PROBABILISTIC BOOLEAN DECISION TREES AND THE COMPLEXITY OF EVALUATING GAME TREES

Michael Saks
Bell Communications Research
Morristown, New Jersey
and
Department of Mathematics
Rutgers University
New Brunswick, New Jersey

Avi Wigderson
Mathematical Sciences Research Institute
Berkeley, California
and
Department of Computer Science
Hebrew University
Givat Ram, Jerusalem, Israel

Abstract

The Boolean Decision tree model is perhaps the simplest model that computes Boolean functions; it charges only for reading an input variable. We study the power of randomness (vs. both determinism and non-determinism) in this model, and prove separation results between the three complexity measures.

These results are obtained via general and efficient methods for computing upper and lower bounds on the probabilistic complexity of evaluating Boolean formulae in which every variable appears exactly once (AND/OR tree with distinct leaves). These bounds are shown to be exactly tight for interesting families of such tree functions.

We then apply our results to the complexity of evaluating game trees, which is a central problem in AI. These trees are similar to Boolean tree functions, except that input variables (leaves) may take values from a large set (of valuations to game positions) and the AND/OR nodes are replaced by MIN/MAX nodes. Here the cost is the number of positions (leaves) probed by the algorithm.

The best known algorithm for this problem is the alpha-beta pruning method. As a deterministic algorithm, it will in the worst case have to examine all positions. Many papers studied the expected behavior of alpha-beta pruning (on uniform trees) under the unreasonable assumption that position values are drawn independently from some distribution. We analyze a randomized variant of alpha-beta pruning, show that it is considerably faster than the deterministic one in worst case, and prove it optimal for uniform trees.

I. Introduction

In order to begin to understand the power and limitations of randomization in algorithms we investigate a simple model for the computation of Boolean functions that has been well studied in the deterministic case: *Boolean decision trees*. In this model, a basic step consists of reading (probing) the value of one of the input variables of the function, and the algorithm branches according to the observed value. The complexity of an algorithm is simply the number of variables that it evaluates in order to compute f on a worst case input; all other computation is free. The *deterministic complexity* $D(f)$ of a function $f(x_1,\ldots,x_n)$ is the minimum complexity of any deterministic decision tree algorithm that computes f. Clearly $D(f)$ is at most n, the number of variables.

In a *randomized Boolean decision tree* algorithm the algorithm is allowed to flip coins, and the choice of variable to evaluate at each stage may depend on these flips. The algorithm must always compute f correctly; we do not allow errors. The cost of such an algorithm on any fixed input is the expected number of variables that the algorithm reads while computing f on that input. The complexity of the algorithm is the maximum (expected) cost over all inputs. The *randomized complexity* $R(f)$ of the function f is the minimum complexity of any randomized algorithm that computes f. (This model resembles ones considered in [MT], [M], [S], but the problems addressed and techniques used are different.)

The deterministic Boolean decision tree model has been studied extensively in a number of contexts. One such is that of graph properties. Let P be a property that holds for some subset of graphs on v vertices and does not depend on the vertex

0272-5428/86/0000/0029$01.00 © 1986 IEEE

names (e.g. connectivity). We wish to determine whether a given (loop free) digraph G with v vertices and unknown edge set has property P by sequentially probing the entries of the adjacency matrix. P can be viewed as a Boolean function of the $n = v(v-1)$ nondiagonal entries of the adjacency matrix, and so the algorithms we are considering are precisely Boolean decision tree algorithms. The key result in this area is the following theorem of Rivest and Vuillemin [RV] (see also [B, Chapt. 8], [KK]), which settled a conjecture of Aanderaa and Rosenberg [R]:

Theorem 1.1. The deterministic complexity of any monotone nontrivial property on graphs of v vertices is $\Theta(v^2) = \Theta(n)$.

Here a property is nontrivial if at least one but not all graphs have it and monotone if it is preserved under the addition of edges. (In fact, it has been shown [KSS] that if v is a prime power then for such properties P, $D(P) = n$; whether this holds for general v is open).

It is natural to ask what happens to graph property complexity when we introduce randomization. For example, it is easy to show that the property "every vertex has an incoming arc" has deterministic complexity $n = v(v-1)$. On the other hand, the randomized algorithm that considers each vertex one at a time in random order and scans the edges into that vertex in random order until it finds some edge into that vertex (or finds that vertex has no edges) has expected cost at most $v(v+1)/2$ on any graph. Thus randomization can save a constant factor of the cost. What is not known is whether randomization can ever save more than this:

Conjecture 1.2 (Karp [K]). The randomized complexity of any monotone nontrivial property on graphs of v vertices is $\Theta(v^2) = \Theta(n)$.

At the moment this conjecture is wide open.

