Figure 1: White to play. This example illustrates some of the basic problems of strategy and tactics that must be evaluated by any chess playing computer in a typical position. The computer (White) must evaluate a variety of possibilities: two good first moves for White include 1 R-B7 and 1 BxN ch. 1 R-B7 threatens BxB. Therefore Black must either exchange Bishops or gain time by the counterattack 1...B-K4. If 1...BxB, White must complete the exchange by playing 2 RxB or 2 BxN ch, and so on. The position is analyzed in detail in the game tree shown in figure 2.

A Computer Chess Tutorial

Norman D Whaland
430 E 9th St, Apt #15
New York NY 10009

On February 20 1977, the Minnesota Open chess tournament was won by a computer program, Northwestern University's Chess 4.5. This was a far better result than any program had previously achieved, considering that all the other entrants in the tournament were human beings. An improved version, Chess 4.6, went on to wrest the world computer chess championship from the Soviet program KAISSA (see "The Second World Computer Chess Championships" by Peter Jennings, January 1978 BYTE, page 108). Professional chess players are beginning to worry about the competition from machines. They would seem to have little to fear at the moment, however. The consensus is that Chess 4.5's tactical skill is impressive but its strategy is weak.

Against such competition, what can a personal computer experimenter expect to accomplish? Perhaps a great deal. There have been few new ideas in computer chess since Claude Shannon (see references) outlined the basic principles in a paper published in 1950. (The superiority of Chess 4.6 is due

primarily to faster hardware.) Experimenters can participate in the search for the conceptual breakthroughs that will be needed before computer programs can be a match for the best human players. With that thought in mind, this article deals with the questions: What is a good structure for a chess program? What are the major functions that it must perform? In what directions can we seek innovations?

The Game Tree

To get a notion of what a chess program must do, let's look at a position from an actual game (see figure 1). First we must grasp the important features of the position. White has an extra pawn, a passed pawn far from Black's King. Black's mobility is very limited: neither the Knight nor the Rook can move. Black's Bishop is attacking White's Rook and, indirectly, the Bishop behind it. Of less importance, because of Black's lack of mobility, is the fact that two of White's pawns are unguarded. White's task is to save

---

GLOSSARY

Analysis: the calculation of variations in order to assess a position or find the best move.

Backward pawn: a pawn that lags behind the pawns on the adjoining files. When the opponent has no pawn on the file, a backward pawn is usually a serious weakness.

Development: the process of initially moving the pieces from their original squares.

Diagonal: a diagonal row of squares on the chessboard.

File: a vertical row of squares on the chessboard.

Material: the chess pieces considered as assets. A pawn is traditionally considered to have a material value of one unit. Programs often use smaller units to avoid using fractions for positional advantages.

Minor piece: a Knight or a Bishop.

Passed pawn: a pawn not hindered by enemy pawns on its file or on adjoining files.

Piece: a chess piece other than a pawn.

Positional advantage: any advantage other than an advantage in material.

Rank: a horizontal row of squares.

Strategy: that aspect of chess concerned with long-range planning.

Tactics: that aspect of chess concerned with move-by-move changes in the position. Tactics include the methods for winning material and advancing strategic plans.

Variation: a sequence of moves considered as one of several from a given starting position.
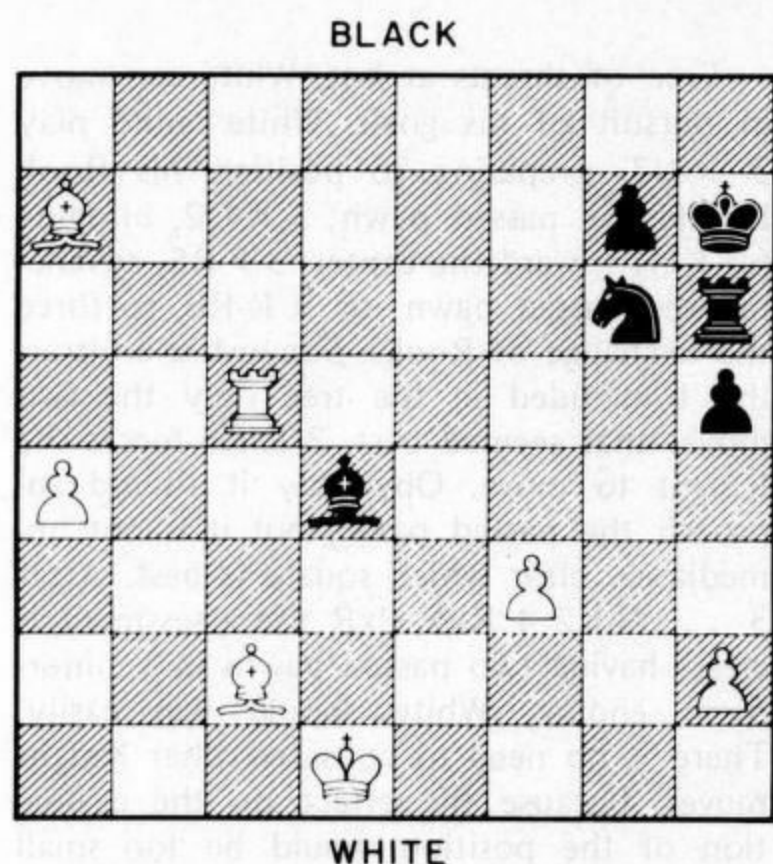
his Rook and to profit from Black's lack of mobility. White should win if he can find satisfactory solutions to these problems.

