A Logspace Algorithm for Tree Canonization

(Extended Abstract)

Steven Lindell †
Haverford College
Haverford, PA 19041-1392

Abstract

We present a solution to the problem of assigning to each directed tree T of size n a unique isomorphism invariant name for T, using only work space $O(\log n)$. Hence, tree isomorphism is computable in logspace. As another consequence, we obtain the corollary that the set of logspace computable queries (*Lspace*) on trees is recursively enumerable. Our results extend easily to undirected trees and even forests.

1. Introduction

In this relatively short presentation, I will outline a logarithmic space algorithm for computing canons of trees. As a trivial corollary, this results in a logarithmic space isomorphism test for trees. As a non-trivial corollary, we obtain a recursively enumerable logic for L- and NL-computable queries on trees. The significance of this fact is elaborated upon at the end of the paper. There also, we show how to extend our results to include undirected trees and forests.

In [AHU, p. 84], a linear time isomorphism algorithm for trees is given based on the classical notion of vertex refinement, which lends itself particularly well to a bottom-up implementation on trees. Subsequent attempts to reduce the complexity of the tree isomorphism problem have concentrated on minimizing the space or the parallel time. In [Ruz], an auxiliary logspace PDA is claimed to solve the problem for trees of $O(\log n)$ bounded degree, implying that isomorphism is in NC. Using the newly developed fundamental technique of tree contraction, [M&R] extend this to trees of arbitrary branching, and give a deterministic $O(\log n)$ parallel time algorithm. Our $O(\log n)$ deterministic space algorithm improves this further, and it introduces what may be a new "programming" technique for doing recursion without necessarily maintaining a stack (more on this later). This implies that tree canonization (and hence isomorphism) can be computed by an EREW (exclusive-read exclusive-write) PRAM in logarithmic parallel time with polynomially many processors [HTCSA, p. 906].

† Research completed while the author was on leave in residence 1990-91, and also partially supported by NSF grant CCR-9003356. Electronic mail address: s_lindell@acc.haverford.edu; telephone: (215) 896-1203.

24th ANNUAL ACM STOC - 5/92/VICTORIA, B.C., CANADA
© 1992 ACM 0-89791-512-7/92/0004/0400...$1.50

2. Input, Output, & Resources

Unless specified otherwise, we assume all trees are **directed**, and hence have a unique **root**. In all figures, the root is shown at the top, with all edges directed downwards.

The **tree** T is presented on a *read-only input* tape as a directed graph of size n, the edges of which are a binary relation on the set of nodes labeled $1, \ldots, n$. There is also an $O(\log n)$ size *read/write work* tape. A unique binary string called the **canon**, $c(T)$, which encodes the information for the edges of the tree, will be put onto a *write-only output* tape. Two trees are isomorphic if and only if they receive identical canons. Before defining the particulars of this canon, we begin with the related notion of a total linear order of tree isomorphism classes.

3. Definitions

Notation: We will use capital letters to denote an entire tree or subtree (for instance T), and the corresponding lower case letter (for instance t) to denote its root, and conversely. We shall identify (for convenience purposes) a subtree with its root, and it will be clear from context which one is being referred to.

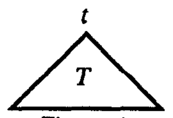

Figure 1

An **edge** from s to t is written as $s \rightarrow t$. Also, $s < t$ means the **label** of s is less than that of t. This arbitrary ordering, induced by the presentation of the nodes for input, will be used in a crucial way to sequence the algorithm even though the labeling must be ignored in the final answer.

Let $|T|$ = the number of **nodes** in the tree T (including the root).
Let $\#t$ = the number of **children** of the node t (its out-degree).

Ordering Trees

Definition 1: Let S and T be two trees. We determine *tree isomorphism order*, $S < T$, by first comparing sizes, then number of children, and then finally a recursive ordering of the children. Define $S < T$ if:

A. $|S| < |T|$, or

B. $|S| = |T|$ and $\#s < \#t$, or

C. $|S| = |T|$, $\#s = \#t = k$, and $(S_1, \ldots, S_k) < (T_1, \ldots, T_k)$ lexicographically, where we inductively assume $S_1 \leq \ldots \leq S_k$ and $T_1 \leq \ldots \leq T_k$ are the ordered subtrees of S and T.

