Alogtime Algorithms for Tree Isomorphism, Comparison, and Canonization

Samuel R. Buss*

Departments of Mathematics & Computer Science,
Univ. of California, San Diego,
La Jolla, CA 92093-0112

Abstract. The tree isomorphism problem is the problem of determining whether two trees are isomorphic. The tree canonization problem is the problem of producing a canonical tree isomorphic to a given tree. The tree comparison problem is the problem of determining whether one tree is less than a second tree in a natural ordering on trees. We present alternating logarithmic time algorithms for the tree isomorphism problem, the tree canonization problem and the tree comparison problem. As a consequence, there is a recursive enumeration of the alternating log time tree problems.

1 Introduction

A tree is a finite, connected, acyclic graph with a distinguished root node. An *isomorphism* between two trees T_1 and T_2 is a bijection between the nodes of T_1 and the nodes of T_2 which preserves the edges and which maps the root of T_1 to the root of T_2. Two trees are *isomorphic* if there is an isomorphism between them.

An implicit component of our definition of "isomorphism" is that the children of a node in a tree are *unordered*. Of course, whenever a tree is drawn, or especially, is represented in on a computer, there is an ordering specified by the representation of the tree. For instance, we define a string representation of trees below, and any string representation of a tree orders the subtrees and nodes of T. This means of course that a given tree may have many different representations. The question of whether two different string representations are representations of the same tree is called the *tree isomorphism problem.* One way to solve the isomorphism is define canonical representations for trees in such way that every (unordered) tree has exactly one canonical representations. The problem of converting an arbitrary string representation of a tree into a canonical representation is called the *tree canonization* problem.

In addition to the tree isomorphism and tree canonization problems, we will consider the tree comparison problem. Below we define a linear ordering, $\prec$, on trees: the *tree comparison* problem is the problem of, given string representations of two trees, to determine whether the first tree is less than ($\prec$) the second.

* Supported in part by NSF grants DMS-9503247 and DMS-9205181.

The main results of this paper give alternating logtime (Alogtime) algorithms for the tree isomorphism, tree comparison and tree canonization problems. There have been several prior related results. First, Aho-Hopcroft-Ullman[1] gave a linear time algorithm for tree isomorphism, based on comparing two trees in a bottom-up fashion. Obviously, linear time is the best possible sequential run time for tree isomorphism, but it is possible to consider refined algorithms in smaller complexity classes, where one restricts the circuit depth, the parallel time, or the Turing machine space usage. Recall that the complexity class NC is the class of predicates solvable by polynomial size, polylog depth circuits; as is well known, uniform NC algorithms are exactly the algorithms that can be solved on PRAM's in polylogarithmic time with polynomially many processors. Ruzzo [13] mentions an NC-algorithm for solving the tree isomorphism problem for trees of logarithmic degree. Miller and Reif [12] later gave an NC-algorithm for solving the tree isomorphism and tree canonization problems for trees of arbitrary degree and depth, based on tree-contraction methods. Finally, in the best prior result, Lindell [8] gave deterministic logarithmic space algorithms for the tree isomorphism, tree comparison and tree canonization problems. It was the logarithmic space algorithms that motivated the work of this paper to give alternating logarithmic time algorithms for these algorithms.

One application of the Alogtime algorithms presented in this paper is to a question that arises from finite model theory. One would like to consider algorithms that compute *intrinsic* properties of trees, which do not depend on the particular representation of the trees. Let $\mathcal{C}$ be a natural complexity class containing Alogtime (e.g., C may be logspace, nondeterministic logspace, NC^k or Alogtime itself). With our alternating logtime algorithm for tree canonization, one can immediately give a recursive enumeration of all intrinsic tree predicates which are computable in $\mathcal{C}$. Namely, one enumerates the $\mathcal{C}$-algorithms which first create a canonical representation of their input tree and then operate only on the canonical representation. It is clear that every algorithm of this type computes a property of trees which is independent of the representation of the tree, and conversely, every $\mathcal{C}$-algorithm which computes a property of trees independent of their representations is equivalent to an algorithm in this enumeration.

In the next section we give technical definitions regarding trees, including the definition of the linear ordering of trees and the string representation of trees. Section 3 is devoted to the alternating logarithmic time algorithm for tree isomorphism. Then, in sections 4 and 5, we present the algorithms for tree comparison and tree canonization.

It is easy to check that our results apply also to labeled trees, with only relatively minor changes to the definitions and algorithms.

2 Technical Definitions for Trees

The *parent* of a node x in a tree T is the unique node adjacent to x which is closer to the root. The *children* of a node x are the nodes of which x is the parent. A node without any children is a *leaf.* A node x is an *ancestor* of a node y if the shortest path from y to the root contains x; in this case we also say y is a descendent of x. If x is a node of T, there is a *subtree* of T rooted at x, namely the node x plus all of its descendents. (Some authors use the terminology "maximal subtree", but for this paper, a subtree is always maximal.) An *immediate subtree* of T is a subtree whose root is a child of T's root node. If S is a proper subtree of T, then the *parent tree* of S is the subtree of T of which S is an immediate subtree.

A second, more formal, inductive definition of tree isomorphism or tree equality, $\equiv$, is given next. The *size* of T, denoted $|T|$, equals the number of leaf nodes in T.

