Reconstructing Evolution of Natural Languages: Complexity and Parameterized Algorithms

Iyad A. Kanj* Luay Nakhleh† Ge Xia‡

Abstract

In a recent article, Nakhleh, Ringe and Warnow introduced *perfect phylogenetic networks*—a model of language evolution where languages do not evolve via clean speciation—and formulated a set of problems for their accurate reconstruction. Their new methodology assumes *networks*, rather than *trees*, as the correct model to capture the evolutionary history of natural languages. They proved the NP-hardness of the problem of testing whether a network is a perfect phylogenetic one for characters exhibiting at least three states, leaving open the case of binary characters, and gave a straightforward brute-force parameterized algorithm for the problem of running time $O(3^k n)$, where k is the number of bidirectional edges in the network and n is its size. In this paper, we first establish the NP-hardness of the binary case of the problem. Then we provide a more efficient parameterized algorithm for this case running in time $O(2^k n^2)$. The presented algorithm is very simple, and utilizes some structural results and elegant operations developed in this paper that can be useful on their own in the design of heuristic algorithms for the problem. The analysis phase of the algorithm is very elegant using amortized techniques to show that the upper bound on the running time of the algorithm is much tighter than the upper bound obtained under a conservative worst-case scenario assumption. Our results bear significant impact on reconstructing evolutionary histories of languages–particularly from phonological and morphological character data, most of which exhibit at most two states (i.e., are binary), as well as on the design and analysis of parameterized algorithms.

1 Introduction

Languages differentiate and divide into new languages via a process similar to how biological species divide into new species: communities separate (typically geographically), the language changes differently in each of the new communities, and in time people from separate communities can no longer understand each other. While this is not the only means by which languages change, it is this process which is referred to when we say, for example, "French and Italian are both descendants of Latin." The evolution of related languages is mathematically modeled as a rooted tree in which internal nodes represent the ancestral languages and the leaves represent the extant languages.

Reconstructing this process for various language families is a major endeavor within historical linguistics, but is also of interest to archaeologists, human geneticists, and physical anthropologists, for example, because an accurate reconstruction of how certain languages evolved can help answer questions about human migrations, the time that certain artifacts were developed, when ancient

*The corresponding author. School of Computer Science, Telecommunications, and Information Systems, DePaul University, 243 S. Wabash Avenue, Chicago, IL 60604-2301. Email: ikanj@cs.depaul.edu. This research was supported in part by DePaul University Competitive Research Grant.

†Department of Computer Science, Rice University, 6100 Main St., MS 132 Houston, TX 77005-1892. Email: nakhleh@cs.rice.edu.

‡Department of Computer Science, Acopian Engineering Center, Lafayette College, Easton PA 18042, USA. Email: gexia@cs.lafayette.edu.

people began to use horses in agriculture, the identity of physically European mummies found in China, etc. (see in particular [9, 10, 16, 22]). Various researchers [2, 3, 6, 18] have noted that if communities are sufficiently separated after they diverge, then the inference of the phylogeny (i.e., evolutionary tree) for the languages can be inferred by comparing the characteristics of the languages (grammatical features, regular sound changes, and cognate classes for different basic meanings), and searching for "perfect phylogenies." However, the problem of determining if a perfect phylogeny exists, and then computing it, is NP-hard [1]. Consequently, efficient techniques for the inference of evolutionary trees for language families were not easily obtained. In the 1990's, various fixed-parameter approaches for the perfect phylogeny problem were developed (although inspired by the biological context rather than the linguistic one; see [5]). Subsequently, Ringe and Warnow worked together to fully develop the methodology (character encoding and algorithmic techniques) needed to apply these algorithms to the Indo-European language family.

However, while the methodology seemed very clearly heading in the right direction, and even seemed to potentially answer many of the controversial problems in Indo-European evolution (see [13, 15, 18, 19, 20, 21]), it became necessary to extend the model to address the problem of how characters evolve when the language communities remain in significant contact. To address this issue, Nakhleh *et al.* introduced the *perfect phylogenetic networks* (PPN) model in which languages do not evolve via a clean speciation process [11, 12]. They proved the NP-hardness of the problem of testing whether a network is a perfect phylogenetic one for characters exhibiting at least three states, leaving open the case of binary characters, and gave a straightforward $O(3^k n)$ time parameterized algorithm for the problem [11], where k is the number of bidirectional edges in the network and n is its size.

In this paper we consider the binary case of the problem. This case is of prime interest on its own since it models the problem of reconstructing evolutionary histories of languages, particularly from phonological and morphological character data, most of which exhibit at most two states [7, 14, 15, 18, 20, 21]. We first prove the NP-hardness of this problem. Then we present a branch-and-bound parameterized algorithm that solves the problem in $O(2^k n^2)$ time. The algorithm employs several interesting structural (network) operations that are very useful in the design of heuristic algorithms for the problem. When analyzed using the standard methods for analyzing parameterized branch-and-bound algorithms, and which usually work under a worst-case scenario assumption, the upper bound obtained on the size of the search tree of the algorithm is $O(3^k)$, matching the upper bound of the trivial brute-force algorithm. This worst-case analysis for a branch-and-search process is usually very conservative— the worst cases can appear very rarely in the entire process, while most other cases permit much better branching and reductions. Instead, we use amortized analysis to show that "expensive" operations can be balanced by efficient ones, and that the actual size of the search tree can be upper bounded by $O(2^k)$. The running time of the algorithm becomes $O(2^k n^2)$. The analysis phase of the algorithm is very elegant illustrating that parameterized algorithms perform much better than their claimed upper bounds, and suggesting that the standard approaches used in analyzing the size of the search tree for parameterized algorithms are very conservative.

2 Inferring evolutionary trees

An evolutionary tree, or phylogeny, for a set L of taxa (i.e., species or languages) describes the evolution of the taxa in L from their most recent common ancestor. Each taxa in L corresponds to a leaf in the evolution tree. The Different types of data can be used as input to methods of tree reconstruction; "qualitative character" data, which reflect specific observable discrete characteristics of the taxa under study, are one such type of data. There are several ways of describing qualitative characters: as partitions of the set of taxa into equivalence classes, or as functions that map the taxa to the distinct states. Qualitative characters for languages are grammatical features, unusual sound changes, and cognate classes for different meanings. The assumption of the historical linguistic methodology is that these qualitative characters evolve in such a way that there is no back-mutation (when characters exhibit parallel evolution we can find most of it and exclude those characters). What this means is that when the state of the qualitative character changes in the evolutionary history of the set of languages, it changes to a state which does not exist anywhere else on earth at that time, nor has it appeared earlier. We now formalize this concept mathematically.

Suppose that T is a rooted tree describing the evolution of a set L of languages. Therefore the leaves in T are the languages in L. Suppose that a qualitative character α is defined for each of the languages in L as a function $\alpha : L \to Z$, where Z denotes the set of integers (i.e. each integer represents a possible state for α). That is, α is a labeling to the leaves in T. We say a qualitative character α is compatible (or "convex") on T if we can extend α to every internal node of the tree T, thus defining a qualitative character α', or a labeling to the internal nodes of T, so that for every state, the nodes in T having that specific state induce a connected subgraph of T. (In other words, $\forall z \in Z$, the set of nodes $\{v \in V(T) : \alpha'(v) = z\}$ induces a connected subgraph of T.)

A different way of casting the above problem which is more intuitive is the following. Given a rooted tree T whose leaves are labeled with integers, decide if the internal nodes in T can be labeled so that each set of nodes in T with the same label induces a connected subgraph of T.

Ringe and Warnow postulated that *all* properly encoded qualitative characters for the Indo-European data should be compatible on the true tree, if such a tree existed. Such a tree is called a *perfect phylogeny*. We have the following definition and theorem.

Definition Let C be a set of qualitative characters defined on a set L of languages. A tree T is a **perfect phylogeny** for C and L if every qualitative character in C is compatible on T.

Theorem 2.1 *Let T be a phylogenetic tree on a set L of n languages, and assume that each language in L is assigned a state for α. Then we can test the compatibility of α on T in $O(n)$ time.*

PROOF. Assume the states of α on the set L are $1, 2, \ldots, r$, for some integer r. We preprocess the input in order to compute the vector $c[1...r]$ defined by $c_i = |\{s \in L : \alpha(s) = i\}|$. Obviously we can compute this vector in $O(n)$ time.

Now, for each $i, 1 \leq i \leq r$, and each node v in the tree T, we define $B_i(v)$ to be

$$B_i(v) = \{x : \ x \text{ is a leaf of } T \text{ below } v \text{ and } \alpha(x) = i\}.$$

Note that if v is a node in T then $0 < |B_i(v)| < c[i]$ implies that in any labeling of the node v for which α is compatible, we must have $\alpha(v) = i$. Hence at each node v there is at most one state i satisfying this condition.

At each node v we will therefore compute the set $States(v)$ defined to be those state(s) i such that $0 < |B_i(v)| < c[i]$, as well as the value $|B_i(v)|$ for every $i \in States(v)$. If for any node v we have $|States(v)| > 1$, then we return "Incompatible", and exit; else, we return "compatible".

We now show how to compute this information. We do this from the bottom up, and it is trivial to compute these values for the leaves. So suppose v is a node in T and we have computed these values at its children, which are $v_1, \cdots, v_l$. Note that the only candidates for elements of $States(v)$

must be drawn from $States(v_1) \cup \cdots \cup States(v_l)$. For each $i \in States(v_1) \cup \cdots \cup States(v_l)$, we set $|B_i(v)| = |B_i(v_1)| + \cdots + |B_i(v_l)|$, and then compare this to $c[i]$ to see if we include i in $States(v)$. Since $|States(v_1) \cup \cdots \cup States(v_l)| \leq l$, we can therefore compute $States(v)$ in $O(l)$ time, where l is the number of children of v. Hence, we can determine the compatibility of α on T in linear time in the number of nodes in T, that is, in $O(n)$ time. □

The initial analysis of the Indo-European data done by Warnow and Ringe in [20] demonstrated that the IE linguistic data is, nevertheless, "almost perfect": they found a tree on which the proportion of compatible characters to incompatible characters was enormous. (Even this was quite surprising; the existence of a tree on which a large proportion of characters is compatible is extremely unlikely in biological data analysis.) This suggested that the basic approach was a good one, but that the model had to be extended.

Largely the problem seemed to be the Germanic subfamily, which seemed to have remained in contact with other languages so that a tree was an inappropriate model of evolution. That is, the IE family must have evolved other than through clean speciation. When the group of languages contains some pairs of related dialects which have evolved in close contact with each other, the ability of the linguist to detect borrowing is greatly reduced. More precisely, whereas borrowing between clearly different speech forms is tightly constrained and clearly different from change in normal genetic descent, borrowing between closely related dialects is largely unconstrained and often indistinguishable from changes which could in principle be of very different types [8, 14, 17]. In this case, a tree model is inappropriate, and the evolutionary process is better represented as a "network" [11].

3 Phylogenetic networks compatibility: preliminaries and complexity

Our model of how languages evolve on networks references an underlying rooted tree (modeling "genetic descent") to which we then add bidirectional edges (modeling how linguistic characters can be transmitted through contact). Therefore, the underlying tree is rooted, and the edges of that tree can be naturally oriented from parent to child, whereas the additional edges are by design bidirectional, since contact between language communities can result in the flow of linguistic characters in both directions. This model was formalized in [11] as follows.

