Incomplete Directed Perfect Phylogeny in Linear Time

Giulia Bernardini\textsuperscript{1}, Paola Bonizzoni\textsuperscript{1}, and Paweł Gawrychowski\textsuperscript{2}

\textsuperscript{1}DISCo, Università degli Studi Milano - Bicocca, Italy
\textsuperscript{2}Institute of Computer Science, University of Wrocław, Poland

Abstract

Reconstructing the evolutionary history of a set of species is a central task in computational biology. In real data, it is often the case that some information is missing: the Incomplete Directed Perfect Phylogeny (IDPP) problem asks, given a collection of species described by a set of binary characters with some unknown states, to complete the missing states in such a way that the result can be explained with a perfect directed phylogeny. Pe’er et al. proposed a solution that takes $\tilde{O}(nm)$ time for $n$ species and $m$ characters. Their algorithm relies on pre-existing dynamic connectivity data structures: a computational study recently conducted by Fernández-Baca and Liu showed that, in this context, complex data structures perform worse than simpler ones with worse asymptotic bounds.

This gives us the motivation to look into the particular properties of the dynamic connectivity problem in this setting, so as to avoid the use of sophisticated data structures as a blackbox. Not only are we successful in doing so, and give a much simpler $\tilde{O}(nm)$-time algorithm for the IDPP problem; our insights into the specific structure of the problem lead to an asymptotically faster algorithm, that runs in optimal $O(nm)$ time.
1 Introduction

A rooted phylogenetic tree models the evolutionary history of a set of species: the leaves are in a one-to-one correspondence with the species, all of which have a common ancestor represented by the root. A way of describing the species is by a set of characters that can assume several possible states, so that each species is described by the states of its characters. Such a representation is naturally encoded by a matrix $A$, $a_{i,j}$ being the state of character $j$ in species $i$.

When, for each possible character state, the set of all nodes that have the same state induces a connected subtree, a phylogeny is called perfect. The problem of reconstructing a perfect phylogeny from a set of species is known to be linearly-solvable in the case when the characters are binary [11], and it is NP-hard in the general case [2]. A popular variant of binary perfect phylogeny requires that the characters are directed, that is, on any path from the root to a leaf a character can change its state from 0 to 1, but the opposite cannot happen [5].

In this paper, we study the Incomplete Directed Perfect Phylogeny problem (IDPP for short) introduced by Pe'er et al. [19], assuming that the characters are binary, directed, and can be gained only once. The input of this problem is a matrix of character vectors in which some character states are unknown, and the question is whether it is possible to complete the missing states in such a way that the result can be explained with a directed perfect phylogeny.

Related work. Besides being relevant in its own right [1,17,18,21,23], the problem of handling phylogenies with missing data arises in various tasks of computational biology, like resolving genotypes with some missing information into haplotypes [16] and inferring tumor phylogenies from single-cell sequencing data with mutation losses [20]. A generalization of the perfect phylogeny model where a character can be gained only once and can be lost at most $k$ times, called the $k$-Dollo model [3,4,6,12], has also been extensively studied. It should be clear that different and efficient solutions for the IDPP problem may highlight novel approaches for the above mentioned computational frameworks.

The approach of Pe'er et al. [19] to the IDPP problem is graph theoretic: their algorithm relies on maintaining the connected components of a graph under a sequence of edge deletions. The use of pre-existing dynamic connectivity data structures for this purpose is the bottleneck in the overall time complexity. A connectivity data structure is fully-dynamic when both edge insertion and deletion are allowed, and decremental when only edge deletion is considered. A long line of results brought down the computational time required for updating the data structure after edge insertions and/or deletions, and for answering connectivity queries, to roughly logarithmic: the following table summarizes the results for both fully-dynamic and decremental connectivity on a graph consisting of $N$ nodes and $M$ edges. For fully-dynamic connectivity we report the update time required for a single edge insertion or deletion, while for decremental connectivity we report the overall time required to eventually delete all of the edges. All of the listed results, except for [13], assume that edge deletions can be interspersed with connectivity queries. The algorithm of Henzinger et al. [13], in contrast, deletes edges in batches ($b_0$ is the number of batches that do not result in a new component) and connectivity queries can be only asked between one batch of deletions and another.
Fully-Dynamic | Update time | Query time
---|---|---
Holm et al. [14] | \(O(\log^2 N)\), amortized | \(O(\log N/\log \log N)\)
Gibb et al. [10] | \(O(\log^4 N)\), worst case | \(O(\log N/\log \log N)\) w.h.p.
Huang et al. [15] | \(O(\log N(\log N)^2)\), expected amortized | \(O(\log N/\log \log \log N)\)

Decremental | Total update time | Query time
---|---|---
Even et al. [8] | \(O(MN)\) | \(O(1)\)
Thorup [24] | \(O(M \log^2 (N^2/M) + N \log^3 N \log \log N)\), expected | \(O(1)\)
Henzinger et al. [13] | \(O(N^2 \log N + b_0 \min\{N^2, M \log N\})\) | \(O(1)\)

