An Optimal Algorithm for Sorting in Trees

Jishnu Roychoudhury
Princeton University, NJ, USA

Jatin Yadav
Indian Institute of Technology Delhi, India

Abstract

Sorting is a foundational problem in computer science that is typically employed on sequences or total orders. More recently, a more general form of sorting on partially ordered sets (or posets), where some pairs of elements are incomparable, has been studied. General poset sorting algorithms have a lower-bound query complexity of $\Omega(wn + n \log n)$, where w is the width of the poset.

We consider the problem of sorting in trees, a particular case of partial orders. This problem is equivalent to the problem of reconstructing a rooted directed tree from path queries. We parametrize the complexity with respect to d, the maximum degree of an element in the tree, as d is usually much smaller than w in trees. For example, in complete binary trees, $d = \Theta(1), w = \Theta(n)$. The previous known upper bounds are $O(dn \log^2 n)$ [19] and $O(d^2 n \log n)$ [1], and a recent paper proves a lower bound of $\Omega(dn \log_d n)$ [17] for any Las Vegas randomized algorithm. In this paper, we settle the complexity of the problem by presenting a randomized algorithm with worst-case expected $O(dn \log_d n)$ query and time complexity.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Sorting, Trees

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2023.7

Related Version *Previous Version*: https://arxiv.org/abs/2205.15912

1 Introduction

Sorting is a foundational problem in computer science that is typically employed on a set or multi-set of elements. Given an input set S of n elements, typical sorting problems determine the underlying total order or linear sequence of the elements. These algorithms assume that the elements are drawn from an ordered domain (such as the domain of integers). Most sorting algorithm use comparisons in which direct comparisons between pairs of elements allows the algorithms to acquire information about the total order [7]. Some recent works consider a restricted version of standard comparison sorting where only a subset of pairs are allowed to be compared [3, 4, 12, 15].

More recently, a more general form of sorting on partially ordered sets (or posets), where some pairs of elements are incomparable, has been studied [2, 5, 6, 8, 9]. Given a input set P of n elements, poset sorting algorithms determine the underlying partial order of the elements through *queries* to an oracle. The oracle's response to a query involving elements x and y is either the relation between x and y or a statement of their incomparability.

Sorting problems are inherently more challenging for partial orders compared to total orders. Sorting algorithms are generally evaluated by two complexity measures: the *query complexity*, the number of comparisons involved, and *total complexity*, the total number of computational operations involved. As each individual query can potentially be quite expensive (for example, requiring physical experiments), query complexity is equally important as total complexity. Faigle and Turán were the first to consider the problem of sorting partial orders [9], providing an algorithm with $O(wn \log n)$ query complexity, where n is the number of elements and w is the width of the poset. Daskalakis et al. [8] improved on this result to present an algorithm with optimal query complexity $O(wn + n\log n)$, and provided a separate algorithm with time complexity $O(w^2 n \log \frac{n}{w})$. A recent work in 2023 by Shaofeng H.-C. Jiang et al. [14] studies a generalized version of the poset sorting problem where a query graph G, defined on the same vertex set as the poset, is provided, and queries can only be made between pairs of elements which correspond to edges in the query graph.

Our work considers sorting tree posets, a particular case of partial orders. In a tree, the value of w is typically very high, leading to existing poset sorting algorithms performing poorly. Therefore, it is important to design efficient customized algorithms for sorting in tree posets with lower query and total complexity. We instead consider the parameter d, the maximum degree of a node in the natural arborescence representation of the tree poset. In contrast to width w, the degree d is typically much smaller; for example, $d = \Theta(1), w = \Theta(n)$ in complete binary trees. As with the poset sorting algorithms, our objective is to determine the underlying structure of the tree.

Apart from the theoretical significance of the problem, there are many practical applications: for example, constructing the evolutionary tree of coronavirus strains or general phylogenetic trees, or network mapping, among others. In addition, the final sorted tree can be used as input for tree searching algorithms, such as [10, 16]. This may be considered a tree analogue to the use of sorting to facilitate binary search in total orders. Onak and Parys [16] considered extending the concept of binary search to trees, whereas Heeringa, Iordan, and Lewis [10] considered searching in *dynamic* trees.

The problem we study is equivalent to the problem of reconstructing a rooted directed tree (arborescence) with **reachability** or **path queries**, as a comparison between two elements x and y of a tree poset can easily be simulated by asking two reachability queries (x, y) and (y, x) in the underlying rooted directed tree structure and vice-versa. This problem has received previous literature attention and there are currently two upper bounds with differing dependency on n and d. Wang and Honorio [19] provide a randomized algorithm with query complexity $O(dn\log^2 n)$, while Afshar et al. [1] provide a randomized algorithm with query complexity $O(d^2 n \log n)$.

Jagadish and Sen [13] considered the problem of reconstructing undirected trees using **separator queries**: given three vertices x, y, z, does y lie on the path between x and z? They give a deterministic algorithm with $O(dn^{1.5}\log n)$ query and time complexity. It should be noted that path queries are weaker than separator queries because there is a directed path from x to y in a rooted directed tree (or equivalently x is an **ancestor** of y) if and only if x lies on the path between the root and y in the underlying undirected tree structure. Nevertheless, their algorithm can be modified to work with path queries as well.

In a recent work, Bastide and Groenland [17] consider the problem of reconstructing connected undirected graphs with **distance queries**. Here, given two nodes (x, y), a query $\ell(x, y)$ returns the number of edges on the shortest path between them. They give a simple deterministic algorithm in the case of trees with a query complexity of $O(dn \log_d n)$. They also give a matching lower bound of $\Omega(dn \log_d n)$ queries for any randomized algorithm. This lower bound directly applies to our problem as well, as distance queries are stronger than comparision queries: for a rooted directed tree with root r, one can find if x is an ancestor of y by asking three distance queries $\ell(x, y)$, $\ell(r, x)$ and $\ell(r, y)$ in the undirected version of the tree. Thus, the fact that one needs $\Omega(dn \log_d n)$ comparision queries to sort a tree is stated as a corollary in their work (Corollary 4.10). Nevertheless, for the sake of simplicity and completeness, we borrow ideas from [17] to provide a shorter and more direct proof of the lower bound for **path queries**.

