Game Tree Searching by Min/Max Approximation*

Ronald L. Rivest
Laboratory for Computer Science, MIT, Cambridge, MA 02139, U.S.A.

Recommended by Hans Berliner

ABSTRACT

We present an iterative method for searching min/max game trees based on the idea of approximating the "min" and "max" operators by generalized mean-valued operators. This approximation is used to guide the selection of the next leaf node to expand, since the approximations allow one to select efficiently that leaf node upon whose value the (approximate) value at the root most highly depends. Experimental results from almost 1,000 games of Connect-Four[1] suggest that our scheme is superior to minimax search with alpha-beta pruning, for the same number of calls to the move routine. However, our scheme has higher overhead, so that further work is needed before it becomes competitive when CPU time per turn is the limiting resource.

1. Introduction

This paper introduces a new technique for searching in game trees, based on the idea of approximating the min and max operators with generalized mean-value operators.

Game playing by computer has a long history, and many brilliant ideas have led us to the point where high-quality play for many games can be obtained with pocket-sized computers. (See [3] for an exposition of previous work in this area, and Pearl's book [13] for an excellent introduction to the mathematical analysis of game-playing programs.)

However, further improvement is certainly possible, and this area of research is still an active one. The combinatorial explosion of possibilities in a game such as chess tax our most powerful computers, and even special-purpose hardware soon reaches its limits. Clearly, the most careful organization and allocation of computational resources is needed to obtain expert-level play.

* This research was supported by NSF grants DCR-8006938 and DCR-8607494.
[1] Connect-Four is a trademark of the Milton-Bradley company.

Artificial Intelligence **34** (1988) 77–96
0004-3702/88/$3.50 © 1988, Elsevier Science Publishers B.V. (North-Holland)

Techniques such as alpha-beta pruning and its successors [6, 2] have been essential in reducing the computational burden of exploring a game tree. Still, new techniques are needed. Nau et al. [10], after much expermentation with existing methods, assert that "A method is needed which will always expand the node that is expected to have the largest effect on the value." This paper suggests such a method.

Our method, "min/max approximation," attempts to focus the computer's attention on the important lines of play. The key idea is to approximate the "min" and "max" operators with generalized mean-value operators. These are good approximations to the min/max operators, but have continuous derivatives with respect to all arguments. This allows us to define the "expandable tip upon whose value the backed-up value at the root most heavily depends" in a nontrivial manner. This tip is the next one to be expanded, using our heuristic.

In Section 2 of this paper I present the essential results about generalized mean values that underly the new method. Then, in Section 3, these ideas are applied to the problem of searching game trees. In Section 4, I give some thoughts regarding implementation details. Section 5 describes our preliminary experimental results. Some final thoughts are presented in Section 6 .

2. Generalized Mean Values

Let $\boldsymbol{a} = (a_1, \ldots, a_n)$ be a vector of n positive real numbers, and let p be a nonzero real number. Then we can define the generalized *p-mean* of $\boldsymbol{a}$, $M_p(\boldsymbol{a})$, by

$$M_p(\boldsymbol{a}) = \left(\frac{1}{n} \sum_{i=1}^{n} a_i^p \right)^{1/p} . \tag{1}$$

Of course, $M_1(\boldsymbol{a})$ is the ordinary arithmetic mean. We can extend our notation to the case $p = 0$ by

$$M_0(\boldsymbol{a}) = \lim_{p \to 0} M_p(\boldsymbol{a}) = (a_1 \ldots a_n)^{1/n} , \tag{2}$$

so that $M_0(\boldsymbol{a})$ is the geometric mean.

We begin with the fact that

$$p < q \Rightarrow M_p(\boldsymbol{a}) \leq M_q(\boldsymbol{a}) , \tag{3}$$

where equality only holds on the right if all the a_i are equal. (See [5] for proofs and other facts about generalized mean values.)

For our purposes, we are most interested in the following two facts:

$$\lim_{p \to \infty} M_p(\boldsymbol{a}) = \max(a_1, \ldots, a_n) , \tag{4}$$

$$\lim_{p \to -\infty} M_p(\boldsymbol{a}) = \min(a_1, \ldots, a_n) . \tag{5}$$

TABLE 1. $M_p(\boldsymbol{a})$ for $\boldsymbol{a} = (10, 21, 29, 32)$

p	-32	-16	-8	-4	-2	-1	0	1	2	4	8	16	32
$M_p(\boldsymbol{a})$	10.4	10.9	11.9	13.9	16.6	18.7	21.0	23.0	24.5	26.5	28.3	29.7	30.7

To illustrate the above facts, consider Table 1 where various values of $M_p(\boldsymbol{a})$ are given for $\boldsymbol{a} = (10, 21, 29, 29, 32)$ and various p.

For large positive or negative values of p, $M_p(\boldsymbol{a})$ is a good approximation to $\max_i(a_i)$ or $\min_i(a_i)$, respectively. However, $M_p(\boldsymbol{a})$, unlike max or min, has *continuous* derivatives with respect to each variable a_i. The partial derivative of $M_p(\boldsymbol{a})$ with respect to a_i is

$$\frac{\partial M_p(\boldsymbol{a})}{\partial a_i} = \frac{1}{n}\left(\frac{a_i}{M_p(\boldsymbol{a})}\right)^{p-1}. \tag{6}$$

The major reason that the generalized means are of interest to us here is that they are more suitable for a "sensitivity analysis" than the min or max functions. We propose that $\partial M_p(\boldsymbol{a})/\partial a_i$ (for large p) is a more useful quantity than $\partial \max(\boldsymbol{a})/\partial a_i$, since the latter is zero unless a_i is the maximum, in which case it is one. This discontinuous behavior is awkward to work with, whereas the derivative (6) is continuous. We also note that the derivative (6) ranges in value from 0 to $n^{-1/p} \approx 1$ for $p \gg \ln n$).

