# Algorithms for Boolean Formula Evaluation and for Tree Contraction

Samuel R. Buss*

Department of Mathematics

University of California, San Diego

October 15, 1991

**Abstract**

This paper presents a new, simpler ALOGTIME algorithm for the Boolean sentence value problem (BSVP). Unlike prior work, this algorithm avoids the use of postfix-longer-operand-first formulas. This paper also shows that tree-contraction can be made ALOGTIME uniform.

The Boolean sentence problem restricted to balanced sentences with only the connectives $\wedge$ and $\vee$ is in online $O(\log(\log n))$ space. Hence every ALOGTIME predicate is deterministic log-time reducible to $\log(\log n)$ space. This balanced and/or BSVP has logarithmic width, linear length, branching programs.

*Supported in part by NSF grants DMS-8902480 and INT-8914569. Part of this work performed while visiting the Czechoslovakian Academy of Sciences. Email address: sbuss@ucsd.edu.

# 1 Introduction

A Boolean sentence is an expression formed from propositional connectives $\wedge$, $\vee$, $\neg$, etc., from constants 1 and 0 (for 'True' and 'False', respectively), and from parentheses. The *Boolean sentence value problem* (BSVP) is the decision problem of, given a Boolean sentence, to determine if the value of the sentence is 1 (i.e., True). The Boolean sentence problem has been shown to be in alternating logarithmic time (ALOGTIME) by Buss [4] and Buss-Cook-Gupta-Ramachandran [5]. Hence the Boolean *formula* value problem is also in ALOGTIME (a formula may have free variables—the problem is to find the truth value of a formula given truth values for its variables).

The prior algorithms of [4] and [5] were based on a construction which necessitated converting an infix Boolean formula to Postfix-Longer-Operand-First (PLOF) form and then using complicated methods for choosing breakpoints in a semi-oblivious fashion. One of the main contributions of this paper is to give a new and simpler ALOGTIME algorithm for the BSVP. In our new algorithm, the sentence remains in infix notation and the method for selecting breakpoints is conceptually simpler.

Just as in [5], the method for solving the BSVP can be adapted to give logarithmic depth, polynomial size, arithmetic circuits for arithmetic formula evaluation. More generally, in this paper we analyse a tree-contraction algorithm of Abrahamson-Dadoun-Kirkpatrick-Przytycka [1] and show that it is ALOGTIME uniform.

Thirdly, we consider a special case of the Boolean sentence problem where the sentence is restricted to be balanced (or at least have logarithmic depth) and to have only the connectives $\wedge$ and $\vee$. We show that this 'Balanced, and/or BSVP' has an $O(\log\log n)$ space algorithm. Indeed there is an $O(\log\log n)$ space algorithm for this problem which scans the input from left to right without ever backing up and which has as memory only a single counter which may be incremented, decremented and tested for being equal to zero. To compare this with prior work, recall that Barrington showed that the BSVP has constant-width, quadratic length branching programs. Our result shows that the 'balanced and/or BSVP' has $\log\log n$ width, linear length branching programs. Unlike Barrington's construction, the branching programs we construct correspond to online computations in that the input symbols controlling the branching program are taken in their input order.

From the above we obtain the surprising result that every problem in ALOGTIME is deterministic log time reducible to $\log\log n$ space (by a Turing machine with random access).

## 2 The ALOGTIME algorithm for BSVP

In this section we present the ALOGTIME algorithm for the Boolean sentence value problem. We presume basic familiarity with the notion of ALOGTIME; however, we briefly review the definitions and relevant properties. For more information see the introductory sections of [4, 5] and the references cited therein.

An ALOGTIME Turing machine is an alternating, multitape Turing machine which runs in logarithmic time; since the runtime is sublinear, the input is accessed via an index tape so that if an integer $i$ is written in binary notation on the index tape then the $i$-th symbol of the input string is accessible to the finite state control of the Turing machine. Without loss of generality the Turing machine may be constrained so that it only reads a single input symbol (and then immediately halts) on any computation path This is because the Turing machine may be designed to guess input symbols as needed and then branch universally to either verify the guess or to continue the computation.

It is well-known that ALOGTIME corresponds to a uniform version of $NC^1$; more precisely, the set of predicates recognizable in ALOGTIME is the same as the set of predicates which have $U_{E^*}$-uniform, log depth, bounded fan-in circuits.

A function $f$ is said to be in ALOGTIME if its bit graph is in ALOGTIME; i.e., if

$$\{\langle x, i, c\rangle : i\text{-th symbol of } f(x) \text{ is } c\}$$

is in ALOGTIME. Some prominent examples of functions known to be in ALOGTIME are counting, multiplication and vector addition. With the aid of counting, it can be seen that any usual parsing operation on parenthesized infix formulas can in performed in ALOGTIME — for example, determining if a formula is properly parenthesis-balanced or identifying matching parentheses can be performed in ALOGTIME by counting parentheses.

It is convenient to picture Boolean sentences as trees with interior nodes labelled with propositional connectives and leaves labelled with 0/1 values. Hence a *node* in a Boolean sentence is (an occurrence of) a non-parenthesis symbol in the sentence and a *leaf* is (an occurrence of) a constant symbol in the sentence. Concepts such as 'depth', 'child', 'father', 'ancestor', 'descendent', etc. that apply to trees thus also apply to Boolean sentences. Furthermore, by usual methods of parenthesis counting, all these concepts are ALOGTIME definable.