Many other previous works (e.g., [11, 18, 20, 21] have also considered the problem of reconstructing directed trees from distance queries. However, these works are limited by the specificity of distance as a query object. For instance, in evolutionary trees, it is far more natural to consider an ancestor relationship than a distance one. Our work provides an optimal algorithm for this most natural query object in trees.

2 Our contribution

We provide an optimal randomized algorithm for sorting tree posets using comparision queries, and equivalently reconstructing rooted directed trees using path queries:

▶ **Theorem 1.** *There exists a randomized algorithm which sorts a tree of n elements and maximum degree d in $O(dn\log_d n)$ expected worst-case query and time complexity.*

Table 1 The known bounds on the query complexity of randomized algorithms.

Work	Lower bound	Upper bound
[19]	$\Omega(n\log n)$	$O(dn\log^2 n)$
[1]	$\Omega(dn + n\log n)$	$O(d^2 n\log n)$
[17]	$\Omega(dn\log_d n)$	–
This work	–	$O(dn\log_d n)$

Both of the previous approaches ([1, 19]) for the upper bound rely on finding an **even-separator** edge, an edge such that the two subtrees remaining after removing the edge both have size at least $\frac{n}{d} - 1$, where n is the number of nodes in the original tree.

In contrast, our approach relies on finding a vertex such that the subtrees around the vertex each have size at most three-quarters that of the original tree. The actual details of finding such a vertex and then analysing the complexity are quite non-trivial. Another key difference is that in both [19] and [1], a bound on degree d is assumed to be known to the algorithm from the outset. In our problem formulation, the maximum degree d is not known and is instead discovered by the algorithm itself. This flexibility significantly improves the applicability of the solution. We only use d as a parameter in the complexity analysis.

3 Preliminaries

To precisely define the notions considered in this paper, we provide some formal definitions, many of which are set-theoretic analogues of graph-theoretic concepts. A partially ordered set, or poset, is a set of elements P together with a binary relation $\succ \subset P \times P$ which is irreflexive, transitive, and asymmetric. For a pair $(a, b) \in \succ$, we write $a \succ b$ and state that a dominates b, or b is dominated by a. If neither $a \succ b$ nor $b \succ a$, we say that a and b are *incomparable* and write $a||b$. A *maximal element*, or *root*, is an element x for which the set $\{y \in P : y \succ x\}$ is empty. Let the set of maximal elements of a poset $\mathcal{P}$ be $M(\mathcal{P})$. An *antichain* $A \subseteq P$ is a subset of mutually incomparable elements. The *width* $w(\mathcal{P})$ of poset $\mathcal{P}$ is the maximum cardinality of an antichain of the poset.

A *tree* is a poset $\tau = (T, \succ)$ where for all elements $t \in T$, the set of ancestors of t in the tree, $Anc_t = \{s \in T : s \succ t\}$, is well-ordered by the relation $\succ$. We will also sometimes refer to the elements of the tree as the *nodes* of the tree. A *child* of an element x is an element y such that $x \succ y$ and the set $\{z \in T : x \succ z \succ y\}$ is empty. For an element $t \in T$, we define ch_t to be the set of children of t. The *parent* of an element x, par_x, is an element y such that

$y \succ x$ and the set $\{z \in T : y \succ z \succ x\}$ is empty. Moreover, we define an *edge* as any pair (a, b) such that $b \in ch_a$, and the *degree* of an element t, deg_t, as the number of edges incident to t, i.e., $deg_t = |ch_t|$ for a maximal element, and $deg_t = |ch_t| + 1$ for all other elements. The *maximum degree* of the tree is $d(\tau) = \max(|M_\tau|, \max_{t \in T} deg_t)$. Note that $d(\tau)$ is dependent on the degree of elements in the tree and the number of maximal elements. The *subtree* of an element t is the set $\{t\} \cup \{s \in T : t \succ s\}$.

Trees in the set-theoretic sense may have multiple maximal elements (corresponding to graph-theoretic *forests*). However, we can easily modify any arbitrary tree to have only one maximal element. Suppose the elements of the tree are $\{t_1, t_2, \ldots, t_n\}$. We introduce a new element t_{n+1}, such that $t_{n+1} \succ t_i$ for all $i \in \{1, \ldots, n\}$, and sort the modified tree. In addition, we never need to make a query involving t_{n+1}, as it is known that t_{n+1} dominates all other elements. Thus, we will only consider trees with one maximal element for the rest of this paper.

Our algorithm finds the *edges* of the tree $\tau = (T, \succ)$, which is sufficient to sort the tree. Note that, after finding the edges, a standard $O(n)$ depth-first search can be run from the root, producing a data structure that stores the DFS pre-visit and post-visit times for all the vertices. This data structure that can respond to comparison queries in $O(1)$, as a vertex x is an ancestor of vertex y (or equivalently $x \succ y$) if and only if $\text{pre}[y] \in (\text{pre}[x], \text{post}[x]]$, thus fully mimicking the oracle.

4 Algorithm Outline

Our main algorithm, `GET-EDGES` (detailed pseudocode can be found in Appendix B), relies on finding a *good* separating element in the tree and then dividing the tree into disjoint sub-problems around this element, such that no sub-problem has a large size. An obvious first choice for such a node is the centroid of the tree:

▶ **Definition 2** (Centroid). *The centroid of a tree T with n nodes is a node c, such that the size of the subtree of c is $\geq \frac{n}{2}$, and for any of its children, the subtree size is $< \frac{n}{2}$.*

It is well known and easy to see that there exists a unique centroid for any tree. Nevertheless, we include the proof in Appendix A (Lemma 8) for the sake of completeness.

To find the centroid c with probability at least $\frac{1}{2}$, we can select a random node x, and find its set of ancestors. This set forms a total order, and can be sorted using a standard comparision-based sorting algorithm. Now, we binary search on this list and return the lowest ancestor with subtree size $\geq \frac{n}{2}$. If x lies in the subtree of c (which occurs with probability at least $\frac{1}{2}$, as the subtree size of c is $\geq \frac{n}{2}$), this process will return c. Since sorting takes $O(n \log n)$ time and $O(\log n)$ subtree size computations during the binary search can be done in $O(n)$ each by iterating over all the nodes, this takes $O(n \log n)$ time.