Definition A *phylogenetic network* on a set L of languages is a rooted directed graph $N = (V, E)$ with the following properties:

(i) $V = L \cup I$, where I denotes added nodes which represent ancestral languages, and L denotes the set of leaves of T.

(ii) E can be partitioned between the edges of a tree $T = (V, E_T)$, and the set of "non-tree" edges or bidirectional edges $E' = E - E_T$. For more convenience in the notation, we will refer to a bidirectional edge by a *b-edge*. The edges in T are oriented from parent to child, and hence T is a directed rooted tree.

(iii) N is "weakly acyclic", i.e., if N contains directed cycles, then those cycles contain *only* edges from E'.

(iv) Every internal node in N has at least two children in T.

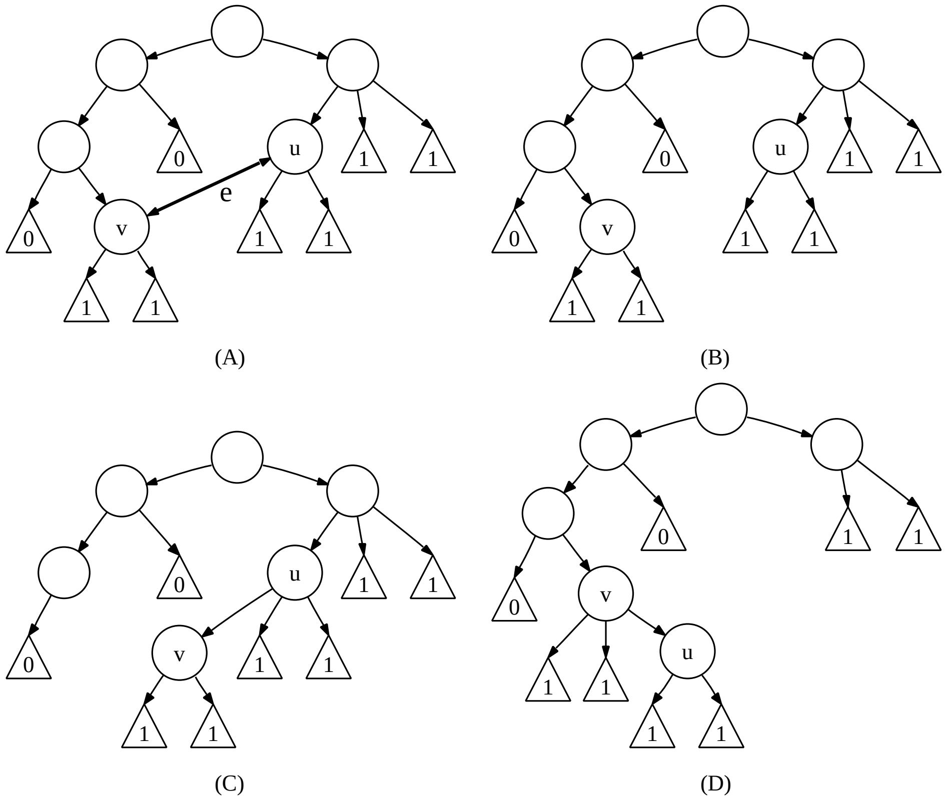

Figure 1: An example of a phylogenetic network with a b-edge e and the networks resulting from the possible assignments to e.

Properties (iii) and (iv) above will be referred to as the *phylogenetic networks properties*.

For a phylogenetic network N, we denote by T_N the underlying tree of N. For a node $u \in N$, we denote by $label(u)$ the label of node u, and by $\pi(u)$ the parent of u in T_N. If e is a b-edge between two nodes u and v in the network N, then e has three possible statuses: (1) the edge e can be simply removed denoting that no transfer took place between the two ancestral languages representing u and v, (2) e can be directed from u towards v denoting that the transfer was from the ancestral language representing u to that representing v, or (3) e can be directed from v towards u denoting that the transfer was from the ancestral language representing v to that representing u. If e is directed from u towards v, then the network is transformed as follows. Remove the edge $(\pi(v), v)$ from N, and make u the new parent of v in the resulting network (that is, add the edge (u, v) as a tree edge to the resulting network). Similarly, if e is directed from v towards u, then the edge $(\pi(u), u)$ is removed from N, and the edge (v, u) is added. Note that if there are t b-edges in N, then the t b-edges induce $O(3^t)$ trees based on 3^t different statuses of the t edges. We denote by Γ the set of the trees induced by the t b-edges in N. Figure 1–(A) shows an example of a phylogenetic network with a single b-edge $e = (u, v)$. Figure 1–(B) shows the resulting network when the b-edge e is removed, Figure 1–(C) shows the resulting network when e is directed into v, and Figure 1–(D) shows the resulting network when e is directed into u.

An assignment to the statuses of the edges in a network N whose leaves are labeled by a character is said to be *successful* if the character is compatible with the tree induced by this assignment. A *successful labeling* for a compatible tree is a labeling to the nodes of T in which all the nodes with the same label induce a connected subgraph of T. Two assignments A and A' *agree* on a set of b-edges S in N if they assign the same status to each b-edge in the set S. Note that the order in which the b-edges that are incident on a certain node are assigned can potentially make a difference in the resulting tree. So when we say that two assignments agree on a set of b-edges we implicitly mean that they also agree on the order in which these edges were assigned. The order of the assignment will not be an issue for us because, as it will be shown in Fact 5.3, every assignment to the b-edges in N has an equivalent one that, for any node in N, it directs at most one b-edge into that node, and hence is not ambiguous.

Definition Let $N = (V, E)$ be a phylogenetic network on L and Γ be the set of trees induced by all the assignments to the b-edges in N. Let C be a set of characters defined on L, and let $c : L \rightarrow Z$ be a character in C. Then c is said to be *compatible* on N if c is compatible on at least one of the trees in Γ. N is called a *Perfect Phylogenetic Network* if all characters in C are compatible on N.

The Character Compatibility on Phylogenetic Networks problem, denoted henceforth by CCPN, was defined as follows [11].

CCPN
Given a phylogenetic network $N = (V, E)$ on a set L, and a set of characters C defined on L, decide if N is a perfect phylogenetic network.

This problem was shown to be NP-hard [11] for the case where each character has *at least* three states. We will consider the case of the CCPN problem in which each character has exactly two states. We will call this problem the Binary Character Compatibility on Phylogenetic Networks, denoted henceforth by BCCPN. This problem is of prime interest on its own in the field of linguistics as was mentioned before (see [7, 14, 15, 18, 20, 21]).

BCCPN
Given a phylogenetic network $N = (V, E)$ on a set L, and a set of characters C defined on L such that each character in C has two states (i.e., binary) decide if N is a perfect phylogenetic network.

Remark 3.1 *Deciding if a network N is perfect phylogenetic on a set of characters C reduces to deciding if every character $c \in C$ is compatible on N. Therefore, without loss of generality, we will denote by* BCCPN *the problem of deciding whether a given binary character c is compatible on N. The mentioning of c becomes irrelevant in this case, and we will simply say N is compatible to denote that the implicit (given) character c is compatible on N.*

Going back to the phylogenetic network given in Figure 1, where the status of the character c on every leaf is indicated by the label on the leaf, this network is compatible because if we direct the b-edge e into v, we get the tree in Figure 1–(C) on which the character c is compatible. Note that the character c is not compatible on the tree given in Figure 1–(B) resulting from removing the b-edge e in the original network, nor on the tree Given in Figure 1–(D) resulting from directing e into v in the original network.

In the next section we study the complexity of the BCCPN problem.

4 On the complexity of BCCPN

In this section we show that the BCCPN problem is NP-complete. This will imply that the CCPN problem is NP-complete as well by specialization, giving an alternative proof to that in [11] for the NP-completeness of the CCPN problem.

Theorem 4.1 BCCPN *is NP-complete.*

PROOF. It is easy to see that BCCPN is in NP: a polynomial time nondeterministic Turing machine can nondeterministically "guess" the status of every b-edge in the network, and verifies the compatibility of the resulting tree in polynomial time.

We show that BCCPN is NP-hard by providing a polynomial time reduction from the 3-SAT problem to BCCPN with a set of characters consisting only of a single character. Recall that the 3-SAT problem is: given a boolean formula F in the conjunctive normal form (CNF) in which each clause contains exactly three literals, decide if F is satisfiable.

Let F be an instance of 3-SAT on n variables $\{x_1, \cdots, x_n\}$. Suppose that $F = C_1 \wedge \cdots \wedge C_m$, where $C_i = (l_i^1 \vee l_i^2 \vee l_i^3)$, for $i = 1, \cdots, n$. We describe next how to construct the corresponding phylogenetic network N.

The construction of N proceeds in three stages. We first construct the *variable gadgets*, then we construct the *clause gadgets*, and finally we construct the *partition gadget*.

The variable gadgets

For every variable x_i in F, we construct the following subnetwork. Associate two nodes x_i and $\bar{x}_i$ in N. (We use the same name for the literal and its corresponding node.) Node x_i has two children a and b, $\bar{x}_i$ has two children c and d, with two b-edges linking a and c, and b and d. Finally a has two leaves labeled 0, b has two leaves labeled 1, c has two leaves labeled 0, and d has two leaves labeled 1. See Figure 2 for an illustration of the V-Gadget for variable x_i.

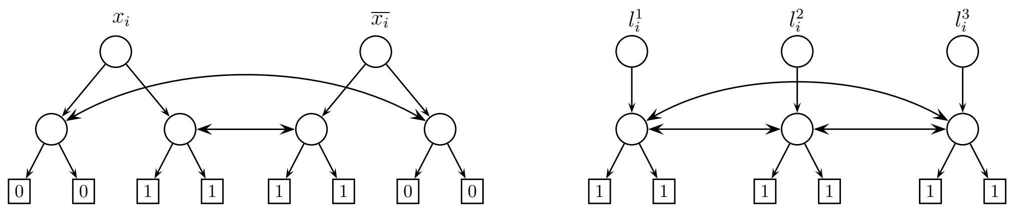

Figure 2: The V-Gadget (left) and the C-Gadget (right).

We refer to this subnetwork by *V-Gadget(x_i)*. Note that in any labeling to the nodes in V-Gadget(x_i) that makes the gadget compatible, a and c must be labeled 0, and b and d must be labeled 1. If τ is a valid truth assignment to the variables in F, then τ assigns each variable and its negation opposite truth values. This truth assignment induces a labeling on the nodes x_i and $\bar{x}_i$ in N that makes the subnetwork of N V-Gadget(x_i) compatible. For instance, if x_i is assigned truth value 1 by τ, then $\bar{x}_i$ is assigned 0. If we label the node x_i in N 0, and the node $\bar{x}_i$ 1, direct the b-edge between a and c from c towards a, and that between b and d from b towards d, then the subtree induced by this assignment is compatible. The case is similar if x_i is assigned 0 and $\bar{x}_i$ 1. Conversely, if there is a labeling to the nodes x_i and $\bar{x}_i$ in the network V-Gadget(x_i) that induces a compatible subtree, then it can be readily seen from the construction of V-Gadget(x_i) that this assignment must assign the nodes x_i and $\bar{x}_i$ opposite labels. This shows that a truth assignment τ to F is valid if and only if each variable gadget in N is compatible. Note also that the subnetwork V-Gadget(x_i) is weakly acyclic not containing any cycles with a tree edge.