By way of example, with $\boldsymbol{a} = (a_1, a_2, a_3, a_4) = (10, 21, 29, 32)$, and $p = 32$, we have $\nabla M_p(\boldsymbol{a}) \approx (2 \times 10^{-16}, 2 \times 10^{-6}, 0.04, 0.90)$. The a_i values near the maximum have much more effect on $M_p(\boldsymbol{a})$ here than do smaller values.

Although we shall not use them in this paper, other forms of generalized mean values exist. For example, if f is any continuous monotone increasing function (such as the exponential function), we can consider mean values of the form

$$f^{-1}\left(\frac{1}{n}\sum_{i=1}^{n} f(a_i)\right). \tag{7}$$

Using $f = \exp(\cdot)$ would yield a good approximation to $\max(\cdot)$, and $f = \ln(\cdot)$ would yield a good approximation to $\min(\cdot)$.

One of the ideas of this paper is that by using the generalized mean values to approximate the min and max functions, we can identify in an interesting way that leaf in a game tree upon whose value the value at the root depends most strongly. This is done by taking derivatives of the generalized mean value functions at each node and using the chain rule. This leaf will be the one to expand next. These ideas will be made more precise in the next sections.

3. Game Tree Searching

3.1. Game trees

Consider a two-person zero-sum perfect information game between players Min and Max which begins in a starting configuration s with Max to move, after which they alternate turns. The game defines a finite tree C of configurations with root s. We split C into subsets *Min* and *Max* depending on whose turn it is to play. For each $c \in C$ we let $S(c)$ denote the set of c's *successors* (or *children*). To move from configuration c a player selects some $d \in S(c)$; his opponent must then move from configuration d. Configurations with no successors are called *terminal configurations*; $T(C)$ will denote the set of terminal configurations—these form the leaves of the tree. The game stops when a terminal configuration is reached. We assume that the relation S is acyclic, so the game always stops. The actual play traces out a path in the tree from the root s to leaf representing a terminal configuration t.

Each leaf $t \in T(C)$ has an associated *value* or *score* $v(t)$; this is the value of the terminal position from Max's point of view. By induction on the structure of the tree we may determine the value $v(c)$ of any configuration $c \in C$ by "backing up" or "minimaxing" the values at the leaves:

$$v(c) = \begin{cases} v(c), & \text{if } c \in T(C), \\ \max\limits_{d \in S(c)} v(d), & \text{if } c \in Max \backslash T(C), \\ \min\limits_{d \in S(c)} v(d), & \text{if } c \in Min \backslash T(C). \end{cases} \tag{8}$$

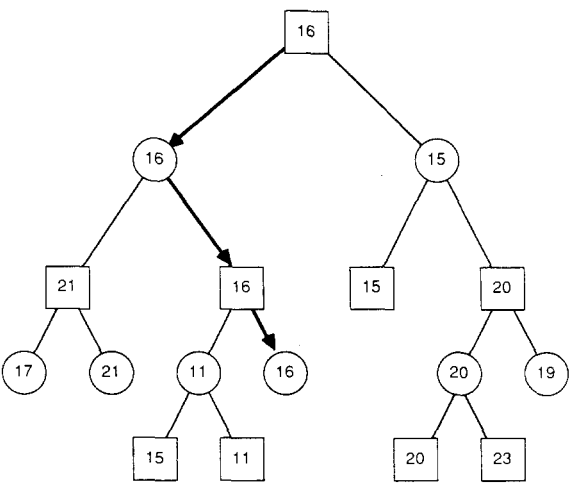

FIG. 1. A small game tree.

Given $v(c)$ for all $c \in C$, optimal moves are easy to determine: if $c \in Max$, then player Max should select any configuration $d \in S(c)$ with maximum value $v(d)$. Similarly, if $c \in Min$, then player Min should select any configuration $d \in S(c)$ with minimum value $v(d)$. Figure 1 shows a small game tree. Configurations in *Max* are shown as squares, those in *Min* are shown as circles. The value of each configuration is shown inside the square or circle. The value of the game is 16; optimal play is indicated with the heavy arrows.

3.2. Searching a game tree

When C is small, the tree can be explored completely, so optimal play is possible. On slightly larger trees minimax search with alpha-beta pruning [6] may produce optimal play even though only a small fraction of the game tree is explored—the portions of the tree that are "pruned" (not explored) are known not to be relevant.

However, for most interesting games the game tree is so large that heuristic approximations are needed.

A heuristic method is usually based on a "static evaluation function" $\hat{v}$ that gives an estimate $\hat{v}(c)$ of the backed-up value $v(c)$ for a nonterminal node c. This estimate is based on "static" features of the current configuration that can be evaluated without further look-ahead (e.g. piece count and an advantage for the player to move).

Our proposed technique requires a single static evaluator $\hat{v}(\cdot)$. Some other methods—most notably the B^* algorithm [2]—require *two* static evaluation functions which are upper and lower bounds on $v(\cdot)$.

A popular approach to handling very large game trees is to select a suitable depth bound d, to estimate the values of nodes at depth d using the static evaluator, and then to compute the backed-up minimax value from the nodes at depth d using alpha-beta pruning.

Given a limit on the computing time available, one can successively compute the backed-up values for depths $d = 1, 2, \ldots$, until one runs out of time, use the move determined by the last search completed. This technique is known as *iterative deepening*.