To save a factor of $\log n$, we will instead use a *pseudo-centroid* as the separating element:

▶ **Definition 3** (Pseudo-Centroid). *A pseudo-centroid of a tree T with n nodes is a node c, such that the size of the subtree of c is $\geq \frac{n}{4}$, and for any of its children, the subtree size is $< \frac{n}{2}$. In particular, note that the centroid is also a pseudo-centroid.*

With this definition in mind, `GET-EDGES` now proceeds in three steps:

1. Find a pseudo-centroid. To accomplish this, we will later describe a method (similar to the above described method for finding a probable centroid) to find a probable (with probability $\geq \frac{1}{8}$) pseudo-centroid c in $O(n)$ expected time. Then, we test if c is a pseudo-centroid, and if it is not, we repeat the whole process.

2. Divide the elements of the tree (except c) into sub-problems around the element c. The subtree of each child of c forms a subproblem and all the elements of the tree except the subtree of c form an additional subproblem. We refer to the former as *downward subproblems* and the latter as the *upward subproblem* for brevity.
3. Solve each subproblem recursively. Each subproblem is a tree which constitutes a (weakly) connected subgraph of the original graph. By the definition of pseudo-centroids, each sub-problem will have $\leq \frac{3n}{4}$ nodes.

We will design the algorithm to test whether a node c is a pseudo-centroid in such a way that:

1. If c is not a pseudo-centroid, the test returns a negative result in $O(n)$ expected time.
2. If c is a pseudo-centroid and divides the tree into subproblems $x_1, x_2, \ldots, x_k$ with $|x_1| \geq \cdots \geq |x_k|$, then the algorithm results a positive result, along with all the k subproblems, in $O(n + \sum_{i=1}^{k} i|x_i|)$ expected time.

Then, the recursion for the expected time complexity becomes

$$T_d(n) = O(n) + \sum_{i=1}^{k}(O(i|x_i|) + T_d(|x_i|))$$

where $k \leq d$, $\sum_{i=1}^{k} |x_i| = n - 1$ and $1 \leq |x_i| \leq \frac{3n}{4}\ \forall\, 1 \leq i \leq k$. We will prove via a charging argument on the *recursion tree* that $T_d(n) = O(dn \log_d n)$.

5 Algorithm Details and Complexity Analysis

We first describe our algorithm `GET-PROBABLE-PSEUDO-CENTROID` for finding a probable (with probability $\geq \frac{1}{8}$) pseudo-centroid in $O(n)$ expected time. We assume sufficiently large n (greater than some constant) in the discussion to follow to avoid minor details such as the possibilities of $\log n = 0$ or $\frac{1}{\log n} > 1$, and for some inequalities to be true.

Define the subtree size of a node i as S_i. Also, let $p = \frac{1}{\log n}$. Now,

1. Randomly sample a subset V of nodes by adding each node to V with an independent probability of p. Thus, $\mathbb{E}(|V|) = \frac{n}{\log n}$. We define the sampled subtree size, F'_i to be the number of elements in V that lie inside the subtree of i. Clearly, $\mathbb{E}(F'_i) = S_i p$. Also, we define the estimated subtree size of i to be $S'_i = \frac{F'_i}{p}$, so that $\mathbb{E}(S'_i) = S_i$.
2. Select a random node x. We will output the lowest ancestor y of x with $S'_y \geq \frac{n}{2} - \frac{1}{p}$ as our (probable) pseudo-centroid. Let Y be the set of ancestors of x, along with x itself. Note that, for any ancestor u of v, $S'_u \geq S'_v$. One way to find this lowest ancestor y is to first sort Y and then binary search on Y. But sorting Y can take $n \log n$ queries. To fix this, we sample a subset Z of Y by adding each node in Y to Z with an independent probability of p. Thus, $\mathbb{E}(|Z|) = \frac{\mathbb{E}(|Y|)}{\log n}$.
3. Sort Z according to the order relation $\succ$. Then, use a modified binary search on Y to find the pseudo-centroid. This modified binary search runs in two phases:
   a. Binary search on Z to find an *estimated* pseudo-centroid within Z.
   b. Find the position of the actual pseudo-centroid from the expected $O(\log n)$ elements of Y that lie between the estimated pseudo-centroid and the immediate next element in Z.

Note that binary searching is valid because for any ancestor u of v, $S'_u \geq S'_v$. We now prove bounds on the complexity of this algorithm and its probability of success.

Algorithm 1 GET-PROBABLE-PSEUDO-CENTROID.

```
Input : A set $T$ of nodes that form a tree $\tau = (T, \succ)$ with $\succ$ unknown, and an
        oracle that answers for any pair of nodes $a \in T, b \in T$ whether $a \succ b$.
Output: A probable pseudo centroid of $T$
$n \leftarrow |T|$
$p \leftarrow \frac{1}{\log n}$
$V \leftarrow \{\}$
for $i \in T$ do
    Add $i$ to $V$ with probability $p$
end
Choose a random element $x$ from $T$.
$Y \leftarrow \{x\} \cup \{y \in T \mid y \succ x\}$
$Z \leftarrow \{$GET-ROOT$(T)\}$
for $i \in Y$ do
    Add $i$ to $Z$ with probability $p$
end
Sort $Z$ according to the order relation $\succ$
Let $Z = \{z_1, z_2, \ldots z_k\}$ where $z_1 \succ z_2 \succ \ldots z_{k-1} \succ z_k$
Define $S'_j$ to be the estimated size of subtree of $j$, $S'_j = \frac{|(\{j\} \cup \{i \in T | j \succ i\}) \cap V|}{p}$
$t \leftarrow z_1$
if $S'_{z_1} \geq \frac{n}{2} - \frac{1}{p}$ then
    Binary search to find the largest $i$, such that $S'_{z_i} \geq \frac{n}{2} - \frac{1}{p}$
    $L \leftarrow \{z_i\}$
    for $u \in Y$ do
        if $(i \neq k$ and $z_i \succ u$ and $u \succ z_{i+1})$ or $(i = k$ and $z_i \succ u)$ then
            $L \leftarrow L \cup \{u\}$
        end
    end
    Sort $L$ according to the order relation $\succ$
    Let $L = \{l_1, l_2, \ldots, l_r\}$, where $l_1 \succ l_2 \ldots l_{r-1} \succ l_r$
    Binary search to find the largest $i$, such that $S'_{l_i} \geq \frac{n}{2} - \frac{1}{p}$
    $t \leftarrow l_i$
end
return $t$
```