By plugging in an appropriate dynamic connectivity structure, the worst case running time of the approach of Pe’er et al. [19], given a matrix describing \(n\) species and \(m\) characters, becomes deterministic \(O(nm \log^2 (n + m))\) (using fully dynamic connectivity structure of Holm et al. [14]), expected \(O(nm \log((n + m)^2/nm) + (n + m) \log^3(n + m) \log \log(n + m))\) (using decremental connectivity structure of Thorup [24]), expected \(O(nm \log(n + m) \log \log(n + m)^2)\) (using fully dynamic connectivity structure of Huang et al. [15]), or deterministic \(O((n + m)^2 \log(n + m))\) (using decremental structure of Henzinger et al. [13]). This should be compared with a lower bound of \(\Omega(nm)\), following from the work of Gusfield on directed binary perfect phylogeny [11] (under the natural assumption that the input is given as a matrix). For \(n = m\), the second algorithm achieves this lower bound at the expense of randomisation (and being very complicated), while for the general case the asymptotically fastest solution is still at least one log factor away from the lower bound.

Inspecting the algorithm of Pe’er et al. [19], we see that it operates on bipartite graphs and only needs to deactivate nodes on one of the sides. It seems plausible that some of the known dynamic connectivity structures are actually asymptotically more efficient on such instances. However, all of them are very complex (with the result of Holm et al. [14] being the simplest, but definitely not simple), and this is not clear. Furthermore, recently Fernández-Baca and Liu [9] performed an experimental study of the algorithm of Pe’er et al. for IDPP [19] with the aim of assessing the impact of the underlying dynamic graph connectivity data structure on their solution. Specifically, they tested the use of the data structure of Holm et al. [14] against a simplified version of the same method, and showed that, in this context, simple data structures perform better than more sophisticated ones with better asymptotic bounds.

**Our results and techniques.** We are motivated to look for simple, ad-hoc methods that make use of the properties of the decremental connectivity as used in IDPP. In this case, the graph is bipartite, and the required updates are vertex deletions from just one of the two sides. We thus start by describing a simple data structure that dynamically maintains the connected components of a bipartite graph with \(N\) nodes on each side, whilst vertices are removed from one side of the graph.

The starting point for our solution is an application of a version of the sparsification technique of Eppstein et al. [7]: we define a hierarchical decomposition of the graph, and maintain a forest representing the connected components of each subgraph in this decomposition. Recall that the original description of this technique focused on inserting and deleting edges, while we are interested in deleting nodes (and only from one side of the graph). Therefore, the decomposition needs to be appropriately tweaked for this particular use case. This allows us to obtain an extremely simple data structure with \(O(N^2 \log N)\) total update time, which we show to imply an \(O(nm \log n)\) algorithm for IDPP.
The main technical part of our paper refines this solution to shave the logarithmic factor and thus obtain an asymptotically optimal algorithm. We stress that while Eppstein et al. [7] did manage to avoid paying any extra log factors by applying a more complex decomposition of the graph than a complete binary tree (used in the conference version of their paper), this does not seem to translate to our setting, as we operate on the nodes instead of the edges. The high-level idea is to amortize the time spent on updating the forest representing the components of every subgraph with the progress in disconnecting its nodes, and re-use the results from the subgraph on the previous level of the decomposition to update the subgraph on the next level. As a consequence, the IDPP problem can be solved in time linear in the input size:

**Theorem 1.** Given an incomplete matrix \(A_{n \times m}\), the IDPP problem can be solved in time \(O(nm)\).

Under the natural assumption that the input is given as a matrix, this is asymptotically optimal [11].

**Paper organization.** In Section 2 we provide a description of the algorithm of Pe’er et al. [19] and a series of preliminary observations. In Section 3 we show a simple and self-contained dynamic connectivity data structure that implies an \(O(nm \log n)\) time solution for the IDPP problem for an incomplete matrix \(A_{n \times n}\). Finally, in Section 4 we present the main result of this paper and describe a dynamic connectivity data structure that implies a linear-time algorithm for IDPP.

## 2 Preliminaries

**Basic definitions.** Let \(G = (V, E)\) be a graph. The subgraph induced by \(V' \subseteq V\) is the graph \(G_{V'} = (V', E \cap (V' \times V'))\). We say that a forest \(F = (V, E)\) represents the connected components of \(G = (V, E)\) when the connected components of \(F\) and \(G\) are the same (note that we do not require that \(E' \subseteq E\)). Throughout the paper, we will use the term node for trees, and vertex for other graphs. We denote by \(S = \{s_1, \ldots, s_n\}\) the set of species and by \(C = \{c_1, \ldots, c_m\}\) the set of characters. A matrix of character states \(A_{n \times m} = [a_{ij}]_{n \times m}\), where each entry is a state from \(\{0, 1, ?\}\) and the rows correspond to the species, is said to be incomplete. The state \(a_{ij}\) of a character \(j\) for a species \(i\) is one, zero or ? depending on whether character \(j\) is present, absent or unknown for species \(i\). A completion \(B_{n \times m}\) of such \(A_{n \times m}\) is obtained by replacing the ? entries of \(A_{n \times m}\) with either 0 or 1: formally, \(B_{n \times m}\) is a binary matrix with entries \(b_{ij} = a_{ij}\) for each \(i, j\) such that \(a_{ij} \neq ?\).