Next we calculate variations — sequences of moves that we would visualize in an actual game before deciding on a move to play. We will follow a systematic procedure that will serve as a first approximation to a computer program. We construct a tree whose nodes represent positions and whose edges represent moves. The variations are the paths from the root to the leaves. Initially, the tree will consist of one node representing the given position. We expand the tree as follows:

Expansion — Choose a leaf that has not been marked as final. (If one cannot be found, the expansion phase is ended.) Either mark it as final or select a set of legal moves in the position represented by the node. To the leaf attach *sons* representing the positions reached by the moves. Repeat from the beginning.

This procedure might yield the tree shown in figure 2. The size of the tree has been limited somewhat for illustrative purposes. Some of the variations I considered and rejected are not included. Most programs generate much larger trees since it is hard to build into a program the chess knowledge needed for rigorous selection of moves. The length of paths in the tree is expressed in *plies* (*half-moves*). A move consists of a play by one player and a response by the other; a ply is a move by one player alone. Because the term *move* can be confusing (the chess literature speaks of looking three *moves* ahead for example, but are two or three moves by the opponent meant to be included?), in discussions of chess program-

ming one speaks more precisely of a 5 or 6 ply look ahead.

In the expansion procedure, no rule was given for deciding whether to expand a node or for selecting the moves. To gain insight into the way human players make these choices, let us consider the variation that runs down the right side of the tree. In the initial position, Black threatens . . . BxR. White can either make a counterthreat or move his Rook to guard the Bishop. Thus the possible moves include 1 BxN ch, 1 R-B7, and 1 R-R5. I rejected the last alternative because the Rook would have less mobility on R5 and it seemed unimportant to keep it on the fifth rank. 1 R-B7 threatens BxB and moving the Bishop to another diagonal allows B-K3, attacking Black's Rook. Therefore, Black must either exchange bishops or gain time by the counterattack 1 . . . B-K4. If 1 . . . BxB, White must complete the exchange by playing 2 RxB or 2 BxN ch. The latter move was omitted because the reply 2 . . . RxB leads to the position at node 13 (see figure 2), already seen to be unsatisfactory for White. After 2 RxB White threatens R-R6 followed by the exchange of all the pieces and the triumphant advance of the Queen's Rook pawn (QRP). Black must play 2 . . . K-N1 or 2 . . . K-R1. The square closer to the center was chosen on general principles.

Figure 2: A game tree developed from the position in figure 1. Each node represents a position; the root, the initial position. The move leading to the position is written in the top of the box, the evaluation of the position in the bottom. The number above the box identifies the node. A node's ply number is its distance from the root.
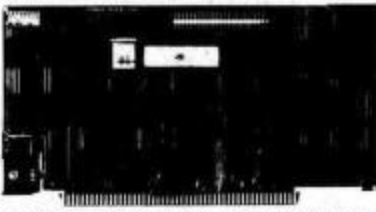
Free of threats at last, White can move in pursuit of his goals. White could play 3 R-N7, preparing to position his Rook behind the passed pawn; 3 K-Q2, bringing his King toward the center; 3 P-R5, advancing the passed pawn; or 3 R-R6, to force the exchange of Rooks. Somewhat arbitrarily, I included in the tree only the two moves that seemed best. 3 R-R6 forces the Knight to move. Obviously it should approach the passed pawn, but it is not immediately clear which square is best. After 3 ... N-K2 4 RxR PxR the assessment is clear: having two passed pawns in a minor-piece ending, White should win easily. There is no need to consider other Knight moves, because the effect on the evaluation of the position would be too small to affect White's choice of move in the initial position.

From this brief discussion we can see some of the factors that determine the selection of moves. When there is a definite threat, it is necessary either to answer the threat directly or to make a counter-threat. Otherwise you must decide which goals are most important and choose the moves that best advance these goals. When two moves have similar effects, not much is learned by including both in the tree, particularly at a deep level.

We can also see some of the reasons for terminating a node (that is, choosing not to expand it). In this example, a node is terminated when the position can be evaluated sufficiently well or when the previous move was not forcing and the side to move has no forcing move that accomplishes anything. At node 14, for example, it is already clear that White doesn't have a won position, and it follows that one of his moves must have been a mistake. Thus we can evaluate the position sufficiently well (but not accurately: further analysis would lower the estimated evaluation given in the figure). At node 5 White has the forcing move 3 B-K3, but after 3 ... R-R1 his position hasn't improved. We consider these moves but don't add them to the tree, because the resulting position is merely compared with the position at node 5, not evaluated.

Once the tree is complete, the next step is to evaluate the terminal positions:

Evaluation — Label each leaf with the value of the position from the point of view of the player whose turn it is to move in the *initial* position. Positive values mean the player has the advantage; negative values mean the player's opponent does. A value of ±1 means an advantage

barely enough to win; a value of ±2 means an easy win (see figure 2).

In the present example, material, mobility and pawn structure were the most important factors in making the evaluation. In a middle game position, King safety would also be taken into consideration.

The final step is a completely mechanical procedure called the *minimax* algorithm, which is guaranteed to choose the best move provided the evaluations are accurate and that the best move at each node is included in the tree.