Definition *(Tree equality). Let S and T be trees. We define $S \equiv T$ by induction on the number of nodes in S and T by defining that $S \equiv T$ holds if and only if*

(a) $|S| = |T| = 1$ *or*
(b) *S and T both have the same number, m, of immediate subtrees, and there is some ordering $S_1, \ldots, S_m$ of the immediate subtrees of S and some ordering $T_1, \ldots, T_m$ of the immediate subtrees of T such that $S_i \equiv T_i$ for all $1 \leq i \leq m$.*

It is easy to check that $S \equiv T$ if and only if there is an isomorphism of S and T. There are many possible ways to define a linear ordering, $\prec$, on trees. The one we give here seems to us to be the most elegant, however, the results of this paper would hold under many other choices for the linear ordering, for instance the linear ordering of [8].

Definition *(Linear ordering of trees). Let S and T be trees. We define $S \preccurlyeq T$ and $S \prec T$ simultaneously by induction on the size of S and T. $S \preccurlyeq T$ holds if and only either $S \prec T$ or $S \equiv T$. $S \prec T$ holds if and only if either $|S| < |T|$ holds or the following conditions hold:*

(a) $|S| = |T|$, *and*
(b) *Let $S_1, \ldots, S_m$ be the immediate subtrees of S ordered so that $S_1 \succcurlyeq S_2 \succcurlyeq S_3 \succcurlyeq \cdots \succcurlyeq S_m$, and let $T_1, \ldots, T_n$ be the immediate subtrees of T, similarly ordered with $T_i \succcurlyeq T_{i+1}$ for all i. Then*
 (b.i) *For some $i \leq \min\{m, n\}$, $S_i \prec T_i$ and for all $1 \leq j < i$, $S_j \equiv T_j$, or*
 (b.ii) *$m < n$ and $T_i \equiv S_i$ for all $1 \leq i \leq m$.*

Following [12], we shall represent trees by strings over the two symbol alphabet containing open and close parentheses. The tree with a single node is denoted by the string "()". If T is a tree with more than one node, if $S_1, \ldots, S_m$ are the immediate subtrees of T (in any order), and if $\alpha_1, \ldots, \alpha_m$ are strings which represent the immediate subtrees, then

$$\text{``}(\alpha_1, \ldots, \alpha_m)\text{''}$$

is a string representing T. Note that different orderings of the subtrees in T can give rise to different string representations for T. Thus trees have more than one string representation, except for highly symmetric trees in which each node has all its children isomorphic.

Definition *The* Tree Isomorphism Problem *is the problem of determining whether two input string representations represent isomorphic trees. The* Tree Comparison Problem *is the problem of, given string representations of two trees* T_1 *and* T_2*, determining whether* $T_1 \prec T_2$*. The* Tree Canonization Problem *is the problem of, given an input string representation of a tree* T*, finding a (usually different) string representation,* $canon_T$*, so that the output* $canon_T$ *is the same for all input string representations of* T*.*

The definition of the tree canonization problem allows considerable flexibility in how $canon_T$ is defined. However, for this paper, we use the following, natural, definition for $canon_T$.

Definition *If* $|T| = 1$*, then* $canon_T$ *is "()", the only string representation of* T*. If* $|T| > 1$*, then let* $S_1, \ldots, S_m$ *be the immediate subtrees of* T *ordered so that* $S_1 \succcurlyeq S_2 \succcurlyeq \cdots \succcurlyeq S_m$*. Then* $canon_T$ *is equal to*

$$\text{“}(canon_{S_1} canon_{S_2} \cdots canon_{S_m})\text{”}.$$

A canonical *string representation is a string which is of the form* $canon_T$ *for some tree* T*.*

It is worth discussing the importance of using string representations of trees. An alternative to string representations would have been a pointer representation, where the nodes of T are assigned integer values and where each node has a pointer to its parent node and a linked list of pointers to its children. The problem with the pointer representation is that we are working with the very weak complexity class of Alogtime, and Alogtime algorithms are incapable of parsing trees specified by pointers (unless Alogtime is equal to logspace). For instance, evaluating the predicate "node x is a descendent of node y" from the pointer representation of a tree is believed to be outside the computational power of Alogtime; in fact, the problem DTC, deterministic transitive closure, can be reduced to the descendent predicate and DTC is known to be complete for logspace [7].

Therefore, we use the more explicit string representation for trees. It is known that Alogtime algorithms are capable of parsing parenthesis languages. In particular, using counting of parentheses, an Alogtime algorithm can determine the depth of a node in a tree, can determine the i-th child of any node in a tree, can compute the ancestor/descendent predicates, etc. In addition, Alogtime algorithms are capable of converting between prefix and infix notations. We will assume that the reader is familiar with these capabilities of Alogtime, and also is familiar with both the circuit characterization and the game characterization of Alogtime. For more information on these aspects of alternating logtime, the reader can consult the introductory portions of [3, 5, 4].

There are logspace algorithms for converting pointer representations of trees into string representations of tree, and vice-versa, so for logspace and for more powerful complexity classes, the use of string representations is equivalent to the use of pointer representations.

Definition *When T is a tree represented by a string, the leaves are ordered by their occurrence in the string; the leaves are numbered from left to right with consecutive integers in the range 0 to $|T|-1$.*

Let S be a subtree of T. The domain, $dom(S)$, is the interval $[i,j]$ where S contains the leaves numbered i through j. We let $mindom(S) = i$, the minimum leaf number in S.

There are Alogtime algorithms which, given the value of $|S|$ and the value $mindom(S)$, can find the depth of the root of S, the last leaf node in S, the parent tree of S, the i-th subtree of S, etc.

For the next section, we will need an Alogtime algorithm which, given a value ℓ and given a leaf x in a tree representation, finds the unique maximum-size subtree S such that x is in the domain of S and $logsize(S) = \ell$ (if any such subtree exists). This can be readily done in Alogtime by guessing the first and last leaves of S, and then checking that S is a maximal size subtree with logsize ℓ by universally branching over all other possible subtrees containing leaf x.

