Let an edge between two vertices N_1 and N_2 exist if and only if N_1 can be transformed into N_2 using exactly one A, D or P operation. Call this graph *binary normal network space* (BNNS). Theorem 6 tells us that BNNS is, in fact, *connected*. That is, given any two binary normal networks, there exists a path connecting them. Again, we have simply shown that a path exists between arbitrary binary normal networks. Finding a shortest path, (i.e., one with a minimum number of edges) in rooted binary tree space is NP-hard (2). Since our algorithm makes use of the rSPR operation, finding the shortest path between two networks using this algorithm is NP-hard, as

well. An example of BNNS when $n = 3$ is given in figure 4.6. The top graph contains more edges to help illustrate the idea that A and D are inverses, since each red edge is accompanied by an opposite green edge, and vice versa. The bottom graph is binary normal network space when $n = 3$ following the technical definition given above.

We now show that, in fact, each rSPR operation can be accomplished using only A and D operations. Since each rSPR operation essentially moves a single edge, we can think of operation as an addition where the rSPR operation will place the new edge and a deletion where the rSPR operation is getting the edge. However, because binary normal networks cannot have redundant edges, there are some cases where simply applying the corresponding A and D operations will yield a network that is not normal. In these cases, we need to apply two of each operation. This obviously increases the number of moves needed. However, we remove rSPR from the list of operations needed. And, since A and D are inverses of each other, we will have that BNNS is connected by the use of a single operation.

Theorem 7. *Given two binary normal networks N_1 and N_2 on the same leaf set, there is a sequence of only A and D operations to change N_1 into N_2.*

Proof. An $rSPR$ operation is essentially an A and a D operation in one. We would like to say that we can simply apply the needed A operation, then apply the needed D operation. There are some cases, however, when the conditions for applying the A operation are not met. We will have two cases to consider. Let T_1 be an arbitrary rooted binary tree, and suppose that $T_2 = P(T_1, v, e)$.

Consider the vertices to have labels given in the rSPR operation P. Observe that, if $b < v$, then $A(T, b, v)$ will violate the assumption that $b \not\leq p$ because of the redundant edge that would be created in (u, v). Similarly, if $p < a$, then the edge (p, v) would be redundant. Again, $A(T, b, v)$ is not allowed.

Case 1: Suppose that neither $b < v$ nor $p < a$.

Then, observe that all of the conditions are met to apply $A(T_1, b, v)$. Because of the assumptions for this case, no redundant edges will occur. Thus, $A(T_1, b, v)$ is a binary normal network. We now have one hybrid vertex v with parents p and u. We can then apply

