Experiments With Some Programs That Search Game Trees

JAMES R. SLAGLE AND JOHN K. DIXON

National Institutes of Health, Bethesda, Maryland*

ABSTRACT. Many problems in artificial intelligence involve the searching of large trees of alternative possibilities—for example, game-playing and theorem-proving. The problem of efficiently searching large trees is discussed. A new method called "dynamic ordering" is described, and the older minimax and Alpha-Beta procedures are described for comparison purposes. Performance figures are given for six variations of the game of kalah. A quantity called "depth ratio" is derived which is a measure of the efficiency of a search procedure. A theoretical limit of efficiency is calculated and it is shown experimentally that the dynamic ordering procedure approaches that limit.

KEY WORDS AND PHRASES: artificial intelligence, heuristic procedures, game-playing, kalah, problem-solving, tree searching, Alpha-Beta procedure

CR CATEGORIES: 3.64

1. *Introduction*

Tree searching is a task which must be done in many different types of computer procedures. Game-playing [1, 3, 4, 11, 13, 15, 16], theorem-proving [7, 8, 19], and many other heuristic programs [8, 18, 19] must perform some kind of tree search. The efficiency of tree searching is crucial to such procedures because tree searching is usually very time-consuming. In this paper we are concerned with techniques for reducing the effort required to search a given tree. All the procedures described in this paper were written in LISP 1.5 and run on an IBM 7094 computer.

As an example of a tree-searching problem, consider the game of checkers. Let us define P as the current position of the game; P includes the location and identity of each man on the board and an indication of whose turn it is to move. The rules of the game permit B different alternative moves from P. Each such move defines a new game position P_x. We say that the successors of P are $P_1, P_2, \cdots, P_B$. In playing the game it is necessary to select one of these alternative possibilities. But in order to make a wise choice, it is necessary to look several moves ahead. If each successor of P_x also has B successors, then two moves ahead, there will be B^2 successors designated by $P_{1,1}, P_{1,2}, \cdots, P_{1,B}, P_{2,B}, \cdots, P_{B,B}$. In general, looking D moves ahead we find B^D positions, each one identified by a D-tuple. This is assuming that B remains constant. In checkers, 10 is a typical value for B. Thus in looking ahead six moves, we have one million positions to consider. A checker-playing procedure would assign a value to each one of the terminal positions by means of some evaluation function, back these values up to the $D = 1$ level by assuming that each player will choose, at each node, that move which is best for himself, and then select that move which has the largest backed-up value.

This work was performed at Lawrence Radiation Laboratory, Livermore, California, operated by the University of California for the Atomic Energy Commission.

* Division of Computer Research and Technology.

Positions are usually evaluated by a value function which assigns a numerical value to each position. We define $V_{A,B,C}$ to be the value of position $P_{A,B,C}$. A value function is usually a linear combination of features. For example, in checkers a simple evaluation polynomial might be two times the man advantage plus three times the king advantage, where "advantage" means the number a player has minus the number his opponent has. If we assume a two-person, zero-sum game, then a large value is favorable to one player and unfavorable to the other player. We call the former player Max and the latter player Min. By convention, it is Max's turn to move at P, the top position of a tree.

The value of a position determined by direct application of the value function is called the static value of the position. The backed-up value of a position is the value obtained by generating successors, evaluating them, and backing up a value by means of some backing-up function.

There are several types of backing-up functions, but the minimax procedure is the one most commonly used in game-playing programs. It is based on the assumption that each player will choose that move which is most advantageous to himself. Thus, we back up the maximum successor value when it is Max's turn to move (called a "Max position") and the minimum successor value when it is Min's turn to move (called a "Min position"). Although other types are possible,[1] all the work described in this paper was done with a minimax backing-up procedure.

The backing-up procedure may be applied recursively to back-up values from any depth. Thus, each position has a static value, a first backed-up value derived from the static values of the immediate successors, a second backed-up value derived from the static values of the successors of the successors, a third backed-up value, and so on. In principle one could back up Win/Lose/Draw values on the complete game tree. Usually, of course, the tree is so large that it is not practical to reach the end, so we must stop at some depth called D_{MAX}.

The relationship between shallow and deep backed-up values of a position is central to the tree-searching problem. The value function is intended to predict which positions are most likely to lead to winning the game. Deeper backed-up values are presumably more accurate than shallow ones. But we expect to find a correlation between shallow and deep values. If the static value of P_1 is larger than the static value of P_2, it is likely that backed-up values will also be larger for P_1 than for P_2. The relation of shallow and deeper values for all ten of the depth-1 positions in one version of the game of kalah is shown in Figure 1. (A similar graph for a different version of kalah is shown in Figure 6.)

The fact that shallow values predict deeper values suggests a way to reduce the size of a large tree. For example, if the static value of one successor to a Max position is very low compared to the others, it is unlikely that deeper search will reveal a backed-up value that is higher than the others. Hence, it may save time to "forward prune" the low-valued successor from the tree, that is, make no deep search under it. However, all of the methods discussed in this paper operate without forward pruning. Thus it is guaranteed that, for a given depth of search, one of the highest valued successors will be chosen.

Since all of the tree-search programs described herein were tested on kalah, a

[1] The M and N procedure [18] backs up some function of the M highest valued successors of a Max position and some function of the N lowest valued successors of a Min position. Preliminary results indicate that the M and N procedure is superior to minimax.

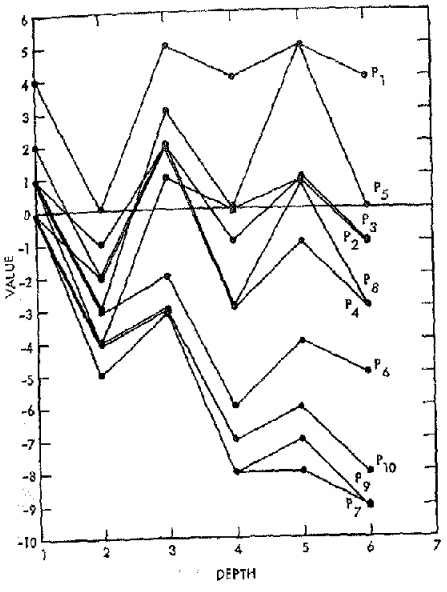

FIG. 1. Value versus depth, 2-in-a-hole kalah. Values are given for the ten depth-1 successors. Values at depth = 1 are static values. Others are backed-up values.

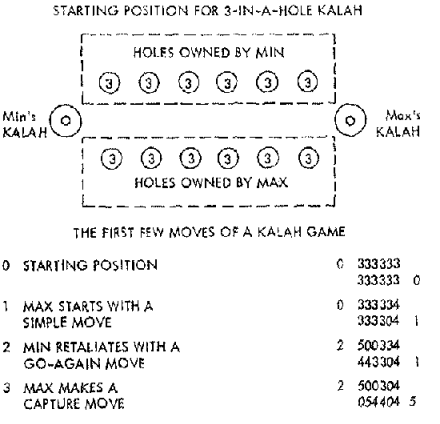

FIG. 2. An illustration of the rules of kalah

brief description of the rules of the game is now given. The game is played with a number of stones and a number of holes. Each player owns one big hole, called a kalah, and six smaller holes. At the beginning of the game the kalahs are empty and there are N stones in each of the other holes.

