

# Optimal Search on Some Game Trees

MICHAEL TARSI

*University of California, Los Angeles, California*

**Abstract.** It is proved that the directional algorithm for solving a game tree is optimal, in the sense of average run time, for balanced trees (a family containing all uniform trees). This result implies that the alpha-beta pruning method is asymptotically optimal among all game searching algorithms.

**Categories and Subject Descriptors.** I 2.8 [Artificial Intelligence] Problem Solving, Control Methods and Search—*graph and tree search strategies*

**General Terms:** None

**Additional Key Words and Phrases** Game strategies, alpha-beta pruning, average optimal algorithms

## 1. Introduction

A game for two players is given, represented by a rooted tree as follows: Every possible position in the game is denoted by a node. The root represents the starting position, while every leaf denotes a possible end. An edge from a node  $x$  to another node  $y$  (in this case  $y$  is called a son of  $x$ ) represents a possible move from position  $x$  to position  $y$ . At the beginning of the game a token is located on the root. Each player, in his turn, moves the token from the node on which it is located to any son of this node. The game is finished when the token is placed on any of the leaves. The leaves are labeled  $\{0, 1\}$ . The player who puts the token on a leaf wins if the leaf is labeled 0 and loses if it is labeled 1. In order to analyze the game, we now define for every node  $x$ ,  $v(x)$ , the value of  $x$  as follows: If  $x$  is a leaf labeled 0 or 1, then  $v(x) = 0$  or  $v(x) = 1$ , respectively. If  $x$  is not a leaf, then  $v(x) = 1$  if there exists  $y$ , a son of  $x$ , such that  $v(y) = 0$ . Otherwise  $v(x) = 0$ .  $v(x) = 1$  implies that the player whose turn it is to play from  $x$  can force a win. By "solving a tree" we mean evaluating the value of the root, finding whether the first player wins or loses. We are looking for an algorithm to solve a game tree which is optimal in the sense of average run time. At the beginning of the search the tree is given but the leaves are "covered" so that one cannot see how they are labeled. The elementary unit of time is defined as that amount required for the "exposure" of a leaf to find whether it is labeled 0 or 1. The cost of a solution is defined as the number of leaves exposed during the solving process. Assume now that there is a given probability  $p$  for each leaf to be labeled 1. This probability is equal for all the leaves, and the value of each

This work was supported in part by the National Science Foundation under Grant MCS 78-18924.

Author's address: Department of Mathematics, University of California, Los Angeles, CA 90024.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0004-5411/83/0700-0389\$00.75

leaf does not depend on those of the others. The cost  $V(A)$  of an algorithm  $A$  for solving a given tree is defined as

$$V(A) = \sum_{s \in S} p(s)V_A(s),$$


where  $s$  is any  $\{0, 1\}$  assignments for the leaves,  $p(s)$  is the probability for  $s$  to occur, and  $V_A(s)$  is the cost of the solution obtained by  $A$ , for the tree with  $s$ . The sum is taken over  $S$ , the set of all possible  $\{0, 1\}$  assignments. An algorithm  $A$  for a tree  $T$  is called optimal if  $V(A)$  is minimal among all the costs of algorithms for solving  $T$ .

Before starting the analysis let us use some examples to demonstrate the concepts of this paper.

I.



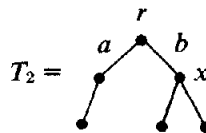
For this tree there exists only one (up to a permutation) reasonable algorithm,  $A_1$ : First expose one leaf. If it is labeled 0, then the value of the root is one. If it is labeled 1, the second leaf should be exposed. The cost of this algorithm is now evaluated (directly from the definition).

			
$s$	$p(s)$	$V_{A_1}(s)$	
0	$(1-p)^2$	1	
0	$p(1-p)$	1	
1	$p(1-p)$	2	
1	$p^2$	2	

$$V(A_1) = \sum p(s)V_{A_1}(s) = 1 + p$$

This result can also be obtained by the following consideration: *One* leaf should be exposed and the probability for the need to expose the other one is  $p$  (when the first is labeled 1).

II.