A phylogenetic rooted tree \(T\) for a binary matrix \(B_{n \times m}\) has the \(n\) species of \(S\) at the leaves, and there is a surjection from the set of characters \(C\) to the internal nodes of \(T\) such that, if a character \(c_j\) is associated with a node \(x\), then \(s_i\) belongs to the leaf set of the subtree rooted at \(x\) if and only if \(b_{ij} = 1\). In other words, all and only the species in a subtree associated with a character \(c_j\) have the character \(c_j\). We say that an incomplete matrix admits a phylogenetic tree if there exists a completion of the matrix that has such a tree. The Incomplete Directed Perfect Phylogeny problem (IDPP for short), introduced by Pe’er et al. in [19], asks, given an incomplete matrix \(A\), to find a phylogenetic tree for \(A\), or determine that no such tree exists.

For a character \(c_j\), the 1-set (resp. 0-set and ?-set) of \(c_j\) in an incomplete matrix \(A\) is the set of species \(\{s_i|a_{ij} = 1\}\) (resp. \(a_{ij} = 0\) and \(a_{ij} = ?\)). For a subset \(S' \subseteq S\) of species, a character \(c\) is \(S'\)-semuniiversal in \(A\) if its 0-set does not intersect \(S'\), that is, if \(A[s, c] \neq 0\) for all \(s \in S'\). It is convenient to represent the character state matrix as a graph: the vertices are \(V = S \cup C\) and the edges are \(S \times C\), partitioned into \(E_1 \cup E_2 \cup E_0\), with \(E_x = \{(s_i, c_j)|a_{ij} = x\}\) for \(x \in \{0, 1, ?\}\). The edges of \(E_1, E_2, E_0\) are called solid, optional, and forbidden, respectively. We denote by \(G(A) = (S \cup C, E_1)\) the bipartite graph consisting only of the solid edges.
Previous solutions. The existence of a phylogenetic tree for $\mathcal{A}$ is linked with the existence, in its graph representation, of a subset of edges with certain properties. Specifically, Pe’er et al. show that finding a subset $D \subseteq (E_1 \cup E_r)$ such that $E_1 \subseteq D$ and $(S \cup C, D)$ is $\Sigma$-free (where a $\Sigma$ is a path consisting of four edges induced by three vertices from $S$ and two vertices from $C$), or determining that such $D$ does not exist, is equivalent to solving the IDPP problem for $\mathcal{A}$.

Pe’er et al. proposed two algorithms for solving the IDPP problem, both working on the graph representation of $\mathcal{A}$ and relying on some graph dynamic connectivity data structure, the main difference between the two being the data structure they use. For ease of presentation, in what follows we will only consider the algorithm they refer to as Alg _A. The algorithm relies on the following key properties: if an incomplete matrix $\mathcal{A}$ admits a phylogenetic tree, and $c$ is a $S$-semiuniversal character (meaning that there are no $0$s in its column), then the incomplete matrix obtained by setting to $1$ all of the entries of column $c$ still admits a phylogenetic tree. Moreover, given a partition $(K_1, \ldots, K_r)$ of $S \cup C$ where each $K_i$ is a connected component of $G(\mathcal{A})$, the incomplete matrix obtained by setting to $0$ all entries corresponding to the edges between $K_i$ and $K_j$, for $i \neq j$, still admits a phylogenetic tree. Then, there is no interaction between the species and characters belonging to different connected components, and the whole reasoning can be repeated on each such component separately.

We denote by $S(K)$ and $C(K)$ the set of species and characters, respectively, of a connected component $K$ of $G(\mathcal{A})$; $\mathcal{A}|_K$ denotes the submatrix of $\mathcal{A}$ corresponding to the species and characters in $K$. Deactivating a character $c$ in $G(\mathcal{A})$ consists in deleting $c$ together with all its incident edges. At a high level, Alg _A works as follows. At each step, for each connected component $K_i$ of $G(\mathcal{A})$, it computes the $S(K_i)$-semiuniversal characters. If, for some $K_i$, no $S(K_i)$-semiuniversal character exists, it can be proven that, for any $D \subseteq (E_1 \cup E_r)$ such that $E_1 \subseteq D$, the graph $(S \cup C, D)$ is not $\Sigma$-free, therefore the process halts and reports that $\mathcal{A}$ does not admit a phylogenetic tree. Otherwise, it sets to $1$ all of the entries of $\mathcal{A}|_{K_i}$ corresponding to the $S(K_i)$-semiuniversal characters, and sets to $0$ the entries of $\mathcal{A}$ between vertices that lay in different connected components. It then deactivates all of the $S(K_i)$-semiuniversal characters and updates the connected components of $G(\mathcal{A})$ using some dynamic connectivity data structure.

