UNIVERSITÀ DEGLI STUDI DI TRIESTE

Incomplete Directed Perfect Phylogeny in Linear Time

This is the peer reviewed (post-print) version of the following article:

Original
Incomplete Directed Perfect Phylogeny in Linear Time / Bernardini, G., Bonizzoni, P., Gawrychowski, P.. - 12808:(2021), pp. 172-185. (17th International Symposium on Algorithms and Data Structures, WADS 2021 online 2021) [10.1007/978-3-030-83508-8_13].

Availability:
This version is available at: 11368/3019970 since: 2022-05-12T11:12:53Z

Publisher:
Springer Science and Business Media Deutschland GmbH

Published
DOI:10.1007/978-3-030-83508-8_13

Terms of use:

Some rights reserved. More details in the PostPrint rights section below.

License:
Digital Rights Management not defined

Publisher copyright

-.-

(Article begins on next page)

21 October 2026

arTs Università degli Studi di Trieste
Archivio della ricerca – postprint

Incomplete Directed Perfect Phylogeny in Linear Time

Giulia Bernardini[1,3](✉), Paola Bonizzoni[1], and Paweł Gawrychowski[2]

[1] Università degli Studi di Milano - Bicocca, Milano, Italy
[2] Institute of Computer Science, University of Wrocław, Wrocław, Poland
[3] CWI, Amsterdam, The Netherlands
giulia.bernardini@cwi.nl

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 directed perfect phylogeny. Pe'er et al. [SICOMP 2004] proposed a solution that takes $\tilde{\mathcal{O}}(nm)$ time (the $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylog factors) 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 $\mathcal{O}(nm \log n)$-time algorithm for the IDPP problem; our insights into the specific structure of the problem lead to an asymptotically optimal $\mathcal{O}(nm)$-time algorithm.

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 standard 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 $\mathcal{A}$, $a_{i,j}$ 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].

© Springer Nature Switzerland AG 2021
A. Lubiw and M. Salavatipour (Eds.): WADS 2021, LNCS 12808, pp. 172–185, 2021.
https://doi.org/10.1007/978-3-030-83508-8_13

In this paper, we study the Incomplete Directed Perfect Phylogeny problem (IDPP for short) introduced by Pe'er et al. [20]. The input of this problem is a matrix of binary 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 [1,18,19,22,24], the problem of handling phylogenies with missing data is crucial in various tasks of computational biology, like resolving genotypes with some missing information into haplotypes [13,17] and inferring tumor phylogenies from single-cell sequencing data with mutation losses [21]; 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. A deep understanding of the IDPP problem leading to new efficient solutions may thus highlight novel approaches for all such important tasks.

The algorithm proposed in [20] solves the IDPP problem for a matrix of n species and m characters in $\tilde{\mathcal{O}}(nm)$ time[1] with a graph-theoretic approach. A crucial step of such algorithm is to maintain 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 existing results for a graph with N vertices 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 the edges. All the listed results, except for [14], assume that edge deletions can be interspersed with connectivity queries. The algorithm of Henzinger et al. [14], 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. [15]	$\mathcal{O}(\log^2 N)$, amortized	$\mathcal{O}(\log N/\log\log N)$
Gibb et al. [10]	$\mathcal{O}(\log^4 N)$, worst case	$\mathcal{O}(\log N/\log\log N)$ w.h.p.
Huang et al. [16]	$\mathcal{O}(\log N(\log\log N)^2)$, expected amortized	$\mathcal{O}(\log N/\log\log\log N)$
Decremental	Total update time	Query time
Even et al. [8]	$\mathcal{O}(MN)$	$\mathcal{O}(1)$
Thorup [25]	$\mathcal{O}(M\log^2(N^2/M) + N\log^3 N\log\log N)$, expected	$\mathcal{O}(1)$
Henzinger et al. [14]	$\mathcal{O}(N^2\log N + b_0\min\{N^2, M\log N\})$	$\mathcal{O}(1)$

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses polylog factors.