▶ **Lemma 4.** *The algorithm GET-PROBABLE-PSEUDO-CENTROID finds a pseudo-centroid with probability $\geq \frac{1}{8}$ in $O(n)$ expected time.*

Proof. It is simple to verify that the algorithm takes $O(n)$ expected time by considering each individual step. Specifically:

1. Sampling V takes $O(n)$ time.
2. Selecting x, computing Y, and sampling Z takes $O(n)$ time.
3. Sorting Z takes $O(|Z| \log |Z|) = O(|Z| \log n)$ time. Since $\mathbb{E}(|Z|) = \frac{|Y|}{\log n} \leq \frac{n}{\log n}$, this takes $O(n)$ expected time. The modified binary search also takes $O(n)$ time, as shown below:
   a. Binary searching Z takes $O(\log n)$ rounds. In each round, we compute the estimated subtree size for some element i, S'_i, which takes $O(\frac{n}{\log n})$ time since $\mathbb{E}(|V|) = \frac{n}{\log n}$. Thus, this step takes $O(n)$ expected time total.

b. Finding the list L of elements that lie in-between takes $O(n)$ time, as it requires only two comparison queries per element in Y. Moreover, since $\mathbb{E}(L) = O(\log n)$, sorting L takes expected $\mathbb{E}(|L|\log|L|) = O(\mathbb{E}(|L|\log n)) = O(\log^2 n) = O(n)$ time, and binary searching L takes $O(\log n)$ rounds. In each round, we compute an estimated subtree size in expected $O(\frac{n}{\log n})$, and thus the total expected time complexity is $O(n)$.

We thus concentrate on the probability of success. Let the centroid be c (this is the unique centroid, not just a pseudo-centroid). We first prove that, for sufficiently large n, if all of the following three events occur, we get a pseudo-centroid.

1. $x = c$ or $c \succ x$
2. $S'_c \geq \frac{n}{2} - \frac{1}{p}$
3. For all $i \in T$, $S'_i \leq S_i + \frac{n}{10}$

The algorithm returns the minimal ancestor y of x with $S'_y \geq \frac{n}{2} - \frac{1}{p}$ (let it be c'). If $c' = c$, we are done. Else, events 1 and 2 imply $c \succ c'$. Now, $S'_{c'} \geq \frac{n}{2} - \frac{1}{p}$ and $S_{c'} \geq S'_{c'} - \frac{n}{10}$ (event 3). Adding these two gives $S_{c'} \geq \frac{n}{2} - \frac{1}{p} - \frac{n}{10} = \frac{2n}{5} - \log n \geq \frac{n}{4}$ for sufficiently large n. Also, $S_{c'} < \frac{n}{2}$ as c' lies strictly inside the subtree of c, and all children of c have sizes $< \frac{n}{2}$. So, for every child r of c', $S_r < S_{c'} < \frac{n}{2}$. Thus, c' is a pseudo-centroid.

We now proceed to proving that the probability of these 3 events all occurring simultaneously is $\geq \frac{1}{8}$ for sufficiently large n. Let us try to get an upper bound on at least one of these events not occurring. Let p_1, p_2, p_3 be the probabilities of events $1, 2, 3$ **not** occurring respectively. We will establish bounds on all three of them. Firstly, $\mathbf{p_1} \leq \mathbf{\frac{1}{2}}$ as at least half of the nodes lie inside the subtree of c.

To get an upper bound on p_2, note that F'_c follows a binomial distribution with parameters $S_c \geq \frac{n}{2}$ and $p = \frac{1}{\log n}$. As any median of a binomial distribution with parameters n and p is $\geq \lfloor np \rfloor$, we have $S'_c = F'_c/p \geq \lfloor \frac{pn}{2} \rfloor / p \geq \frac{n}{2} - \frac{1}{p}$ with probability at least $\frac{1}{2}$. Therefore, $\mathbf{p_2} \leq \mathbf{\frac{1}{2}}$.

Finally, we will prove that $\mathbf{p_3} \leq \mathbf{\frac{1}{4}}$ for sufficiently large n. Indeed, consider an element i, with k elements in its subtree, $u_1, u_2, \ldots u_k$, where $k = S_i$. Let X_j be an indicator variable for whether u_j was chosen in V. All X_j's are independent and identically distributed with $\mathbb{E}(X_j) = p$. Also, $F'_i = X_1 + X_2 + \ldots X_k$, and $\mathbb{E}(F'_i) = kp$. Since we want a total error of $\leq \frac{np}{10}$ in F'_i, we use the Chernoff bound with $\mu = kp, \delta = \frac{n}{10k} \geq \frac{1}{10}$ to get:

$$\begin{aligned}
\mathbb{P}\left(S'_i > S_i + \frac{n}{10}\right) &= \mathbb{P}\left(F'_i > kp + \frac{np}{10}\right) \\
&= \mathbb{P}\left(F'_i > \mu(1+\delta)\right) \\
&\leq \exp\left(-\frac{\mu}{3}\delta \min(1,\delta)\right) \\
&\leq \exp\left(-\frac{kp}{3}\frac{n}{10k}\frac{1}{10}\right) \\
&= \exp\left(\frac{-np}{300}\right) \qquad (1)
\end{aligned}$$

Using the union bound, $p_3 \leq n \cdot \exp\left(\frac{-np}{300}\right) = n\exp\left(\frac{-n}{300\log n}\right) \leq \frac{1}{4}$ for sufficiently large n.