If we can satisfy the definition by reversing the roles of S and T, then we say $S > T$. Otherwise, if neither $S < T$ or $S > T$, we say $S = T$ (that is, they are *isomorphic*, $\cong$). We leave it to the reader to verify that this is a total linear order on tree isomorphism classes (but this is fairly obvious).

Tree Canons

Infix notation, where the embedded subexpressions are ordered, serves well as a fairly standard canonical representation of trees. It is defined inductively, but actually computed only at the end of the paper.

Definition 2: The canon of a tree, $c(T)$, is defined to be a string over the set of symbols $\{[,], \bullet\}$. We shall use juxtaposition to denote string concatenation.

Basis:	The canon of the one node tree is $\bullet$.
Induction:	The canon of a tree T with subtrees ordered $T_1 \leq \ldots \leq T_k$ is $[\, c(T_1) \ldots c(T_k)\,]$.

Example: The canon of the following tree is [•[••]].

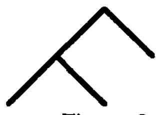
Figure 2

This clearly provides an isomorphism invariant name for each tree in a natural way.

4. The Algorithm

Logspace Tree Traversal

The algorithm will depend heavily on a simple technique for depth first traversal of a tree using only logarithmic space. To illustrate this technique we employ three functions, each of which is easily computed in logspace. In the descriptions below, the node t is identified by its label, and the *order* referred to must be some logspace computable total linear order on the set of labels (it is not actually passed, but recomputed on an as needed basis). We will be traversing trees in different ways, depending on the given order, but the basic skeleton will remain the same. Refer to the following figure:

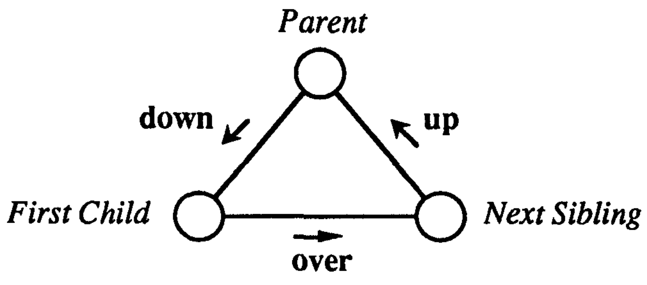

Figure 3

Function:	**Description:**
Parent(t):	The function which returns the label of the *parent* of t, if it exists, and 0 otherwise (when t is the root). To implement, search through all nodes looking for the unique node p such that $p \rightarrow t$.
First Child(t, *order*):	The function which returns the *first child* of t (with respect to *order*) if it exists, and 0 otherwise (when t is childless). To implement, search all nodes for the smallest node c (using the given order) such that $t \rightarrow c$.
Next Sibling(t, *order*):	The function which returns the *next sibling* of t (with respect to *order*) if it exists, and 0 otherwise (when t is the biggest child in the family). To implement, search all nodes for the smallest node c larger than t (using the given order) such that *parent*(t) $\rightarrow c$.

To compute $\#t$, the number of children of any node t, we call *first child* followed by *next sibling* repeatedly until failure. The total number of successful calls is the answer.

Depth-first traversal of T is based on the three procedures below.

Procedure:	**Description:**
down:	go down to the *first child*, if it exists.
over:	move over to the *next sibling*, if it exists.
up:	back up to the *parent*, if it exists.

To perform a depth-first traversal, we need only remember our initial position (for termination), and our last move (to prevent retracing). In the beginning, or if our last move was **down** or **over** (i.e. we are visiting a new node), then we call **down**, **over**, or **up** (the first one to succeed in that priority). Otherwise, our last move was **up** (backtracking), and we call **over** or **up** in priority. We terminate upon returning to our initial position, which requires space log n to store.