The leaves of a Boolean sentence are numbered from left to right beginning with 1. The *rank* of a leaf is the largest integer $d$ such that $2^d$ divides the number of the leaf. Both the number and the rank are readily seen to be computable in ALOGTIME.

If $U$ and $V$ are nodes, we write $U \rhd V$ to mean that $U$ is an ancestor of $V$. Similarly, $U \unrhd V$ means that $U = V$ or $U \rhd V$. The least common ancestor of $U$ and $V$ is denoted $lca(U, V)$ and is the lowest node $W$ such that $W \unrhd U$ and $W \unrhd V$. (Trees, by convention, have root at the top so 'lowest' means of greatest depth.) The least common ancestor function is clearly computable in ALOGTIME.

We shall henceforth assume that all Boolean sentences do not contain negation signs and hence correspond to binary trees. This assumption can be made without loss of generality since there is an ALOGTIME algorithm which converts any Boolean formula containing negation signs to an equivalent one without negation signs. We also assume w.l.o.g. that there are exactly $2^{d+1} - 1$ leaves in a Boolean sentence: this can always be ensured by adding extra $(\cdots \wedge 1)$'s to the sentence — this will at most double the length of the formula. (The only reason for the latter assumption is to simplify the exposition of the algorithm below.)

The rest of this section is devoted to the proof of:

**Theorem 1** *The BSVP is in ALOGTIME.*

The ALOGTIME algorithm for the BSVP will be presented as a pebbling game played by two players called the Pebbler and the Challenger. The pebbling game is described next in a top-down fashion.

In the pebbling game, the Pebbler places "pebbles" on nodes of the Boolean sentence: each pebble is labelled with a truth value 0 or 1 and represents an assertion by the Pebbler that the subformula rooted at the pebbled node has that truth value. After the Pebbler has placed some pebbles, the Challenger must respond by challenging one of the pebbled positions: this represents an assertion by the Challenger that the pebble value is incorrect, and furthermore, by challenging a pebble position $U$, the Challenger asserts that every pebble position $V \lhd U$ is correctly labeled. As an extra condition, we require that the Challenger may not challenge any node $W$ such that either $W \rhd U$ where $U$ has already been challenged or $W \unlhd V \lhd U$ for some previously challenged $U$ and some $V$ that was already pebbled when $U$ was challenged.

The pebbling game is devised so that the Pebbler has a winning strategy if the Boolean sentence has value 1 and otherwise the Challenger has a winning strategy. A lot of the rules of the game (such as the 'extra condition' above) are designed so that a play of the game can be run and evaluated in alternating log time. The reader should not be misled by the fact that rules seem somewhat arbitrary and strict — the point is that the strictness of the rules helps with the alternating log time calculation while preserving the property that the correct player has a winning strategy.

The pebbling game is organized into rounds: during each round, the Pebbler pebbles a set of nodes (the set of nodes is specified below) and then the Challenger either challenges a newly pebbled node or rechallenges the previously challenged node. A player loses the game by making a clearly false assertion; e.g., asserting an incorrect value for a leaf, asserting incompatible input and output values for a Boolean gate, etc. A player may also lose the game by violating the rules detailed below.

The pebbling game will yield an ALOGTIME algorithm for the BSVP since the following conditions will hold:

(1) The Pebbler has a winning strategy iff the Boolean sentence has value $1$.

(2) The game lasts $\leq \lceil \log n \rceil$ rounds, where $n$ is the number of leaves in the formula.

(3) Complete information about the Pebbler's and Challenger's moves can be given with a constant number of bits per round (12 bits per round in the implementation below).

(4) There is an ALOGTIME algorithm which, given the Boolean sentence

and the $O(\log n)$ bits specifying the moves of the players, determines the winner of the game.

The conversion of the pebbling game into an alternating log time Turing machine $M$ is quite straightforward. On input a Boolean sentence $\varphi$, $M$ computes $O(\log n)$ bits describing a play of the game using existential guesses for the Pebbler's moves and universal choices for the Challenger's moves; $M$ then accepts iff the Pebbler won this play of the game. By (2), (3) and (4), $M$ runs for $O(\log n)$ time, and by (1) and the definition of alternation, $M$ accepts $\varphi$ iff $\varphi$ has value $1$.

Condition (3) that only $O(1)$ many bits of information may be given per round is rather stringent. It is also crucial for obtaining an ALOGTIME algorithm. Since a constant number of bits is not sufficient for specifying an arbitrary pebble point (that would require $O(\log n)$ bits), it is necessary that the rounds of the game be at least partly "oblivious" to the previous rounds. We say that a game is *open* at round $i$ if the winner of the game is not determined by the players' moves before round $i$. The game, as implemented by the Turing machine $M$, lasts $O(\log n)$ rounds even though the game may not be open for all the rounds. The game will be defined so that the loser of the game is the first player is make an 'obvious' mistake (as defined below) and thus players's moves after the first such mistake are irrelevant to the outcome of the game.

