The Boolean formula value problem is in ALOGTIME (Preliminary Version)

Samuel R. Buss*
Department of Mathematics
University of California, Berkeley

January 1987

Abstract

The Boolean formula value problem is in alternating log time and, more generally, parenthesis context-free languages are in alternating log time. The evaluation of reverse Polish notation Boolean formulas is also in alternating log time. These results are optimal since the Boolean formula value problem is complete for alternating log time under deterministic log time reductions. Consequently, it is also complete for alternating log time under AC^0 reductions.

1. Introduction

The Boolean formula value problem is to determine the truth value of a variable-free Boolean formula, or equivalently, to recognize the true Boolean sentences. N. Lynch [11] gave log space algorithms for the Boolean formula value problem and for the more general problem of recognizing a parenthesis context-free grammar. This paper shows that these problems have alternating log time algorithms. This answers the question of Cook [5] of whether the Boolean formula value problem is log space complete — it is not, unless log space and alternating log time are identical. Our results are optimal since, for an appropriately defined notion of log time reductions, the Boolean formula value problem is complete for alternating log time under deterministic log time reductions; consequently, it is also complete for alternating log time under AC^0 reductions. It follows that the Boolean formula value problem is not in the log time hierarchy.

There are two reasons why the Boolean formula value problem is interesting. First, a Boolean (or propositional) formula is a very fundamental concept

*Supported in part by an NSF postdoctoral fellowship.
The author's electronic address is `buss@cartan.berkeley.edu`.

of logic. The computational complexity of evaluating a Boolean formula is therefore of interest. Indeed, the results below will give a precise characterisation of the computational complexity of determining the truth value of a Boolean formula. Second, the existence of an alternating log time algorithm for the Boolean formula problem implies the existence of log depth, polynomial size circuits for this problem and hence there are (at least theoretically) good parallel algorithms for determining the value of a Boolean sentence.

As mentioned above, N. Lynch [11] first studied the complexity of the Boolean formula problem. It follows from Lynch's work that the Boolean formula value problem is in NC^2, since Borodin [1] showed that $LOGSPACE \subseteq NC^2$. Another early significant result on this problem was due to Spira [17] who showed that for every formula of size n, there is an equivalent formula of size $O(n^2)$ and depth $O(\log n)$. An improved construction, which also applied to the evaluation of rational expressions, was obtained by Brent [2]. Spira's result was significant in part because because it implied that there might be a family of polynomial size, log depth circuits for recognizing true Boolean formulas. In other words, that the Boolean formula value problem might be in (non-uniform) NC^1. However, it was not known if the transformations of formulas defined by Brent and Spira could be done in NC^1. Recent progress was made by Miller and Reif [13] who found improved algorithms for computing rational functions on PRAM's in $O(\log n)$-time; however, they did not give improved circuits for the Boolean formula value problem. Very recently, Cook and Gupta [7] and, independently, Ramachandran [14] were successful in giving polynomial size, $O(\log n \log\log n)$-depth circuits for the Boolean formula value problem.

Our result improves on Cook, Gupta and Ramachandran since alternating log time is a subset of NC^1. Indeed, Ruzzo [15] showed that alternating log time is the same as U_{E^*}-uniform NC^1, where in this setting U_{E^*}-uniformity is a kind of alternating log time uniformity. It is shown below that alternating log time can also be characterized as the set of decision problems which have $ALOGTIME$-uniform formulas. We do not use Spira's or Brent's restructuring methods but instead give a new algorithm based on a transformation of Boolean formulas to Postfix-Longer-Operand-First ($PLOF$) formulas. It is still open whether Spira's and Brent's original restructuring algorithms are in NC^1 or $ALOGTIME$.

Definition: Let Σ be the alphabet $\{\wedge, \vee, \neg, 0, 1, (,)\}$. The *Boolean formulas* are given by the following inductive definition:

(a) 0 and 1 are Boolean formulas.

(b) If α and β are Boolean formulas, then so are $(\neg\alpha)$, $(\alpha \wedge \beta)$ and $(\alpha \vee \beta)$.

Definition: $|\alpha|$ is the length of α, i.e., the number of symbols in the string α.

The above definition of Boolean formulas uses $\wedge$ and $\vee$ as infix operators. For our purposes it is also useful to consider formulas in postfix (reverse Polish) notation. The postfix formulas we consider satisfy an unusual additional condition; namely, that for any binary operator, the longer operand must appear first.

Definition: The *Postfix-Longer-Operand-First* formulas, or $PLOF$ formulas, are defined by the following inductive definition:

(a) 0 and 1 are $PLOF$ formulas.

(b) If α is a $PLOF$ formula then so is $\alpha\neg$.

(c) If α and β are $PLOF$ formulas and if $|\alpha| \geq |\beta|$ then $\alpha\beta\vee$ and $\alpha\beta\wedge$ are $PLOF$ formulas.

The value of a Boolean or $PLOF$ formula is defined in the usual way, where 0 and 1 represent *False* and *True*, respectively.

Main Theorem 1 *The problem of determining the truth value of a Boolean formula (or: $PLOF$ formula) is in alternating log time.*

An important area in research in computational complexity is to study the circuit complexity of various decision problems (see Savage [16] or Cook [5] for instance). A related, but less popular, approach to computational complexity is to study the *formula complexity* of decision problems where a formula is a circuit with fan-out one. It is well known that in the non-uniform setting a decision problem has polynomial size, log depth circuits if and only if it has polynomial size formulas. (One direction of the proof just expands the circuit to a formula, the other uses the result of Spira mentioned above.) Ruzzo [15] showed that a decision problem has U_{E^*}-uniform polynomial size, log depth circuits if and only if it is in $ALOGTIME$. The Main Theorem 1 plus the discussion on reductions in §3 allows us to improve this to

Theorem 2 *Let D be a decision problem. The following are equivalent:*

(a) *D is in $ALOGTIME$;*

(b) *D has $ALOGTIME$-uniform, polynomial size formulas.*

(c) *D has $ALOGTIME$-uniform, polynomial size, log depth formulas.*

Corollary 3 *There are polynomial size, log depth, $ALOGTIME$-uniform formulas for the Boolean formula value problem.*

The equivalence of (a) and (c) in Theorem 2 is essentially a restatement of Ruzzo's theorem. (See §2 for the definition of $ALOGTIME$-uniform.)

The outline of this paper is as follows: In §2 we present some background on $ALOGTIME$, define *$ALOGTIME$-uniform* and how the Boolean formula value problem is presented to an $ALOGTIME$ Turing machine, and discuss the definability of syntactic notions in $ALOGTIME$. In §3 log time reductions between decision problems are defined; these reductions preserve the property of being alternating log time computable or being in the log time hierarchy. In

§4 we give an $ALOGTIME$ reduction of the Boolean formula value problem to the $PLOF$ formula value problem. Then in §5 an $ALOGTIME$ algorithm for the latter problem is given, thus finishing the proof of Main Theorem 1. In §6 we show that recognizing any parenthesis language (in the sense of Lynch) has an $ALOGTIME$ algorithm. Finally, §7 discusses various problems which are complete for alternating log time under deterministic log time reductions.

2. Background on Alternating Log Time

Our basic model for computation is the multitape Turing machine. The random access model of Chandra, Kozen and Stockmeyer [3] is adopted; namely, the input tape is randomly accessed rather than being scanned sequentially: there is a special work tape onto which an address i can be written in binary and each state of the Turing machine can read the i-th input symbol. This convention is necessary since otherwise not all of the input could be reached within $O(\log n)$ time. The usual definition for alternating log time ($ALOGTIME$) is adopted, so

$$ALOGTIME = \bigcup_c ATIME(c \cdot \log(n) + c).$$

Here $ATIME(h(n))$ denotes the class of predicates computable in time $h(n)$ with unbounded alternations; similarly, $TIME(h(n))$ and Σ_k-$TIME(h(n))$ are the classes of predicates computable in time $h(n)$ with no alternations and with k alternations (beginning with existential states), respectively. Thus,

$$LOGTIME = \bigcup_c TIME(c \cdot \log(n) + c).$$

$$\Sigma_k\text{-}LOGTIME = \bigcup_c \Sigma_k\text{-}TIME(c \cdot \log(n) + c).$$

The log time hierarchy is $LH = \bigcup_k \Sigma_k$-$LOGTIME$.

