# 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 $O(2^k n)$. 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 impacts 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, Texas A&M University, College Station, TX 77843-3112. Email: gexia@cs.tamu.edu.

people began to use horses in agriculture, the identity of physically European mummies found in China, etc. (see in particular [9, 14]). Various researchers [4, 7, 15] 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 [6]). 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 [12, 15, 16, 18]), 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 [10, 11]. 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 [10], 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. We first prove the NP-hardness of this problem. Then we present a branch-and-bound parameterized algorithm that solves the problem in $O(2^k n)$ time. The algorithm employs several interesting structural (network) operations that could be 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 $O(2^k n)$. The analysis phase of the algorithm is very elegant illustrating once more (see [2]) 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. Due to the lack of space, some proofs have been omitted from the paper, and the others are given in the appendix.

## 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 taxon in $L$ corresponds

to a leaf in the evolution tree. Qualitative character data reflect specific observable discrete characteristics of the taxa under study. Qualitative characters are usually described 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 or parallel evolution. This is formalized as follows.

Suppose $T$ 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 $\alpha$ 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 $\alpha$). That is, $\alpha$ is a labeling to the leaves in $T$. We say a qualitative character $\alpha$ is compatible (or "convex") on $T$ if we can extend $\alpha$ to every internal node of the tree $T$, thus defining a qualitative character $\alpha'$, or a labeling to the internal nodes of $T$, so that for every state, the nodes having that state form a connected subgraph of $T$. (In other words, $\forall z \in Z, \{v \in V(T) : \alpha'(v) = z\}$ is connected.)

To gain an intuition out of the above, deciding whether a tree is compatible on a certain character is equivalent to the following problem. Given a rooted binary 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 forms a connected subtree.

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*:

**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 the character $\alpha$. We can test the compatibility of $\alpha$ on $T$ in $O(n)$ time.*

The initial analysis of the Indo-European data done by Warnow and Ringe in [17] demonstrated that the Indo-European linguistic data is, nevertheless, "almost perfect". 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 Indo-European 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, 13]. In this case, a tree model is inappropriate, and the evolutionary process is better represented as a "network" [10].

# 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. We formalize this 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 $T$.

We refer to properties (iii) and (iv) above 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) $e$ is 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 $\Gamma$ the set of the trees induced by the $t$ b-edges in $N$.

An assignment to the statuses of the edges in $N$ is said to be *successful* if the induced tree by this assignment is a compatible tree. 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 ceratin 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 these edges were assigned in. The order of the assignment will not be an issue for us in the remainder of the discussion.

**Definition** Let $N = (V, E)$ be a phylogenetic network on $L$, $\Gamma$ 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 $c : L \to Z$ 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 $\Gamma$. $N$ is called a *Perfect Phylogenetic Network* if all characters in $C$ are compatible on $N$.

We define the Character Compatibility on Phylogenetic Networks problem, denoted henceforth by CCPN, as follows.

**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 [10] 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.

**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.

**Theorem 3.1 (Theorem 6.1, Appendix)** BCCPN *is NP-complete.*

Theorem 3.1 implies that the CCPN problem is NP-complete as well by specialization, giving an alternative yet different proof to that in [10].

**Remark 3.2** *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 $N$.*

# 4 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 $k$ [5]. The area of parameterized algorithms and complexity was introduced mainly in the work of Downey and Fellows [5], 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 [11], 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 edge in the network, considered one by one, 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 start by presenting some facts and operations. Most

of these facts are simple and easy to verify. We omit the proofs and leave their verification to the interested reader. Fact 4.1–Fact 4.7, and Proposition 4.8, are essential to understanding how the algorithm works and to verifying its correctness.

