Brief Announcement: Reconstructing Binary Trees in Parallel

Ramtin Afshar
Univ. of California, Irvine
afsharr@uci.edu

Michael T. Goodrich
Univ. of California, Irvine
goodrich@uci.edu

Pedro Matias
Univ. of California, Irvine
pmatias@uci.edu

Martha C. Osegueda
Univ. of California, Irvine
mosegued@uci.edu

ABSTRACT

We study the parallel query complexity of reconstructing binary trees from simple queries involving their nodes. We show that a querier can efficiently reconstruct a binary tree with a logarithmic number of rounds and quasilinear number of queries, with high probability, for various types of queries.

KEYWORDS

tree reconstruction; phylogenetic trees; parallel algorithms

1 INTRODUCTION

Binary trees are ubiquitous in computational applications, including search trees and phylogenetic trees, the latter of which are rooted binary trees that represent evolutionary relationships among a group of organisms. In this paper, we study how to learn binary trees in terms of a ***query-complexity*** measure, $Q(n)$, which is the total number of queries of a certain type needed to reconstruct a given tree. Previous work in this area has focused on sequential reconstruction problems, where queries are issued and answered one at a time. For example, in pioneering work for this research area, Kannan *et al.* [1] show that an n-node binary phylogenetic tree can be reconstructed sequentially from $O(n \log n)$ three-node ***relative-distance*** queries, $closer(x, y, z)$, which is given three leaf nodes, x, y, and z, and the response is a determination of which pair, (x, y), (x, z), or (y, z), has the lowest common ancestor (lca). In this paper we are interested in parallel binary tree reconstruction. To this end, we use a ***round-complexity*** parameter, $R(n)$, which measures the number of rounds of queries needed to reconstruct a tree such that the queries issued in any round comprise a batch of independent queries.

We show that an n-node rooted binary tree can be reconstructed from three-node ***relative-distance queries*** with $R(n)$ that is $O(\log n)$ and $Q(n)$ that is $O(n \log n)$, with high probability. We also show that such a tree can be reconstructed from ***path queries***, which ask whether a given node, u, is an ancestor of a given node, w, with $R(n)$ that is $O(\log n)$ and $Q(n)$ that is $O(n \log n)$, w.h.p. This improvies the previous best bound for $Q(n)$, due to Wang and Honorio [2], who had a $Q(n)$ bound of $O(n \log^2 n)$.

SPAA '20, July 15–17, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400229

2 PARALLEL RELATIVE-DISTANCE QUERIES

Our first parallel reconstruction algorithm uses a randomized divide-and-conquer approach. In our case, the division process is random three-way split through a vertex separator, rather than a separator-based binary split. Initially, all leaves belong to a single partition, L. Then two leaves, a and b, are chosen uniformly at random from L and each remaining leaf, c, is queried in parallel against them using relative-distance queries. Notice that the lowest common ancestor of a and b splits the tree into three parts. Given a and b, the other leaves are split into three subsets (R, A, and B) according to their query results (shown in Fig. 1(a)):

- A: leaves close to a, i.e., for which $closer(a, b, c) = (a, c)$
- B: leaves close to b, i.e., for which $closer(a, b, c) = (b, c)$
- R: remaining leaves, i.e., for which $closer(a, b, c) = (a, b)$

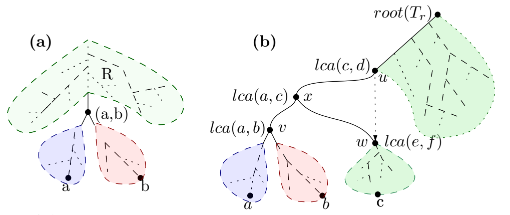

Figure 1: (a) The subgroups leaves are split into. (b) A graph detailing how to attach T_a, T_b and T_r.