For this tree we evaluate the costs of two different algorithms:  $A_2, a$ , which starts with the leaf under the edge  $a$ , and  $A_2, b$ , which first solves the subtree rooted at  $x$ . For  $A_2, a$ , if the first leaf is labeled 1, the whole tree is solved ( $v(r) = 1$ ); otherwise (probability  $1 - p$ ) we have to evaluate  $v(x)$ , which is equivalent to solving  $T_1$ . Thus

$$V(A_2, a) = 1 + (1 - p)V(A_1) = 2 - p^2.$$

For  $V(A_2, b)$  it seems that we have to find the costs of evaluating  $v(x)$  for  $v(x) = 0$  and for  $v(x) = 1$  separately, to add 1 to the cost when  $v(x) = 1$  (in this case the third leaf should be exposed), and to multiply each cost by its probability. Actually this is the same as taking just the average cost of evaluating  $x$  and adding the probability for the need to expose the third leaf (multiplied by its cost, which is 1 in this case). Thus

$$V(A_2, b) = V(A_1) + (1 - p^2) = 2 - p^2 + p.$$

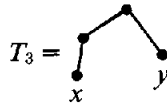
(From now on we always use the average costs of some partial evaluation, although the cost and the result of the evaluation are not independent. This method is justified

by the theorem which says that the expectation of the sum of random variables is equal to the sum of expectations, even if they are dependent. For more details see, for example, [2, p. 222, Th. 1].)

By considering all feasible algorithms one can easily verify that  $A_2, a$  is an optimal algorithm for  $T_2$ , for all values of  $p$ . (Note: Comparing  $V(A_2, a)$  and  $V(A_1)$ , for some values of  $p$  solving  $T_2$  costs more than solving  $T_1$ , while the opposite occurs for other values of  $p$ .)

We now give an example of a tree for which the optimal algorithm varies with  $p$ .

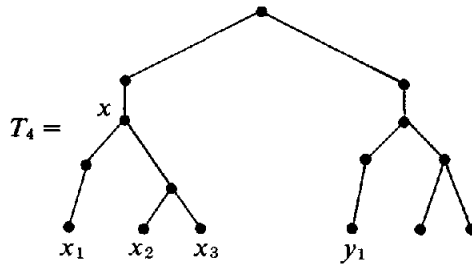
III.



Starting with  $x$  the cost is  $2 - p$ , while the cost is  $1 + p$  if we first expose  $y$ .

In all the examples above the optimal algorithms were “straight” in the sense that whenever a descendant of a node  $x$  is exposed,  $v(x)$  is completely evaluated before the exposure of any leaf which is not a descendant of  $x$ . However, there are trees for which a straight optimal algorithm does not exist, as we see in the following example:

IV.



Assume that  $p$  is almost 1. In this case, starting with  $x_1$ , it probably takes just one step to solve the whole tree, but if  $x_1$  is found to be labeled 0, then it is better to “jump” to  $y_1$ , which probably requires one more step, instead of proceeding with  $x_2, x_3$ , which probably requires two additional steps. The optimal algorithm for high values of  $p$  (with additional effort one can verify that this “nonstraight” algorithm is optimal for every  $p$ ) is not straight because it may leave the subtree rooted at  $x$  before finding  $v(x)$ .

Examples III and IV show that finding an optimal algorithm for an arbitrary and sufficiently large tree is probably a very hard job. The algorithm should depend on  $p$  and should “jump” on the tree from one branch to another.

The main result of this paper is that the situation is not that bad for a certain family of trees. For this family, namely, the balanced trees, there always exists a straight algorithm which is optimal for every value of  $p$ . Before stating the theorem we introduce additional notation.

*Definition.* A rooted tree is called balanced if

- (a) all the leaves have the same distance  $d$  from the root and
- (b) the degrees of all nodes at a given level are equal.

(The level of a node is its distance from a leaf; the leaves are at level 0, the root at level  $d$ .)

