The *-Minimax Search Procedure for Trees Containing Chance Nodes

Bruce W. Ballard
Department of Computer Science, Duke University, Durham, NC 27706, U.S.A.

Recommended by H.H. Nagel

ABSTRACT

An extention of the alpha-beta tree pruning strategy to game trees with 'probability' nodes, whose values are defined as the (possibly weighted) average of their successors' values, is developed. These '-minimax' trees pertain to games involving chance but no concealed information. Based upon our search strategy, we formulate and then analyze several algorithms for *-minimax trees. An initial left-to-right depth-first algorithm is developed and shown to reduce the complexity of an exhaustive search strategy by 25–30 percent. An improved algorithm is then formulated to 'probe' beneath the chance nodes of 'regular' *-minimax trees, where players alternate in making moves with chance events interspersed. With random ordering of successor nodes, this modified algorithm is shown to reduce search by more than 50 percent. With optimal ordering, it is shown to reduce search complexity by an order of magnitude. After examining the savings of the first two algorithms on deeper trees, two additional algorithms are presented and analyzed.*

1. Introduction

Many games involving chance events, such as the roll of dice or the drawing of playing cards, can be modeled by introducing 'probability' nodes into standard *minimax* trees. In this paper, we shall use the symbols + and − to denote maximizing and minimizing nodes, respectively, and * (pronounced 'star') to denote a probability node. We define the *value* of a *node as the weighted average of the values of its successors, which may occur with differing probabilities. A sample '*-minimax' tree, as we shall call trees made up of +, − and *nodes, appears in Fig. 1. Backed-up values for non-terminal nodes are shown in parentheses. The value of the *node, whose successors have been assumed to be equally likely, has been computed as $\frac{1}{2}(2-4) = -1$.

*This research has been supported in part by AFOSR, Air Force Command, AFOSR 81-0221.

Artificial Intelligence 21 (1983) 327–350
0004-3702/83/$3.00

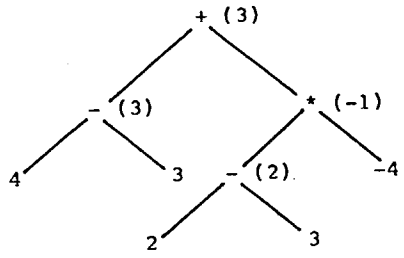

FIG. 1. A sample *-minimax tree.

In this paper we shall develop a search strategy for *-minimax trees, then describe and analyze several algorithms based upon it. Our algorithms reduce to the familiar alpha-beta procedure [2] for degenerate *-minimax trees, i.e. those with only + and −nodes. Readers unfamiliar with ordinary minimax trees should refer to Section 3 and perhaps consult Nilsson [1] or any of [2–5]. To facilitate analysis, we shall assume that all descendents of a *node are equally likely. The algorithm we present can be extended, in a direct way, to the more general case.

For the most part, *-minimax trees retain the properties of ordinary minimax trees. In particular, they pertain to 2-person, 0-sum, perfect information games. By 'perfect information' we mean that neither player conceals information about the current state of the game, or possible future states, that is useful to him and that would be useful to the other player. Many dice games (e.g. craps, backgammon, and board games such as monopoly) satisfy these criteria, as do some card games (e.g. casino blackjack).

Unlike ordinary minimax trees, where +nodes always lead to −nodes and vice versa, trees for *-minimax games exhibit many forms. For instance, the top portion of a tree for casino blackjack, where the strategy of the dealer ('house') is predetermined, thus eliminating branches beneath −nodes, is given (in simplified form) in Fig. 2. Compare the structure of this tree fragment, with its notable absence of alternation between + and −nodes, with the backgammon tree fragment of Fig. 3.

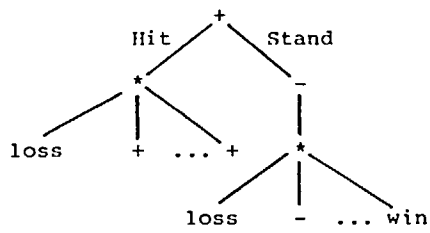

FIG. 2. Portion of a casino blackjack tree.

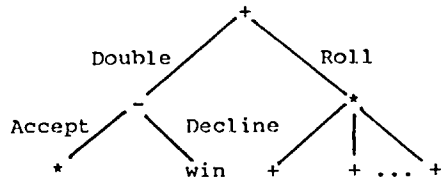

FIG. 3. Portion of a backgammon tree.

2. The *-Minimax Search Problem

Having defined and given examples of *-minimax trees, we now consider the question of searching these trees. At the very least, we want to retain the alpha-beta 'cutoff' power of ordinary minimax trees. However, the presence of *nodes provides opportunities for additional forms of cutoffs. Our strategy is based on the fact that *lower and upper bounds on the value of a *node can be derived by exploring one or more of its children.* Our search algorithm will (indirectly) associate such lower and upper bounds with each *node. Since alpha and beta values will have been passed into a *node, we can discontinue search below it if the lower *-bound ever exceeds beta, or if the upper *-bound ever becomes less than alpha. In the former case, the −player will have already found a path that holds his opponent to less than the lower limit of the *node value. In the latter case, + will have already found a way to do better than the upper limit of the *node value. Thus, optimal play by both players will assure that the *node in question is never reached, rendering further exploration beneath it futile.

As an example of a possible '*cutoff', suppose the (leaf) values of a particular tree are integers between 0 and 10, inclusive, and that a *node with 4 equally likely successors has had 2 of its successors searched. This situation is shown in Fig. 4. Knowing the values of these 2 children, we can say that the *smallest* value subsequent search can assign to the *node is $\frac{1}{4}(5+3+0+0)$ or 2. Similarly, the *greatest* possible value of the *node is $\frac{1}{4}(5+3+10+10)$ or 7. Thus, a cutoff can occur if the alpha value passed to * is ≥ 7, or if the beta value is ≤ 2. We shall formulate a search strategy to take advantage of this form of potential cutoff. In addition, our strategy will compute *new* alpha and beta values for use *below* *nodes.

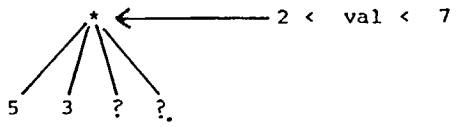

FIG. 4. Interim bounds on a *node.

3. A Strategy for Searching *-Minimax Trees

Before looking at ways in which *cutoffs can occur, it will be useful to recall the circumstance in which nodes of an ordinary minimax tree can be pruned, without loss of accuracy. At each +node, the +player will choose the successor with the *highest* value, while at −nodes the −player will choose the successor with the *smallest* value. Consider Fig. 5. Having determined the value (either terminal or backed-up) of the node below move *a* as 4, move *b* is 'refuted' by move *x*, which establishes that the value of move *b* is no more than 3 (perhaps less). Move *y* may be 'cut off' since its value (indicated by '?') has no bearing on either the value of the root of the tree or on deciding the best move from the root. Since the node with ? as value could be the root of a sizable subtree, the searching of many thousands of nodes may have been eliminated.