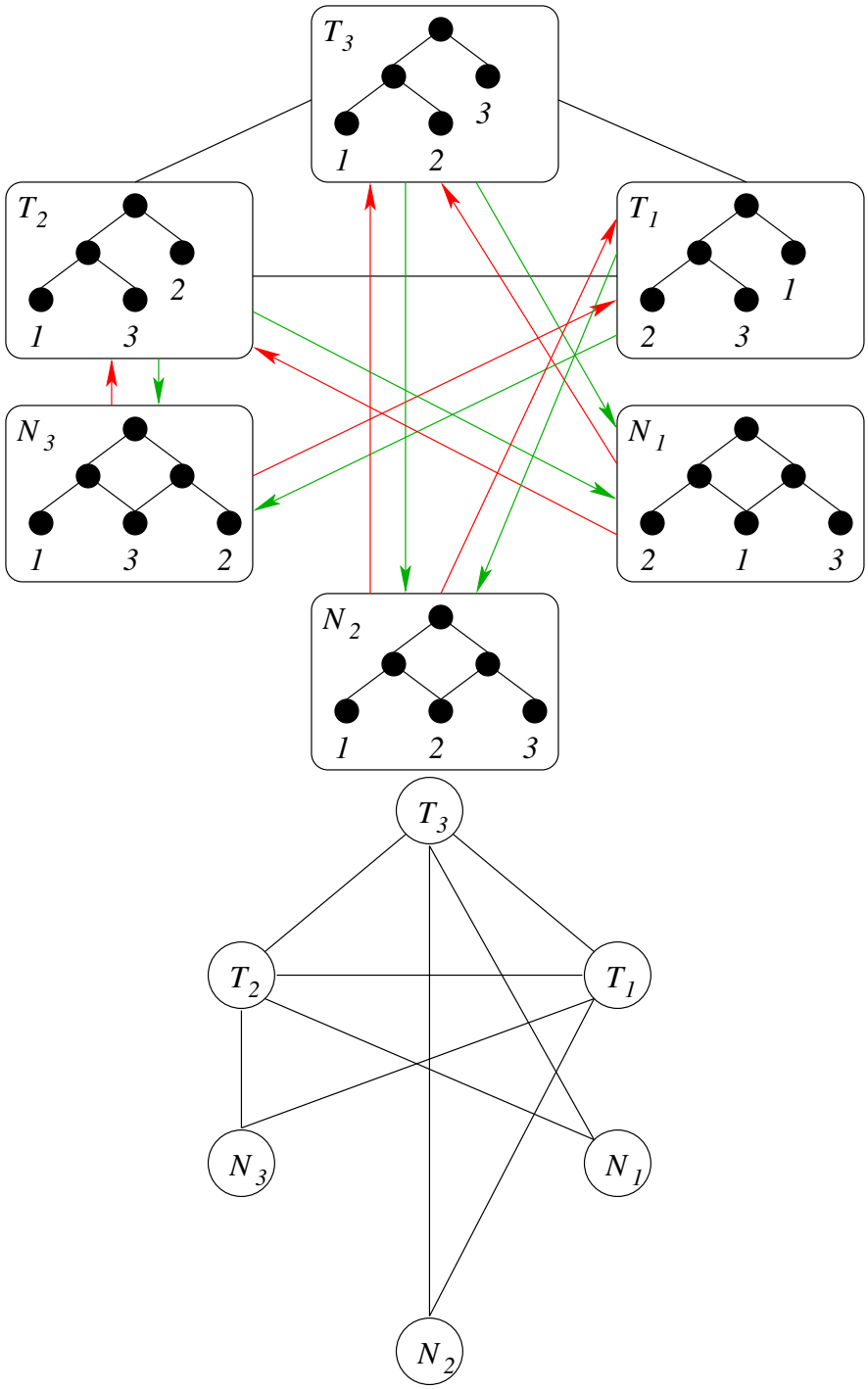


Figure 4.6 Binary normal network space when $n = 3$. The top graph has black edges (rSPR), red edges (D) and green edges (A) representing the operations that can be used to obtain other networks in the graph. The bottom graph is binary normal network space when $n = 3$ following the technical definition given in the text.

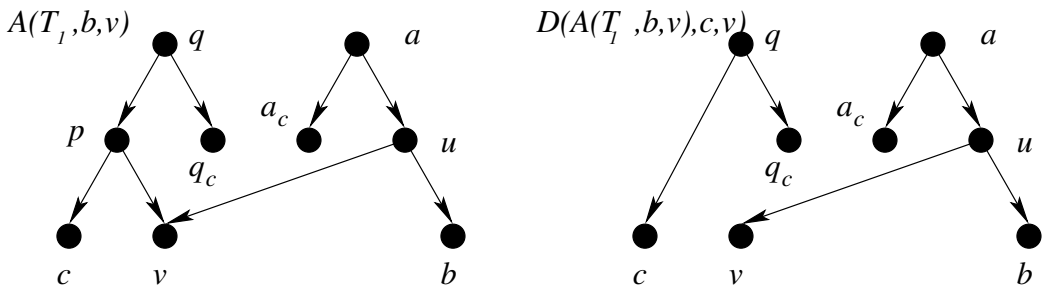


Figure 4.7 A sketch of the result of applying A and D operations instead of an rSPR operation. The left is $A(T_1, b, v)$, and the right is $D(A(T_1, b, v), c, v)$.

$N = D(A(T_1, b, v), c, v)$ to delete the edge (p, v), thus making v normal in N. We wish to show that $T_2 = N$. A sketch of this is given in figure 4.7 in which the left networks is $A(T_1, b, v)$ and the right is $D(A(T_1, b, v), c, v)$.

Observe that the only vertices affected by this operation are $p, q, q_c, c, v, a, a_c, b$ and u, where q_c is the child of q not equal to p, and a_c is the child of a not equal to b. However, it is easy to see that each of these vertices appear exactly in T_2 as they do in N. The parents of q are unaffected by any of the operations, and thus are the same in T_1, T_2 and N. The children of q are c and q_c in both networks. The parent of c is q in T_2 and N. We did not change the children of c or q_c, so they will be the same in each of T_2 and N as they were in T_1 for each vertex. The parent of c and q_c is q in both T_2 and N. Since we did not change the parent of a, it will have the same parent in T_1, T_2 and N. The children of a are u and a_c in N and T_2. The children of a_c are unaffected, and will be the same in T_1, T_2 and N. The only parent of a_c is a in all of T_1, T_2 and N. The only parent of v and b is u in both N and T_2. Furthermore, we did not change the children of either v or b. Thus, they will have the same children in T_2 and N as they did in T_2. The new vertex u has parent a and children v and b in both T_2 and N. Finally, p is not present in either T_2 or N. All other vertices were unaffected by either operation and will appear in T_2 and N precisely as they did in T_1. Thus, $N = T_2$. That is, we can apply one addition, then one deletion operation to T_1 to get T_2 rather than applying an rSPR operation.

Note that we cannot have $b < v$ and $p < a$ at the same time, since this would create cycles. Thus, the following cases are mutually exclusive events.

Case 2: Suppose that $b < v$.

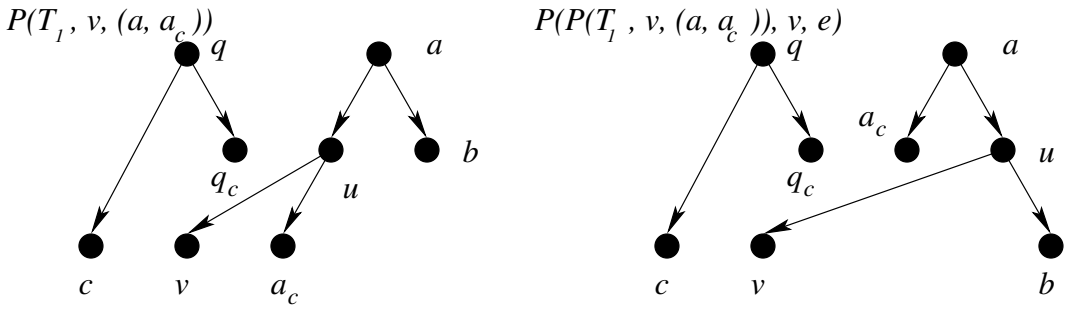


Figure 4.8 A sketch of the result of applying two rSPR operations to a tree to replace a single rSPR operation. The left is $P(T_1, v, (a, a_c))$, and the right is $P(P(T_1, v, (a, a_c)), v, e)$, where $e = (a, b)$.