**THEOREM.** *If  $T$  is a balanced game tree, then the following algorithm named SOLVE is optimal for solving  $T$  with any value of  $p$ :*

*Expose the leaves from left to right, skipping a leaf whenever there is sufficient information to evaluate one of its ancestors.*

SOLVE is defined and its cost for uniform trees is evaluated in [5].

**PROOF.** An algorithm is called  $n$ -straight if, whenever a descendant of a node  $x$  of level  $n$  or less is exposed during the solution by  $A$ ,  $v(x)$  is evaluated before the exposure of any leaf which is not a descendant of  $x$ .

Obviously, every algorithm is 0-straight and for balanced trees every  $d$ -straight algorithm is equivalent (up to an automorphism of the tree) to SOLVE.

Thus, in order to prove the theorem, we have to prove the existence of a  $d$ -straight optimal algorithm. We do this by induction, showing that the existence of an  $(n-1)$ -straight optimal algorithm ( $n \leq d$ ) implies the existence of an optimal  $n$ -straight one.

Assume  $A$  is an optimal algorithm for the solution of a balanced game tree and that  $A$  is  $(n-1)$ -straight but not  $n$ -straight. This means that there exists a node  $x$  of level  $n$  such that, under some circumstances, some descendants of  $x$  are exposed and then  $x$  is left before  $v(x)$  is found. Since  $A$  is  $(n-1)$ -straight, this happens only if some, but not all, say  $t$ , of  $x$ 's sons had been found to have the value 1 before  $x$  was left. We now define some quantities related to the work of  $A$ .

#### Notation

$p_1$	The probability that a son of $x$ has the value 1.
$\alpha$	The average cost of evaluating a son of $x$ .
$q = p_1^t$	The probability that the first $t$ sons of $x$ all have the value 1.
$\beta$	The average cost for $A$ to solve the first $t$ sons of $x$ .
$1 - p_2$	The probability that $A$ solves the tree after leaving $x$ (finding $t$ 1-sons of $x$ ) without going back to the $(t+1)$ st son of $x$ .
$\gamma$	The average cost spent by $A$ , after leaving $x$ , until either going back to the $(t+1)$ st son of $x$ or solving the tree without coming back to $x$ .
$V_1$	The average cost of completing the solution when one son of $x$ is known to have the value 0 ( $v(x) = 1$ ), without any information about the rest of the tree.
$V_2$	The average cost of completing the solution, knowing $v(x) = 1$ , having the information about the rest of the tree, obtained by $A$ before solving the $(t+1)$ st son of $x$ .
$V_3$	The cost of completing the solution after the $(t+1)$ st son of $x$ is found to have the value 1.

Figure 1 explains the work of  $A$  and the above-defined quantities. The starting point is that moment when  $A$  first starts exposing the descendant of  $x$ , although some work might be done before.

$\beta$  can be expressed in terms of  $\alpha$  and  $p_1$ . The first  $t$  sons of  $x$  are solved when one of them is found to have the value 0 or all of them the value 1. Thus

$$\beta = \alpha + p_1\alpha + p_1^2\alpha \cdots + p_1^{t-1}\alpha = \alpha \frac{1 - p_1^t}{1 - p_1},$$

$$p_1^t = q,$$

and so we get

$$\alpha = \beta \frac{1 - p_1}{1 - q}. \quad (1)$$

Now we define two variations of the algorithm  $A$ , namely,  $B$  and  $C$ .  $B$  does not enter to  $x$  but starts with the rest of the tree, as  $A$  does after finding  $t$  1-sons of  $x$ .  $B$