Returning to the case of general Boolean functions, the obvious question is: does randomization ever save more than a constant factor in the complexity of a Boolean function? The following two examples show that it does.

Example 1.1. Let $U(h)$ denote the uniform binary tree of height h. Consider the function F^h, with variable set corresponding to the $n = 2^h$ leaves, that is computed by interpreting $U(h)$ as a circuit with internal nodes representing NAND gates. (Up to complementation of the inputs this is equivalent to interpreting alternating levels of internal nodes as $\wedge$ and $\vee$.) It is easy to see that $D(F^h) = n$. Consider the randomized evaluation algorithm A_h defined recursively: choose a child of the root at random and evaluate its subtree recursively using A_{h-1}. If it evaluates to 0, then $F^h = 1$, otherwise recursively evaluate the other child of the root. Observe that if F^h is 1 then with probability at least $1/2$ we only evaluate one child of the root. Denoting by $\alpha_0(h)$ (resp. $\alpha_1(h)$) the maximum complexity of A_h on inputs that evaluate to 0 (resp. 1) we obtain the recurrence

$$\alpha_0(h) \leq 2\alpha_1(h-1)$$

$$\alpha_1(h) \leq \alpha_0(h-1) + \alpha_1(h-1)/2$$

Thus

$$R(F^h) = O\left(\left(\frac{1+\sqrt{33}}{4}\right)^h\right)$$

$$= O\left(n^{\log_2\left(\frac{1+\sqrt{33}}{4}\right)}\right)$$

$$= O(n^{.753\ldots}).$$

This example was given by Snir [S] who gave an $O\left(n^{\log_4 3}\right)$ upper bound and proved a linear lower bound in a deterministic model that is more powerful than ours.

Example 1.2 (due to Ravi Boppana). Let $U_3(h)$ be the uniform rooted ternary tree of height h and let G^h be the function with variable set corresponding to the $n = 3^h$ leaves, computed by inductively assigning to each internal node the majority of the values of its children. It is easy to show $D(G^h) = n$. Now consider the evaluation algorithm B_h that is recursively defined as follows: choose two children of the root at random and evaluate them recursively using B_{h-1}. If they agree in value then G^h has that value, otherwise recursively evaluate the remaining child. This leads to a recurrence

$$R(G^h) = \frac{8}{3} R(G^{h-1})$$

which yields

$$R(G^h) = (\frac{8}{3})^h = O(n^{.893\ldots}).$$

These two examples show that randomization can yield a substantial cost savings in the complexity of some functions and leaves open the question: how much smaller than $D(f)$ can $R(f)$ be?

To begin a discussion of lower bounds on $R(f)$, we introduce the notion of the nondeterministic complexity $N(f)$ of a Boolean function f. This is simply the number of variables that it may be necessary to evaluate to prove the value of f. It is easy to see that $N(f)$ is the maximum size of a clause occurring in either the DNF or CNF of f. Clearly we have

Proposition 1.3. For any Boolean function f, $R(f) \geq N(f)$

Now, an elegant argument (observed by Blum and others) sharply limits the gap between nondeterminism and determinism in this model:

Theorem 1.4. For any Boolean function f, $D(f) \leq N(f)^2$.

It follows that $R(f) \geq \sqrt{D(f)}$.

The functions F^h in example 1.1 have nondeterministic complexity $2^{h/2} = \sqrt{n}$ (for h even) and thus demonstrate that Theorem 1.4 is tight. For the functions G^h in example 1.2, the nondeterministic complexity is 2^h. In both cases the complexity achieved by the proposed randomized algorithms falls somewhere between the nondeterministic and deterministic complexity. However, this does not exclude the possibility that there are other randomized algorithms for these functions that achieve or nearly achieve the lower bound given by the nondeterministic complexity.

This brings us to the question: are the algorithms presented in examples 1.1 and 1.2 optimal? The following argument suggests that they are not. These algorithms have the property that whenever they begin to probe nodes in the subtree of a node they finish evaluating that node before examining any leaves not in its subtree. Such an algorithm fails to take advantage of the possibility of random sampling: it might be beneficial to sample a few nodes in different subtrees to decide which to evaluate first. It turns out that sampling of this type can be used to produce a better algorithm for evaluating G^h. On the other hand, a consequence of the main result of this paper is

Theorem 1.5. Algorithm A_h is optimal for computing F^h and thus

$$R(F^h) = \Theta\left(\left(\frac{1+\sqrt{33}}{4}\right)^h\right) = \Theta\left(D(F^h)^{.753\ldots}\right).$$

In the body of this paper, we investigate the randomized complexity of tree functions: functions that can be represented by an arbitrary tree with the variables corresponding to leaves and the interior nodes to $\wedge$ and $\vee$ gates. Our main contribution is to develop techniques for proving upper and lower bounds (Theorems 2.5 and 2.7) on the randomized complexity of any tree function. We show that for certain special classes of trees (functions), including uniform trees, these bounds agree, implying Theorem 1.5.