The clause gadgets

For every clause $C_i = (l_i^1 \vee l_i^2 \vee l_i^3)$ in F, we construct the following subnetwork. Note that each of the literals in C_i appears in some V-Gadget. As a matter of fact, each literal in C_i appears as a node in exactly one V-Gadget. Construct three nodes a_i^1, a_i^2, and a_i^3, each with two leaves labeled 1, and add the b-edges (a_i^1, a_i^2), (a_i^2, a_i^3), and (a_i^1, a_i^3). (Note that it is not necessary to add two leaves to each of the three nodes. This is a technicality imposed by the constraint stating that each internal node in a phylogenetic network must have at least two children in the underlying tree.) Now add tree edges from the node in the V-Gadget corresponding to the literal l_i^1 to a_i^1, from the node in the V-Gadget corresponding to l_i^2 to a_i^2, and from the node in the V-Gadget corresponding to l_i^3 to a_i^3. This completes the construction of the subnetwork corresponding to the clause C_i. We will refer to this subnetwork by *C-Gadget(C_i)*. See Figure 2 for an illustration of the C-Gadget for clause $C_i = (l_i^1 \vee l_i^2 \vee l_i^3)$.

Note that each of the C-Gadget corresponding to a clause is linked to the V-Gadgets corresponding to the variables that appear in the clause. It is easy to see that each C-Gadget is weakly acyclic, and the whole subnetwork determined by the C-Gadgets and the V-Gadgets is weakly acyclic as well. Since any truth assignment τ to F that satisfies F must satisfy each clause C_i in F, for every i, there exists a literal l_i^j in C_i ($j \in \{1, 2, 3\}$), such that l_i^j is assigned 1 by τ. Without loss of generality, assume this literal is l_i^1. Now the node corresponding to the literal l_i^1 in the V-Gadget containing l_i^1 will be labeled 1, and the subnetwork determined by C-Gadget(C_i) is compatible. This can be seen by directing the b-edge between a_i^1 and a_i^2 from a_i^1 towards a_i^2, and between a_i^1 and a_i^3 from a_i^1 towards a_i^3, and finally removing the b-edge between a_i^2 and a_i^3. On the other hand, if the subnetwork determined by C-Gadget(C_i) is compatible, then at least one of the three nodes corresponding to the literals l_i^1, l_i^2, and l_i^3 must be labeled 1, and hence the corresponding literal is labeled 1 satisfying clause C_i. This shows that clause C_i in F is satisfiable by a valid truth assignment if and only if C-Gadget(C_i) is compatible.

So far, the subnetwork constructed above and determined by the V-Gadgets and the C-Gadgets ensures the following: The nodes in this subnetwork can be labeled, and the b-edges can be assigned, so that all the nodes in each resulting subtree rooted at a node corresponding to a literal have the same labels if and only if there exists a valid truth assignment to F that satisfies F (consequently, if and only if F is satisfiable). This captures the core of the reduction from 3-SAT to BCCPN. What remain are only some technicalities to complete the construction of the underlying rooted tree of N, and ensure that, upon an assignment to the b-edges of N in the V-Gadgets and C-Gadgets, the remaining b-edges in the network N guarantee that N can be partitioned so that all the nodes labeled 0 form a connected subtree, and all the nodes labeled 1 for a connected subtree, in the resulting tree induced by the assignment to the b-edges in N.

The partition gadget

The partition gadget is constructed as follows. Add a node r as the root of T_N, and add two children r_0 and r_1 of r. Add two leaves labeled 0 with parent r_0, and one leaf labeled 1 with parent r_1. Add tree edges from r_1 to every node in a V-Gadget corresponding to a literal (i.e, to every node of the form x_i or $\bar{x}_i$ in a V-Gadget), thus making r_1 the parent of all these nodes. Add b-edges from r_0 to every node in a V-Gadget corresponding to a literal. This completes the construction of N. See Figure 3 for an illustration of the partition gadget and the whole network.

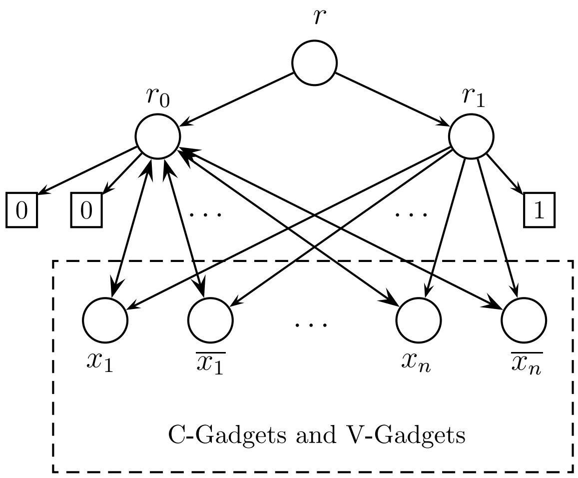

Figure 3: The partition gadget and the whole network.

It is not difficult to verify that N as constructed above is a phylogenetic network. In particular, the underlying structure of N (i.e., if we remove the b-edges from N) is a tree rooted at r, each internal node in N has at least two children, and N is weakly acyclic since the partition gadget does not create any cycles containing tree edges.

The above construction gives a polynomial-time reduction that takes an instance F of 3-SAT and produces an instance N of BCCPN.

If F is satisfiable, then there exists a valid truth assignment τ to the variables in F that satisfies every clause in F. Since τ is valid, we can label the nodes corresponding to the literals in F in every V-Gadget by the truth values assigned by τ to their corresponding literals, and direct the b-edges as described above so that to make each V-Gadget compatible. This also makes all the nodes of label 1, and label 0 in each V-Gadget, form connected subtrees within each gadget satisfying that all the nodes in each subtree have the same label. Since τ satisfies every clause in F, by the above discussion, we can direct the b-edges in every C-Gadget so that to make the C-Gadget compatible. After this step, all the nodes in any subtree rooted at a node corresponding to a literal have the same label. Now to show that N is compatible, we need to show how the remaining b-edges in N can be assigned so that the nodes labeled 0 form a connected subtree and the nodes labeled 1 form a connected subtree. At this point each of the nodes except r, r_0, and r_1 has a label. In particular, the nodes corresponding to the literals in the V-Gadgets are labeled. We label r with 1, r_0 with 0, and r_1 with 1 (note that r_0 and r_1 are forced to be labeled as such), and direct all the b-edges between r_0 and the nodes with label 0 in the V-Gadgets corresponding to literals, from r_0 towards these nodes. Note that by doing this, we are cutting off all the edges between these nodes and their parent r_1. This operation makes all the nodes of label 0 connected, and all those labeled 1 connected, which in turn, makes all the nodes of label 0 in N form a connected subtree, and so do the nodes of label 1.

Conversely, suppose that N is compatible. Then each V-Gadget is compatible, and by the above discussion, in every V-Gadget, the node corresponding to the variable and the node corresponding to the negation of this variable are assigned different labels. Now if we assign the corresponding variables the truth values determined by the labels of these nodes in the V-Gadgets we get a valid truth assignment τ for F. Since each C-Gadget is compatible, by the above discussion, the clause corresponding to the gadget must be satisfiable. This shows that F is satisfiable.

It follows that 3-SAT is reducible to BCCPN in polynomial time. Consequently, BCCPN is NP-complete. This completes the proof. □

5 A parameterized algorithm for BCCPN

A parameterized problem is a set of pairs of the form (x, k) where x is the input instance and k is a positive integer called the *parameter*. A parameterized problem is said to be *fixed-parameter tractable*, if the problem can be solved in time $f(k)|x|^c$, where f is a computable function of the parameter k, $|x|$ is the input size, and c is a constant independent of k [4]. The area of parameterized algorithms and complexity was introduced mainly in the work of Downey and Fellows [4], and is based on the core observation that for many practical occurrences of intractable problems some parameters remain small, even if the problem instances are large. Therefore, if we have an algorithm for a problem which runs in time $f(k)|x|^c$ for some fixed c, then the exponential growth in the running time no longer depends on the input size, but just the parameter (via the function $f(k)$). If we assume that k is fixed (or small), we have a polynomial time solution whose exponent does not depend on k.

Taking the advantage of the fact the the number of b-edges in the phylogenetic network is small [12], the BCCPN problem can be naturally parameterized by the number of b-edges, k, in the phylogenetic network. We call this problem the PARAMETERIZED BCCPN problem. It is easy to see that the PARAMETERIZED BCCPN problem can be solved in $O(3^k n)$ time, where n is the number of nodes in the phylogenetic network, by enumerating the status of every b-edge in the network, then checking whether the resulting induced tree is compatible. We will significantly improve on this upper bound next. The algorithm we present is a decision algorithm deciding if the network is compatible or not. The algorithm can be easily modified so that, during this processes, the status of every edge is kept track of and returned as a witness to the solution when the decision is positive. We present the algorithm and prove its correctness in this section and we analyze its running time in the next section. We start by presenting some definitions, facts, and operations.

Assumption 5.1 *Let (N, k) be an instance of* PARAMETERIZED BCCPN. *If there is at most one leaf in N of label 0 (resp. 1), then N is compatible. This is true since if we label all the internal nodes in N with 1 (resp. 0), then every assignment to the b-edges in N is a successful assignment. Since these particular cases can be identified and resolved in $O(n)$ time, we will assume henceforth that at any stage of the algorithm, there are at least two leaves of label 0 and at least two leaves of label 1 in the network.*

Definition Let N be a phylogenetic network. An internal node s in N is said to be a *splitting node* if there exists a successful assignment to the b-edges in N that results in a compatible tree T, such that there is a valid labeling for the nodes in T with all the nodes in the subtree rooted at s labeled with the same label, and all the other nodes in the tree labeled with the other (different) label.

Definition Let N be a phylogenetic network and suppose that s is a splitting node in N. Let A be a successful assignment to the b-edges in N, and let T be the tree induced by A. The assignment A is said to *respect* the splitting node s, if there is a valid labeling for the nodes in T with all the nodes in the subtree rooted at s labeled with the same label, and all the other nodes in the tree labeled with the other (different) label.

Remark 5.2 *Observe that, if we assume the statements in Assumption 5.1, then for any compatible phylogenetic network N there is at least one splitting node in N.*

Fact 5.3 *If A' is an assignment to the b-edges in N, then there exists an equivalent assignment A to the b-edges in N such that for any node u in N, A directs at most one b-edge into u. In particular, if N is compatible and s is a splitting node in N, then there exists a successful assignment A to the b-edges in N that respects s, and such that for every node u in N, A directs at most one b-edge into u.*

Proof. Let A' be an assignment to the b-edges in N. Let u be a node in N. If A' directs more than one b-edge towards u, let (a, u) and (b, u) be the first two such b-edges in the order given by the assignment. Suppose, without loss of generality, that A' directs the b-edge (a, u) first. Then A' can be replaced by another assignment A'' that removes the b-edge (a, u), directs (b, u) towards u, and agrees with A' on the assignment to the other b-edges and their respective assignment order. Moreover, the resulting network structure is unaffected by this change. Applying this argument repeatedly, we end up with an assignment that directs at most one b-edge towards u. Now we apply this argument repeatedly to the nodes in the resulting network with respect to the resulting assignment. We eventually obtain an assignment A, and that for every node u in N, directs at most one b-edge into u, and that is equivalent to A'. In particular, since the resulting structure is unaffected by this change, if N is compatible and A' is a successful assignment to the b-edges in N that respects the splitting node s, then there exists a successful assignment A to the b-edges in N that respects s, and such that for every node u in N, A directs at most one b-edge into u. $\square$

Figure 4–(A) shows an example of a network with two b-edges incident on u: e_1 and e_2. Suppose that an assignment A directs both e_1 and e_2 into u with e_1 being directed first. The network resulting from directing e_1 into u is shown in Figure 4–(B). The network/tree resulting from directing e_2 into u in the network in Figure 4–(B) is shown in Figure 4–(C). The network/tree resulting from Figure 4–(A) by an assignment A' that removes e_1 and directs e_2 into u is show in Figure 4–(D). Note that the two networks/trees in Figure 4–(C) and Figure 4–(D) are equivalent, and hence these two assignments A and A' are equivalent.