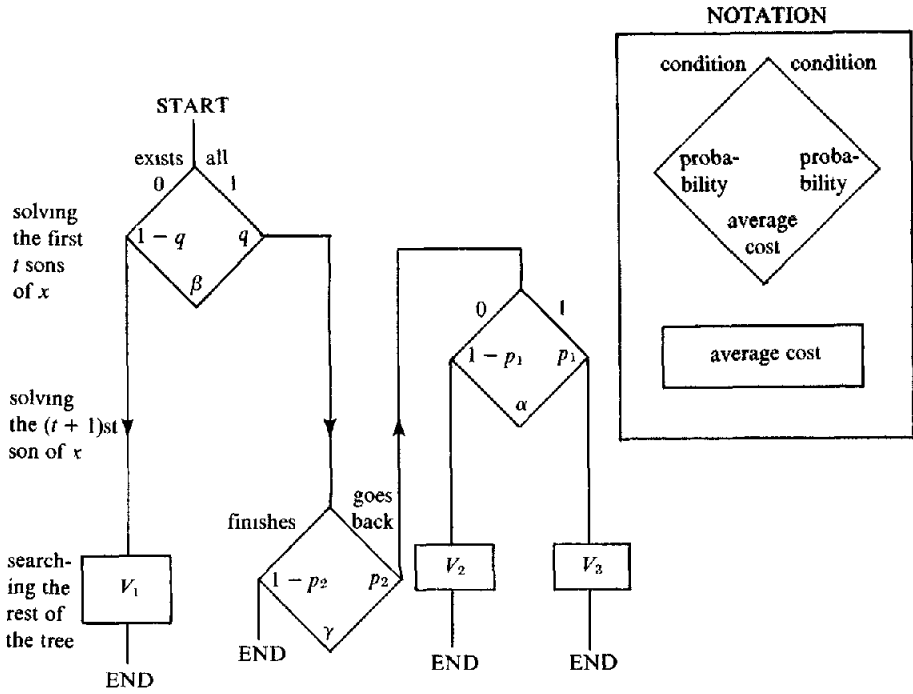


FIG 1 Algorithm A

goes to  $x$  under the same conditions which make  $A$  go back to the  $(t + 1)$ st son of  $x$ .  $C$  starts in the same way as  $A$  but solves  $t + 1$  sons, one more than  $A$  does, before leaving  $x$ .

Algorithms  $B$  and  $C$  are represented by Figures 2 and 3. By  $V(A)$ ,  $V(B)$ ,  $V(C)$  we denote the costs of the corresponding algorithms, measured since the "start" point. From the definitions, using the flowcharts, we obtain

$$\begin{aligned}
 V(A) &= (1 - q)V_1 + qp_2(1 - p_1)V_2 + qp_1p_2V_3 + qp_2\alpha + \beta + q\gamma, \\
 V(B) &= p_2(1 - q)V_2 + qp_2(1 - p_1)V_2 + qp_1p_2V_3 + qp_2\alpha + p_2\beta + \gamma, \\
 V(C) &= (1 - q)V_1 + q(1 - p_1)v_1 + qp_1p_2V_3 + q\alpha + \beta + qp_1\gamma.
 \end{aligned}$$

Since  $A$  is optimal,  $V(A) \leq V(B)$ , which gives us

$$(1 - q)V_1 + \beta \leq p_2(1 - q)V_2 + p_2\beta + (1 - q)\gamma.$$

Multiplying by  $q(1 - p_1)/(1 - q)$  and applying eq. (1), we get

$$q(1 - p_1)V_1 + q\alpha \leq qp_2(1 - p_1)V_2 + qp_2\alpha + q(1 - p_1)\gamma,$$

and adding  $(1 - q)V_1 + qp_1p_2V_3 + \beta + qp_1\gamma$  to both sides, we obtain  $V(C) \leq V(A)$ , which means that  $C$  is at least as good as  $A$ . Since  $A$  is assumed to be optimal, so is  $C$ . Thus, evaluating one more son before leaving  $x$  does not compromise the optimality. Applying this technique to all sons of  $x$  leads to an optimal  $n$ -straight algorithm, and the induction is complete.