For familiar games (e.g. chess) as one increases d the accuracy of the value computed for the root seems to improve. However, there are "pathological" games for which increasing d seems to yield *less* accuracy [8, 9]. We assume our game is nonpathological.

A different class of heuristics are the *iterative* heuristics, which "grow" the search tree one step at a time. At each step a tip node (or leaf) of the current tree is chosen, and the successors of that tip node are added to the tree. Then the values provided by the static evaluator at the new leaves are used to provide new backed-up values to the leaves' ancestors. The tree grown by an iterative heuristic need not be of uniform depth: some branches may be searched to a much greater depth than other branches.

Examples of such iterative techniques are the Berliner's B* algorithm [2], Nilsson's "arrow" method [11], Palay's probability-based method [12], and McAllester's "conspiracy number" method [7].

The heuristic proposed in this paper is an iterative technique in the above sense.

3.3. Iterative search heuristics details

We now formalize how a tree is explored using an iterative search heuristic, and how estimates are "backed up" to the root.

We begin with some straightforward definitions.

If $d \in S(c)$, we say that c is the father of d. We denote the father of d by $f(d)$, and define $f(s) = s$. We call c a *sibling* of d if $f(c) = f(d)$, and we call c an *ancestor* of d if $c = d$ or if c is an ancestor of $f(d)$. We let $A(d)$ denote the set of d's ancestors. Note that c is both an ancestor and a sibling of itself.

If E is a partial game tree, then for any $c \in E$ the *subtree of E rooted at c*, denoted E_c, is $\{x \mid x \in E \& c \in A(x)\}$, the set of all $x \in E$ which have c for an ancestor.

If E is a partial game tree, then $c \in E$ is a *tip of E* if c has no successors in E, i.e. $S(c) \cap E = \emptyset$. The set of tips of E is denoted $T(E)$. The tips of C are the terminal configurations. We say that a tip x of E is *expandable* if x has successors.

If E is a partial game tree, and c is an expandable tip of E, then to *expand E at c* means to add the successors $S(c)$ of c to E. If we expand E at a tip c, then E remains a partial game tree.

We let $\hat{v}$ denote our static evaluation function. We assume that $\hat{v}(c) = v(c)$ if $c \in T(C)$; our static evaluation function is exact on terminal positions. For nonterminal positions c $\hat{v}(c)$ is merely an estimate of $v(c)$.

For any partial game tree E and any $c \in E$ we define the backed-up estimates $\hat{v}_E(c)$ of $v(c)$ in a manner identical to the way the correct values are backed-up using equation (8), except that we are backing up the static evaluations from the tips of E rather than backing them up from the terminal positions of C:

$$\hat{v}_E(c) = \begin{cases} \hat{v}(c), & \text{if } c \in T(E), \\ \max\limits_{d \in S(c)} (\hat{v}_E(d)), & \text{if } c \in Max \backslash T(E), \\ \min\limits_{d \in S(c)} (\hat{v}_E(d)), & \text{if } c \in Min \backslash T(E). \end{cases} \tag{9}$$

If $E = \{s\}$ (our partial game tree is the minimal possible game tree), then $\hat{v}_E(s) = \hat{v}(s)$. On the other hand, if $E = C$ then our estimates are exact: $\hat{v}_E(c) = v(c)$ for all c. We expect that $\hat{v}_E(s) \to v(s)$ as $E \to C$ for our "non-pathological" game. (The use of the limiting notation "$\to$" here is merely suggestive and not formal.)

We observe that if we expand E at c, then we can update $\hat{v}_E$ by recomputing equation (9) only at nodes in $A(c)$ (first at c, then $f(c)$, and on up the tree until s is reached).

The general process of partially exploring a game tree by an iterative heuristic can be formalized as follows:

Step 1. Initialize E to $\{s\}$, and $\hat{v}_E(s)$ to $\hat{v}(s)$.
Step 2. While $E \neq C$, and while time permits, do:
(a) Pick an expandable tip c of E.
(b) Expand E at c.
(c) Update $\hat{v}_E(c)$ at c and the ancestors of c up to the root s, using equation (9).

The major unspecified detail here is in Step 2(a)—which expandable tip c of E should we pick? The purpose of this paper is to provide a new answer to this question.

Another intriguing question is how one might gauge the accuracy of the current estimate of the value at the root, in case one wishes to use a termination condition based on accuracy instead of a termination condition based on time. We do not pursue this question is this paper.

3.4. Penalty-based iterative search methods

We now present a general method for choosing which leaf to expand in an iterative method; our technique is a specific instance of this method.

We assume that for each node $c \in E$ an estimate $\tilde{v}_E(c)$ of $v(c)$ is available; for example, we may have $\tilde{v}_E(c) = \hat{v}_E(c)$. If $c \in T(E)$, we assume that $\tilde{v}_E(c) = \hat{v}(c)$. Otherwise, we assume $\tilde{v}_E(c)$ only depends upon $\tilde{v}_E(d)$ for $d \in S(c)$, and whether c is in *Min* or *Max*.

Although one could in principle work just with the backed-up estimates $\hat{v}_E(\ )$, the derivation of our method is clearest if we permit $\tilde{v}_E(\ )$ to differ from $\hat{v}_E(\ )$. Nonetheless, our actual implementation will use $\tilde{v}_E(\ ) = \hat{v}_E(\ )$ for efficiency reasons.

We assign a nonnegative "penalty" (or "weight") to every edge in the game tree such that edges representing bad moves are penalized more than edges representing good moves. We let $w(c)$ denote the weight on the edge between c and its father $f(c)$, and define $w(s)$ to be zero.