Actually, we believe that the bound $R(f) = \Omega\left(D(f)^{.753\ldots}\right)$ holds for all Boolean functions, i.e., that the greatest savings in complexity is achieved for F^h.

Conjecture 1.6. For any Boolean function f

$$R(f) = \Omega\left(D(f)^{\log_2\left(\frac{1+\sqrt{33}}{4}\right)}\right).$$

Finally, our results have application to the analysis of algorithms for game tree search. Programs for two person games like chess typically select a player's next move by first constructing a tree whose nodes represent the set of possible positions that can be reached in the next few moves (nodes at depth i in the tree represent positions reached after i moves). Some heuristic is then used to estimate the value of the positions represented by each leaf. Values of the internal nodes in the tree are computed as follows: If the node represents a position where it is the player's turn then its value is the maximum of the values of its children (MAX node), otherwise it is the minimum of the values of its children (MIN node). The problem, given such a game tree, is to determine which child of the root

has the maximum value; this will be the player's next move. This is essentially equivalent to finding the value of the root. Note that in the case that the leaf values are from {0,1}, the MAX nodes are OR gates, the MIN nodes are AND gates and evaluating the tree is evaluating its tree function.

The best known heuristic for evaluating game trees is the (α,β) pruning procedure (see [KM]), which has been studied in detail ([P1],[P2],[R],[T]). To analyze the complexity of this or any procedure, we need a model for the possible input. Under worst case input, any deterministic algorithm will require evaluation of all leaves. Hence researchers have looked at a model that assumes the leaf values are independent and identically distributed, and analyzed the expected behavior of various algorithms. However, this assumption is unrealistic; typically values assigned to game positions are highly correlated.

We suggest that a more realistic approach is to look at randomized algorithms under worst case input. Our bounds for Boolean functions together with a randomized variant of an algorithm of Pearl [P2] are used to obtain tight upper and lower bounds on the randomized complexity of evaluating uniform game trees of arbitrary degree.

The remainder of this paper consists of four sections. Section II formalizes some notation. Sections III and IV present our general upper and lower bounds. Section V presents some tight complexity results implied by these bounds. Only the main proofs are given in this preliminary report.

II. Notation and Preliminaries

Recall that $D(f)$, $R(f)$ and $N(f)$ are the deterministic, randomized and nondeterministic complexities of the Boolean function f. We summarize the discussion in section 1 with:

Proposition 2.1. For any Boolean function f,

$$D(f) \geq R(f) \geq N(f) \geq \sqrt{D(f)}$$

It will be useful to consider complexity measures subject to restrictions on the input. Let L be a subset of $\{0,1\}^n$. The complexity of an algorithm (in the deterministic, randomized or nondeterministic model) relative to L is its worst case cost over inputs in L. The complexity of f relative to L is the minimum complexity of any algorithm with respect to L. We define $D_0(f)$ (resp. $D_1(f)$) to be the deterministic complexity of f when the input x of f is restricted so that $f(x) = 0$ (resp. $f(x) = 1$). $R_0(f)$, $R_1(f)$, $N_0(f)$ and $N_1(f)$ are similarly defined. (Note that restricting to inputs such that $f(x) = 0$ does not make the complexity equal 0 because our model requires that a successful algorithm "discover" for itself that $f(x) = 0$.) The following is obvious.

Proposition 2.2. For any function f

$$\max\{D_0(f), D_1(f)\} \leq D(f)$$
$$\max\{R_0(f), R_1(f)\} \leq R(f)$$
$$\max\{N_0(f), N_1(f)\} \leq N(f)$$

It is important to note that these inequalities (in the first two cases) may be strict. This is because the best algorithm for computing f when $f = 0$ need not be the best for computing f when $f = 1$. On the other hand, if these two algorithms are the same, then equality will hold.

We are interested in the class of functions that can be represented by formulae over AND, OR and NOT in which each variable appears exactly once. We will represent such functions by rooted trees with interior nodes labelled by AND or OR and leaves labelled by distinct variables (possibly complemented). More precisely an *F-tree* T is a rooted tree with node set $NODES(T)$ consisting of interior nodes, $INT(T)$, and leaves, $LEAVES(T)$, together with a partition of $INT(T)$ into two sets $AND(T)$ and $OR(T)$. We associate to each F-tree T the Boolean function f_T on variables $\{x_u | u \in LEAVES(T)\}$ obtained by interpreting each interior node v as an AND or an OR depending on the set to which it belongs. f_T is called the *tree function* associated with T. The value of an interior node v, which is a function of the leaf values, is denoted x_v; thus $f_T = x_r$ where r is the root. We will often say "T evaluates to 1" when we mean "$f_T = 1$."