We then recusively construct the trees, T_a, for $A \cup \{a\}$, T_b, for $B \cup \{b\}$, and T_r, for R, in parallel. The remaining challenge, of course, is to merge these trees to reconstruct the complete tree, T. The subtree of T formed by subset $A \cup B$ is rooted at an internal node, $v = lca(a, b)$; hence, we can create a new node, v, label it "$lca(a, b)$" and let T_a and T_b be v's children. If $R = \emptyset$, then we are done. Otherwise, we need to determine the parent of v in T; that is, we need to link v into T_r. To identify the parent of v in T, let us assume inductively that each internal node in T_r has a label "$lca(c, d)$," since we recursively label each internal node in T with such a label. The crucial observation is to note if there exists an edge $(u \longrightarrow w)$ in T_r, such that u is labeled "$lca(c, d)$" and $closer(a, c, d)$ is (a, z) for $z \in \{c, d\}$, and w is either leaf z or an ancestor of z labeled"$lca(e, f)$" with $closer(a, e, f) = (e, f)$, (See Figure 1(b)). If $(u \longrightarrow w)$ exists then this edge must be where the parent of v belongs in T, and if there is no such edge, the parent of v is the root of T and the sibling of v is the root of T_r. Thus, we can determine the edge $(u \longrightarrow w)$ by performing a query, $closer(a, c, d)$, for each each node v in T (where the label of v is "$lca(c, d)$") in parallel.

Theorem 2.1. *we can reconstruct an n-node binary tree with $O(n \log n)$ relative-distance queries in $O(\log n)$ rounds w.h.p.*

3 PARALLEL PATH QUERIES

Let T be a degree-d tree, for $d = 2$ (we use the parameter, d, here so we can give a more general algorithm in the full paper). Define a ***path query***, $path(x, y) = 1$, iff x is an ancestor of y in T. Our approach to reconstructing T using parallel path queries is to use a separator-based divide-and-conquer strategy, that is, find a "near" edge-separator in T, divide T using this edge, and recurse on the two remaining subtrees in parallel. The difficulty, of course, is that the querier has no knowledge of the edges of T; hence, the very first step, finding a "near" edge-separator, is a bottleneck computation. Fortunately, as we show in the following lemma, if v is a randomly-chosen vertex, then, with probability depending on d, the path from root r to v includes an edge-separator.

Lemma 3.1. *Let $T = (V, E, r)$ be an rooted tree of degree d and let v be a vertex chosen uniformly at random from V. Then, with probability at least $\frac{1}{d}$ an even-edge-separator is one of the edges on the path from r to v.*

Definition 3.2. (splitting-edge) In a degree-d rooted tree, an edge $(parent(s), s)$ is a splitting-edge if $\frac{|V|}{d+2} \le |D(s)| \le \frac{|V|(d+1)}{d+2}$.

Our algorithm assumes the existence of a randomized method, find-splitting-edge, which returns a splitting edge in T, with probability $\Omega(1/d)$, and otherwise returns *Null*. Our reconstruction algorithm is therefore a randomized recursive algorithm that takes as input a set of vertices, V, with a (known) root vertex $r \in V$, and returns the edge set, E, for V. At a high level, our algorithm is to repeatedly call the method, find-splitting-edge, until it returns a splitting-edge, at which point we divide the set of vertices using this edge and recurse on the two resulting subtrees.

In more detail, during each iteration of a **while** loop, we choose a vertex $v \in V$ uniformly at random. Then, we find the vertices on the path from r to v and store them in a set, Y, using the fact that a vertex, z, is on the path from r to v if and only if $path(z, v) = 1$. Then, we attempt to find a splitting-edge using function find-splitting-edge. If find-splitting-edge is unsuccessful, we give up on vertex, v, and restart the **while** loop with a new choice for v. Otherwise, find-splitting-edge succeeded and we cut the tree at the returned splitting-edge, (u, v). All vertices, $z \in V$, where $path(v, z) = 1$ belong to the subtree rooted at v, thus belonging to V_1. Whereas the remaining vertices belong to V_2 and the partition containing both u and rooted at r. Thus, after cutting the tree we recursively reconstruct-rooted-tree on V_1 and V_2.

The main idea for our efficient tree reconstruction algorithm lies in our find-splitting-edge method, which we describe next. This method takes as input the vertex v, the vertex set Y, (comprising the vertices on the path from r to v), and the vertex set V. With probability depending on d, the output of this method is a splitting-edge; otherwise, the output is *Null*. Our algorithm performs a type of "noisy" search in Y to either locate a likely splitting edge or return *Null* as an indication of failure.