Rather than applying a single rSPR operation, we can apply two rSPR operations. This becomes less efficient in terms of rSPR operations, but if we can express both rSPR operations as A and D operations, then we reduce the number of types of operations needed. Observe that $T_2 = P(P(T_1, v, (a, a_c)), v, e)$. Instead of reattaching the parent edge of v on the edge (a, b), we first reattach the parent edge of v on the edge (a, a_c), then apply another rSPR operation to reattach the parent edge of v to the edge (a, b). We have simply added an intermediate rSPR operation that will not change the final tree. A sketch of this is given in figure 4.8 in which the left tree is $P(T_1, v, (a, a_c))$, and the right is $P(P(T_1, v, (a, a_c)), v, e)$.

We add this extra rSPR operation because we cannot apply the desired A operation, since we will get a redundant edge (u, v). We get that $a < b \leq v$. Thus, since T_1 is a tree, we know that a_c is not related to v. We can apply $A(T_1, a_c, v)$ without making (u, v) redundant, since there was not already an (a_c, v)-path. We can then apply $N = D(A(T_1, a_c, v), c, v)$. In N, $b \not< v$. We still avoid the final case, since it would, again, imply cycles. By an argument similar to the one found in case 1, $N = A(T_1, v, (a, a_c))$. Now, we can apply the first case, since we no longer have $b < v$. We get that $T_2 = D(A(D(A(T_1, a_c, v), c, v), b, v), a_c, v)$. That is, we can change T_1 into T_2 using two A and two D operations instead of an rSPR operation.

Case 3: Suppose that $p < a$.

We apply a very similar argument as in case 2. That is, rather than trying to represent a single rSPR operation as an A and a D operation, we will represent the rSPR operation as two A and two D operations. First, note that $T_2 = P(P(T_1, v, (q, q_c)), v, e)$. This adds

an intermediate rSPR operation, as in case 2, that allows us to apply appropriate A and D operations to avoid redundant edges. We get that $T_2 = D(A(D(A(T_1, q_c, v), c, v), b, v), q_c, v)$. In this case, the first A and D operations are applied to make $p \not\leq a$. Then, as in case 2, the second A and D change the new tree into T_2, as desired.

□

As an example, consider the networks in figure 4.5. The algorithm given certainly shows that the vertices representing the networks are connected by a path. However, the algorithm uses rSPR operations. Instead, we can skip the rSPR operation entirely and observe that $N_2 = A(D(N_1, 1, b), 4, 3)$. We can skip the intermediate step of changing T_1 into a different tree T_2, and get from N_1 to N_2 using only A and D operations. Note that, in this case, we actually reduced the number of operations needed to change N_1 into N_2. This might not be the case in general, however.

Since A and D are inverses of each other, and we have eliminated the need for rSPR, we are essentially using a single operation type, which is desirable. However, we increase the distance between some networks in BNNS. Exactly how much more desirable one property is over the other has yet to be determined. Recall that the algorithm given to find a path between two binary normal networks required the use of no more than $4n-6$ operations, $2n-2$ of which were rSPR operations. Since we have shown that we can replace rSPR operations with no worse than four A and D operations, we can find a path connecting two arbitrary binary normal networks in BNNS that has at most $(n-2) + (n-2) + 4(2n-2) = 10n - 12$ edges corresponding to only A and D operations. An example of BNNS when $n = 3$ using only A and D operations is given in figure 4.9. Observe that this graph is simply a cycle.

4.5 Counting binary normal networks

We know that the number of unrooted binary trees on n leaves is $(2n-5)!!$. This can be found by induction by counting the number of edges of each binary tree on a fixed number of leaves, each of which gives a spot to attach the new leaf. To count the number of rooted binary trees on n leaves, observe that a rooted binary tree on n leaves can be considered an unrooted

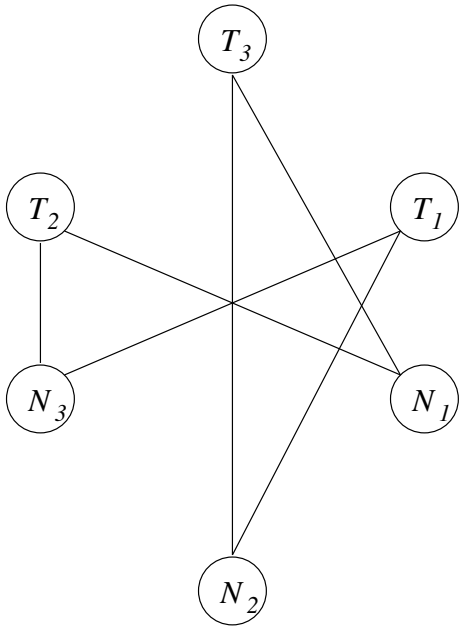


Figure 4.9 Binary normal network space with $n = 3$ in which edges represent a single A or D operation, and not rSPR operations.

binary tree on $n + 1$ leaves where the root is a leaf. With this slight alteration, the number of rooted binary trees is actually $(2(n + 1) - 5)!! = (2n - 3)!!$. We would like to know how many binary normal networks exist on the same leaf set. The first upper bound we discuss is based on the fact that normal networks are regular.