> Backup — Select an unevaluated node, all of whose sons have been evaluated. If the node is at even ply, label it with the maximum of the sons' values; at odd ply, choose the minimum. Repeat from the beginning until all of the nodes have been evaluated. Then choose the move leading to the ply-1 node with the greatest value.
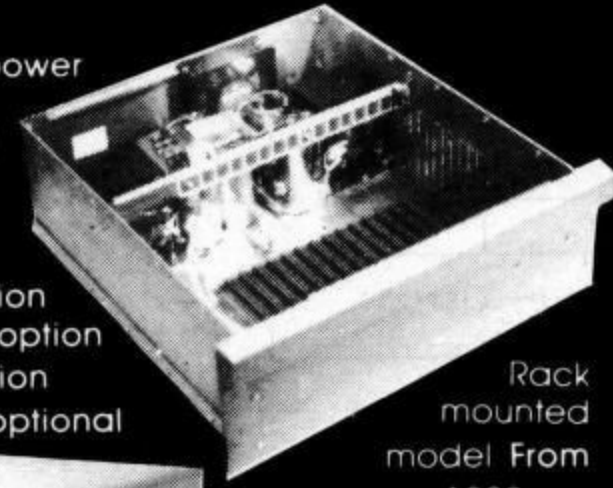
This method of assigning evaluations to non-terminal nodes is based on the assumption that each player always makes the best move. The minimax algorithm will *not* always choose the move that affords the best winning chances against a weak opponent.

Our 3 part procedure for generating a game tree is somewhat unnatural. For one thing, a person analyzing a position would return to the expansion phase if the moves originally selected didn't work out as well as expected. Also, the evaluation phase reflects the human assessment process poorly. No provision is made for recording degree of confidence in the evaluation. Human players make relatively coarse absolute evaluations: they judge which of two similar positions is better, but do not attempt to assign slightly different values to them.
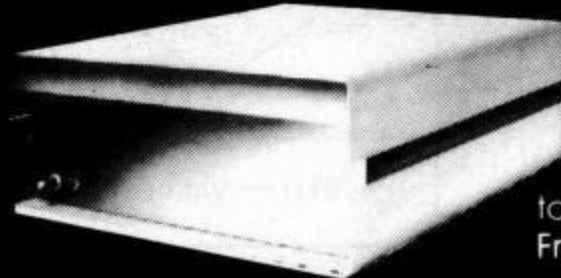
In chess programs, expansion, evaluation and backup are carried out simultaneously. One reason is that time can be saved by using backed-up values to demonstrate that some nodes need not be expanded at all. For example, the variation 1 BxN ch RxB 2 RxP ch gives White a great advantage; we say that 2 RxP ch refutes 1 ... RxB. Once one refutation is found, it is pointless to look for another: 2 R-B7 need not be considered if not considered already. What does this mean in terms of the minimax algorithm? Once node 3 has been assigned the value +1.4, we know that the value of the minimizing node 2 will not be any greater. Similarly, once node 7 has the value +2.4, we know that the value of the maximizing node 6 will not be any less. Therefore the minimax algorithm will not choose the value of node 6, and it
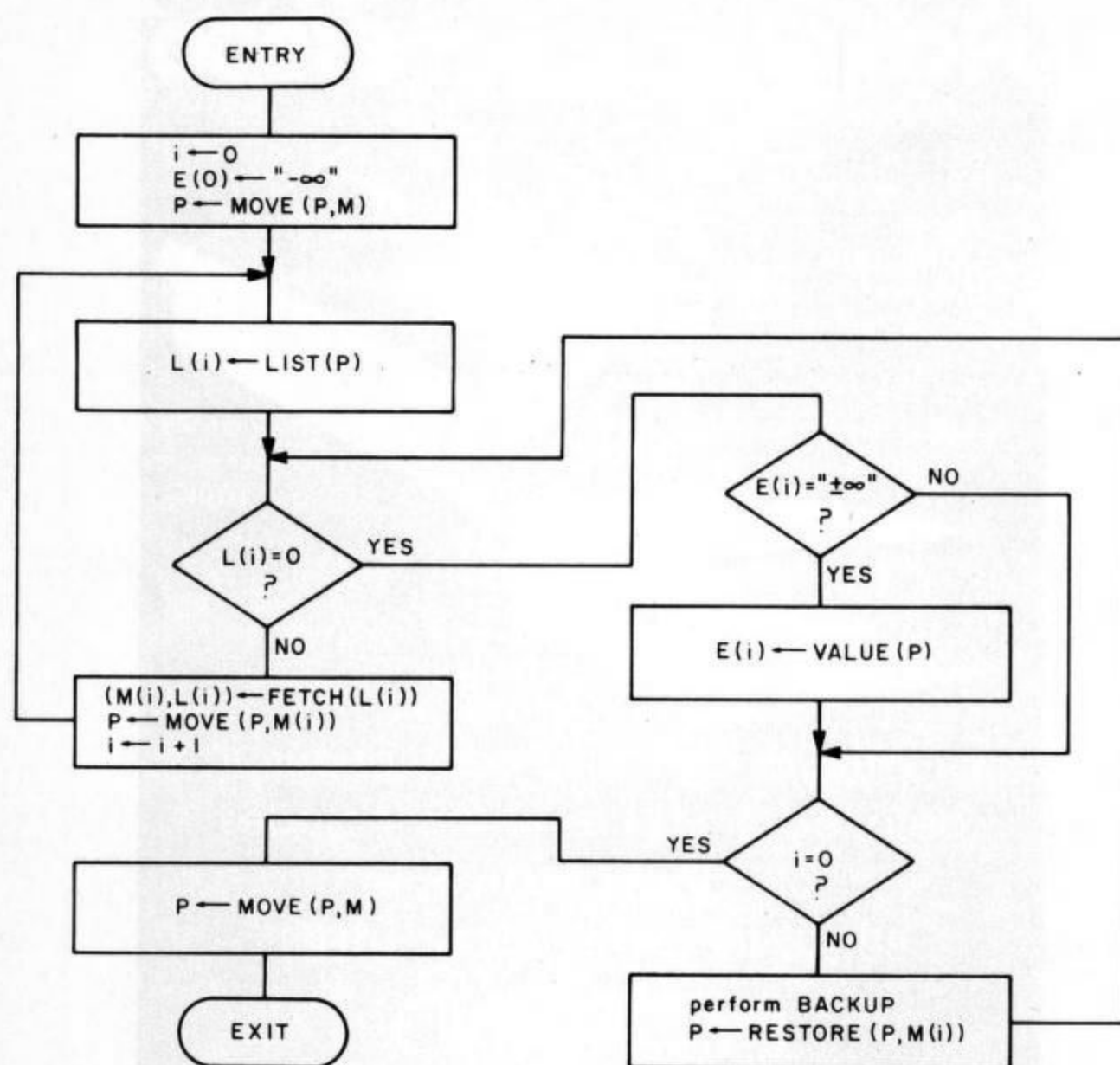
Figure 3: A routine to choose a move. A ply *table (so called because it is indexed by the ply number, i) is used to choose moves. (A ply is a move on the part of one player; two plies equal one chess move.) The entries in the ply table correspond to nodes in the game tree (see figure 2). Each entry contains three fields: L(i), a pointer to the list of moves selected at each node; M(i), the move currently being processed; and E(i), the evaluation. Most of the subroutines are written as functions in order to show which data areas they use and affect. Only those data areas that play a central role are indicated. ∞ refers to a number which is larger than any returned by subroutine VALUE. Its additive inverse, −∞, is used as the initial value of E(0).*