*Applications*

Our analysis so far has been confined to game trees with bivalued terminal nodes. However, the theorem also has projections on the complexity of game-playing programs. In such programs the terminal nodes are assigned continuous values

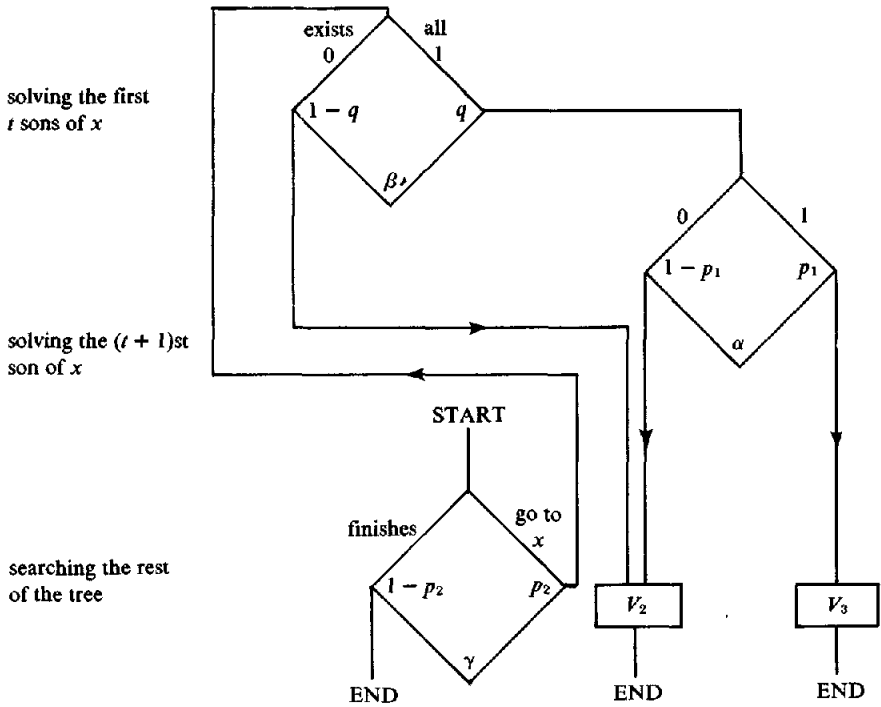


FIG. 2. Algorithm B.

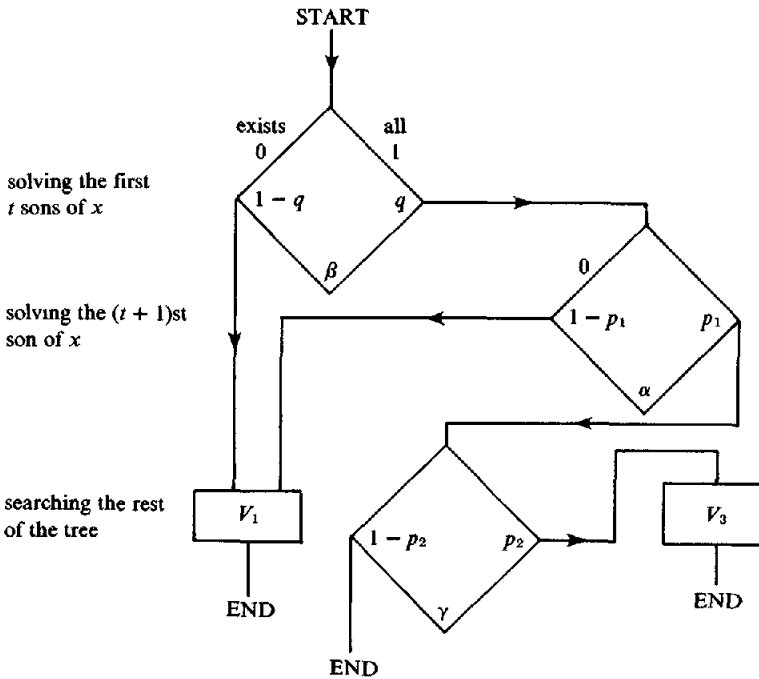


FIG. 3. Algorithm C

representing estimates of the strength or promise of the corresponding game positions, and the task is to determine the minimax value of the root node by examining the least number of terminal nodes.

The standard testing ground adapted for comparing the efficiency of various game-searching strategies is a uniform game tree of depth  $d$  and degree  $n$ , where the terminal positions are assigned random values independently drawn from a common, continuous distribution function  $F$ .

We refer to such a tree as a  $(d, n, F)$ -tree.