Tree functions are necessarily monotone, but it is easy to see that this causes no loss of generality since any nonmonotone function in this class is obtained from some tree function by complementing some input variables.

If T and T' are F-trees we say T' is a *subtree* of T if T' is obtained by assigning values to some of the variables of T and simplifying. If v is any node, the subtree rooted at v (in the standard sense) is a subtree under this definition.

An F-tree is *alternating* if the *AND* and *OR* gates are assigned to alternate levels of interior vertices. Note that every tree function can be represented by an alternating F-tree. Note also that the function F^h from example 1.1 is equivalent (up to complementation of input) to the alternating uniform F-tree of height h.

By a simple adversary argument we can show that the deterministic complexity of any tree function is equal to the number of variables:

Proposition 2.3. For any F-tree T,$D(f_T) = |LEAVES(T)|$.

In what follows, most of our results are stated for binary trees. This usually entails no loss of generality since every function tree can easily be converted to an equivalent binary one. For binary trees, the children of a node v are designated $L(v)$ and $R(v)$. The left and right subtree of a tree T are T_L and T_R.

The following function of four variables a_0,b_0,a_1,b_1 will play an important role in our results.

$$\Psi(a_0,b_0,a_1,b_1) = \frac{a_0a_1+b_0b_1+a_1b_1}{a_1+b_1}.$$

III. Directional Algorithms and Upper Bounds

To obtain an upper bound on $R(f_T)$ we will analyze a class of randomized algorithms called directional algorithms that generalize the algorithm in example 1.1. Let $T_1,...,T_k$ be the subtrees rooted at the children of the root. A *randomized directional algorithm* A for T is specified by a set of randomized directional algorithms $A_1,...,A_k$ for $T_1,...,T_k$ and a probability distribution on the set of permutations of $1,...,k$. A is performed by selecting a permutation σ of $1,...,k$ according to this distribution and evaluating $T_1,...,T_k$ in σ order using $A_1,...,A_k$, until the value at T is known.

Let $d(T)$ denote the minimum expected number of variables evaluated on worst case input over all directional algorithms and $d_0(T)$ (resp. $d_1(T)$) be the analogous quantity when the input is restricted so that $f_T = 0$ (resp. 1). While we have noted that the randomized algorithm that achieves $R_0(f_T)$ may not be the same one achieving $R_1(f_T)$, when we restrict to directional algorithms we obtain

Lemma 3.1. There is a single directional algorithm that attains $d_0(T)$ and $d_1(T)$ and hence $d(T) = \max\{d_0(T),d_1(T)\}$.

We will prove the

Theorem 3.2. Let T be a binary F-tree with root r. The cost of the optimal directional algorithm satisfies the recurrence

$$d_1(T) = d_1(T_L) + d_1(T_R)$$

$$d_0(T) = \max\Big\{d_0(T_L),\ d_0(T_R),\ \Psi(d_0(T_L),d_0(T_R),d_1(T_L),d_1(T_R))\Big\}$$

if $r \in AND(T)$, and

$$d_0(T) = d_0(T_L) + d_0(T_R)$$

$$d_1(T) = \max\Big\{d_1(T_L),\ d_1(T_R),\ \Psi(d_1(T_L),d_1(T_R),d_0(T_L),d_0(T_R))\Big\}$$

if $r \in OR(T)$, with the initial conditions

$$d_0(T) = d_1(T) = 1 \text{ if } |T| = 1 .$$

Proof. Assume $GATE(r) = AND$ (the other case is similar). A randomized directional algorithm on a binary F-tree T is specified by directional algorithms for T_L and T_R together with the probability p_L that the left branch is evaluated first. Whenever T evaluates to 1, both branches evaluate to 1 and must be evaluated by the algorithm, hence

$$d_1(T) = d_1(T_L) + d_1(T_R)$$

If T evaluates to 0 then at least one branch evaluates to 0, and, in worst case the other evaluates to 1. If T_L evaluates to 0 then the expected cost is $d_0(T_L) + (1-p_L)d_1(T_R)$. Otherwise, T_R is 0 (and T_L is 1) and the expected cost is $d_0(T_R) + p_L d_1(T_L)$. Hence

$$d_0(T) = \min_{0 \le p_L \le 1} \max \begin{Bmatrix} d_0(T_L) + (1-p_L)d_1(T_R), \\ d_0(T_R) + p_L d_1(T_L) \end{Bmatrix}$$

The value of p_L achieving this minimum is:

0 if $d_0(T_R) \ge d_0(T_L)+d_1(T_R)$,

1 if $d_0(T_L) \ge d_0(T_R)+d_1(T_L)$, and

$\frac{d_0(T_L)-d_0(T_R)+d_1(T_R)}{d_1(T_R)+d_1(T_L)}$ otherwise,

which when substituted above gives the expression for $d_0(T)$ given in the theorem. □

This theorem provides a general linear time procedure for upper bounding the randomized complexity of f_T.