Strictly speaking, $ALOGTIME$ is a set of predicates; however, we shall say a function f is in $ALOGTIME$ if and only if the predicate

$$A_f(c, i, z) \iff \text{“the } i\text{-th symbol of } f(z) \text{ is } c\text{”}$$

is in $ALOGTIME$. Similar conventions apply to a function being in $LOGTIME$ or in Σ_k-$LOGTIME$. Any such function must have polynomial growth rate.

It is well known that every symmetric function is in non-uniform NC^1; in addition, Ruzzo showed that every U_{E^*}-uniform NC^1 function is in $ALOGTIME$. It follows that any sufficiently uniform, symmetric function is in $ALOGTIME$. An important example is the counting function; namely, let $Count$ be the symmetric function from $\{0,1\}^*$ to $\mathbb{N}$ so that

$$Count(x_1, \ldots, x_n) \ = \ \text{the number of } x_j\text{'s equal to 1.}$$

The function *Count* is in fact in $ALOGTIME$ as the usual polynomial size, log depth circuits for A_{Count} are U_{E^*}-uniform.

A generalization of the *Count* function is the summation of a vector of integers. The function which computes the sum of n integers each n bits long is also in $ALOGTIME$.

For the purposes of describing alternating log time Turing machines we assume that the alphabet of the Turing machine contains the alphabet Σ for Boolean formulas plus additional work symbols and input and output symbols as required. Of course, when using a family of circuits or formulas for computing a predicate it is necessary to code the input in binary. This is can be done by coding each input symbol as a fixed length string of bits; for example, since Σ has seven symbols, a three bit code could be used. Similar conventions would apply to coding the output of a Turing machine.

The concept of what it means for a family of circuits to compute a predicate is well known and widely used (Savage [16]). Another related concept is that of a family of formulas. One definition of a formula is that it is a circuit with fanout one. A formal definition is:

Definition: Let Σ^+ be $\Sigma \cup \{x\}$. The *formulas in the wide sense* are defined inductively by;

(1) If $w \in \{0,1\}^*$, i.e., w is a string of 0's and 1's, then xw is a formula in the wide sense. When w is the dyadic representation of $i \in \mathbb{N}$, then xw is the literal denoting the variable x_i.

(2) 0 and 1 are formulas in the wide sense.

(3) If α and β are formulas in the wide sense, then so are $(\neg\alpha)$, $(\alpha \wedge \beta)$ and $(\alpha \vee \beta)$.

We generally refer to formulas in the wide sense as just formulas and let the context distinguish them from Boolean formulas[†]. A *family of formulas* is an infinite sequence of formulas such that the n-th formula uses only literals $x_1, ..., x_n$. A given predicate is computed by a family of formulas if and only if the value of the predicate on inputs n bits long is expressed by the n-th formula with the literal x_i denoting the i-th input bit.

Ruzzo [15] discusses the relationship between a large number of uniformity conditions on circuits. We shall use a condition called $ALOGTIME$-uniformity for formulas. This is similar in spirit to the U_{BC}-uniformity of Ruzzo, originally introduced by Borodin and Cook [4]; except that alternating log time computablility is substituted for log space computability.

Definition: A family of formulas is *$ALOGTIME$-uniform* if and only if the predicate F is in $ALOGTIME$ where

[†]A better but nonstandard usage would be to use the name "Boolean sentence" for what we called "Boolean formulas", as "sentence" is a common designation for a variable-free formula.

$F(c, i, 0^n) \iff$ "the n-th formula has c as its i-th symbol".

It is easy to verify that $ALOGTIME$-uniform, polynomial size, log depth formulas are U_{E^*}-uniform circuits in the sense of Ruzzo. Conversely, a family of U_{E^*}-uniform circuits has an equivalent family of $ALOGTIME$-uniform, polynomial size, log depth formulas.

A natural and important first question is whether the recognition problem for Boolean formulas is in $ALOGTIME$; in other words, is there an alternating log time Turing machine, which given as input a string of symbols from Σ, determines if the input is a Boolean formula. The answer is yes; the $ALOGTIME$ algorithm uses the fact that counting is $ALOGTIME$ computable. (Ibarra, Jiang and Ravikumar [9] use a different approach and obtain the stronger result that one-sided Dyck languages on k letters are in $ALOGTIME$.) For our purposes, it is convenient to describe the alternating log time Turing machine by specifying a finite game with two players: the first player is the `Affirmer` and the second is the `Denier` — the `Affirmer`'s plays are existential moves for the Turing machine and the `Denier`'s plays are universal moves. Of course, to obtain an alternating log time algorithm, the game must have $O(\log n)$ plays; each play is one bit of information. Furthermore it must be possible to determine who won the game in alternating log time. The alternating log time Turing machine will accept if and only if the `Affirmer` has a winning stategy for the game (since the game is finite, one of the players will have a winning strategy). To describe an algorithm to recognize Boolean formulas, consider the following game:

The input string is $A = \alpha_1\alpha_2\cdots\alpha_n$ where each $\alpha_i \in \Sigma$; assume $n > 1$. Let $A[i, j]$ denote the substring $\alpha_i\alpha_{i+1}\cdots\alpha_j$ of A. The moves of the game are as follows.

- The `Denier` first plays a number i such that $1 \leq i < n$. It takes $\log(n)$ bits to specify i.
- The `Affirmer` then plays two numbers j and k such that $1 \leq j \leq i < k \leq n$. This takes $2\log(n)$ bits.
- That concludes the game.

The winner of the game is determined as follows:

(a) If $\alpha_i\alpha_{i+1}$ are an illegal pair of adjacent symbols, say (),)(, 0(, $\wedge\vee$, $0\neg$, etc., then the `Denier` wins. Also, if $i = 1$ but α_1 is not (then the `Denier` wins.

(b) Otherwise, if α_i is the symbol $\neg$, then the `Affirmer` will win if and only if $j = i - 1$, α_j is (, α_k is) and the substring $A[j, k]$ has equal numbers of left and right parentheses.

(c) Otherwise, if α_i is $\wedge$ or $\vee$, then the Affirmer will wins if and only if α_j is (, α_k is) and both of the substrings $A[j + 1, i - 1]$ and $A[i + 1, k - 1]$ have equal numbers of left and right parentheses.

(d) If none of the above apply then the `Affirmer` wins the game.

It is not difficult to see that the game is correctly designed so that the `Affirmer` has a winning strategy if and only if A is a Boolean formula; the `Affirmer`'s strategy is to pick j and k so that $A[j,k]$ is the subformula of A with α_i as its outermost logical connective. The game obviously has only $O(\log n)$ plays and since *Count* is in alternating log time, there is an alternating log time algorithm for determining which player won.

There is actually a slightly simpler algorithm for recognizing Boolean formulas; however, the point of using the above algorithm was to demonstrate how alternating log time suffices for parsing Boolean formulas. This technique can be used to defining many simple syntactic properties of formulas. It should be clear that the following predicate on a string $A = \alpha_1 \cdots \alpha_n$ is in $ALOGTIME$:

$$\begin{aligned} Scope(i,j,A) \iff & \text{“}\alpha_i \text{ is a connective } \neg, \vee \text{ or } \wedge \text{ and } \alpha_j \text{ is a symbol} \\ & \text{in the scope of } \alpha_i\text{”}. \end{aligned}$$

The definition of α_j being in the scope of α_i is that α_i is a logical connective and α_j is a symbol in one of the operands of α_i.