Proposition 14. *The number of binary normal networks on n leaves with label set $X = \{x_1, x_2, ..., x_n\}$ is at most* $\sum_{2=n-2}^{2n-4} \binom{2^n - n - 2}{s}$.

Proof. The number of regular networks is easy to count. Each regular network is uniquely determined by its set of clusters. Each must include the singleton clusters and the root must have cluster $\{1, 2, 3, ..., n\}$. Furthermore, each cluster is non-empty. Then, each unique subset of the remainder of the power set of $\{1, 2, 3, ..., n\}$ uniquely determines a regular network. Thus, the number of regular networks is $2^{2^n - n - 2}$. Since we know that normal networks are also regular, this number can act as an (extremely crude) overestimate of the number of binary normal networks. Note that, since normal networks are regular, each vertex in a normal network has a unique cluster. We showed in proposition 6 that the number of vertices in a binary normal network on n leaves is bounded below by $2n - 1$ and above by $3n - 3$. We must

include the singleton clusters and the root cluster of $\{1, 2, 3, ..., n\}$. Thus, we are free to choose the remaining clusters from the power set of $\{1, 2, 3, ..., n\}$. Of course, some of these options will not yield binary normal networks. However, we will certainly get all possible binary normal networks on the given leaf set through this method. We must have at least $2n - 1$ vertices and can have at most $3n - 3$ vertices. Since we are forced to include $n + 1$ vertices, we can choose between $n - 2$ and $2n - 4$ additional clusters to add to the base set. Excluding the base set and the empty set, there are $2^n - n - 2$ clusters from which to choose. This gives us $\sum_{s=n-2}^{2n-4} \binom{2^n - n - 2}{s}$ as an upper bound for the number of binary normal networks. See table 4.1 for some figures for this upper bound and how it compares to the other upper bounds that we have. □

We can also use the operations discussed above to get a better upper bound on the number of binary normal networks.

We begin with a proposition. We wish to show that the order in which we apply A operations does not affect the outcome. That is, given a binary tree and several A operations, applying the A operations should yield the same binary normal network, independent of order. We need a new definition to continue. Let P be a sequence of vertices $v_1, v_2, ..., v_k$. We say a network *displays a path* P if, for each i such that $1 \leq i \leq k - 1$, either $(v_i, v_{i+1}) \in E(N)$ or v_i and v_{i+1} are the same vertex, (i.e. there is a vertex u such that v_i and v_{i+1} are both in the label set of u). This definition certainly includes the traditional definition of a path, in which each (v_i, v_{i+1}) is an edge in the network. It also includes a slight variation that is necessary in the following proposition.

Proposition 15. *Consider a binary normal network N. Let (p_1, h_1) and (p_2, h_2) be two hybrid edges in N with $h_1 \neq h_2$. Then, the networks obtained from deleting these edges in either order are the same. That is $D(D(N, c_1, h_1), c_2, h_2) = D(D(N, c_2, h_2), c_1, h_1)$.*

Proof. Consider a binary normal network N with $t \geq 2$ hybrids. Let h_1 and h_2 be two hybrids with parents p_1 and p_2, respectively. Let p_1 and p_2 have normal children c_1 and c_2, respectively. Consider a vertex $v \notin \{p_1, p_2\}$ and a leaf x. Let $N' = D(N, c_1, h_1), N'' = D(N', c_2, h_2)$. Let $M' = D(N, c_2, h_2), M'' = D(M', c_1, h_1)$. Essentially, N'' is obtained from N by removing

(p_1, h_1), then (p_2, h_2), whereas M'' is obtained from N by removing (p_2, h_2), then (p_1, h_1). We wish to show that $N'' = M''$.

Suppose $v \not< x$ in N. Since we only delete edges and vertices, we do not add any paths. That is, $v \not< x$ in either M'' or N''.

Suppose $v < x$ in N. Consider a (v, x)-path P in N. We wish to show that N'' displays P if and only if M'' displays P. We have several cases to consider. Before that, observe that a path with c_1 that begins at a proper ancestor of c_1 must contain p_1, since not having p_1 would imply that c_1 is hybrid, which contradicts our assumption that c_1 is normal for our deletion operation to be valid.

Case 1: Suppose that P contains neither (p_1, h_1) nor (p_2, h_2) in N. If P does not contain p_1 or p_2, and thus not c_1 or c_2, then we do not delete any edges in P, nor do we identify and edges in P, and P is displayed by N', N'', M' and M'' in the traditional sense. If P does contain p_1, then p_1 and c_1 will be identified. That is, there will be a vertex with label $\{p_1, c_1\}$. The same is true for p_2 and c_2, if p_2 is in P. All of the labels of P in N will still be present and none of the edges in P were deleted. Thus, (v_i, v_{i+1}) is an edge, except when $v_i \in \{p_1, p_2\}$ where v_i and v_{i+1} are identified to the same vertex. We have that N', N'', M' and M'' all display the path P, though they might not contain the path itself.