Figure 2 (top) shows the starting position for three-in-a-hole kalah. A player wins if he gets more than half the stones in his kalah.

To make a move, a player first picks up all the stones in one of his holes. He then proceeds counterclockwise around the board, putting one stone in each hole, including his own kalah, but skipping his opponent's kalah until all the picked-up stones are gone. What happens next depends on where the last stone lands. There are three alternatives. If the last stone lands in the player's own kalah, he makes another move. This is called a "go-again."

The second alternative is called a "capture." If the last stone lands in an empty hole owned by the player, and if the opponent's hole directly opposite contains at least one stone, then the player captures all the stones in the opponent's hole. The player places all the captured stones and his own last stone in his kalah, and the opponent moves next. The third alternative is the simplest case. If the last stone lands so that neither a go-again nor a capture occurs, then the opponent moves next.

There are two conditions which end the game. If a player gets more than half of the stones in his kalah, the game is over and he is the winner. If all the holes owned by one player, say Min, become empty (even if it is not his turn to move), then all the stones remaining in Max's holes are put in Max's kalah and the game is over. In either case the winner is the player who has more stones in his kalah at the end of the game.

These rules are illustrated by several moves shown in Figure 2 (bottom).

TABLE I. MINIMAX RESULTS*

D_{MAX}		No. stones per hole							
		1	2	3	4	5	6	Av	B
1	NBP	10	10	10	10	10	10	10	10
	Time	1	1	1	1	1	1	1	
2	NBP	98	98	106	116	108	60	98	10
	Time	2	2	2	2	2	2	2	
3	NBP	676	724	818	1,022	1,055	329	770	9.16
	Time	10	10	11	12	12	8	10.5	
4	NBP	4,380	5,512	6,834	9,862	6,727	1,907	5,840	8.74
	Time	63	74	141	106	75	45	84	
5	NBP	19,168	41,014	61,241	125,843	44,695	12,441	50,236	8.73
	Time	310	491	673	1,369	489	236	595	
6	NBP	73,794	295,296	536,000†	1,090,836	292,196	80,209	394,721	8.58
	Time	1,046	3,677	5,500†	11,211	3,221	1,155	4,302	

* NBP = number of bottom positions; time is given in seconds; B = branching factor.
† Estimated (this problem would not run since it needed more than the available free storage).

An obvious value function to use in this game is the number of stones in Max's kalah minus the number of stones in Min's kalah. This value function, called the kalah advantage, was used in all the programs described in this paper.

Now we consider results obtained with the simplest kind of tree-searching procedure, the simple minimax procedure. This method examines every possible successor down to D_{MAX}. The positions at the bottom of the tree are evaluated and the results are backed up to the first level by the minimax backing-up procedure. The procedure moves to the first depth-1 position with the highest backed-up value. Table I gives search time and size of tree generated in terms of the number of positions at the bottom of the tree (NBP). Six different forms of kalah were tested to lend greater generality to conclusions.

The number of stones in each hole at the start of the game was varied from one to six. This provides six similar but distinctly different games on which to test the procedures.

2. *Alpha-Beta*

Alpha-Beta[2] is a tree-search procedure that is faster than minimax but still equivalent in the sense that both procedures will always choose the same depth-1 successor at best, and will assign the same value to it. Alpha-Beta is typically several orders of magnitude faster than minimax. It saves time by not searching certain branches of the tree. Under certain conditions the values of certain branches do not affect the value which is ultimately backed up to higher levels of the tree. Hence, there is no point in evaluating these branches. When the Alpha-Beta program detects these

[2] The Alpha-Beta procedure was first used by Newell, Shaw, and Simon in 1958 (see [8], p. 56), but was not given a specific name. The procedure is discussed in more detail by Edwards and Hart in [6].

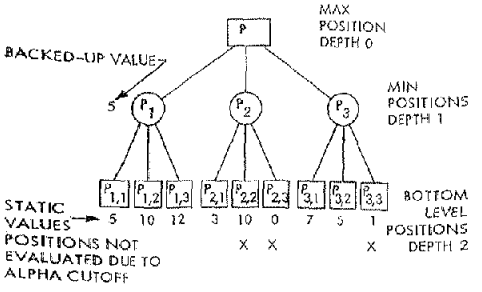

FIG. 3. Example of Alpha-Beta procedure, case of $D_{MAX} = 2$.

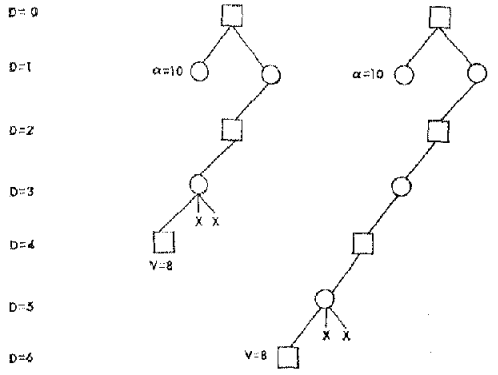

FIG. 4. Deep alpha cutoffs. × means cutoff.

conditions, it stops work on one branch and skips to another. This event is called an alpha or beta cutoff.

To see how the Alpha-Beta program works, consider the example shown in Figure 3. Alpha-Beta starts just like the minimax procedure by evaluating all the successors of P_1. The minimum of these static values is then backed up to P_1 since P_1 is a Min position. The backed-up value of P_1 is alpha and has the value 5 in the example.

Alpha is a lower limit for the backed-up value of the top position, P. Since P is a Max position, we back up the value of the largest valued successor of P. Since we have evaluated only one successor at this time, we do not know what the final value of P will be, but we do know that it will be 5 or larger. The value of alpha may change as the other successors are evaluated, but it can only increase, not decrease.

Having evaluated P_1, the procedure begins work on P_2. An alpha cutoff takes place at $P_{2,1}$ since $V_{2,1} = 3$ is less than alpha. Since P_2 is a Min position, $V_{2,1}$ is an upper limit for V_2. Since V_2 is less than alpha, P_2 is definitely eliminated as candidate for the largest valued successor of P. There is no point in evaluating the other successors of P_2 so the procedure begins work on P_3 next.

The alpha cutoffs save the machine a good deal of time. In the example shown in Figure 3 there is an alpha cutoff at $P_{2,1}$, which means that the machine need not bother to evaluate $P_{2,2}$ or $P_{2,3}$. A second alpha cutoff occurs at $P_{3,2}$, which eliminates $P_{3,3}$. Thus in this example the Alpha-Beta program would evaluate only six of the bottom-level successors while a minimax program would evaluate all nine.