3 The Alogtime Algorithm for Tree Isomorphism

We let $\log n$ denote the logarithm base 2 of n rounded down to the nearest integer. The logsize, $logsize(T)$, of a tree T is defined to equal $\log|T|$.

Definition *Let S be a subtree of a tree T. Let $T = T_0, T_1, \ldots, T_k = S$ be the (unique) sequence of subtrees of T such that each T_{i+1} is an immediate subtree of T_i. The* size-signature *of S in T is defined to equal the sequence $\langle |T_0|, |T_1|, \ldots, |T_k| \rangle$.*

If S' is a subtree of a (possibly different) tree T', then S and S' are similar *provided that both:*

(1) *They have the same size-signature (as subtrees of their respective trees), and*
(2) *They are isomorphic, i.e., $S \equiv S'$.*

It is obvious that the notion of size-signature is invariant under isomorphism. That is to say, if trees T_1 and T_2 are isomorphic under the mapping f, and if S_1 is a subtree of T_1 and $S_2 = f(S_1)$ is its isomorphic copy in T_2, then S_1 and S_2 have the same size-signatures in T_1 and T_2, respectively.

By the usual parsing and counting techniques, there is an Alogtime procedure which, given a string representation of a tree T and given a subtree S of T, produces the size signature of S in T.

Definition *Let $T_1 \not\equiv T_2$ be trees. Let S be a subtree of T_1. We say that S* distinguishes *T_1 from T_2 provided that S is a proper subtree of T and:*

(1) *The logsize of S is strictly less than the logsize of the parent tree of S, and*
(2) *The number of subtrees of T_1 which are similar to S is not equal to the number of subtrees of T_2 which are similar to S.*

Lemma 1. *Suppose $|T_1| = |T_2|$ and $T_1 \not\equiv T_2$. Then there is a subtree S of T_1 which distinguishes T_1 from T_2.*

Proof. Since the trees are not isomorphic, there must be some immediate subtree S_1 of T_1 such that the number of immediate subtrees of T_1 which are isomorphic to S_1 is not equal to the number of immediate subtrees of T_2 which are isomorphic to S_1.

Now if the logsize of S_1 is less than the logsize of T_1, then S_1 is a subtree distinguishing T_1 from T_2. In particular, suppose that there is another immediate subtree S_1' of T_1 such that $|S_1'| \geq |S_1|$: then $|T_1| \geq 2 \cdot |S_1|$ which implies that the logsize of S_1 is strictly less than the logsize of T_1. Similarly, if there is more than one immediate subtree of T_2 with size greater than or equal to the size of S_1, then $|T_1| = |T_2| \geq 2 \cdot |S_1|$ and again the logsize of S_1 is less than the logsize of T_1.

Therefore, either S_1 distinguishes T_1 from T_2 or the following conditions hold: (1) S_1 is the unique maximum-sized proper subtree of T_1 and all other immediate subtrees of T_1 have size less than the size of S_1, and (2) T_2 has at most one subtree with size equal to the size of S_1. If T_2 has no subtree with size equal to the size of S_1, then any leaf of S_1 distinguishes T_1 from T_2, since no leaf of T_2 can have the same size signature. Finally, suppose T_2 has a subtree S_2 which has size equal to the size of S_1. By the choice of S_1, $S_1 \not\equiv S_2$. Thus, by the induction hypothesis, there is a subtree R of S_1 which distinguishes S_1 from S_2. Since S_1 and S_2 are the unique immediate subtrees of T_1 and T_2 of size $|S_1|$, the subtree R also distinguishes T_1 from T_2. □

We shall shortly present a precise definition of uniform logarithmic depth circuits for solving the tree isomorphism problem, which will suffice to show that the tree isomorphism problem is in Alogtime. However, as an intuitive aid, we first sketch the general idea of the alternating logarithmic time algorithm for tree isomorphism. As mentioned above, one can view an Alogtime algorithm as a game between players: the first player is asserting that the two trees are non-isomorphic, while the second player is asserting that the two trees *are* isomorphic. The input to the game consists of two string representations of two trees T_1 and T_2, and we denote this instance of the game $G[T_1, T_2]$. The game begins with the first player identifying a subtree S_1 that distinguishes T_1 from T_2. Then the two players play a log time game to count the number of subtrees of T_1 and of T_2 which are similar to S_1. If these numbers are equal, the second player wins and otherwise the first player wins. The inputs to the counting game consists of assertions of the form "S is similar to S_1" where S ranges over all subtrees of T_1 and T_2 (or at least over all subtrees which are the same size of S_1). Determining the truth of these assertions involves (a) comparing the size-signatures of S and S_1 and (b) checking whether $S \equiv S_1$. Part (a) is easily done in Alogtime, and part (b) involves a recursive call to the tree-isomorphism algorithm, $G[S, S_1]$.

Of course, care must be taken to show that the entire game $G[T_1, T_2]$, including the recursive calls to the game, uses only $O(\log n)$ rounds, where n is the maximum size of T_1 and T_2. For this, it is important to remember that the number of rounds in the play of game is defined to equal the number of bits of information exchanged by the two players (i.e., if one of the player sends ℓ bits, that counts as ℓ rounds). The fact that only $O(\log n)$ rounds are needed will depend crucially on two facts: firstly, that the logsize of S_1 is strictly less than the logsize of T_1 so the recursive calls are nested at most to the depth $\log n$, and secondly, that only $O(\mathit{logsize}(T_1) - \mathit{logsize}(S_1))$ many rounds of the game elapse between the beginning of the game $G[T_1, T_2]$ and the recursive call to $G[S, S_1]$.