To compute $|T|$, the number of nodes in any subtree T, we execute the traversal procedure starting at t, ordering nodes simply by their labels, and adding one to a global counter (initialized to 1) each time a new node is visited. Alternatively, we could just cycle through all nodes, and add one to a global counter every time we encounter a descendant of t. Node p is a descendant of t if repeated application of *Parent*(p) eventually yields t.

Outline of the Comparison Algorithm

Our goal is to compare two trees to determine their tree isomorphism order: $S < T$, $S > T$, or $S = T$. Even though many parts of the algorithm run in linear time, the entire algorithm will most certainly not, as one of the consequences of working in logarithmic space is the necessity of recomputing things many times in order to conserve space.

Compare sizes

Given two trees, S and T, we first compare their sizes (Definition 1, part A). If they are not equal in size, $|S| \neq |T|$, then answer appropriately. If they are equal in size, $|S| = |T|$, then continue by comparing the number of children of their respective roots (Definition 1, part B). Again, the answer is determined unless they have the same number of children, $\#s = \#t$, in which case we proceed to lexicographic comparison of the children (Definition 1, part C).

Break into Blocks

Next, we check that S and T have the same size children, in the hope that we can determine an inequality without having to use recursion. To start, count the number of minimal size children of S, and do the same for T. If S has more, then $S < T$. If T has more, then $S > T$. Otherwise, remember that minimal size, and count the number of children of S of the next biggest size, and do the same for T. Compare the results in a similar fashion, and continue until an inequality between sizes is found, or else the children of S and T are exhausted. Using this method, we can imagine that the children are partitioned into blocks of increasing size (recomputation will often be necessary here). From now on, we assume that S and T have an equal number of children, all of whose subtrees are of matching cardinalities. So the stage is set for recursive comparison.

Recursive Comparison

To do this, we will simultaneously traverse S and T using cardinality as our order (and labels to resolve any ambiguity). One major difference is that we may retraverse the children of a node many times, going from the first child to the last child and then back again to the first child.

Up to now, there has been no use of recursion, which means that the calling graph of our procedures is acyclic. Hence use of a stack for local variables (the environment) and the program counter (the instruction pointer) results in an increase in storage of only $O(\log n)$. But that will all change here, where we are going to make use of recursion in a special limited way.

Storage space limitations will prevent us from recursively comparing two children of S or two children of T (we would have to maintain a stack of passed parameters). Rather, we use a *two-pronged* depth-first search and only *cross-compare* a child of S with a child of T. The dual positions will be maintained at matching levels and keep track of the trees' roots. To affect a recursive comparison call between a subtree S' of S and a subtree T' of T, we simply move two pebbles from the configuration (s, t) to the children (s', t') to enter the called environment. In order to return to the original calling environment, we simply move the two pebbles back to their parents. In this way, we do not need to maintain a stack of passed parameters (nodes). This is similar to using a "GOSUB" type of calling mechanism for subroutines.

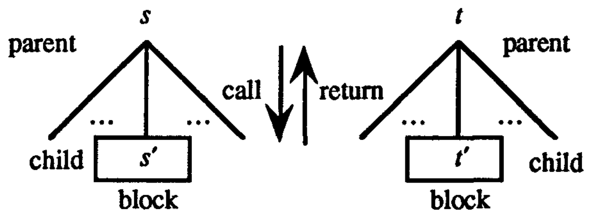

Figure 4

We will work only within the equicardinality blocks consisting of children of S and children of T. We can maintain the current block size by recomputing it upon return from the pair (s', t') by remembering our last position: *block size* $= |S'| = |T'|$.

Sequencing and Storage

Start with the block of minimal size children. If no lexicographic inequality (in tree isomorphism order) is determined within this block using recursive cross-comparisons, we proceed to the next equicardinality block, and continue in this fashion.

If the block under consideration has $k \geq 2$ children, then by the equicardinality condition, any recursive call has size at most n / k. This allows for $O(\log k)$ bits of local storage to be maintained on a stack, which will permit us to have any fixed number of counters whose range is from 1 to k (for loops). If $k = 1$, then no space is available, but in this case no local storage will be necessary. We do this by employing a form of tail recursion, which can be eliminated by using the "GOTO" construct of reputed infamy, where program flow of control jumps to a new location without a recollection of where it came from (but we shall always affect a "RETURN" despite this). Its use appears crucial in our situation.