Although the example is given for a tree of only three levels, it is clear that the procedure will work just the same below any Max position, P_x, at any depth in a large tree, provided only that there are at least two levels below P_x. If there were more levels below $P_{1,1}$ in the example then we could use the backed-up value of $P_{1,1}$ instead of the static value. If there are more levels above P, then we would back up the final value of P. Moreover, it is possible to pass a value of alpha down from the top of a large tree. Thus an alpha established at depth 1 could be used to produce cutoffs at depths 2, 4, and 6. These deep cutoffs are illustrated in Figure 4.

Alpha is defined by the values of the successors of a Max position (odd depths)

TABLE II. ALPHA-BETA RESULTS*

D_{MAX}		No. stones per hole						Alpha-Beta, av ($DR = 0.733$)	Minimax, av ($DR = 1.00$)
		1	2	3	4	5	6		
1	NBP	10	10	10	10	10	10	10	10
	Time	1	0.42	0.91	0.41	0.41	0.43	1	1
2	NBP	70	75	66	60	37	15	54	98
	Time	1.3	1.2	1.3	1.3	1.2	0.83	1.2	2
3	NBP	380	381	300	275	253	68	276	770
	Time	6.9	6.1	6.1	7.2	5.5	1.6	5.5	10.5
4	NBP	1,315	1,066	1,285	1,237	639	131	946	5,840
	Time	20.0	24.5	34.7	30.5	17.4	4.7	23.4	84
5	NBP	4,168	5,349	5,686	5,213	3,049	759	4,037	50,736
	Time	101.1	134.0	142.6	137.1	60.4	14.7	98.3	595
6	NBP	6,886	15,685	18,008	20,726	12,359	2,989	12,775	394,721
	Time	182.9	471.5	848.3	515.9	294.3	75.6	364.8	4,302

* NBP = number of bottom positions; time is given in seconds.

while alpha cutoffs occur among successors of a Min position (even depths). It is possible to define another variable, beta, which is established at even depths and generates cutoffs at odd depths. The action of beta cutoffs is exactly the inverse of that for alpha cutoffs. In fact, it is the usual practice of the author who did the programming (Dixon) to write the functions in LISP which evaluate Max position and then have the computer automatically write the corresponding functions for Min positions simply by interchanging Max and Min, alpha and beta, $<$ and $>$, and so on.

The results of applying an Alpha-Beta tree search to kalah are given in Table II. The effect of alpha and beta cutoffs is to make the tree grow more slowly with depth. Thus, the advantage of Alpha-Beta over minimax depends on depth. It is about twice as good at $D_{MAX} = 3$ and about thirty times as good at $D_{MAX} = 6$. This dependence on depth of tree-search procedures is typical.

Hence, it would be convenient to have a depth-independent measure of the relative efficiency of a tree-searching program. Such a measure is DR, the depth ratio, defined as

$$\mathrm{DR} = \frac{\log N}{\log N_{MM}},$$

where N is the number of nodes at the bottom of the tree, and N_{MM} is the number of nodes at the bottom of the tree in a minimax search. DR is a number between 0 and 1 and indicates the effective depth of a search procedure in comparison to the minimax procedure. For example, a DR = 0.667 would indicate that the program in question could search a tree to depth 6 with approximately as much effort as the minimax procedure would need to search to depth 4. Thus DR = 1 for simple minimax procedure.

DR may be converted into relative tree size and an estimate of computer running time, thus:

$$\frac{T}{T_{MM}} \approx \frac{N}{N_{MM}} = \frac{B^{(\mathrm{DR})D}}{B^D} = B^{(\mathrm{DR}-1)D},$$

where N_{MM} is the number of nodes and T_{MM} is the running time of our standard, the minimax procedure.

The Alpha-Beta procedure has a DR of 0.733 at $D_{MAX} = 6$. It is clear that the Alpha-Beta procedure is a substantial improvement in tree-searching technique.

The Alpha-Beta procedure is equivalent to minimax in the sense that the two procedures will always choose the same depth-1 successor as best and will always give the same value for that successor. All the other procedures described in this paper have the same equivalence property with respect to the minimax procedure, except that some programs may choose another depth-1 successor of equal value. If several depth-1 successors have the same final value, and if that value is the maximum depth-1 value, then the one which is first evaluated will be chosen as best. Samuel [16] refers to this as the "hazardless" property.

3. *Fixed Ordering*

The number of cutoffs generated by the Alpha-Beta procedure depends on the order in which the successors are evaluated. Consider Figure 3, for example. If the machine had evaluated P_2, then P_3, then P_1, alpha would have been 0, then 1, then 5, and there would have been no alpha cutoffs at all.

This fact suggests the possibility of improving the Alpha-Beta procedure by ordering successors of a position in order to generate a large number of alpha and beta cutoffs.

This can in fact be done by ordering the successors by their static values. The largest valued successor of a Max position is put first and the reverse order is used for successors of a Min position. This procedure is based on the assumption that the static value of a position is positively correlated with the deeper, backed-up value of that position.

This procedure is equivalent to the Alpha-Beta procedure except that it will sometimes choose a different depth-1 successor but one of equal value. Thus, if the order in which the successors are evaluated is changed, the program may choose a different one as best.

The results of a fixed-order, Alpha-Beta program are shown in Table III. The same starting positions of kalah were used as before so that the data can be compared. Let us consider the typical case of five-in-a-hole kalah searched to $D_{MAX} = 6$. Counting positions at the bottom, this tree has a full size of 292,196 by minimax search. Alpha-Beta needs to look at only 12,359 positions, while fixed ordering cuts the search down to only 2,515 positions. Thus, by this measure, the fixed-ordering program represents an improvement of two orders of magnitude over the simple minimax program. The depth ratio of the tree is 0.589 at $D_{MAX} = 6$.

It is obvious that the program does not run any faster if ordering is done at the bottom level of the tree, since this would only mean that the bottom positions are evaluated twice. However, it is not intuitively clear whether it is worthwhile to order at the next level above the bottom or not. The question of where to stop ordering was experimentally investigated by means of a parameter called SWD. SWD (SWitch Depth) is the number of levels above the bottom of the tree at which the program stops ordering and reverts to the plain Alpha-Beta procedure. Thus, SWD = 0 means that ordering takes place at all levels including the bottom, SWD = 1 means that the bottom level is not ordered, and so on.