We assume that $w(c)$ is computable from $\tilde{v}_E(f(c))$, and the values $\{\tilde{v}_E(d) \mid d$ is a sibling of $c\}$. (Recall that c is its own sibling.)

For example, we might define $w(c)$ to be $\alpha + (\tilde{v}_E(d) - \tilde{v}_E(c))^2$, where $\alpha > 0$ is the penalty for descending a level and where d is the sibling of c which optimizes $\tilde{v}_E(d)$. (Our actual proposal will be somewhat different.)

We define the "penalty" $P(c)$ of a tip $c \in T(E)$ to be the sum of the penalties of all the edges between c and the root s:

$$P(c) = \sum_{d \in A(c)} w(d) .$$

Our idea is then to expand that tip node t which has the *least* penalty $P(t)$. We add t's children to the tree, update the estimate $\tilde{v}_E(c)$ for every $c \in A(t)$, and update the penalties on the edges between each $c \in A(t)$ and its children.

The min/max approximation technique presented here is such a penalty-based scheme.

We now describe the implementation of a generic penalty-based scheme in more detail.

Let E denote the current partial game tree.

For any $c \in E$, and any $d \in E_c$, we define $P_c(d)$ to be the sum of the weights of the edges between d and c; $P_c(d)$ is the penalty of d relative to the subtree E_c rooted at c. (So $P(d) = P_s(d)$.)

For any node $c \in E$, we define $b(c)$ to be the expandable tip node in the subtree E_c which minimizes $P_c(x)$. Ties are resolved arbitrarily (e.g. by selecting the leftmost such node). If none of E_c's tip nodes are expandable, we define $b(c)$ to be the special value ω.

For any node $c \in E$, we define $a(c)$ as follows. If $b(c) = \omega$ then $a(c) = \omega$. If c is an expandable tip of E then $a(c) = c$. Otherwise $a(c)$ is that $d \in S(c)$ such that $b(c) \in E_d$. Think of $a(c)$ as an "arrow" from c to one of its children, such that following successive arrows leads from c to $b(c)$. (See [11] for the origin of this "arrow" terminology.)

With each node c of E we store $\tilde{v}_E(c)$, $a(c)$, and $\pi(c) = P_c(b(c))$, the penalty of the best expandable tip $b(c)$ of E_c relative to the subtree E_c, or else ∞ if $b(c) = \omega$. Note that $P_c(b(c)) = 0$ if $b(c) = c$ is an expandable tip node.

The weight $w(c)$ need not be stored; it is computable from the value of $\tilde{v}_E$ at the c's father and siblings.

We note that $\tilde{v}_E(c)$, $a(c)$, and $\pi(c)$ are computable from the corresponding values for c's children. Specifically, $\tilde{v}_E(c)$ is computable this way by assumption, $a(c)$ is the child d of c which minimizes $\pi(d) + w(d)$, and $\pi(c)$ is $\pi(a(c)) + w(a(c))$. (If all the $b(d_i)$ are ω, then $a(c) = \omega$ and $\pi(c) = \infty$.)

A penalty-based algorithm begins with $E = \{s\}$, $a(s)$, and $\pi(s) = 0$.

At each step of the iterative expansion procedure, the following steps are performed:

Step 1. If $b(s) = \omega$ stop—the tree has no expandable tips (i.e. $E = C$).

Step 2. Set x to s.

Step 3. While $a(x) \neq x$, set $x \leftarrow a(x)$. (Now x is the expandable tip node which minimizes $P(x)$.)

Step 4. Add the sucessors of x to E.

Step 5. Compute $\tilde{v}_E(d)$ for all successors d of x. For each expandable child d of x, initialize $a(d)$ to d and $\pi(d)$ to 0. For each terminal child d of x, initialize $a(d)$ to be ω and $\pi(d)$ to be ∞.

Step 6. Recompute $\tilde{v}_E(x)$, $a(x)$, and $\pi(x)$ from the corresponding values at x's children.

Step 7. If $x = s$ stop, otherwise set $x = f(x)$ and go back to Step 6.

When the algorithm terminates in the last step, then it has traced a path from the root s down to the best expandable tip $x = b(s)$ in E_s, added all the successors of x to E, and updated the $\tilde{v}_E$, a, and π values where necessary by a traversal back up the tree from x to the root s.

3.5. Searching by min/max approximation

The "min/max approximation" heuristic is special case of the penalty-based search method, where the penalties are defined in terms of the derivatives of the approximating functions.

Consider the partial game tree of Fig. 2. Here we have a tree E of size 5; we assume all the tips of E are expandable. The value $\hat{v}_E(c)$ is given at each node c, based on the estimates $\hat{v}(t) = 2$, $\hat{v}(w) = 10$, $\hat{v}(x) = 12$.

If we expand at w (say, because that is the most promising line of play), we may obtain Fig. 3.

Note that $\hat{v}_E(w)$ has changed from 10 to 11.

Which node should be expanded next? In the conventional framework this question is difficult to answer.

To answer this question we propose picking a relatively large p (e.g., $p = 10$), and computing an approximation $\tilde{v}_E(c)$ to $\hat{v}_E(c)$ for each $c \in E$ by the analog to equation (9) wherein "max" has been approximated by "M_p", and "min" has been approximated by "M_{-p}":

$$\tilde{v}_E(c) = \begin{cases} \hat{v}(c), & \text{if } c \in T(E), \\ M_p(\tilde{v}_E(d_1), \ldots, \tilde{v}_E(d_k)), & \text{if } c \in Max \backslash T(E), \\ M_{-p}(\tilde{v}_E(d_1), \ldots, \tilde{v}_E(d_k)), & \text{if } c \in Min \backslash T(E), \end{cases} \tag{10}$$

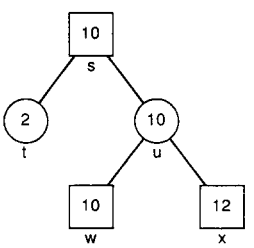

FIG. 2. A partial game tree.