The algorithm presented in example 1.1 is the optimal directional algorithm for the uniform alternating F-tree. Theorem 1.5 asserts that this is optimal, which will follow from the lower bound of the next section. It should be pointed out however that not all tree functions have an optimal algorithm that is directional.

IV. Reluctant Input Distributions and Lower Bounds

To prove a lower bound on $R(f_T)$ we have to work harder. Yao [Y] has observed that a lower bound on the randomized complexity of a computation in this (or any) model can be obtained by choosing a specific probability distribution on the possible inputs and lower bounding the expected cost of every deterministic algorithm when run against that input distribution. More precisely, his result says:

Lemma 4.1. $R(f)$ is equal to the maximum over all probability distributions on inputs of the minimum expected cost of any deterministic algorithm.

To prove a lower bound on $R(f_T)$, we will use lemma 4.1. A natural distribution on inputs to choose is the distribution that sets each input x_u to 1 independently with some probability p. Pearl [P2] showed that if $p \ne \frac{\sqrt{5}-1}{2}$, then a deterministic directional algorithm runs in expected time $0(\sqrt{2^h}) = 0(\sqrt{n})$ which is the nondeterministic lower bound. If $p = \frac{\sqrt{5}-1}{2}$ then this algorithm takes $0\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) = 0(n^{.694\ldots})$. Tarsi [T] showed that the directional algorithm is optimal against this adversary.

However, the following argument suggests that there may be stronger adversaries. If the adversary is opposed by a directional algorithm then any time the algorithm evaluates a node whose children both have value 0, only one of the children will be evaluated. Against such an algorithm, it would be better to restrict the adversary to inputs which do not have a node both of whose children evaluate to 0. We call these *reluctant* inputs.

We will analyze adversary distributions that give nonzero probability only to reluctant inputs; we call such distribution a *reluctant* distribution. Using this idea we obtain the following lower bound.

Define the functions $\ell_0(T),\ell_1(T)$ inductively by

$$\ell_0(T) = \ell_1(T) = 1 \text{ if } |T| = 1$$

otherwise,

$$\ell_1(T) = \ell_1(T_L) + \ell_1(T_R)$$

$$\ell_0(T) = \min \left\{ \ell_0(T_L) + \ell_1(T_R),\ \ell_0(T_R) + \ell_1(T_L),\ \Psi(\ell_0(T_L),\ell_0(T_R),\ell_1(T_L),\ell_1(T_R)) \right\}$$

if $r \in AND(T)$, and

$$\ell_0(T) = \ell_0(T_L) + \ell_1(T_R)$$

$$\ell_1(T) = \min \left\{ \ell_1(T_L) + \ell_0(T_R),\ \ell_1(T_R) + \ell_0(T_L),\ \Psi(\ell_1(T_L),\ell_1(T_R),\ell_0(T_L),\ell_0(T_R)) \right\}$$

if $r \in OR(T)$.

We prove

Theorem 4.3. For any F-tree T, $R_0(f_T) \ge \ell_0(T)$ and $R_1(f_T) \ge \ell_1(T)$.

Proof. Our notion of complexity of Boolean functions assumes unit cost to read a variable. We will need to generalize this to the case where the cost of evaluating a leaf is a function of the leaf and its value. A *leaf cost function* pair on T is a pair c_0, c_1 of nonnegative real valued functions defined on $LEAVES(T)$ such that $c_i(v)$ represents the cost of evaluating v when $x_v = i$. The randomized complexity of f_T relative to these leaf costs, $R_0(f_T;c_0,c_1)$, $R_1(f_T;c_0,c_1)$ and $R(f_T;c_0,c_1)$, are the natural generalizations of $R_0(f_T)$, $R_1(f_T)$ and $R(f_T)$.

The main step in our lower bound is the following lemma, that shows that the randomized complexity of a tree T with leaf costs c can be lower bounded by that of a smaller tree.