TABLE III. FIXED-ORDERING RESULTS*

D_{MAX}		Fixed-ordering,† no. stones per hole						Fixed-ordering, av	Alpha-Beta, av
		1	2	3	4	5	6		
1	NBP	10	10	10	10	10	10	10	10
	Time	0.44	0.43	0.42	0.43	0.43	0.43	0.43	1
2	NBP	37	38	50	31	20	15	32	54
	Time	1.2	1.2	1.8	1.3	1.2	0.99	1.3	1.2
3	NBP	61	83	174	222	228	66	137	276
	Time	2.7	2.9	5.3	5.7	5.4	1.8	3.7	5.5
4	NBP	86	94	433	258	273	106	208	946
	Time	4.4	6.0	19.7	16.8	13.6	5.5	11.0	23.4
5	NBP	322	741	1,482	3,101	1,348	496	1,246	4,037
	Time	12.3	20.1	56.6	64.3	37.8	13.1	34.0	98.3
6	NBP	344	831	3,521	3,804	2,515	976	1,998	12,775
	Time	17.8	54.8	280.5	254.6	107.9	36.6	125.4	364.8
7	NBP	1,037	3,064						
	Time	40.6	118.9						
8	NBP	842	2,269	11,073					
	Time	67.6	160.2	666.9					

* NBP = number of bottom positions; time is given in seconds.
† DR = 0.589 at D_{MAX} = 6.

TABLE IV. SEARCH TIME VERSUS SWD

	3-in-a-hole kalah, D_{MAX} = 4			
	SWD (slow ordering function)			
	0	1	2	3
Time, sec	55.8	25.0	22.4	29.1
Number CONSes	135,504	58,821	50,389	65,753
	SWD (fast ordering function)			
Time, sec	22.8	18.3	19.4	28.0
Number CONSes	53,396	41,134	43,188	62,881

Experimental results showing the effect of SWD on computer running time are shown in Table IV. Early results showed that SWD = 2 was fastest. Running times formed a broad minimum with SWD = 1 and SWD = 3 being within about 30 percent of the minimum. SWD = 0 ran 50 to 100 percent longer. "Number CONSes" in Table IV refers to the number of times the LISP function CONS was called. Each call to CONS consumes one word of computer memory to create new list structure. "CONS" is derived from "construct." The number of CONSes is often a good measure of computer effort.

At a later time, the ordering function was rewritten and made much faster. Results with the new ordering function are also shown in Table IV. SWD = 1 was best this time with SWD = 2 and SWD = 3 close and SWD = 0 much slower. So we can conclude that the best value for SWD depends on the speed of the ordering function in relation to the other functions of the program.

4. *Dynamic Ordering*

Ordering the successors on the basis of static values makes tree-searching go a good deal faster. But the ordering obtained by static values is not always correct. As the deep search progresses under a given position our state of knowledge about that position gradually improves and it becomes possible to make more and more accurate estimates of the true deep value of the position. Occasionally we discover that our original estimate based on the static value was quite wrong and that the position we have chosen to evaluate first will really have a very bad (low for successor of Max position) backed-up value. Intuition suggests that, if not too much work has already been done on the position, it might be wise to stop work, return the estimated bad value, reorder the positions, and make another choice for the first position to evaluate. Then later when the original position is evaluated, we have a larger value of alpha (or a lower value for beta) and will get a great many cutoffs which would otherwise have been missed.

Consider the example shown in Figure 5. The list of numbers after "Fixed Order" shows the order in which the depth-2 positions would be evaluated by a fixed-order program. The × indicates that the position was not evaluated because of an alpha cutoff.

The sequence of events in the dynamic case is as follows: First, the positions at depth 1 are ordered on the basis of their static values. The static values are $P_1 = 10$, $P_2 = 9$, and $P_3 = 6$. The static value of P_2 is called A. This value is important because P_2 is the next choice in case the decision is made to reorder. Next, the machine begins to evaluate P_1. $P_{1,1}$ is evaluated and the result (-1) is compared to A $(+9)$. Since -1 is a great deal lower than 9, the machine decides to abandon P_1 and reorder. The new order is $P_2 = 9$, $P_3 = 6$, $P_1 = -1$.

Now A is set to 6 and P_2 and is evaluated. $P_{2,1}$ has the value 5 which is less than A, but not very much less. The machine decides to continue with P_2. $P_{2,2}$ and $P_{2,3}$ are evaluated and the Min value 5 is backed up to P_2. Alpha is now assigned the value 5 and the rest of successors are evaluated by the Alpha-Beta procedure. An alpha cutoff occurs at $P_{3,1}$, saving two depth-2 evaluations, and again at $P_{1,1}$, saving another two evaluations. A total of six depth-2 positions are evaluated by the dynamic-ordering procedure. This compares with seven positions evaluated by the fixed-order procedure.

The tree in Figure 5 has a maximum depth of only 3. This is the most shallow tree on which dynamic reordering can be used. On deeper trees the procedure is

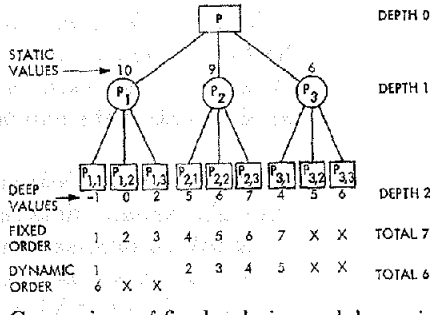

FIG. 5. Comparison of fixed ordering and dynamic ordering

more complex. If the depth of the tree is 4 or more, then we can dynamically reorder the positions at level 3. Since these are the successors of a Min position, we attempt to explore the lowest valued position first. B is the name of the second best (second lowest valued) position. The decision to reorder will be made if deeper search reveals that the value of the best (lowest valued) position is probably going to be too much larger than B. Thus the procedure for B cutoffs is analogous to that for A cutoffs. The roles of "greater" and "less" are interchanged. The relation between A and B is just the same as the relation between alpha and beta.

If the tree has a maximum depth of 5 or greater, then two or more A's may exist in the tree at the same time. One will be generated at depth 1 (A_1) and another at depth 3 (A_3). The value of A below level 3 is the larger of A_1 or A_3. An A cutoff below level 3 will cause reordering at level 1 if the returned value meets cutoff criteria with respect to A_1. Otherwise reordering will take place at level 3. If the tree has a maximum depth of 6 or greater then there will be two or more B's and an analogous procedure can be used to resolve conflicts between the several B's. The procedure is recursive so it can be used on trees of any depth and reordering can take place at any level except the bottom.

This is called the A-B, Alpha-Beta procedure. B is the variable used to decide about reordering successors of a Min position. It should be noted that A and B are always shallow or estimated values, while alpha and beta are always deep values which have been backed up from the bottom of the tree.

The decision to reorder is the crucial element in the success of the A-B, Alpha-Beta procedure. An early program, called deep A-B, made the reorder decision by comparing A or B with deep or backed-up values. Deep A-B proved to be slower than fixed-order Alpha-Beta.

Too much time was invested in the first choice before the reorder decision was made. Another program, called SHALLOWAB, made the reorder decision by comparing A and B with shallow static values at each level while the program was still working its way toward maximum depth. SHALLOWAB proved to be better than ordered Alpha-Beta where parameters were properly adjusted.