We now define in complete detail the rules of the Pebbling game. In round $i \geq 1$ there will be leaves $L_i$, $C_i$, $R_i$ and nodes $A_i$ and $B_i$ (see Figure 1). As a useful mnemonic think of $A$, $B$, $L$, $C$ and $R$ as standing for 'Above', 'Below', 'Left', 'Center' and 'Right'. The following four conditions will be satisfied for as long as the game is open.

($\alpha$) $A_i \rhd B_i \unrhd C_i$.

($\beta$) $A_i$ is the lowest (and last) challenged note. $B_i$ is the highest pebbled position satisfying ($\alpha$). (There will always be such a node $B_i$ if the game is still open at round $i$.) Informally, the players have agreed at $B_i$ but disagree at $A_i$.*

*For readers familiar with prior expositions of pebbling games: the rules of the game force the "subformula of disagreement" to be 1-scarred; i.e., if $D \lhd A_i$ is a pebbled node, then $D \unlhd B_i$.

($\gamma$) $L_i$ and $R_i$ are leaves of rank $d-i$ and $C_i$ is of rank $> d-i$.

($\delta$) Every leaf $D$ which is a descendent of $A_i$ but is not a descendent of $B_i$ has number in the range

$$(L_i - \delta_i, L_i + \delta_i) \cup (R_i - \delta_i, R_i + \delta_i)$$

where $\delta_i = 2^{d-i-1}$. Here we are identifying leaves with their numbers in left-to-right order; the rounded parentheses indicate open intervals. Thus $L_i - \delta_i$ and $L_i + \delta_i$ are the leaves closest to $L_i$ of rank exactly $d-i-1$, and similarly for $R_i - \delta_i$ and $R_i + \delta_i$.

The pebbling game proceeds as follows:

*Round 0:* The Pebbler must pebble the root node with value 1 and also also pebble the unique leaf of rank $d$ (recall that there are exactly $2^{d+1}-1$ leaves). The Challenger must challenge the pebble at the root. In preparation for round 1, set $A_1$ equal to the root, set $B_1 = C_1$ equal to the leaf numbered $2^d$, and set $L_1$ and $R_1$ equal to the leaves numbered $2^{d-1}$ and $2^d + 2^{d-1}$.

*Round $i$:* Let $U_i$ be $lca(L_i, C_i)$ and $V_i$ be $lca(C_i, R_i)$. Note that $U_i$ and $V_i$ are distinct. Let $U_i^1$ and $U_i^2$ be the left and right children of $U_i$, respectively, and let $V_i^1$ and $V_i^2$ be the left and right children of $V_i$, respectively (see Figure 2). The Pebbler must do the following two things:

(a) State whether $A_i \rhd U_i \rhd V_i$ or $A_i \rhd V_i \rhd U_i$ or $U_i \unrhd A_i \rhd V_i$ or $V_i \unrhd A_i \rhd U_i$ or $U_i, V_i \unrhd A_i$. (Since $A_i$, $U_i$ and $V_i$ are ancestors of $C_i$, exactly one will hold; so three bits of information specifies which one.)

(b) Give six truth values which are placed on pebbles on the nodes $U_i$, $V_i$, $U_i^1$, $U_i^2$, $V_i^1$, $V_i^2$ (six bits of information).†

†A slightly more complicated definition of the game could save two bits of information by not having the Pebbler pebble $U_i$ and $V_i$. This would still preserve the essential properties of the game since the truth values of $U_i$ and $V_i$ are determined by the truth values of their inputs.

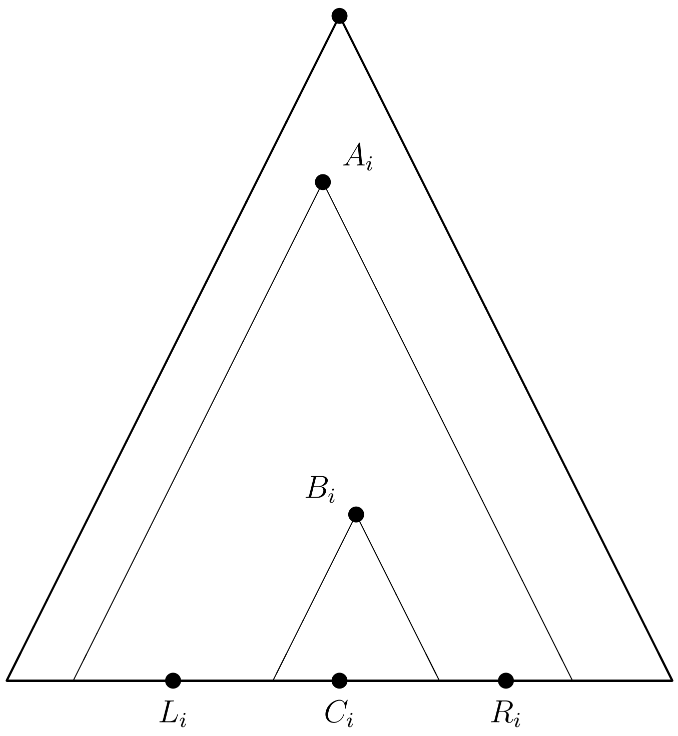



**Figure 1**

The triangles delineate subtrees. In general, it is not necessary for $L_i$ and $R_i$ to be descendents of $A_i$. On the other hand, it is also permissable for $L_i$ and $R_i$ to be descendents of $B_i$.