Lexicographic Determination

This is the heart of the algorithm. It may come as a surprise that the lexicographic comparison in part C of Definition 1 can be determined only from the set of cross-comparisons between children of S and children of T, without ever directly comparing two siblings. It is even more surprising that this can be achieved in logspace.

There are several cases, some of which are redundant, but enhance the clarity of presentation. Let $n = |S| + |T|$, and let $k =$ the number of children in the equicardinality block under consideration. It should be noted that the "worst-case" occurs when all the children of S and T are all the same size (and hence $k = \#s = \#t$, the same k as in part C of Definition 1).

$k = 0$: There are no children of S and T, and hence we answer $S = T$, since all one node trees are isomorphic. There is no recursion at all.

$k = 1$: Each of S and T have only one child of the given size – call them S' and T'. The result of comparing S' and T' is the answer we desire. Since $|S'| + |T'| \leq n - 2$, we have no space to record anything when we call the recursive comparison between S' and T'. So upon returning, we then look back down at where we came from, and realize (by counting) that there was only one child of that size. We then know that the recursive call was made from this very case (and there is only one such call here), and hence can continue execution back in the calling environment, from the point we left off (right here). If S' and T' were unequal, then no further computation is necessary, and we simply pass this answer along for the original comparison between S and T. If $S' = T'$, then we move on to the next block.

$k = 2$: Let S', S'' and T', T'' be the two children of S and T in the given size block under consideration. We claim that the outcome of the four cross comparisons determines the lexicographic comparison desired (this requires proof – see general case). Since $|S'| = |S''| = |T'| = |T''| \leq n/4$, we will have $O(1)$ bits of local storage with which to work with. These bits can be used to keep track of the calls that have been completed (by pushing return addresses on a stack), and their results. A little reflection, though, will reveal that the return addresses are unnecessary, since it will be possible to sequence the comparisons by simply going to the next largest (with respect to labels) pair (s'', t'') upon returning from the pair (s', t').

$k \geq 2$: This case subsumes the above case of $k = 2$, but is substantially more complicated. The key idea will be a function which computes the *order profile* of a child c in the block, which is the number of <'s, >'s, and ='s which are tabulated by a comparison of c with all the k children in the corresponding block of the other family (place c on the L.H.S. of all comparisons for definiteness). This requires $O(\log k)$ of local storage, both to record the three numbers, and to sequence the cross-comparisons. Recomputation will be necessary.

First, we search for a child in each family with minimal order profile, that is, one with no >'s. If T has no such child, then there is a child of S which is smaller than all the children in T, and its order profile contains all <'s. This forces $S < T$, even though the child under consideration may not be the smallest child. If the reverse situation occurs, then this forces $S > T$. Otherwise, there is a child S' of S with no >'s, and a child T' of T with no >'s, hence they must be isomorphic to each other by minimality, $S' = T'$. But there may be many minimal children, so we must compare the number of ='s in the respective order profiles of S' and T', which tells us the number of minimal children in each family (things equal to the same thing are equal to each other). If S' has more ='s, then T has more minimal children, and hence $S > T$. The reverse situation yields $S < T$. Otherwise, S and T have the same number of minimal children, and it will be necessary for us to compare larger children in the same block. We do this by taking the number of ='s in the last step (the number of minimal children) and use this number to set a *threshold*, h. This threshold is then used to search for the next minimal children of S and T, by ignoring up to h >'s in their order profile (this is like dropping the h lowest scores). The process is then repeated as above, and the threshold is incremented each time until reaching k, at which point we proceed to test the next equicardinality block.

If we reach the end of all the equicardinality blocks without uncovering an inequality, the trees must be isomorphic, and we return the answer $S = T$.