The first step to establishing the last two facts is that we need a succinct way to specify the subtree S_1 of T_1. Our specification scheme will depend heavily on the fact that S_1 has logsize strictly less than the logsize of its parent tree. The first player can therefore uniquely specify S_1 as a subtree of T_1 by specifying (a) the logsize of S_1 and (b) the minimum number w such that $w \cdot 2^{\mathit{logsize}(S_1)} \leq \mathit{mindom}(S_1)$. In particular, this means that the numbers of the leaves of S_1 are a subinterval of

$$[w \cdot 2^{\mathit{logsize}(S_1)}, (w+3) \cdot 2^{\mathit{logsize}(S_1)} - 2]$$

Actually, since the game invokes itself recursively, we need more than just a succinct representation of S_1, we need to be able to succinctly represent a nested series of subtrees, each of successively smaller logsize. We will do this by using *trinary* strings over the alphabet $\{0, 1, 2\}$ to represent the substrings, according to the following construction.

Definition *We define now the trinary strings that define subtrees that arise during the play of the game above.*

Let $S_0 = T, S_1, S_2, \ldots, S_k$ be a sequence of subtrees of T, with each S_{i+1} a proper subtree of S_i (not necessarily an immediate subtree) and each S_i having logsize strictly less than its parent tree. Let $n = \mathit{logsize}(T)$ and let $i_m = \mathit{logsize}(S_m)$. The sequence $\langle S_0, \ldots, S_k \rangle$ has succinct representation defined as follows:

(1) *If $k = 0$, the empty string, ϵ, represents $\langle S_0 \rangle = \langle T \rangle$.*
(2) *If $k = 1$, the trinary string $2^{n-i_1} w_1$ where $w_1 \in \{0,1\}^{n-i_1+2}$ is the binary representation of the largest number (both the number and its binary representation are denoted w_1) such that $w_1 \cdot 2^{i_1} \leq \mathit{mindom}(S_1)$.*
(3) *For $k = 2$, the trinary string $2^{n-i_1} w_1 2^{i_1 - i_2} w_2$ where w_1 satisfies the conditions of (2) and where $w_2 \in \{0,1\}^{i_1 - i_2 + 2}$ is the binary representation of the largest number such that $w_1 2^{i_1} + w_2 2^{i_2} \leq \mathit{mindom}(S_2)$.*
(4) *More generally, for any value of $\ell \geq 0$, the trinary string representing the sequence of trees $\langle S_0, \ldots, S_\ell \rangle$ is the string $\sigma_1 \cdots \sigma_\ell$, where each σ_p is a*

string $2^{i_{p-1}-i_p}w_p$ *where* $w_p \in \{0,1\}^{i_{p-1}-i_p+2}$ *is the binary representation of the largest value, also denoted* w_p*, such that*

$$m_p \;=\; \sum_{j=1}^{p} w_j 2^{\mathit{logsize}(S_j)} \;\le\; \mathit{mindom}(S_p).$$

It is easy to verify, by induction on p*, that* $\mathit{dom}(S_p)$ *is contained in the interval* $[m_p, m_p + 3\cdot 2^{\mathit{logsize}(S_p)} - 2]$*. Therefore the value of* w_p *will satisfy*

$$w_p \cdot 2^{\mathit{logsize}(S_p)} \le 3 \cdot 2^{\mathit{logsize}(S_{p-1})},$$

from which it is immediate that $i_{p-1} - i_p + 2$ *bits are sufficient for the binary representation of* w_p*.*

When w *represents a sequence* $\langle S_0, \ldots, S_k\rangle$*, we sometimes abuse notation and say that* w represents *the subtree* S_k*.*

It is easy to check, but very important to note, that the number of symbols in the trinary string representing the sequence $\langle S_0, \ldots, S_k\rangle$ is $O(\log n - \mathit{logsize}(S_k))$. For the rest of this section, we let $\mu > 0$ be a constant such that the trinary strings w representing subtrees S_k of T_1 and T_2 have length $|w| \le \mu(\log n - \mathit{logsize}(S_k))$.

In addition, it is important that there is an Alogtime algorithm which, given a string representation of a tree T and given a trinary string α, determines whether the string properly represents a subtree of T and, if so, which subtree if represents. This is not however, so difficult: By counting the total number, ℓ, of 2's in α, one obtains the logsize, $n-\ell$, of the represented subformula S_k (if it exists). Then decompose the string α into substrings of 2's and binary substrings. For each binary string w, if it is preceded by total of j 2's, then view this as the integer $w\cdot 2^{n-j}$; adding up these integers gives the multiple of $2^{n-\ell}$ less than or equal to $\mathit{mindom}(S_k)$. From this, straightforward Alogtime parsing techniques give the formula S_k.

Theorem 1. *The tree isomorphism problem is in alternating logarithmic time.*

Proof. It will suffice to construct logarithmic depth, Alogtime-uniform, bounded fan-in circuits for the tree isomorphism problem. We present a construction of the circuit; but will omit the straightforward proof that the construction can be made Alogtime uniform. The circuit is essentially a direct implementation of the game described informally above.

Fix a tree size n. We may assume that the inputs T_1 and T_2 both have the same size $|T_1| = |T_2| = n$, since otherwise they are not isomorphic. We describe the construction of a circuit $\mathcal{C}_n$ which recognizes whether the trees are isomorphic. We will use names such as "$C_{1,2}[u,v]$" for gates in $\mathcal{C}_n$ where u and v are trinary strings of length $\le \mu\log n$ which potentially name subtrees of T_1 or T_2. The circuit $\mathcal{C}_n$ will contain the following gates:

$C^+_{1,a}[u,v]$**:** Here $a \in \{1,2\}$ and u and v are equal length strings which are intended to represent subtrees of T_1 and T_a. This gate $C_{1,a}[u,v]$ is a two input AND gate. The first input to the AND gate is a logarithmic depth circuit which outputs *True* provided u represents a subtree R_1 of T_1 and v represents a subtree R_2 of T_a and provided that R_1 and R_2 have identical size signatures. The second input to the AND gate is the circuit $C_{1,a}[u,v]$.