---

Then the Challenger must challenge one of the pebbled nodes $A_i$, $U_i$, $V_i$, $U_i^1$, $U_i^2$, $V_i^1$, $V_i^2$ (there are seven possibities so three bits of information suffice). The Challenger must choose to challenge a node $W$ such that $A_i \trianglerighteq W$ and such that $B_i \ntrianglerighteq W$.

For round $i+1$, the values of $L_{i+1}$, $C_{i+1}$, $R_{i+1}$ are set according to Tables 1 and 2. And $A_{i+1}$ is the node challenged by the Challenger in round $i$ and $B_{i+1}$ is the highest pebbled node below $A_{i+1}$ (there will always be a unique choice for $B_{i+1}$ if the game is still open).

**Remark:** There are a number of degenerate cases for round $i$. For

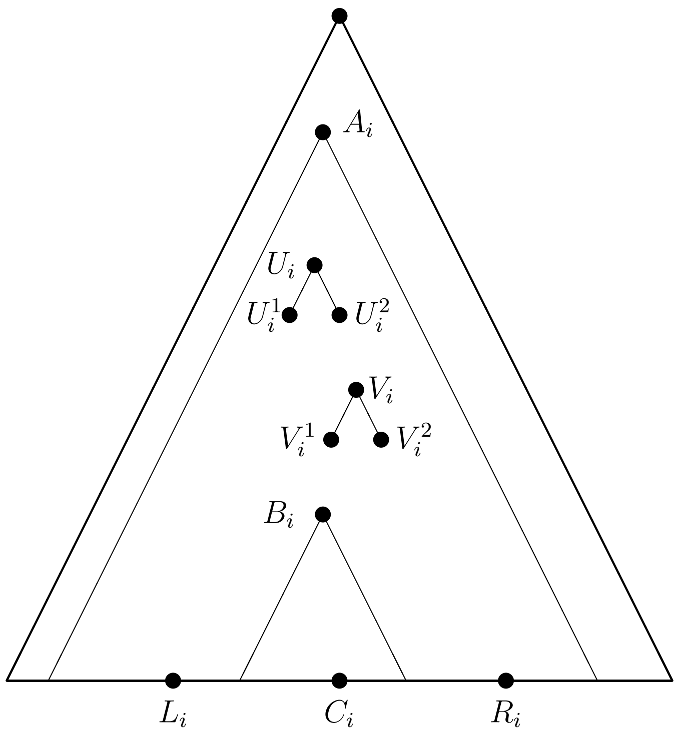



**Figure 2**

Handdrawn lines indicate paths in the tree.

---

example, it is possible that $U_i^2 = V_i$ and that the Pebbler pebbles this node twice in round $i$. If the Pebbler has a winning strategy, it must require that the Pebbler consistently give the same value to a repebbled node. As another degenerate case, we may have $B_i \trianglerighteq U_i, V_i$. In this case, no matter what values are pebbled on the six nodes, the Challenger must rechallenge $A_i$ (or lose the game). Likewise, if all six pebble positions are ancestors of $A_i$ then the Challenger must repebble $A_i$.

To understand the way that the values of $L_{i+1}$, $C_{i+1}$ and $R_{i+1}$ are set in Table 1, consider as an example the case where $U_i \triangleright V_i$ and the Challenger has challenged $U_i^2$. If $U_i^2 = V_i$ then the winner of the game has been determined by this move because the Pebbler has specified values to the inputs $V_i^1$ and

| Challenged Node | Pebbler says $U_i \rhd V_i$ | Pebbler says $V_i \rhd U_i$ |
|---|---|---|
| $V_i^1$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i - 2^{\delta_i}$<br>$L_{i+1} = L_i + 2^{\delta_i}$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i - 2^{\delta_i}$<br>$L_{i+1} = L_i - 2^{\delta_i}$ |
| $V_i^2$ | $C_{i+1} = R_i$<br>$R_{i+1} = R_i + 2^{\delta_i}$<br>$L_{i+1} = R_i - 2^{\delta_i}$ | $C_{i+1} = R_i$<br>$R_{i+1} = R_i + 2^{\delta_i}$<br>$L_{i+1} = R_i - 2^{\delta_i}$ |
| $U_i^1$ | $C_{i+1} = L_i$<br>$R_{i+1} = L_i + 2^{\delta_i}$<br>$L_{i+1} = L_i - 2^{\delta_i}$ | $C_{i+1} = L_i$<br>$R_{i+1} = L_i + 2^{\delta_i}$<br>$L_{i+1} = L_i - 2^{\delta_i}$ |
| $U_i^2$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i + 2^{\delta_i}$<br>$L_{i+1} = L_i + 2^{\delta_i}$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i - 2^{\delta_i}$<br>$L_{i+1} = L_i + 2^{\delta_i}$ |
| $U_i$ or $V_i$ | Game ends | Game ends |

| Challenged Node | Pebbler says $A_i \rhd U_i, V_i$ | Pebbler says $U_i, V_i \unrhd A_i$ | Pebbler says $U_i \unrhd A_i \rhd V_i$ | Pebbler says $V_i \unrhd A_i \rhd U_i$ |
|---|---|---|---|---|
| $A_i$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i + 2^{\delta_i}$<br>$L_{i+1} = L_i - 2^{\delta_i}$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i - 2^{\delta_i}$<br>$L_{i+1} = L_i + 2^{\delta_i}$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i + 2^{\delta_i}$<br>$L_{i+1} = L_i + 2^{\delta_i}$ | $C_{i+1} = C_i$<br>$R_{i+1} = R_i - 2^{\delta_i}$<br>$L_{i+1} = L_i - 2^{\delta_i}$ |