**Fact 4.1** *Let $u$ be a node in a network $N$. If $N$ is compatible, then there exists a successful assignment to the edges in $N$ in which at most one b-edge incident on $u$ is directed towards $u$.*

**Fact 4.2** *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 $N$. Then $N$ is compatible if and only if there exists a successful assignment $A$ to the b-edges in $N$ in which $(u, u')$ is removed.*

We will assume that we have an operation called **Clean($N$)** that removes b-edges between nodes of different labels in a phylogenetic network $N$. According to Fact 4.2, the operation is correct in the sense that the network resulting from applying the operation **Clean($N$)** is compatible if and only if $N$ is compatible. Moreover, it can be readily seen that the phylogenetic network properties, such as weak acyclicity, and the property that each internal node has at leat two children, are preserved by this operation.

**Fact 4.3** *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 $N$. Then $N$ is compatible if and only if there exists a successful assignment $A$ to the b-edges in $N$ in which $(u, u')$ is not removed, i.e., in which $(u, u')$ is either directed towards $u$ or towards $u'$.*

**Fact 4.4** *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 $N$. Then $N$ is compatible if and only if there exists a successful assignment $A$ to the b-edges in $N$ in which $(u, u')$ is removed.*

**Fact 4.5** *Let $u$ be a node in a network $N$. Let $N'$ be the network obtained by applying the operation* **Reduce** *given in Figure 1 to $N$. Then $N'$ is compatible if and only if $N$ is compatible.*

**Fact 4.6** *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 $v$.*

**Fact 4.7** *Let $u$ and $u'$ be two non-root nodes in a phylogenetic network $N$. Suppose that $label(\pi(u)) \neq label(u) = label(u')$, and that $(u, u')$ is a b-edge in $N$. Suppose further that the operation* **Reduce** *is not applicable to any node in $N$. Then the network $N'$ obtained by applying the operation* **Merge** *given in Figure 1 is a phylogenetic network that is compatible if and only if $N'$ is compatible.*

**Proposition 4.8 (Proposition 6.2, Appendix)** *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.*

We call a pair of nodes $\{u, u'\}$ satisfying the conditions in Proposition 4.8 a *nice pair*. Proposition 4.8 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 **Is_Compatible** 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

**Reduce($u$)**

1. **if** $u$ has two leaves with different labels **then reject**;
2. **if** all the children of $u$ are leaves and there are no b-edges incident on $u$ **then**
   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**
   set $label(u) = label(w)$;
4. **if** $u$ is labeled and has an unlabeled child $w$ with no b-edge incident on $w$ **then**
   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;

**Merge($\langle u, v \rangle$)**

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

**Is_Compatible**

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. **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 $label(\pi(u)) = labe(\pi(u')) = 1$ and remove the b-edge $(u, u')$;

Figure 1: The algorithm **Is_Compatible** and the subroutines **Reduce** and **Merge**

instances. We implicitly assume that after each step, the network $N$ and the parameter $k$ are updated accordingly. The algorithm is given in Figure 1. Note that the subroutines **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.

**Theorem 4.9 (Theorem 6.3, Appendix)** *The algorithm* **Is_Compatible** *is correct.*

# 5 Analysis of the algorithm Is_Compatible

In this section we analyze the running time of the algorithm. 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 specific technique that we use was suggested very recently to analyze parameterized algorithms in [2]. The application of the technique so far seems to be problem-dependent, and applying it to a given problem requires developing the necessary tools and operations that render the application of the technique feasible and beneficial. The technique in general suggests labeling the nodes of a search tree to record the reduction in the parameter size for each branching process. Then performing an amortized analysis on each path in the search tree. Doing each of these steps, and then showing that the operations along each root-leaf path can be balanced in such case, is a highly nontrivial task that depends on the given problem. However, if the above has been done, 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 $(N, k)$. Let $\alpha$ be a node in the search tree with an associated parameter $k'$. If we perform an $r$-sided branch at $\alpha$ by reducing the parameter $k'$ in each branch by $a_1, \cdots, a_r$, respectively, then such a branch is called an $(a_1, \cdots, a_r)$-*branch*. We will always assume that in an $(a_1, \cdots, a_r)$-branch, we have $a_1 \leq \cdots \leq a_r$. Each branch $(a_1, \cdots, a_r)$ can be associated with 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 all branches are of the form $(a_1, \cdots, a_r)$. If the branches of the algorithm are not classified within a single form, then we can look at all 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.