$C_{1,a}[u,v]$**:** Here u, v and a are as above. This gate is relevant only if u and v represent subtrees R_1 and R_2 of T_1 and T_a (respectively) with R_1 and R_2 having the same size signature. The gate $C_{1,a}[u,v]$ is intended to compute whether R_1 is isomorphic to R_2. The circuit consists of the conjunction of gates $D^+_{1,a}[uw,v]$ for all trinary strings w such that w is in $2^i\{0,1\}^{i+2}$ for some $i \geq 1$ and such that $|uw| \leq \mu \log n$. The conjunction is implemented as a tree of two-input AND gates with $D^+_{1,a}[uw,v]$ being an input at depth $|w|+2$.

$D^+_{1,a}[uw,v]$**:** Here u, v, w and a are as above. The gate $D^+_{1,a}[uw,v]$ outputs *True* provided either (i) it is not the case that uw represents a subtree of T_1 which has logsize strictly less than the logsize of its parent tree, OR (ii) the gate $D_{1,a}[uw,v]$ computes *True*.

$D_{1,a}[uw,v]$**:** Here u, v, w and a are as above. The gate $D_{1,a}[uw,v]$ outputs *True* provided that the number of trinary strings x in $2^i\{0,1\}^{i+2}$ with $|x| = |w|$ for which $C^+_{1,a}[uw,vx] = \mathit{True}$ is equal to the number for which $C^+_{1,1}[uw,ux] = \mathit{True}$. There are $3^{|w|}$ many trinary strings $x \in \{0,1,2\}^{|w|}$; therefore, using the Alogtime circuits for counting, the gate $D_{1,a}[uw,v]$ can be computed by a circuit of depth $O(|w|)$ which has as inputs the gates $C^+_{1,a}[uw,vx]$ and $C^+_{1,1}[uw,ux]$.

The description of the circuit $\mathcal{C}_n$ is now complete, since the output of $\mathcal{C}_n$ is just the gate $C^+_{1,2}[\epsilon,\epsilon]$.

There are a couple things which must be verified to see that $\mathcal{C}_n$ is a correct, logarithmic depth circuit. The fact that $\mathcal{C}_n$ has logarithmic depth is immediate from the construction. Indeed any gate $X[u,v]$ with X one of $C^+_{1,a}$, $C_{1,a}$, $D^+_{1,a}$ or $D_{1,a}$ will be at depth $\Omega(|u|+|v|) = O(\log n)$ in the circuit. It is important for the depth analysis, that the conjunction for the subcircuit $C_{1,a}[u,v]$ be correctly balanced so that the inputs $D^+_{1,a}[uw,v]$ occur at depth $|w|+2$ in the conjunction. This balancing of AND's is illustrated in Figure 1, where we show the top few levels of the left half of a conjunction of inputs D_w, showing where the three inputs D_0, D_{00} and D_{01} occur.

To verify that the circuit correctly computes tree isomorphism, one proceeds by induction on $(\mu \log n) - |u|$ that $C^+_{1,a}[u,v] = \mathit{True}$ if and only if u and v specify isomorphic subtrees. Suppose u has maximum length for which there is a v, $|u| = |v|$, such that $C^+_{1,a}[u,v]$ incorrectly computes the isomorphism predicate for the two subtrees R_1 and R_2 represented by u and v. It is immediate from the definitions that if R_1 and R_2 are isomorphic, then $C^+_{1,a}[u,v]$ must be true. So suppose R_1 and R_2 are not isomorphic. By Lemma 1, there is a subtree S of R_1 which distinguishes R_1 from R_2. Let uw be the string which represents S: $D^+_{1,a}[uw,v]$ will be false and therefore $C_{1,a}[u,v]$ is false.

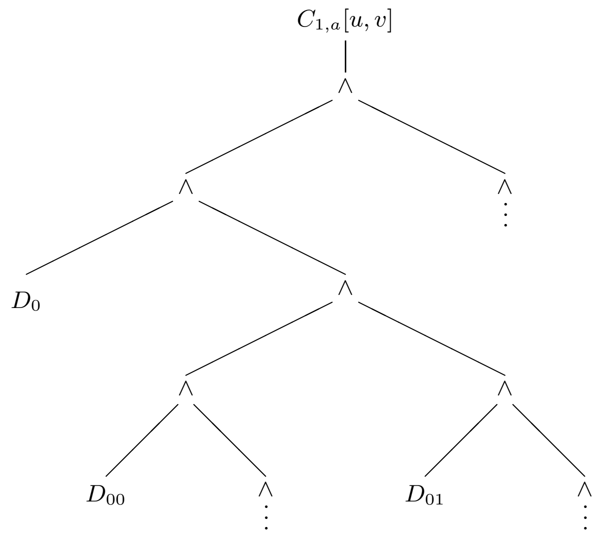

Fig. 1. This figure illustrates the balancing of the conjunction computing $C_{1,a}[u,v]$. The D_w's are intended to be replaced by inputs $D^+_{1,a}[uw,v]$, although these are relevant only for $|w| \geq 2$; and the above circuit is intended only to illustrate the idea of the construction of the conjunction.

□

4 An Alogtime Algorithm for Tree Comparison

In the next two sections, we give Alogtime algorithms for the tree comparison and tree canonization problems. We'll start with the tree comparison problem, and then see that a solution to the tree comparison problem can be used to give a solution to the tree canonization problem.