Suppose that α_i and α_j are not parentheses; we say that α_i is *to the left of* α_j if and only if there is a subformula $(\beta \vee \gamma)$ or $(\beta \wedge \gamma)$ of A such that either $|\beta| \geq |\gamma|$, α_i is in β and α_j is in γ, or $|\gamma| > |\beta|$, α_i is in γ and α_j is in β. The predicates

$$\begin{aligned} Left(i,j,A) \iff & \text{“Neither } \alpha_i \text{ nor } \alpha_j \text{ is a parenthesis and} \\ & \alpha_i \text{ is to the left of } \alpha_j \text{ in } A\text{”}, \\ Before(i,j,A) \iff & Left(i,j,a) \vee Scope(i,j,A). \end{aligned}$$

are in $ALOGTIME$.

Similar predicates for reverse Polish notation formulas are also in alternating log time. The recognition problem for such formulas is easily solved using counting: suppose A is a string containing only the symbols 0, 1, $\neg$, $\vee$, $\wedge$. Assign weights to these characters as follows:

$$k(\alpha) = \begin{cases} -1 & \text{if } \alpha \text{ is } \vee \text{ or } \wedge \\ 0 & \text{if } \alpha \text{ is } \neg \\ 1 & \text{if } \alpha \text{ is 0 or 1.} \end{cases}$$

Then A is a reverse Polish notation formula if and only if (1) $\sum_{i=1}^{n} k(\alpha_i) = 1$ and (2) for each $1 \leq j < n$, $\sum_{i=1}^{j} k(\alpha_i) > 0$. This is easily checked in alternating log time since the summation of a vector of integers is in alternating log time (or alternatively, since all the weights are -1, 0 or 1, it can computed by just using counting).

It is also easy to see that the recognition problem for $PLOF$ formulas is in $ALOGTIME$.

3. Log Time Reductions

An important tool for the characterization of the computation complexity of decision problems is the use of reductions. When A and B are decision problems, i.e., predicates, a *(many-one) reduction from A to B* is a unary function f such that for all x, $x \in A$ if and only if $f(x) \in B$.

Definition: Let $\mathcal{C}$ be a complexity class (e.g., $LOGTIME$, Σ_k-$LOGTIME$, LH, $ALOGTIME$) and let f be a reduction of A to B. Then f is a $\mathcal{C}$-reduction of A to B if and only if f is of polynomial growth rate and the predicate A_f is in $\mathcal{C}$ where A_f is defined as in §2.

Note Σ_k-$LOGTIME$ reductions are the same as Π_k-$LOGTIME$ reductions.

The next two theorems express the properties we need reductions to satisfy: Theorem 4 helps us establish the upper bound for the complexity of the Boolean formula value problem and Theorem 5 gives the means to demonstrate a lower bound.

Theorem 4 *If $B \in ALOGTIME$ and there is an $ALOGTIME$ reduction from A to B then $A \in ALOGTIME$.*

Proof: Let M be an alternating Turing machine which accepts B in time $c \cdot \log(n) + c$. (Our logarithms are always base 2.) So for any input of length n, M halts within $c \cdot \log(n) + c$ steps on every computational path. Let f be an alternating log time reduction from A to B. We construct an alternating Turing machine N which accepts A in $O(\log n)$ time. On input x, the machine N runs as follows:

(1) First N determines an upper bound m^* to the length $|f(x)|$ of the input on which M will be simulated. N does this by computing the length n of its own input x and then determining m^*. Indeed, since f is of polynomial growth rate, $|f(x)| \leq |x|^r + r$ for some constant r, hence $m^* = r \cdot n + r$ suffices. (This can all be done deterministically in $O(\log n)$ time, see Lemma 13 of §7.) Then N marks off $c \cdot \log(m^*) + c$ blank tape cells delimited by \$'s (say). The tape head is positioned at the left end of the block of blank tape cells. This tape is called the *clock tape* and is used to limit the simulation of M.

(2) Then N simulates M with input $f(x)$ as follows (the simulation algorithm is hardwired of course): For moves of M which don't query the input tape, N merely directly simulates M. For moves of M which do query the input tape, N existentially guesses the input symbol and then branches universally to either (a) accept the guess as correct and continue the simulation of M or (b) challenge the guess and use the $ALOGTIME$ algorithm for A_f to compute the correct input symbol — after this challenge N halts in an accepting state if the guessed input symbol was correct or halts in a rejecting state otherwise.

(3) For each move of M which is simulated, N moves the clock tape head one square to the right. If during the simulation, M enters an accepting or a rejecting state then N halts and accepts or rejects, respectively. However, if the clock tape head reaches the delimiting \$ then M's time has expired and N halts in a rejecting state.

That completes the description of N. Note that without the use of the clock tape N might not always halt in $O(\log n)$ time since the simulation of M could guess contradictory input symbols which, if unchallenged, could cause M to loop. (However, we add parenthetically that there is a general way to add a clock to log time Turing machines.)

Q.E.D. Theorem 4.

Theorem 5 *Suppose $B \in \Sigma_k$-LOGTIME where $k \geq 1$.*

(a) *If there is a deterministic log time reduction from A to B then A is in Σ_{k+1}-LOGTIME.*

(b) *If $s \geq 1$ and there is a Σ_s-LOGTIME reduction from A to B then A is in Σ_{k+s}-LOGTIME.*

In particular, the log time hierarchy (LH) is closed under LH-reductions.

Proof: It suffices to prove (b) since (a) is a special case of (b) with $s = 1$. Suppose M is an alternating Turing machine which recognizes B in time $c \cdot \log(n) + c$ with k alternations. The Turing machine which N which recognizes A runs as follows:

(1) As in the previous proof, N begins by deterministically setting up the clock tape to limit the simulation of M to $c \cdot \log(m^*) + c$ moves.

(2) Before beginning the simulation of M, N predetermines the nondeterministic choices to be made while simulating M. It does this by alternating k times to choose k blocks of $c \cdot \log(m^*) + c$ bits. The i-th bit of the j-th block determines the i-th nondeterministic choice made during the j-th alternation of M — so the simulation of M is now completely deterministic.

(3) Then N guesses existentially a string of $c \cdot \log(m^*) + c$ input symbols and universally chooses $c \cdot \log(m^*) + c$ *challenge* bits. The string of input symbols is to be the appropriate symbols which M should read from its input tape during the simulation and the challenge bits specify whether the input symbols are correct. A challenge bit equal to 1 indicates that N will halt the simulation of M when the corresponding input symbol would be used and will then use a Π_s-*LOGTIME* algorithm for A_f to verify the correctness of the guessed input symbol.

(4) Finally, N deterministically simulates M until either M halts, a challenge bit is on, or the clock tape indicates that time has expired.

That completes the description of N. If k is odd, the algorithm of N is a Σ_{k+s}-$LOGTIME$ algorithm. If k is even, it would be a Σ_{k+s+1}-$LOGTIME$ algorithm; but this case can be improved by modifying step (3) to universally choose the input symbols and existentially choose the challenge bits and then later use a Σ_s-$LOGTIME$ algorithm for A_f to verify the incorrectness of the universally chosen input symbol. That gives a Σ_{k+s}-$LOGTIME$ algorithm when k is even.

Q.E.D. Theorem 5.

There are several different natural ways to characterize a decision problem as complete for alternating log time. The key issue is what type of reduction between decision problems should be used. The main choices are (a) LH-reductions, (b) Σ_k-$LOGTIME$ reductions for a fixed $k \geq 1$ and (c) deterministic log time reductions. Probably the most natural choice is to use LH-reductions partly since the property of LH-reducibility is transitive and partly since an LH-reduction is essentially a uniform version of an AC^0 reduction. To see this, recall that an AC^0 reduction is one given by a family of constant depth, polynomial size circuits with unbounded fanin (see Cook [5]). Note that a Σ_1-$LOGTIME$ predicate can be expressed as a depth two, polynomial size circuit with unbounded fanin as follows: the circuit is a disjunction of clauses, one clause for each potential accepting computation path of the Σ_1-$LOGTIME$ Turing machine; each clause is the conjunction of literals corresponding the input values checked on that computation path. It is easy to extend this to show that a Σ_k-$LOGTIME$ predicate can be expressed as a depth $k+1$, polynomial size circuit with unbounded fanin.