An important parameter of the A-B procedure is DELTA. DELTA is the quantity which determines how stubborn the procedure is about reordering. More exactly, reordering takes place if the current estimated value of the first choice is less than A − DELTA or greater than B + DELTA. Intuitively, we would expect that with a large enough DELTA, the procedure would never reorder and would run just the same as the fixed-order procedure. Conversely, with a small DELTA, the program would be slowed by an excessive number of reorderings. This is, in fact, what happens. Typical results are shown in Table V. NVAL in Table V is the number of times the value function is called. NVAL is a measure of the size of the tree, but it is larger than NBP. Time in Table V is given in seconds. NCONS is the number of times the LISP function CONS is called. NREO is the number of reorderings which occur.

It is evident that DELTA must be carefully chosen if SHALLOWAB is to be more efficient than the fixed-order program. The running times and tree size of SHALLOWAB are shown in Table VI. SHALLOWAB produces a tree that is typically 10 percent smaller than ordered Alpha-Beta and has a DR of 0.582. This advantage tends to increase with depth. Epsilon, shown in Table VI, is discussed next.

Since the SHALLOWAB program makes reordering decisions by comparing

TABLE V. EFFECTS OF DELTA ON SHALLOWAB PROGRAM

DELTA	2-in-a-hole kalah, $D_{MAX} = 5$				3-in-a-hole kalah, $D_{MAX} = 5$			
	NVAL	Time, sec	NCONS	NREO	NVAL	Time, sec	NCONS	NREO
0	1,915	55.6	128,015	150	2,204	76.1	173,754	143
1	1,395	40.1	94,058	89	1,700	60.0	135,361	86
2	1,242	35.8	81,635	54	2,401	82.6	186,575	68
3	1,437	37.2	85,229	34	2,513	83.3	186,617	41
4	1,121	28.3	65,329	20	2,620	84.2	189,085	29
5	797	20.4	46,652	3	2,689	83.7	186,950	19
6	996	20.2	47,192	6	2,730	76.2	176,351	3
7	1,006	20.1	47,292	0	2,789	76.8	179,367	0
8	1,015	20.3	47,381	0	2,810	77.6	179,577	0

TABLE VI. SHALLOWAB RESULTS*

D_{MAX}	SHALLOWAB,† No. stones per hole						SHALLOWAB, av	Fixed-order, av
	1	2	3	4	5	6		
1 NBP	10	10	10	10	10	10	10	10
Time	0.43	0.43	0.42	0.42	0.52	0.43	0.44	0.43
2 NBP	37	38	50	31	20	15	32	32
Time	1.2	1.7	1.3	2.0	1.3	1.0	1.4	1.3
3 NBP	61	83	146	282	194	68	139	137
Time	3.6	4.0	5.9	8.6	5.8	2.4	5.0	3.7
4 NBP	86	94	360	547	150	99	206	208
Time	7.0	9.0	23.5	32.7	14.4	7.1	15.6	11.0
5 NBP	322	608	1,637	2,695	785	476	1,087	1,246
Time	20.3	26.9	94.0	126.0	35.9	18.0	53.5	34.0
6 NBP	319	588	2,664	4,665	1,120	1,675	1,838	1,998
Time	30.6	64.8	275.5	430.0	89.2	80.9	161.3	125.4
7 NBP	968	2,059						
Time	66.3	158.4						
8 NBP								
Time								

* NBP = number of bottom positions; time is given in seconds.
† SWD = 1; SWDD = 1; DELTA = 5; EPSILON = 1.5; DR = 0.582 at $D_{MAX} = 6$.

values derived at different depths, it is natural to ask if such values are really comparable. Figure 6(a) shows that they are not. There is a pronounced rhythmic rise and fall of the value-versus-depth function. This is caused by the fact that each player has a temporary advantage when it is his turn to move. This causes values from even depths to be an average of 1.5 smaller than values from odd depths. Thus, values obtained from even depths would tend to cause too many A-type reorderings, whereas values obtained from odd depths would tend to cause too many B-type reorderings. However, the effect can be canceled by adding 1.5 to the value of all even depth positions. The results of adding this constant, called epsilon, are shown in Figure 6(b).

Many variations of SHALLOWAB were tested in attempts to improve its per-

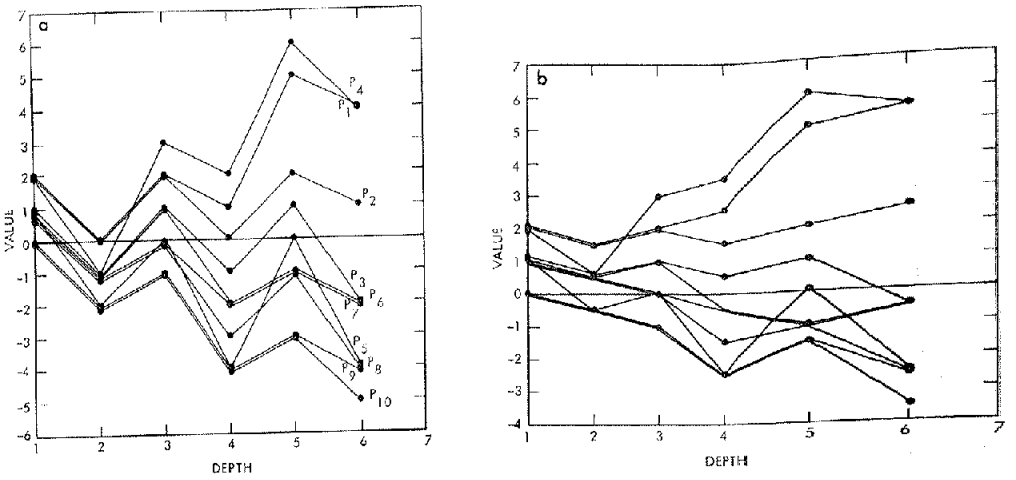

FIG. 6

(a) Value versus depth, 4-in-a-hole kalah

(b) Value plus epsilon versus depth, 4-in-a-hole kalah

TABLE VII. SAVELIST RESULTS*

D_{MAX}		SAVELIST,† No. stones per hole						SAVELIST, av	Fixed-order, av
		1	2	3	4	5	6		
1	NBP	10	10	10	10	10	10	10	10
	Time	0.43	0.43	0.42	0.42	0.52	0.43	0.44	0.43
2	NBP	37	38	50	31	20	15	32	32
	Time	1.2	1.7	1.3	2.0	1.3	1.0	1.4	1.3
3	NBP	61	83	146	282	194	68	139	137
	Time	2.7	2.5	5.3	6.7	5.1	1.8	4.0	3.7
4	NBP	86	94	353	543	150	99	221	208
	Time	4.9	7.1	18.3	22.6	11.4	5.1	11.6	11.0
5	NBP	320	595	1,254	2,294	785	476	954	1,246
	Time	14.2	23.7	66.6	100.9	25.9	13.0	40.7	34.0
6	NBP	300	492	2,045‡	4,179	1,119	1,675	1,625	1,998
	Time	21.3	52.5	195.0‡	334.3	69.9	61.2	122.3	125.4

* NBP = number of bottom positions; time is given in seconds.
† DELTA = 4; SWD = 1; SWDD = 1; DR = 0.572 at D_{MAX} = 6.
‡ Estimated by extrapolation since this problem would not fit into memory.

formance. The one variation which proved most efficient is a program called SAVELIST. When this program decides to reorder, it saves the tree which has already been generated in the form of a list. Thus it never has to generate any position twice.