Algorithm 1 summarizes the high-level structure of Alg _A: for the sake of clarity, we only included the steps that compute the information needed for determining whether $\mathcal{A}$ has a phylogenetic tree, and we left out the operations that actually reconstruct the tree. A complete pseudocode and a proof of correctness of the algorithms can be found in [19].

Algorithm 1: The high-level structure of Alg _A [19].

```
1 while there is at least one character in $G(\mathcal{A})$ do
2     Find the connected components of $G(\mathcal{A})$
3     for each connected component $K_i$ of $G(\mathcal{A})$ with at least one character do
4         Compute the set $U$ of all characters in $K_i$ which are $S(K_i)$-semiuniversal in $\mathcal{A}$
5         if $U = \emptyset$ then return FALSE
6             Deactivate every $c \in U$
7     return TRUE
```

Preliminary results. Our goal is to improve Alg _A by optimizing its bottleneck, that is maintaining the connected components of $G(\mathcal{A})$. We will represent the connected components of a bipartite graph $G$ using the following lemma, and call the resulting representation a list-representation of $G$. 
Lemma 2. The connected components of a bipartite graph $G = (S \cup C, E)$ can be represented in $O(|S| + |C|)$ space so that, given a vertex, we can access its component, including the size and a pointer to the list of species and characters inside, in constant time, and move a vertex to another component (or remove it from the graph) also in constant time.

Proof. Each component of $G$ is represented by a doubly-linked list of its vertices (more precisely, a list of species and a list of characters), and also stores the size of the list. An array of length $n + m$, indexed by the vertices of $G$, stores a pointer from each vertex to its component and another pointer from each vertex to its position in the list of that component. The components are, in turn, organised in a doubly-linked list. Such representation takes space linear in the number of vertices and allows us to access all the required information in constant time. Further, removing or moving a vertex to another component takes constant time.

Given a list-representation of $G$, we represent its connected components with another graph $F = (V, E')$ consisting of rooted stars as follows. For each component $K$, we define the central vertex $v \in K$ to be the first vertex on the list of $K$. Then, we add an edge $(u, v)$ to $E'$, for any $u \in K$ with $u \neq v$. This construction can be implemented in $O(|V|)$ time. Observe that we only guarantee that the connected components of $G$ and $F$ are the same, but $E'$ is not required to only consist of the edges of $G$. We can use the list-representation of $G$ to simulate access to the adjacency lists of $F$ without constructing it explicitly, as stated by the following lemma.

Lemma 3. Given a bipartite graph $G = (S \cup C, E)$ and a list-representation of $G$, the access to the adjacency lists of a star forest $F$ representing the connected components of $G$ can be simulated in constant time without constructing $F$ explicitly.

Proof. To access the adjacency list of a vertex $v$ we first look up its component $K$ and retrieve the first vertex $u$ on the list of $K$. By Lemma 2, this operation requires constant time. If $u = v$, then the adjacency list of $v$ is the list of vertices of $K$ stored in the list-representation of $G$. Otherwise, the adjacency list of $v$ consists only of a single vertex $u$.

We are interested in solving the following special case of decremental connectivity:

Problem: $(N_\ell, N_r)$-DC
Input: a bipartite graph $G = (S \cup C, E)$ with $N_\ell = |S|$ and $N_r = |C|$.
Update: deactivate a character $c \in C$.
Query: return the connected components of the subgraph induced by $S$ and the remaining characters.

When analysing the complexity of $(N_\ell, N_r)$-DC, we allow preprocessing the input graph $G$ in $O(N_\ell N_r)$ time, and assume that all characters are eventually deactivated when analysing the total update time. We can of course deactivate multiple characters at once by deactivating them one-by-one.

The overall time complexity of Algorithm 1 depends on the complexity of $(N_\ell, N_r)$-DC as follows.

Lemma 4. Consider an $n \times m$ incomplete matrix $A$. If the $(n, m)$-DC problem can be solved in $f(n, m)$ total update time and $g(n, m)$ query time, then the IDPP problem can be solved for $A$ in time $O(nm + f(n, m) + \min\{n, m\} \cdot g(n, m))$.  

5
Proof. There are three nontrivial steps in every iteration of the while loop: finding the connected components in line 2, computing the semiuniversal characters of every connected component in line 4, and finally deactivating characters in line 6. Every character is deactivated at most once, so the overall complexity of all deactivations is $O(f(n,m))$. We claim that in every iteration of the while loop, except possibly for the very last, (1) at least one character is deactivated, and (2) there exist two species that cease to belong to the same connected component. (1) is immediate, as otherwise we have a connected component $K_i$ with no $S(K_i)$-semiuniversal characters and the algorithm terminates. To prove (2), assume otherwise, then we have a connected component $K_i$ such that $S(K_i)$ does not change after deactivating all $S(K_i)$-semiuniversal characters. But then in the next iteration the set of $S(K_i)$-semiuniversal characters is empty and the algorithm terminates. (1) and (2) together imply that the number of iterations is bounded by $\min\{n,m\}$. The overall complexity of finding the connected components is thus $O(\min\{n,m\} \cdot g(n,m))$.

