A Simple Linear Time Algorithm for Binary Phylogeny

João Meidanis
Computer Science Dept.
University of Campinas
Brazil
meidanis@dcc.unicamp.br

Erasmo G. Munuera
Computer Science Dept.
University of Campinas
Brazil
erasmogm@dcc.unicamp.br

March 31, 2020

Abstract

The Binary Phylogeny Problem is to reconstruct a tree describing the evolutionary history of a group of taxa for which a binary matrix of characteristics is given. Gusfield [5] presented an $O(mn)$-time algorithm for this problem with n taxa and m characteristics. This bound is tight if the input is given as an $n \times m$ matrix. In this paper we show that a faster algorithm is possible provided the input is given as a list of the "1" positions in the matrix. We present a simple algorithm for this problem based on coloring the vertices of a bipartite graph. The running time of our algorithm is $O(n + m + r)$, where r is the number of "1"s in the matrix.

Keywords: combinatorial problems, phylogenetic trees, data structures, design of algorithms.

1 Introduction

A *phylogenetic tree* gives an interpretation of the evolutionary history of a group of taxa. An important problem in science is to build such a tree from data relating the taxa. In one version of this problem, which we call the *Binary Phylogeny Problem*, the tree is constructed from a $n \times m$ binary matrix where each row corresponds to a taxon and each column to a character,

with "1" in position i, j if taxon i has character j. Here, as in the rest of the paper, n indicates the number of taxa and m, the number of characters.

The goal is to build a rooted tree in which the taxa are leaves and such that each character corresponds to a rooted subtree that contains exactly the taxa that have this character.

An $O(nm^2)$ algorithm to solve this problem was proposed by Camin and Sokal [3]. Later, a simple property of the matrix allowed an $O(n^2m)$ algorithm. Recently, Gusfield [5] presented an $O(nm)$ algorithm. All these results assume that the input is given as an $n \times m$ matrix. However, the input can be represented more efficiently as a bipartite graph.

The matrix format needs $\Omega(nm)$ space to store it, and in this case Gusfield's upper bound is tight because an algorithm must look at all positions in the matrix before it can even decide whether there is a phylogeny or not.

The bipartite graph format is defined as follows. Let X be the set of rows of the matrix M and Y the set of columns. There is an edge between $i \in X$ and $j \in Y$ if and only if $M[i, j] = 1$. This format requires only $O(n+m+r)$ space, where r is the number of ones in M. If M is sparse this represents a big savings. Furthermore, for the bipartite graph format one can design a linear time algorithm, that is, one running in $O(n + m + r)$ time. Notice that for the bipartite graph $G = (V, E)$ we have $|V| = |X| + |Y| = n + m$ and $|E| = r$. Even if the matrix is not sparse, the new algorithm is simpler and more elegant than previous ones.

Agarwala, Fernández-Baca, and Slutzki [1] were the first to find efficient solutions to the binary phylogeny problem for sparse matrices. Meidanis and Munuera [6] also gave a fast algorithm based on PQ-trees [2]. The advantages of the algorithm presented here are the following:

- production of a tree with minimum number of internal nodes, instead of one internal node per character as some of the previous constructions.
- the coloring scheme used has applications in other problems involving binary matrices, for instance, testing for the consecutive ones property. It is known that all matrices with a phylogeny have the consecutive ones property but not vice-versa [5].
- use of very simple data structures.

In addition, it is interesting to note that the order of processing the characters is different in each approach. Agarwala, Fernández-Baca, and

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

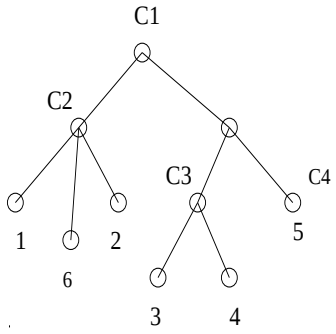

Figure 1: A matrix with an associated phylogeny.

Slutzki process the characters in nonincreasing order by size; the present approach processes them in non decreasing order; and the PQ-tree approach processes them in an arbitrary order.

We remark that Gusfield's algorithm [5] can also be adapted to run in linear time if the input is a bipartite graph, but again our algorithm is simpler.

The rest of this paper is organized as follows. In Section 2 we present some definitions used in this paper. Section 3 contains the main results. Concluding remarks appear in Section 4.

2 Definitions

Let M be an $n \times m$ binary matrix. We say that M **admits a phylogeny** or **has a phylogeny** when it is possible to build a rooted tree T for which the rows of M are in one-to-one correspondence with the leaves of T and such that for each column A of M the leaves that correspond to rows with a "1" in column A are precisely the leaves of a rooted subtree of T. In this case any such tree T is called a **phylogenetic tree** or a **phylogeny** for M. The phylogeny may not be unique.