5. Results

Canonization

Once we know that tree isomorphism order is logspace computable (using the comparison algorithm, above), it is a simple matter to compute the canon of any tree. To print the canon of a tree, $c(T)$, we traverse T (using the traversal procedure, above) with tree isomorphism order, and print '[' when we go **down** the tree, '•' when we move over in the tree, and ']' when we back **up** the tree. Therefore, we have proved:

Theorem 1: The canon of a directed tree can be computed in deterministic space $O(\log n)$.

Isomorphism

Two trees are isomorphic if and only if they have the same canon. Hence, we obtain the following trivial corollary:

Corollary 1: Directed tree isomorphism is computable in in deterministic space $O(\log n)$.

This logspace isomorphism test for trees, "improves" (in a complexity-theoretic sense) on the best known algorithms, since space $O(\log n)$ is contained in parallel time $O(\log n)$ on an EREW PRAM with polynomial many processors [HTCSA]. Of course, containment is not known to be strict, even for $L \subseteq P$.

Logical Consequences

It is already known that first-order logic together with counting quantifiers and a least-fixed-point operator is a logic for *Ptime*, the polytime computable queries, on trees [D&M]. The main goal of this research was to provide a logic for *Lspace*, the logspace computable queries, on trees. This turns out to be equivalent to showing that the set of logspace computable intrinsic properties on trees (each property given an appropriate finite description) is recursively enumerable. Something is an intrinsic property of trees only if it is isomorphism invariant. The property of being a binary tree is one example.

Let us assume that each logspace computable intrinsic property of trees is given by a logarithmic-space-bounded Turing Machine which takes an *encoding* of a tree as its input (see assumptions at the beginning of this paper), and outputs *yes* or *no*. To compute an intrinsic property, the TM must necessarily be under the further restriction of giving the same answer for different encodings of the same tree up to isomorphism.

To show that the set of logspace computable intrinsic properties on trees is recursively enumerable (r.e.), we must demonstrate a list of logarithmically space bounded machines such that:

- every machine on the list computes an intrinsic property of trees
- every logspace computable intrinsic property of trees is on the list

Call a TM logarithmically *boxed* if some code has been attached to assure that it doesn't exceed a space bound of $k \cdot \log n$. (akin to polynomially clocking). The r.e. list of all logarithmically boxed TM's satisfies the latter condition but not the former. The r.e. logic $FO + DTC$ invented by [Imm] satisfies the former condition but not the latter (here we must recode the formulas into machines). Our main result implies a direct solution satisfying both conditions:

Corollary 2: There is a recursively enumerable logic for *Lspace*, the logspace computable queries, on trees.

Proof: It is sufficient to show that the set of logspace computable intrinsic properties of trees is r.e. We do this as follows:

- *Prefix every logarithmically boxed TM by the logspace tree canonizer.*

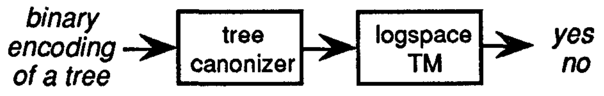

Figure 5

To see that this listing yields a recursive enumeration of all the logspace computable intrinsic properties of trees, one needs to observe that the tree canon can be converted into a standard tree encoding in logspace, and that logspace transducer machines are closed under composition.

6. Extensions

Undirected Trees and Forests

All of the above results are true in the more general context of undirected trees and forests, and very little extra proof is required.

Theorem 2: The canon of an undirected tree can be computed in deterministic space $O(\log n)$.

Proof: In an undirected tree, any vertex can be considered a root. So find a *minimal* vertex r, such that the tree rooted at r is minimal with respect to tree isomorphism order, and mark that as the root using an additional $\log n$ bits of space (if there are several minimal roots, pick the one with the smallest label). Now proceed to compute the ordinary tree canon, and whenever one needs to ask $x \rightarrow y$, check to make sure that x is closer to r than y. (this enforces directedness away from r). The result is a canon of the original undirected tree.

Theorem 3: The canon of a forest can be computed in deterministic space $O(\log n)$.