*Definition.* Let  $A$  be a deterministic algorithm which searches a  $(d, n, F)$ -tree to determine the minimax value of its root, and let  $I_A(d, n, F)$  denote the expected number of terminal positions examined by  $A$ . The quantity

$$\mathcal{R}_A(n, F) = \lim_{d \rightarrow \infty} [I_A(d, n, F)]^{1/d}$$

is called the *branching factor* corresponding to the algorithm  $A$  [4].

The alpha-beta ( $\alpha$ - $\beta$ ) pruning algorithm is the most commonly used procedure in game-playing applications. Yet, though the exponential growth of game-tree searching is slowed significantly by that algorithm, quantitative analyses of its effectiveness are still under way [1, 3-5, 7]. Recently, Pearl [6] has shown that

$$\mathcal{R}_{\alpha-\beta} = \frac{\xi_n}{1 - \xi_n}, \quad (2)$$

where  $\xi_n$  is the positive root of the equation  $x^n + x - 1 = 0$ . The expression  $\xi_n/(1 - \xi_n)$  also represents the branching factor of SOLVE under the condition  $p = \xi_n$ , and it lower bounds the branching factor of all directional algorithms which evaluate continuous-valued trees [5]. These results establish  $\alpha$ - $\beta$  as asymptotically optimal over the class of directional algorithms but leave unsettled the question of its global optimality. Indeed, Stockman [8] introduced a nondirectional algorithm called SSS\* which consistently examined fewer nodes than  $\alpha$ - $\beta$ . Hopes were then raised that the superiority of Stockman's algorithm reflected an improved branching factor over that of  $\alpha$ - $\beta$ .

The optimality of  $\alpha$ - $\beta$  strongly hinges upon the complexity of solving bivalued game trees. Pearl has shown [5] that if any algorithm can solve a bivalued  $(d, n, p = \xi_n)$ -tree with branching factor  $R_b$ , then another algorithm (called SCOUT) can be devised for evaluating a continuous-valued  $(d, n, F)$ -tree with a branching factor identical to  $R_b$ .

At the same time the task of solving any bivalued game tree is equivalent to the task of verifying an inequality proposition regarding the minimax value of a continuous-valued game tree [5] of identical structure, and consequently the former cannot be more complex than the latter. These considerations imply that the optimal branching factor associated with evaluating a continuous-valued tree is identical to that associated with solving a standard bivalued game tree in which the terminal nodes are assigned the values 1 and 0 with the probabilities  $\xi_n$  and  $1 - \xi_n$ , respectively.

The main theorem of this paper states that SOLVE spends the minimal average search effort on the standard game tree and, therefore, that any algorithm which solves such a game tree must, on the average, examine at least  $(\xi_n/(1 - \xi_n))^d$  terminal positions.

This, together with (2), establishes the asymptotic optimality of  $\alpha$ - $\beta$  over all game-searching algorithms, directional as well as nondirectional.

## REFERENCES

1. BAUDET, G.M. On the branching factor of the alpha-beta pruning algorithm. *Artif. Intell.* 10 (1978), 173-199.
2. FELLER, W. *An Introduction to Probability Theory and Its Applications*. Wiley, New York, 1968.
3. FULLER, S.H., GASCHNIG, J.G., AND GILLOGLY, J.J. An analysis of the alpha-beta pruning algorithm. Department of Computer Science Rep., Carnegie-Mellon Univ., Pittsburgh, Pa., 1973.
4. KNUTH, D.E., AND MOORE, R.N. An analysis of alpha-beta pruning. *Artif. Intell.* 6 (1975), 293-326.
5. PEARL, J. Asymptotic properties of minimax trees and game-searching procedures. *Artif. Intell.* 14, 2 (1980), 113-138.
6. PEARL, J. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Commun. ACM* 25, 8 (Aug. 1982), 559-564.
7. SLAGLE, J.R., AND DIXON, J.K. Experiments with some programs that search game trees. *J. ACM* 2, 1969, 189-207.
8. STOCKMAN, G. A minimax algorithm better than alpha-beta? *Artif. Intell.* 12 (1979), 179-196.

RECEIVED JULY 1981, REVISED MAY 1982; ACCEPTED MAY 1982