Now, let us suppose we fail if at least one of the events $1, 2, 3$ does not occur. The probability of failure is the sum of the probability of event 1 not occurring and the probability of event 1 occurring but at least one of the events 2 and 3 not occurring. Noticing that event 1 is independent of both events 2 and 3, and applying the union bound, we get that the probability of failure is at most:

$$p_1 + (1-p_1)(p_2+p_3) \leq p_1 + \frac{3}{4}(1-p_1) = \frac{3}{4} + \frac{p_1}{4} \leq \frac{7}{8}$$

Therefore, we find a pseudo-centroid with probability at least $\frac{1}{8}$. ◀

Algorithm 2 TEST-PSEUDO-CENTROID.

Input : A set T of nodes that form a tree $\tau = (T, \succ)$ with $\succ$ unknown, and an oracle that answers for any pair of nodes $a \in T, b \in T$ whether $a \succ b$. Also a candidate pseudo-centroid c

Output : A pair (b, S), where b is a boolean denoting if c is a pseudo-centroid. If b is `true`, S is the set of subproblems when T is divided into around c.

```
$n \leftarrow |T|$
$H \leftarrow T \setminus c$
$S \leftarrow []$
$R \leftarrow \{x \in H : x \succ c\}$
if $|R| > \frac{3n}{4}$ then
    return (false,[])
end
Append $R$ to $S$
$H \leftarrow H \setminus R$
while $H \neq \emptyset$ do
    Let $H = h_1, h_2, \ldots, h_m$
    Choose a random element $x$ from $H$
    for $i = 1, \ldots, m$ do
        if $h_i \succ x$ then
            $x \leftarrow h_i$
        end
    end
    $X \leftarrow \{x\} \cup \{y \in H : x \succ y\}$
    if $|X| \geq \frac{n}{2}$ then
        return (false,[])
    end
    Append $X$ to $S$
end
return (true,$S$)
```

We now describe our algorithm TEST-PSEUDO-CENTROID, which simultaneously tests if an element c is a pseudo-centroid and divides the tree into subproblems around c if it is. The algorithm first finds the upward subproblem in $O(n)$, removes the elements from the list of unaccounted elements, and tests if the subtree of c has size at least $\frac{n}{4}$, returning `false` if it does not. After this, we repeatedly find one of the downward subproblems, check if the subproblem has size less than $\frac{n}{2}$ (returning `false` if it does not), and remove the subproblem elements from the list of unaccounted elements until there are no unaccounted elements remaining, at which point we return the list of subproblems. We can find a downward subproblem in $O(|H|)$, where H is the set of unaccounted elements, by doing the following. First, set x to be a random element from H. Then, iterate through all elements in H. If h_i is the current element, set $x \leftarrow h_i$ if $h_i \succ x$. Thus, the final value of x will be that of one of the children of c. Then, we find all elements in the subtree of x by comparing each element in H against x, and thus find a subproblem in $O(|H|)$ time.

The algorithm correctness is obvious and as such we prove its complexity.

▶ **Lemma 5.** *Suppose c divides the tree into k subproblems $x_1, \ldots, x_k$, such that $|x_1| \geq \cdots \geq |x_k|$. Then:*

- *If c is not a pseudo-centroid, `TEST-PSEUDO-CENTROID` runs in $O(n)$ expected time.*
- *If c is a pseudo-centroid, `TEST-PSEUDO-CENTROID` runs in $O(n + \sum_{i=1}^{k} i|x_i|)$ expected time.*

Proof. We first prove that `TEST-PSEUDO-CENTROID` runs in $O(n)$ expected time when c is not a pseudo-centroid. There are two cases. The first is when the size of the subtree of c is less than $\frac{n}{4}$. This is detected immediately after finding the upward subproblem, which takes only $O(n)$ time. The second case is that one of the downward subproblems x_l has size at least $\frac{n}{2}$. In this case, note that we have a probability of at least $\frac{1}{2}$ of finding x_l in each iteration, as the subproblem found contains the random element chosen at the beginning of the iteration. Thus, we are expected to detect failure in a constant number of iterations. Since each iteration runs in $O(n)$, the expected complexity is $O(n)$.

We now prove that `TEST-PSEUDO-CENTROID` runs in $O(n + \sum_{i=1}^{k} i|x_i|)$ expected time when c is a pseudo-centroid. First, $O(n)$ time is used to find the upward subproblem.

Let us analyse the expected number of comparisions between two downward subproblems x_i and x_j with $i \neq j$. The probability that x_i is found before x_j is $\frac{|x_i|}{|x_i|+|x_j|}$. When this happens, each element of subproblem x_j is compared against twice, once when finding the maximal element of x_i and once when finding the list of elements of x_i. Similarly, we can account for the expected comparisions when x_j is found before x_i, and the total expected number of comparisions between x_i and x_j is therefore $2|x_j|\frac{|x_i|}{|x_i|+|x_j|} + 2|x_i|\frac{|x_j|}{|x_i|+|x_j|} = \frac{4|x_i||x_j|}{|x_i|+|x_j|}$

The cost of finding the upward subproblem x_p has already been accounted for. Regardless, we also charge a cost of $4\frac{|x_p||x_j|}{|x_p|+|x_j|}$ for the number of comparisons between x_p and x_j for simplicity, which is acceptable since we are proving an upper bound. In addition, there are some comparisons made between two elements of the same subproblem x_i. However, there are only $O(|x_i|)$ such comparisons, and thus the total over all subproblems is $O(n)$.

We observe that for any $1 \leq j < i \leq k$, $\frac{|x_i||x_j|}{|x_i|+|x_j|} \leq \min(|x_i|, |x_j|) = |x_i|$. The total number of comparisions is therefore bounded by:

$$O(n) + \sum_{1 \leq j < i \leq k} O(|x_i|) = O(n) + \sum_{i=1}^{k} O(i|x_i|)$$

It is easy to see that in each iteration both the runtime and the number of comparisions are $\Theta(|H|)$, so the runtime also follows the same bound of $O(n) + \sum O(i|x_i|)$. ◀

We finally prove the main result of the paper by showing that `GET-EDGES` runs in expected $O(dn \log_d n)$ time.