By plugging in a dynamic connectivity structure, the worst case running time of the algorithm of [20], given a matrix of n species and m characters, becomes deterministic $\mathcal{O}(nm \log^2(n+m))$ (using fully dynamic connectivity structure of Holm et al. [15]), expected $\mathcal{O}(nm \log((n+m)^2/nm) + (n+m)\log^3(n+m)\log\log(n+m))$ (using decremental connectivity structure of Thorup [25]), expected $\mathcal{O}(nm \log(n+m)(\log\log(n+m))^2)$ (using fully dynamic connectivity structure of Huang et al. [16]), or deterministic $\mathcal{O}((n+m)^2 \log(n+m))$ (using decremental structure of Henzinger et al. [14]). 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 algorithm of [20] using [25] 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.

A closer look to the algorithm of [20], that we describe in more details in Sect. 2, reveals that it operates on bipartite graphs and only deletes vertices 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. [15] 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 [20] 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. [15] 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 simpler, ad-hoc methods for the specific type of decremental connectivity that is used in IDPP: vertex deletion from just one side of a bipartite graph. We start by describing a simple structure that dynamically maintains the connected components of a bipartite graph with N vertices on each side, whilst vertices are removed from one of the sides. The starting point for our solution is an application of a particular version of the sparsification technique of Eppstein et al. [7]: we define a hierarchical decomposition of a bipartite 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 vertices (and only from one side of the graph). We thus tweak the decomposition for our particular use case, obtaining an extremely simple data structure with $\mathcal{O}(N^2 \log N)$ total update time, which we show to imply an $\mathcal{O}(nm \log n)$ 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 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 trans-

late to our setting, as we operate on vertices instead of edges. The high-level idea of our solution is to amortize the time spent on updating the forest representing the components of every subgraph with the progress in disconnecting its vertices, 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. Under the natural assumption that the input is given as a matrix, this is asymptotically optimal [11].

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

Paper Organization. In Sect. 2 we provide a description of the algorithm of Pe'er et al. [20] and a series of preliminary observations. In Sect. 3 we show a simple and self-contained decremental connectivity data structure that considers the removal of vertices from one side of a bipartite graph. This structure implies an $\mathcal{O}(nm \log n)$ time solution for the IDPP problem. Finally, in Sect. 4 we present our main result and describe a decremental connectivity data structure for removing vertices from one side of a bipartite graph that implies a linear-time algorithm for IDPP.

2 Preliminaries

Let $G = (V, E)$ be a graph. The subgraph induced by a subset of vertices $V' \subseteq V$ is $G_{V'} = (V', E \cap (V' \times V'))$. We say that a forest $F = (V, E')$ represents the connected components of a graph G when the connected components of F and G are the same (note that we do not require $E' \subseteq E$). 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 $\mathcal{A}_{n\times m} = [a_{ij}]_{n\times m}$, 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} is one, zero or ? depending on whether character j is present, absent or unknown for species i. A completion $\mathcal{B}_{n\times m}$ of such $\mathcal{A}_{n\times m}$ is obtained by replacing the ? entries of $\mathcal{A}_{n\times m}$ with either 0 or 1: formally, $\mathcal{B}_{n\times m}$ is a binary matrix with entries $b_{ij} = a_{ij}$ for each i, j such that $a_{ij} \neq ?$.

A *directed perfect phylogeny* for a binary matrix $\mathcal{B}_{n\times m}$ is a rooted tree $\mathcal{T}$ whose leaves are bijectively labelled by S and such that there is a surjection from the characters C to the internal nodes of $\mathcal{T}$ with the following property: if $c_j \in C$ is associated with a node x, then s_i is a leaf of the subtree rooted at x if and only if $b_{ij} = 1$. In particular, the term *directed* means that characters can be gained but not lost on any root-to-leaf path. We say that an incomplete matrix admits a directed perfect 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 in [20], asks, given an incomplete matrix $\mathcal{A}$, to find a directed perfect phylogenetic tree for $\mathcal{A}$, or determine that no such tree exists.