Typical SAVELIST results are shown in Table VII. It is about 20 percent more efficient than the fixed-order program, and has a DR of 0.572.

Tables VI and VII refer to the parameter SWDD. This parameter is the number of levels at which reordering is inhibited, so that only fixed ordering takes place. The optimum value for this parameter was found to be 1. Thus, in a tree of depth 6,

dynamic ordering takes place at the top four levels, fixed ordering at depth 5, and simple Alpha-Beta with no ordering at depth 6.

It can be concluded that dynamic ordering represents a modest improvement over fixed ordering.

5. *Perfect Ordering*

Perhaps the reason that dynamic ordering produced so modest an improvement is that the ordering of the successors is already nearly perfect. Theoretical analysis shows that if the ordering is perfect so that every possible alpha or beta cutoff actually occurs, the tree would grow at about the one-half power of its usual rate. More exactly:

THEOREM 1. *If perfect ordering is achieved at every level, so that every possible alpha or beta cutoff occurs, then the number of positions at the bottom of the tree of depth D and constant branching factor B is:*

$$N_D = 2B^{D/2} - 1 \qquad \text{for } D \text{ even},$$

$$N_D = B^{(D+1)/2} + B^{(D-1)/2} - 1 \qquad \text{for } D \text{ odd}.$$

This theorem is attributed to Michael Levin in [6]. It is illustrated in Figure 7. A proof of the theorem is now given since, to the authors' knowledge, no proof has been previously published.

PROOF. Let us call the number of positions to be evaluated N_D where D is the maximum depth of the tree. Figure 7 is an aid to following this proof. It is convenient to count N_D in two parts. Those under P_1 we call X_D. The others we call Y_D. Thus by definition,

$$N_D = X_D + Y_D. \tag{1}$$

The first alpha cutoffs occur at $D = 2$. There are no cutoffs in the X part so

$$X_2 = B. \tag{2}$$

The X part of the search establishes an optimum value for alpha, so in the Y part all possible alpha cutoffs will be attained. At $D = 2$ we need to evaluate only one successor to each position at the $D = 1$ level. Thus,

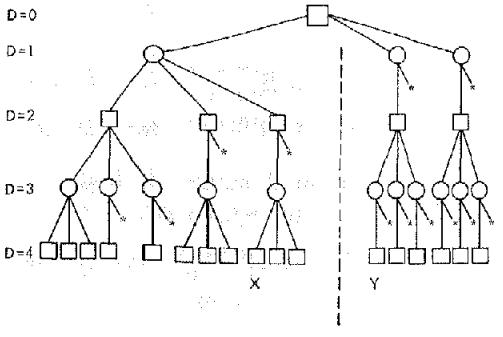

FIG. 7. Alpha and beta cutoffs in a ternary tree which is perfectly ordered. * means cutoff.

$$Y_2 = B - 1, \tag{3}$$

$$N_2 = X_2 + Y_2 = B + B - 1 = 2B - 1. \tag{4}$$

Now the theorem for $D = 2$ is established. Next we prove the theorem for all $D \geq 2$ by induction. But first, some lemmas are needed.

LEMMA 1. *The Alpha-Beta search under P_1 to depth D must evaluate the same number of positions as a search to depth $D - 1$ under P.*

The lemma is true by symmetry. The search under P_1 is the same as the search under P except that Max positions are exchanged for Min positions. Alpha and beta start at their initial values in both cases. The Alpha-Beta algorithm is symmetrical with respect to Max and Min positions provided we assume that each set of successors is in perfect order (monotonic increasing for successors of a Max position and monotonic decreasing for successors of a Min position). Hence we have a recursive formula for the X part:

$$X_D = N_{D-1}, \qquad D \geq 2. \tag{5}$$

Now we consider the Y part.

LEMMA 2. *Every possible alpha cutoff occurs in the Y part of the tree.*

The X part of the search under P_1 establishes an optimum value for alpha. No other successor has a higher backed-up value. Because of the perfect ordering of successors at every level, an alpha cutoff will take place every time the first successor of a Min position is evaluated.

LEMMA 3. *No beta cutoffs will take place in the Y part of the tree.*

In order to have a beta cutoff, there must be at least two successors to a Min position, but an alpha cutoff always occurs at the first successor.

Lemma 2 implies that every Min position in the Y part of the tree will have exactly one successor. If D is odd and we increase D by one, we add exactly one successor to each Min position at the bottom of the tree in the Y part. Hence,

$$Y_D = Y_{D-1} \quad \text{for} \quad D \text{ even}. \tag{6}$$

Lemma 3 implies that every Max position in the Y part of the tree will have exactly B successors. Hence, if D is even and we increase D by one, we must give B successors to each Max position at the bottom of the tree. Thus,

$$Y_D = BY_{D-1} \quad \text{for} \quad D \text{ odd}. \tag{7}$$

LEMMA 4.

$$Y_D = (B - 1)B^{(D-2)/2} \quad \textit{for} \quad D \textit{ even}, \tag{8}$$

$$Y_D = (B - 1)B^{(D-1)/2} \quad \textit{for} \quad D \textit{ odd}. \tag{9}$$

We prove this lemma by induction. Assume (8) holds for $D - 1$. D will be odd in this case. Replacing D by $D - 1$ in (8) we get

$$Y_{D-1} = (B - 1)B^{(D-3)/2} \quad \text{for} \quad D \text{ odd}. \tag{10}$$

By (7) we have

$$Y_D = BY_{D-1} = (B - 1)B^{(D-1)/2} \quad \text{for} \quad D \text{ odd}. \tag{11}$$

This shows that (9) follows from (8). Now we show that (8) follows from (9). Replacing D with $D - 1$ in (9) we get

$$Y_{D-1} = (B - 1)B^{(D-2)/2} \quad \text{for} \quad D \text{ even.} \tag{12}$$

Applying (6),

$$Y_D = Y_{D-1} = (B - 1)B^{(D-2)/2} \quad \text{for} \quad D \text{ even.} \tag{13}$$

Since we have already shown that the lemma is correct for $D = 2$ in eq. (3), this concludes the proof of Lemma 4.