**Tables 1 and 2**

---

$V_i^2$ and to the output $U_i^2 = V_i$ of a Boolean gate and because the Challenger has challenged the value of the output, thereby accepting the Pebbler's values for the inputs. So in this case either the Pebbler or the Challenger has made an obviously false assertion. On the other hand, suppose $U_i^2 \rhd V_i$ so the game may still be open. Referring to Table 1, this implies $C_{i+1} = C_i$, $L_{i+1} = L_i + 2^{d-i-1}$, $R_{i+1} = R_i + 2^{d-i-1}$, $A_{i+1} = U_i^2$, and $B_{i+1} = V_i$. It is easily seen that every leaf $W$ such that $U_i^2 \unrhd W$ and $V_i \not\unrhd W$ has number in the set

$$\left(L_i, L_i + 2^{d-i}\right) \cup \left(R_i, R_i + 2^{d-i}\right).$$

This is because condition $(\delta)$ held at the beginning of round $i$. Of course, this set is equal to

$$\left(L_{i+1} - 2^{d-i-1}, L_{i+1} - 2^{d-i-1}\right) \cup \left(R_{i+1} - 2^{d-i-1}, R_{i+1} - 2^{d-i-1}\right).$$

Thus condition $(\delta)$ holds again for round $i+1$. To see that condition $(\gamma)$ holds at round $i+1$, note that it is trivial that if $L_i$ and $R_i$ have rank $d-i$, then $L_i \pm 2^{d-i-1}$ and $R_i \pm 2^{d-i-1}$ have rank $d-i-1$.

The rest of the cases in Table 1 where $U_i \rhd V_i$ are easily checked in same fashion with the aid of Figure 2. The cases $V_i \rhd U_i$ are reflections of the cases where $U_i \rhd V_i$. We leave it to the reader to verify the rest of Tables 1 and 2.

Now we define precisely who the winner of a play of the game is — this is done by defining the loser of the game by formalizing what an 'obvious mistake' or 'obviously false assertion' is. It also must be shown that (a) someone loses the game by the end of Round $d = \lfloor \log_2 n \rfloor$, and (b) Pebbler has a winning strategy iff the Boolean sentence has value 1, and (c) the loser of a particular play of the game can be determined in ALOGTIME.

**Definition** The 'obvious mistakes' that cause a player to lose the game are:

- Asserting incompatible values to the inputs and output of a gate. For the Pebbler this occurs when the input nodes are either leaves or are pebbled and the output node has been pebbled incompatibly. For the Challenger, this occurs when he challenges the output of a gate whose input nodes are either leaves or pebbled.
- Asserting an incorrect value for a leaf. For the Pebbler, this can occur if she pebbles a leaf incorrectly; for the Challenger, if he challenges a correctly pebbled leaf.
- (Pebbler only) Being incorrect about whether $U_i \rhd V_i$, $A_i \rhd U_i$, $A_i \rhd V_i$.
- (Pebbler only) Pebbling a node with both $0$ and $1$.
- (Challenger only) Challenging a node which is above a previously challenged node.
- (Challenger only) Challenging a node which is at or below a previously agreed upon pebble value. A pebble value is defined to be 'previously agreed upon' if the pebble was placed in an earlier round and in that round the Challenger challenged an ancestor of the pebble.