First, we need some technical definitions about trees.

Definition *Let T be a tree. If S_1 and S_2 are subtrees of T, then S_2 is the* maxsize subtree of S_1 *if and only if S_2 is an immediate subtree of S_1 and every other subtree of S_1 has size strictly smaller than the size of S_2.*

A subtree S of T is called a unimax *subtree of T if and only if there is a sequence of subtrees $S_0 = T, S_1, \ldots, S_k = S$ such that each S_{i+1} is the maxsize subtree of S_i.*

Note there is at most one unimax subtree S_k at a given depth k in a tree T. Let T_1 and T_2 be two trees which we are trying to compare and suppose $|T_1| = |T_2|$.

Definition *Suppose S is a unimax subtree of T_1 and let R_1 be the parent tree of S. We say S* unimax $\succ$-distinguishes *T_1 and T_2 if and only if there is a* *unimax subtree R_2 of T_2 with the same size signature as R_1 such that every immediate subtree of R_2 has size strictly less than the size of S.*

Lemma 2. *If S unimax $\succ$-distinguishes T_1 and T_2, then $T_1 \succ T_2$.*

Proof. This is immediate from the definition of $\succ$, the definition of unimax subtree, and induction on the depth of S. $\square$

Lemma 3. *The problem of determining whether there is a subtree S which unimax $\succ$-distinguishes T_1 and T_2 is in alternating logarithmic time.*

Proof. This is very simple. The Alogtime algorithm first guesses the identity of the subtree S and then verifies the syntactic conditions of unimax $\succ$-distinguishing using the usual Alogtime counting and parsing techniques. $\square$

When we can't unimax $\succ$-distinguish two trees, then we must instead "weakly $\succ$-distinguish" them.

Definition *If S and R are trees, then the* multiplicity *of S in R is equal to the number of immediate subtrees of R which are isomorphic to S.*

Since tree isomorphism is in Alogtime, and by the usual Alogtime parsing techniques, there are Alogtime algorithms for computing and comparing the multiplicities of subtrees.

Definition *We say S* weakly $\succ$-distinguishes *T_1 and T_2 if and only if the following conditions hold:*

(1) *The parent tree R_1 of S is a unimax subtree of T_1, but S is not a unimax subtree.*
(2) *There is a (necessarily unique) unimax subtree R_2 of T_2 with the same size signature as R_1.*
(3) *For each subtree S_2 of R_2 such that $|S_2| > |S|$, the multiplicity of S_2 in R_1 is greater than or equal to its multiplicity in R_2.*[2]
(4) *For each subtree S_2 of R_2 such that $|S_2| = |S|$ and $S_2 \succ S$, the multiplicity of S_2 in R_1 is greater than or equal to its multiplicity in R_2.*
(5) *The multiplicity of S in R_1 is strictly greater than the multiplicity of S in R_2.*

Lemma 4. *If S weakly $\succ$-distinguishes T_1 and T_2, then $T_1 \succ T_2$.*

Proof. This is immediate from the definition of $\succ$.

Lemma 5. *If $|T_1| = |T_2|$ and $T_1 \succ T_2$, then there is a subtree S of T_1 which either unimax or weakly $\succ$-distinguishes T_1 from T_2.*

[2] We could use "equal to" in place of "greater than or equal to" in (3) and (4), but our choice will save some implementation details in the circuit constructed below.

Proof. The proof is by induction on $|T_1|$. Order the immediate subtrees of T_1 as $R_1 \succcurlyeq R_2 \succcurlyeq \cdots \succcurlyeq R_m$ and the immediate subtrees of T_2 as $S_1 \succcurlyeq S_2 \succcurlyeq \cdots \succcurlyeq S_n$. Find the unique i such that $R_j \equiv S_j$ for all $1 \le j < i$ and such that either $i = n+1 \le m$ or $R_i \succ S_i$. If $i > 1$, then $S = R_i$ is not a unimax subtree of T_1 and obviously weakly $\succ$-distinguishes T_1 and T_2. If $i = 1$ and $|R_1| > |S_1|$, then $S = R_1$ either unimax or weakly $\succ$-distinguishes T_1 and T_2 depending on whether $logsize(R_1) = logsize(T_1)$. Finally, if $i = 1$ and $|R_1| = |S_1|$, then the induction hypothesis gives a subtree S of R_1 which unimax or weakly $\succ$-distinguishes R_1 and S_1. This S also unimax or weakly $\succ$-distinguishes T_1 and T_2. □

We now give an informal definition of the Alogtime algorithm for tree comparison by describing a two player game that runs for logarithmically many rounds: the first player is asserting that $T_1 \succ T_2$, whereas the second player is asserting that $T_1 \preccurlyeq T_2$. The game, denoted $G'[T_1, T_2]$ will invoke itself recursively on subtrees of T_1 and T_2. Since we already have an Alogtime algorithm for tree isomorphism, we are allowed to invoke that algorithm during the play of the game for tree comparison. The game $G'[T_1, T_2]$ starts with the first player specifying a subtree S of T_1 and asserting either that S unimax $\succ$-distinguishes T_1 and T_2 or weakly $\succ$-distinguishes T_1 and T_2. In the first case, the two players verify whether S actually does unimax $\succ$-distinguish T_1 and T_2, and then the first player wins iff it does so. In the second case, the second player will choose one of the conditions (1)-(5) of the definition of "S weakly $\succ$-distinguishes T_1 and T_2" that must be verified. Conditions (1)-(3) and (5) are easily checked with an Alogtime algorithm — in this case, the first player wins iff the one of these conditions picked by the second player is determined to be valid. If the second player picks condition (4), then the second player also specifies a subtree S_2 of R_2 as a candidate subtree for showing that condition (4) fails. Then the first player chooses either to check either that S_2 does not satisfy $|S_2| = |S|$ or has multiplicity in R_1 greater than or equal to its multiplicity in R_2 or that $S \succ S_2$ (anyone of these checks would be sufficient to show that S_2 does not refute condition (4)). If any of these checks succeed the first player wins. The check that $S \succ S_2$ involves a recursive invocation of the game $G'[S, S_2]$.