Now we must add in the X part and prove the main theorem by induction. First we show that the expression for odd D follows from the equation for even D. Replacing D by $D - 1$ in the expression for even D we get

$$N_{D-1} = 2B^{(D-1)/2} - 1 \quad \text{for} \quad D \text{ odd.} \tag{14}$$

Applying (5) we get

$$X_D = N_{D-1} = 2B^{(D-1)/2} - 1 \quad \text{for} \quad D \text{ odd.} \tag{15}$$

Substituting (9) and (15) in (1) we get

$$N_D = 2B^{(D-1)/2} + (B - 1)B^{(D-1)/2} - 1 \quad \text{for} \quad D \text{ odd;} \tag{16}$$

simplifying,

$$N_D = B^{(D+1)/2} + B^{(D-1)/2} - 1 \quad \text{for} \quad D \text{ odd.} \tag{17}$$

Equation (17) agrees with the theorem. Now we show that the even part of the theorem follows from the odd part. Replacing D with $D - 1$ in (17) we get

$$N_{D-1} = B^{(D)/2} + B^{(D-2)/2} - 1 \quad \text{for} \quad D \text{ even.} \tag{18}$$

Applying (5) we get

$$X_D = N_{D-1} = B^{(D)/2} + B^{(D-2)/2} - 1 \quad \text{for} \quad D \text{ even.} \tag{19}$$

Substituting (19) and (8) in (1) we get

$$N_D = B^{(D)/2} + B^{(D-2)/2} - 1 + (B - 1)B^{(D-2)/2} \quad \text{for} \quad D \text{ even.} \tag{20}$$

Simplifying we get

$$N_D = 2B^{D/2} - 1 \quad \text{for} \quad D \text{ even,} \tag{21}$$

which agrees with the even part of the theorem. Since we have already established the theorem for $D = 2$ in eq. (4), this completes the proof of the theorem.

The first few N_D's given by these formulas are:

		No cutoff
N_2	$2B - 1$	B^2
N_3	$B^2 + B - 1$	B^3
N_4	$2B^2 - 1$	B^4
N_5	$B^3 + B^2 - 1$	B^5
N_6	$2B^3 - 1$	B^6

Note that the rate of growth with respect to depth is not uniform. For D even,

$$\frac{N_D}{N_{D-1}} = \frac{2B^N - 1}{B^N + B^{N-1} - 1} \simeq \frac{2B^N}{B^N + B^{N-1}} = 2\left(1 + \frac{1}{B}\right)^{-1} \simeq 2\left(1 - \frac{1}{B}\right).$$

For D odd,

$$\frac{N_D}{N_{D-1}} = \frac{B^N + B^{N-1} - 1}{2B^{N-1} - 1} \simeq \frac{1}{2}B + \frac{1}{2}.$$

Thus in the limiting case of large B, the size of the tree increases by the factor 2 from odd to even depths and by the factor $B/2$ from even to odd depths.

Now let us consider the theoretical advantage of Alpha-Beta over minimax, in the case where we get every possible cutoff. For even D,

$$\mathrm{DR}_{PO} = \frac{1}{D} LN_B(N) = \frac{1}{D} LN_B(2B^{D/2} - 1);$$

$$\mathrm{DR}_{PO} \simeq \frac{1}{D} LN_B(2B^{D/2}) = \frac{1}{D} LN_B(B^{D/2}) + \frac{1}{D} LN_B 2;$$

$$\mathrm{DR}_{PO} \simeq \frac{1}{2} + \frac{1}{D} LN_B 2.$$

For the typical cases, $B = 10$, $D = 6$, and $D = 4$,

$$\mathrm{DR}_{PO} \simeq \frac{1}{2} + \frac{1}{6} LN_{10} 2 = \frac{1}{2} + \frac{1}{6} 0.301 \simeq 0.5 + 0.05;$$

$$\mathrm{DR}_{PO} \simeq 0.55 \quad \text{for} \quad D = 6;$$

$$\mathrm{DR}_{PO} \simeq 0.575 \quad \text{for} \quad D = 4.$$

The DR = 0.572 of the SAVELIST procedure at $D_{\mathrm{MAX}} = 6$ is quite close to the theoretical limit of 0.550.

To confirm the hypothesis that the limit of perfect ordering was being approached, a program was written which artificially put the successors at every level into perfect order. This program is very slow but it does determine the size of a tree which is perfectly ordered at every level. The results are shown in Table VIII.

6. *Comparison With Other Procedures*

We now discuss some other efficient tree-searching procedures which have been proposed and compare them with the A-B procedure. We consider five procedures, proposed by Amarel [2], Doran and Michie [5], Nilsson [14], Slagle and Bursky [20], and Samuel [16]. All these procedures have some form of what Amarel calls "attention control." They periodically reconsider the partly searched tree to decide where to work next. Thus they make immediate use of the information gained from the expansion of a node in an attempt to keep the procedure working on that part of the tree which appears to be most promising at any time.

Samuel's procedure [16] is the one most similar to A-B. The main difference is that Samuel makes reorder decisions only at even depths; thus it is like one-half of the A-B procedure. It generates A cutoffs but no B cutoffs. Samuel's procedure uses

TABLE VIII. PERFECT ORDERING COMPARED WITH OTHER PROGRAMS
Figures given are numbers of nodes at the bottom of the tree (NBP).

	Stones per hole	*Minimax*	*Alpha-Beta*	*Fixed-order, SWD = 1*	*SHALLOWAB, SWD = 1, SWDD = 1, DELTA = 5*	*SAVELIST, SWD = 1, SWDD = 1, DELTA = 4*	*Perfect-order*	
							SWD = 1	*SWD = 0*
$D_{MAX} = 2$	1	98	70	37	37	37	37	17
	2	98	75	38	38	38	38	17
	3	106	66	50	50	50	37	19
	4	116	60	31	31	31	31	19
	5	108	37	20	20	20	20	19
	6	60	15	15	15	15	15	15
$D_{MAX} = 3$	1	676	380	61	61	61	61	61
	2	724	381	83	83	83	83	74
	3	818	300	174	146	226	156	117
	4	1,022	275	222	282	114	157	118
	5	1,055	253	228	194	181	147	111
	6	329	68	66	68	76	76	72
$D_{MAX} = 4$	1	4,380	1,318	86	86	90	111	101
	2	5,512	1,066	94	94	100	100	100
	3	6,834	1,285	433	360	346	220	196
	4	9,682	1,237	258	547	192	263	224
	5	6,727	639	273	150	223	221	151
	6	1,907	131	106	99	110	103	110
$D_{MAX} = 5$	1	19,168	4,168	322	322	320	90	
	2	41,014	5,349	741	608	595	100	
	3	61,241	5,686	1,482	1,637	1,254	229	
	4	125,843	5,213	3,101	2,695	2,294	227	
	5	44,695	3,049	1,348	785	785	156	
	6	12,441	759	496	476	476	519	

a delta of zero, so presumably it does more flitting about than the *A-B* procedure. Samuel also uses two types of forward pruning.

The other four tree-search routines are intended for general problem-solving rather than just game-playing. They search the tree to find an answer rather than searching to a fixed depth. They can, of course, be adapted to game-playing with slight modification.

All these routines make an estimate of the difficulty of solving each node and then back up these estimates, taking account of parallel problems, to the top of the tree. It is then possible to descend the tree, choosing the most meritorious successor at each fork, eventually arriving at the most meritorious unexpanded node. The Amarel procedure includes work already done from the top of the tree in the estimate of the difficulty of solving each node. Thus the Amarel procedure is intended to find the shortest proof rather than the one which can be most quickly found. One version of Doran and Michie's graph traverser also has this feature.