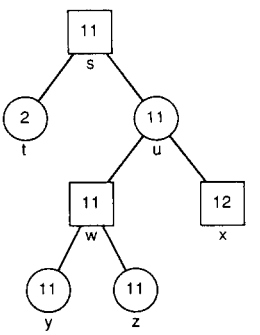

FIG. 3. An expanded game tree.

where $S(c) = \{d_1, \ldots, d_k\}$.[2] The values $\tilde{v}_E(c)$ should be good approximations to $\hat{v}_E(c)$ if p is sufficiently large.

However, we are less interested in the approximate values than we are in their derivatives with respect to each argument. In this manner we will be able to derive the requisite penalties. (These will be the negatives of the logarithms of the derivatives.)

Let

$$D(x, y) = \frac{\partial \tilde{v}_E(x)}{\partial \tilde{v}_E(y)} \tag{11}$$

where y is any node in E_x, the subtree of E rooted at x. Thus $D(s, c)$ measures the sensitivity of the root value $\tilde{v}_E(s)$ to changes in the tip value $\tilde{v}_E(c)$.

We wish to expand next that expandable tip c with largest value $D(s, c)$. We hope thereby to reduce the uncertainty in $\tilde{v}_E(s)$ in the most efficient manner.

The idea of choosing the tip c with maximum $D(s, c)$ can be formulated as a penalty-based iterative heuristic. We define the weight on the edge between $f(x)$ to x to be

$$w(x) = -\log(D(f(x), x)) . \tag{12}$$

By the chain rule for derivatives, we have

$$D(s, x) = \prod_{c \in A(x)} D(f(c), c) . \tag{13}$$

Since we want to expand the expandable tip x with the *largest* $D(s, x)$, we should choose the expandable tip with the *least* penalty, since the penalties are defined by

$$P_s(x) = \sum_{c \in A(x)} w(c) , \tag{14}$$

so that the tip x with largest value $D(s, x)$ is the one with least total penalty $P_s(x)$.

We now redraw Fig. 2 as Fig. 4, using our approximations with $p = 10$. Each node is labelled inside with $\tilde{v}_E(c)$. Each edge from a configuration c to one of its children d is labelled with $w(d)$. Below each tip c of the tree is written the penalty $P_s(c)$ in brackets. Each such $P_s(c)$ is the sum of the weights $w(d)$ on the edges between c and the root.

[2] We must assume from here on that $\hat{v}(c) > 0$ for all c. This does not affect the structure of the game, since we can add a constant δ to all static values $\hat{v}(c)$ if necessary. However, it does preclude our using the "negamax" formulation of games [6].

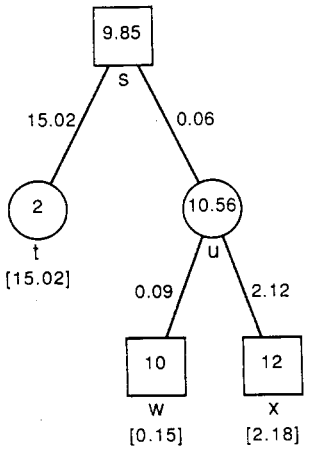

FIG. 4. Game tree with min/max approximations.

According to our rule, w is the best node to expand, since the value at the root depends most strongly on the value at w.

We note that our $\tilde{v}_E$ and w satisfy the computability requirements needed for the penalty-based implementation described above.

We now redraw Fig. 4 as our new Fig. 5, where $\hat{v}_E(c)$ is drawn inside node c as before, and the pair $[a(c), \pi(c)]$ is placed to the right of node c. Similarly, Fig. 6 is our revised version of Fig. 5, after w has been expanded. In both figures the edges are labelled with the $w(c)$ values as in Fig. 4 for the reader's convenience, although these do *not* need to be explicitly stored.

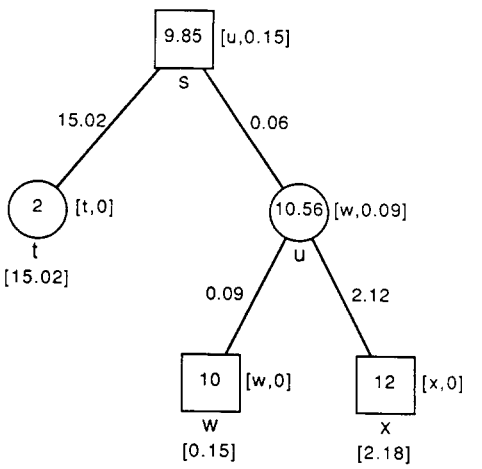

FIG. 5. Game tree with a and π values.

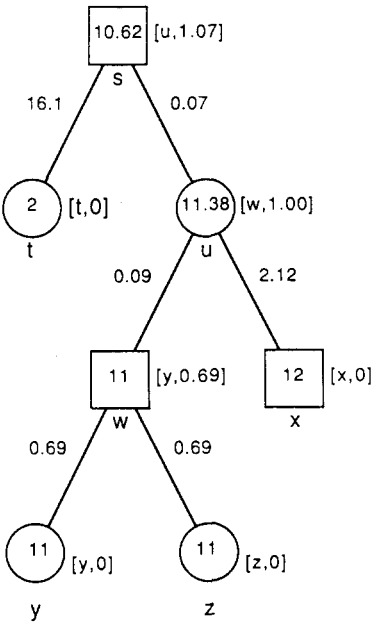

FIG. 6. Expanded game tree with a and π values.