By condition ($\delta$), there are at most $2^{d-i+1}$ leaves which are below $A_i$ but not at or below $B_i$. It follows that in round $d$ there are at most two such leaves and thus, since the sentence contains only binary connectives, one of the players will be forced into an 'obvious mistake' before round $d$. Furthermore, the Pebbler (respectively, the Challenger) has a winning stategy iff the Boolean sentence has value 1 (respectively, 0); since, for the Pebbler the winning strategy is to always tell the truth and, for the Challenger, the winning strategy is to always challenge a lowest incorrectly pebbled node which is not below a previously agreed upon node. It remains to show that the winner of a play of the game can be determined in ALOGTIME. For this, it will suffice to show that the positions $L_i$, $C_i$, $R_i$, $A_i$ and $B_i$ can be obtained in ALOGTIME since then the winner can be determined in ALOGTIME by nondeterministically guessing the first 'obvious mistake', checking that it is indeed an obvious mistake, and then branching universally to verify that there was no earlier obvious mistake. We first explain how to obtain $L_i$ and $R_i$ in ALOGTIME: in the description of the game, we always have $L_{i+1} = L_i \pm \delta_i$ or $L_{i+1} = R_i - \delta_i$ and have $R_{i+1} = R_i \pm \delta_i$ or $R_{i+1} = L_i + \delta_i$, where $\delta_i = 2^{d-i-1}$. Assignments of the form $L_{i+1} = R_i - \delta_i$ and $R_{i+1} = L_i + \delta_i$ are called *mixed*. We say that the next mixed assignment after round $i$ occurs in round $j$ if there is a mixed assignment at round $j$ and no mixed assignment in any round $k$ with $i < k < j$; if the mixed assignment in round $j$ is of the form $L_{j+1} = R_j - \delta_j$ then we define $LR_i$ to be equal to $R_i$, otherwise the mixed assignment is $R_{j+1} = L_j + \delta_j$ and $LR_i$ is defined to be equal to $L_i$. If there is no mixed assignment after round $i$, then $LR_i$ is arbitrarily defined to be $L_i$. Since it is easy to tell whether an assignment is mixed in a round $k$ of a play of the pebbling game by referring to Table 1—the question of whether $LR_i$ equals $R_i$ or $L_i$ is easily resolved in ALOGTIME. Now $LR_{i+1}$ is always equal to $LR_i \pm \delta_i$ and whether it is plus or minus $\delta_i$ is easily determined from the play of the game and the rules of Table 1. Thus $LR_i$ can be determined by summing a vector of values $\pm\delta_i$; this is a particularly easy vector summation and is in ALOGTIME. To find the value of $L_i$, first find the least value of $j$ such that for all $j \le k < i$, $L_{k+1}$ is defined by an unmixed assignment $L_{k+1} = L_k + \epsilon_k\delta_k$ with $\epsilon_k = \pm 1$ (so $j = 0$ or $L_j$ is given by a mixed assignment). Then, if $j = 0$, $L_i = 2^{d-1} + \sum_{k=0}^{i-1} \epsilon_k\delta_k$ and if $j > 0$, $L_i = LR_j + \sum_{k=0}^{i-1} \epsilon_k\delta_k$. This vector summation provides an ALOGTIME algorithm to compute $L_i$. Essentially the same construction gives an ALOGTIME algorithm for determining $R_i$.

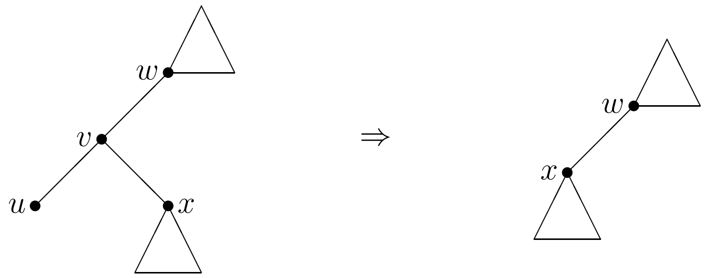
To calculate $C_i$, one determines the maximum value $j \leq i$ such that $C_j = L_{j-1}$ or $R_{j-1}$ according to Table 1, or, if no such $j$ exists, use $j = 0$; and then $C_i = C_j$. To calculate $U_i$, $U_i^1$, $U_i^2$, $V_i$, $V_i^1$, and $V_i^2$ is now easy since they are the least common ancestors of $L_i$, $R_i$ and $C_i$ and the children of the least common ancestors. Finally, $A_i$ and $B_i$ may be found from the values of the $U$'s and $V$'s and the play of the game according to Table 1.

That completes the proof that there is an alternating logarithmic time algorithm for the Boolean sentence value problem.

# 3 ALOGTIME Tree Contraction

The algorithm of the previous section is based on a method of splitting a tree (representing a formula) into subtrees so that the problem of finding the value of the tree is reduced to finding values of the subtrees. The subtrees are generated by picking breakpoints (the nodes $U_i$ and $V_i$). There is an elegant way to describe the breakpoints which are picked in the $i$-th round; namely, they are least common ancestors of the leaves of rank $\geq d - i$. The algorithm for the BSVP was presented as a Pebbling game — in essence, this is a top-down procedure. However, in parallel processing applications it is often useful to have a bottom-up algorithm which can be executed efficiently in parallel by multiple processors. Such algorithms have been given by Brent [3] Miller-Reif [8] and Abrahamson, et. al. [1]; these algorithms allow $n/\log n$ many processors to evaluate a Boolean sentence in time $O(\log n)$. All of these algorithms are based on tree contraction algorithms; the tree contraction algorithm of Abrahamson, et. al. is particularly elegant and direct, and in this section we show that their tree contraction algorithm is describable (computable) in alternating log time.

We first briefly review Abrahamson, et. al.'s tree contraction algorithm. They construct a tree by applying "prune" and "bypass" procedures which transform a binary tree as pictured: In Figure 3 the leaf node $u$ is removed in the prune operation and then its parent node $v$ is removed (bypassed) and $v$'s other child becomes the child of $v$'s parent $w$ (the triangles represent arbitrary subtrees). Clearly the prune-bypass operation preserves the property of being a binary tree. If $x$ and $y$ are nodes in a binary tree then we say $x$ is a *right (left) descendent* of $y$ iff $x$ is the right (left) child of $y$ or $x$ is a descendent of that child. It is clear that if $x$ and $y$ are not removed in a prune-bypass




The Prune-Bypass Operation Applied to $u$
**Figure 3**

---

operation, then the property of $x$ being a left (right) descendent of $y$ is preserved.