Since each side of a branch $(a_1, \cdots, a_r)$ corresponds to a reduction in the parameter by a value $a_i$ ($i \in \{1, \cdots, r\}$), the coordinates of a branch are usually taken to be positive integers. However, these coordinates need not be integers. If we allow the coordinates to be positive real numbers, then we can define the notion of a *characteristic function* of a branch $(a_1, \cdots, a_r)$ in a similar fashion as before, to be the function $f(x) = x^{-a_r} + x^{-a_{r-1}} + \cdots + x^{-a_1} - 1$. It can be shown, using basic results in real analysis (see [3] for a proof), that in the interval $(0, \infty)$ $f(x)$ has a unique root $x_0$, and that $f(x) > 0$ if and only if $x < x_0$. Equivalently, this is saying that the solution $x_0$ to this recurrence, which gives the upper bound $O(x_0^k)$ on the number of leaves in the search tree, satisfies $x_0^k - x_0^{k-a_r} - x_0^{k-a_{r-1}} - \cdots - x_0^{k-a_1} = 0$.

Therefore, whether the coordinates of the branch are integers or not becomes irrelevant at this point since as long as the number $x_0$ satisfies this equation, $O(x_0^k)$ is an upper bound on the number of leaves of the search tree. However, an explanation to how these non-integer coordinates come into the picture in the context of parameters, branches, and search trees, is in order here. When we have a branch of the form $(a_1, \cdots, a_r)$, where $a_1, \cdots, a_r$ are positive real numbers, how do we interpret this branch in conjunction with the notion of a search tree? In the standard notion of a branch, the coordinates of a branch correspond to the reduction of the parameter along each side of the branch. This notion makes sense when these coordinates are positive integers. To illustrate how such a branch can be interpreted when the coordinates are positive real numbers, consider a branch of the form $(n_1 + s, n_2, \cdots, n_p)$, where $n_1, \cdots, n_p$ are positive integers, and $s$ is a real positive number. Such branches may arise when $s$ is a "surplus" or a "gain" resulting from another operation (which could be a branch or not) $(b_1, \cdots, b_q)$ on the $n_1$-side of the branch $(n_1, n_2, \cdots, n_p)$. When it comes to interpreting the branch $(n_1+s, n_2, \cdots, n_p)$ with respect to the notion of the search tree and the reduction in the parameter, we look at this branch as a $(n_1, n_2, \cdots, n_p)$-branch followed by a surplus $s$ on the $n_1$-side. Now to calculate an upper bound on the size of the search tree, we can simply use the root of the characteristic function of the branch $(n_1 + s, n_2, \cdots, n_p)$ (see [3]). Therefore, hypothetically, we can look at the branch $(a_1, \cdots, a_r)$ as a branch with real-valued coordinates and use the root of its characteristic function to compute an upper bound on the size of the search tree. However, such a branch should be understood as a standard branch with integer coordinates, with the accumulation of certain surplus along some of its sides.

In this section we will show that the number of leaves of 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 4.8, we know that a nice pair $\{u, u'\}$ exists before each branch. By Fact 4.3, 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 should give us an $O(2^k)$ upper bound on the size of the search tree. However, there is a subtle point here that can 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. 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 guarantee 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 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 trifle matter, and the analysis now takes a

new turn.

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 give a *credit* of value $1/2$ when it gets labeled and there is a b-edge $e$ incident on one of its children. We say that this b-edge $e$ will carry a *debit* of value $1/2$.

Note that each b-edge can carry at most two debits, each of value $1/2$, corresponding to the credits given by the parents of the two nodes incident on this b-edge. Ultimately, the value of a debit/credit will correspond to a reduction in the parameter of the same value.

**Proposition 5.1 (Proposition 6.5, Appendix)** *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 node $v$. Then there exists a node in the subtree $T_v$ of $T_N$ rooted at $v$ that can give a credit of value* $1/2$.