Lemma 5 shows the correctness of the above game for tree comparison. It remains to see that the game only takes $O(\log |T_1|)$ rounds. A careful analysis of the number of rounds is best done by examining the circuit for tree comparison which is given below in the next proof; however, there are certainly reasons to hope that the above game can be formalized so as to use only logarithmically many bits of communication between the players. First, we have already developed succinct representations for subformulas. Second, whenever S weakly $\succ$-distinguishes two trees, it must be the case that *logsize*(S) is strictly less than the logsize of the parent tree of S. This latter fact is because the parent tree of S is unimax, whereas S is not, and therefore the parent tree of S is at least twice the size of S.

Theorem 2. *The tree comparison problem is in alternating logarithmic time.*

Proof. It will suffice to construct logarithmic depth, Alogtime uniform, bounded fanin circuits for the tree comparison problem. We present a construction of the circuit, but omit the straightforward proof that the construction can be made Alogtime uniform. The circuit is essentially a direct implementation of the game G' above.

Fix a tree size n. We may assume that $|T_1| = |T_2|$ since otherwise the tree comparison problem is trivial. We describe the construction of a circuit $\mathcal{C}'_n$ which performs tree comparison. Some of the gates in the circuit will be given names such as $E[u,v]$ where u and v are trinary strings of length $\leq \mu \log n$ which potentially name subtrees of T_1 and T_2. $\mathcal{C}'_n$ contains gates:

$E[u,v]$**:** Here u and v are equal length strings which represent subtrees U_1 and U_2 of T_1 and T_2, respectively. This gate is relevant to the circuit computation only when u and v represent subtrees U_1 and U_2 which have the same size signature.
The gate $E[u,v]$ is intended to compute whether $U_1 \succ U_2$. The topmost gate of $E[u,v]$ is a disjunction. The first input to the disjunction is a logarithmic depth circuit that checks whether there is a subtree of U_1 which unimax $\succ$-distinguishes U_1 and U_2. This circuit exists by Lemma 3. The second input to the disjunction is a disjunction (actually a tree of two-input disjunctions) of the gates $F^+[uw,v]$ for all trinary strings w such that such that $w \in 2^i\{0,1\}^{i+2}$ for some $i \geq 1$ and such that $|uw| \leq \mu \log n$. The disjunction is implemented as a tree of two-input OR gates with $F[uw,v]$ being an input at depth $|w|+2$.

$F^+[uw,v]$**:** Here u, v, and w are as above. The gate $F^+[uw,v]$ is a conjunction of (i) the circuit $F[uw,v]$ and of (ii) a logarithmic depth circuit which outputs *True* iff the following conditions hold:

> the string uw represents a non-unimax subtree S of T_1, which has a unimax parent tree R_1; there is a unimax subtree R_2 of T_2 with the same size signature as R_1; the multiplicity of S in R_1 is greater than the multiplicity of S in R_2; and the maxsize subformula of R_2 (if it exists) has multiplicity in R_1 greater than or equal to its multiplicity in R_2.

$F[uw,v]$**:** Here u, v and w are as above. The gate $F[uw,v]$ is the conjunction of the gates $H[uw,vw^*]$ for all $w^* \in 2^i\{0,1\}^{i+2}$ with $i \geq 1$ and $|w^*| \leq |w|$. The gate $H[uw,vw^*]$ is at depth $|w^*|+2$ in the conjunction (it is actually a tree of two-input conjunctions).

$H[uw,vw^*]$: Here u, v, w and w^* are as above. This gate is the disjunction of (i) the circuit $E[uw,vw^*]$ provided $|w| = |w^*|$, or otherwise this input can be set to *False*. and (ii) a logarithmic depth circuit with outputs *True* iff at least one of the following conditions hold:

> The string vw^* is not a valid representation of subtree S_2 in T_2; or $S_2 \equiv S$; or $|S_2| < |S|$; or the multiplicity of S_2 in R_1 is greater than or equal to its multiplicity in R_2.

The description of the circuit $\mathcal{C}'_n$ is now complete, since the output of $\mathcal{C}_n$ is just the gate $E[\epsilon,\epsilon]$.

There are a couple things which must be verified to see that $\mathcal{C}'_n$ is a correct, logarithmic depth circuit. The fact that $\mathcal{C}'_n$ has logarithmic depth is immediate from the construction. Indeed any gate $X[u,v]$ with X one of E, F^+, F, or H will be at depth $\Omega(|u|+|v|) = O(\log n)$. For this, it is important that the disjunctions for $E[u,v]$ and the conjunctions for $F[uw,v]$ be correctly balanced with inputs at the correct depths.