The standard method of having a search algorithm recognize opportunities for cutoffs such as these is to associate so-called 'alpha' and 'beta' values with each node *n* of the tree. The *alpha* value tells how well + can do if node *n* is encountered during optimal play by both players. Similarly, the *beta* value tells how badly − can make + do, again assuming perfect play reaches node *n*. If the value of a node is ever determined to *exceed* its *beta* value, or to be *less than* its *alpha* value, this must mean that optimal play will not lead to the node in question. Therefore, further searching is pointless. In fact, subsequent nodes can be cut off if a successor value *equals* the alpha or beta value. In this case, one of the players will have found another line of play, with at least as good a value for him, which has already been searched. If a player has two or more equally good moves, it doesn't matter which one is made.

Consider now the partially searched *-minimax tree of Fig. 6, and assume that leaf values range from −10 to +10, inclusive. The alpha-beta values of 4 and 5 for the *node at depth 2 indicate that if the value of this node is determined to be ≤ 4, or ≥ 5, search beneath it can be discontinued. Suppose now that, as shown, there are 3 successors, the first of which has been searched and found to have a value of 2. If −10 and +10 are limits on leaf values, then the value of the *node lies between $\frac{1}{3}(2-10-10)$ and $\frac{1}{3}(2+10+10)$, i.e. between −6 and $7\frac{1}{3}$. Since −6 is not greater than 5, nor is $7\frac{1}{3}$ less than 4, we must continue searching children of the *node. However, before so doing, we ask ourselves, *what values of the lower-level node to be searched will entail a cutoff*

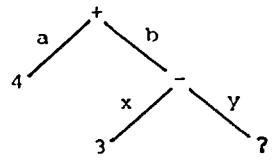

FIG. 5. A conventional minimax cutoff.

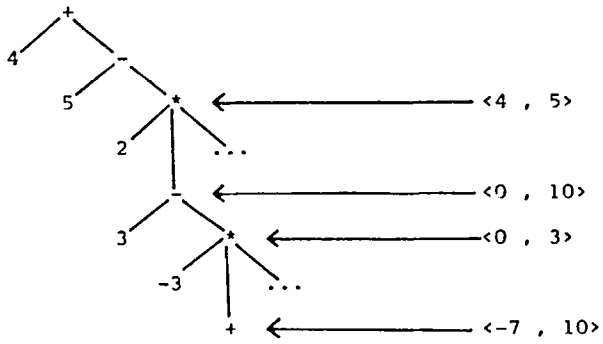

FIG. 6. A partially searched *-minimax tree.

*at the *node*?* Denoting this value by V, we want to know V for which

$$\tfrac{1}{3}(2+V+10)\leqslant 4, \quad \text{i.e.} \quad V\leqslant 0$$

or

$$\tfrac{1}{3}(2+V-10)\geqslant 5, \quad \text{i.e.} \quad V\geqslant 23.$$

These values for V can now be used as alpha-beta values for the lower-level node to be searched. Having assumed 10 as an upper bound on game values, however, we will use 10 rather 23 as the beta value. Thus, as indicated in Fig. 6, 0 and 10 serve as alpha-beta values for the −node at depth 3.

Suppose now that we search the first descendent of this −node at depth 3 and find a value of 3. Since 3 is not less than the 0 alpha value, we continue searching, and the next node to be searched is the *node at depth 4. However, since 3 is less than the current beta value of 10, we pass down 3 as the new beta value for the *node, while the alpha value of 0 is unchanged.

Having reached the *node at depth 4, we can discontinue searching if its value is found to be less than or equal to 0, or greater than or equal to 3. In the latter case, − will have already found a way to hold his opponent to 3, and surely won't give + the chance to achieve 3 or better at the *node. In the former case, a cutoff below the *node will be followed immediately by a cutoff at the parent −node, which will immediately cause a cutoff at its parent *node, which in turn will entail a cutoff below the top −node. This possibility for two or more cutoffs to occur without intervening leaf searches is without counterpart in conventional minimax trees.

By reasoning as above, we can determine, after seeing the −3 successor of the *node at depth 5, that the *node value lies between $-7\frac{2}{3}$ and $5\frac{2}{3}$, and that −7 and 10 should serve as alpha-beta values for the +node to be searched next.

4. An Algorithm for Searching *-Minimax Trees

We now formalize the reasoning presented above. Let L and U denote lower

and upper bounds on all possible game (leaf) values. Let $V_1, V_2, \ldots, V_N$ be the values of the N successors of a *node, whose ith successor is about to be searched. After returning from the ith node, a cutoff will occur if

$$\frac{(V_1 + \cdots + V_{i-1}) + V_i + U*(N-i)}{N} \leqslant \text{alpha} \tag{1a}$$

or if

$$\frac{(V_1 + \cdots + V_{i-1}) + V_i + L*(N-i)}{N} \geqslant \text{beta}\,. \tag{1b}$$

Letting A_i represent the alpha value for the ith successor, we have

$$A_i = N*\text{alpha} - (V_1 + \cdots + V_{i-1}) - U*(N-i) \tag{2a}$$

where 'alpha' denotes the alpha value of the present *node. Similarly, letting B_i represent the new beta value, we have

$$B_i = N*\text{beta} - (V_1 + \cdots + V_{i-1}) - L*(N-i) \tag{2b}$$

where 'beta' is the beta of the *node. In the actual implementation, we will want to assure that all A's are $\geqslant L$ and all B's are $\leqslant U$. From the equations above we see that up-to-date A and B values can be computed efficiently if they are initialized as

$$A_1 = N*(\text{alpha} - U) + U, \tag{3a}$$

$$B_1 = N*(\text{beta} - L) + L \tag{3b}$$

and updated by

$$A_{n+1} = A_n + U - V_n, \tag{4a}$$

$$B_{n+1} = B_n + L - V_n\,. \tag{4b}$$

Note that, when $N = 1$, A_1 and B_1 take on the alpha-beta values themselves.

