ALGORITHMIC RECONSTRUCTION OF TREES

Michari Tarsi
Judea Pearl

December 1984
Report No. CSD-840061

UCLA-
December 1984

Algorithmic Reconstruction of Trees

Michael Tarsi and Judea Pearl

Cognitive Systems Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024

(judea@UCLA-LOCUS.arpa)

This work was supported in part by the National Science Foundation, Grant #DSR 83-13875

Algorithmic Reconstruction of Trees

Michael Tarsi and Judea Pearl

ABSTRACT

We give an algorithm to reconstruct a tree out of local relationships among its leaves. For rooted trees the following information is available: On any three leaves u, v and w we can test whether the deepest common ancestor of u and v is or is not on the path from the root to w. For unrooted trees we perform tests on groups of four leaves and identify which pairs are connected by disjoint paths. In both cases we show that a tree with n leaves and bounded maximal degree can be reconstructed using $0(n \log n)$ tests. If the degree is not bounded it takes $0(n^2)$ tests.

Algorithmic reconstruction of rooted trees

M. Tarsi and J. Pearl

Let T be a rooted tree with n leaves $x_1, x_2 \cdots, x_n$. A leaf x_i is said to be the leader of the triple (x_i, x_j, x_k) if the path from x_i to x_j contains the deepest common ancestor of x_j and x_k. If a triple does not have a leader then the common ancestor of all three leaves is also the common ancestor of any two of them.

We present here an algorithm to reconstruct a tree where the available information is the leader (if there exists any) of every triple of leaves. We will try to minimize the number of triples for which we ask who the leader is.

In order to state the algorithm we first make the following observation.

Lemma 1

Let k be the maximum number of sons of a node in a rooted tree T with n leaves. There exists a node v of T such that $\frac{n}{k+1} < des(v) \le \frac{nk}{k+1}$ where $des(v)$ is defined to be the number of leaves which are descendant of v, and $des(v)=1$ if v is a leaf

Proof

Let v_o be the root of T. Define v_{i+1} to be that son of v_i which has the largest $des(.)$ value among all the sons of v_i. We thus defined a sequence $v_0, v_1, \cdots v_i, v_{i+1}, \cdots v_m$ where the last term (v_m) is a leaf. Let v_j be the first node in the sequence $v_1, v_2, \cdots v_m$ with $des(v_j) \le \frac{kn}{k+1}$. v_j does exist, because $des(v_o) = n$ and $des(v_m) = 1$. Now, from

$$des(v_{j-1}) > \frac{kn}{k+1}$$

we obtain

$$des(v_j) \geq \frac{des(v_{j-1})}{k} > \frac{n}{k+1}$$

as required.

The algorithm:

Let T be a rooted tree with leaves $x_1, x_2, \cdots x_n$. Every node of T which is not a leaf has at least 2 and at most k sons. Our algorithm constructs a sequence of trees $T_2, T_3, \cdots T_n$, where T_2 is the tree (space for diagram) $T_n = T$ and T_{i+1} is obtained by adding x_{i+1} as a new leaf to T_i. T_i would be the subtree of T containing the leaves $x_1 \cdots x_i$ where a non leaf node which does not have any sons is removed and any node which remains with just one son is replaced by an edge joining the son directly to its father. The location where x_{i+1} should be added to T_i is found in the following "binary search-like" algorithm.

Procedure add (integer i) Begin

1. $T_c = T_i$ (T_c is a subtree of T_i to which x_{i+1} is to be added. It becomes progressively smaller by eliminating those sections of T_i known not to contain x_{i+1} (statements 8, 9, and 10).

2. s: = the number of leaves in T_c.

3. If $s=2$ let $\bar{v}$ be the root of T_c and x_j, x_k its two leaves.

4. If $s>2$ select as $\bar{v}$ any node of T_c for which $\frac{s}{k+1} < des(\bar{v}) \leq \frac{sk}{k+1}$ (lemma 1) and let x_j, x_k be two leaves whose common ancestor is $\bar{v}$.

5. Ask for the leader of the triple (x_{l+1}, x_j, x_k) (with respect to T).

6. If $s > 2$ then begin.

7. Define a partition of T_c into 2 subtrees: T_{c_1} rooted at $\bar{v}$ with all the descendants of v and $T_{c_2} = T_c - T_{c_1}$ in which $\bar{v}$ is considered a leaf.

8. If x_{l+1} is the leader of (x_{l+1}, x_j, x_k) then set $T_l = T_{c_2}$.

9. If there is no leader, set $T_c = T_{c_1}$ from which the 2 sons of $\bar{v}$ whose descendants are v_k and v_j are removed with all their descendants.

10. If x_j (or x_k) is the leader, set T_c = the subtree of T_{c_1} rooted at that son of $\bar{v}$ which is the ancestor of x_k (or x_j, respectively).

11. GO TO 2 END

12. If s=2 then begin

13. If x_j (or x_k) is the leader add a new node on the edge of x_k or x_j, respectively, and make it the father of x_{l+1}

14. If x_{l+1} is the leader add a new root and make x_{l+1} and the old root $\bar{v}$ his sons.

15. If there is no leader, make x_{i+1} a son of $\bar{v}$.

16. END END Add

Complexity Analysis

Whenever the procedure **add** is applied to construct T_{i+1} out of T_i it starts to search on a tree with i leaves. After each leadership test (statement 5) the search proceeds on a sub tree which might contain at most a fraction $\frac{k}{k+1}$ of the leaves of the previous subtree. Thus, the number of steps (leadership tests) can be at most $\log_{\frac{k+1}{k}}(i)$.

The complexity of the entire algorithm is the sum of this amount over $i=2, 3, \cdots n = \log_{\frac{k+1}{k}}(n!) = 0\,(n\log n)$ for every fixed number k. However, if the degree k is not bounded, the construction of T_{i+1} out of T_i might take up to i steps which leads to a total complexity of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = 0\,(n^2)$. This upper bound will actually be achieved in a star-like tree, where all n leaves are sons of the root.

The number of different binary (and thus any fixed $k \geq 2$) trees on n labeled leaves can be lower bounded by $n!$ using the following construction: Take a simple path $a_{1,} a_2 \cdots a_n$, make a_1 the root, and for every permutation $P = X_{i1}, X_{i2} \cdots X_{in}$ construct a binary tree $T(P)$ making every x_{i_j} the son of a_j. This shows that spending $0\,(n \log n)$ tests is the best possible for this kind of problem. (Every leadership test provides one of four possible answers and this gives two bits of information.)

The number of trees possible in the case where k is not bounded can be estimated as follows: Since no node of T has just one son, the total number of nodes in T is less than twice the number of leaves--$2n$. On $2n$ labeled nodes there exist $2n^{2n-2}$ different spanning trees, thus $2n^{2n-2}$ is an upper bound to our tree-counting problem. To identify one of these spanning trees would require at least $\log(2n^{2n-2})$ tests, which is still $0(n \log n)$; thus, our algorithm, with $0\,(n^2)$, is not guaranteed optimality in this case.