Proof: In the directed case, it is easy to locate the roots of all the trees. Use a global counter k to keep track of how many trees in the forest have been canonized, and initialize it to 0. Given a value for k, search for a k-minimal tree (one with exactly k other trees below it in tree isomorphism order, using the root labels to resolve ambiguity) in the forest, and canonize (print the canon of) that tree. When complete, increment k by 1, and continue until k reaches the number of roots in the forest. For an undirected forest, just combine this idea with the proof of Theorem 2, being careful to compute k-minimality without counting a tree twice. This can be achieved by always checking to see if a given "root" of an undirected tree is really minimal with respect to all other possible roots of the same tree (the nodes connected it).

Nondeterministic Logspace

It is obvious that the proof of Corollary 2 extends to yield a recursively enumerable logic for *NL*space, the nondeterministic logspace computable queries, on trees. Combined with the ideas above, we obtain logics for L and NL on undirected trees and forests.

7. Directions for Future Work

Binary Tree Isomorphism

Can the results presented be strengthened to put tree canonization in NC^1? In particular, is binary tree isomorphism in NC^1? In studying this, it will likely be necessary to work with the transitive closure of the tree. This is similar to the use of infix expression representation of trees in Buss' work on ALOGTIME (a uniform version of NC^1) [Bus].

A Logic for *Ptime* on Arbitrary Graphs

One of the most outstanding problems in this area of study concerns the existence of an r.e listing of all the *polytime* intrinsic properties of *graphs*. A positive answer to this broader question would indirectly yield a logic for *Ptime* (the polynomial time computable queries), whereas a negative answer would imply $P \neq NP$. There are two reasons why this is true: #1: Graph canonization lies in Δ_2 of the polynomial time hierarchy, and the supposition $P = NP$ would imply $\Delta_2 = P$ which would imply a polytime graph canonization algorithm, and then we can repeat the same machine prefixing trick as above. #2: the set of NP queries is known to possess an r.e. logic, namely $\Sigma_1^{\ 1}$ [Fag], and by Cook's Theorem on NP-completeness, $P = NP$ would imply a constructive correspondence between P and NP, resulting in an r.e. listing of *Ptime*.

8. Acknowledgements

Thanks to Steve Tate for suggesting that this result could extend to forests and unrooted trees, to Dave Barrington for helpful suggestions, to Neil Immerman for his support, to Sam Buss for clarification of the input format of trees for NC^1 computations, and to Mark Fulk for the very last observation in the paper. Also thanks to Loren Rosen for some help on terminology, and the anonymous STOC committee for requesting that the connection with logtime EREW PRAMs be mentioned. A very special thanks to my wife, Suzanne, for so patiently listening to the many discussions of this argument. Without her support, many of the important details might not have been ironed out.

References

[AHU] Aho, Hopcroft, Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Bus] S.R. Buss, "The Boolean formula value problem is in ALOGTIME," 19th ACM STOC Symp. 1987, pp. 123-131.

[D&M] P. Dublish, S.N. Maheshwari, "Query Languages which Express all PTIME Queries for Trees and Unicyclic Graphs," Proc. of MFCS 90, Lec. Notes in Computer Science, Vol. 452, Springer Verlag.

[Fag] R. Fagin. "Generalized First-Order Spectra and Polynomial Time Recognizable Sets," in *Complexity of Computation*, (R. Karp, Ed.) Proc. SIAM-AMS #7 (1974), pp. 43-73.

[Imm] N. Immerman, "Languages that Capture Complexity Classes," SIAM J. of Computing, vol. 16, 1987, pp. 760-778.

[M&R] G.L. Miller & J.H. Reif, "Parallel Tree Contraction Part 2: Further Applications," SIAM J. of Computing, vol. 20, no. 6, (Dec. 91) pp. 1128-1147.

[Ruz] W.L. Ruzzo, "On Uniform Circuit Complexity," J. Comp. Sys. Sci., 21:2 (1981) 365-383.

[HTCSA] *Handbook of Theoretical Computer Science*, Volume A, *Algorithms and Complexity*, edited by J. van Leeuwen, MIT Press, Elsevier, 1990.