Case 2: Suppose that P contains one of (p_1, h_1) or (p_2, h_2). Without loss of generality, suppose that P contains (p_1, h_1). Then, in N', the path is broken. That is, N' does not display P because (p_1, h_1) is neither an edge in N', nor are p_1 and h_1 identified. In N'', since we only delete more edges, P will still not be displayed. In M', however, P will still exist, since we assumed that P did not contain (p_2, h_2). It is possible that P contains p_2 and not h_2. This means, however, that P also contains c_2. After applying D, c_2 now has label $\{c_2, p_2\}$. Then, M' still displays P. In either case, however, since we delete (p_1, h_1) in M'', P will no longer be displayed, since (p_1, h_1) is not an edge in M'', nor are p_1 and h_1 identified. A basic sketch of this case is given in figure 4.10. Observe that, in both M'' and N'', the path P, which was present in N, is no longer displayed by either network. Other cases have sketches very similar to this one.

Case 3: Suppose that P contains both (p_1, h_1) and (p_2, h_2). In N', (p_1, h_1) is no longer

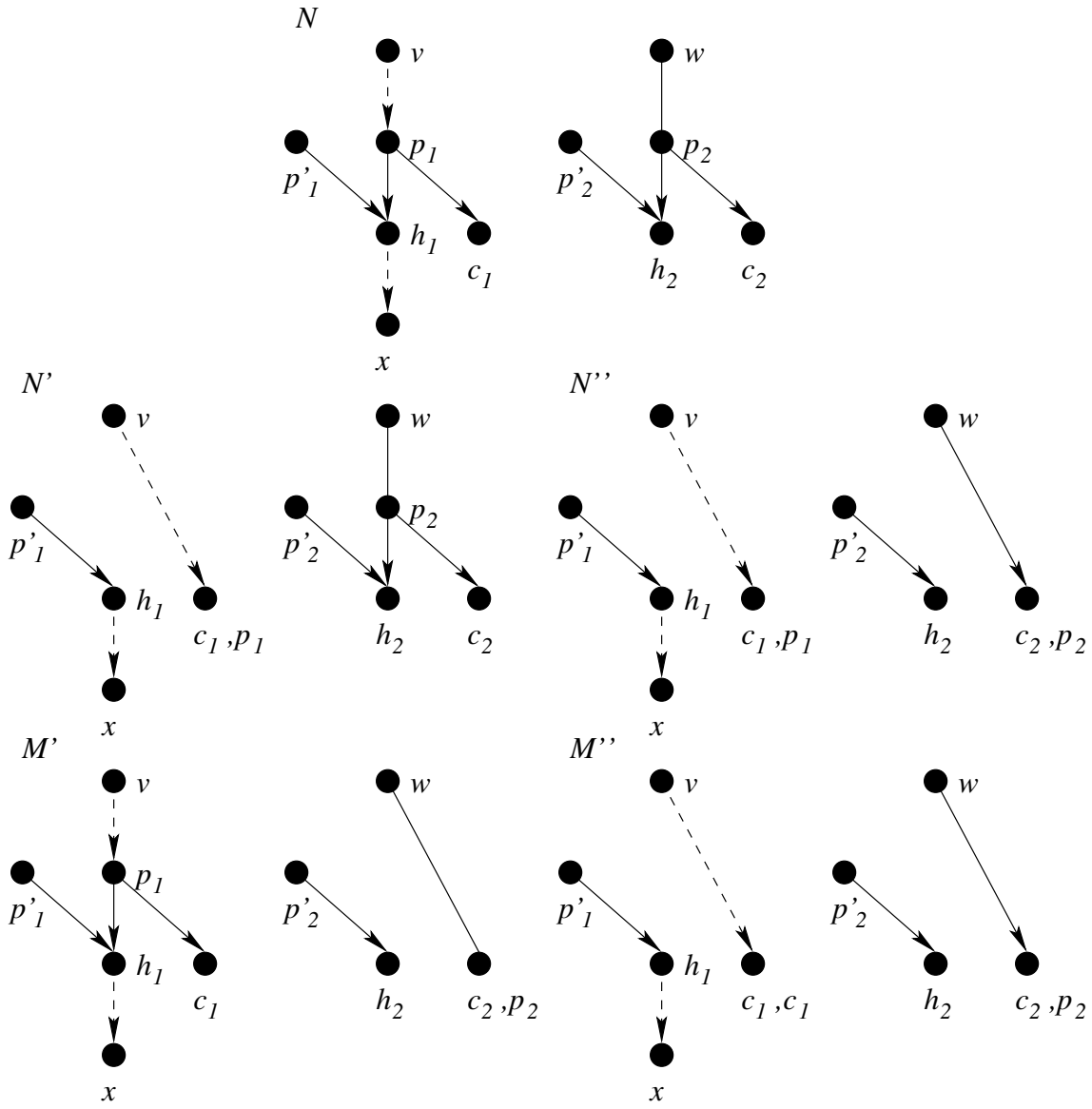


Figure 4.10 A binary normal network N, and networks $N' = D(N, c_1, h_1)$, $N'' = D(N', c_2, h_2)$, $M' = D(N, c_2, h_2)$, and $M'' = D(M', c_1, h_1)$, the results of deleting hybrid edges (p_1, h_1) and (p_2, h_2) in opposite order. The path from v to x is present in N, but broken in both N'' and M''.

an edge, and we have not identified p_1 with h_1. In M', (p_2, h_2) is no longer an edge, and we have not identified p_2 with h_2. Thus, P is displayed by neither N' nor M'. Furthermore, since we only delete more edges and vertices when we apply another deletion operation, we get that neither M'' nor N'' display P.