The Abrahamson, et. al. tree contraction algorithm is:

**Input:** A binary, ordered tree $T_0$ with $n$ leaves
**Step (1):** Number the leaves from 1 to $n$ in left-to-right order
**Step (2):** Loop with $i = 1, \ldots, \lfloor \log n \rfloor$:
Apply the prune-bypass operation to all leaves of odd number
Divide the remaining leaf numbers by 2 and call the resulting tree $T_i$.
Endloop

We have made some inessential modifications to the algorithm. First, nodes are numbered beginning with 1 instead of 0. Second, we allow the root node to be bypassed (in Figure 3, this would make $x$ the new root). Third, the prune-bypass operations may be carried out in any order since we are only concerned with describing the trees $T_i$; however, Abrahamson, et. al.'s technique of pruning and bypassing first the leaves of number congruent to 1 mod 4 and then the leaves of number congruent to 3 mod 4 still works to give an efficient parallel algorithm.

Clearly each $T_i$ is a binary tree and $T_i$ has exactly $\lfloor n/2^i \rfloor$ leaves. Thus $T_{\lfloor \log n \rfloor}$ has exactly one node and the tree contraction is completed at this stage. The internal nodes in $T_i$ are also internal nodes of $T_0$; likewise every leaf of $T_i$ was a leaf of $T_0$. The $T_0$-*number* and $T_0$-*rank* of a leaf are defined to be the number and rank that the leaf originally has in $T_0$.

**Theorem 2** (a) *The leaves of $T_i$ are precisely the leaves $x$ of $T_0$ which have $T_0$-rank $\geq i$.*

(b) *If $x$ and $y$ are nodes in $T_i$ then their least common ancestor in $T_i$ is the same as their least common ancestor in $T_0$.*

(c) *An internal node $u$ of $T_0$ is a node in $T_i$ iff $u$ is the least common ancestor of two leaves of ranks $\geq i$.*

**Proof** (a) is trivial from the definition of the tree-contraction algorithm. (b) is easily proved by noting that least common ancestors are preserved by a prune-bypass operation. For half of (c), note that (a) and (b) imply that the least common ancestor of two leaves of ranks $\geq i$ is a node in $T_i$. For the other half of (c), suppose it is not the case that $u$ is the least common ancestor of two leaves of ranks $\geq i$. In this case, it must be the case that either every right descendent leaf of $u$ has $T_0$-rank $< i$ or (dually) every left descendent leaf of $u$ in $T_0$ has $T_0$-rank $< i$. But since leaves of $T_0$-rank $< i$ do not appear in $T_i$, it is impossible for $u$ to be a node in $T_i$ since $u$ would have no right (respectively, left) descendent leaf in $T_i$ contradicting the fact that $T_i$ is a binary tree with $u$ an interior node.

It follows immediately from Theorem 2 that the trees $T_i$ are ALOGTIME computable from $T_0$. If $x$ is a node in $T_0$, it can be determined if $x$ is a node in $T_i$ by either checking if $x$ is a leaf of rank $\geq i$ or guessing two leaves $y$ and $x$ with $T_0$-rank $\geq i$ and verifying that $x$ is the least common ancestor of $y$ and $z$.

# 4 A log-log space algorithm for a balanced BSVP

In this final section we give a very simple algorithm for the BSVP when the logical connectives are restricted to be $\land$ and $\lor$. This algorithm runs in logspace but is simpler than N. Lynch's log space algorithm for the BSVP with arbitrary binary connectives [7]. When our algorithm is applied to balanced Boolean sentences, it uses only $O(\log\log n)$ space on inputs of length $n$. This yields the rather surprising consequences: (1) every ALOGTIME predicate is deterministic log time reducible to a $\log\log n$ space problem, and (2) there are linear length, log width branching programs for the Balanced BSVP restricted to connectives $\land$ and $\lor$. Consequence (1) follows since it is known that the Balanced BSVP restricted to $\land$ and $\lor$ is complete for ALOGTIME under deterministic log time reductions. It is open whether the $\log\log n$ space bound is optimal.

Since $\log\log n$ space is not space constructible [6], it is useful to think of a $\log\log n$ space Turing machine being initialized with a workspace of size $\log\log n$; however, we shall only consider inputs on which the Turing machine will use only $\log\log n$ space and, in particular, the Turing machine gains no advantage from being able to sense the ends of its worktape. A $\log\log n$-space Turing machine accesses its input via a usual tape head which may be moved left or right one square at a time. However, the $\log\log n$ space algorithm below will only move the tape head obliviously from left-to-right with no backing up.

**Theorem 3** *There is a* $\log\log n$ *space algorithm for the Balanced BSVP restricted to connectives* $\land$ *and* $\lor$*. Moreover, this algorithm scans the input from left-to-right only and its only memory is a single counter which may by incremented by one, decremented by one and tested for zero.*

A Boolean sentence of $n$ symbols is said to be balanced if it has depth $\lceil \log_2 n \rceil$; the algorithm of Theorem 3 actually uses space $\log d$ on inputs which are Boolean sentences of depth $d$.