It remains to bound the overall complexity of computing the semiuniversal characters by $O(nm)$. This has been implicitly done in [19, proof of Theorem 12], but we provide a full explanation for completeness. For every character $c \in C$, we maintain the count of solid and optional edges connecting $c$ (in the graph representation of $A$) with the species that belong to its same connected component (of $G(A)$). Assuming that we can indeed maintain these counts, in every iteration all the semiuniversal characters can be generated in $O(m)$ time, so in $O(\min\{n,m\} \cdot m) = O(nm)$ overall time.

To update the counts, consider a connected component $K$ that, after deactivating some characters, is split into possibly multiple smaller components $K_1, K_2, \ldots, K_k$. Note that we can indeed gather such information in $O(n+m)$ time, assuming access to a representation of the connected components before and after the deactivation. We assume that the connected components are maintained with the list-representation described in Lemma 2, and therefore we can access a list of the vertices in every $K_i$. Then, we consider every pair $i, j \in \{1, 2, \ldots, k\}$ such that $i \neq j$, $C(K_i) \neq \emptyset$ and $S(K_j) \neq \emptyset$. We iterate over every $c \in K_i$ and $s \in K_j$, and if $(s,c)$ is an edge in the graph of $A$ (observe that it cannot be a solid edge, as $K_i$ and $K_j$ are distinct connected components) we decrease the count of $c$. By first preparing lists of components $K_i$ such that $C(K_i) \neq \emptyset$ and $S(K_i) \neq \emptyset$, this can be implemented in time bounded by the number of considered possible edges $(s,c)$, and every such possible edge is considered at most once during the whole execution. Therefore, the overall complexity of maintaining the counts is $O(nm)$. Additionally, we need $O(nm)$ time to initialise the $(n,m)$-DC structure.

Before we proceed to design an efficient solution for the $(N, N)$-DC problem, we first show that it is in fact enough to consider the $(N, N)$-DC problem.

Lemma 5. Assume that the $(N, N)$-DC problem can be solved in $f(N)$ total update time and $g(N)$ query time. Then, for any $N' \geq N$, both the $(N, N')$-DC problem and the $(N', N)$-DC problem can be solved in $O(N'/N \cdot f(N))$ total update time and $O(N'/N \cdot g(N))$ query time.

Proof. We first consider the $(N, N')$-DC problem. We create $\lfloor N'/N \rfloor$ instances of $(N, N)$-DC by partitioning $C$ into groups of $N$ vertices (except for the last group that might be smaller). In each instance we have the same set of species $S$. Deactivating a character $c \in C$ is implemented by deactivating it in the corresponding instance of $(N, N)$-DC. Overall, this takes $O(N'/N \cdot f(N))$ time. Upon a query, we query all the instances in $O(N'/N \cdot g(N))$ time. The output of each instance can be converted to a star forest representing the connected components in $O(N)$ time. We take the union of all these forests to obtain an auxiliary graph on at most $\lfloor N'/N \rfloor \cdot (N - 1) = O(N')$ edges, and find its connected components in $O(N')$ time. Assuming that $f(N) \geq N$, this takes $O(N'/N \cdot f(N))$ overall time and gives us the connected components of the whole graph.
Now we consider the \((N', N)\)-DC problem. We create \([N'/N]\) instances of \((N, N)\)-DC by partitioning \(S\) into groups of \(N\) vertices, and in each instance we have the same set of characters \(C\). Thus, deactivating a character \(c \in C\) is implemented by deactivating it in every instance. Overall, this takes \(O(N'/N \cdot f(N))\) time. A query is implemented exactly as above by querying all the instances and combining the results in \(O(N'/N \cdot f(N))\) time.

\[\square\]

3 \((N, N)\)-DC in \(O(N^2 \log N)\) Total Update Time and \(O(N)\) per Query

Our solution for the \((N, N)\)-DC problem is based on a hierarchical decomposition of \(G\) into multiple smaller subgraphs as in the sparsification technique of Eppstein et al. \cite{Eppstein98} (as mentioned in the introduction, appropriately tweaked for our use case). The decomposition is represented by a complete binary tree \(DT(G)\) of depth \(\log N\). We identify the leaves of \(DT(G)\) with the characters \(C\). Each node \(v\) corresponds to the set of characters \(C_v\) identified with the leaves in the subtree of \(v\), and is responsible for the subgraph \(G_v\) of \(G\) induced by \(C_v\) and the whole set of species \(S\). Thus, the root is responsible for the whole \(G\), see Figure 1. Each node \(v\) explicitly maintains a list-representation of the connected components of \(G_v\), denoted \(\text{components}(v)\). We stress that, while \(\text{components}(v)\) is explicitly maintained, we do not explicitly store \(G_v\) at every node \(v\). The initial preprocessing required to construct \(DT(G)\) together with \(\text{components}(v)\) for every node \(v\), given \(G\), takes \(O(B^2)\) time by the following argument. First, we construct \(\text{components}(v)\) for every leaf \(c\). This can be done in \(O(B)\) time per leaf by simply iterating the neighbours of \(c\) in \(G\). Second, we proceed bottom-up and compute \(\text{components}(v)\) for every inner node \(v\) in \(O(B)\) time using the following lemma.