Algorithm 1: The high-level structure of Alg_A [20].

```
while there is at least one character in G(A) do
    Find the connected components of G(A)
    for each connected component K_i of G(A) with at least one character do
        Compute the set U of all characters in K_i which are
        S(K_i)-semiuniversal in A
        if U = ∅ then return FALSE
        Deactivate every c ∈ U
    return TRUE
```

The 1-set (resp. 0-set and ?-set) of a character c_j in an incomplete matrix $\mathcal{A}$ is the set of species $\{s_i | a_{ij} = 1\}$ (resp. $a_{ij} = 0$ and $a_{ij} = ?$). For a subset $S' \subseteq S$, a character c is S'-semiuniversal in $\mathcal{A}$ if its 0-set does not intersect S', that is, if $\mathcal{A}[s, c] \neq 0$ for all $s \in S'$. It is convenient to represent the character state matrix $\mathcal{A}$ as a graph: the vertices are $V = S \cup C$ and the edges are $S \times C$, partitioned into $E_1 \cup E_? \cup E_0$, with $E_x = \{(s_i, c_j) | a_{ij} = x\}$ for $x \in \{0, 1, ?\}$. The edges of $E_1, E_?, E_0$ are called *solid*, *optional*, and *forbidden*, respectively. We denote by $G(\mathcal{A}) = (S \cup C, E_1)$ the bipartite graph consisting only of the solid edges. A Σ is a subgraph induced by three vertices from S and two vertices from C, consisting of exactly four edges that form a path of length 4.

Previous solutions. Pe'er et al. [20] consider a graph representation of the input matrix $\mathcal{A}$, and show that finding a subset $D \subseteq (E_1 \cup E_?)$ such that $E_1 \subseteq D$ and $(S \cup C, D)$ is Σ-free, or determining that no such D exists, is equivalent to solving IDPP. Their main algorithm exploits this characterization and the following properties: (i) if $\mathcal{A}$ admits a phylogenetic tree, then so does the matrix obtained by setting to 1 all the entries of column c, for each S-semiuniversal c; (ii) 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 matrix obtained by setting to 0 all entries corresponding to edges between K_i and K_j, for $i \neq j$, admits a phylogenetic tree if $\mathcal{A}$ does; and (iii) if there is a component K_i with no $S(K_i)$-semiuniversal characters, then for any $D \subseteq (E_1 \cup E_?)$ such that $E_1 \subset D$, the graph $(S \cup C, D)$ is not Σ-free (and thus $\mathcal{A}$ has no phylogenetic tree). It follows that there is no interaction between the species and characters belonging to different connected components, and therefore 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 $\mathcal{A}$ consisting of the species and characters in K. *Deactivating* a character c in $G(\mathcal{A})$ consists in deleting c and all its incident edges. At a high level Alg_A, the main algorithm of [20], works as follows. At each step, for each connected component K_i of $G(\mathcal{A})$, it computes the set U of $S(K_i)$-semiuniversal characters. If $U = \emptyset$, because of property (iii) $\mathcal{A}$ does not admit a phylogenetic tree, and the process halts. Otherwise, it sets to 1 the entries of $\mathcal{A}|_{K_i}$ corresponding to U, and sets to 0

the entries of $\mathcal{A}$ between vertices that lay in different connected components. It then deactivates all the characters in U and updates the connected components of $G(\mathcal{A})$ using some dynamic connectivity structure. Algorithm 1 summarizes this process: 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 construct the tree.

2.1 Preliminary Results

Our goal is to improve Alg_A by optimizing its bottleneck, that is maintaining the connected components of $G(\mathcal{A})$. We start by describing a data structure that conveniently represents the connected components of a bipartite graph G.

Lemma 1. *The connected components of a bipartite graph $G = (S \cup C, E)$ can be represented in $\mathcal{O}(|S| + |C|)$ 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 a pointer from each vertex to its position in the list of its 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. □

We denote by $\mathsf{cc}(G)$ the data structure of Lemma 1, which encodes the connected components of G. A graph $F = (V, E')$ consisting of a forest of *rooted stars* [23] can be straightforwardly obtained from $\mathsf{cc}(G)$ as follows. For each component K, we define the central vertex $v \in K$ to be the head of the doubly-linked list of characters of K in $\mathsf{cc}(G)$. Then, we add an edge (u, v) to E', for any $u \in K$ with $u \neq v$. This construction can be implemented in $\mathcal{O}(|V|)$ time. Although we do not require E' to be a subset of the edges of G, by construction the connected components of F and G are the same. The useful property is that we can use $\mathsf{cc}(G)$ to simulate access to the adjacency lists of F without constructing it explicitly, as stated by the following lemma.