Our find-splitting-edge algorithm consists of two phases. We enter **Phase 1** if the size of path Y is too big, i.e., $|Y| > |V|/K = \frac{|Y|}{C_2 \log |V|}$, where C_2 is a predetermined constant and $K = C_2 \log |V|$. The purpose of this phase is either to pass a shorter path including an even-edge-separator to the second phase or to find a splitting-edge in this iteration. The search on the set Y is noisy, because it involves random sampling. In particular, we take a random sample S of size $m = C_1\sqrt{|V|}$ from path Y (where C_1 is a predetermined constant). We include r and v, the two endpoints of the path Y, to S. Then, we estimate the number of descendants of s, $D(s)$, for each $s \in S$. To estimate this number for each $s \in S$, we take a random sample X_s of K elements from V and we perform queries to find $count(s, X_s)$. Here, we use $m{\cdot}K \in O(\sqrt{|V|}\log|V|)$ queries in a single round. Then, if all the estimates were less than $K/(d + 1)$, we return *Null* as an indication of failure (we guess that all the nodes on the path Y have too few descendants to be a separator). Similarly, if all the estimates were greater than $\frac{Kd}{d+1}$, we return *Null* (we guess that all the nodes on the path Y have too many descendants to be a separator). If there exists a node s such that $\frac{K}{d+1} \le count(s, X_s) \le \frac{Kd}{d+1}$, we check if s is a splitting-edge by counting its descendants using verify-splitting-edge. This function takes vertex s and the full vertex set V to return edge (find-parent$(s, V), s)$ if it is s splitting-edge and return *Null* otherwise.

If neither of these three cases happens, we perform queries to sort elements of S using a trivial quadratic work parallel sort which takes $O(m^2) \in O(|V|)$ queries in a single round. We know that two consecutive nodes w and z exist on the sorted order of S, where $count(w, X_w) > \frac{Kd}{d+1}$ and $count(z, X_z) < \frac{K}{d+1}$. We find all the nodes on Y starting at w and ending at z, and use this as our new Y. In **Phase 2**, we expect a path of size under $|Y|/K$, which is true with high probability. Otherwise, we just return *Null*. In this phase, we estimate the number of descendants much like we did in the previous phase, except the only difference is that we estimate the number of descendants for all the nodes on our new path Y. If there exists a node $s \in Y$ such that $\frac{K}{d+1} \le count(s, X_s) \le \frac{Kd}{d+1}$, we verify if it is a splitting-edge, as described earlier.

Finally, let us describe how we find the parent of a node s in V. We first find, Y, the set of ancestors of v in V in parallel using $|V|$ queries. Let $x > y$ describe the total order of nodes in path Y, where for any $x, y \in Y : x > y$ if and only if $path(x, y) = 1$. The parent of s is the lowest vertex on this path. Then, the key idea is that if $|Y| \in O(\sqrt{|V|})$, then we can sort the elements using $O(|V|)$ queries. If the path is greater than this amount, then we take S, a sample of size $O(\sqrt{|V|})$ from the path. Next, we sort this sample to obtain $x_1 < \cdots < x_m$ for S and then find all of the nodes in Y which are less than the smallest sample x_1. Finally, we replace Y with these descendants and repeat the whole procedure once again. We can prove that if we repeat this sampling idea, then with high probability after only two iterations of sampling, the size of the path is $O(\sqrt{|V|})$, and therefore, all the nodes of Y can be sorted to return the minimum.

Theorem 3.3. *We can reconstruct an n-node binary tree with $O(n \log n)$ path queries in $O(\log n)$ roundes w.h.p.*

REFERENCES

[1] Sampath K. Kannan, Eugene L. Lawler, and Tandy J. Warnow. 1996. Determining the Evolutionary Tree Using Experiments. ***Journal of Algorithms*** 21, 1 (1996), 26–50. https://doi.org/10.1006/jagm.1996.0035

[2] Zhaosen Wang and Jean Honorio. 2019. Reconstructing a Bounded-Degree Directed Tree Using Path Queries. In ***57th IEEE Allerton Conference on Communication, Control, and Computing***. See also arxiv.org/abs/1606.05183.