can be eliminated from the tree without expanding node 12. Although in this example only one branch can be eliminated in this way, it is an important method for limiting the size of the "bushier" trees generated by chess programs.

We have seen that there are three major aspects of chess reasoning that need to be analyzed to create a chess program: selection, termination and evaluation. The handling of these functions by existing programs is only a crude approximation to the human reasoning process. It has proven particularly difficult to limit the number of moves considered at each node without inadvertently eliminating the best move. Consequently, Chess 4.6 uses no selection or termination at a depth of less than six plies, and generates trees with hundreds of thousands of nodes. Even those programs that exercise some selection generate, in most cases, trees too large to store in programmable memory. Fortunately our procedure

can be reformulated so as to require only a small part of the tree to be retained in memory at any time.

The Depth-First Minimax Procedure

A tree can be traversed systematically by the following procedure:

> Start at the root — At each step, move to the leftmost unmarked son and mark it. If there is no unmarked son, move to the father. If there is no father, stop. (The terms *son, father* and *brother* are analogous to those in a family tree.)

The depth-first minimax procedure traverses the game tree in this way, simultaneously doing the expansion, evaluation and backup. Storage is required only for one path from root to leaf and for the brothers of the nodes on the path.

Figure 3 shows one way to organize the procedure. The processing is centered on the *ply table,* so named because it is indexed by the ply number i. The entries in the ply table correspond to nodes in the game tree. Each entry contains three fields: L(i), a pointer to the list of moves selected at the node; M(i), the move currently being processed; and E(i), the evaluation. The data area P contains the board position. As the tree is traversed, P is modified to show the position at the current node. At the start of the routine, the position is as it was presented to the opponent. The routine applies the move in location M to the position, chooses its move, stores it in M, and applies it to the position.

The subroutines named in figure 3 are discussed briefly here and in greater detail in the following sections. MOVE applies a move to the board representation P. It may also update auxiliary information describing the position. RESTORE simply reverses the changes made by MOVE. LIST generates the list of selected moves and places a pointer to the list in L(i). If the list is empty, L(i) is set equal to zero. FETCH moves the first move on the list to M(i) and advances the pointer L(i) to the next move. VALUE places the evaluation of a terminal position in E(i). BACKUP moves the evaluations E(i) up the table in accordance with the minimax rules.