▶ **Theorem 6.** *`GET-EDGES` sorts a tree of n elements and maximum degree d in $O(dn \log_d n)$ expected worst-case query and time complexity.*

Proof. Due to Lemmas 4 and 5, the recurrence relation for the time complexity of `GET-EDGES` is

$$T_d(n) = O(n) + \sum_{i=1}^{k} (O(i|x_i|) + T_d(|x_i|))$$

where $x_1, \ldots, x_k$ are the subproblems divided around the pseudo-centroid, and $|x_1| \geq \cdots \geq |x_k|$.

We prove the complexity of this recurrence relation by considering the *recursion tree* of the algorithm. The root of this recursion tree is the entire tree, and for any node, its children are the subproblems divided around the pseudo-centroid. Note that this recursion tree has exactly n nodes.

We immediately note that the sum of all $O(n)$ terms is $O(n \log n)$, as the recursion tree has $O(\log n)$ levels (since child subproblems have size at most $\frac{3}{4}$ that of a parent subproblem), and the sum of subproblem sizes at each level is at most n.

We now consider the sum of all $\sum_{i=1}^{k} O(i|x_i|)$ terms. For each node in the recursion tree, we rank its children $x_1, \dots, x_k$ in non-increasing order of subproblem size, such that $|x_1| \geq \cdots \geq |x_k|$. Then, the cost of a child node x_i is $O(i|x_i|)$. Note that since this subproblem has size $|x_i|$, its subtree in the recursion tree also has $|x_i|$ nodes, including itself. We can now distribute the cost of $O(i|x_i|)$ by charging a cost of i to each node in the subtree.

Let v_z be the total cost charged to a node z. We now prove $v_z = O(d \log_d n)$ for all z. Consider the path from node z to the root of the recursion tree. For each node b in the path with parent a, if b had charged a cost of i to z, then at least i children of a have size at least $|s_b|$, where $|s_b|$ is the subtree size of b. However, we note then that the subtree size of a is at least $i|s_b|$. Moreover, the root has subtree size n.

Specifically, let a be an ancestor of node z, and let b be the unique node which is a child of a and an ancestor of z (or z itself). If we let $f_{z,a}$ be the rank of b within the children of a (according to the ranking scheme defined above), then we have:

$$v_z = \sum_{a \in A_z} f_{z,a}$$

where A_z is the set of ancestors of node z in the recursion tree. We also know that $\prod_{a \in Anc_z} f_{z,a} \leq n$, since if there are $f_{z,a}$ nodes with subtree size at least $|s_b|$, the subtree size of a is at least $f_{z,a}|s_b|$. In addition, we know that $f_{z,a} \leq d$ for all (z,a) (as each node in the recursion tree has at most d children), and that z has at most $O(\log n)$ ancestors. We use this information to show in Lemma 9 in the Appendix that $v_z = O(d \log_d n)$.

Since the sum of all $O(n)$ terms is $O(n \log n)$, and the sum of all $\sum_{i=1}^{k} O(i|x_i|)$ terms is $O(dn \log_d n)$, the total complexity is $O(n \log n + dn \log_d n) = O(dn \log_d n)$, as required. ◀

Note that the analysis of our algorithm is tight for complete $d-$ary trees. In the next section, we prove a matching lower bound considering the case of *almost* complete $d-$ary trees.

6 Lower bound

A lower bound of $\Omega(dn \log_d n)$ **distance queries** to reconstruct a tree of maximum degree d is proved in [17], and as a corollary this bound is valid for our case (**path queries**) as well, as each path query can be answered using 3 distance queries. Here, borrowing ideas from [17], we give a simpler and more direct proof of this lower bound in the case of path queries.

Assume $d \geq 3$. Consider a tree with $k+1$ levels, where for each $i = 0, 1, \dots k$, the i^{th} level is defined as the set of nodes at a distance i from the root. All nodes on the level k are leaves. Each node at a level $0 \leq i < k-1$ has exactly d children, and each node on the level $k-1$ has exactly 1 child. In other words, the first k levels form a complete $d-$ary tree, where for each $i = 0, 1, \dots k-1$, the i^{th} level contains d^i nodes, and there are d^{k-1} nodes on level k. The number of nodes in the tree is therefore $n = d^0 + d^1 + \dots d^{k-1} + d^{k-1}$. Suppose the value of d and the whole subgraph consisting of the first k levels $0, 1, \dots k-1$ are provided for free by the oracle, without any queries. Let the set of the nodes at level $k-1$ be X and the set of the nodes at level k be Y. Let $m = |X| = |Y| = d^{k-1}$ be the number of nodes on the last level. Note that $m \geq \frac{n}{3}$. Let's call such a tree a *balanced tree* and let $\mathcal{B}$ be the set of all balanced trees. Note that the number of balanced trees is $B = |\mathcal{B}| = m!$, as we need to choose a distinct parent in X for each node in Y.

We want to apply Yao's principle to prove a lower bound on the number of queries required to reconstruct a *balanced tree.* Ideally, we would want to be able to consider each node in Y independently and prove a lower bound on the number of queries required to find its parent. But the fact that each node in X has exactly 1 child means that the parents of the nodes in Y have some correlation. So, let's first relax this condition, even though it would mean that the maximum degree can now exceed d. Let's define the set of *relaxed trees*, $\mathcal{T}$ to be the set of all the trees where the first k levels are as provided, and the parent of any node in Y can be any node in X. Note that $|\mathcal{T}| = m^m$, as for each node in Y, there are m choices for its parent.

First, we prove that if the input tree was chosen at random from $\mathcal{T}$, any deterministic algorithm must make at least $\Omega(mdk)$ queries with a very high probability. Clearly, any query that can give some useful information must be of this form: given a node y on the last level and a node x from the first k levels, is x an ancestor of y?

▶ **Lemma 7.** *Assume $d \geq 3$ and $k \geq 10$. Any deterministic algorithm that reconstructs a tree chosen at random from $\mathcal{T}$ uses at least $\frac{mdk}{8}$ queries with a* **NO** *answer, with probability $\geq 1 - e^{-m}$*

Proof. Choosing the tree at random from $\mathcal{T}$ is equivalent to choosing the parent of each of the m nodes in Y, independently and uniformly at random from X. Suppose we want to find the parent (say p) of some node $y \in Y$.