Lemma 2. *Given a bipartite graph $G = (S \cup C, E)$ and $\mathsf{cc}(G)$, the access to the adjacency lists of a forest of rooted stars F with the same connected components as G can be simulated in constant time without constructing F explicitly.*

Proof. To simulate the access to the adjacency list of a vertex v, we first look up its component K in $\mathsf{cc}(G)$ and retrieve the head u of the doubly-linked list of characters of K. By Lemma 1, this operation requires constant time. If $u = v$, then the adjacency list of v is the list of vertices of K stored in $\mathsf{cc}(G)$. Otherwise, the adjacency list of v consists only of a single vertex u. □

Our intent is to solve the following special case of decremental connectivity.

Problem: (N_ℓ, 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_ℓ, N_r)-DC, we allow preprocessing the input graph G in $\mathcal{O}(N_\ell N_r)$ time, and assume that all characters will be 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_ℓ, N_r)-DC as follows.

Lemma 3. *Consider an $n \times m$ incomplete matrix $\mathcal{A}$. 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 $\mathcal{A}$ in time $\mathcal{O}(nm + f(n, m) + \min\{n, m\} \cdot g(n, m))$.*

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 $\mathcal{O}(f(n, m))$. 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 $\mathcal{O}(\min\{n, m\} \cdot g(n, m))$.

It remains to bound the overall complexity of computing the semiuniversal characters by $\mathcal{O}(nm)$. This has been implicitly done in [20, 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 $\mathcal{A}$) with the species that belong to its same connected component of $G(\mathcal{A})$ (recall that $G(\mathcal{A})$ consists only of the solid edges of the graph representation of $\mathcal{A}$). Assuming that we can indeed maintain these counts, in every iteration all the semiuniversal characters can be generated in $\mathcal{O}(m)$ time, so in $\mathcal{O}(\min\{n, m\} \cdot m) = \mathcal{O}(nm)$ 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 $\mathcal{O}(n + m)$ 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 representation described in Lemma 1, 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 $\mathcal{A}$ (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 $\mathcal{O}(nm)$. Additionally, we need $\mathcal{O}(nm)$ time to initialise the (n, m)-DC structure. □

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

Lemma 4. *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 $\mathcal{O}(N'/N \cdot f(N))$ total update time and $\mathcal{O}(N'/N \cdot g(N))$ query time.*

Proof. We first consider the (N, N')-DC problem, in which $|S| < |C|$. We create $\lceil N'/N \rceil$ instances of (N, N)-DC by partitioning C into groups of N vertices (the last group 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 $\mathcal{O}(N'/N \cdot f(N))$ time. To answer a query, we first query all the instances in $\mathcal{O}(N'/N \cdot g(N))$ time. The output of each instance can be converted to a forest of rooted stars with the same connected components in $\mathcal{O}(N)$ time. We take the union of all these forests to obtain an auxiliary graph with at most $\lceil N'/N \rceil \cdot (N-1) = \mathcal{O}(N')$ edges, and find its connected components in $\mathcal{O}(N')$ time. Assuming that $f(N) \geq N$, this takes $\mathcal{O}(N'/N \cdot f(N))$ overall time and gives us the connected components of the whole input graph.