From the above formulation we derive the following search procedure for *nodes:

```
Star1(board, alpha, beta)
{
 local A, B, i, v, vsum, AX, BX, s[];

 determine the N successors s1, s2, ..., sN
 if (N = = 0)
   return(Term(board));
 A = N * (alpha - U) + U;
 B = N * (beta - L) + L;
 vsum = 0;
 for (i = 1; i < = N; i + +) {
     AX = max(A, L);
```

```
    BX = min(B, U);
    v = Eval(s[i], AX, BX);
    if (v < = A)
      return(alpha);
    if (v > = B)
      return(beta);
    vsum = vsum + v;
    A = A + U - v;
    B = B + L - v;
  }
  return(vsum/N);
}
```

This code makes use of (1) a Term procedure, to evaluate terminal positions; (2) an Eval procedure which, depending on which player is to move next, invokes either Max or Min; and (3) a procedure to *generate* the successors of a node.

5. A 'Better' Algorithm for 'Regular' *-Minimax Trees

The strategy developed above assumes that each successor of a *node could be either a − or a +node, independent of its sister nodes. In many *-minimax games, however, most *nodes fall into one of two classes: those with only +successors, and those with only −successors. In these games, chance events are used to determine *legal moves* (as in backgammon), or the *outcome* of a move (as in blackjack), or both, but not to determine who is to have the next move. (If chance is used to decide who makes the very *first* move, this one-time event is unrelated to search matters.) We shall refer to games such as these as *regular* *-minimax games. Trees corresponding to these games alternate between + and −nodes, as do ordinary minimax trees, but with *nodes interspersed. Thus, on a path from the root we encounter node types of *, +, *, −, *, +, *, −, and so forth.

For the newly-defined class of regular *-minimax trees, we can devise a 'better' search procedure, to be called Star2, which is later shown to be greatly superior to the Star1 procedure it directly extends. The algorithm underlying Star1 was based on a strict depth-first control strategy. Thus, if X and Y are successors of some *node, we cannot examine children of X, suspend work with X to begin searching beneath Y, and then later return to additional nodes beneath X. Consider a *node all of whose children are −nodes. In this situation, the left-to-right restriction imposed by a depth-first control strategy has two drawbacks. First, for a given *node, it forces us to look at all N leaves beneath all but the last −node searched (unless some leaf takes on the minimum value over all possible leaves). Second, we assumed that unprocessed −nodes could have the maximum possible game value. Each of these problems is answered by the modification developed below.

If a *node being examined is worse than a previously searched *node, a preliminary 'probing' of just one child of each –node can substantially reduce the number of nodes explored before a cutoff occurs. If W_i denotes the value of some child of the ith –node, and as before V_i denotes the (true) value of the ith –node, we will obtain a cutoff below the *node if

$$\frac{(V_1 + \cdots + V_{i-1}) + V_i + (W_{i+1} + \cdots + W_N)}{N} \leqslant \text{alpha} \tag{5}$$

which yields an A_i value of

$$A_i = \text{alpha} * N - (V_1 + \cdots + V_{i-1}) - (W_{i+1} + \cdots + W_N) \tag{6}$$

which can be efficiently computed by initializing to

$$A_1 = \text{alpha} * N - (W_2 + \cdots + W_N) \tag{7}$$

and updating by

$$A_{n+1} = A_n + W_{n+1} - V_n . \tag{8}$$

Since all W's are $\leqslant U$, the values computed for A in (6) are never less than corresponding values in (2a).

If the *node being searched is not better than all previously searched alternatives, so that a cutoff will occur, then unless we are quite unlucky in selecting the particular children of –nodes to explore, these tighter bounds for A will allow for an earlier cutoff than the formulas given earlier. Since we cannot speak with confidence about how *large* the true value of a –node might be without looking at all its successors, no special use can be made of the B in (2b) during the preliminary probing stage.

We formalize these ideas in the following procedure. In order to detect possible cutoffs during the probing phase, and to avoid a subscript range error the last time through the second loop, the calculations specified by (7) and (8) have been 'distributed' into disjoint places in the code of the new procedure. In keeping with the discussion above, this code pertains to a *node followed by –nodes. A related procedure for *nodes followed by + nodes will also be needed.

```
Star2Min(board, alpha, beta)
{
 local A, B, i, v, vsum, AX, BX, s[ ], w[ ];

 determine the N successors s1, s2, . . . , sN
 if (N = = 0)
   return(Term(board));

 A = N * (alpha – U);
 B = N * (beta – L);
 BX = min(B, U);
```

```
for (i = 1; i < =N; i + +) {
    A = A + U;
    AX = max(A, L);
    w[i] = Probe(s[i], AX, BX);
    if (w[i] < = A)
      return(alpha);
    A = A − w[i];
}
vsum = 0;
for (i = 1; i < =N; i + +) {
    B = B + L;
    A = A + w[i];
    AX = max(A, L);
    BX = min(B, U);
    v = Min(s[i], AX, BX);
    if (v < = A)
      return(alpha);
    if (v > = B)
      return(beta);
    vsum = vsum + v;
    A = A − v;
    B = B − v;
}
return(vsum/N);
}
```

Here we have made calls to (1) a standard Min procedure for −nodes, and (2) a new procedure, Probe(x, y, z), whose job is to return Term(x), if x is a leaf, otherwise to choose some successor s and x, either at random or by appeal to a static evaluation function, and return Min(s, y, z). In the event that preliminary probing fails to obtain a cutoff, the modified algorithm given above reverts to the original algorithm, albeit with a tighter A bound and therefore with an equal or better opportunity for an early cutoff (as shown in Section 6.2.1.3). Rather than exhaustively searching −nodes one by one, however, more elaborate behavior is possible (as described in Section 7).

6. Analysis of *-Minimax Algorithms

In analyzing the efficiency of a search procedure for a class of game trees, one begins by specifying a subclass for study. This involves deciding on (a) the *overall structure* of trees; (b) a way of assigning *values* to leaves; and (c) the *criterion* to be measured. We want our analysis of *-minimax search to resemble that for ordinary alpha-beta wherever possible (e.g. [2–5]). Since virtually all study of alpha-beta has chosen to count the number of leaves encountered as the efficiency measure, we will do so as well. Furthermore, most analyses of ordinary alpha-beta have considered so-called 'complete' N-ary trees, where all leaves occur at a fixed depth D and all nonterminal

nodes have exactly N successors. We define the class of **-complete N-ary trees* by inserting a *node above each node of a complete N-ary tree, and giving these *nodes $N-1$ additional successor nodes of the same type. These trees satisfy the definition of 'regular' *-minimax trees as given in the previous section. Corresponding to a complete N-ary tree of depth D, which has $N ** D$ leaves, is a *-complete N-ary tree of depth $2D$ having $N ** 2D$ leaves. We will investigate the efficiency of the *-minimax algorithms on *-complete N-ary trees of depth 3, since they correspond to minimax trees (of depth 2) allowing the simplest cutoffs of standard alpha-beta (see Fig. 5). The leftmost part of a *-complete 2-ply binary tree appears in Fig. 7.