**Lemma 6.** Let \(v\) be an inner node of \(DT(G)\), and \(v_\ell, v_r\) be its children. Given \(\text{components}(v_\ell)\) and \(\text{components}(v_r)\) we can compute \(\text{components}(v)\) in \(O(B)\) time.

**Proof.** We construct star forests representing the connected components of \(\text{components}(v_\ell)\) and \(\text{components}(v_r)\) in \(O(B)\) time and take their union. Then we find the connected components of this union in \(O(B)\) time and save them as \(\text{components}(v)\). \(\square\)

We proceed to explain how to solve the \((N, N)\)-DC problem in \(O(N \log N)\) time per update and \(O(N)\) time per query. The query simply returns \(\text{components}(r)\), where \(r\) is the root of \(DT(G)\). The update is implemented as follows. Deactivating a character \(c\) possibly affects \(\text{components}(v)\)
for all ancestors \( v \) of the leaf corresponding to \( c \). In particular, \( \text{components}(c) \) becomes a collection of isolated nodes and can be recomputed in \( \mathcal{O}(1 + |S|) = \mathcal{O}(N) \) time. We iterate over all proper ancestors \( v \), starting from the parent of \( c \). For each such \( v \), let \( v_L \) and \( v_R \) denote its left and right child, respectively. We can assume that \( \text{components}(v_L) \) and \( \text{components}(v_R) \) have been already correctly updated. We compute \( \text{components}(v) \) from \( \text{components}(v_L) \) and \( \text{components}(v_R) \) by applying Lemma 6 in \( \mathcal{O}(N) \) time. When summed over all the ancestors, the update time becomes \( \mathcal{O}(N \log N) \), so \( \mathcal{O}(N^2 \log N) \) over all deactivations.

By Lemmas 4 and 5, this implies that, given an incomplete matrix \( A_{n \times m} \), the IDPP problem can be solved in time \( \mathcal{O}(nm \log \min\{n, m\}) \) without using any dynamic connectivity data structure as a blackbox.

### 4 \((N, N)\)-DC in \( \mathcal{O}(N^2) \) Total Update Time and \( \mathcal{O}(N) \) Time per Query

Our faster solution is also based on a hierarchical decomposition \( \text{DT}(G) \) of \( G \). As before, every node \( v \) stores \( \text{components}(v) \), so a query simply returns \( \text{components}(r) \). The difference is in implementing an update. We observe that, if for some ancestor \( v \) of the leaf corresponding to \( c \), the only change to \( \text{components}(v) \) is removing \( c \) from its connected component, then this also holds for all of the subsequent ancestors and they can be updated in constant time each. This suggests that we should try to amortise the cost of an update with the progress in splitting \( \text{components}(v) \) into smaller components.

We will need to compare the situation before and after the update, and so introduce the following notation. A node \( v \) of \( \text{DT}(G) \) is responsible for the subgraph \( G_{v_k} \) before the update and for the subgraph \( G'_{v_k} \) after the update; \( \text{components}(v) \) and \( \text{components}'(v) \) denote the connected components of \( G_v \) and \( G'_{v_k} \), respectively. The crucial observation is that \( \text{components}'(v) \) is obtained from \( \text{components}(v) \) by removing \( c \) from its connected component and, possibly, splitting this connected component into multiple smaller ones, while leaving the others intact.

Deactivating a character \( c \) begins with updating naively \( \text{components}(c) \) in \( \mathcal{O}(N) \) time. Then we iterate over the ancestors of \( c \) in \( \text{DT}(G) \). Let \( v_i+1 \) be the currently considered ancestor, \( v_i \) the ancestor considered in the previous iteration, and \( u_i \) be the other child of \( v_{i+1} \) (sibling of \( v_i \)). Let the component of \( G_{v_i} \) containing \( c \) be \( K \). As observed above, the components of \( G'_{v_i} \) are the same as the components of \( G_{v_i} \) except that \( K \) is replaced by possibly multiple components \( K_1, K_2, \ldots, K_k \), where \( \bigcup_{j=1}^{k} K_j = K \setminus \{c\} \). If \( k = 1 \) then we trivially remove \( c \) from its connected component in every \( G_{v_j} \), for \( j = i + 1, i + 2, \ldots \) and terminate the update, so we can assume that \( k \geq 2 \). We further assume that, after having updated the components of \( G_{v_i} \), we obtained a list of pointers to \( K_1, K_2, \ldots, K_k \). Let \( L \) be the connected component of \( c \) in \( G_{v_{i+1}} \), with \( K \subseteq L \) because the subgraphs are monotone with respect to inclusion on any leaf-to-root path. Now the goal is to transform \( G_{v_{i+1}} \) into \( G'_{v_{i+1}} \), to update its components (using \( \text{components}'(v_i) \) and \( \text{components}(u_i) \)), and additionally to obtain a list of pointers to the components obtained by splitting \( L \). See Figure 2 for an illustration.