The idea of an operation giving a credit and paying a debit 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 that will be payed as a debit by other operations later on. When a node gives a credit of value $1/2$, we associate this credit with a certain b-edge. We would like this b-edge to pay the debit of value $1/2$ back when the edge gets removed and the parameter $k$ is reduced. An edge might need to pay two debits, each of value $1/2$, to two nodes. When a b-edge gets removed, the

b-edge should contribute by a reduction of value 1 to the parameter. However, if an edge has to pay a debit of value 1/2, then when this edge is removed, the edge can only contribute a value 1/2 to the reduction of the parameter. Similarly, if the edge has a debit of value 1, then the edge can no longer contribute to the reduction in the parameter when its removed because it has to pay its debit of value 1. Let us look at the operations performed by the algorithm to see how this works. We consider the operations one by one.

### 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 have given a total credit of value 1 (1/2 each), and this credit may have been associated with the b-edge $e$. Therefore, when $e$ is removed, $e$ might need to pay back a debit of value 1. Consequently, an edge removed by **Clean** can pay its debit.

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 equal to the value of the debit this edges needs to pay.

### $(1, 1)$-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 5.1, labeling $\pi(u')$ will give a credit of value 1/2. Since $\pi(u')$ was unlabeled before this operation, no credit was given by $\pi(u')$. Hence, only a debit of value 1/2 could have been associated with the edge $e$. This is the debit 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 debit 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 $(1, 1)$-branch, and the b-edge removed by this branch has paid its (possible) debit.

### $(1, 1, 1)$-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 have any debit to pay. Labeling $\pi(u)$ gives a credit of value 1/2 by Proposition 5.1. 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 have any debit to pay 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 branches computed above are the effective branches where the reduction of the parameter gained from the non-branching operations has been combined with the branching operations. The worst branch is the $(1, 1)$-branch which gives an upper bound of $O(2^k)$ on the number of leaves in the search tree. This was an intuitive look at the amortized analysis of the algorithm. We have:

**Lemma 5.2 (Lemma 6.6, Appendix)** *Let $\mathcal{T}$ be the search tree corresponding to the algorithm* **Is_Compatible** *on an instance $(N, k)$. Then the number of leaves of $\mathcal{T}$ is $O(2^k)$.*

**Theorem 5.3 (Theorem 6.7, Appendix)** *The* Parameterized BCCPN *problem can be solved in time $O(2^k n)$ where $n$ is the number of nodes in the network.*

## 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] J. Chen, I. Kanj, and G. Xia. Labeled search trees and amortized analysis: improved upper bounds for NP-hard problems. In *ISAAC'03*, volume 2906 of *LNCS*, pages 148–157. Springer, 2003, to appear in *Algorithmica*.

[3] J. Chen, I. Kanj, and G. Xia. A note on search trees. Technical Report TR05-006 (available at http://www.cti.depaul.edu/research/technical.asp), School of CTI, DePaul University, April 2005.

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

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

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

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

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

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

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

[18] 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.

# 6 Appendix

**Theorem 6.1** 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 $x_i$.

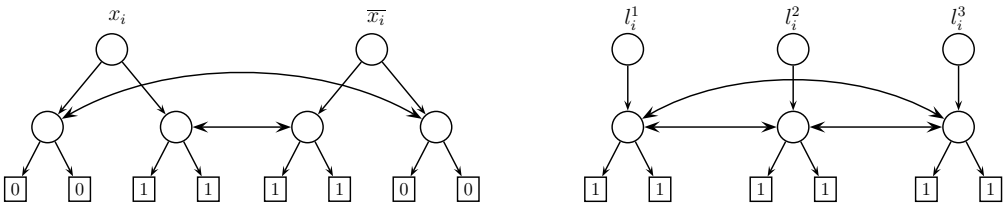


Figure 2: The V-Gadget (left) and the C-Gadget (right).

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 $\tau$ is a valid truth assignment to the variables in $F$, then $\tau$ 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 $\tau$, 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 $\tau$ 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 a degree in the underlying tree larger than 1.) 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 $\tau$ 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 $\tau$. 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.