Programs that generate a large tree generally use a depth-first search and have an overall structure similar to that shown in figure 3. The inflexibility of the depth-first search is a significant disadvantage, though. For example, suppose that shallow analysis of the first ply-1 move casts doubt on its
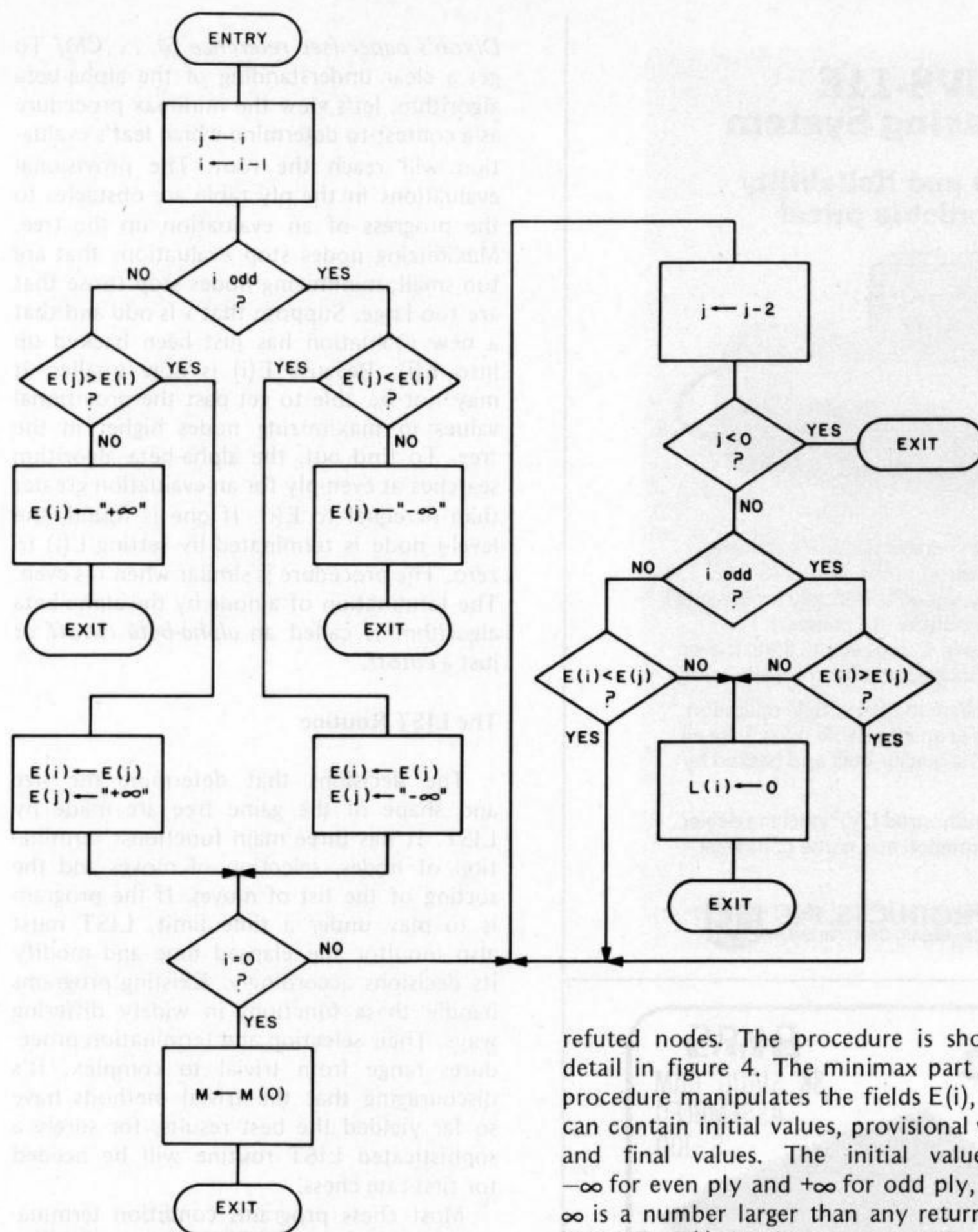
```
                        ENTRY

                         j ← i
                        i ← i-1

              NO        i odd        YES
                          ?

    YES   E(j)>E(i)              E(j)<E(i)   YES
              ?                     ?
              NO                    NO

        E(j)←"+∞"              E(j)←"-∞"

           EXIT                  EXIT

        E(i)←E(j)             E(i)←E(j)
        E(j)←"+∞"             E(j)←"-∞"

              NO   i=0
                    ?
                   YES
             M ← M(0)

               EXIT


                    j ← j-2

                    i < 0     YES      EXIT
                      ?
                      NO

          NO        i odd        YES
                      ?

    E(i)<E(j)   NO        NO   E(i)>E(j)
        ?                          ?
      YES                        YES
                  L(i) ← 0

                   EXIT
```

Figure 4: The BACKUP routine. The lefthand side of the flowchart depicts the minimax algorithm, a method which is guaranteed to choose the best move provided that the evaluations of the nodes in the game tree are accurate and that the best move at each node is included in the tree. The right side of the flowchart illustrates the alpha-beta algorithm, used to "prune" refuted nodes — that is, nodes which are known to represent inferior positions. Trimming the tree in this way reduces the amount of information that must be stored in memory and speeds up the evaluation process (see text).

value. Time might be saved by proceeding at once to the other moves and returning to the first move only if they seem no better. But in a depth-first search, the decision to terminate a variation cannot be changed on the basis of later information. Consequently, programs that generate small trees usually maintain the entire tree in programmable memory. Then it is possible to skip around in the tree, expanding those nodes that look most promising. Although such programs aren't structured like depth-first programs, they perform many of the same functions, and so the following discussion of the subroutines partially applies to them.