Figure 1 shows a matrix with a corresponding phylogeny. In this figure C1, C2, and C3 represent the columns whereas 1, 2, 3, 4, and 5 represent the rows of M. Figure 2 shows a matrix which has no phylogeny.

We will use extensively the following well-known characterization of this property [5]. In the statement below we identify each column A with the

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Figure 2: A matrix with no phylogeny.

set of rows that have a "1" in column A.

Theorem 1 *A binary matrix M has a phylogenetic tree if and only if for each pair of columns A and B of M either $A \subseteq B$ or $B \subseteq A$ or else $A \cap B = \emptyset$.*

We sometimes use the term **taxon** to refer to a row of M and the term **character** to refer to a column of M. Also, we will routinely identify a character with a set of taxa as we did above.

Our graph notation is standard (see, for instance, [4]). We use the notation **adj(v)** to denote the set of vertices adjacent to v.

A binary matrix M can be represented as a bipartite graph $G = (V, E)$ with $V = X \cup Y$, where X is the set of rows of M and Y, the set of columns, and $E = \{vA \mid v \in X, A \in Y, \text{and } M[v, A] = 1\}$. If columns are viewed as subsets of X as we did earlier, then we can also write simply $E = \{vA \mid v \in A\}$.

3 The Algorithm

In this section we will present a simple algorithm to verify if a given matrix possesses a phylogeny. The algorithm also constructs a phylogenetic tree if there is one.

We assume the input is given as a bipartite graph $G = (X \cup Y, E)$, and elements of Y are viewed as subsets of X. The algorithm works in two phases.

The first phase sorts the characters by size (degree in G). This can be done in linear time using bucket sort or a similar technique.

The second phase uses Theorem 1 to verify, for each pair of columns A and B whether $A \subseteq B$, $B \subseteq A$, or $A \cap B = \emptyset$. Since they are ordered by size, if we know that $|A| \leq |B|$ it suffices to check whether $A \subseteq B$ or $A \cap B = \emptyset$. To check containment we color the vertices of each set in turn

with a different color and use the concept of *forbidden colors*, which we explain in the sequel.

In the beginning all vertices have color 0. This color will never be forbidden. We process each set in ascending order by size, giving each one a new color from the list $1, 2, 3, \ldots$. The color of a set is used to paint all its elements. If an element x of a set A that is being processed has already some nonzero color c, this color c is declared forbidden, meaning that we don't want to see c again after painting A. If we encounter a forbidden color again while painting a different set, the input does not have a phylogeny.

The algorithm is shown in Figure 3. Notice that new forbidden colors are stored in a waiting stack during the painting of the current set, and only after this set has been completely colored are they marked as forbidden. Also, after all sets have been processed, an extra pass thru all vertices in X is needed to check for the presence remaining of forbidden colors.

3.1 Proof of correctness

Throughout this section we will use the notation $A \preceq B$ meaning that A was placed before B in the first phase of the algorithm. This implies that $|A| \leq |B|$, but the converse is not true.

Matrix M admits a phylogeny if and only if the following property holds for all columns A and B:

$$A \subseteq B \text{ or } B \subseteq A \text{ or } A \cap B = \emptyset \tag{1}$$

We will initially prove that, if there are two columns A and B for which (1) fails, the algorithm returns false.

Without loss of generality suppose $A \preceq B$. This implies $|A| \leq |B|$. Also, without loss of generality take B as the first set processed after A for which (1) fails. Let $E = A \cap B$ and $\overline{E} = A \setminus E$ (set difference). Since (1) is not valid and $|A| \leq |B|$, we must have both E and $\overline{E}$ nonempty. Let v be any vertex in E and u any vertex in $\overline{E}$.

Let's focus on the coloring phase. When A is processed, its taxa will all get a certain color $a \neq 0$. Later, when B is processed, taxon v will get a new color, b, and its color c at this moment will be declared forbidden. Color c may or may not be the same as a, but in any case u will have the same color as v when B is about to be processed, because of the choice of B. Indeed, if v has color c this is because some set C processed after A and before B contains v. But then $A \preceq C \preceq B$ and $v \in A \cap C$ imply $A \subseteq C$, hence u got color c as well.

```
01  Initialize color(v) ← 0 for all v ∈ X
02  Initialize all colors as permitted (not forbidden)
03  new_color ← 1
04  for each vertex A ∈ Y  in nondecreasing order of size do
05      for each vertex v ∈ adj (A) do
06          if color(v) ≠ 0 then
07              if color(v) is forbidden then
08                  return false
09              else
10                  put color(v) in waiting stack
11              end if
12          end if
13          color(v) ← new_color
14      end for
15      new_color ← new_color + 1
16      remove all colors from waiting stack,
                and mark these colors as forbidden
17  end for
18  for each v ∈ X do
19      if color(v) is forbidden then
20          return false
21      end if
22  end for
23  return true
```

Figure 3: Coloring algorithm for the binary phylogeny problem.