Another but less natural choice of reducibility is Σ_1-$LOGTIME$ reducibility. However, we shall use the yet stronger property (c) of deterministic log time reducibility. Although it is not transitive, it is perhaps the strongest possible reasonable notion of reducibility. Thus we establish below that the various forms of the Boolean formula value problem are complete for $ALOGTIME$ under deterministic log time reductions. Of course this implies that they are also complete under AC^0 reductions.

4. Translation of Boolean formulas to $PLOF$ formulas

As a first step towards proving the Main Theorem we give an alternating log time reduction from the Boolean formula value problem to the $PLOF$ formula value problem. That is to say, there is an $f\colon \Sigma^* \to \Sigma^*$ so that for every Boolean formula A, $f(A)$ is an equivalent $PLOF$ formula and so that the predicate

$$A_f(c,i,A) \iff \text{“}c \text{ is the } i\text{-th symbol in } f(A)\text{”}.$$

is in ALOGTIME. Of course $A_f(c,i,A)$ is false if $|f(A)| < i$; it will always be the case that $|f(A)| \leq |A|$; in fact, $|f(A)|$ will be equal to the total number of symbols other than parentheses in A.

The definition of f and A_f is very simple, just let

$$A_f(c,i,A) \iff \text{“there is an } \alpha_j\text{, the } j\text{-th symbol of } A\text{, such that } \alpha_j \text{ is the symbol } c \text{ and there are } i-1 \text{ symbols } \alpha_k \text{ such that } Before(k,j,A)\text{”}.$$

It is clear that A_f is in $ALOGTIME$ since the α_j is unique and may be existentially guessed, since counting is in $ALOGTIME$, and since the predicate $Before(k,j,A)$ is in $ALOGTIME$. It is immediate from inspection that when A is a Boolean formula then $f(A)$ is an equivalent $PLOF$ formula and hence A and $f(A)$ have the same truth value.

N. Lynch [11] showed that Boolean formulas can be translated to postfix notation and vice-versa by log space algorithms. It is clear that the function f can be modified to give an alternating log time algorithm for Lynch's translation of Boolean formulas to postfix notation. Also, her map from postfix notation to infix notation is in alternating log time; again, the crucial idea is the use of counting — the details are left to the reader.

5. The algorithm for the $PLOF$ formula value problem

Since there is an alternating log time algorithm which translates a Boolean formula into an equivalent $PLOF$ formula, it will suffice to prove the next theorem in order to prove Main Theorem 1.

Theorem 6 *There is an alternating log time algorithm for determining the truth value of a variable-free $PLOF$ formula.*

The input to the alternating log time algorithm is $PLOF$ formula A which is a string of symbols, $A = \alpha_1 \cdots \alpha_n$ from the alphabet $\{0, 1, \neg, \vee, \wedge\}$. Hence the length $|A|$ of A is n. The algorithm will be described in terms of a game Γ_n between the `Affirmer` and the `Denier`.

Definition: Let A be as above and suppose $1 \le j \le k \le n$. Then $A[j,k]$ is the string $\alpha_j \alpha_{j+1} \cdots \alpha_k$. The *subformulas* of A are the $PLOF$ formulas of the form $A[j,k]$.

For $1 \le k \le n$, A_k is the unique subformula of A of the form $A[j,k]$ for some j. The subformula A_k *contains* the symbol α_i if and only if $i \le k$ and $|A_k| > k - i$. In other words, A_k contains α_i if and only if A_k is $A[j,k]$ and $j \le i \le k$.

Definition: Let A be as above and $1 \le i \le j \le n$. If $i < j$, then k is 1-*selected by* (i,j) (*in* A) if and only if k is the largest number $\le j$ such that A_k contains α_{i+1}. Otherwise, in the degenerate case $i = j$, we define k to be 1-selected by (i,j) if and only if $k = j$.

More generally, k is $(s+1)$-*selected by* (i,j) if and only if k is 1-selected by (k_0, j) where k_0 is s-selected by (i,j).

The game Γ_n is played by the `Affirmer` who wishes to prove A has Boolean value 1 (or *True*) and the `Denier` who wishes to prove A has value 0. In each move both players assert the truth value of a subformula of A. The game Γ_n

is "oblivious" of A in that the choice of subformula under consideration in a given move is independent of A *insofar as is possible*; thus the game is called Γ_n instead of Γ_A. The obliviousness is obtained by having a range (L_i, R_i) in round i of the game — the players must assert the truth value of the subformula A_{k_i} where k_i is s_i-selected by (L_i, R_i). The numbers L_i, R_i, and s_i depend only on i, n and the prior answers of the players during the game.

Since this "obliviousness" is one of the crucial ideas in the definition of the alternating log time algorithm, we discuss it further. If we try to base an algorithm on either Spira's or Brent's algorithm for restructuring Boolean formulas, the following problem arises: after the `Affirmer` and `Denier` play a log time game giving the truth values to subformulas which arise from Spira/Brent restructuring, it is then apparently impossible to use an alternating log time algorithm to determine who won the game. The reason for this is that there is no known alternating log time algorithm for determining which subformula was being considered at the i-th move of the game, since the subformula at move i depends on the earlier subformulas which likewise depend on their earlier subformulas, etc. This would give rise to an alternating $O(\log^2 n)$-time algorithm for finding the i-th subformula. Actually, Ramachandran [14] improved upon this by showing that Brent restructuring has an alternating $O(\log n \log\log n)$-time algorithm and Cook and Gupta [7] obtained the same result for a variant of Spira's algorithm.

Our method is to transform the Boolean formula into $PLOF$ formula and then, instead of using Spira or Brent restructuring, use a different method of dividing the formula in half, so that a "divide-and-conquer" algorithm will work. The crucial point is that after the `Affirmer` and `Denier` finish the game there will be an alternating log time algorithm which can determine for any i what subformula was considered during the i-th move and hence can determine who won the game. The idea of transforming the Boolean formula to a $PLOF$ formula was foreshadowed by Lynch's log space algorithm [11] which evaluates longer subformulas first.

The moves of the game Γ_n are grouped into triples — the j-th triple consists of moves numbered $3j-2$, $3j-1$ and $3j$. As mentioned above, each move i has associated with it numbers L_i, R_i, s_i and k_i; in addition there are numbers l_i, r_i, l_i^A and r_i^A. We call (l_i, r_i) the *global range*; it is the "publicly acknowledged" range of disagreement between the `Affirmer` and `Denier`. The A-*specific range* (l_i^A, r_i^A) is the actual range of disagreement. The number s_i is always either 1 or 2. The ranges are defined so that

$$l_i \leq l_i^A < r_i^A \leq r_i$$

is always guaranteed to hold during any (unfinished) play of Γ_n for any $PLOF$ formula A.

The only exception to the moves being grouped in triples is the very first move numbered 0. The initial move has paramenters $L_0 = l_0 = l_0^A = 0$ and $R_0 = r_0 = r_0^A = n$. The `Affirmer` must first answer 1 and the `Denier` must answer 0 or they forfeit the game. This initial move is included only to make the description of the game more uniform.

The A-specific range (l_i^A, r_i^A) is set so that the `Affirmer` and `Denier` have disagreed on the truth value of the subformula $A_{r_i^A}$ but have agreed on the truth values of subformulas of A which together contain each symbol α_m with $m \leq l_i^A$. (This latter fact will require proof.) Each triple of moves halves the global range (l_i, r_i); more precisely, for all $i = 3j - 2$,

$$l_i - r_i = \left\lfloor \frac{n}{2^{j-1}} \right\rfloor .$$

For notational convenience, set

$$Half_j = \left\lfloor \frac{n}{2^{j-1}} \right\rfloor - \left\lfloor \frac{n}{2^j} \right\rfloor .$$

and

$$SQuart_j = \left\lfloor \frac{1}{2} Half_j \right\rfloor$$

$$LQuart_j = \left\lceil \frac{1}{2} Half_j \right\rceil .$$

Thus $SQuart_j + LQuart_j = Half_j$; $SQuart$ and $LQuart$ stand for "small quarter" and "large quarter".