Now we consider the (N', N)-DC problem. We create $\lceil N'/N \rceil$ 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. This takes $\mathcal{O}(N'/N \cdot f(N))$ total time. A query is implemented exactly as above by querying all the instances and combining the results in $\mathcal{O}(N'/N \cdot f(N))$ time. □

3 (N, N)-DC in $\mathcal{O}(N^2 \log N)$ Total Update Time and $\mathcal{O}(N)$ Time 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. [7] (as mentioned in the introduction, appropriately tweaked for our use case). The decomposition is represented by a complete binary tree

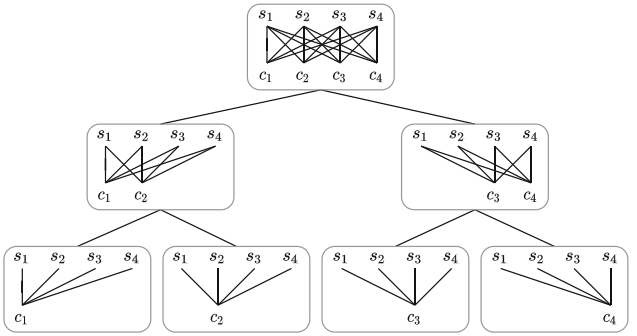

Fig. 1. The decomposition tree of $K_{4,4}$.

$\mathrm{DT}(G)$ of depth $\log N$. We identify the leaves of $\mathrm{DT}(G)$ 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 Fig. 1. Each node v maintains $\mathsf{cc}(v)$, the connected components of G_v represented as per Lemma 1. We stress that, while $\mathsf{cc}(v)$ is explicitly maintained, we do not explicitly store G_v at every node v. Given G, the preprocessing required to construct $\mathrm{DT}(G)$ together with $\mathsf{cc}(v)$ for every node v takes $\mathcal{O}(N^2)$ time by the following argument. First, we construct $\mathsf{cc}(G_c)$ for every leaf c. This can be done in $\mathcal{O}(N)$ time per leaf by simply iterating the neighbours of c in G. We then proceed bottom-up and compute $\mathsf{cc}(v)$ for every inner node v in $\mathcal{O}(N)$ time using the following lemma.

Lemma 5. *Let v be an inner node of $\mathrm{DT}(G)$, and v_ℓ, v_r be its children. Given $\mathsf{cc}(v_\ell)$ and $\mathsf{cc}(v_r)$ we can compute $\mathsf{cc}(v)$ in $\mathcal{O}(N)$ time.*

Proof. We construct the forests of rooted stars representing the connected components of $\mathsf{cc}(v_\ell)$ and $\mathsf{cc}(v_r)$ in $\mathcal{O}(N)$ time and take their union. Then we find the connected components of this union in $\mathcal{O}(N)$ time and save them as $\mathsf{cc}(v)$. □

We proceed to explain how to solve the (N, N)-DC problem in $\mathcal{O}(N \log N)$ time per update and $\mathcal{O}(N)$ time per query. The query simply returns $\mathsf{cc}(r)$, where r is the root of $\mathrm{DT}(G)$. The update is implemented as follows. Deactivating a character c possibly affects $\mathsf{cc}(v)$ for all ancestors v of leaf c. In particular, $\mathsf{cc}(c)$ becomes a collection of isolated vertices and can be recomputed in $\mathcal{O}(1 + |S|) = \mathcal{O}(N)$ time. We iterate over all ancestors v, starting from the parent of c. For each such v, let v_ℓ and v_r be its left and right child, respectively. We can assume that $\mathsf{cc}(v_\ell)$ and $\mathsf{cc}(v_r)$ have been already correctly updated. We compute $\mathsf{cc}(v)$ from $\mathsf{cc}(v_\ell)$ and $\mathsf{cc}(v_r)$ by applying Lemma 5 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 3 and 4, this implies that, given an incomplete matrix $\mathcal{A}_{n\times m}$, 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.

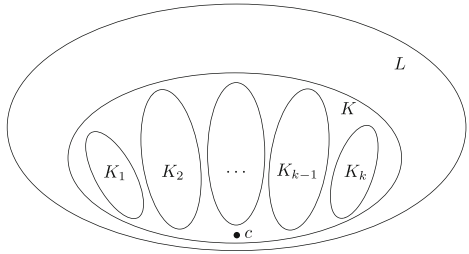

Fig. 2. After having removed c from K to obtain $K_1, K_2, \ldots, K_k$, we want to remove c from L.