From Fig. 7 we can see that no cutoff is possible at the topmost level, since the root is a *node. (During actual play, the chance event will have occurred by the time we are ready to select a move, so only one +successor of the top *node will be searched anyway.) The left +node however permits a cutoff below its second successor. By the time this *node has been reached, an alpha value will be available. We will consider this critical portion of the tree, which is given in Fig. 8.

In an attempt to capture the sorts of leaf dependencies that have been observed in practice, we follow Fuller, Gaschnig and Gillogly [3] by assigning distinct, uniformly spaced values to the arcs below a node, and defining a leaf value as the sum of the arc values on the path to it from the root. Since we want our methods to apply to (perhaps differently shaped) trees where leaves occur at various levels, we cannot use simply the values of 1 through N as arc labels, as Fuller et al. [3] did. To enable all successors of a node to share the same a priori probability of being best, we want the arc value from a + or −node to its 'best' successor to be 0, and the 'average' arc value out of a *node to be 0. Therefore, to the arcs of −nodes we assign values of $0, 1, \ldots, N-1$, and to the arcs of +nodes we assign values of $0, -1, \ldots, -(N-1)$. Assuming N is even, we assign to the arcs of *nodes the values of $-\frac{1}{2}N, \ldots, -2, -1, 1, 2, \ldots, \frac{1}{2}N$.

Readers familiar with studies of the alpha-beta procedure [2–5] will ap-

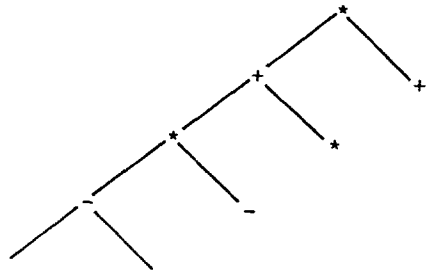

FIG. 7. The leftmost portion of a 2-ply *-complete binary tree.

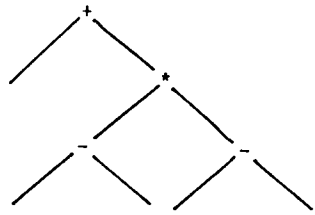

FIG. 8. Opportunities for a cutoff in a *-complete 2-ply binary tree.

preciate the algebraic complexity involved in obtaining precise closed-form performance figures for even shallow trees (e.g. depth 3). Accordingly, the following analysis, which determines both asymptotic complexity and exact values for various branching factors, combines simulation with analytic techniques where appropriate.

6.1. Best-case analysis

We first establish the best case behavior of the Star1 and Star2 procedures on the class of *-complete trees just defined. We will derive asymptotic values in closed form, then present exact figures for various branching factors arrived at by empirical means.

As in ordinary alpha-beta search, the first successor of the +node must be fully searched, and we will obtain the greatest number of cutoffs if the *best* node is searched first. Since this value, which we have arranged to be 0, cannot be improved upon, it will serve as alpha value for all the remaining *nodes, which have backed-up values of $-1, -2, \ldots, -(N-1)$ and can be searched in any order without affecting search efficiency. Since the node with -1 as value is almost best, it will take longest to search, while the $-(N-1)$ node will be dispensed with most quickly.

We will find the number of leaves searched for the particular *node having backed-up value of $w - \frac{1}{2}N$ (a value chosen to simplify the algebra). From this result we add up values as w runs from $-(\frac{1}{2}N - 1)$ to $\frac{1}{2}N - 1$ to obtain the overall search efficiency under the +node in question.

6.1.1. *Best-case analysis of the Star1 procedure*

To obtain a cutoff below a *node, we will need to begin exploring a jth descendent of it for which

$$V_1 + V_2 + \cdots + V_j + U*(N-j) \leq 0 \tag{9}$$

where V_i denotes the backed-up value of the ith descendent of the *node and 0 is the active alpha value. The values of the $-$nodes beneath * range from $w - N$ to w, excluding $w - \frac{1}{2}N$. To guarantee the earliest possible cutoff, we will want to look first at the nodes with *low* values. If $j > \frac{1}{2}N$, which we shall see below is

always true, since the smallest j will be about $0.55N$, then V_1 through V_j will take on the values $w-N$ through $w-(N-j)$, excluding $w-\frac{1}{2}N$. Letting $k = N-j$, the number of nodes which need not be searched if a cutoff occurs at node j, we can substitute values for V_1 through V_j into the equation above and then multiply each side by -1 to obtain

$$[k+(k+1)+\cdots+N]-\tfrac{1}{2}N+(k-N)*w \geq k*U. \tag{10}$$

But the summation is easily written in closed form, and U (the maximum leaf value) is $0+\frac{1}{2}N+(N-1)$, or $\frac{3}{2}N-1$. After cancelling the $\frac{1}{2}N$ terms, we have

$$\tfrac{1}{2}N*N-\tfrac{1}{2}(k*k-k)+(k-N)*w \geq k*(\tfrac{3}{2}N-1) \tag{11}$$

which can be written as a quadratic (in k) as

$$k*k+(3N-2w-3)*k-(N*N-2Nw) \leq 0 \tag{12}$$

from which the quadratic formula yields k as the floor of (i.e. the largest integer not greater than) the expression

$$\tfrac{1}{2}(3+2w-3N+\text{sqrt}(13N*N-20Nw-18N+4w*w+12w+9)) \tag{13}$$

which for large N becomes

$$\tfrac{1}{2}(-3N+2w+\text{sqrt}(13N*N-20Nw+4w*w)). \tag{14}$$

This formula reveals reduction in searching the worst, median, and nearly-best *nodes of 44.9%, 30.3%, and 0%, respectively, which correspond to w values of $-(\frac{1}{2}N-1)$, 0 and $\frac{1}{2}N-1$, respectively, and associated j values of $0.551N$, $0.697N$ and $1.0N$, respectively. To determine the total number of nodes pruned, we add up the above formula as w runs from $-(\frac{1}{2}N-1)$ to $\frac{1}{2}N-1$. For large N, this sum can be found by integration. Once again ignoring lower-order terms, we obtain an asymptotic fractional savings of 0.279. We can therefore state the following.

Result 1. The asymptotic best-case behavior of algorithm Star1 on the +node of a *-complete 2-ply N-ary tree is to examine approximately $0.721N**3$ of the $N**3$ leaves beneath it.

Table 1 gives exact values for the best case performance of the left-to-right Star1 procedure for various values of N. Note the convergence toward the region of 72 percent, as predicted by the analysis above.