We see from Fig. 6 that tip y is next to be expanded. Here we are still following the most promising line of play for both players. Soon, however, the search will return to expand node x, since $P_s(x) = 1.34$ in Fig. 6 is not much larger than $P_s(y) = 1.07$. As the nodes under w are expanded and the tree under w gets larger, the dependence of $\tilde{v}_E(u)$ on any leaf in w's tree will diminish, and eventually x will be selected as the next tip to be expanded.

4. Implementation

To implement our min/max approximation idea, one must decide how to cope with the computational difficulty of computing the generalized *p-means*.

One can make the argument that one should just "bite the bullet" and compute the generalized means as specified above, in spite of the large computational cost involved in taking powers and roots. This would give the most "accurate" result.

Computing the generalized means exactly might also allow improved play in another manner. After the search of the game tree is completed, a move from the root needs to be selected and played. It is most natural to do this by first computing the backed-up estimate $\hat{v}_E(s)$ at the root. However, it may actually be better to use the min/max approximation $\tilde{v}_E(s)$, i.e. selecting the child of the root with maximum $\tilde{v}_E(c)$ instead of the child with maximum $\hat{v}_E(c)$. This

has the potential for improved play since the min/max approximation will favor a move whose min/max value can be achieved in several ways over a move whose min/max value can be achieved in only one way. (For example, note that $M_{10}(35, 40) > M_{10}(30, 40)$ although $\max(35, 40) = \max(30, 40)$.) The min/max approximation pays attention to good backup or secondary moves.

Another approach is to skip the computation of the generalized mean values altogether, and use the appropriate min or max values instead. (I.e. use $\hat{v}_E$ instead of $\tilde{v}_E$ everywhere.) Since the generalized mean values are intended to approximate the min and max functions anyway, this may not introduce very much error. The main point of using the generalized mean values was for their derivatives, not for the values themselves. We call this variation the "reverse approximation" idea.

We note that with the "reverse approximation" idea, the weight $w(c)$ is computable from equation (6) as

$$w(c) = \log(n) + (p-1) \cdot (\log(\hat{v}_E(d)) - \log(\hat{v}_E(c))) \tag{15}$$

where c has n siblings, and where d is that sibling of c with the most favorable value $\hat{v}(d)$ for the player to move from $f(c)$. Here we might view the $\log(n)$ term as that portion of the penalty associated with descending another level in the tree, and the remaining terms as the penalty associated with making less than optimal moves. If the static evaluation function always returns integers in some range, then each $v_E(c)$ is an integer in the same range, and each $w(c)$ can be computed using table-lookups for the logarithm computations. Our implementation (described later) uses this idea, except that the value $\log(n)$ is replaced by a constant.

It is not clear how to "optimally" choose the parameter p. As p gets large, the heuristic should grow very deep but narrow trees. For small p, the heuristic should grow rather broad trees. (For $p = 1$, it would grow the tree in more or less of a breadth-first manner, since all derivatives at the same level would be close.) One could conceivably use different p values at different levels of the tree. Choosing a large value of p corresponds to having a high degree of confidence in the accuracy of the values returned by the static evaluator, while a small p corresponds to a low degree of confidence.

5. Experimental Results

In this section we present some initial experimental results demonstrating that our approach can produce play superior to that produced by minimax search with alpha-beta pruning, for the same number of calls to the underlying "move" operator. However, when CPU time rather than calls to the move operator is the limiting resource, minimax search with alpha-beta pruning seems to play better.

5.1. The game: Connect-Four

We chose the game of *Connect-Four* as a basis for our experiments because it is commercially available and well known, yet simple to describe and implement.

The game equipment consists of a plastic frame, together with sets of red and black plastic tokens. The frame has a 6×7 array of cells arranged vertically; there are 7 columns of 6 cells each. It is possible to drop a token in at the top of any column; it then falls down to the lowest unoccupied cell in the column and is visible to both players.

A legal move consists of dropping a token into any column which contains at least one unoccupied cell.

Black moves first; after that the players alternate turns.

The first player to create a line of four tokens of his color in a row wins the game. This line may be horizontal, vertical, or diagonal. It is possible to have a tied game.

5.2. The static evaluator

In our implementation the *move* and *unmove* operators also implemented the static evaluation function. This static evaluation function was used by all game-playing strategies, so that differences in playing ability would not be due to differences in the static evaluators.

By convention, Black is Max and Red is Min. The static evaluator returned integers in the range 1 to 1023 where 1 is reserved to denote a win by Red and 1023 is reserved to denote a win by Black. The value 512 thus denotes a middle or neutral value.

To describe the static evaluation function, we define a "segment" to be set of four cells in a line (i.e. where a winning four-in-a-row can be placed). The *score for a segment* is defined to be zero if the segment contains no tokens or tokens of both colors. Otherwise it depends on the number of tokens present and their color. For a segment containing only black tokens: one black token scores 1, two black tokens scores 10, and three black tokens score 50. If the tokens are red the signs are reversed.

The static evaluation for a non-winning position is the sum of three components:

(1) The neutral value 512.

(2) A "move bonus" of 16 the player whose turn it is to play. (i.e. 16 for Black, or -16 for Red).

(3) The sum, over all possible segments, of the score for that segment.

This sum is truncated if necessary to keep it in the range 2 to 1022.