Before telling what the values for the parameters l_i, r_i, s_i, L_i, R_i, l_i^A and r_i^A are during the game, let us first explain what the players `Affirmer` and `Denier` are required to do. The subformula under consideration is A_{k_i} where k_i is s_i-selected by (L_i, R_i) — each player plays a single bit 0 or 1 for *False* or *True*, respectively, according to:

Rule (★): First the `Affirmer` asserts a value for the subformula A_{k_i}. Then depending on whether k_i is in the A-specific range, the `Denier` plays according to one of the following three cases:

(a) If $l_i^A < k_i < r_i^A$ then the `Denier` also asserts a value for the subformula A_{k_i}.

(b) If $k_i \leq l_i^A$ then the `Denier` agrees with the `Affirmer` by giving the same answer.

(c) If $r_i^A \leq k_i$ then the `Denier` disagrees with the `Affirmer`.

For the next round the A-specific range is set as follows:

- If the `Affirmer` and `Denier` agreed, set $l_{i+1}^A = \max(l_i^A, k_i)$ and $r_{i+1}^A = r_i^A$.
- If they disagreed, set $l_{i+1}^A = l_i^A$ and $r_{i+1}^A = \min(r_i^A, k_i)$.

That completes the description of Rule (★) and we are now ready to describe the details of the game Γ_n. Move (0) for Γ_n has already been described. For move (1) the global and A-specific ranges are set by $l_i = l_i^A = 0$ and $r_i = r_i^A = n$. The parameters for moves $(3j-2)$, $(3j-1)$ and $(3j)$ are set as follows: (for ease in subscripting let $m = 3j - 2$)

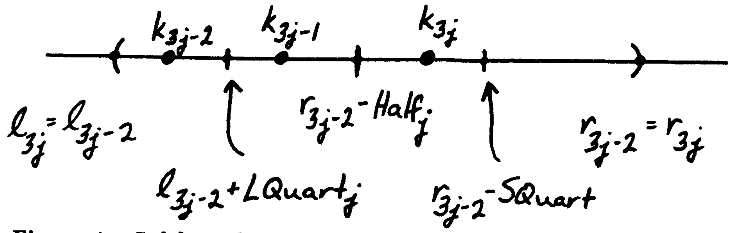

Figure 1: Subformulas selected during a triple of moves. (Drawn with correct relative magnitudes except k_{3j-2} may be larger than $l_{3j-2} + LQuart_j$.)

Set $l_{3j} = l_{3j-1} = l_{3j-2}$ and $r_{3j} = r_{3j-1} = r_{3j-2}$.

Move $(3j-2)$**:** For $i = 3j-2$, let $s_i = 1$, $L_i = l_m$ and $R_i = r_m - Half_j$. So k_i is 1-selected by the range $(l_i, r_i - Half_j)$. The players answer in this and the successive moves according to Rule ($\bigstar$).

Move $(3j-1)$**:** Now let $i = 3j-1$. For this move set $s_i = 2$, $L_i = l_m$, and $R_i = r_m - Half_j$ (so only the value of s_i changes from the previous move).

Move $(3j)$**:** Now let $i = 3j$. Set $s_i = 1$, $L_i = r_m - Half_j - 1$ and $R_i = r_m - SQuart_j$.

After move $(3j)$ set l_{3j+1} and r_{3j+1} as follows:

(a) If the `Affirmer` and `Denier` disagreed on either move $(3j-2)$ or $(3j-1)$, set $l_{3j+1} = l_m$ and $r_{3j+1} = r_m - Half_j$. (Note that move $(3j)$ is ignored in this case.)

(b) If they agreed on both moves $(3j-2)$ and $(3j-1)$ but disagreed on move $(3j)$, then set $l_{3j+1} = l_m + LQuart_j$ and $r_{3j+1} = r_m - SQuart_j$.

(c) If they agreed on all three moves, set $l_{3j+1} = l_m + Half_j$ and $r_{3j+1} = r_m$.

That finishes the description of how the game Γ_n is played – the game ends when $l_i + 1 = r_i$ which happens after $O(\log n)$ moves, in fact in $3\lfloor \log_2 n \rfloor + 1$ moves. (The 3 is because moves come in triples and the $+1$ is for the first move.) Thus there are exactly $6\lfloor \log_2 n \rfloor + 2$ bits played by the `Affirmer` and `Denier` combined.

The alternating log time algorithm for the *PLOF* formula value problem runs as follows: first $6\lfloor \log_2 n \rfloor + 2$ bits are alternately existentially and universally generated — these bits are the players' moves in the game Γ_n. Second, the algorithm determines who won the game. The `Affirmer` will lose the game unless she has forced the `Denier` into an incorrect, contradictory, or blatantly false answer. There are four ways in which the `Denier` could lose the game:

(1) If during some move, part (b) or (c) of Rule (★) applied but the `Denier` did not properly agree or disagree with the `Affirmer`.

(2) If during some move, part (a) of Rule (★) applied and the selected subformula A_{k_i} is a literal 0 or 1 but the `Denier` asserted that it had value 1 or 0, respectively.

(3) If there are two subformulas A_{k_i} and A_{k_m} such that A_{k_m} is $A_{k_i}\neg$ (so $k_m = k_i + 1$) and such that during moves m and i, part (a) of Rule (★) applied and the `Denier` answered identically for the truth values of A_{k_i} and A_{k_m} in moves i and m.

(d) Similarly for $\wedge$ or $\vee$; there are three subformulas A_{k_i}, A_{k_m} and A_{k_s} such that A_{k_m} and A_{k_s} are the direct subformulas of A_{k_i} but the `Denier` gave answers during moves i, k and s which contradict the definition of the operators $\vee$ and $\wedge$ (and part (a) of Rule (★) applied for moves i, m and s).

There is an alternating log time algorithm which determines if one of these four conditions holds; to see this, we need the following $ALOGTIME$ subroutines:

(i) Compute $Half_j$, $SQuart_j$ and $LQuart_j$ as a function of j and A: this is clearly in alternating log time.

(ii) Compute l_{3j-2} and r_{3j-2} as a function of j, A and the plays by the `Affirmer` and `Denier` during the game: For each triple of moves the global range is changed so that

$$l_{3s+1} = l_{3s-2} + \gamma_s$$

$$r_{3s+1} = r_{3s-2} - \delta_s$$

where γ_s is 0 or $LQuart_s$ or $Half_s$ and where δ_s is 0 or $SQuart_s$ or $Half_s$. Clearly γ_s and δ_s are easily computed from the players' moves in Γ_n. (Note: this is where we are using the "obliviousness" of Γ_n.) So l_{3j-2} and r_{3j-2} can be computed by

$$l_{3j-2} = \sum_{1\le s<j} \gamma_s,$$

$$r_{3j-2} = n - \sum_{1\le s<j} \delta_s$$

by the beforementioned alternating log time algorithm for the summation of a vector of integers.

(iii) Compute s_i, k_i, L_i and R_i as a function of j, A and the plays of the `Affirmer` and `Denier` during the game: The values of s_i, L_i and R_i are trivially computed in alternating log time by using *(i)* and *(ii)*. Then k_i can be computed by using the parsing techniques discussed in §2.

(iv) Determine if part (a), (b) or (c) of Rule (★) applied in move i of Γ_n — as a function of i, A and the plays of the `Affirmer` and `Denier` during the game: Check (in alternating log time) the following two conditions

(β) Is there an $s < i$ such that $k_s \geq k_i$ and such that the `Affirmer` and `Denier` agreed in move (s)?

(γ) Is there an $s < i$ such that $k_s \leq k_i$ and such that the `Affirmer` and `Denier` disagreed in move (s)?

If only (β) holds then part (b) of Rule (★) applied during move i, if only (γ) holds then part (c) applied, if neither holds then part (a) applied, and finally, if both hold then the Rule (★) had already been violated earlier in the game and hence the winner (namely the `Affirmer`) had already been determined before move i.