These cases show us that P is displayed by N'' if and only if P is displayed by M'' for any path displayed by N. We now show that all paths displayed by N'' or M'' are paths displayed by N, as well.

Consider a path Q displayed by N'' or M''. If none of the vertices from Q have multiple labels, then the vertices all appear as they did in N. That is, Q is a path displayed by N.

Suppose Q contains a vertex v with multiple labels. We have two cases to consider.

Case 1: Suppose that v has two labels. Without loss of generality, suppose v has label $\{p_1, c_1\}$. We deleted p_1 from N and renamed c_1 to be $\{p_1, c_1\}$. Let u be the parent of v. Consider u in N. By construction, we either have that u is a parent of p_1, or that u is a parent of c_1. Notice that, if u is indeed a parent of c_1, then c_1 has parents u and p_1, making it hybrid. This contradicts the fact that c_1 is normal, which we assumed in order for $D(N, c_1, h_1)$ to be a valid operation. Thus, u must have been a parent of p_1, and Q must have been displayed by N.

Case 2: Suppose that v has three labels. This means that, either $p_2 = c_1$ or $p_1 = c_2$ and v has label $\{p_1, p_2 = c_1, c_2\}$ or $\{p_2, c_2 = p_1, c_1\}$. Let u be the parent of v. If $p_2 = c_1$, then, in N, we must have had edges (p_1, c_1) and (p_2, c_2) adjacent to each other, and u must have been the parent of $p_1, p_2 = c_1$, or c_2. If u had been the parent of c_1 or c_2, then we would have contradicted the assumption that both c_1 and c_2 were normal. Thus, u must have been the parent of p_1. Similarly, if $p_1 = c_2$, we get that u must be the parent of p_2, and not c_1 or c_2. Thus, we have either u, p_1, c_1, c_2 or u, p_2, c_2, c_1 as a sequence of vertices in P such that each sequential pair is an edge. The rest of Q is unaffected. Thus, Q must actually be displayed by N in both cases.

This shows us that each path displayed by N'' or M'' is actually a path displayed by N, as well. Thus, we know that, for all paths P, N'' displays P if and only if M'' displays P.

Now, consider a vertex v and a leaf x. Consider all (v, x)-paths. Since a (v, x)-path is

displayed by N'' if and only if it is displayed by M'', we can say that $v < x$ in N'' if and only if $v < x$ in M''. Thus, the cluster of v is the same in N'' as it is in M''. And, since a regular network is uniquely determined by its clusters, and N'' and M'' have the same set of vertices, the networks N'' and M'' are the same. Thus, deleting these edges will yield the same network, no matter what the order of deletion is. □

Because the order of deletion does not affect the outcome, and each deletion has an inverse, the order in which we add the corresponding inverse operations will not affect the outcome. We can use this to find an upper bound for the number of binary normal networks.

Corollary 2. *The number of binary normal networks on n leaves with label set* $X = \{x_1, x_2, ..., x_n\}$ *is at most* $\sum_{t=0}^{n-2} \frac{\binom{4n^4-14n^3+13n^2-4n}{t}}{2^t}(2n-3)!!$.

Proof. We first count the number of addition operations that can be applied to a binary tree. Each addition operation is defined by two vertices; the vertex which will become a hybrid vertex, and the vertex that will become the other child of the hybrid vertex's parent. We know that, in a binary tree, there are $2n-2$ edges and $2n-1$ vertices.

Consider the case when, for two distinct addition operations $A_1(T, c_1, h_1)$ and $A_2(T, c_2, h_2)$ on a given rooted binary tree T, $c_1 = c_2$. We can apply one of A_1 or A_2. When we apply the other addition operation, however, the choice for c_i becomes ambiguous. This is due to the fact that, when we delete the edges with D operations, we delete vertices in the network. Thus, to avoid this ambiguity, we will create dummy vertices on each edge. That is, for each edge (a, b) in T, delete (a, b), add $(n-2)$ vertices $(a, b)_1, (a, b)_2, ..., (a, b)_{n-2}$ and edges $((a, b)_i, (a, b)_{i+1})$ for each i such that $1 \leq i \leq n-3$. We choose to add $n-2$ edges since we cannot have more than $n-2$ hybrid vertices in the network by lemma 2. Each pair of vertices in the network yields a unique addition operation while also avoiding the ambiguity that arises when we leave out the dummy vertices. After all addition operations have been applied, we can clean up the excess out-degree 1 vertices. There will be $(2n-1)(n-2)+(2n-1) = 2n^2-3n+1$ vertices in the altered tree T'. Each pair of vertices yields a unique addition operation option. Note that we cannot choose c and h to be on the same set of $(a, b)_i$ vertices. Thus, there are, at most, $(2n^2-3n+1)(2n^2-4n) = 4n^4-14n^3+13n^2-4n$ choices for addition operations. Note that

n	$(2n-3)!!$	x	y	z	2^{2^n-n-2}
5	1.0E2	2.6E9	2.1E10	2.5E5	3.4E7
10	3.4E7	1.0E36	2.6E38	5.2E34	4.4E304
15	2.1E14	1.6E68	1.3E72	6.1E90	1.1E9859
20	8.2E21	5.9E103	1.5E109	1.5E175	1.6E315646
25	1.2E30	5.6E141	4.7E148	2.8E288	2.5E10100882
30	5.0E38	4.1E181	1.1E190	7.6E409	*