6.1.2. *Best-case analysis of the Star2 procedure*

If the successors of −nodes are optimally ordered, so that their first successors

TABLE 1. Best-case leaf exploration of Star1 for various *-complete 2-ply trees

N	2	4	6	8	10	20	30	40
Number	5	40	138	336	670	5560	18990	45320
Percent	62.5	62.5	63.9	65.6	67.0	69.5	70.3	70.8

are minimal, then if a cutoff will occur at all, it will occur during the preliminary probing phase. This means that only one successor per −node will need to be looked at by Star2, whereas all N of all but the last −node were examined by Star1. As revealed above, an average of 72.1 percent of the N −nodes below each of the N *nodes, i.e. 0.721 $N*N$ nodes, will be examined for all but the best *node, whose $N*N$ leaves must all be considered. This gives us:

Result 2. The asymptotic best-case behavior of algorithm Star2 on the +node of a *-complete 2-ply N-ary tree is to examine approximately $1.721N**2$ of the $N**3$ leaves beneath it.

This result is encouraging because, like the $O(N**2)$ best case alpha-beta result for depth 3 trees [2], it shows that a wise algorithm can hope to reduce search complexity by a factor of N. For the most part, we achieved this reduction without significantly increasing the conceptual complexity of the algorithm, its overhead, or the additional space needed (which will in fact be only $N*D$ cells for an N-ary tree of depth D). Table 2 gives exact values for the best case performance of the Star2 procedure for various values of N.

6.2. Average-case analysis

Since average-case analysis is more difficult than best-case analysis, we decided to investigate the expected-case performance of the Star1 and Star2 procedures mainly by empirical means. To do this, we coded the algorithms in the 'C' language to be run on our PDP-11/70 system. Since the foregoing algebraic best-case analysis ignores lower-order terms, and thus cannot yield reliable

TABLE 2. Best-case leaf exploration of Star2 for various *-complete 2-ply trees

N	2	4	6	8	10	20	30	40
Number	5	25	58	105	166	677	1532	2732
Percent	62.5	39.1	26.9	20.5	16.6	8.5	5.7	4.3

values for small values of N, we preceded our average-case experimentation by running each algorithm on an optimally ordered tree. The results of this exact analysis were given in Tables 1 and 2. Finally, after competing our empirical study, we undertook a simplified algebraic analysis of average-case Star1 performance, which led to an iterative formula which will be presented after describing the results of the empirical study.

6.2.1. *Empirical average-case study*

Using the UNIX pseudo-random number generator, we generated and gathered statistics on 1000 *-complete trees for each of several branching factors. In generating the successors of a node, all $N!$ permutations of successor arcs were assumed to be equally likely. We did this because (a) it is simple to implement; (b) it corresponds to completely 'uniformed' static evaluation capabilities, thus giving a conservative picture of what to expect in practice; and (c) it has been adopted by previous researchers, thus enabling a comparison of *-minimax trees against ordinary minimax trees. In our implementation, the leftmost *node is searched exhaustively, since as we have observed no cutoff can occur beneath it. In the event that preliminary probing failed to result in a cutoff, the N leaves seen were counted twice if subsequent search required a full search of the −node above them. Thus, the results obtained represent a conservative estimate of average case analysis, which is itself a conservative estimate of how well one can expect to do in practice.

6.2.1.1. *Average-case analysis of the Star1 procedure*
Table 3 presents the average-case results for the initial left-to-right Star1 procedure. It can be seen that the average case savings appears to be about 21 percent, roughly $\frac{3}{4}$ times the best-case savings of 28 percent.

6.2.1.2. *Average-case analysis of the Star2 procedure*
We have seen that with optimal ordering, the search complexity of Star2 on regular *-complete trees can be reduced from O(N ** 3) to O(N ** 2). This may lead us to expect a significant improvement in its average-case behavior as well. Table 4 summarizes the results of Star2 performance.

In addition to simply counting leaf explorations, we decided to gather information on how many *node cutoffs were made during the preliminary probing phases. An interesting result was that roughly half the *nodes for

TABLE 3. Average-case leaf exploration of Star1 for various *-complete 2-ply trees

N	2	4	6	8	10	20	30	40
Number	7.1	53.9	178	418	810	6389	21382	50425
Percent	88.8	84.1	82.5	81.6	81.1	79.9	79.2	78.8

TABLE 4. Average-case leaf exploration of Star2 for various *-complete 2-ply trees

N	4	6	8	10	20	30	40
Cutoffs							
Probing	1.3	2.0	2.8	3.5	8.1	12.6	17.4
Regular	0.7	1.5	2.5	3.5	8.4	13.4	18.3
Leaves seen							
Number	48	139	293	531	3341	10109	22390
Percent	75.4	64.5	57.3	53.1	41.8	37.4	35.0

which a cutoff occurred were cut off during the probing phase. Also, we see that for a branching factor greater than about 20, Star2 looks at fewer than half the leaves explored by Star1.

6.2.1.3. *Discussion of average-case Star2 results*

In the *-complete trees we have been considering, where each non-terminal node has N successors, there will be $N!$ ways of ordering the arcs below each node. In both the Star1 and Star2 procedures, searching is left-to-right on the * nodes below the root. In this case, *each *node better than all its predecessors must be fully searched.* If each of the $N!$ permutations is equally likely, the expected number of such fully-searched *nodes is the 'harmonic' function, given by

$$H(N) = 1 + 1/2 + \cdots + 1/N.$$

This formula is easily verified by induction: given $H(N-1)$, adding an Nth node worse than all the others will have no effect of the searching of the previous $N-1$ nodes, while the new node will be fully searched only if it is placed first in the permutation, which happens one time in N. Thus, the number of leaf nodes searched beneath best-so-far *nodes is $N*N*H(N)$. If in Table 4 we subtract from N the average number of cutoffs that occurred, we observe perfect agreement with $H(N)$ (within the 0.1 tolerance due to rounding). For instance, with $N = 40$, we see that an average of 4.3 *nodes were found superior to previously searched *nodes. This accounts for $4.3*40*40$ or 6880, of the leaves that were explored. The remaining 35.7 *nodes can be seen to have had an average of (22390 − 6880)/35.7 or 434, of the 1600 leaves beneath them examined. The reader will recall that not all the additional savings of Star2 is made possible by preliminary cutoffs, but also by the lower values assigned for the A's. For instance, the 18.3 *nodes which led to a 'regular' cutoff did so after searching fewer than (22390 − 6880)/18.3 or 848 leaves. This figure represents about 70 percent of the average number of (50425 − 6880)/35.7 or 1220 leaves examined by Star1 for the corresponding *nodes.

6.2.2. *Algebraic average-case analysis of Star1*

We have observed that the expected number of fully-searched *nodes is given by the harmonic function $H(N)$, so that the number of leaves searched beneath them is

$$A(N) = N * N * H(N)$$

We will now determine $B(N)$, the number of leaves searched beneath the remaining *nodes, which we add to $A(N)$ to find $C(N)$, the total average number of leaves searched in an N-ary *-complete 2-ply tree.