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

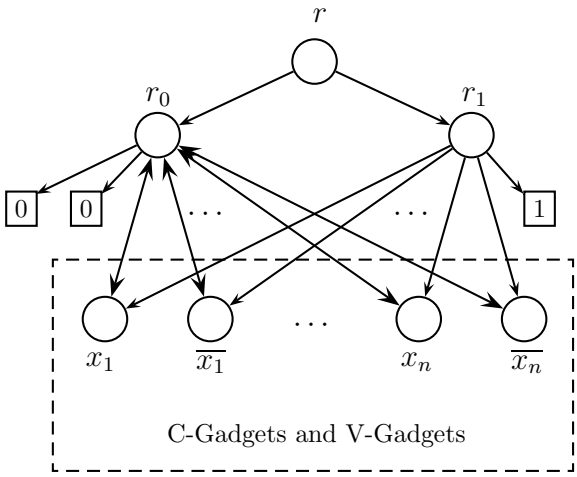


Figure 3: The partition gadget and the whole network.

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 $\tau$ to the variables in $F$ that satisfies every clause in $F$. Since $\tau$ is valid, we can label the nodes corresponding to the literals in $F$ in every V-Gadget by the truth values assigned by $\tau$ 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 $\tau$ 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 $\tau$ 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. □

**Proposition 6.2** *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 a 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, $w_1$ 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. □

**Theorem 6.3** *The algorithm* **Is_Compatible** *is correct.*

PROOF. Step 1 of the algorithm 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 Fact 4.5 and Fact 4.7, and from the discussion about the subroutine **Clean**. 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 4.8, we only need to show that the network $N$ satisfies the phylogenetic network 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 two operations **Clean** and **Merge** preserve the phylogenetic network properties given that the operation **Reduce** is not applicable. Moreover, step 4 of **Is_Compatible** preserves the phylogenetic network properties (since it only labels the nodes and possibly removes some b-edges). Therefore, it suffices to show that after the execution of step 2 of the algorithm, i.e., the operation **Reduce**, the phylogenetic network properties are maintained. The only place in **Reduce** where the phylogenetic properties may be violated is in step 2 where a node $u$ is removed together with its children and a leaf is added to $\pi(u)$. But this operation does not decrease the number of children of any internal node in the resulting network, nor does it affect the weak acyclicity of the network. It follows by an inductive argument that the phylogenetic network 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 branch 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 considers 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 4.4. 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 4.4. 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 4.4. 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 4.4.

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. □

**Fact 6.4** *Let $v$ be an unlabeled node in a phylogenetic network $N$, and suppose that* **Reduce** *is not applicable to $N$. 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 $\{v_{i-1}, \cdots, v_2\}$ 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 $i \neq 1$ be the smallest index such that $v_i$ has a b-edge incident on it. Since $v$ is unlabeled, and **Reduce** is not applicable to $N$, such $i$ must exist by step 3 of **Reduce**. By the choice of $i$, all the nodes $v_{i-1}, \cdots, v_2$ have no b-edges incident on them. Moreover, the nodes $v_{i-1}, \cdots, v_2$ are unlabeled, otherwise, by step 3 of **Reduce**, $v$ would be labeled since there are no b-edges incident on $v_{i-1}, \cdots, v_2$. □

**Proposition 6.5** *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 node $v$. Then there exists a node in the subtree $T_v$ of $T_N$ rooted at $v$ that can 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 6.4, 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. □

**Lemma 6.6** *Let $\mathcal{T}$ be the search tree corresponding to the algorithm* **Is_Compatible** *on an instance $(N, k)$. Then the number of leaves of $\mathcal{T}$ is $O(2^k)$.*