Table 4.1 Figures for the counts of the number of rooted binary trees, binary normal networks, and regular networks.

this is already an overestimate, because some additions would not be allowed in our definition of a binary normal network, since we do not allow redundant edges. Since there are $(2n-3)!!$ binary trees, there are, at most, $(4n^4-14n^3+13n^2-4n)(2n-3)!!$ binary normal networks with one hybrid edge. Similarly, since the proposition above showed that the order of addition does not matter, there are, at most, $\binom{4n^4-14n^3+13n^2-4n}{t}(2n-3)!!$ binary normal networks with t hybrids for $1 \leq t \leq n-2$. Then, we can simply add these together to get an estimate of $\sum_{t=0}^{n-2} \binom{4n^4-14n^3+13n^2-4n}{t}(2n-3)!!$ for the number of binary normal networks.

Note that most networks will be counted several times this way. Consider a network N with t hybrids. There are 2^t different unique parent maps that each yield a unique binary tree. Each of these trees will include N in its list of possible binary normal networks that can be obtained from addition operations. Thus, each network is counted 2^t times. A better estimate, then, would be $\sum_{t=0}^{n-2} \frac{\binom{4n^4-14n^3+13n^2-4n}{t}}{2^t}(2n-3)!!$.

Included in table 4.1 are some quantities to show how our improved upper bound compares to our very crude upper bound given by the regular network count. □

Let $x = \sum_{t=0}^{n-2} \frac{\binom{4n^4-14n^3+13n^2-4n}{t}}{2^t}(2n-3)!!$ and $y = \sum_{t=0}^{n-2} \binom{4n^4-14n^3+13n^2-4n}{t}(2n-3)!!$ and $z = \sum_{2=n-2}^{2n-4} \binom{2^n-n-2}{s}$.

CHAPTER 5. CONCLUSION

5.1 Discussion

In the author's admittedly biased opinion, the class of normal networks is an interesting and viable research tool for the subject of phylogenetics. Many of the results in this paper show this. Whereas most research on phylogenetics has been done in trees, we wish to consider networks, which are gaining interest more and more every day. Again, networks are far too general to be of practical use. We wish to limit our research to a particular class of networks that are more complicated than trees, but not to the point of being unusable. We feel that normal networks offer a nice intermediate step.

In chapter 2, we gave tight bounds on the number of vertices and edges that a normal network can have, as well as bounds that specific cases of normal networks can have. We also showed bounds on types of edges and vertices that normal networks can have. More importantly, the bounds on the number of edges and vertices in each of the results is at most quadratic on the number of leaves, and the number of hybrid vertices is linear on the number of leaves. The number of edges and vertices that a tree can have are both linear on the number of leaves. Also, trees cannot have hybrid edges. Thus, normal networks give a nice step up in complexity from trees in all of these bounds. Furthermore, since, in normal networks, internal vertices must have at least one NPLD, there is a tree-like structure that still exists.

In chapter 3, we showed that the tree containment problem is solvable in polynomial time. Since most research being done is on trees, especially gene trees, any network that is found in an attempt to explain all of the given gene trees being considered should aim to display all or most of the given gene trees. This result shows that it is computationally easy to do this for a given tree.

In chapter 4, we gave an extension of the rooted subtree pruning and regrafting operation. Since the rSPR operation does not apply to normal networks, we found two new operations supplementing rSPR to show that binary normal network space is, in fact, connected. That is, given two binary normal networks on the same leaf set, there is a set of these three operations to change one into the other. In fact, we also showed that rSPR can be replaced with the two new operations completely, thus making rBTS connected using only the new operations. We also touched briefly on counting binary normal networks by showing that these operations can sometimes be applied without concern for the order in which they are applied.

Though normal networks are still relatively new, these results, along with other results already published by other authors, give a compelling argument that normal networks are an interesting class of networks to study.

5.2 Future work

Much work is still to be done with normal networks. Here, we discuss some possible directions that research might follow in the future.

5.2.1 Chapter 2

We have found general bounds for normal networks. Some other interesting cases of normal networks might arise. Bounds on vertices and edges of different types might be useful.

5.2.2 Chapter 3

The tree containment problem and cluster containment problem are solvable in polynomial time for normal networks. The next step is to construct normal networks to display a given set of input trees. Also of interest would be a way to "normalize" a network the way a cover digraph "regularizes" a general network. That is, a cover digraph of a general network is regular, and we wish to find a way to find a normal network that is related to a general graph. Then, we can relate general networks by their normalized versions and apply the algorithm given in chapter 3 to test if trees are displayed by them.