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

We will need to compare the situation before and after an update, and so we introduce the following notation. A node v of $\mathrm{DT}(G)$ is responsible for the subgraph G_v before the update and for the subgraph G'_v after the update; $\mathsf{cc}(v)$ and $\mathsf{cc}'(v)$ denote the connected components of G_v and G'_v, respectively. The crucial observation is that $\mathsf{cc}'(v)$ is obtained from $\mathsf{cc}(v)$ 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 $\mathsf{cc}(c)$ in $\mathcal{O}(N)$ time. Then we iterate over the ancestors of c in $\mathrm{DT}(G)$. 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 $\mathsf{cc}'(v_i)$ and $\mathsf{cc}(u_i)$), and additionally to obtain a list of pointers to the components obtained by splitting L. See Fig. 2 for an illustration.

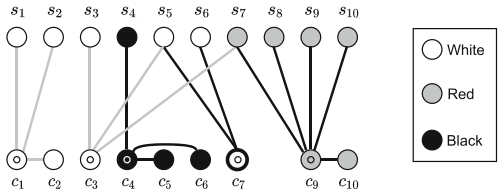

Fig. 3. The auxiliary graph implicitly constructed for a node v_{i+1} after deactivating c_8. Black edges are used for the star forest of v_i, grey edges for the star forest of u_i; an inner circle identifies the central vertices. K_1 is the component of c_9; c_7 is the next vertex to be considered in the visit, and it will eventually become red.

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 5, we will use 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 them, we simulate access to the adjacency lists of every vertex in both forests using $\mathsf{cc}'(v_i)$ and $\mathsf{cc}(u_i)$, as per Lemma 2. In turn, this allows us to simulate access to the adjacency list of every vertex in the auxiliary graph. See Fig. 3 for an example of the auxiliary graph.

By renaming the components we can assume that $|K_1| \geq |K_2|, |K_3|, \ldots, |K_k|$. In order to determine the new connected components after the removal of c, we will visit the vertices of L: 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; whenever retrieving the colour of a vertex 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 $\mathcal{O}(N)$ 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 $\mathcal{O}(|K'|)$ time. See Fig. 3 for an example.

Lemma 6. *The total time spent on all calls to the search procedure in the current iteration is* $\mathcal{O}(|L| - |K_1|)$.

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 $\mathcal{O}(k-1) = \mathcal{O}(|L| - |K_1|)$ overall. It remains to bound the number of all 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 $k - 1 = \mathcal{O}(|L| - |K_1|)$.

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, depending on whether they originate from $\mathsf{cc}'(v_i)$ or $\mathsf{cc}(u_i)$. 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 $\mathcal{O}(|L| - |K_1|)$. □

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 $\mathcal{O}(n)$. 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 vertex of $\bigcup_{j=2}^{k} K_j$ pays 1. Observe that the size of the connected component containing such a vertex 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 vertices of G_v due to deactivating the characters in the subtree of v (recall that in the end all such characters are deactivated).

Lemma 7. *The total cost paid by the vertices of G_v, over all 2^d deactivations affecting v, is $\mathcal{O}(N \cdot d)$.*

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:

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

□

There are $N/2^d$ nodes of $\mathrm{DT}(G)$ affected by 2^d deactivations, making the sum of the second parts:

$$\sum_{d=0}^{\log n} N/2^d \cdot n \cdot d < N^2 \sum_{d=0}^{\infty} d/2^d = \mathcal{O}(N^2).$$

Overall, the total update time is $\mathcal{O}(N^2)$, so by Lemmas 3 and 4 we arrive at the main result of this paper.

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

Acknowledgements. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 872539. GB was supported by the Netherlands Organisation for Scientific Research (NWO) under project OCENW.GROOT.2019.015 "Optimization for and with Machine Learning (OPTIMAL)".