That completes the proof that the winner of the game can be determined in alternating log time. In order to finish the proof of Theorem 6 and Main Thoerem 1 we must show that the `Affirmer` has a winning strategy if and only if the Boolean formula A is true (has value 1). This will be clear provided we can prove that during any game in which the `Denier` has obeyed Rule (★), the following two conditions hold: (1) for any move i and any symbol α_s with $s \leq l_i^A$, the symbol α_k occurs in some previously selected subformula A_{k_m} with $m < i$; and (2) the A-specific range is included in the global range, i.e.,

$$l_i \leq l_i^A < r_i^A \leq r_i.$$

These follow readily from the next lemma:

Lemma 7 *In the definition of moves $(3j-2)$, $(3j-1)$ and $(3j)$, it is always the case that*

(a) $l_{3j-2} < k_{3j-2} \leq r_{3j-2} - Half_j$,

(b) $l_{3j-2} + LQuart_j \leq k_{3j-1} \leq r_{3j-2} - Half_j \leq r_{3j-2} - SQuart_j$,

(c) $r_{3j-2} - Half_j \leq k_{3j} \leq r_{3j-2} - SQuart_j \leq r_{3j-2}$,

(d) *For every t such that $l_{3j-2} < t \leq l_{3j-2} + LQuart_j$, the symbol α_t is in one of the subformulas $A_{k_{3j-2}}$ or $A_{k_{3k-1}}$. For every t such that $l_{3j-2} < t \leq l_{3j-2} + Half_j$, the symbol α_t is in one of the subformulas $A_{k_{3j-2}}$, $A_{k_{3j-1}}$ or $A_{k_{3j}}$.*

Proof: (Refer to Figure 1 above.)

(a) It is obvious that $l_{3j-2} < k_{3j-2} \leq r_{3j-2} - Half_j$ by the definition on k_{3j-2}.

(b) From the definitions of k_{3j-2} and k_{3j-1} we know that $A_{k_{3j-1}}$ is the subformula $A[k_{3j-2}+1, k_{3k-1}]$. Suppose $k_{3j-1} < r_{3j-2} - Half_j - 1$ (otherwise the entire lemma is trivially true). Clearly, $\alpha_{1+k_{3j-1}}$ can not be a negation sign ($\neg$) since this would contradict the fact that k_{3j-1} was 1-selected by $(k_{3j-2}, r_{3j-2} - Half_j)$. Hence $A_{k_{3j-1}}$ is an operand of a binary operator α_s and indeed it is the first operand. The latter statement is shown by noting that $A_{k_{3j-1}}$ being the second operand would violate the definition of k_{3j-2}, since a better choice for the value of k_{3j-2} would have been $k_{3k-1} + 1$. Since A is a $PLOF$, the first operand of α_s is at least as long as its second operand; namely,

$$(s-1) - k_{3j-1} \leq k_{3j-1} - k_{3j-2}.$$

Also, $s > r_{3j-2} - Half_j$ by the definition of k_{3j-1}. So

$$\begin{aligned} 2k_{3j-1} &\geq (s-1) + k_{3j-2} \\ &> (r_{3j-2} - Half_j - 1) + (l_{3j-2} + 1). \end{aligned}$$

Hence

$$\begin{aligned} 2(k_{3j-1} - l_{3j-2}) &> (r_{3j-2} - l_{3j-2}) - Half_j \\ &= \left\lfloor \frac{n}{2^{j-1}} \right\rfloor - \left(\left\lfloor \frac{n}{2^{j-1}} \right\rfloor - \left\lfloor \frac{n}{2^j} \right\rfloor \right) \\ &= \left\lfloor \frac{n}{2^j} \right\rfloor. \end{aligned}$$

So, $k_{3j-1} \geq l_{3j-2} + LQuart_j$.

Part (c) of the lemma is obvious from the choice of k_{3j}.

It follows by the discussion regarding part (b) that the first claim of part (d) holds. Let u be 1-selected by $(k_{3j-1}, r_{3j-2} - SQuart_j)$; we claim that $u \geq l_{3j-2} + Half_j$. This claim implies that $u = k_{3j}$ and that the second part of (d) holds. The proof of the claim is similar to the proof in (b) above that $k_{3j-1} > l_{3j-2} + LQuart_j$ and goes as follows: If $u < r_{3j-2} - SQuart_j$ then the subformula A_u must be the first and longer operand of a binary operator α_v with $v > r_{3j-2} - SQuart_j$. Hence,

$$(v-1) - u \leq u - k_{3j-1}$$

and thus

$$\begin{aligned} 2u &\geq v - 1 + k_{3j-1} \\ &\geq (r_{3j-2} - SQuart_j) + (l_{3j-2} + LQuart_j). \end{aligned}$$

From whence, $u \geq l_{3j-2} + Half_j$ follows easily.

Q.E.D. Lemma 7, Theorem 6 and Main Theorem 1.

We have now completed the proof that the Boolean formula value problem has an alternating log time algorithm. It should be remarked that some authors formulate the Boolean formula value problem slightly differently: they allow variables to appear in the Boolean formula and then the Boolean formula value problem has as inputs a Boolean formula plus a truth assignment to the variables. It is easy to see that there is an alternating log time reduction from this alternative formulation to our formulation. Hence there is an alternating log time algorithm for this alternative formulation.

However, there is another radically different way of formulating the Boolean formula value problem; namely as a table of nodes with pointers from a node to its parent and children. In this formulation the formula value problem is equivalent to the reachability problem and hence is logspace complete — this was shown by Dowd and Statman [6] and independently by Tompa.

6. Parenthesis Languages are in *ALOGTIME*

Parenthesis languages, first studied by McNaughton [12], are context-free grammars of the form $(\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$ which have two distinguished terminal symbols "(" and ")" in $\mathcal{T}$; the variable $S \in \mathcal{V}$ is the start symbol, $\mathcal{V}$ is the set of variables and every production in $\mathcal{P}$ is of the form

$$A \to (x)$$

where $A \in \mathcal{V}$, $x \in (\mathcal{V} \cup \mathcal{T})^*$ and x contains no parentheses. (The sets $\mathcal{V}$, $\mathcal{T}$ and $\mathcal{P}$ are finite and disjoint.)

As an example, the set of Boolean formulas with value 1 (*True*) is a parenthesis language with $\mathcal{V} = \{T, F\}$, $S = T$, $\mathcal{T} = \Sigma$ and the productions $P \in \mathcal{P}$ being:

$$\begin{array}{ll}
T \to 1 & T \to (T \wedge T) \\
F \to 0 & F \to (T \wedge F) \\
T \to (\neg F) & F \to (F \wedge T) \\
F \to (\neg T) & F \to (F \wedge F) \\
T \to (T \vee T) & T \to (F \vee T) \\
T \to (T \vee F) & F \to (F \vee F)
\end{array}$$

Lynch [11] showed that every parenthesis context-free language is in log space and Cook and Gupta [7] showed that parenthesis context-free languages have log space uniform, polynomial size circuits of depth $O(\log n \log \log n)$ — we improve this to:

Main Theorem 8 *Every parenthesis context-free language is in alternating log time.*

Proof: (Outline) We shall sketch briefly how the proof of Main Theorem 1 can be extended to handle a general parenthesis language. Let $G = (\mathcal{V}, \mathcal{T}, \mathcal{P}, S)$ be a parenthesis language. For $A \in \mathcal{V}$, let G_A be the parenthesis language $(\mathcal{V}, \mathcal{T}, \mathcal{P}, A)$, i.e., G_A is the language generated by starting with the variable A. We define the *G-formulas* to be the strings of symbols which are members of some G_A; note that G-formulas are not necessarily members of the parenthesis language G.