Suppose the algorithm knows a node z at some level $0 \leq i \leq k-1$, such that, given the set of queries asked in the past, p can be any of the nodes in the subtree of z at the level $k-1$, with equal probability. Initially, $i = 0$. When i equals $k-1$, we are done. Given z, let q_1 be the first vertex queried by the algorithm, and for each $u > 1$, define q_u to be the next queried vertex if the answer to q_{u-1} was **NO** (meaning that q_{u-1} is not an ancestor of y). Suppose that, out of the children $z_1, z_2, \ldots, z_d$ of z, z_r was the child which is an ancestor of y. The variable r is distributed uniformly among $[d]$. Let q_j be the first node in the sequence $q_1, q_2, \ldots$ that lies in the subtree of z_r (note that such a node must exist). Clearly, all of $q_1, q_2, \ldots q_{j-1}$ must be queried by the algorithm, all resulting in an answer of **NO**. When the algorithm queries q_j, instead of a **YES/NO** answer, the oracle replies that y lies in the subtree of z_r, and that the current query (q_j) doesn't count towards the total number of queries. This step only gives the algorithm extra information without any charge on the number of queries, and the algorithm can choose to ignore this information and query q_j again (we give a formal proof justifying this step in the Appendix, in Lemma 10). For each $t \in [d]$, let f_t be the smallest index h, such that q_h lies in the subtree of z_t. The number (say $Q_{y,i}$) of $t \in [d]$ with $f_t < f_r$ is a random integer in $\{0, 1, \ldots d-1\}$. Clearly at least $Q_{y,i}$ queries must be made before the algorithm knows the child z_r whose subtree contains y. The total number of queries asked therefore is at least a sum of $m \cdot (k-1)$ independent random variables each in the range $[0, d-1]$ with mean $\frac{d-1}{2}$. Applying Hoeffding's inequality yields that the probability that the number of queries asked is less than $m(d-1)(k-1)/4$ is at most $\exp\left(\dfrac{-2(m(k-1)(d-1)/4)^2}{m(k-1)(d-1)^2}\right) = e^{-(m(k-1))/8}$, and using $k \geq 10, d \geq 3$ yields the required result. ◀

This means that, given $d \geq 3, k \geq 10$, for any deterministic algorithm, there are at most $L = e^{-m}|\mathcal{T}| = e^{-m}m^m$ trees for which it takes less than $\frac{mdk}{8}$ queries. Also, the number of balanced trees is $B = m!$ Therefore, $L \leq \frac{B}{2}$ using $n! \geq \frac{n^n}{e^{n-1}}$ for all $n \geq 1$. We claim that, any deterministic algorithm which is able to correctly reconstruct all balanced trees must ask at least $\frac{mdk}{8}$ queries with a **NO** answer, for at least half of the balanced trees. Suppose this were not the case for a deterministic algorithm D. Consider an algorithm D' which is able to correctly reconstruct all trees in $\mathcal{T}$. D' first runs the algorithm D. If the original tree was balanced, the algorithm must already have the tree, and can verify it by querying for all the edges (note that if the tree was balanced, all these queries result in a **YES** answer). If the verification fails, D' just asks all the $\binom{n}{2}$ queries to reconstruct the tree from scratch. So, for at least $B/2 + 1 > L$ trees, D' makes $\leq \frac{mdk}{8}$ queries with a **NO** answer, a contradiction to Lemma 7.

It then follows from Yao's principle that the worst-case query complexity of any randomized algorithm must be at least $\frac{mdk}{16} \geq \frac{ndk}{48} = \Omega(nd\log_d(n))$. We note that this result holds even when $k < 10$ as there is a pre-existing bound of $\Omega(nd)$ which is only a constant off $\Omega(nd\log_d(n))$, since $n = O(d^{10})$ for $k < 10$.

7 Concluding Remarks

In this paper, we have considered the problem of sorting in trees, a particular case of partial orders. and presented an **optimal** randomized algorithm for sorting a tree poset in worst-case expected $O(dn\log_d n)$ query and time complexity, where d is the maximum degree of the tree. An interesting direction for future work is resolving the **deterministic** query complexity of tree sorting. Currently, to the best of our knowledge, the best known upper bound is $O(dn^{1.5}\log n)$ [13], and the best known lower bound is $\Omega(nd\log_d n)$.

References

1 Ramtin Afshar, Michael T. Goodrich, Pedro Matias, and Martha C. Osegueda. Reconstructing Biological and Digital Phylogenetic Trees in Parallel. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.3.

2 Ramtin Afshar, Michael T Goodrich, Pedro Matias, and Martha C Osegueda. Parallel network mapping algorithms. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 410–413, 2021.

3 Noga Alon, Manuel Blum, Amos Fiat, Sampath Kannan, Moni Naor, and Rafail Ostrovsky. Matching nuts and bolts. In *SODA*, pages 690–696. Citeseer, 1994.

4 Indranil Banerjee and Dana Richards. Sorting under forbidden comparisons. In *15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016.

5 Arindam Biswas, Varunkumar Jayapaul, and Venkatesh Raman. Improved bounds for poset sorting in the forbidden-comparison regime. In *Conference on Algorithms and Discrete Applied Mathematics*, pages 50–59. Springer, 2017.

6 Jean Cardinal and Samuel Fiorini. On generalized comparison-based sorting problems. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 164–175. Springer, 2013.

7 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

8 C Daskalakis, RM Karp, E Mossel, SJ Riesenfeld, and E Verbin. Sorting and selection in posets. *SIAM Journal on Computing*, 40(3), 2011.

9 U Faigle and Gy Turán. Sorting and recognition problems for ordered sets. *SIAM Journal on Computing*, 17(1), 1988.

10 Brent Heeringa, Marius Cătălin Iordan, and Louis Theran. Searching in dynamic tree-like partial orders. In *12th international conference on Algorithms and data structures (WADS)*, 2011.

11 Jotun J Hein. An optimal algorithm to reconstruct trees from additive distance data. *Bulletin of mathematical biology*, 51(5):597–603, 1989.

12 Zhiyi Huang, Sampath Kannan, and Sanjeev Khanna. Algorithms for the generalized sorting problem. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 738–747. IEEE, 2011.