Each of the $N - H(N)$ *nodes not better than all preceding *nodes will be rejected when its value is determined to be less than the value of the best *node seen so far. Recall that in the *-complete trees we are considering, *nodes have backed-up values of $0, -1, \ldots, -(N-1)$. Suppose a given *node has a value of n, and m is the value of the best *node seen so far, where $n < m$. In addition to the *node of value m, any of the *nodes with a value less than m (but not equal to n) can precede the n-valued node. The number of ways this can occur is given by

$$D(M) = \sum_{s=0}^{N-M-2} \binom{N-M-2}{s} * (s+1)! * (N-s-2)! \tag{15}$$

where $M = |m|$. A cutoff will occur below this n-valued *node when we have searched a jth successor for which

$$\frac{V_1 + \cdots + V_{j-1} + V_j + U * (N-j)}{N} \leq m\,. \tag{16}$$

But the values of the −nodes below the n-valued *node are $n - \frac{1}{2}N, \ldots, n-1$, $n+1, \ldots, n+\frac{1}{2}N$, with an average value of n. For large N, j becomes large as well, and the expected value of the relative difference between $(V_1 + \cdots + V_j)$ and jn approaches 0.

Although the average expected value of a function is not in general the same as the expected value of the average of the function, in the present nearly-linear situation, it gives a good approximation. Thus, using the 'jn' value derived above as an approximation, we obtain

$$jn + (\tfrac{3}{2}N - 1) * (N-j) \leq mN \tag{17}$$

whose high-order terms give

$$2jn + 3N * N - 3jN \leq 2mN \tag{18}$$

which reduces to give a value for j, which we denote by $J(m, n)$, of

$$J(m,n) = \frac{3N - 2m}{3N - 2n} * N\,. \tag{19}$$

TABLE 5. Algebraic average-case analysis of Star1 compared with empirical findings

N	4	6	8	10	20	30	40
Algebraic	57	188	438	846	6557	21805	51174
Empirical	54	178	418	810	6389	21382	50425
Percent difference	5.6	5.3	4.6	4.3	2.6	1.9	1.5

This means that the average number of nodes searched beneath *nodes that are not better than all preceding *nodes is

$$B(N) = \frac{\sum_{n=1}^{N-1} \sum_{m=0}^{n-1} D(m) * J(-m, -n) * N}{N!}. \tag{20}$$

By supplying the expressions for the D and J functions, simplifying, and adding $A(N)$, we find the total number of leaves searched to be

$$C(N) = N * N * H(N) + N * \sum_{n=1}^{N-1} \sum_{m=0}^{n-1} \frac{3N-2m}{3N-2n} * \sum_{s=0}^{N-m-2} \frac{(s+1)}{\prod_{x=N-s-1}^{N-1} x} * \prod_{t=N-m-1-s}^{N-m-2} t. \tag{21}$$

Table 5 gives leaf exploration figures for specific values of N, and compares the experimentally derived results against them.

Since simplifications were made in deriving $B(N)$, asymptotic agreement is not guaranteed. However, it can be seen that the agreement is reasonably good (less than 2 percent deviation) for larger values of N.

6.3. Summary of results for two-ply trees

We summarize the preceding results in Fig. 9.

6.4. Empirical analysis for deeper trees

Having observed an appreciable savings for trees of depth 3, we decided to investigate the performance of the *-minimax algorithms on deeper trees. Since even levels of our trees are associated with chance events rather than player moves, 3-ply and 4-ply trees have depths of 5 and 7, respectively, while the leaves of trees of depth 4 and 6 correspond to positions where chance events have occurred but not yet been responded to. The empirical analysis performed is analogous to that described above. Results are given in Table 6. Readers can

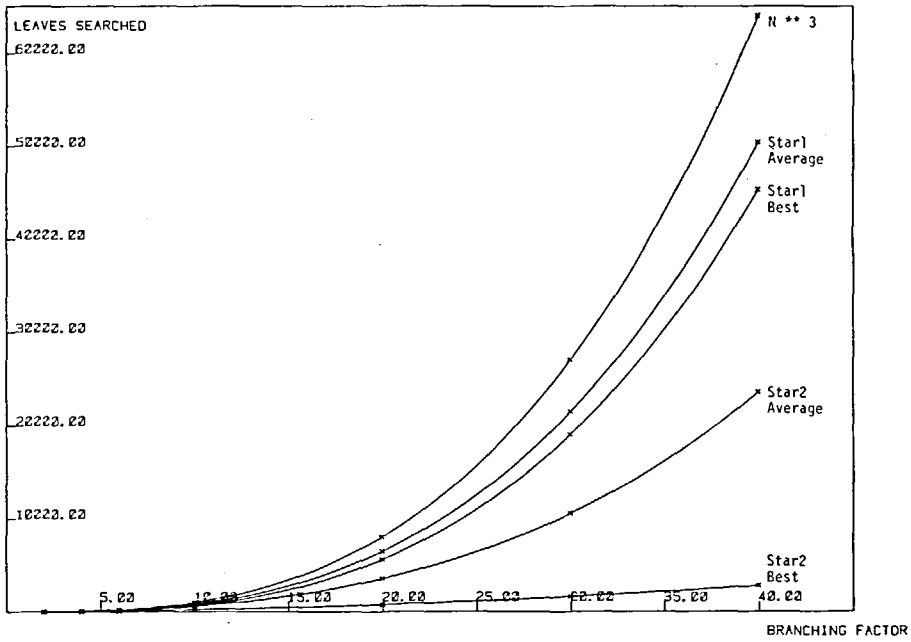

FIG. 9. A graphical summary of best-case and average-case results for various *-minimax algorithms.

determine the approximate number of leaves explored by multiplying the percentages given by $B ** D$, where B represents branching factor and D represents depth. For instance, the best-case leaf exploration of Star2 for 3-ply (depth 5) trees of branching factor 10 is about $0.11*(10**5)$ or about 11000.

7. Additional Modifications

When a cutoff has not occurred below a given *node during the preliminary probing phase, Star2 performs an exhaustive search of −nodes. Instead of doing this, we might consider just one additional child of each −node below the *node in question. If a cutoff has still not occurred, a third of each −node would be considered, and so forth, until either a cutoff occurs or an exhaustive search is completed. We refer to this modified form of Star2 as the 'cyclic Star2.5' procedure.