Fact 5.4 *Let u and u' be two nodes in a network N such that $label(u) \neq label(u')$, and suppose that (u, u') is a b-edge in N. If N is compatible and s is a splitting node in N, then there exists a successful assignment to the b-edges in N that respects s and in which the b-edge (u, u') is removed.*

Proof. Let A be a successful assignment that respects s. By Fact 5.3, we can assume that A directs at most one b-edge towards any node in N. In particular, we can assume that at most one b-edge is directed into u, and at most one b-edge is directed into u' by A. If A removes the b-edge (u, u') then A is the desired assignment and we are done. Suppose now that A does not remove (u, u'), and hence A either directs this b-edge into u or into u'. Assume, without loss of generality, that A directs the b-edge (u, u') into u'. Since at most one edge is directed into u' by A, and since $label(u') \neq label(u)$, u' must be the splitting node s. (This is true because all the nodes labeled

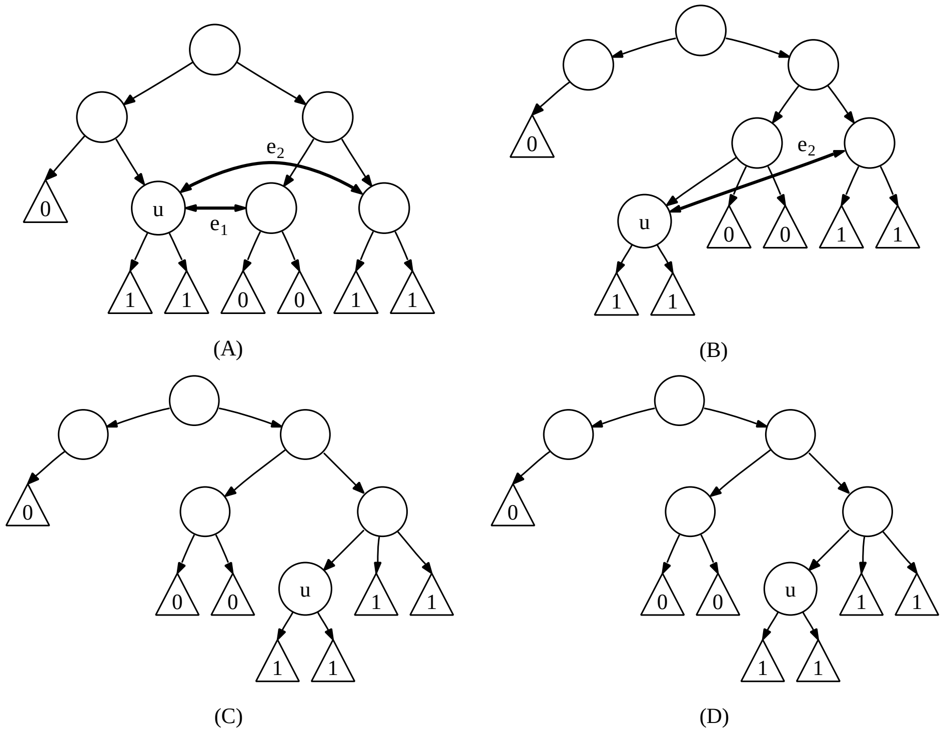

Figure 4: An illustration for Fact 5.3.

with the same label as u' have to end up being descendants of u' in the resulting tree. Notice that since (u, u') is the only b-edge directed into u', the position of u' in the tree induced by A has been fixed as a child of u which has a different label from u'.) But then if we change the status of the b-edge (u, u') in A to be removed, we still have a successful assignment that respects the splitting node $s = u'$. $\square$

Fact 5.5 *Let u and u' be two nodes in a network N such that $label(u) = label(u')$. Suppose that (u, u') is a b-edge in N. If N is compatible and s is a splitting node in N, then there exists a successful assignment to the b-edges in N that respects s and in which (u, u') is not removed, i.e., in which (u, u') is either directed towards u or towards u'.*

PROOF. Let A be a successful assignment to N that respects s. If A does not remove (u, u') then A is the desired assignment and we are done. Suppose now that A removes (u, u'). If $u = s$ (resp. $u' = s$) is the splitting node, then let A' be the assignment that agrees with A on its assignment to all the b-edges except to (u, u'), where A' directs (u, u') into u' (resp. into u). If none of u or u' is the splitting node s, then let A' be the assignment that agrees with A on its assignment to all the b-edges except to (u, u'), where A' either directs (u, u') towards u (or towards u'). It is straightforward to verify that A' is a successful assignment that respects s in each of the above cases. $\square$

Fact 5.6 *Let u and u' be two nodes in a network N such that $label(u) = label(u') = label(\pi(u)) = label(\pi(u'))$. Suppose that (u, u') is a b-edge in N. If N is compatible and s is a splitting node in N, then there exists a successful assignment to the b-edges in N that respects s and in which (u, u') is removed.*

Proof. Let A be a successful assignment to N that respects s. Observe first that since N is weakly acyclic (N satisfies the phylogenetic networks properties), none of u or u' can be a splitting node in the induced tree; otherwise, the parent of that node has to become its descendant, which is only possible if the network is not weakly acyclic. If A does not remove (u, u'), let A' be the assignment that agrees with A on its assignment to all b-edges except to (u, u'), where A' removes (u, u'). It is straightforward to verify that A' is a successful assignment that respects s. This follows from the fact that the parents of u and u' have the same labels as u and u' and they all have to end up being descendants of the same splitting node, which is different from u and u', in the induced tree. □

The main algorithm, **Phylogenetic_Compatibility**, which solves the Parameterized BC-CPN problem is given in Figure 8. The algorithm **Phylogenetic_Compatibility** tries every node in N as the splitting node. For each node selected as the splitting node, it calls the subroutine **Is_Compatible** to check whether there exists a successful assignment to N that respects the selected splitting node. Thus, the subroutine **Is_Compatible** works under the assumption that the splitting node is given. The subroutine **Is_Compatible** utilizes the subroutines **Clean**, **Reduce**, and **Merge**, given in Figure 5, Figure 6, and Figure 7, respectively. These subroutines apply some operations to reduce the network N, and also work under the assumption that the splitting node has been selected. We first prove that the modifications performed by the subroutines **Clean**, **Reduce**, and **Merge** to the network are valid in the sense that if there exists a successful assignment to N that respects the chosen splitting node before these subroutines are applied, then so is the case in the resulting network after the subroutines have been applied. We will also need to show that the phylogenetic networks properties (i.e., weak acyclicity and the property that each internal node has at least two children) are preserved after the application of these subroutines.

Proposition 5.7 *The operation performed by the subroutine* **Clean** *given in Figure 5 is valid.*

Proof. Let u and u' be two nodes in N such that $label(u) \neq label(u')$ and such that (u, u') is a b-edge. Let N' be the network resulting from applying the subroutine **Clean** to N and removing the b-edge (u, u') as described by the subroutine. First observe that when **Clean** is applied to N it removes a b-edge from N, and clearly this does not affect the phylogenetic networks properties. Therefore N' satisfies the phylogenetic networks properties. By Fact 5.4, if N is compatible and s is a splitting node in N, then there exists a successful assignment to the b-edges in N that respects s and in which the b-edge (u, u') is removed. Therefore, N is compatible and s is a splitting node in N if and only if N' is compatible and s is a splitting node in N'. This shows that the operation performed by **Clean** is a valid operation. □

Clean((u, u'))
Precondition: $label(u) \neq label(u')$ and (u, u') is a b-edge

1. remove the b-edge (u, u') from N;

Figure 5: The subroutine **Clean**.

Proposition 5.8 *The operations performed by the subroutine* **Reduce** *given in Figure 6 are valid.*

PROOF. We will show that each step in **Reduce** is valid.

If **Reduce** rejects in step 1 then obviously N is not compatible. This can be seen as follows. Suppose that step 1 in **Reduce** applies to a node u and let A be any successful assignment to the edges in N. Let T be the tree induced by the assignment A. Since T is compatible, there exists a successful labeling to the nodes of T in which all nodes with label 0 induce a connected subgraph of T, and all nodes of label 1 induce a connected subgraph of T. Suppose that u is labeled 0 in this labeling. The argument is analogous if u is labeled 1. Since u has a leaf-child of label 1, this is only possible if the leaf-child of u of label 1 is the only leaf with label 1 in N, contradicting Assumption 5.1.

Suppose that step 2 of **Reduce** applies to a node u. Note that by step 1 of **Reduce**, all the children of u must be of the same label. Note also that in any successful labeling to the nodes of an induced compatible tree of N, the label of u must be the same as the label of its children. This can be seen as follows. If u is labeled differently than its children in a compatible tree T, since all children of u are leaves and there is no b-edge incident on u, then it must be the case that u has exactly one child and its the single leaf in N with that specific label (otherwise the nodes of the same label as that leaf-child of u do not induce a connected subgraph of T), contradicting Assumption 5.1. Therefore, u can be labeled with the same label as its children.

Now if u is the splitting node and there is a leaf l' in N with the same label as u (and its children) that is not a child of u, then since all the children of u are leaves (and hence has no b-edges incident on them) and there is no b-edge incident on u, there is no assignment to the b-edges that would result in l' being a descendant of u in the induced tree, contradicting the working hypothesis that u is the splitting node. Consequently, the subroutine rejects the instance in this case. If u is the splitting node and all the leaves in N with the same label as u are children of u, then clearly if we label all other internal nodes in N with the label $(1 - label(u))$ and remove the b-edges from N, we obtain a compatible tree and the algorithm can accept the instance in this case.

Suppose now that u is not a splitting node, and let N' be the network resulting from applying step 2 in **Reduce**, which basically shrinks the subtree rooted at u to a single leaf with the same label as u. Then N' satisfies the phylogenetic networks properties since N' results from N by removing the subtree rooted at u and adding a leaf-child with the same label as u: Clearly such an operation does not affect the weak acyclicity of N, nor does it destroy the property that each internal node has at leat two children (the parent of u still satisfies this property by virtue of adding another leaf-child to it after cutting u off). Again note that by step 1 of **Reduce**, all the children of u must be of the same label and that in any successful labeling to the nodes of an induced compatible tree of N, the label of u must be the same as the label of its children. Moreover, since there is no b-edge incident on u, in any assignment A to the b-edges in N, u will remain a child of its current parent. It is straightforward to see now that a successful assignment to the b-edges in N that respects the splitting node in N is also a successful assignment to the b-edges in N' that respects the splitting node (which is still a node in N') and vice versa.