The essential idea of the algorithm is based on the following property of the connectives $\land$ and $\lor$: either the value of the first operand of an $\land$ (or $\lor$) determines the output value or the value of the second operand is equal to the output value (and the value of the first operand determines which case holds). Thus, in the first case, the second operand may be ignored, while in the second case, the value of first operand and the gate type may be forgotten since the second operand's value is equal to the output value.

**Algorithm for BSVP with connectives $\wedge$ and $\vee$:**

**Input:** a Boolean sentence $\varphi$.
**Initialize:** $\mathit{Ignore\_Flag} = 0$
$\mathit{Ignore\_Depth} = 0$
Input tape head at leftmost symbol of $\varphi$
**Loop:**
Read current symbol $\alpha$ from input tape
**If** $\mathit{Ignore\_Flag} = 1$
If $\alpha =$ "(", increment *Ignore_Depth* by 1
**If** $\alpha =$ ")"
decrement *Ignore_Depth* by 1
if $\mathit{Ignore\_Depth} = 0$, set $\mathit{Ignore\_Flag} = 0$
**Endif**
**Endif**
**If** $\mathit{Ignore\_Flag} = 0$
If $\alpha =$ "0" set $\mathit{Val} = 0$
If $\alpha =$ "1" set $\mathit{Val} = 1$
If $\alpha =$ "$\wedge$" and $\mathit{Val} = 0$, set $\mathit{Ignore\_Flag} = 1$
If $\alpha =$ "$\vee$" and $\mathit{Val} = 1$, set $\mathit{Ignore\_Flag} = 1$
**Endif**
Move input tape head right one square
**Endloop**
**Output:** *Val*

This algorithm is easily implemented on a Turing machine which scans its input from left-to-right. The one bit registers *Val* and *Ignore_Flag* may be remembered by the finite state control. The *Ignore_Depth* register may be incremented, decremented and tested for zero; its maximum value is the depth of the Boolean formula. Thus if the Boolean formula is balanced, *Ignore_Depth* may be stored in $\log\log n$ space.

**Corollary 4** *Every ALOGTIME predicate is deterministic log time, many-one reducible to* $\log\log n$ *space.*

This corollary is immediate from Theorem 3 since in [4, 5] it is shown that the Balanced BSVP restricted to connectives $\wedge$ and $\vee$ is complete for ALOGTIME under deterministic log time reductions. Since $\log\log n$ is not space constructible, it is useful to define a many-one reduction of a decision problem $A \subset \Sigma^*$ to $\log\log n$ space to be a function $f$ and a Turing machine $M$ such that, for all $x \in \Sigma^*$, $x \in A$ iff $f(x)$ is accepted by $M$ and such that, for all $x \in \Sigma^*$, $M$ uses at most $\log\log|f(x)|$ space on input $f(x)$.

**Theorem 5** *The Balanced BSVP, with* $\wedge$*'s and* $\vee$*'s only, has logarithmic width, linear length branching programs.*

See Barrington [2] for information on branching programs. Theorem 5 should be compared with Barrington's theorem that the Balanced BSVP has quadratic length, constant width branching programs.

**Proof** Let $0 < d \leq n$. We shall show how to contruct a branching program of width $4 \cdot d$ and length $n$ which recognizes true Boolean sentences of length $n$ and depth $\leq d$. Taking $d = \lceil \log n \rceil$ this proves the theorem. The branching program can be viewed as a graph on $n+1$ columns of height $4d$. Each of the $4d$ rows corresponds to a particular set of values for *Ignore_Flag*, *Val*, and *Ignore_Depth*. The transition rules from the $i$-th column to the $i+1$-st column is based on the $i$-th symbol of the input in the obvious way based on the above algorithm. □

A natural question to ask is whether Theorem 3 can be used to separate the class P of polynomial time predicates from ALOGTIME; since, after all, both $\log\log n$ space and deterministic log time are proper subclasses of $P$. However, it is not possible to apply Theorem 3 to provide a straightforward polynomial time diagonalization of ALOGTIME: the problem is that the deterministic log time reductions may be functions of arbitrarily high degree polynomial growth rate and these can not be computed by a single polynomial time algorithm.


## Acknowledgements

I wish to thank Pierre McKenzie for bringing the tree-contraction algorithm of Abrahamson-Dadoun-Kirkpatrick-Przytycka to my attention; this was the catalyst for the work of this paper. I benefitted greatly from long-ago conversations with Vijaya Ramachandran about formula evaluation. Pavel Pudlák suggested an improvement which has been incorporated into the $\log\log n$ space algorithm.

## References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka, *A simple parallel tree contraction algorithm*, Journal of Algorithms, 10 (1989), pp. 287–302.

[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] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[4] 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.

[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] A. R. Freedman and R. E. Ladner, *Space bounds for processing contentless inputs*, Journal of Computer and System Sciences, 11 (1975), pp. 118–128.

[7] N. A. Lynch, *Log space recognition and translation of parenthesis languages*, J. Assoc. Comput. Mach., 24 (1977), pp. 583–590.

[8] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, in Proceedings of the 26th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, 1985, pp. 478–489.