The correctness of the circuit $\mathcal{C}'_n$ is proved by induction on the lengths of the strings u, v. The gate $E[u,v]$ is intended to compute *True* iff u and v represent subtrees U_1 and U_2 of T_1 and T_2 (resp.) with $U_1 \succ U_2$. The gate $F^+[uv,w]$ is intended to output *True* iff uv represents a subtree S of U_1 which weakly $\succ$-distinguishes U_1 and U_2. The gate $H[uw,vw^*]$ outputs *True* iff it is not the case that vw^* represents a subtree S_2 which violates one of the conditions (3) or (4) of the definition of "S weakly $\succ$-distinguishes U_1 and U_2". In view of Lemma 5, it is straightforward to verify that the circuit correctly performs tree comparison. One point that deserves further justification is the manner in which w^* is chosen in $F^+[uw,v]$: the string w^* is chosen so that $H[uw,vw^*]$ can decide whether vw^* represents a subtree S_2 which violates condition (4) of the definition of "S weakly $\succ$-distinguishes U_1 and U_2". The reason we can restrict w^* to have $|w^*| \le |w|$ is that condition (4) only considers formulas S_2 which have size greater than or equal to S; this means that $\mathit{logsize}(S_2) \ge \mathit{logsize}(S)$ and therefore, by the definition of trinary representation, $|w^*| \le |w|$. In addition, if S_2 is not a maxsize subformula of R_2, then S_2 has logsize strictly less than the logsize of R_2 and therefore S_2 does have a trinary string representation. The verification of condition (3) for S_2 the maxsize subtree of R_2 was already taken care by the subcircuit (ii) described under the heading $F^+[uw,v]$. □

5 An Alogtime Algorithm for Tree Canonization

Given the Alogtime algorithms for tree isomorphism and tree contraction, it will be easy to prove the existence of an Alogtime algorithm for tree canonization. Recall that the definition of canon_T, the canonization of T, was given in section 2.

Theorem 3. *The tree canonization problem is in alternating logarithmic time.*

Proof. It will suffice to show that there is an Alogtime algorithm which determines the orders of the leaves of an input tree in the canonization of T. As usual, the input tree T is specified by its string representation, i.e., a string of parentheses. Consider two leaves, x and y, with leaf numbers i and j, with $i < j$. Find the node z in the tree which is the least common ancestor of x and y. Let S_x be the immediate subtree of z containing x and S_y be the immediate subtree of z containing y. If $S_x \not\succ S_y$, then let x precede y in a reordering of the tree; otherwise let y precede x. Applying this construction to all leaf nodes of T, we get a consistent reordering of the leaves of T which induces a canonization of T.

Operations such as finding least common ancestors can be done in Alogtime, so the Alogtime algorithm for tree comparison gives an Alogtime algorithm for tree canonization. □

6 Conclusions

Our main results show that tree isomorphism, tree comparison and tree canonization are in alternating logarithmic time; improving on the logarithmic space algorithms of [8].

There are number of other problems known to be in Alogtime, including the Boolean Formula Value Problem [3, 5, 4], and the word problem for S_5 [2]. These problems are also known to be complete for Alogtime under deterministic log time reductions. It is still open whether the tree isomorphism, comparison and canonization problems are also complete for Alogtime.

A problem with a similar name to the tree isomorphism problem is the "subtree isomorphism problem". This is the problem of, given two trees T_1 and T_2, determining whether there is injection from T_1 to T_2 which preserves the edge relation. In other words, whether T_1 is isomorphic to a subgraph of T_2 induced by some subset of the nodes of T_2. A polynomial-time deterministic algorithm for the subtree isomorphism problem was first found by [11]; and [6, 9] gave random NC (RNC) algorithms for subtree isomorphism. [6] also proved that bipartite matching is reducible to subtree isomorphism; this means that subtree isomorphism cannot be in logspace or Alogtime unless bipartite matching is. Thus it appears that subtree isomorphism is substantially more difficult than tree isomorphism.

We conclude by mentioning one of our favorite open problems about alternating logtime: is the 2-sided Dyck language recognizable in alternating logarithmic time. Equivalently, is the word problem for free groups with two generators decidable in alternating logarithmic time? This language was shown in [10] to be in logspace.

Acknowledgement. We thank Steven Lindell for helpful discussions during the formulation of the algorithms in this paper.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. D. A. M. Barrington, *Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1*, J. Comput. System Sci., 38 (1989), pp. 150–164.
3. S. R. Buss, *The Boolean formula value problem is in ALOGTIME*, in Proceedings of the 19-th Annual ACM Symposium on Theory of Computing, May 1987, pp. 123–131.

4. ———, *Algorithms for Boolean formula evaluation and for tree contraction*, in Arithmetic, Proof Theory and Computational Complexity, P. Clote and J. Krajíček, eds., Oxford University Press, 1993, pp. 96–115.
5. S. R. Buss, S. A. Cook, A. Gupta, and V. Ramachandran, *An optimal parallel algorithm for formula evaluation*, SIAM J. Comput., 21 (1992), pp. 755–780.
6. P. B. Gibbons, R. M. Karp, G. L. Miller, and D. Soroker, *Subtree isomorphism in in random NC*, Discrete Applied Mathematics, 29 (1990), pp. 35–62.
7. N. Immerman, *Languages that capture complexity classes*, SIAM Journal on Computing, 16 (1987), pp. 760–778.
8. S. Lindell, *A logspace algorithm for tree canonization*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 400–404.
9. A. Lingas and M. Karpinski, *Subtree isomorphism is NC reducible to bipartite perfect matching*, Information Processing Letters, 30 (1989), pp. 27–32.
10. R. J. Lipton and Y. Zalcstein, *Word problems solvable in logspace*, J. Assoc. Comput. Mach., 24 (1977), pp. 522–526.
11. D. W. Matula, *Subtree isomorphism in* $O(n^{5/2})$, Annals of Discrete Mathematics, 2 (1978), pp. 91–106.
12. G. L. Miller and J. H. Reif, *Parallel tree contraction part 2: Further applications*, SIAM Journal on Computing, 20 (1991), pp. 1128–1147.
13. W. L. Ruzzo, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.