Consider step 3 of **Reduce**. Let u be an unlabeled node with a labeled child w such that there is no b-edge incident on w.

Suppose first that w is marked as the splitting node. Then the algorithm is working under the assumption that w is the splitting node in N. Since there is no b-edge incident on w, any assignment to the b-edges in N leaves w a child of u. Since w is a splitting node under the working hypothesis, and it remains a child of u in any successful assignment, the label of u and w will be different in any successful labeling of a compatible tree resulting from a successful assignment to the b-edges in N. Therefore, under the working hypothesis that w is the splitting node, the labeling operation in step 3 of **Reducing** is correct in this case, and any successful assignment to N that respects the splitting node will also be a successful assignment to the resulting network that respects the splitting node, and vice versa. Moreover, since the only change to the network performed in this case is the labeling of u, the resulting network still satisfies the phylogenetic networks properties.

Suppose now that w is not marked as the splitting node. By the same observation as above, w will always remain a child of u in any assignment to the b-edges if N. Since w is not the splitting node under the working hypothesis, and w is a child of u, the label of u must be the same as the label of w in any successful assignment to the b-edges in N. This can be seen as follows. Let T be a compatible tree and consider a successful labeling of T. If $label(w) \neq label(u)$ in this successful labeling, then since u is the parent of w in T, all nodes with the same label as w must belong to the subtree rooted at w, and hence w would be a splitting node, contradicting our working hypothesis. Therefore, under the working hypothesis that w is not the splitting node, the labeling operation in step 3 of **Reducing** is correct in this case, and any successful assignment to N that respects the splitting node will also be a successful assignment to the resulting network that respects the splitting node, and vice versa. Moreover, since the only change to the network performed in this case is the labeling of u, the resulting network still satisfies the phylogenetic networks properties.

The correctness of step 4 follows by a similar argument to that of step 3.

The validity of step 5 can be easily seen since adding more leaves to u of the same label will not affect any successful assignment nor will it affect the splitting node or destroy the phylogenetic networks properties.

If u has two leaves with a certain label, then any other leaf with the same label can be removed without affecting any successful assignment. This can be seen as follows. Since u has two leaves of the same label, any successful labeling must label u with the same label as these two leaves. Therefore, having more leaves with the same label will not affect any successful labeling. Consequently, any successful assignment to N that respects the splitting node will also be a successful assignment to the resulting network that respects the splitting node, and vice versa. Moreover, since the only change to the network performed in this case is the removal of some leaves of u, and since u in the resulting network has two leaves, the resulting network still satisfies the phylogenetic networks properties, and step 6 of **Reduce** is valid. $\square$

Proposition 5.9 *Let N be a phylogenetic network and suppose that the subroutine* **Reduce** *is not applicable to any node in N. Then the operations performed by the subroutine* **Merge** *given in Figure 7 are valid.*

PROOF. Let u and u' be two nodes in a phylogenetic network N such that $label(\pi(u)) \neq label(u) = label(u')$, and such that (u, u') is a b-edge in N. Suppose further that the operation **Reduce** is not applicable to any node in N. The subroutine **Merge** cuts u off its parent, merges the two nodes u and u' to form a new node w with $\pi(w) = \pi(u')$, and makes all the b-edges that were incident on u and u', incident on the new node w. Let N' be the network resulting from this operation.

The fact that the operation preserves the phylogenetic networks properties is not difficult to see. The operation clearly preserves the weak acyclicity of the network: if N' is not weakly acyclic then N would not be. Since the only internal node that some of its children are cut off by this operation is $\pi(u)$, and since $\pi(u)$ is labeled and step 5 of **Reduce** is not applicable to $\pi(u)$, it follows that $\pi(u)$ has two leaves of the same label that remain after the application of the operation. Therefore, N' satisfies the phylogenetic networks properties.

Reduce(u)

1. **if** u has two leaf-children with different labels **then reject**;
2. **if** all the children of u are leaves and there is no b-edge incident on u **then**
   if u is marked as the splitting node **then**
   if there is a leaf in N that is not a child of u
   and of the same label as the children of u **then reject**;
   else accept;
   else
   remove u and its children and replace them with a leaf l;
   label l with the same label as the children of u;
   add the tree edge $(\pi(u), l)$;
3. **if** u is unlabeled and has a labeled child w (w could be a leaf) with no b-edge incident on w **then**
   if w is marked as the splitting node **then** set $label(u) = 1 - label(w)$;
   else set $label(u) = label(w)$;
4. **if** u is labeled and has an unlabeled child w with no b-edge incident on w **then**
   if w is marked as the splitting node **then** set $label(w) = 1 - label(u)$;
   else set $label(w) = label(u)$;
5. **if** u is labeled and has at most one leaf-child **then**
   add two leaves as children to u of the same label as u;
6. **if** u has more than two leaves with the same label **then** remove all of them except two;

Figure 6: The subroutine **Reduce**.

Suppose now that there exists a successful assignment A to N that respects the splitting node. Since $label(u) = label(u')$, by Fact 5.5, we can assume that A either directs (u, u') towards u or towards u'. It is not difficult to verify that the assignment A' to N' that assigns to a b-edge (c, w) the same value assigned by A to its corresponding b-edge (c, u) or (c, u') in N, disregards the assignment of A to the b-edge (u, u'), and agrees with A on its assignment to all the other b-edges, is a successful assignment that respects the splitting node in N'. The converse is also true. □

Merge($\langle u, u' \rangle$)
Precondition: $label(\pi(u)) \neq label(u) = label(u')$ and (u, u') is a b-edge

1. cut off the tree edge $(\pi(u), u)$ from N;
2. remove the b-edge (u, u');
3. identify the two nodes u and u' (i.e., merge the two nodes into one new node);
4. let the new node be w; set $label(w) = label(u')$ and $\pi(w) = \pi(u')$ (add the tree edge $(\pi(u'), w)$);
5. make the children of both u and u' children of w;
6. shift all the b-edges that are incident on u and u' to make them incident on w without changing the other endpoints of the b-edges;
7. **if** u or u' is marked as the splitting node **then** mark the new node w as the splitting node;

Figure 7: The subroutine **Merge**.

Fact 5.10 *Let N be a phylogenetic network such that N is invariant under the application of the operation* **Reduce** *to every node u in N. If v is an unlabeled node in N, then there exists at least one b-edge incident on a node in the subtree of T_N rooted at v.*

PROOF. This follows from step 2 and 3 in **Reduce**, and the fact that each non-leaf node in a phylogenetic network has at least two children (and hence the subtree rooted at v has leaves). If there is no b-edge incident on any node in the subtree of T_N rooted at v, then v would be labeled by the repeated application of step 2 in **Reduce** (starting at the leaf-nodes in the subtree rooted at v and going bottom-up to v), followed by the application of step 3 in **Reduce**. □

Proposition 5.11 *Let N be a phylogenetic network such that none of the operations* **Reduce**, **Clean**, *or* **Merge** *is applicable to N. Then there exist two nodes u and u' in N such that: (1) $label(u) = label(u')$, (2) (u, u') is a b-edge in N, and (3) all children of u and u' are leaves.*

PROOF. Define an internal node w in N to be a *deepest* node if all its children are leaves. Note that since step 3 of **Reduce** is not applicable to any node in N, every deepest node in N must be labeled, and by step 2 of **Reduce** and the fact that all the children of a deepest node are leaves, every deepest node must have at least one b-edge incident on it.

Let w_1 be a deepest node in N, and let (w_1, w_2) be a b-edge incident on w_1. If w_2 is a deepest node, set $u = w_1$ and $u' = w_2$. Since both w_1 and w_2 are deepest nodes, w_1 and w_2 are labeled, and hence u and u' are labeled. Moreover, $label(u) = label(u')$ since **Clean** is not applicable. Since (u, u') is a b-edge, it follows that u and u' satisfy the statement of the proposition.

Now suppose that w_2 is not a deepest node. Let T_{w_2} be the subtree of T_N rooted at w_2. Since w_2 is an internal node and N is a phylogenetic network, w_2 has descendants, and T_{w_2} contains leaves. Since step 2 of **Reduce** is not applicable, there must exist a deepest node in the subtree T_{w_2}. Let w_3 be such a deepest node, and let (w_3, w_4) be a b-edge incident on w_3. Now repeat the same process. We claim that this process must halt with the desired u and u'. This can be easily seen as follows. Suppose this process does not halt. Define the following walk W in N. Add every edge of the form (w_i, w_{i+1}) to W. Add to W every path that is traced in this process from a node w_i to a deepest descendant node in T_{w_i}. By the assumption that the process does not halt, W is an infinite walk on N. Since N has finitely many nodes, W must contain a simple closed path P. By the weak acyclicity of N, the path P cannot contain any tree edges, and hence all the edges on P are b-edges. By looking at W, one readily sees that every b-edge of the form (w_i, w_{i+1}) in W is followed by a tree path from w_{i+1} to one of its deepest descendants (unless the process halts). Since P does not contain any tree edges, P must consist of a single b-edge of the form (w_i, w_{i+1}), a contradiction. It follows that this process halts with the desired vertices u and u'. This completes the proof. □

We call a pair of nodes $\{u, u'\}$ satisfying the three conditions in Proposition 5.11 a *nice pair*. Proposition 5.11 establishes the existence of a nice pair in any phylogenetic network N to which none of the operations **Reduce**, **Clean**, or **Merge** is applicable. Now we are ready to present the main algorithm **Phylogenetic_Compatibility** which solves the PARAMETERIZED BCCPN problem. The algorithm is a branch-and-search process. Each stage of the algorithm starts with an instance (N, k) of the problem, and tries to reduce the parameter k by identifying and eliminating some b-edges. During this process, some nodes in N get labeled. Then the algorithm recursively works on the reduced instances. We implicitly assume that after each step, the network N and the parameter k are updated accordingly. Furthermore, we will assume that Assumption 5.1 is valid before each operation performed by the algorithm and its subroutines. The algorithm is given in Figure 8. Note that the subroutines **Clean**, **Reduce**, and **Merge** do not perform any branching and can be very useful in the design of heuristic algorithms for the problem. The algorithm itself performs exactly two different branches, which are the ones given in **Case 2** and **Case 3** of step 4.

Is_Compatible (N, k)

1. **if** $k = 0$ and N is not compatible **then** reject;
2. **while Reduce** is applicable to a node in N apply it;
3. **if** any of **Clean** or **Merge** is applicable **then** apply it and **go to** step 1;
4. let $\{u, u'\}$ be a nice pair in N; $\{*$ assume without loss of generality that $label(u) = label(u') = 1$ $*\}$

   Case 1. Both $\pi(u)$ and $\pi(u')$ are labeled

   remove the b-edge (u, u');

   Case 2. Exactly one of $\pi(u)$ and $\pi(u')$ is labeled, say $\pi(u)$. Branch as follows

   first side of the branch: set $label(\pi(u')) = 1$ and remove the b-edge (u, u');

   second side of the branch: set $label(\pi(u')) = 0$;

   Case 3. (Both $\pi(u)$ and $\pi(u')$ are unlabeled.) Branch as follows

   first side of the branch: set $label(\pi(u)) = 0$;

   second side of the branch: set $label(\pi(u')) = 0$;

   third side of the branch: set $label(\pi(u)) = label(\pi(u')) = 1$ and remove the b-edge (u, u');

Phylogenetic_Compatibility

Input: an instance (N, k) of Parameterized BCCPN where N is a phylogenetic network and k is a positive integer

Output: yes/no decision based on whether N is compatible or not

1. **for** every node s in N **do**
   - 1.1. $N' = N$;
   - 1.2. mark s as the splitting node in N';
   - 1.3. call **Is_Compatible** on (N', k);
   - 1.4. **if Is_Compatible** returns yes **then return** yes;
2. **return** (no);

Figure 8: The subroutine **Is_Compatible** and the algorithm **Phylogenetic_Compatibility**.