PROOF. The nodes in $\mathcal{T}$ correspond to the operations of the algorithm. We label the nodes in $\mathcal{T}$ as follows. Assign to each node $\alpha$ a *label* whose value is equal to the parameter reduction of the operation specified by the node $\alpha$. More precisely, if the operation from the parent of a node $\alpha$ in $\mathcal{T}$ to $\alpha$ is the $a_i$-side of an $(a_1, \cdots, a_i, \cdots, a_r)$ branch, then the label of $\alpha$ is $a_i$; if the operation from the parent of $\alpha$ to $\alpha$ is a non-branching operation that reduces the parameter value by $c$, then the label of $\alpha$ is $c$. The root of $\mathcal{T}$ specifies a dummy operation whose parameter reduction is 0, and the label of the root is 0. The non-branching operations reduce the parameter without any branching, and hence, correspond to the 1-child nodes in $\mathcal{T}$. For a root-leaf path $P$ in $\mathcal{T}$, we define the *credit of* $P$, $cr(P)$, to be the the sum of the labels of all non-branching nods on $P$. Let $x_1(P)$ be the number of nodes on $P$ corresponding to the first side of a $(1, 1, 1)$-branch, $x_2(P)$ the number of nodes on $P$ corresponding to the second side of a $(1, 1, 1)$-branch, and $x_3(P)$ the number of nodes on $P$ corresponding to the third side of a $(1, 1, 1)$-branch. From the above discussion of each of the operations in the algorithm, when the algorithm halts, the credits given by the nodes in $N$ are paid as debits by the corresponding b-edges of $N$. This can be interpreted as follows. For each first side of a $(1, 1, 1)$-branch with a corresponding node $\alpha$ in the search tree, a credit of value $1/2$ is given. Since all the credits are paid at the end of the algorithm, there must exist non-branching operations along each root-leaf path of the subtree of $\mathcal{T}$ rooted at $\alpha$ that pay the debit corresponding to this credit of value $1/2$. Similarly for the second side of a $(1, 1, 1)$-branch. Now for each third side of a $(1, 1, 1)$-branch, the credit given is 1, and hence along each root-leaf path of the subtree of $\mathcal{T}$ rooted at $\alpha$ there exist non-branching operations that pay back this credit of value 1. We say that the path $P$ is *compressible* if $cr(P) \geq x_1(P)/2 + x_2(P)/2 + x_3(P)$, or

equivalently $Sur(P) = cr(P) - (x_1(P)/2 + x_2(P)/2 + x_3(P)) \geq 0$. The notation $Sur(P)$ denotes the *surplus* of path $P$. Since at the end of the algorithm all credits are paid back, each root-leaf path $P$ in $\mathcal{T}$ is compressible. Therefore it suffices to prove the above lemma for a search tree $\mathcal{T}$ in which each root-leaf path is compressible.