Lemma 4.4. Let T be a binary F-tree and c_0, c_1 be leaf cost functions on T. Let v be a node whose children y and z are leaves. Let T' be the F-tree obtained by deleting y and z from T and define leaf costs for T' by:

$$\left.\begin{aligned} c'_0(w) &= c_0(w) \\ c'_1(w) &= c_1(w) \end{aligned}\right\} \text{ if } w \neq v .$$

$$c'_1(v) = c_1(y) + c_1(z)$$

$$c'_0(v) = \min\{\Psi(c_0(y),c_0(z),c_1(y),c_1(z)),$$
$$c_0(y) + c_1(z), c_0(z) + c_1(y)\}$$

if $v \in AND(T)$,

$$c'_0(v) = c_0(y) + c_0(z)$$

$$c'_1(v) = \min\{\Psi(c_1(y),c_1(z),c_0(y),c_0(z)),$$
$$c_1(y) + c_0(z), c_1(z) + c_0(y)\}$$

if $v \in OR(T)$. Then for $i = 0,1$,

$$R_i(f_T;c_0,c_1) \geq R_i(f_{T'};c'_0,c'_1).$$

Theorem 4.3 follows from lemma 4.4 by applying this shrinking procedure inductively on a tree T with unit leaf costs, and shrinking to a single node. The leaf costs associated to that node will be $\ell_1(T)$ and $\ell_0(T)$, and the theorem follows.

Proof of Lemma 4.4. Assume $v \in AND(T)$ (the other case is similar). Let S' denote the input distribution for T' that maximizes the minimum cost of any deterministic algorithm. Then by lemma 4.1, the best deterministic algorithm costs $R(f_{T'},c_0,c_1)$ when opposed by S'. Let p be the probability that $x_v = 0$ under S'. Define the adversary strategy S for T as follows: Choose an input for T' using S'. If $x_v = 1$ then set $x_y = x_z = 1$. If $x_v = 0$ then set $x_y = 0$ and $x_z = 1$ with probability $\frac{c_1(y)}{c_1(y)+c_1(z)}$ and $x_z = 0$ and $x_y = 1$ with probability $\frac{c_1(z)}{c_1(y)+c_1(z)}$. Then p_y, the probability that $x_y = 0$, is $p\ \frac{c_1(y)}{c_1(y)+c_1(z)}$ and $p_z = p\ \frac{c_1(z)}{c_1(y)+c_1(z)}$.

The conclusion of the lemma follows from the following:

Claim: For any deterministic algorithm A that evaluates T there is an algorithm A' for T' such that the cost of A when opposed by S is at least the cost of A' when opposed by S'.

We will suppose that T is a minimum counterexample to the claim and derive a contradiction. Let A be the optimal deterministic algorithm for T when opposed by S and let w be the leaf for which x_w is queried first. Let A_0 (resp. A_1) denote the branch of A followed if the initial query yields $x_w = 0$ (resp. $x_w = 1$). We distinguish two cases; the second case is the nontrivial one.

Case i. $w \neq y$ or z. Let T_0 (resp. T_1) be the subtrees of T given by setting $x_w = 0$ (resp. $x_w = 1$). Let S_0 (resp. S_1) be the input distribution for T_0 (resp. T_1) defined by conditioning S on $x_w = 0$ (resp. $x_w = 1$). Define T'_0, T'_1, S'_0, S'_1 similarly in terms of T' and S'_0. By the minimality of T the claim holds for T_0 and T_1, hence: there exist A'_0 (resp. A'_1) such that the cost of A_0 (resp. A_1) when opposed by S_0 (resp. S_1) is at least the cost of A'_0 (resp. A'_1) when opposed by S'_0 (resp. S'_1). Now define the algorithm A' to be: evaluate x_w and then do A'_0 (resp. A'_1) if $x_w = 0$ (resp. $x_w = 1$). It is easy to see that the cost of A' opposed by S' on T' is at most the cost of A opposed by S on T contradicting that T is a counterexample.

Case ii. $w = y$ or $w = z$. Assume $w = y$ (without loss of generality). The idea is to define two algorithms A_{now} (which evaluates x_v immediately) and A_{later} (which defers evaluation of x_v) for evaluating T' and show that one of them does at least as well against S' as A does against S. (This is similar to the proof technique of Tarsi [T].)

Let K_0 (resp. K_1) denote the expected cost of A_0 (resp. A_1) on T given $x_y = 0$ (resp. $x_y = 1$), excluding the cost of evaluating x_z.

Now we define

A_{now}: evaluate x_v. If $x_v = 0$ then do A_0. If $x_v = 1$ then do A_1, branching according to $x_z = 1$ when necessary.

A_{later}: Do A_1 on T' replacing evaluation of x_z by evaluation of x_v.

Let r_0 (resp. r_1) denote the probability that A_{later} evaluates x_v against S' when $x_v = 0$ (resp. $x_v = 1$). Note that by the definitions of S and A this is the same as the conditional probability that A evaluates x_z given $x_y = 1$ and $x_z = 0$ (resp. $x_y = 1$ and $x_z = 1$).