In practice, to avoid the unpleasant time and space overhead involved in doing the breadth-first traversal required by the cyclic Star2.5 algorithm, we might resort to exhaustive searching of the remaining children of each −node after having considered a predetermined number of its children. We refer to

TABLE 6. Empirical analysis of leaf exploration (in percent) for *-complete trees of varying depth and branching factor

Branching factor	Depth	Star1		Star2	
		Best	Average	Best	Average
4	3	62	84	45	87
4	4	58	81	38	75
4	5	55	77	38	64
4	6	56	77	32	40
4	7	54	—	28	—
6	3	64	82	30	71
6	4	56	77	24	61
6	5	56	74	21	38
6	6	55	73	17	25
6	7	53	—	11	—
10	3	67	81	18	57
10	4	55	73	14	46
10	5	58	73	11	22
10	6	54	—	8	12
10	7	56	—	4	—
20	3	70	80	9	44
20	4	55	70	7	32
20	5	60	—	5	12
20	6	55	—	3	—

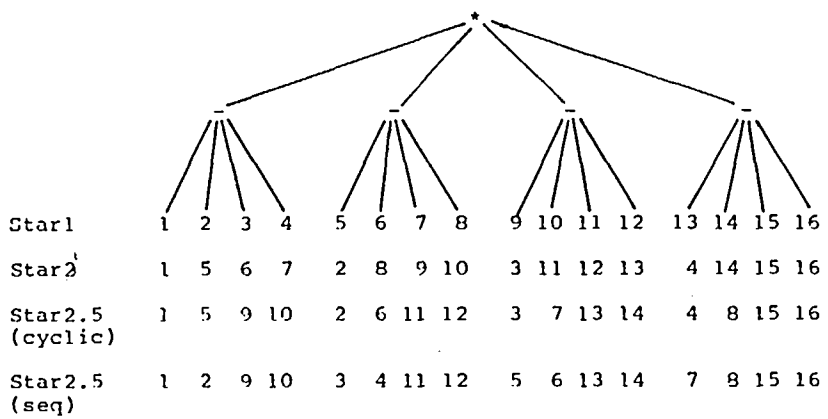

FIG. 10. Order of leaf exploration for various *-minimax algorithms. The Star2.5 procedures are given with a probing factor of 2.

this value as the 'probing factor'. With a probing factor of 1, cyclic Star2.5 reduces to Star2. With a probing factor of 0 it reduces to Star1.

A different way to modify Star2 would be to have just one probing phase per −node but to consider several of its children during it. Although this strategy is probably inferior, in terms of expected number of leaves to be examined, to the cycling described above, it requires considerably less overhead. We call this the 'sequential Star2.5' procedure. As before we use the term 'probing factor' to denote the number of children of each −node to be examined before we resort to an exhaustive search. With a probing factor of 1, sequential Star2.5 reduces to Star2. With a probing factor of 0 or N it reduces to Star1.

To clarify the exact behavior of the algorithms presented thus far, an example is given in Fig. 10 of the order of leaf explorations beneath a *node for a tree with a branching factor of 4.

7.1. Empirical average-case study of the Star2.5 algorithms

To evaluate the Star2.5 procedures, we generated 100 trees of depth 3 and

TABLE 7. Empirical average-case study of the Star2.5 algorithms on a tree of depth 3 and branching factor 20

	Star2.5 (cyclic)		Star 2.5 (seq.)	
Probing factor	Prelim cutoffs	Percent leaves	Prelim cutoffs	Percent leaves
1	8.0	40.7	8.0	40.7
2	11.3	34.4	11.1	36.8
3	12.7	31.7	13.0	34.7
4	13.7	30.2	14.2	34.7
5	14.4	29.3	14.8	36.6
6	14.8	28.7	14.9	39.8
7	15.2	28.4	15.4	41.6
8	15.5	28.1	16.5	45.2
9	15.7	28.0	15.9	46.7
10	16.0	27.9	16.0	50.4
11	16.2	27.9	16.4	52.4
12	16.3	27.9	16.2	56.3
13	16.3	27.9	16.5	58.7
14	16.3	27.9	16.4	61.8
15	16.3	27.9	16.2	65.6
16	16.3	27.9	16.3	68.0
17	16.3	27.9	16.3	71.1
18	16.3	27.9	16.5	73.7
19	16.3	27.9	16.4	76.7
20	16.3	27.9	16.5	79.6

branching factor 20, then ran each of the Star2.5 algorithms for probing factors ranging from 1 to 20. Since all cutoffs that will occur in the best-case situations for Star2 occur during the initial probing phase, only average-case analysis is worth considering. The results appear in Table 7.

As indicated in Table 7, the number of probing cutoffs for cyclic Star2.5 increases with probing factor, but the rate of increase falls off quickly. In fact, virtually no additional cutoffs were observed after a probing factor of $\frac{1}{2}N$ was reached, since all possible cutoffs had occurred beneath the $N - H(N)$ *nodes better than all preceding *nodes.

The sequential version of Star2.5 shows improvement as the probing factor increases from 1 to 3, then levels off and quickly begins to degrade, ultimately reaching the figure for Star1. This is because as the probing factor increases, we are able to obtain cutoffs beneath more *nodes, but at the expense of doing more work for those *nodes that would have been cutoff with a smaller probing factor.

7.2. A final modification

We have seen that cutoffs can occur only beneath *nodes which are worse than some previously searched *node. For this reason, it is useful to find the best *node, or at least a good one, before carrying out exhaustive or nearly exhaustive search beneath inferior *nodes. Experimentation with the existing algorithms indicates that if the best *node is seen first, then for leaf dependencies of the sort defined in Section 6, and for a branching factor of 40, the average-case search complexity of Star2 is reduced from 35 percent to 30 percent. Similarly, the average-case performance of Star2.5 with 2-node cycling is reduced to 26.3 percent. We have formulated and begun to experiment with an algorithm called Star3 which incorporates probing beneath *nodes. Since additional overhead is needed for the initial probing, and since one cannot realistically expect to find the best *node each time, the actual behavior of Star3 will not be as good as these benchmark figures.

8. Remaining Considerations

We complete our presentation and analysis of the *-minimax search problem, and our solution to it, by devoting brief attention to some remaining topics.

8.1. The efficacy of search

In developing and examining algorithms for *-minimax trees, we have assumed that search proceeds until either a leaf is encountered or an allowable form of cutoff occurs. In this case, the move selected is guaranteed to be optimal, i.e. to have an expected value at least as good as the alternatives. As we shall observe in Section 8.4, however, it is seldom possible to carry out a complete search,

and so in practice the values of many non-terminal nodes must be determined by static evaluation.

When complete search is not possible, both intuition and empirical observation suggest that deeper search will result in better play. However, recent results by Nau [6] show that for ordinary minimax trees, deeper search can in some situations result in making worse moves. Despite the existence of such 'pathological' trees (the terminology of [6]), it is probably advantageous for most games of interest, including the *-minimax games we have been considered, to search as deeply as possible (at least as deeply as current technology permits).