We start by initialising \( G'_{v_{i+1}} \) to be \( G_{v_{i+1}} \), and by removing \( c \) from \( L \). As in the proof of Lemma 6, we will work with an auxiliary graph consisting of the union of two star forests representing the connected components of \( G'_{v_i} \) and \( G_{u_i} \), respectively. However, instead of explicitly constructing these forests, we simulate access to the adjacency lists of every vertex in both forests using \( \text{components}'(v_i) \) and \( \text{components}(u_i) \), as explained in the proof of Lemma 3. In turn, this allows us to simulate access to the adjacency list of every vertex in the auxiliary graph. See Figure 3 for an example of the auxiliary graph.

By renaming the components we can assume that \( |K_1| \geq |K_2|, |K_3|, \ldots, |K_k| \). We will visit the
vertices of $L$ in order to determine the new connected components after the removal of $c$: when doing so, we will use different colours to represent vertices whose new connected component contains $K_1$ (red), vertices whose new component is different from the one of $K_1$ (black) and vertices whose new component is still unknown (white). Initially, the vertices of $K_1$ are red and all of the other vertices of the auxiliary graph are white. This initialisation is done implicitly, meaning that we will assume that all the vertices of $K_1$ are red and the rest are white without explicitly assigning the colours, and whenever retrieving the colour of a node $u$ we first check if $u \in K_1$, and if so assume that it is red. This allows us to implement the initialisation in constant time instead of $O(N)$ time. We will perform the visit of $L$ by running the following search procedure from an arbitrarily chosen vertex of each $K_j$, for $j = 2, 3, \ldots, k$.

The search procedure run from a vertex $x$ first checks if $x$ is white, and immediately terminates otherwise. Then, it starts visiting the vertices of the connected component of $x$ in the auxiliary graph: at any moment, each vertex in such component is either white or red. As soon as the search encounters a red vertex, it is terminated and all the vertices visited in the current invocation are explicitly coloured red. Otherwise, the procedure has identified a new connected component $K'$ of $G'_{v_{i+1}}$. The vertices of $K'$ are removed from $L$, all vertices of $K'$ are coloured black in the auxiliary graph, and a new component $K'$ of $G'_{v_{i+1}}$ is created in $O(|K'|)$ time. Inspect Figure 3 for an example.

Figure 3: The auxiliary graph implicitly constructed for a node $v_{i+1}$ after deactivating $c_8$. Black edges are used for the star forest of $v_i$, grey edges for the star forest of $u_i$; an inner circle identifies the central vertices. $K_1$ is the rightmost component; $c_7$ is the next vertex to be considered, and it will eventually become red.
Lemma 7. The total time spent on all calls to the search procedure in the current iteration is \(O(|L| - |K_1|)\).

Proof. All vertices visited in the current iteration belong to \(L\). The search is terminated as soon as we encounter a red vertex, and all vertices of \(K_1\) are red from the beginning. Therefore, each run of the search procedure encounters at most one vertex of \(K_1\), and we can account for traversing the edge leading to this vertex separately paying \(O(k - 1) = O(|L| - |K_1|)\) overall. It remains to bound the number of other traversed edges. This is enough to bound the overall time of the traversal, because every edge is traversed at most twice, and the number of visited isolated vertices is at most \(k - 1 = O(|L| - |K_1|)\).

For any other edge \(e = \{u, v\}\), we have \(u, v \in L\) but \(u, v \notin K_1\). These edges can be partitioned into two forests by considering whether they originate from \(\text{components}(v_i)\) or \(\text{components}(u_i)\). Consequently, we must analyse the total number of edges in a union of two forests spanning \(L \setminus K_1\). But this is of course \(O(|L| - |K_1|)\), proving the lemma.

We now need to analyse the sum of \(|L| - |K_1|\) over all the iterations. Because \(\bigcup_{j=1}^{k} K_j \subseteq L\), we can split this expression into two parts:

1. \(|L \setminus \bigcup_{j=1}^{k} K_j|\),
2. \(\sum_{j=2}^{k} |K_j|\).

Because the sets \(L \setminus \bigcup_{j=1}^{k} K_j\) considered in different iterations are disjoint, the first parts sum up to \(O(n)\). It remains to bound the sum of the second parts. This will be done by the following argument. Consider an arbitrary \(G_v\) corresponding to a subgraph induced by all the species and a subset of \(2^d\) characters. Whenever its connected component \(K\) is split into smaller connected components \(K_1, K_2, \ldots, K_k\) after deactivating a character \(c\) in the subtree of \(v\), the second part \(\sum_{j=2}^{k} |K_j|\) is distributed among the vertices of \(\bigcup_{j=2}^{k} K_j\). That is, each node of \(\bigcup_{j=2}^{k} K_j\) pays 1. Observe that the size of the connected component containing such a node decreases by a factor of at least 2, because \(|K_2|, |K_3|, \ldots, |K_k| \leq |K|/2\). To bound the sum of second parts, we analyse the total cost paid by all the nodes of \(G_v\), due to deactivating the characters in the subtree of \(v\) (recall that in the end all such characters are deactivated).