We shall shift our viewpoint from the recognition of members of G to the evaluation of G-formulas (this technique is due to McNaughton). The value of a G-formula will be a member of $\wp(\mathcal{V})$, the power set of $\mathcal{V}$ (so there is only a finite number of possible values). Given any string $\alpha \in (\mathcal{V} \cup \mathcal{T})^*$, the *value* of α is the set $\{A : \alpha \in G_A\}$. Thus to determine if $\alpha \in G$ one computes the value of α and checks if it contains the symbol S as a member.

Each production $P \in \mathcal{P}$ is of the form

$$A_0 \to (\alpha_1 A_1 \alpha_2 A_2 \cdots \alpha_n A_n \alpha_{n+1})$$

where each $A_i \in \mathcal{V}$ and each α_i is in $\mathcal{T}^*$ and is parenthesis-free. Let $\mathbb{A}$ denote the set of sequences $\langle \alpha_1, \ldots, \alpha_{n+1} \rangle$ obtained in this way from a $P \in \mathcal{P}$. We define *Postfix-Longer-Operands-First* G-formulas, or, for short, $PLOF_G$ formulas as follows: for each $(n+1)$-ary sequence $s \in \mathbb{A}$ and each permutation σ on n elements, s_σ is an n-ary function $s_\sigma : \wp(\mathcal{V})^n \to \wp(\mathcal{V})$, so that if $Y_1, \ldots, Y_n \subset \mathcal{V}$ then $s_\sigma(Y_1, \ldots, Y_n)$ is the set Y_0 such that for all $A \in \mathcal{V}$, $A \in Y_0$ if and only if there is a production $P \in \mathcal{P}$ of the form

$$A \to (\alpha_1 A_1 \alpha_2 \cdots \alpha_n A_n \alpha_{n+1})$$

where $A_i \in Y_{\sigma(i)}$ for $i = 1, 2, \ldots n$ and where s is $\langle \alpha_1, \alpha_2, \ldots, \alpha_{n+1} \rangle$. Now the $PLOF_G$ formulas are formulas which use the operators s_σ in postfix (reverse Polish) notation with operands of each s_σ required to be in non-increasing order of length.

Because of the use of the permutations σ it is clear that for every G-formula there is an equivalent $PLOF_G$ formula. It should also be clear that there is an alternating log time algorithm for converting a potential member of G into an equivalent $PLOF_G$ formula. (If α can not be converted then either the parentheses in α are not balanced or α has a substring of the form $(\beta_1(\gamma_1)\beta_2 \cdots \beta_n(\gamma_n)\beta_{n+1})$ where the β_i's contain no parentheses and the sequence $\langle \beta_1, \ldots, \beta_{n+1} \rangle$ is not in $\mathbb{A}$. These conditions can be recognized in alternating log time and imply that α is not in G.)

The evaluation of a $PLOF_G$ formula is again described in terms of a game between the `Affirmer` and `Denier`. Now the players must specify a subset of $\mathcal{V}$ as a value for each subformula instead of Boolean values 0 and 1. The game and the determination of the winner proceed much as in the game Γ_n described above but with an important modification: the triples of moves are replaced by $(k+1)$-tuples of moves where k is maximum arity of the function symbols s_σ. If global range (l_i, r_i) has been set for $i = (k+1)j - k$, then set $L = l_i$ and $R = r_i - Half_j$. The subformulas selected during the next $k+1$ moves are as follows:

In move $i = (k+1)j - k$, k_i is 1-selected by (L, R).

In move $i = (k+1)j - (k-m)$, k_i is $(m+1)$-selected by (L, R) (for $0 \leq m < k$).

In move $i = (k+1)j$, k_i is 1-selected by $(R-1, r_{(k+1)j-k} - SQuart_j)$.

As before, $r_i - l_i = \left\lfloor \frac{n}{2^{j-1}} \right\rfloor$ when $i = (k+1)j - k$.

The reader may check that all the details work out.

Q.E.D. Main Theorem 8.

7. Complete Problems for *ALOGTIME*

The results obtained above are the best possible since we can prove the Boolean formula value problem and the *PLOF* formula value problem are alternating log time complete.

Main Theorem 9

(a) *The Boolean formula value problem is ALOGTIME-complete under deterministic log time reductions.*

(b) *The PLOF formula value problem is ALOGTIME-complete under deterministic log time reductions.*

(c) *The postfix (reverse Polish) Boolean formula value problem is ALOGTIME-complete under deterministic log time reductions.*

Corollary 10 *The Boolean formula value problem, the PLOF formula value problem and the postfix notation Boolean formula value problem are each complete for ALOGTIME under* AC^0 *reductions.*

Corollary 11 *There is a deterministic log time reduction from the (infix notation) Boolean formula value problem to the postfix notation Boolean formula value problem, and vice-versa.*

Corollary 11 is rather surprising at first glance; however, it should be noted that it does not assert that the natural translation of infix formulas to postfix formulas is in deterministic log time. Indeed, the proofs of Main Theorem 1 and Main Theorem 9 give a fairly complicated translation.

A. Yao [18] and J. Hastad [8] prove that there is an oracle which separates the polynomial time hierarchy; this immediately implies that the (unrelativized) log time hierarchy is proper. Hence the computational complexity of the Boolean formula value problem can be precisely characterized by:

Corollary 12 *The Boolean formula value problem is in ALOGTIME but not in the log time hierarchy.*

The general idea of the proof of Theorem 9(a) is as follows. Let A be a predicate in *ALOGTIME* and M be an alternating Turing machine which recognizes A such that M always halts in time $c \cdot \log n + c$ on inputs of length n. We shall define below a deterministic log time function f such that for any input x to M, $f(x)$ is a variable-free Boolean formula which evaluates to *True* if and only if M accepts x, i.e., $x \in A$. The Boolean formula $f(x)$ will essentially be the execution tree of M on input x; the $\vee$'s and $\wedge$'s in $f(x)$ correspond to existential and universal configurations of M, respectively, and the 0's and 1's correspond to rejecting and accepting configurations of M. The difficult part of the proof of Theorem 9(a) is showing that f can be picked so that A_f is a deterministic log time predicate.

We assume without loss of generality that each configuration of M has at most two possible successor states. For each configuration s of M there are two successor configurations denoted $\ell(s)$ and $r(s)$; the degenerate cases are $\ell(s) = r(s)$ when s is a deterministic configuration and $s = \ell(s) = r(s)$ when s is a halting configuration. More generally, for $\rho \in \{\ell, r\}^*$, we define $\rho(s)$ in the obvious way so that $0(s) = s$ and $\rho r(s) = r(\rho(s))$ and $\rho\ell(s) = \ell(\rho(s))$ where 0 is the empty word. Let $I(x)$ denote the initial configuration of M on input x. We define Boolean formulas $\alpha(\rho, x)$ where $|\rho| \le c \cdot \log(|x|) + c$ as follows:

Case (1): If $|\rho| = c \cdot \log(|x|) + c$ then define $\alpha(\rho, x)$ to be the literal 1 if $\rho(I(x))$ is an accepting configuration of M and to be the literal 0 otherwise.

Case (2): If $|\rho| < c \cdot \log(|x|) + c$ then set $\phi_\ell = \alpha(\rho\ell, x)$ and $\phi_r = \alpha(\rho r, x)$ and define $\alpha(\rho, x)$ to be the formula $(\phi_\ell \wedge \phi_r)$ if $\rho(I(x))$ is a universally branching configuration of M and to be $(\phi_\ell \vee \phi_r)$ otherwise.

The function f is defined by $f(x) = \alpha(0, x)$.

It is clear that $f(x)$ is a variable free Boolean formula which has value 1 (i.e., *True*) if and only if $x \in A$. To prove Theorem 9(a) it remains to show that f is a deterministic log time function; or equivalently, that there is a deterministic log time Turing machine N which on input x and i produces as output the i-th symbol of $f(x)$. Basically what N needs to do is determine the location of the i-th symbol of $f(x)$ by determining the string $\rho \in \{\ell, r\}^*$ of maximum length such that the i-th symbol of $f(x)$ is in the subformula $\alpha(\rho, x)$ of $f(x)$. After determining this ρ, N simulates M to determine the corresponding configuration $\rho(I(x))$ of M on input x and from this obtains the correct i-th symbol of $f(x)$.