13 M. Jagadish and Anindya Sen. Learning a bounded-degree tree using separator queries. In Sanjay Jain, Rémi Munos, Frank Stephan, and Thomas Zeugmann, editors, *Algorithmic Learning Theory - 24th International Conference, ALT 2013, Singapore, October 6-9, 2013. Proceedings*, volume 8139 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2013. `doi:10.1007/978-3-642-40935-6_14`.

14 Shaofeng H.-C. Jiang, Wenqian Wang, Yubo Zhang, and Yuhao Zhang. Algorithms for the generalized poset sorting. *arXiv preprint*, 2023. `arXiv:2304.01623`.

15 William Kuszmaul and Shyam Narayanan. Stochastic and worst-case generalized sorting revisited. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1056–1067. IEEE, 2022.

16 K. Onak and P. Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2006.

17 Carla Groenland Paul Bastide. Optimal distance query reconstruction for graphs without long induced cycles. *arXiv preprint*, 2023. `arXiv:2306.05979`.

18 Lev Reyzin and Nikhil Srivastava. On the longest path algorithm for reconstructing trees from distance matrices. *Information processing letters*, 101(3):98–100, 2007.

19 Zhaosen Wang and Jean Honorio. Reconstructing a bounded-degree directed tree using path queries. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 506–513. IEEE, 2019.

20 Michael S Waterman, Temple F Smith, Mona Singh, and William A Beyer. Additive evolutionary trees. *Journal of theoretical Biology*, 64(2):199–213, 1977.

21 Li Zhang, Valerie King, and Yunhong Zhout. On the complexity of distance-based evolutionary tree reconstruction. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, page 444. SIAM, 2003.

A Omitted Proofs

▶ **Lemma 8.** *For any tree, the centroid exists and is unique.*

Proof. Consider the following algorithm. We start at the root, and keep recursing to the child with subtree size $\geq \frac{n}{2}$ until there is no such child. This maintains the invariant that any centroid must be inside the subtree of the current element. This process must terminate as the number of elements is finite, and it can only terminate on a centroid (let it be c). Any other centroid must lie in the subtree of c, but all children of c have subtree sizes less than $\frac{n}{2}$, so c must be the only centroid. ◀

▶ **Lemma 9.** *Given integers $1 \leq d \leq n$, and $k = O(\log n)$. For any k real numbers $x_1, x_2, \ldots x_k$, satisfying $\prod_{i=1}^{k} x_i \leq n$, and $1 \leq x_i \leq d$ for every i, $\sum_{i=1}^{k} x_i = O(d \log_d(n))$*

Proof. Consider any valid tuple $(x_1, x_2, \ldots x_k)$ which maximizes the sum. There can be at most $\log_d n$ values equal to d as the product is $\leq n$, and all values are ≥ 1. If there were two values p, q both in the range $(\sqrt{d}, d)$, we could have replaced them by $d, \frac{pq}{d}$, getting a strictly greater sum. So, there is at most one value in the range $(\sqrt{d}, d)$. Thus, the sum is $\leq k\sqrt{d} + d + d \log_d n = O(\sqrt{d} \log n) + d + d\frac{\log n}{\log d} = O(d \log_d n)$. ◀

Let $\mathcal{O}$ be an oracle that answers path queries with **YES** or **NO** (given vertices x, y, is x an ancestor of y in the rooted directed tree?). Say the root of the tree is already known. For a deterministic algorithm D, let $c(D, I)$ be the number of queries that D asks $\mathcal{O}$ to reconstruct a rooted directed tree I.

We will now consider a different oracle that can (a finite number of times) refuse to answer the path query and instead give some other information for free (without charging the current query to the cost). We will prove that any lower bound that can be proved for such an oracle can be extended to $\mathcal{O}$ as well. Formally, let $\mathcal{O}'$ be an oracle that, given two vertices x, y returns a pair (P, Q), where P is **YES/NO/IGNORE**, and Q is any node of the tree that is an ancestor of y. Call such a response to a query *charged* if P equals **YES/NO** and *free* otherwise. $\mathcal{O}'$ can give at most n^2 *free* responses, where n is the number of vertices in the tree. For a deterministic algorithm D', let $c'(D', I)$ be the number of **charged** responses given by $\mathcal{O}'$ while D' recovers I.

▶ **Lemma 10.** *For any deterministic algorithm D that queries $\mathcal{O}$, there exists a deterministic algorithm D' that queries $\mathcal{O}'$, such that for any tree I, $c'(D', I) = c(D, I)$.*

Proof. Given D, let us construct an algorithm D' such that $c(D, I) = c'(D', I)$ for all I. Consider a query made by D to O, with a response P, that is either **YES** or **NO**. D' simply keeps on making that same query to $\mathcal{O}'$ until it gets a charged response, say (P', Q'). Clearly P' must be equal to P, and D' can choose to ignore Q'. So, D' gets one charged response for every query made by D. ◀

This proof can also be extended to the case where the definitions of both $c(D, I)$ and $c'(D', I)$ are changed to the number of **NO** responses received by the algorithm.

B Algorithm Pseudocode

Algorithm 3 GET-EDGES.

```
Input  : A set $T$ of nodes that form a tree $\tau = (T, \succ)$ with $\succ$ unknown, and an oracle that
         answers for any pair of nodes $a \in T, b \in T$ whether $a \succ b$.
Output : The set of the $|T| - 1$ edges of $\tau$
$n \leftarrow |T|$
if $n \leq 1$ then
    return {}
end
while true do
    $c \leftarrow$ GET-PROBABLE-PSEUDO-CENTROID($T$)
    $(b, S) \leftarrow$ TEST-PSEUDO-CENTROID($T, c$)
    if $b =$ true then
        $E \leftarrow \emptyset$
        for $X \in S$ do
            $E \leftarrow E \cup$ GET-EDGES($X$)
            Let $X = y_1, \dots, y_m$
            if $c \succ y_1$ then
                $E \leftarrow E \cup \{(c, \max_{y \in X} y)\}$
            end
            else
                $E \leftarrow E \cup \{(\min_{y \in Anc_c} y, c)\}$
            end
        end
        return $E$
    end
end
```