For the algorithm in chapter 3 to work, the leaf sets of the network and of the tree need to be the same. Consider, instead, a collection of gene trees in which some trees are missing some species. Then, the leaf set of any species tree will need to be the union of all leaf sets of the gene trees. Thus, this algorithm does not apply as currently stated. It would be interesting to find a generalization of this algorithm. Note, however, that the very definition of *display* would need to be changed in order to account for this difference in leaf sets. This would be relevant to studies in *super trees* or *super networks*, which are trees and networks constructed based on several smaller input trees and networks. These input trees and networks frequently will be missing species, since studies are often carried out on relatively small sets of closely related species. If, in the end, we want to have a network that represents all living species, then we expect to see an enormous amount of input trees and networks, most of which will be missing most of the other species being considered.

Finally, the algorithm given is theoretical. A computer implementation has not been constructed. Such a construction would be desirable.

5.2.3 Chapter 4

The operations in chapter 4 are simply ones that happen to work nicely on the set of normal networks. There could be other operations that accomplish the same results, just as rSPR is merely one of several well-known tree operations. Further development of such operations might prove useful.

We also only showed that the operations work for binary normal networks. It is most likely simple to extend this to the non-binary case, though probably somewhat tedious.

We also discussed counting normal networks. Even counting the number of binary networks is somewhat difficult. Better counting methods would certainly be interesting. Simply counting the number of normal networks might be interesting, even if the method is computationally difficult.

BIBLIOGRAPHY

[1] Baroni, M.; Semple, C.; Steel, M. (2004). A framework for representing reticulate evolution. *Annals of Combinatorics*, *8* (4), 391–408.

[2] Bordewich, M.; Semple, C. (2004). On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combinatorics*, *8* (4), 409–423.

[3] Cardona, G.; Rosselló, F.; Valient, G. (2008). Comparison of tree-child phylogenetic networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, *6*(4), 552–569.

[4] Degnan, J.; Rosenberg, N. (2006). Discordance of species trees with their most likely gene trees. *PLoS Genetics*, *5* e68.

[5] Gusfield, D. (2005). Optimal, efficient reconstruction of root-unkonw phylogenetic networks with constrained and structured recombination. *Journal of Computer and System Sciences*, *7*(3), 381–398.

[6] Huson, D.; Rupp, R.; Berry, V.; Gambette, P.; Paul, C. (2009). Computing galled networks from real data. *Bioinformatics*, *25* i85–i93.

[7] van Iersel, L.; Keijsper, J.; Kelk, S.; Stougie, L. (2008). Constructing level-2 phylogenetic networks from triplets. *Research in Computational Molecular Biology*, 450–462.

[8] van Iersel, L.; Semple, C.; Steel, M. (2010). Locating a tree in a phylogenetic network. *To appear in Information Processing and Letters*, arXiv:1006.3122v1.

[9] Kanj, I; Nakleh, L.; Than, C.; Xia, G. (2008). Seeing the trees and their branches in the network is hard. *Theoretical Computer Science*, *401* (1-3), 153–164.

[10] Maddison, W. (1997). Gene trees in species trees. *Systematic Biology*, *46*(3), 523–526.

[11] Mallet, J.; Beltrán, M.; Neukirchen, W.; Linares, M. (2007). Natural hybridization in heliconiine butterflies. *BMC Evolutionary Biology*, *7*(28).

[12] Page, R.; Charleston, M. (1997). From gene to organismal phylogeny: Reconciled trees and the gene tree/species tree problem. *Molecular Phylogenetics and Evolution*, *7*(2), 231–240.

[13] Robinson, D. (1971). Comparison of labeled trees with valency three. *Journal of Combinatorial Theory, Series B*, *11*, 105–119.

[14] Rokas, A.; Williams, B.; King, N.; Carroll, S. (2003). Genome-scale approaches to resolving incongruence in molecular phylogenies. *Nature*, *425*, 798–804.

[15] Rosenberg, N.; Tao, R. (2008). Discordance of species trees with their most likely gene trees: The case of five taxa. *Systematic Biology*, *57*(1), 131–140.

[16] Semple, C.; Steel, M. (2003). Phylogenetics. *Oxford Lecture Series in Mathematics and its Applications, 24*, *Oxford University Press, Oxford*.

[17] Steel, M.; Dress, A.; Böcker, S. (2000). Simple but fundamental limitations on supertree and consensus tree methods. *Systematic Biology*, *49*(2), 363–368.

[18] Willson, S. (2007). Properties of normal phylogenetic networks. *Bulletin of Mathematical Biology*, *72*, 340–358.

[19] Willson, S. (2007). Reconstruction of some hybrid phylogenetic networks with homoplasies from distances. *Bulletin of Mathematical Biology*, *69*(8), 2561–2590.

[20] Willson, S. (2007). Unique determination of some homoplasies at hybridization events. *Bulletin of Mathematical Biology*, *69* (5), 1709–1725.

[21] Willson, S. (2008). Reconstruction of certain phylogenetic networks from the genomes at their leaves. *Journal of Theoretical Biology*, *252*, 338–349.

[22] Willson, S. (2011). Regular networks can be uniquely determined from their trees. *IEEE/ACM Transactions on Computational Biology and Informatics*, *8*(3), 785–796.