Lemma 13 (Dowd [6]) *There is a deterministic log time Turing machine which, on input x, outputs the value $n = |x|$ coded in binary.*

Proof: The operation of the Turing machine proceeds as follows. First determine the least value of i such that $n < 2^i$: this is easily done in $O(\log n)$ time since we assume that the index tape and its tape head are unaffected when the input tape is accessed. Once the value 2^i is obtained, written on the input tape, it is now easy to do a binary search to determine the value of n. Finally the index tape is copied to the output tape. □

Lemma 14 *If $\rho \in \{\ell, r\}^*$ then $|\alpha(\rho, x)| = 2^{s+2} - 3$ where $s = c \cdot \log(|x|) + c - |\rho|$.*

Proof: This is easily shown by induction on s. □

We next give the algorithm for N which computes the i-th symbol of $f(x)$. The naive method of executing this algorithm will not be $O(\log n)$ time so we shall later indicate how to improve its execution.

Input: x, i

Step (1): Compute $n = |x|$.

Step (2): Compute $d = c \cdot \log n + c$. (This is easy since our logarithms are base two.)

Step (3): If $i \geq 2^{d+2} - 3$, output a blank symbol and halt.

Step (4): Set ρ equal to the empty word.
Set s equal to $d + 2$.
Set j equal to i.

Step (5): (Loop while $s \geq 2$)
Select one case (exactly one must hold):

- Case (5a): If $j = 0$, output "(" and halt.
- Case (5b): If $0 < j < 2^{s-1} - 2$, set $j = j - 1$ and set $\rho = \rho\ell$.
- Case (5c): If $j = 2^{s-1} - 2$, exit to step (6).
- Case (5d): If $2^{s-1} - 2 < j < 2^s - 4$, set $j = j - (2^{s-1} - 2)$ and set $\rho = \rho r$.
- Case (5e): If $j = 2^s - 4$, output ")" and halt.

Set $s = s - 1$.
If $s \geq 2$, reiterate step (5) otherwise exit to step (6).

Step (6): Simulate M for $|\rho|$ steps to determine the configuration $\rho(I(x))$.
If $|\rho| < d$ and $\rho(I(x))$ is a universally branching state, output "$\wedge$".
Otherwise, if $|\rho| < d$, output "$\vee$".
Otherwise, if $\rho(I(x))$ is an accepting state, output "1".
Otherwise, output "0".

It should be clear by inspection that this algorithm correctly computes the i-th symbol of $f(x)$. In an iteration of the loop in step (5), it has already been ascertained that the i-th symbol of $f(x)$ is the j-th symbol of the subformula $\alpha(\rho, x)$. The subformula $\alpha(\rho, x)$ is of the form $(\phi_\ell * \phi_r)$ where $*$ is either $\vee$ or $\wedge$; the five cases correspond to the j-th symbol being (a) the initial parenthesis, (b) in the subformula ϕ_ℓ, (c) the logical connective symbol, (d) in the subformula ϕ_r or (e) the final parenthesis.

Except for step (5), each step in the algorithm takes $O(\log n)$ time. In particular, for step (6), the simulation of M is hardwired and N simulates each operation of M with only one operation. Step (5), however, is more difficult: there are $O(\log n)$ iterations of the loop and each iteration takes $O(\log n)$ time in a naive implementation — we need each iteration to take constant time.

The reason that each iteration takes $O(\log n)$ time is that in case (5d), for example, to subtract $2^{s-1} - 2$ from j both the high and low order bits of j must be modified; but j has $O(\log n)$ bits so it takes too much time just to move the tape head from one end of j to the other. Similar problems arise in comparing j to $2^{s-1} - 2$ and $2^s - 4$. Also, even when just decrementing j by 1 in case (5b) it may take $O(\log n)$ time to propagate a borrow.

Fortunately all these problems con be avoided by a simple trick. Before starting step (5), N breaks j into two parts: the low order $2 + \log d$ bits of j

are stored on a tape in *unary* notation; the remaining high order bits of j are kept on a different tape in binary notation. So to decrement j by 1, N merely changes one tape square on the unary tape and moves that tape head one square. To subtract $2^{s-1} - 2$ from j, N need only change two squares on the unary tape and modify one square of the binary tape (since $j \leq 2^s - 4$). A complication arises when there is a carry or borrow out of the $(2 + \log d)$-th bit position of j. N handles this by allowing the unary tape to overflow (and cause a carry) or underflow (and cause a borrow). To do this the unary tape is initialized with a marker indicating where the overflow or underflow occurs; since the unary part of j is changed by -1 or $+2$ at most $d = c \cdot \log n + c$ times, at most one marker is needed. During the iterations of the loop in step (5) N remembers whether or not an underflow/overflow has occurred. N also initializes the binary tape with a marker which indicates how far the borrow or carry will propagate.

We can now summarize how N executes step (5) in $O(\log n)$ time. First j is split into binary high-order and unary low-order parts — these are stored on separate tapes along with borrow/carry information. Then the loop is executed for $s = d+2$ to $s = 3+\log d$ maintaining the value of j in the split binary/unary form. After these iterations the higher-order, binary portion of j is equal to zero. The unary portion of j is now converted back to binary notation and the remaining iterations of the loop with $s = 2 + \log d$ to $s = 2$ are executed in the normal naive fashion with j in binary notation.

The above proves Theorem 9(a). Theorem 9(b) can be proved by a similar, somewhat easier, deterministic log time reduction of an arbitrary $ALOGTIME$ predicate to the $PLOF$ formula value problem. Theorem 9(c) is a consequence of 9(b).

8. Conclusion

We have shown that the Boolean formula value problem is complete for alternating log time under deterministic log time reductions. A variant of this problem, the postfix notation Boolean formula value problem is also alternating log time complete under deterministic log time reductions. This complements the result of Ladner [10] that the circuit value problem is complete for polynomial time under log space reductions. It is a longstanding open problem whether for every circuit there is an equivalent formula with the size of the formula bounded by a polynomial of the size of the circuit; we conclude that this unlikely to be the case unless $P = ALOGTIME$.

Acknowledgments

I have benefited greatly from discussions with Vijaya Ramachandran and Stephen Cook and from correspondence with Martin Dowd.

References

[1] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

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

[3] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[4] S. A. Cook, *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*, in Proceedings of the 11-th Annual ACM Symposium on Theory of Computing, 1979, pp. 338–345.

[5] ——, *A taxonomy of problems with fast parallel algorithms*, Information and Control, 64 (1985), pp. 2–22.

[6] M. Dowd, *Notes on log space representation.* typewritten manuscript, 1986.

[7] A. Gupta, *A fast parallel algorithm for recognition of parenthesis languages*, Master's thesis, University of Toronto, January 1985.

[8] J. Hastad, *Almost Optimal Lower Bounds for Small Depth Circuits*, vol. 5 of Advances in Computing Research, JAI Press, 1989, pp. 143–170.

[9] O. H. Ibarra, T. Jiang, and B. Ravikumar, *On some languages in* nc^1 *(summary).* manuscript, 1987.

[10] R. E. Ladner, *The circuit value problem is log space complete for P*, SIGACT News, 7 (1975), pp. 18–20.

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

[12] R. McNaughton, *Parenthesis grammars*, J. Assoc. Comput. Mach., 14 (1967), pp. 490–500.

[13] 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.

[14] V. Ramachandran, *Restructuring formula trees.* Unpublished manuscript, May 1986.

[15] W. L. Ruzzo, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[16] J. E. Savage, *The Complexity of Computing*, John Wiley, 1976.

[17] P. M. Spira, *On time hardware complexity tradeoffs for Boolean functions*, in Proceedings of the Fourth Hawaii International Symposium on System Sciences, 1971, pp. 525–527.

[18] A. C.-C. Yao, *Separating the polynomial time hierarchy by oracles*, in Proceedings of the 26th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1985, pp. 1–10.