Although play is likely to be better with deeper search, no general statement can be made as to how much better it can be expected to be. For many games of chance, *strategy* appears to be much more important than search, and in fact the currently top-ranked backgammon program (BKG) does not carry out a tree search. Although its author indicates that he selected backgammon for study because he "wanted was a domain where it is possible to... make a judgment... without having to worry about... exhaustive analysis" (Berliner [7]), he states elsewhere that "the deeper one could look, the better [a] program would play" [8].

Perhaps one reason search has received so little attention for games such as backgammon is that the $O(N ** (2D-1))$ complexity of the 'obvious' search strategy appears infeasible. For example, Berliner [9] observes that

> "the throw of a pair of dice can produce 21 different results, and each such throw can be played about 20 different ways in the average position. Thus a look-ahead would have to acquiesce to a branching factor of about 400 for each ply of look-ahead; an exponential growth rate than could not be tolerated for very long."

In this paper, however, we have presented an algorithm which reduces this branching factor of 400 to 677/20 or about 34. Since we have assumed equally likely chance outcomes (however, see Section 8.3) and made other pedagogical assumptions, this should be treated as only an approximate figure for a particular game such as backgammon.

8.2. An unusual form of cutoff

Knuth and Moore [2] have shown that whereas the alpha-beta algorithm for minimax trees is more powerful than the more obvious branch-and-bound strategy, there is no uniformly stronger method. This assumes, however, that we must determine the precise value of the root, not just the best move. Thus, if the root node of a minimax tree has N successors, and the first $N-1$ of them have been searched and all found to have the lowest possible game value (e.g. 'forced loss'), alpha-beta will still search the remaining node, even though this node is known to be at least as good as any of the alternatives! In the degenerate case, this would mean searching a tree with only one branch (denoting a 'forced' move) from the root. Needless to say, existing game-playing programs typically respond without search in these situations.

When searching the last successor of the root of a *-minimax tree, a stronger form of cutoff can be made. In particular, we can discontinue search, knowing that the rightmost node is *strictly better* than the alternatives, even though we may not know its exact value. This is because the value of a *node is *partially* determined by the value of *each* of its successors, while a −node is *fully* determined by *one* of its successors. We implement this form of cutoff by discontinuing search beneath the *rightmost* *node when its lower *-value exceeds the *alpha* value (rather than beta) passed into it. Being able to cut off below the node corresponding to the best move, without knowing its exact value, can be important in reducing search time for *-minimax trees, especially with a narrow branching factor below + and −nodes (e.g. in casino blackjack, with a branching factor of 2 beneath +nodes).

8.3. Differing probabilities below a *node

Neither algorithm presented above considers the situation where not all outcomes of the chance event are equally likely. If P_i denotes the probability with which the ith successor of a node occurs, then the left side of (1a) is replaced by

$$(P_1V_1 + \cdots + P_{i-1}V_{i-1}) + P_iV_i + U*(1 - P_1 - \cdots - P_i) \tag{22}$$

and (2a), (3a) and (4a) are modified accordingly.

In searching ordinary minimax trees, the static evaluation function, or a similar 'plausible move generator', is often used to determine the order in which to consider successors of a node. When the probabilities of outcomes differ in *-minimax trees, a potentially useful strategy is to examine more likely successors first, since their values will more strongly influence the *node value. However, one must weigh against this the likelihood of a useful (i.e. extreme) value, and also the probable number of nodes to be pruned (i.e. below sister nodes) if a cutoff does occur. In *-minimax trees, where cutoffs are harder to come by, the typical tradeoff between the likelihood and benefit of a cutoff is compounded in the case of differing probabilities. Decisions as to which combination of strategies to adopt are probably best made by considering the idiosyncrasies of the particular game under consideration.

As described in [10], we have recently shown that advantages can be obtained over minimax by treating the −nodes of ordinary minimax trees as though they were *nodes with weights determined by an estimate of the 'fallibility' of the opponent.

8.4. Incorporating *-minimax search into a complete game program

In programming actual minimax games, adjustments are often made to a pure alpha-beta search because of the overwhelming size of most search trees. In particular, a *static evaluation* function is generally used to rank successor nodes in what appears (before searching) to be best-to-worst order, hoping to assure

early cutoffs; a *depth bound* is often maintained in some form to preclude searching prohibitively deep nodes; *forward pruning* is performed, meaning that some nodes which look unpromising are not searched at all; a *transposition table* is maintained to avoid searching the same position more than once if it appears in several places ('transpositions') in the search tree; and so forth. In practice, we would expect such modifications to be made to the *-minimax procedures as well, although the underlying algorithms need not be changed.

ACKNOWLEDGMENT

The experimental results given in Table 6 and the plots given in Fig. 9, were produced by Andrew Reibman, a graduate student in our department. The author wishes to thank Dr. Donald Loveland, Dave Mutchler and Andrew Reibman for valuable discussions during the course of our research. We are also grateful to Tom Truscott and an anonymous reviewer for comments on the form and content of the manuscript. Computer time for the empirical studies reported herein was provided for by a grant from the Air Force Office of Scientific Research.

REFERENCES

1. Nilsson, N.J., *Principles of Artificial Intelligence* (Tioga Publishing Company, Palo Alto, CA, 1980).
2. Knuth, D.E. and Moore, R.W., An analysis of alpha-beta pruning, *Artificial Intelligence* **6** (1975) 293–326.
3. Fuller, S.H., Gashnig, J.G. and Gillogly, J.J., An analysis of the alpha-beta pruning algorithm, Dept. of Computer Science Rept., Carnegie-Mellon University, Pittsburgh, PA, 1973.
4. Newborn, M.M., The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores, *Artificial Intelligence* **8** (1977) 137–153.
5. Baudet, G.M., On the branching factor of the alpha-beta pruning algorithm, *Artificial Intelligence* **10** (1978) 173–199.
6. Nau, D.S. Pathology on game trees: a summary of results, *Proc. First National Conf. Artificial Intelligence* (1980) 102–104.
7. Berliner, H.J., Computer backgammon, *Scientific American* **242** (1980) 64–72.
8. Berliner, H.J., An examination of brute force intelligence, in: *International Joint Conference on Artificial Intelligence* (IJCAI, Cambridge, MA, 1981).
9. Berliner, H.J., Backgammon computer program beats world champion, *Artificial Intelligence* **14** (1980) 205–220.
10. Reibman, A.L. and Ballard B.W., Non-minimax search strategies for use against fallible opponents, *Proc. Third National Conf. Artificial Intelligence* (1983) to appear.

Received April 1982; revised version received November 1982