In backing up the estimates of difficulty, most of the procedures take account of parallel problems at either conjunctive nodes or disjunctive nodes. Doran and Michie's graph traverser, however, can handle only disjunctive nodes. Slagle's MULTIPLE procedure is the most general since it can handle an arbitrary Boolean function at each node.

Cutoffs take a different form in these procedures since they search for a solution wherever it may appear rather than searching the tree to a fixed depth. The alpha or beta cutoff does not occur. Cutoffs do occur in the sense that unpromising nodes are not expanded; however, these are always tentative cutoffs, while alpha and beta cutoffs are permanent. The sharp distinction between forward and backward pruning also disappears in most of these procedures. Certain versions of Amarel's procedure and Doran and Michie's procedure use operator selection rules which have the irrevocable character of forward pruning.

In general, the relationship of the A-B procedure to these other procedures is that A-B is similar in general concept but specialized for the task of game-playing. A-B is also a simplification of most of these procedures since A-B does not estimate the effort required to expand a node nor is any consideration given to the size of parallel portions of the tree.

7. *Conclusions*

Several fast tree-searching procedures have been described. These have been tested on six different variations of the game kalah. The fixed-ordering program is much faster than the Alpha-Beta or minimax programs. The dynamic-ordering programs are only slightly faster than fixed ordering because the limit of perfect ordering is being approached. The parameters used in these search routines must be adjusted for best results and optimum values will depend on relative speeds of the various parts of the program.

Future efforts to produce faster tree-searching routines by increasing the number of alpha and beta cutoffs through improved ordering and reordering procedures are not likely to yield substantial improvements for the game of kalah, since the limit of perfect ordering is now being approached.

The Alpha-Beta procedure produces a worthwhile improvement in searching efficiency at depths of 2 or more. The fixed-order procedure produces a worthwhile improvement at depths of 4 or more. Dynamic ordering becomes worthwhile at depths of 6 or more. Hence, we can conclude that the more complex procedures are more useful for deeper trees.

REFERENCES

* Not cited in text

1. Adelson-Velskiy, G. M., Arlasarov, V. L., and Uskov, A. G. Programme playing chess. Rep. on Symp. on Theory and Computing Methods in the Upper Mantle Problem. (Translation from Russian; source unknown.)
2. Amarel, Saul. An approach to heuristic problem solving and theorem proving in the propositional calculus. In Hart, J. F., and Takasu, S. (Eds.), *Systems and Computer Science*, U. of Toronto Press, Toronto, Ontario, Canada, 1967, pp. 125–220.
3. Baylor, G. W., and Simon, H. A. A chess mating combinations program. Proc. AFIPS 1966 Spring Joint Comput. Conf., Vol. 28, pp. 431–447.
4. Bernstein, A., Roberts, M. DeV., Arbuckle, T., and Belsky, M. A. A chess playing program for the IBM 704. Proc. 1958 Western Joint Comput. Conf., Vol. 13, pp. 157–159.
5. Doran, J. E., and Michie, D. Experiments with the graph traverser program. *Proc. Roy. Soc.* {A}, *294*, 1437 (1966), 235–259.
6. Edwards, D. J., and Hart, T. P. The α-β heuristic. Artif. Intel. Memo No. 30 (revised), MIT Research Laboratory of Electronics and Computation Center, Cambridge, Mass., Oct. 28, 1963.

7. ERNST, G. W., AND NEWELL, A. *Generality and GPS*. Center for Study of Information Processing, Carnegie Inst. of Technology, Pittsburgh, Pa., Jan. 1967.
8. FEIGENBAUM, E., AND FELDMAN, J. (EDS.). *Computers and Thought*. McGraw-Hill, New York, 1963.
*9. GELERNTER, H. Realization of a geometry theorem proving machine. Proc. Int. Conf. on Information Processing, UNESCO House, Paris, 1959, pp. 273–282. (Reprinted in [8], pp. 134–152.)
*10. GELERNTER, H., HANSEN, J. R., AND LOVELAND, D. W. Empirical explorations of the geometry theorem machine. Proc. 1960 Western Joint Comput. Conf., Vol. 17, pp. 143–147. (Reprinted in [8], pp. 153–163.)
11. KISTER, J., STEIN, P., ULAM, S., WALDEN, W., AND WELLS, M. Experiments in chess. *J. ACM 4*, 2 (April 1957), 174–177.
*12. MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., HART, T. P., AND LEVIN, M. I. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Mass., 1962.
13. NEWELL, A., SHAW, J. C., AND SIMON, H. A. Chess playing programs and the problem of complexity. *IBM J. Res. Develop. 2* (Oct. 1958), 320–335.
14. NILSSON, N. J. Searching problem-solving and game-playing trees for minimal cost solutions. IFIP Congress 68, Booklet H: Applications 3, pp. H125–H130.
15. SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM J. Res. Develop. 3* (July 1959), 211–229. (Reprinted, with minor additions and corrections, in [8], pp. 71–105.)
16. SAMUEL, A. L. Some studies in machine learning using the game of checkers. II—Recent Progress. *IBM J. Res. Develop. 11*, 6 (Nov. 1967), 601–617. Stanford Artif. Intel. Project Memo No. 52, Stanford U., Stanford, Calif., June 5, 1967.
*17. SIMON, H. A., AND SIMON, P. A. Trial and error search in solving difficult problems: Evidence from the game of chess. *Behavioral Sci. 7* (Oct. 1962), 425–429.
18. SLAGLE, J. R. Game Trees, m & n minimaxing, and the m & n alpha-beta procedure. Artif. Intel. Group Rep. No. 3, UCRL-4671, Lawrence Radiation Laboratory, U. of California, Livermore, Calif., Nov. 1963.
19. SLAGLE, J. R. A multipurpose theorem proving heuristic program that learns. Proc. IFIP Congress 65, Vol. 2, pp. 323–324 (Spartan Books, Washington, D. C.).
20. SLAGLE, J. R., AND BURSKY, P. Experiments with a multipurpose, theorem-proving heuristic program. *J. ACM 15*, 1 (Jan. 1968), 85–99.
*21. SLAGLE, J. R. A heuristic program that solves symbolic integration problems in freshman calculus. *J. ACM 10*, 4 (Oct. 1963), 507–520. (Reprinted in [8], pp. 191–203.)
*22. TURING, A. M. Digital computers applied to gains. In Bowden, B. V. (Ed.), *Faster Than Thought: A Symposium on Digital Computing Machines*, Pitman, London, 1953, Ch. 25, pp. 286–310.
*23. WEISSMAN, C. *LISP 1.5 Primer*. Dickenson Pub. Co., Belmont, Calif., 1967.

RECEIVED AUGUST, 1967; REVISED OCTOBER, 1968