Lemma 8. The total cost paid by the nodes of \(G_v\), over all \(2^d\) deactivations affecting \(v\), is \(O(N \cdot d)\).

Proof. We claim that in the whole process there can be at most \(2^{t+1}\) deactivations incurring a cost from \([N/2^{t+1}, N/2^t]\). Assume otherwise, then there exists a vertex \(x\) charged twice by such deactivations. As a result of the first deactivation, the size of the connected component containing \(x\) drops from less than \(N/2^t\) to below \(N/2^{t+1}\). Consequently, during the next deactivation that charges \(x\) the cost must be smaller than \(N/2^{t+1}\), a contradiction. As we have \(2^d\) deactivation overall, the total cost can be at most:

\[
\sum_{t=0}^{d} 2^{t+1} \cdot N/2^t = O(N \cdot d)
\]

as claimed.

To complete the analysis, we observe that there are \(N/2^d\) nodes of \(\text{DT}(G)\) such that we have \(2^d\) deactivations affecting \(v\). The sum of the second parts is thus:

\[
\sum_{d=0}^{\log n} N/2^d \cdot n \cdot d < N^2 \sum_{d=0}^{\infty} d/2^d = O(N^2).
\]
Overall, the total update time is hence $O(N^2)$. By Lemmas 4 and 5, it implies the following:

**Theorem 1.** Given an incomplete matrix $A_{n \times m}$, the IDPP problem can be solved in time $O(nm)$.

**References**

[1] Ali Bashir, Chun Ye, Alkes L Price, and Vineet Bafna. Orthologous repeats and mammalian phylogenetic inference. *Genome Research*, 15(7):998–1006, 2005.

[2] Hans L Bodlaender, Michael R Fellows, Michael T Hallett, H Todd Wareham, and Tandy J Warnow. The hardness of perfect phylogeny, feasible register assignment and other problems on thin colored graphs. *Theoretical Computer Science*, 244(1-2):167–188, 2000.

[3] Paola Bonizzoni, Chiara Braghin, Riccardo Dondi, and Gabriella Trucco. The binary perfect phylogeny with persistent characters. *Theoretical Computer Science*, 454:51–63, 2012.

[4] Paola Bonizzoni, Simone Ciccolella, Gianluca Della Vedova, and Mauricio Soto. Beyond perfect phylogeny: Multisample phylogeny reconstruction via ilp. In 8th ACM-BCB, pages 1–10, 2017.

[5] Joseph H Camin and Robert R Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, pages 311–326, 1965.

[6] Mohammed El-Kebir. Sphyr: tumor phylogeny estimation from single-cell sequencing data under loss and error. *Bioinformatics*, 34(17):i671–i679, 2018.

[7] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[8] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981.

[9] David Fernández-Baca and Lei Liu. Tree compatibility, incomplete directed perfect phylogeny, and dynamic graph connectivity: An experimental study. *Algorithms*, 12(3):53, 2019.

[10] David Gibb, Bruce Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *arXiv:1509.06464*, 2015.

[11] Dan Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21(1):19–28, 1991.

[12] Dan Gusfield. Persistent phylogeny: a galled-tree and integer linear programming approach. In 6th ACM-BCB, pages 443–451, 2015.

[13] Monika Rauch Henzinger, Valerie King, and Tandy Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.

[14] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[15] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log \log n)^2)$ amortized expected time. In 28th SODA, pages 510–520. SIAM, 2017.

[16] Gad Kimmel and Ron Shamir. The incomplete perfect phylogeny haplotype problem. *Journal of Bioinformatics and Computational Biology*, 3(02):359–384, 2005.
[17] B. Kirkpatrick and K. Stevens. Perfect phylogeny problems with missing values. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(5):928–941, 2014.

[18] Masato Nikaido, Alejandro P Rooney, and Norihiro Okada. Phylogenetic relationships among cetartiodactyls based on insertions of short and long interpersed elements: hippopotamuses are the closest extant relatives of whales. *Proceedings of the National Academy of Sciences*, 96(18):10261–10266, 1999.

[19] Itsik Pe’er, Tal Pupko, Ron Shamir, and Roded Sharan. Incomplete directed perfect phylogeny. *SIAM Journal on Computing*, 33(3):590–607, 2004.

[20] Gryte Satas, Simone Zaccaria, Geoffrey Mon, and Benjamin J Raphael. Scarlet: Single-cell tumor phylogeny inference with copy-number constrained mutation losses. *Cell Systems*, 10(4):323–332, 2020.

[21] R. V. Satya and A. Mukherjee. The undirected incomplete perfect phylogeny problem. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 5(4):618–629, 2008.

[22] Yossi Shiloach and Uzi Vishkin. An $o(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[23] Kristian Stevens and Dan Gusfield. Reducing multi-state to binary perfect phylogeny with applications to missing, removable, inserted, and deleted data. In *10th WABI*, pages 274–287. Springer, 2010.

[24] Mikkel Thorup. Decremental dynamic connectivity. *Journal of Algorithms*, 33(2):229–243, 1999.