References

1. Bashir, A., Ye, C., Price, A.L., Bafna, V.: Orthologous repeats and mammalian phylogenetic inference. Genome Res. **15**(7), 998–1006 (2005)
2. Bodlaender, H.L., Fellows, M.R., Hallett, M.T., Wareham, H.T., Warnow, T.J.: The hardness of perfect phylogeny, feasible register assignment and other problems on thin colored graphs. Theoret. Comput. Sci. **244**(1–2), 167–188 (2000)
3. Bonizzoni, P., Braghin, C., Dondi, R., Trucco, G.: The binary perfect phylogeny with persistent characters. Theoret. Comput. Sci. **454**, 51–63 (2012)
4. Bonizzoni, P., Ciccolella, S., Della Vedova, G., Soto, M.: Beyond perfect phylogeny: Multisample phylogeny reconstruction via ilp. In: 8th ACM-BCB, pp. 1–10 (2017)
5. Camin, J.H., Sokal, R.R.: A method for deducing branching sequences in phylogeny. Evolution, pp. 311–326 (1965)
6. El-Kebir, M.: Sphyr: tumor phylogeny estimation from single-cell sequencing data under loss and error. Bioinformatics **34**(17), i671–i679 (2018)
7. Eppstein, D., Galil, Z., Italiano, G.F., Nissenzweig, A.: Sparsification-a technique for speeding up dynamic graph algorithms. J. ACM **44**(5), 669–696 (1997)
8. Even, S., Shiloach, Y.: An on-line edge-deletion problem. J. ACM **28**(1), 1–4 (1981)
9. Fernández-Baca, D., Liu, L.: Tree compatibility, incomplete directed perfect phylogeny, and dynamic graph connectivity: An experimental study. Algorithms **12**(3), 53 (2019)
10. Gibb, D., Kapron, B., King, V., Thorn, N.: Dynamic graph connectivity with improved worst case update time and sublinear space. arXiv:1509.06464 (2015)
11. Gusfield, D.: Efficient algorithms for inferring evolutionary trees. Networks **21**(1), 19–28 (1991)
12. Gusfield, D.: Persistent phylogeny: a galled-tree and integer linear programming approach. In: 6th ACM-BCB, pp. 443–451 (2015)
13. Halperin, E., Karp, R.M.: Perfect phylogeny and haplotype assignment. In: Proceedings of the Eighth Annual International Conference on Resaerch in Computational Molecular Biology, pp. 10–19 (2004)

14. Henzinger, M.R., King, V., Warnow, T.: Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. Algorithmica **24**(1), 1–13 (1999)
15. Holm, J., De Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM **48**(4), 723–760 (2001)
16. Huang, S.E., Huang, D., Kopelowitz, T., Pettie, S.: Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In: 28th SODA, pp. 510–520. SIAM (2017)
17. Kimmel, G., Shamir, R.: The incomplete perfect phylogeny haplotype problem. J. Bioinform. Comput. Biol. **3**(02), 359–384 (2005)
18. Kirkpatrick, B., Stevens, K.: Perfect phylogeny problems with missing values. IEEE/ACM Trans. Comput. Biol. Bioinf. **11**(5), 928–941 (2014)
19. Nikaido, M., Rooney, A.P., Okada, N.: Phylogenetic relationships among cetartiodactyls based on insertions of short and long interpersed elements: hippopotamuses are the closest extant relatives of whales. Proc. Natl. Acad. Sci. **96**(18), 10261–10266 (1999)
20. Pe'er, I., Pupko, T., Shamir, R., Sharan, R.: Incomplete directed perfect phylogeny. SIAM J. Comput. **33**(3), 590–607 (2004)
21. Satas, G., Zaccaria, S., Mon, G., Raphael, B.J.: Scarlet: Single-cell tumor phylogeny inference with copy-number constrained mutation losses. Cell Syst. **10**(4), 323–332 (2020)
22. Satya, R.V., Mukherjee, A.: The undirected incomplete perfect phylogeny problem. IEEE/ACM Trans. Comput. Biol. Bioinf. **5**(4), 618–629 (2008)
23. Shiloach, Y., Vishkin, U.: An $o(\log n)$ parallel connectivity algorithm. J. Algorithms **3**(1), 57–67 (1982)
24. Stevens, K., Gusfield, D.: Reducing multi-state to binary perfect phylogeny with applications to missing, removable, inserted, and deleted data. In: Moulton, V., Singh, M. (eds.) Algorithms in Bioinformatics. WABI 2010. Lecture Notes in Computer Science, vol. 6293. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15294-8_23
25. Thorup, M.: Decremental dynamic connectivity. J. Algorithms **33**(2), 229–243 (1999)