5.3. Resource bounds

In our playoffs, each strategy was allocated a fixed amount of resources to use

in computing its move. This was a fixed bound per turn; a strategy could not save on the computation for one turn and use it later.

Two different resource bounds were used: elapsed CPU time (measured in seconds), and calls to the basic "move" subroutine (measured in thousands of calls). Note that for the move bound, the min/max heuristic must explicitly pay for moving down the tree from the root every time.

Since the game is relatively small, the resource bounds we used are also rather modest.

The implementation was done in C on a DEC MicroVax workstation; the experiments were run in parallel overnight on ten such workstations.

5.4. Minimax search with alpha-beta pruning

The implementation of minimax search with alpha-beta pruning was relatively straightforward. The usual depth-first search with a depth bound was employed. A depth bound of two ply was initially searched, and then the search was repeatedly restarted with a larger depth boundary until the time or move bound was reached (i.e. iterative deepening was used). Then the result of the last *complete* search was used as alpha-beta's move. (The inefficiency due to throwing away this last partial search we call "fragmentation inefficiency".) Typical search depths ran from 6 ply to more than a dozen near the end of the game. No information was carried over from a search at one depth to the search at the next. The children of a node were searched in order of their static evaluations, best-first.

5.5. Penalty-based heuristic

Our implementation of the min/max heuristic worked as follows:

(1) The 'reverse approximation" was used; the value computed for a node was its true backed-up min/max value, based on the tree computed so far.

(2) The penalty on an edge was computed to be 0.05 plus the absolute value of the difference between the natural logarithm of the value of the node and the natural logarithm of the value of his "best" sibling (the one with the best backed-up score, as viewed from the point of view of the person making the choice).

The constant 0.05 was chosen on the basis of earlier preliminary testing. It must be admitted that the performance of the scheme was sensitive to this constant; further search is needed to make the computation of penalties more robust.

5.6. Results

For each experiment, we considered 49 different starting positions. Each starting position was defined by specifying the first two moves of the game.

TABLE 2. Experimental results

Resource bound per turn	MM wins	AB wins	Ties
1 second	41	46	11
2 second	40	42	16
3 seconds	36	44	18
4 seconds	39	52	7
5 seconds	30	55	13
Total	186	239	65
1000 moves	47	35	16
2000 moves	50	35	13
3000 moves	42	47	9
4000 moves	49	42	7
5000 moves	61	31	6
Total	249	190	51

(There are exactly seven opening moves, and seven responses.) For each starting position, two games were played—one with alpha-beta (AB) moving first, and one with min/max approximation (MM) moving first. Thus a complete experiment consists of 98 games. For each experiment, it was recorded how many times each strategy won, and how many ties occured. One experiment was run for each of five posible time bounds (1 second to 5 seconds, in one-second intervals), and for five possible move bounds (1000 moves to 5000 moves, in 1000-move increments). Thus, 490 games were played for each resource bound, and 980 games played altogether.

We see that based on time usage alone, alpha-beta seems to be superior to our implementation of the min/max approximation approach.

However, if we base our comparison on move-based resource limits, the story is reversed: min/max approximation is definitely superior.

We note that the number of *distinct* positions considered by alpha-beta was approximately three times larger than the number of distinct positions considered by min/max when a time bound was in effect. When a move bound was in effect the number of distinct positions considered by each strategy was roughly equal; the fragmentation lossage of alpha-beta seemed to equal the inefficiencies of the min/max routine having to redescend the tree for each expansion.

We note that our implementation of minimax search with alpha-beta pruning called the move operator approximately 3500 times per second, while our implementation of the min/max heuristic called the move operator approximately 800 times per second. When special-purpose hardware is used, or when the move operator is expensive to implement, the move-based comparison would be more relevant. For software implementations more development of the min/max approach is needed to reduce the computational overhead per call to the move operator.

Overall, we find these experimental results very encouraging.

We note, as Nau [9] points out, that there are many subtle methodological questions that arise when trying to compare heuristics by playing them off against each other, since the "evenness" of play or variation in quality of play during different portions of the game can have a dramatic influence on the results.

This completes our description of the initial experiments that have been performed demonstrating the quality of play produced by our heuristic. Of course, further empirical validation of these ideas is needed, and the approach needs to be refined and made more efficient. Since our idea is relatively new, one may expect that further development and optimizations may occur that could improve its competitiveness even further.

6. Discussion

We first discuss some of the general features of penalty-based schemes.

First, we note that penalty-based schemes—like all iterative schemes—requires that the tree being explored be explicitly stored. Unlike depth-first search schemes (e.g. minimax search with alpha-beta pruning), penalty-based schemes may not perform well unless they are given a large amount of memory to work with.

Second, we note that the penalty-based schemes are oriented towards improving the value of the estimate $\tilde{v}_E(s)$ at the root, rather than towards selecting the best move to make from the root. For example, if there is only one move to make from the root, then a penalty-based scheme may search the subtree below that move extensively, even though such exploration can't affect the decision to be made at the root. By contrast, the B^* algorithm [2], another iterative search heuristic, is oriented towards making the best choice, and will not waste any time when there is only one move to be made.

Third, the penalty-based schemes as presented require that a tip be expanded by generating and evaluating *all* of the tip's successors. Many search schemes are able to skip the evaluation of some of the successors in many cases.