The BACKUP Routine

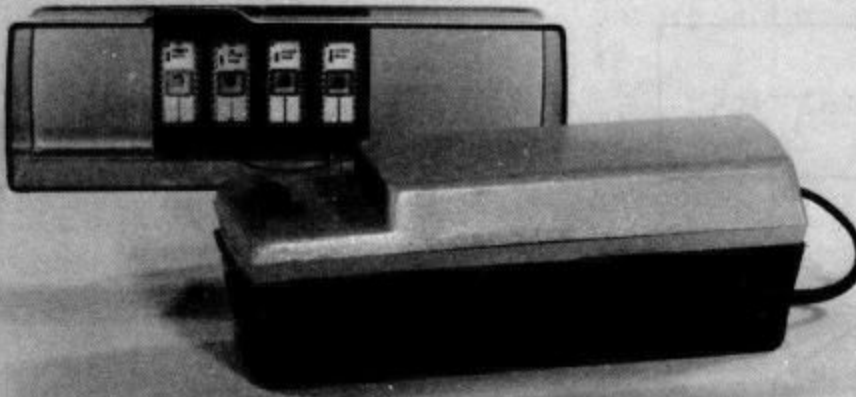The movement of values up the tree is controlled by BACKUP, which also prunes refuted nodes. The procedure is shown in detail in figure 4. The minimax part of the procedure manipulates the fields E(i), which can contain initial values, provisional values, and final values. The initial values are $-\infty$ for even ply and $+\infty$ for odd ply, where ∞ is a number larger than any returned by VALUE. E(i) is always set to the initial value when the table entry is not being used. The values produced by VALUE are final values. Whenever a final value E(i) appears in the ply table, BACKUP compares it with the value E(i−1). E(i) replaces E(i−1) if i is even and E(i−1) is greater than E(i) or if i is odd and E(i−1) is less than E(i). E(i−1) then contains a provisional value. A provisional value becomes final when the move list at its ply becomes empty. Whenever E(1) replaces E(0), M(0) is saved in M. As a result, M ultimately contains the first move in the list L(0) that produces a maximum final value in E(1).

The Alpha-Beta Algorithm

The elimination of refuted moves from the tree is accomplished by a procedure called the *alpha-beta* algorithm. *[The alpha-beta algorithm is discussed in Slagle and*

Dixon's paper (see reference 6). . . .CM] To get a clear understanding of the alpha-beta algorithm, let's view the minimax procedure as a contest to determine which leaf's evaluation will reach the root. The provisional evaluations in the ply table are obstacles to the progress of an evaluation up the tree. Maximizing nodes stop evaluations that are too small; minimizing nodes stop those that are too large. Suppose that i is odd and that a new evaluation has just been backed up into $E(i)$. Because $E(i)$ is now smaller, it may not be able to get past the provisional values in maximizing nodes higher in the tree. To find out, the alpha-beta algorithm searches at even ply for an evaluation greater than or equal to $E(i)$. If one is found, the level-i node is terminated by setting $L(i)$ to zero. The procedure is similar when i is even. The termination of a node by the alpha-beta algorithm is called an *alpha-beta cutoff* or just a *cutoff*.

The LIST Routine

The decisions that determine the size and shape of the game tree are made by LIST. It has three main functions: termination of nodes, selection of moves and the sorting of the list of moves. If the program is to play under a time limit, LIST must also monitor the elapsed time and modify its decisions accordingly. Existing programs handle these functions in widely differing ways. Their selection and termination procedures range from trivial to complex. It's discouraging that the trivial methods have so far yielded the best results, for surely a sophisticated LIST routine will be needed for first-rate chess.

Most chess programs condition termination primarily on depth and the availability of certain types of forcing moves. The simplest method would be to terminate always at some fixed depth. Then VALUE would have to give special handling to positions with an exchange in progress, lest material be reckoned incorrectly. Consequently, many programs use two depth limits. Beyond the first limit are selected only certain forcing moves, typically checks and captures. Termination occurs, of course, when there are none. At the second depth limit termination always occurs.

Other criteria for termination have been tried. The Ostrich program (developed on a Data General Supernova minicomputer at McGill University in Montreal, Canada) terminates variations in which material is sacrificed and not recovered within three plies. Several people have suggested that termination should occur only if the position can be accurately evaluated. The

Circle 294 on inquiry card.

Newell-Shaw-Simon program used this philosophy. When the entire tree is maintained in programmable memory, termination decisions as such need not be made at all. For example, the program COKO expands those nodes that promise the greatest yield of information, no leaf being permanently excluded from consideration.
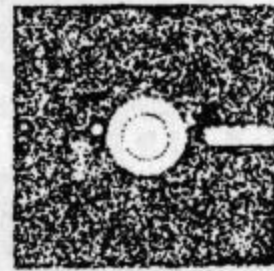
For selection and sorting, LIST might assign to each legal move a plausibility rating designed to indicate the probability that the move will prove best. Many programs don't explicitly assign a rating; nevertheless, it is convenient to imagine that their decisions are based on an implicit rating. Selection and sorting can then be done as follows: select all moves with ratings greater than some threshold. If too few moves are selected, add highest-rated moves to make up the minimum number. (The threshold and number of moves might depend on depth.) Sort the selected moves by rating.

For sorting, the requirements on the rating procedure are not stringent. It suffices that moves good enough to cause cutoffs often appear early in the list. Occasional inaccurate ratings will merely increase the processing time, not cause a blunder. The number of cutoffs can be markedly increased by simply assigning high ratings to a few easily defined categories of moves: captures, checks, moves by attacked pieces, etc. Another simple rating method is to assign a high rating to moves that have proven to be good in other parts of the tree. For example, the "killer" heuristic assigns to a refutation found at one node a high rating at its brother nodes. This heuristic works well in positions containing threats, because all moves that ignore the threat can be refuted by the same reply.