Theorem 5.12 *The algorithm* **Phylogenetic_Compatibility** *is correct.*

Proof. The algorithm **Phylogenetic_Compatibility** tries every node as the splitting node and then calls the subroutine **Is_Compatible**. If N is compatible, then there exists a successful assignment to the b-edges in N and a successful labeling to the induced compatible tree. By Remark 5.2, there exists a node in N which is a splitting node in this case. Therefore, if we show that the subroutine **Is_Compatible** which works under the assumption that the splitting node is given, is correct, then it will follow that the algorithm **Phylogenetic_Compatibility** is correct.

We look now at the subroutine **Is_Compatible**. Step 1 of the subroutine is correct because if $k = 0$ then N must be a phylogenetic tree, and the compatibility of N can be checked using Theorem 2.1. The correctness of steps 2 and 3 follows from Proposition 5.7, Proposition 5.8, and Proposition 5.9. Note that, by the way the statements in the subroutine **Is_Compatible** are ordered, when **Merge** is executed, **Reducing** is not applicable, and hence the assumptions in the statement of Proposition 5.9 hold true. Therefore, we only need to verify step 4. First we need to justify the existence of a nice pair at this point of the algorithm. By Proposition 5.11, we only need to show that the network N satisfies the phylogenetic networks properties whenever we are at step 4 of the algorithm. Note first that, by the way the algorithm is designed, when any of the operations **Clean** or **Merge** is invoked, the operation **Reduce** is not applicable. It has been shown that the operations in **Clean**, **Reduce**, and **Merge** preserve the phylogenetic networks properties. Moreover, step 4 of **Is_Compatible** preserves the phylogenetic networks properties (since it only labels the nodes and possibly removes some b-edges). It follows, by an inductive argument, that the phylogenetic networks properties are preserved each time step 4 of the algorithm is about to be executed, given that the network passed to the algorithm originally is a phylogenetic network. The correctness of the branches in step 4 can be seen as follows. First notice that each of u and u' should have a parent. Otherwise, one of the them is the root and is a deepest node. This means that all the other nodes in N including u' are leaves, a contradiction (since N could not contain any b-edges and step 1 should conclude the algorithm). Since $\{u, u'\}$ is a nice pair, both nodes u and u' are labeled. The algorithm only describes the case when both u and u' are labeled 1. The other case is exactly the same with 0s replaced by 1s and 1s by 0s in the branches. Note that none of $\pi(u)$ or $\pi(u')$ can be labeled 0, otherwise, since both u and u' are labeled 1, **Merge** would be applicable. The three cases given in step 4 clearly cover all possible scenarios since: (1) either both $\pi(u)$ and $\pi(u')$ are labeled, or (2) exactly one of them is labeled, or (3) none of them is labeled. Now we justify the correctness of the branch (if any) in each of the three cases.

In **Case 1** no branching is needed, and the correctness of this step follows from Fact 5.6. In **Case 2**, we note that since **Merge**$(\langle u, u'\rangle)$ is not applicable and label $\pi(u) = 1$, $label(\pi(u))$ must be 1. Now either $label(\pi(u')) = 1$ or $label(\pi(u')) = 0$. In the first side of the branch where we set $label(\pi(u')) = 1$, the removal of the b-edge (u, u') is again correct by Fact 5.6. In **Case 3**, we know that either one of $\pi(u), \pi(u')$ is labeled 0, or none of them is, an hence, both of them are labeled 1. In the latter case the b-edge (u, u') can be removed by Fact 5.6. Therefore the case accounts for all possible scenarios. This proves the correctness of the branch in step 4. Now how do we know that the algorithm terminates?

Observe first that each time step 4 of the algorithm is executed, at least one b-edge will be removed. This can be seen as follows. If **Case 1** is executed then the b-edge (u, u') is removed. If **Case 2** is executed, then in the first side of the branch the b-edge (u, u') is removed. In the second side of the branch, $label(\pi(u'))$ is set to 0, and when **Merge**$(\langle u', u\rangle)$ is called next, the b-edge (u, u') will be removed. If **Case 3** is executed, then in the first and second sides of the branch, the b-edge (u, u') will be removed when **Merge**$(\langle u, u'\rangle)$ or **Merge**$(\langle u', u\rangle)$ is called next. In the third side of the branch the b-edge (u, u') is removed by Fact 5.6.

Each execution of **Clean** removes at least one b-edge from N. Each execution of **Merge** removes one b-edge. An execution of **Reduce** may end up adding two leaves to an internal node, but once two leaves have been added to an internal node no more leaves will be added to this internal node. Therefore, the total number of leaves that can be added by **Reduce** is bounded by twice the number of nodes in N. Any other execution of **Reduce** either ends up labeling some nodes or removing nodes and edges from N. This proves the correctness of the whole algorithm. Therefore, if the instance has a solution then a solution will be found by the algorithm, otherwise, a negative answer will be reported by the algorithm. □

In the next section we will analyze the running time of the algorithm **Phylogenetic_Compatibility**. Since the algorithm **Phylogenetic_Compatibility** ends up calling the subroutine **Is_Compatible** $O(n)$ times, it suffices to analyze the running time of **Is_Compatible** and multiply it by $O(n)$. We will refer to the subroutine **Is_Compatible** by the algorithm **Is_Compatible** in the remainder of the paper.

6 Analysis of the algorithm Is_Compatible

In this section we analyze the running time of the algorithm **Is_Compatible**. Since the algorithm is a branch-and-bound process, its execution can be depicted by a search tree. The running time of the algorithm is proportional to the number of root-to-leaf paths, or equivalently the number of leaves in the search tree, multiplied by the time spent along each such path. Therefore, the main step in the analysis of the algorithm is deriving an upper bound on the number of leaves in the search tree.

Most proposed branch-and-search algorithms for NP-hard problems were analyzed based on a worst-case scenario, which assumes the worst local structure occurring during the whole search process. This worst-case analysis for a branch-and-search process is very conservative — the worst cases can appear very rarely in the entire process, while most other cases permit much better branching and reductions.

In the current paper we use an amortized analysis approach. This allows us to capture the following notion: an operation by itself may be very costly in terms of the size of the search tree that it corresponds to, however, this operation might be very beneficial in terms of introducing many efficient branches and reductions in the entire process. Therefore, the expensive operation can be well balanced by the induced efficient operations. We show how this technique can be applied to the Parameterized BCCPN problem. First we start with some preliminaries on search trees.

Let $\mathcal{T}$ be the search tree for the algorithm **Is_Compatible** on an input instance (N, k). The nodes in $\mathcal{T}$ correspond to the operations of the algorithm. Let α be a node in the search tree with an associated parameter k'. If we perform an r-sided branch at α ($r > 1$) by reducing the parameter k' in each branch by the values $a_1, \cdots, a_r$, respectively, then such a branch is called an $(a_1, \cdots, a_r)$-*branch*. In such case the node α in $\mathcal{T}$ has r children $\alpha_1, \cdots \alpha_r$, and the associated parameter with α_i is $k' - a_i$, $i = 1, \cdots r$. If the operation at α is a non-branching operation that reduces the parameter k' by a value q, then α has a single child in $\mathcal{T}$ with an associated parameter equals to $k' - q$. Let $T(k')$ be the number of leaves in the subtree rooted at α. If the operation at α is a branching operation $(a_1, \cdots, a_r)$, then $T(k')$ satisfies the recurrence $T(k') \leq T(k' - a_1) + \cdots + T(k' - a_r)$; if the operation at α is a non-branching operation that reduces the parameter k' by a value q, then $T(k')$ satisfies the recurrence $T(k') \leq T(k' - q)$. To solve these recurrences, we can associate with each branch $(a_1, \cdots, a_r)$ a characteristic polynomial of the form $p(x) = x^{-a_r} + x^{-a_{r-1}} + \cdots + x^{-a_1} - 1$.

The unique root x_0 of $p(x)$ in the interval $(0, \infty)$ gives an upper bound of $O(x_0^k)$ on the number of leaves in the search tree of an algorithm whose branches are all of the form $(a_1, \cdots, a_r)$. If the branches of the algorithm cannot be classified within a single form, then we can look at all the branches performed by the algorithm, and upper bound the number of leaves in $\mathcal{T}$ by $O(x_{max}^k)$, where x_{max} is the largest root among all roots of the characteristic polynomials corresponding to the branches performed by the algorithm. This is a well-known method for analyzing the size of the search tree, which has been commonly used in the literature.

In this section we will show that the number of leaves in the search tree of the algorithm **Is_Compatible** is $O(2^k)$. At this point an explanation of a subtlety is in order. This upper bound may look trivial at a first glance. By Proposition 5.11, we know that a nice pair $\{u, u'\}$ exists before each branch. By Fact 5.5, there exists a successful assignment that either directs the b-edge (u, u') towards u or towards u'. So it looks like we can always branch with a $(1, 1)$-branch resulting from directing the b-edge (u, u') towards u and reducing the parameter by 1 in the first side of the branch, and directing it towards u' and reducing the parameter by 1 in the second side of the branch. This would give us an $O(2^k)$ upper bound on the size of the search tree. However, there is a subtle point here that could be easily overlooked. When we branch along any of the two sides, say by directing the b-edge towards u', we end up cutting the node u' from its parent. This reduces the number of children of $\pi(u')$, and the resulting network may no longer satisfy the phylogenetic network property stating that each internal node has at least two children, which is very essential to proving the existence of a nice pair in the network (see Proposition 5.11). Similarly, when the b-edge is directed towards u. To overcome this problem, whenever we branch by cutting a certain node from its parent, we ensure at this point that the parent has been assigned a label, and hence, when **Reduce** is applied in the next step (before any subsequent branch takes place) the phylogenetic networks properties will be restored by step 5 of **Reduce**. Therefore, we now branch by assigning the nodes labels rather than branching at the edges. This is no longer a trivial matter, and the analysis now takes a new turn.

As we will discuss below, the branches in the algorithm can be classified into two branches: $(1, 1)$-branches and $(1, 1, 1)$-branches. The latter branch corresponds to a characteristic polynomial of root 3, and a worst-case analysis gives an $O(3^k)$ upper bound on the size of the search tree, matching the bound of a trivial brute-force algorithm that enumerates each of the three statuses of every b-edge. Differing from the common analysis techniques based on the worst-case scenario, we present next a novel way for analyzing the size of the search tree using amortized techniques. We will show that the $(1, 1, 1)$-branches give some "credit" along each path of the subtree of $\mathcal{T}$ rooted at this operation. We first classify the operations performed by the algorithm that affect the parameter k into the following three categories.