The expected cost of running A against S is

$$(4.1)\quad cost(A) = p_y(c_0(y) + K_0) + (1-p_y)(c_1(y) + K_1) + p_z r_0 c_0(z) + (1-p)r_1 c_1(z).$$

The first term is the contribution to the cost of the case $x_y = 0$. The second is the cost if $x_y = 1$, excluding the cost of evaluating x_z. The third and fourth come from the cost of evaluating x_z if $x_z = 0$ and $x_z = 1$.

Similarly, the expected costs of running A_{now} and A_{later} against S on T are:

$$(4.2)\quad cost(A_{\text{later}}) = (1-p)(c'_1(v) + K_1) + (p)(c'_0(v) + K_0)$$

$$(4.3)\quad cost(A_{\text{now}}) = K_1 + (1-p)r_1 c'_1(v) + (p)r_0 c'_0(v)$$

To show that one of these is less than $cost(A)$ it is enough that

$$(4.4)\quad p_y cost(A_{\text{now}}) + p_z cost(A_{\text{later}}) \leq (p_y+p_z)cost(A).$$

Substituting in (4.1), (4.2) and (4.3) into (4.4) and using the definitions of p_y, p_z and $c'_1(v)$, (4.4) can be reduced to

$$(4.5)\quad \frac{1}{c_1(y)+c_1(z)}\Big[c_1(y)(c_0(y)+c_1(z)-c'_0(v)) + r_1 c_1(z)(c_0(z)-c'_0(v))\Big] \geq 0.$$

Now $0 \leq r_0 \leq 1$, so by convexity we need only check that (4.5) holds for $r_0 = 1$ and $r_0 = 0$. When $r_0 = 1$ we need

$$c'_0(v) \leq \frac{c_0(y)c_1(y) + c_1(y)c_1(z) + c_0(z)c_1(z)}{c_1(y) + c_1(z)}$$

$$= \Psi(c_0(y),\ c_0(z),\ c_1(y),\ c_1(z))$$

and when $r_1 = 0$ we need

$$c'_0(v) \leq c_0(y) + c_1(z)$$

Had x_z been the first query of A instead of x_y then the analogous derivation would require also

$$c'_0(v) \leq c_0(z) + c_1(y).$$

The definition of $c'_0(v)$ was chosen to satisfy these inequalities. □

V. Tight Bounds

Note the similarity between the recurrences for ℓ_0,ℓ_1 and d_0,d_1. We would like to get conditions under which the two recurrences coincide. It is useful to define the functions $a_0(T)$ and $a_1(T)$ by the recurrence:

$$a_1(T) = a_1(T_L) + a_1(T_R)$$
$$a_0(T) = \Psi(a_0(T_L),a_0(T_R),a_1(T_L),a_1(T_R))$$

if $v \in AND(T)$ and

$$a_1(T) = a_0(T_L) + a_0(T_R)$$
$$a_0(T) = \Psi(a_1(T_L),a_1(T_R),a_0(T_L),a_0(T_R))$$

if $v \in OR(T)$, with the initial condition

$$a_0(T) = a_1(T) = 1 \text{ if } |T| = 1\ .$$

The recurrence for a_0,a_1 dominates the one for ℓ_0,ℓ_1 and is dominated by the recurrence for d_0,d_1. We can now state the following useful criterion for the lower and upper bounds to match, which is easily proved by induction on the depth of T.

Lemma 5.1. Let T be an F-tree and suppose that for each node $v \in AND(T)$

$$a_0(L(v)) + a_1(R(v)) \geq a_0(R(v))$$

and

$$a_0(R(v)) + a_1(L(v)) \geq a_0(L(v)) ,$$

and for each $v \in OR(T)$

$$a_1(L(v)) + a_0(R(v)) \geq a_1(R(v))$$

and

$$a_1(R(v)) + a_0(L(v)) \geq a_1(L(v)) .$$

Then

$$a_i(T) = \ell_i(T) = d_i(T) = R_i(f_T)$$

for $i = 0,1$.

We can show that the conditions of this lemma hold for *nearly balanced* trees. A binary F-*tree* T *is nearly balanced* if

(i) The left and right subtrees of T_R are each subtrees in T_L.

(ii) The left and right subtrees of T_L are each subtrees in T_R.

Examples of alternating F-trees that are nearly balanced are uniform trees, trees where every leaf is at height h or $h-1$, Fibonacci trees, and binomial trees.

Theorem 5.2. If T is nearly balanced then

$$R(f_T) = a(T) = \ell(T) = d(T)$$

that is, the optimal algorithm for T is directional, and the optimal adversary distribution for T is reluctant.

We also can show that the upper and lower bounds match for the class of skew F-trees. The skew F-tree S^n is the alternating tree defined inductively: S^0 is a single vertex and S^n has left subtree a single vertex and right subtree S^{n-1}, and the root of S^n is an *AND* if n is odd and *OR* if n is even. Observe $D(f) = n$ and $N(f) = \lceil \frac{n+1}{2} \rceil$. Using lemma 5.1 we can show

Theorem 5.3. The randomized complexity of evaluating S^n equals $\frac{n}{2} + \Theta(log\ n)$.