According to the algorithm **Is_Compatible**, each branch node in $\mathcal{T}$ is either a $(1,1)$-branch, or a $(1,1,1)$-branch. We say that a search tree $\mathcal{T}_0$ is *normalized* if: (1) for every 1-child node $\alpha$ in $\mathcal{T}_0$, the child of $\alpha$ is a leaf; and (2) every branch node in $\mathcal{T}_0$ is either a $(1,1)$-branch, or a $(1,1,1)$-branch. We can use the following procedure to convert a general search tree $\mathcal{T}$ into a normalized search tree $\mathcal{T}_0$, with a one-to-one correspondence between the root-leaf paths in the two trees, and such that the corresponding root-leaf paths in the two trees have the same credit. Let the leaves of the original search tree $\mathcal{T}$ be $\alpha_1, \ldots, \alpha_t$. We first construct, based on the tree $\mathcal{T}$, a search tree $\mathcal{T}'$ with leaves $\alpha'_1, \ldots, \alpha'_t$, as follows. For each $i$, let the path from the root to the leaf $\alpha_i$ in $\mathcal{T}$ be $P_i$. If $Sur(P_i) = 0$, then leave the path $P_i$ unchanged and let $\alpha'_i$ in $\mathcal{T}'$ be $\alpha_i$. If $Sur(P_i) > 0$, then add to $P_i$ a new leaf $\alpha'_i$ with label $Sur(P_i)$ and make $\alpha'_i$ the unique child of $\alpha_i$ (thus $\alpha_i$ becomes a 1-child non-leaf node in $\mathcal{T}'$). To obtain the normalized tree $\mathcal{T}_0$, we further perform the following operation. We remove all non-branching nodes: for each 1-child node $\alpha$ in the tree with a child $\beta$, where $\beta$ is not a new leaf created in $\mathcal{T}'$, remove the edge $(\alpha, \beta)$, merge the two nodes $\alpha$ and $\beta$, and assign a label to the resulting (new) node equal to the label of $\alpha$ (this corresponds to removing the non-branching operation specified by $\beta$). The resulting tree $\mathcal{T}_0$, with leaves $\alpha'_1, \ldots, \alpha'_t$, is a normalized search tree. Let $P_i$ be the path from the root to the leaf $\alpha_i$ in $\mathcal{T}$ and let $P'_i$ be the path from the root to the leaf $\alpha'_i$ in $\mathcal{T}_0$. Since no $(1,1,1)$-branch nodes are changed or re-labeled in the above procedure, we have $x_1(P_i) = x_1(P'_i)$, $x_2(P_i) = x_2(P'_i)$, and $x_3(P_i) = x_3(P'_i)$. Moreover, if $Sur(P_i) = 0$, then $P_i$ is not changed by the above transformation, and the path $P'_i$ is the same as $P_i$ giving $Sur(P'_i) = 0$. On the other hand, if $Sur(P_i) > 0$, then by our construction, the only node on $P'_i$ that is not a $(1,1)$-branch or a $(1,1,1)$-branch, is the 1-child node whose child is the leaf $\alpha'_i$ with a label $Sur(P_i)$. Thus, $Sur(P'_i) = Sur(P_i)$, and the paths $P_i$ and $P'_i$ have the same credit. Consequently, each root-leaf path in $\mathcal{T}_0$ is compressible.

From the above discussion, for each general search tree satisfying the condition in the lemma, there is a normalized search tree with the same number of leaves that also satisfies the conditions in the lemma. Thus, it suffices to prove the proposition for normalized search trees. We do this by induction on the number of nodes in a normalized search tree $\mathcal{T}$. The lemma certainly holds true if the tree $\mathcal{T}$ consists of a single node or has only one leaf. Now assume that $|\mathcal{T}| > 1$ and that $\mathcal{T}$ has more than one leaf. Since $\mathcal{T}$ is normalized, the root $\alpha$ of $\mathcal{T}$ must be a branch node, which is either a $(1,1)$-branch node, or a $(1,1,1)$-branch node.

Suppose first that the root $\alpha$ of $\mathcal{T}$ is a $(1,1)$-branch. Let $\beta_1$ and $\beta_2$ be the children of $\alpha$ labeled 1. Let $\mathcal{T}_1$ be the subtree rooted at $\beta_1$ in $\mathcal{T}$. By re-setting the label of $\beta_1$ to 0, the subtree $\mathcal{T}_1$ becomes a valid normalized search tree for the algorithm **Is_Compatible** on input $(N', k-1)$, where $N'$ is the network resulting from $N$ by the operation specified by the node $\beta_1$. Moreover, each root-leaf path in $\mathcal{T}_1$ is compressible since the node $\beta_1$ in $\mathcal{T}$ is not a child of a $(1,1,1)$-branch. Now by the inductive hypothesis, the number of leaves in $\mathcal{T}_1$ is bounded by $2^{k-1}$. Similarly, re-setting the label of $\beta_2$ to 0 makes the subtree rooted at $\beta_2$ a valid normalized search tree with no more than $2^{k-1}$ leaves. Adding the number of the leaves in the two subtrees, we get that the number of leaves in the search tree $\mathcal{T}$ is bounded by $2^{k-1} + 2^{k-1} = 2^k$.