So we are left with a vertex u such that color(u) is forbidden. This color will be caught either when the next set containing u is processed or in the end of the algorithm (lines 18–22), if no such set exists after B.

To conclude the proof, let's show that if (1) holds for every pair of columns A and B, then the algorithm returns true.

We will do that by showing that no forbidden color is ever detected in lines 7 or 19, so that the algorithm will reach the last line and will return true. Indeed, let A be the set corresponding to a forbidden color a and let B be the set during the process of which color a became forbidden. Of course we have $A \cap B \neq \emptyset$ and $A \preceq B$, hence $|A| \leq |B|$. Since (1) holds, we must have $A \subseteq B$. Therefore, color a was totally eradicated by B's color and there is no opportunity for future detection.

3.2 Complexity

We already saw that the sorting phase can be done in $O(n + m + r)$ time.

It is straightforward to check that each line of the algorithm in Figure 3 is executed $O(n+m+r)$ times. The most tricky part is perhaps the processing of line 16, but this is bounded by the number of times line 10 is executed, since everything removed from the stack must have been placed there earlier. Line 10 is executed at most once for each edge of the graph, so the linear bound holds.

As for the space complexity, the extra space needed is just a vector of integers to hold color(v) for each $v \in X$ and a vector of booleans to keep the forbidden information for colors. Since there is one color per set $A \in Y$, the total extra space is $O(n + m)$.

3.3 Tree construction

We can easily modify the algorithm given in Figure 3 to build a phylogenetic tree T at the same time. Of course, the resulting object will only be a valid tree if the algorithm returns true.

Each node of T will contain the following information:

- a list of characters, which we will call *c-list*, and
- a list of children (for internal nodes), or a taxon (for leaves).

In addition, according to the definition of phylogenetic tree, each character A must correspond to a node in the tree. This node is such that its

descendent leaves are exactly the taxa in A. In our construction, this node is unique and will be the only node containing A in its c-list.

Characters are processed in nondecreasing order of size, so our tree is built from the bottom up. When a given character is processed, either a new node is build and this character becomes the only element in the new node's c-list, or the character is included in the c-list of an already existing node. Notice that when a new node is created, all its children have been created already, because of the processing order.

During the processing of a character, we keep a list of nodes corresponding to the colors put in the waiting stack. Colors correspond to characters, which in turn are associated to nodes. Previously used colors correspond to already constructed nodes.

The following modifications in the algorithm will yield the tree.

1. when we begin processing a character, we initialize a children list with the empty list;

2. when we find a taxon with zero color in a character, we build a leaf node with it and add it to the children list;

3. when we find a taxon with a nonzero color in a character, we add the corresponding already existing node to the children list. Care must be taken in not adding the same node twice. A presence bit is sufficient for that.

4. when we finish processing a character, we examine the children list. If it contains only one node, we add the character to this node's c-list; otherwise, we build a new node with these children, and whose c-list will contain just the current character;

5. the same procedure is performed for the final loop in lines 18–22 (keeping a children list, etc.). But, in this case, there is no character to add. As a result, the root may have an empty character list.

The above additions do not modify the asymptotic time and space complexities.

4 Conclusions

We have presented a simple, efficient algorithm for checking the existence of a phylogeny for a binary matrix coded as a bipartite graph. The algorithm

runs in time $O(n + m + r)$ where $n \times m$ is the size of the matrix and r is the number of "1" entries.

The space complexity of the algorithm is also linear in n, m, and r. This algorithms improves on previous ones in that it constructs a more compact tree, uses simple data structures, and has linear complexity.

The coloring technique used by the algorithm is a powerful idea and may perhaps be extended to other important problems, for instance, checking the consecutive ones property for a binary matrix.

Acknowledgements.

We thank the Brazilian agencies FAPESP, CNPq, and FAEP/UNICAMP for providing financial support for this work.

References

[1] R. Agarwala, D. Fernández-Baca, and G. Slutzki. Fast algorithms for inferring evolutionary trees. Technical Report TR 92-19, Dept. of Computer Science, Iowa State University, 226 Atanasoff, Ames, IA 50011, USA, July 1992.

[2] K. S. Booth and G. S. Leuker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Sys. Sci.*, 13(3):335–379, December 1976.

[3] J. Camin and R. Sokal. A method for deducing branching sequences in phylogeny evolution. *Evolution*, 19:311–326, 1965.

[4] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[5] D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21:19–28, 1991.

[6] J. Meidanis and E. G. Munuera. A linear time algorithm for binary phylogeny using PQ-trees. Unpublished manuscript, 1995.