The bounds on $r(T)$ given by Theorems 3.2 and 4.3 be extended to the nonbinary case, but the expressions in the recurrence are rather unpleasant and we omit them. For uniform trees of arbitrary degree and height, we obtain an exact complexity result.

Theorem 5.4. The randomized complexity of evaluating an alternating uniform F-tree U_h^d of height h and degree d equals $\Theta\left(((d-1+\sqrt{d^2+14d+1})/4)^h\right)$.

Note that when $d = 2$ this specializes to Theorem 1.5.

Finally, let us discuss what our results imply for game tree evaluation. Since game trees are min-max trees with arbitrary values at the leaves any lower bound for the randomized complexity of tree functions carries over to game trees, so theorems 4.3 and 5.4 provide such lower bounds.

To get an upper bound (within a constant factor) we use a randomized variant of an algorithm of Pearl [P2], that converts an algorithm for computing tree functions to one for evaluating game trees. For uniform trees, we can prove that the cost of this algorithm is at most a constant times the cost of the directional algorithm for the corresponding tree function and hence we get

Theorem 5.5. The randomized complexity of evaluating a uniform game tree of height h and degree d is $\Theta\left(((d-1+\sqrt{d^2+14d+1})/4)^h\right)$.

Proof. Denote by $g(d,h)$ the minimum expected cost of evaluating a game tree of degree d and height h and by $b(d,h)$ the randomized complexity of evaluating the (alternating) Boolean tree function on the same tree with *AND* assigned to *MIN* nodes and *OR* to *MAX* nodes. Consider the following algorithm for evaluating uniform game trees: choose a child of the root at random and evaluate it recursively. Call its value t. For each other child of the root test whether its value is greater than t by interpreting each leaf value as a boolean variable that is equal to 1 if its value is greater than t, optional and applying the (directional) algorithm. Now (recursively) evaluate those nodes that have been found to have value higher than t. (The expected number is $\frac{d-1}{2}$). Hence

$$g(d,h) \leq \frac{d+1}{2}\, g(d,h-1) + (d-1)b(d,h-1)\,.$$

We claim that there is a positive constant K such that

$$g(d,h) \leq K\ b(d,h)\,.$$

The existence of such a constant is a routine computation using the inequality

$$\frac{d+1}{2} < (d-1+\sqrt{d^2+14d+1}/4) = \lim_{d\to\infty} b(d,h)^{1/d}$$

References

[BvEBL] M. R. Best, P. van Emde Boas and H. W. Lenstra, Jr., A sharpened version of the Aanderaa-Rosenberg Conjecture, Report ZW 30/74, Mathematische Centrum Amsterdam, 1974.

[KSS] J. Kahn, M. Saks and D. Sturtevant, A topological approach to evasiveness, *Combinatorica 4* (1984), 297-306.

[KK] D. J. Kleitman and K. J. Kwiatkowski, Further results on the Aanderaa-Rosenberg conjecture, *J. Combinatorial Theory (B) 28* (1980), 85-95.

[KM] D. E. Knuth and R. W. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence 6* (1975), 293-326.

[MT] U. Manber and M. Tompa, The complexity of problems on probabilistic nondeterministic and alternating decision trees, *JACM 32* (1985), 740-732.

[M] F. Meyer auf der Heide, Nondeterministic versus probabilistic linear search algorithms, *Proc. 26th Annual Symposium on Foundations of Computer Science,* 1985, 65-73.

[P1] J. Pearl, Asymptotic properties of minimax trees and game-searching procedures, *Artificial Intelligence 14* (1980), 113-126.

[P2] J. Pearl, The solution for the branching factor of the alpha beta pruning algorithm and its optimality, *Comm. ACM 25* (1982), 559-564.

[RV] R. Rivest and S. Viullemin, On recognizing graph properties from adjacency matrices, *Theor. Comp. Sci. 3* (1978), 371-384.

[R] I. Roizen, On the average number of terminal nodes examined by alpha-beta, UCLA Cognitive Systems Laboratory Technical Report, 1981.

[Ro] A. L. Rosenberg, On the time required to recognize properties of graphs: A Problem, *SIGACT News 5* #4 (1973), 15-16.

[S] M. Snir, Lower bounds for probabilistic linear decision trees, *Theoretical Computer Science 38* (1985), 69-82.

[T] M. Tarsi, Optimal search on some game trees, *JACM 3* (1983), 389-396.

[Y] A. Yao, Probabilistic computations: towards a unified measure of complexity, *Proc. 18th Annual Symposium on Foundations of Computer Science,* 1977, 222-227.