1. Non-branching operations. These include the following operations.
   (a) Operations performed by **Clean**. Each such operation removes a b-edge from N and decreases the parameter k by 1.
   (b) Operations performed by **Merge**. Each such operation removes a b-edge from N and decreases the parameter k by 1.
   (c) Operations performed in **Case 1** of the algorithm. Each such operation removes a b-edge and reduces the parameter by 1. Note also that these operations do not involve any branching.
2. $(1, 1)$-branches: these are the operations performed in **Case 2** of the algorithm. Note that each such operation is a 2-sided branch which reduces the parameter by 1 on each side.

3. $(1,1,1)$-branches: these are the operations performed in **Case 3** of the algorithm. Each such operation is a 3-sided branch that reduces the parameter by 1 along each side.

We would like to show that the number of leaves in $\mathcal{T}$ is bounded by $O(2^k)$. The $(1,1)$-branches give us this bound. However, the $(1,1,1)$-branches are worse, and give an upper bound of $O(3^k)$ on the number of leaves of $\mathcal{T}$. We will show next that the $(1,1,1)$-branches can be balanced by the non-branching operations. We start with the following definitions.

Definition A node is said to possess a *credit* of value 1/2 if it is labeled and there is a b-edge incident on one of its children. A node is said to give a credit of value 1/2 after a certain operation if the node did not possess any credit before the operation, and it possesses a credit of value 1/2 after the operation.

Ultimately, the value of a credit will correspond to a reduction in the parameter of the same value.

Fact 6.1 *Let v be an unlabeled node in a phylogenetic network N, and suppose that* **Reduce** *is not applicable to any node in N. Let T_v be the subtree of T_N rooted at v. Let $P = (v_1 = v, v_2, \cdots, v_r = l)$ be a path from v to any leaf l in T_v. Then there exists a node $v_i \neq v$ on P such that v_i has a b-edge incident on it, and all the nodes $\{v_j : 1 < j < i\}$ are unlabeled and have no b-edges incident on them. (Note that such a set of nodes might be empty and in which case the latter condition is vacuously satisfied.)*

PROOF. Let i be the smallest index in $\{2, \cdots, r-1\}$ such that v_i has a b-edge incident on it. Since v is unlabeled, and **Reduce** is not applicable to any node in N, such i must exist, otherwise v would be labeled by step 3 in **Reduce**. By the choice of i, all the nodes in $\{v_j : 1 < j < i\}$ have no b-edges incident on them. Moreover, the nodes in $\{v_j : 1 < j < i\}$ are unlabeled, otherwise, by step 3 of **Reduce**, v would be labeled since there are no b-edges incident on any of these nodes. □

Proposition 6.2 *Let N be a phylogenetic network and let v be an unlabeled node in N. Suppose that a side of a branch in the algorithm is assigning a label to v. Then there exists a node in the subtree T_v of T_N rooted at v that will give a credit of value 1/2.*

PROOF. First observe that whenever the algorithm branches, the subroutine **Reduce** is not applicable to N. If we look at a side of a branch of the algorithm that assigns a label to a node in N, then this side of the branch is assigning a label to a parent $v = \pi(u)$ of a node u in a nice pair. This side of the branch might end up cutting u from $v = \pi(u)$. Since N satisfies the phylogenetic networks properties, v must have a child w different from v. Moreover, before this branch v was unlabeled, and hence w cannot be a leaf and must be an internal node (otherwise v would be labeled by step 3 of **Reduce**). If w has a b-edge incident on it, then by the definition, v can give a credit of value 1/2. Now suppose that w does not have a b-edge incident on it. Let $P = (v = v_1, v_2 = w, \cdots, l)$ be a path from v to a leaf in T_v that passes through w. By the way the algorithm works, before this side of the branch **Reduce** is not applicable. By Fact 6.1, there is a node $v_i \neq v$ on P such that v' has a b-edge incident on it, and such that all the nodes in the set S=$\{w = v_2, \cdots, v_{i-1}\}$ between v and v_i are unlabeled and have no b-edges incident on them. Notice that the set S contains w and hence in not empty. When **Reduce** is next called (note that **Reduce** will be called repeatedly before the next branch by the algorithm) v will be labeled. Step 4 of **Reduce** will label all the nodes in the set S, and in particular node v_{i-1}. Now at that time the algorithm will assign label to node v_{i-1} which has a child v_i with a b-edge incident on it. It follows that node v_{i-1} will give a credit of value 1/2. We note that this credit is given before the next branch by the algorithm and hence can be associated with the previous (side of the) branch. This completes the proof. □

The idea of an operation giving a credit is an intuitive way of looking at the whole set of operations in the algorithm as an interleaved set in which some operations balance the others. When a node gives a credit of a certain value, this credit will correspond to a reduction in the parameter. When a node gives a credit of value 1/2, we can associate this credit with a b-edge incident on one of its children. Note that a b-edge (u, v) can have at most two credits associated with it, each of value 1/2, resulting from the possible credits given by the nodes $\pi(u)$ and $\pi(v)$. Therefore, if the b-edge is removed, its removal may cause at most two nodes to lose their possessed credits since no b-edge will be incident on a child of theirs anymore. Ultimately, the value of a credit will correspond to a reduction in the parameter of the same value. Before we show the latter statement, let us assume it for the time being and look at the operations performed by the algorithm to gain an intuition on how this method works.

Non-branching operations: A **Clean** operation removes a b-edge e between two labeled nodes u and v where $label(u) \neq label(v)$. The b-edge contributes to a reduction in the parameter of value 1. If $\pi(u)$ is labeled and $\pi(v)$ is labeled, then $\pi(u)$ and $\pi(v)$ might possess a total credit of value 1 (1/2 each), and this credit may have been associated with the b-edge e. When e is removed, e might cause these two nodes to lose their credits. Consequently, an edge removed by **Clean** can compensate for the loss of credit it incurs. Similarly for the other non-branching operations: each will result in a reduction of the parameter of value 1, which is in the worst case not smaller than the value of the possibly lost credit caused by the removal of the b-edge.

$(1, 1)$**-branches:** Suppose the algorithm executes the branch in **Case 2**. On the first side of the operation $\pi(u')$ is labeled and a b-edge $e = (u, u')$ is removed. By proposition 6.2, labeling $\pi(u')$ will give a credit of value 1/2. Since $\pi(u')$ was unlabeled before this operation, no credit was possessed by $\pi(u')$. Hence, only a credit of value 1/2 could have been associated with the edge e due to the credit of value 1/2 that could have been given by $\pi(u)$ (which is labeled). Therefore, the credit gained by labeling $\pi(u')$ can serve to pay for the credit possibly lost by the removal of e, thus canceling each other out, and the total reduction in the parameter in this side of the branch is equal to 1. The other side of the branch is similar yielding a reduction of value 1. Therefore, this branch is effectively a $(1, 1)$-branch.

$(1, 1, 1)$**-branches:** Suppose the algorithm executes **Case 3**. On the first side of the branch $\pi(u)$ is labeled with a label different from $label(u)$. When **Merge** is called next, the b-edge $e = (u, u')$ will be removed, and the two nodes u and u' will be merged. Since before this operation was executed both $\pi(u)$ and $\pi(u')$ were unlabeled, the b-edge e does not cause any credit loss. Labeling $\pi(u)$ gives a credit of value 1/2 by Proposition 6.2. Therefore, the total "effective" reduction in the parameter along this side of the branch is 3/2. Similarly, in the second side of the branch we get an effective reduction in the parameter of value 3/2. Now in the third side of the branch both $\pi(u)$ and $\pi(u')$ will be labeled with the same label and the edge e is removed. Labeling $\pi(u)$ gives a credit of value 1/2, and similarly for $\pi(u')$, and the edge e does not cause any credit loss since both nodes $\pi(u)$ and $\pi(u')$ were unlabeled before this operation, and thus could not have given any credit before. The total reduction in the parameter along this side of the branch has an effective value of 2. Therefore, the algorithm in this case effectively branches with a $(3/2, 3/2, 2)$-branch.

The worst branch in the above branches is the $(1, 1)$-branch giving an upper bound of $O(2^k)$ on the size of the search tree. That was an intuitive look at the amortized analysis of the algorithm.

We formally prove this statement below.

Lemma 6.3 *Let $\mathcal{T}$ be the search corresponding to the algorithm* **Is_Compatible** *on an instance (N, k). The number of leaves of $\mathcal{T}$ is $O(2^k)$.*

PROOF. We first prove the following statement.

Statement. Let α be a node in the search tree $\mathcal{T}$ corresponding to the algorithm **Is_Compatible** on an instance (N, k) and let $\mathcal{T}_\alpha$ be the subtree of $\mathcal{T}$ rooted at α. Let (N_α, k_α) be the resulting network at α, and assume that there are ℓ nodes in N_α that possess credit, where $0 \le \ell \le 2k_\alpha$. Then the number of leaves in $\mathcal{T}_\alpha$ is bounded by $2^{k_\alpha - \ell/2}$.

We proceed by induction on k_α. If $k_\alpha = 0$ then $\ell = 0$. There are no b-edges in the network N_α in this case, and the algorithm Is_Compatible decides the compatibility of N_α in step 0 using Theorem 2.1 and without performing any branches. Therefore the number of leaves in $\mathcal{T}_\alpha$ is 1, which is bounded by $2^{k-\ell/2}$ as claimed.

If $k_\alpha = 1$ then there is exactly one b-edge (u, v) in N_α. Since there are no b-edges in T_u and T_v in N_α, the nodes u and v must be labeled by Fact 5.10. Since N_α is a phylogenetic network, each internal node in N_α must have at least two children, and in particular, the nodes $\pi(u)$ and $\pi(v)$. By Fact 5.10, and since N_α contains only the b-edge (u, v), these two children must be labeled (the subtrees rooted at these nodes in N_α contain no b-edges). By step 3 of **Reduce**, $\pi(u)$ and $\pi(v)$ must be labeled as well. Now If $label(u) \ne label(v)$ then the b-edge (u, v) will be removed by **Clean**. If $label(u) = label(v)$ and the label of $\pi(u)$ or the label of $\pi(v)$ is different from the label of u and v, then the b-edge (u, v) will be removed by **Merge**. If the labels of u, v, $\pi(u)$, and $\pi(v)$ are all equal, then the b-edge (u, v) will be removed by **Case 1** in step 4 of the algorithm Is_Compatible. It follows that in all cases the b-edge (u, v) will be removed by the algorithm without any branching. Since (u, v) is the only b-edge in N_α, after removing this b-edge, the algorithm **Is_Compatible** will proceed to solve the resulting instance in step 0. All in all, no branches will be performed by the algorithm when solving the instance (N_α, k_α) and the number of leaves in $\mathcal{T}_\alpha$ is 1, which is bounded by $2^{k_\alpha - \ell/2}$.

Suppose now that the above statement is true for any value of k' satisfying $0 \le k' < k_\alpha$. The operation performed by the algorithm at α can be classified into one the three categories above.