For selection, the plausibility rating must be more accurate. A best move markedly better than the second best move must only rarely receive a rating low enough to cause its rejection. Simple criteria that are adequate for sorting are bound to fail. The rating must be based on all of the move's important effects, which can in turn be determined only by elaborately tracing the relationships of the pieces. For this reason, programs that use selection generally maintain a tactical description of the position. In the program we are considering, it is the responsibility of the MOVE routine to keep such information current.

The VALUE Routine

The evaluation is usually computed as a sum of numerical scores, each representing one aspect of the position. Chess programmers tend to include only those aspects that

are easiest to program. Unfortunately they are not always the most important ones. The traditional chess literature is more explicit about evaluation than about selection and termination. The books in the bibliography are particularly helpful.

The evaluation of a position depends mainly on material, mobility and vulnerability. The calculation of material is straightforward, although experts disagree about the exact values of the men. Chess 4.6 uses the values: P=100, N=325, B=350, R=500 and Q=900. Like most programs, it adjusts the material score to encourage the exchange of pieces when ahead in material. The values of the pieces vary with the strategic character of the position: Rooks are better when the opponent has weak pawns, Knights are better in blocked positions, and so on. Such considerations are important, but I know of no program that takes them into account.

The assessment of mobility is more difficult. Counting the legal moves of each man is easy but inadequate. It is necessary also to take into account the exclusion of men from squares controlled by the opponent and the immobilization of men by defensive functions, such as the shielding or guarding of a man or important square. Detecting these factors is complicated and may involve tracing the relationships between several men.

Under vulnerability we have to consider unguarded pieces, the safety of the King, weaknesses in the pawn structure and pieces exposed to attack by less valuable men. Pawn weaknesses are easy to detect, and most programs take them into account. Measuring danger to the King is more complicated, but it is easy to detect some of the relevant features, such as disturbances of the King's pawn cover or the absence of friendly minor pieces nearby. Detecting unguarded and exposed pieces seems to be relatively simple, but oddly it is often neglected.

The MOVE Routine

Because of the rapid expansion of the game tree with depth, most of the processing time is spent in selecting and evaluating the terminal positions. It is therefore desirable for MOVE to maintain, along with the current position, information helpful to the LIST and VALUE routines. For example, it is more efficient for MOVE to keep track of changes in the material score than for VALUE to scan the board to do the same thing. Also, some programs maintain lists of the locations of each side's men to facilitate the generation of moves.

We have seen that sophisticated LIST

and VALUE routines would have to detect relationships between the men. Since each move changes only some of the relationships, it is more efficient to compute them in MOVE than to compute them all from scratch in LIST and VALUE. In general, the features needed for selection are the same as those needed for evaluation. For example, a backward pawn affects the evaluation and also suggests moves for both sides. The possessor of the pawn will try to advance it or protect it, while his opponent will try to prevent its advance and win it. Likewise any advantage suggests moves to maintain and exploit it; any disadvantage, moves to eliminate or mitigate it.

Levels of Skill

The United States Chess Federation rates its members at eight levels of skill based on performance in tournaments. In descending order they are Senior Master, Master, Expert and Classes A through E. From time to time computer programs have played in rated tournaments. Until recently their performance has been in the Class C or Class D range. Against this background the strong showing of Chess 4.5 startled everyone. At the conclusion of the Minnesota Open its rating had risen to Expert. It is still too early to assess its true strength, however. Although it is strong tactically, its grasp of strategy is well below the Expert level. The weak showing of Class A players against Chess 4.5 was caused largely by their unfortunate tendency to get into tactically complex positions, thereby playing into the computer's strength. The program may not be so successful once people learn how to play against it.

The sudden improvement in Chess 4.5 coincided with its transfer to a faster machine, enabling it to search two plies deeper in most positions. This supports the belief that chess skill depends mainly on the number of moves one can see ahead. It's difficult to give a precise equivalence between depth of search and level of skill, though. The following rule of thumb is, I think, close enough to the truth to give some idea of the design requirements for strong programs. Let a search depth of four plies correspond to Class C, and assume that each additional two plies yields an increase of one level of skill. Thus, play at the Expert level would require a 10 ply search.

The Exponential Explosion

The depth of search is limited by the increase in the size of the game tree with depth. Suppose that B moves are selected

Search Depth	Number of Terminal Nodes
2	59
3	929
4	1,800
5	27,900
6	54,000
7	837,000

Table 1: Tree size as a function of search depth (D), assuming exhaustive search and the maximum possible number of alpha-beta cutoffs. The branching factor (B) is assumed to be 30. The number of terminal nodes is $2B^{D/2}-1$ when D is even and $B^{(D+1)/2}+B^{(D-1)/2}-1$ when D is odd.

at each node. This number is called the *branching factor* or *fanout*. If D represents the depth of search, the tree has B^D leaves; the tree grows exponentially with depth. A typical position might have 30 legal moves, and if no selection is exercised, the tree will have 27,000 leaves at a depth of three plies. This is probably already too large a tree to examine with a microcomputer in a reasonable time. We have not, however, yet taken alpha-beta pruning into consideration.