Fourth, we note that penalty-based schemes may appear inefficient compared to depth-first schemes, since the penalty-based schemes spend a lot of time traversing back and forth between the root and the leaves of the tree, whereas a depth-first approach will spend most of its time near the leaves. We imagine that the penalty-based schemes could be adapted to show similar efficiencies, at the cost of not always selecting the globally least-penalty tip to expand. The algorithm would be modified in Step 7 to ascend to its successor some of the time and to *redescend* in the tree by returning to Step 3 in the other cases. (However, if $b(x) = \omega$ the algorithm must ascend.) The decision to redescend may be made probabilistically, perhaps as a function of the depth of

x, or the change noted so far in $\pi(x)$. For example, one might continue to redescend from the node x found in Step 3 until the number of leaves in E_x exceeds the depth of x. We have not explored these alternatives.

Finally, we observe that penalty-based schemes do spend some time evaluating non-optimal lines of play. However, the time spent examining such lines of play decreases as the number of non-optimal moves in the line increases, according to the weights assigned to those non-optimal moves.

We see how our "min/max approximation" heuristic will allocate resources in a sensible manner, searching shallowly in unpromising parts of the tree, and deeper in promising sections. We might also call this approach the "decreasing derivative heuristic," since the nodes are expanded in order of decreasing derivative $D(s, x)$.

It is important to note that the efficiencies exhibited by alpha-beta pruning can also appear with our scheme. Once a move has been refuted (shown to be non-optimal), its weight will increase dramatically, and further exploration down its subtree will be deferred. However, this depends on the static evaluator returning meaningful estimates. If the static evaluator were to return only constant values except at terminal positions, our scheme would perform a breadth-first search. (We observe that the scheme of McAllester [7] performs like alpha-beta search in this case.)

Other penalty-based schemes are of course possible. We note two in particular:

(1) If we can compute an estimate $p(c, d)$ that the actual play would progress from configuration c to successor configuration d, given that play reaches c, then we can define the weight $w(d)$ to be $-\log(p(c, d))$. With this definition, the tip node to be expanded next is the tip node estimated to be *most likely* to be reached in play. This idea was originally proposed by Floyd (see [6]), although it does not seem to have been seriously tried.

(2) If we estimate for each node c the probability that c is a forced win for Max (see [9] for discussions of this idea) then we can select the tip node to expand upon which our estimate at the root depends most heavily. This can be done, in a manner similar to our min/max approximation technique, beginning with the formulas in [9]. This idea was suggested by David McAllester.

7. Open problems

(1) How should one best choose which generalized mean value functions, or penalty functions, to use?

(2) Can our ideas be combined effectively with more traditional approaches? In particular, what is the best way to blend the efficiency of depth-first search with our ideas?

(3) How well can these ideas be parallelized? (See [1] for a fascinating discussion on how to parallelize the alpha-beta heuristic.)

(4) How well does our approach work in games where sacrifices are important? (Connect-Four seems not to be such a game.) Will useful sacrifice plays be discovered?

(5) How sensitive is our approach to "noise" in the static evaluation function, compared to traditional approaches, in terms of the resulting quality of play?

(6) How well does the min/max approximation scheme work on pathological games?

8. Conclusion

We have presented a novel approach to game tree searching, based on approximating the min and max functions by suitable generalized mean-value functions. Experimental results indicate that our scheme outplays alpha-beta with iterative deepening, when both schemes are restricted to the same number of calls to the move operator.

ACKNOWLEDGMENT

I would like to thank Hans Berliner, Charles Leiserson, and David McAllester for some stimulating discussions and comments. Kai-Yee Ho and Mark Reinhold helped to produce the experimental results. The referees made numerous valuable suggestions on the presentation of these results.

REFERENCES

1. Baudet, G., The design and analysis of algorithms for asynchronous multiprocessors, Computer Science Tech. Rept. CMU-CS-78-116, Carnegie-Mellon University, Pittsburgh, PA, 1978.
2. Berliner, H., The B* tree search algorithm: A best-first proof procedure, *Artificial Intelligence* **12** (1979) 23–40.
3. Barr, A. and Feigenbaum, E.A. (Eds.), *The Handbook of Artificial Intelligence* **1** (Kaufmann, 1981) Section II.C.
4. Campbell, M.S. and Marsland, T.A., A comparison of minimax tree search algorithms, *Artificial Intelligence* **20** (1983) 347–367.
5. Hardy, G.H., Littelwood, J.E. and Polya, G., *Inequalities* (Cambridge University Press, Cambridge, U.K., 1934).
6. Knuth, D.E. and Moore, R.W., An analysis of alpha-beta pruning, *Artificial Intelligence* **6** (1975) 293–326.
7. McAllester, D.A., A new procedure for growing min-max trees, *Artificial Intelligence*, to appear.
8. Nau, D.S., An investigation of the causes of pathology in games, *Artificial Intelligence* **19** (1982) 257–278.
9. Nau, D.S., Pathology on game trees revisited, and an alternative to minimaxing, *Artificial Intelligence* **21** (1983) 221–244.
10. Nau, D.S., Purdom, P. and Tzeng, C.-H., An evaluation of two alternatives to minimax, in: *Proceedings on Uncertainty and Probability in Artificial Intelligence*, University of California, Los Angeles, CA (1985) 232–235.

11. Nilsson, N.J., Searching Problem-solving and games-playing trees for minimal-cost solutions, *Proc. IFIP* **2** (1968) 1556–1562.
12. Palay, A.J., *Searching with Probabilities* (Pitman, Boston, MA, 1985).
13. Pearl, J., *HEURISTICS: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley, Reading, MA, 1984).
14. Roizen, I. and Pearl, J., A minimax algorithm better than alpha-beta? Yes and no, *Artificial Intelligence* **21** (1983) 199–220.
15. Stockman, G.C., A minimax algorithm better than alpha-beta?, *Artificial Intelligence* **14** (1979) 179–196.

Received August 1986; revised version received April 1987