Suppose now that the root $\alpha$ of $\mathcal{T}$ is a $(1,1,1)$-branch. Let $\beta_1$, $\beta_2$, and $\beta_3$ be the children of $\alpha$ labeled 1. Let $\mathcal{T}_1$ be the subtree rooted at $\beta_1$ in $\mathcal{T}$. Every path $P_i$ from the root $\alpha$ to a leaf $\alpha_i$ in $\mathcal{T}_1$ contains the node $\beta_1$, and hence $x_1(P_i) \geq 1$. Since the path $P_i$ is compressible, we have $Sur(P_i) = cr(P_i) - (x_1(P_i)/2 + x_3(P_i)/2 + x_2(P_i)) \geq 0$. It follows that $cr(P_i) \geq 1/2$, and the label

of the leaf $\alpha_i$ is at least 1/2. Therefore, in the tree $\mathcal{T}$ we can "shift" a 1/2 unit from the label of each leaf in the subtree $\mathcal{T}_1$ to the node $\beta_1$, by adding a 1/2 unit to the label of $\beta_1$ and subtracting 1/2 unit from the label of every leaf in $\mathcal{T}_1$. Now the label of $\beta_1$ becomes 3/2. Similarly, we can add a 1/2 unit to the label of the node $\beta_2$ and subtract a 1/2 unit from the label of every leaf in the subtree rooted at $\beta_2$, and we can add 1 unit to the label of the node $\beta_3$ and subtract 1 unit from the label of every leaf in the subtree rooted at $\beta_3$. This makes the label of $\beta_1$ and $\beta_2$ become 3/2 and the label of $\beta_3$ become 2. Note that the resulting search tree is still normalized, with the difference that the root $\alpha$ now becomes a $(3/2, 3/2, 2)$ -branch node, and that the label of each leaf in $\mathcal{T}$ has been decreased. In particular, each root-leaf path $P_i$ in the resulting tree is still compressible (with the value $x_1(P_i)$ reduced by 1, or $x_2(P_i)$ reduced by 1, or $x_3(P_i)$ reduced by 1, and the value $cr(P_i)$ decreased by 1/2, 1/2, or 1, respectively). Therefore, we end up with a normalized search tree $\mathcal{T}$ whose root is a $(3/2, 3/2, 2)$-branch in which all root-leaf paths are compressible. Let $\gamma_1$ and $\gamma_2$ be the children of $\alpha$ labeled by 3/2, and $\gamma_3$ be the child of $\alpha$ labeled by 2. Consider the subtree $\mathcal{T}'_1$ rooted at $\gamma_1$. By re-setting the label of $\gamma_1$ to 0, the subtree $\mathcal{T}'_1$ becomes a valid normalized search tree for the algorithm on input $(N', k - 3/2)$, where $N'$ is the network resulting from $N$ by the operation specified by $\gamma_1$. Moreover, each root-leaf path in $\mathcal{T}'_1$ is compressible. Now by the inductive hypothesis, the number of leaves in $\mathcal{T}'_1$ is bounded by $2^{k-3/2}$, Similarly, the number of nodes in the subtree rooted at $\gamma_2$ is bounded by $2^{k-3/2}$. Re-setting the label of $\gamma_3$ to 0 makes the subtree rooted at $\gamma_3$ a valid normalized search tree with no more than $2^{k-2}$ leaves. Adding the number of the leaves in the two subtrees, we get that the number of leaves in the search tree $\mathcal{T}$ is bounded by $2.2^{k-3/2} + 2^{k-2} = 2^{k-1/2} + 2^{k-2} \leq 2^k$. This completes the inductive proof and the proof of the proposition. □

**Theorem 6.7** *The* PARAMETERIZED BCCPN *problem can be solved in time $O(2^k n)$ where $n$ is the number of nodes in the network.*

PROOF. By Theorem 4.9, the algorithm **Is_Compatible** solves the PARAMETERIZED BCCPN problem correctly. Thus, it suffices to show that the running time of the algorithm is $O(2^k n)$. 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 5.2, 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 $O(n + k)$. It follows that the running time of the algorithm is $O(2^k(n + k)) = O(2^k n)$. This completes the proof. □