If this operation is a non-branching operation that results in the removal of a b-edge e, then since α is a non-branching operation, α has a single child β in $\mathcal{T}$. Let $\mathcal{T}_\beta$ be the subtree of $\mathcal{T}$ rooted at β. The number of leaves in $\mathcal{T}_\alpha$ is equal to that in $\mathcal{T}_\beta$. By the above discussion, the removal of e will decrease the parameter k_α by at least 1, and causes at most two nodes in N_α to lose their possessed credit. Let (N_β, k_β) be the resulting instance at node β, and let ℓ' be the number of nodes possessing credit in N_β. Then $k_\beta \le k_\alpha - 1$ and $\ell' \ge \ell - 2$. By induction, the number of leaves in $\mathcal{T}_\beta$ is bounded by $2^{k_\beta - \ell'/2} \le 2^{k_\alpha - 1 - (\ell - 2)/2} = 2^{k_\alpha - \ell/2}$. Therefore the number of leaves in $\mathcal{T}_\alpha$ is bounded by $2^{k_\alpha - \ell/2}$ as claimed.

If the operation performed at α is a $(1, 1)$-branch, let β and γ be the two children of α in $\mathcal{T}$. Let $\mathcal{T}_\beta$ and $\mathcal{T}_\gamma$ be the subtrees of $\mathcal{T}$ rooted at β and γ, respectively. Let (N_β, k_β) and (N_γ, k_γ) be the resulting instances at β and γ, and ℓ' and ℓ'' be the numbers of nodes that possess credit in N_β and N_γ, respectively. From the above discussion, each side of the $(1, 1)$-branch causes the removal of one b-edge from N_α. Also, the operation in each side of the branch causes at least one node to give a credit and at most one node to lose its possessed credit. Therefore, $k_\beta \le k_\alpha - 1$, $k_\gamma \le k_\alpha - 1$, $\ell' \ge \ell$, and $\ell'' \ge \ell$. By induction, the number of leaves in $\mathcal{T}_\beta$ is bounded by $2^{k_\beta - \ell'/2} \le 2^{k-1-\ell/2}$ and the number of leaves in $\mathcal{T}_\gamma$ is bounded by $2^{k_\gamma-\ell''/2} \leq 2^{k-1-\ell/2}$. It follows that the number of leaves in $\mathcal{T}_\alpha$ is bounded by the number of leaves in $\mathcal{T}_\beta$ plus the number of leaves in $\mathcal{T}_\gamma$, which is bounded by $2^{k-\ell/2}$ as claimed.

If the operation performed by the algorithm is a $(1,1,1)$-branch, let β, γ, and θ be the children of α in $\mathcal{T}$. Let $\mathcal{T}_\beta$, $\mathcal{T}_\gamma$, and $\mathcal{T}_\theta$ be the subtrees of $\mathcal{T}$ rooted at β, γ, and θ, respectively. Let (N_β, k_β), (N_γ, k_γ), and (N_θ, k_θ) be the resulting instances at β, γ, and θ, respectively, and let ℓ', ℓ'', ℓ''' be the numbers of nodes that possess credit in N_β, N_γ, and N_θ, respectively. From the above discussion, in the first and second sides of the branch in **Case 3** (step 4 in the algorithm), we will end up labeling one node and removing one b-edge (u, u'). The removal of the b-edge decreases the parameter by at least 1. Moreover, the removal of the b-edge (u, u') does not cause any node to lose its possessed credit, because this removal can only cause the nodes $\pi(u)$ and $\pi(u')$ to lose credit, and these two nodes were unlabeled before the operation, and hence possessed no credit. On the other hand, the labeling in each of the first two sides of the operation causes at least one node to give a credit. Therefore, we have $k_\beta \leq k_\alpha - 1$, $k_\gamma \leq k_\alpha - 1$, $\ell' \geq \ell + 1$, and $\ell'' \geq \ell + 1$. In the third side of the branch, we remove at least one b-edge causing the decrease of the parameter by at least 1, and we label two nodes that were unlabeled before. By a similar argument, the removal of the b-edge does not cause the loss of any credit due to the fact that the nodes that could lose credit were unlabeled and never possessed any credit. Now the labeling of the nodes $\pi(u)$ and $\pi(u')$ will cause at least two nodes to give credit. These two nodes that will give credit are distinct because the subtrees rooted at $\pi(u)$ and $\pi(u')$ are disjoint. Therefore, $k_\gamma \leq k_\alpha - 1$ and $\ell''' \geq \ell + 2$. Inductively, the number of leaves in $\mathcal{T}_\beta$ is bounded by $2^{k_\beta-\ell'/2} \leq 2^{k_\alpha-1-(\ell+1)/2} = 2^{k_\alpha-\ell/2-3/2}$, the number of leaves in $\mathcal{T}_\gamma$ is bounded by $2^{k_\gamma-\ell''/2} \leq 2^{k_\alpha-1-(\ell+1)/2} = 2^{k_\alpha-\ell/2-3/2}$, and the number of leaves in $\mathcal{T}_\theta$ is bounded by $2^{k_\theta-\ell'''/2} \leq 2^{k_\alpha-1-(\ell+2)/2} = 2^{k_\alpha-\ell/2-2}$. It follows that the number of leaves in $\mathcal{T}_\alpha$ is bounded by $2^{k_\alpha-\ell/2-3/2} + 2^{k_\alpha-\ell/2-3/2} + 2^{k_\alpha-\ell/2-2} \leq 2^{k_\alpha-\ell/2}$, as claimed.

Since any operation performed by the algorithm belongs to one of the above three categories, the number of leaves in $\mathcal{T}_\alpha$ is bounded by $2^{k_\alpha-\ell/2}$ and the above statement follows.

Now if we apply the statement to α with α being the root of the tree $\mathcal{T}$, then $k_\alpha = k$ and $l = 0$, and we get that the number of leaves in $\mathcal{T}$ is $O(2^k)$. This completes the proof. □

Theorem 6.4 *The* Parameterized BCCPN *problem can be solved in time $O(2^k n^2)$ where n is the number of nodes in the network.*

Proof. By Theorem 5.12, the algorithm **Is_Compatible** solves the Parameterized BCCPN problem correctly. Let $\mathcal{T}$ be the search tree of the algorithm on an instance (N, k) of the problem. The running time of the algorithm is the number of leaves in the search multiplied by the time spent on any root-leaf path. By Lemma 6.3, the number of leaves in $\mathcal{T}$ is $O(2^k)$. Let P be a root-leaf path in $\mathcal{T}$. On every node on P the algorithm might need to call the subroutines **Clean**, **Reduce**, and **Merge** on every node in N, which could take $O(n+k)$ time since the size of N is $O(n+k)$ (note that N has n nodes and hence $n-1$ tree edges, and k b-edges). However this need not be the case with a careful implementation of each of these subroutines. Instead of calling **Clean** at each node of the tree, we only call it on the nodes on which the operation is applicable. This can be done as follows. For every node in N, we partition its neighbors defined by the b-edges into three lists: those that are unlabeled, those labeled with 0, and those labeled with 1. We call **Clean** whenever a node u is labeled. When **Clean** is called on a node u that has just been labeled, we look at the list of its neighbors defined by the b-edges that have opposite labels. This labeling of u results in the removal of the b-edges from u to all these neighbors. The time spent by **Clean** in each such call is proportional to the number of b-edges removed in the call. We also need to update the adjacency lists of the nodes that are adjacent (via b-edges) to u. This also takes time proportional to the number of b-edges incident on u. This update is only done once when the node is labeled (a node never gets re-labeled in the whole algorithm). Therefore, we can say that the total time spent by **Clean** on a root-leaf path is proportional to the size of the network, which is $O(n+k)$.

A similar analysis shows that the time taken by **Reduce** and **Merge** along P is also $O(n+k)$. An additional multiplicative factor of $O(n)$ results from trying every node in N as the splitting node. It follows that the running time of the algorithm is $O(2^k(n+k)n) = O(2^k n^2)$. $\square$

References

[1] H. Bodlaender, M. Fellows, and T. Warnow. Two strikes against perfect phylogeny. In *Proceedings of ICALP'92*, LNCS, pages 273–283. Springer Verlag, 1992.

[2] A.J. Dobson. Unrooted trees for numerical taxonomy. Unpublished manuscript.

[3] A.J. Dobson. Lexicostatistical grouping. *Anthropological Linguistics*, 11:216–221, 1969.

[4] R. Downey and M. Fellows. *Parameterized Complexity.* Springer, New York, 1999.

[5] J. Felsenstein. Numerical methods for inferring evolutionary trees. *Quarterly Review of Biology*, 57:379–404, 1982.

[6] H.A. Gleason. Counting and calculating for historical reconstruction. *Anthropological Linguistics*, 1:22–32, 1959.

[7] Russell D. Gray and Quentin D. Atkinson. Language-tree divergence times support the anatolian theory of indo-european origin. *Nature*, 426(6965):435–439, November 2003.

[8] W. Labov. *Principles of language change, Vol. 1: internal factors.* Blackwell, Oxford, 1994.

[9] V.H. Mair, editor. *The Bronze Age and Early Iron Age Peoples of Eastern Central Asia.* Institute for the Study of Man, Washington, 1998.

[10] J.P. Mallory. *In Search of the Indo-Europeans.* Thames and Hudson, London, 1989.

[11] L. Nakhleh. *Phylogenetic Networks.* PhD thesis, The University of Texas at Austin, 2004.

[12] L. Nakhleh, D. Ringe, and T. Warnow. Perfect phylogenetic networks: A new methodology for reconstructing the evolutionary history of natural languages. *LANGUAGE*, 2005. In press.

[13] D. Ringe. Some consequences of a new proposal for subgrouping the IE family. In B.K. Bergen, M.C. Plauche, and A. Bailey, editors, *24th Annual Meeting of the Berkeley Linguistics Society, Special Session on Indo-European Subgrouping and Internal Relations*, pages 32–46, 1998.

[14] D. Ringe, T. Warnow, and A. Taylor. Indo-European and computational cladistics. *Transactions of the Philological Society*, 100(1):59–129, 2002.

[15] D. Ringe, T. Warnow, A. Taylor, A. Michailov, and L. Levison. Computational cladistics and the position of Tocharian. In V. Mair, editor, *The Bronze Age and early Iron Age peoples of Eastern Central Asia*, pages 391–414. 1998.

[16] R.G. Roberts, R. Jones, and M.A. Smith. Thermoluminescence dating of a 50,000-year-old human occupation site in Northern Australia. *Science*, 345:153–156, 1990.

[17] M. Ross. Social networks and kinds of speech-community events. In R. Blench and M. Spriggs, editors, *Archaeology and language I: theoretical and methodological orientations*, pages 209–261. Routledge, London, 1997.

[18] A. Taylor, T. Warnow, and D. Ringe. Character-based reconstruction of a linguistic cladogram. In J.C. Smith and D. Bentley, editors, *Historical Linguistics 1995, Volume I: General issues and non-Germanic languages*, pages 393–408. Benjamins, Amsterdam, 2000.

[19] T. Warnow. Mathematical approaches to comparative linguistics. *Proc. Natl. Acad. Sci.*, 94:6585–6590, 1997.

[20] T. Warnow, D. Ringe, and A. Taylor. Reconstructing the evolutionary history of natural languages. Technical Report 95-16, Institute. for Research in Cognitive Science, Univ. of Pennsylvania, 1995.

[21] T. Warnow, D. Ringe, and A. Taylor. Reconstructing the evolutionary history of natural languages. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 314–322, 1996.

[22] J.P. White and J.F. O'Connell. *A Prehistory of Australia, New Guinea, and Sahul*. Academic Press, New York, 1982.