The effectiveness of the alpha-beta algorithm depends on how well the move list is sorted. The greatest possible reduction in tree size is achieved when the best move is always first on the list. Table 1 shows the tree size under this condition, assuming a branching factor of 30. Clearly, exhaustive search beyond six plies is impossible for a small computer. To play

Branching Factor	Depth of Search	Tactical Skill
31.6	4	Class C
15.8	5	
10.0	6	Class B
7.2	7	
5.6	8	Class A
4.6	9	
4.0	10	Expert
3.5	11	
3.2	12	Master
2.9	13	
2.7	14	
2.5	15	

Table 2: Depth of search (D) and tactical skill as a function of the branching factor (B). It is assumed that the alpha-beta algorithm reduces the effective branching factor to $B^{2/3}$ and that 10,000 terminal nodes can be processed. These assumptions yield the formula $B=10^{6/D}$.

strong chess a microcomputer will have to use selection. The question is: how much?

To derive a relationship between the branching factor and the depth of search, we have to make some assumptions. Let us assume that we must limit the size of the tree to 10,000 leaves, and that the alpha-beta algorithm reduces the effective branching factor from B to $B^{2/3}$. Then table 2 gives the desired relationship. Although much guesswork went into this table, it seems safe to conclude that an Expert-level program must be very fast or very selective.

The TECH Program

How simple can a program be and still play reasonable chess? The TECH program was developed in order to answer that question. It would be a good model to follow if you want to have a running program in the shortest possible time. Despite its simplicity, or perhaps because of it, TECH placed higher in computer chess tournaments than some of the more complicated programs. It is good enough to defeat only inexperienced human players, but that is true of most programs. For the newcomer to chess programming, the design of a TECH type program would be a good way to gain experience.

TECH considers all moves to a fixed depth, beyond which it considers only captures. The evaluation of terminal positions is based only on material. Hence there is no need for a VALUE routine; the evaluation is computed on the run whenever captures occur. When the program has an advantage of two pawns or more, it reduces the value of its own pieces slightly so that exchanges are favored. TECH sorts moves for two purposes: to increase the frequency of alpha-beta cutoffs, and to bring factors other than material to bear on the choice of a move. At ply 2 and lower, captures are considered first and the killer heuristic is used. The positions at ply 1 are assigned a rating that includes such factors as the number of legal moves, the advancement of the center pawns, and the proximity of the pieces to the center, to the enemy King, and to passed pawns. The program expands the ply-1 nodes in descending order of the rating, which thus breaks ties in the backed-up evaluation.

Because TECH does very little processing at each node, it is able to generate a relatively large tree. Cutoffs are frequent; basing the evaluation only on material ensures that the alpha-beta comparison will often give an equal result. The ply 1 rating procedure could be made more elaborate

without slowing down the program noticeably. It would be interesting to see how much the program's play could be improved in this way.

New Directions

Chess programming is still a young field. There are many ideas that have never been tried or never been developed sufficiently to determine their value. Experimentation by computer enthusiasts could play a major role in developing the innovations that will be needed for a Master-level chess program. Some of the less successful chess programs use ideas worth further consideration. Papers describing some of these programs are listed in the bibliography. Additional ideas can be found by comparing the behavior of programs and human players.

Some Ideas for the Future

Chess games between computers are often dull because the programs don't follow any plan. They pursue general goals such as development and control of the center, but don't formulate goals specifically appropriate for the position at hand. Goals are represented in the evaluation and rating procedures. Setting a specific goal is accomplished by making changes in these procedures. For example, the general goal of center control might be implemented in part by a term in the evaluation polynomial for the number of pieces bearing on the center. A routine for setting specific goals might add a term for the number of pieces bearing on a center square that the routine had determined to be particularly important.

Here are some of the types of specific goals that occur frequently:

- Get control of a key square.
- Attack an area of the board where the opponent is weak.
- Free an immobile piece.
- Save an attacked man.
- Maneuver a particular piece to a square where it will have a strong influence.

It should be fairly easy to determine how to modify the evaluation and rating procedures in such a way as to set these goals. However, it might be difficult to devise a procedure for choosing the specific goals.

Most chess programs spend almost all of their time considering silly moves. There are two main types of silly moves: moves irrelevant to the important goals of the position, and sacrifices that gain nothing

that could be worth the cost. To safely reject irrelevant moves, the program must avoid overlooking important goals, lest it reject a vital move and blunder. Because of the difficulty of writing a comprehensive goal setting routine, it is not surprising that highly selective programs haven't performed well. Nevertheless, it is only a matter of time until enough chess knowledge is formalized to permit accurate selection. In the meantime, it might be possible to devise an algorithm that would reliably identify at least some of the silly moves.

Chess programs usually compare moves or positions by assigning numbers to them and then comparing the numbers. This method precludes certain possibilities of reliably rejecting moves. Suppose we have an algorithm that, given two similar positions, lists all of the important differences between them, together with limits on the effect each difference could have on the evaluation. It is then sometimes' possible to determine which position is better, even though it might not be possible to evaluate either position reliably. We saw an example of this in the analysis of figure 1, where 2 . . . K−N1 appeared to be clearly better than 2 . . . K−R1. The position-comparing algorithm could be used for selection and for a variant of a alpha-beta pruning. We meet with a familiar difficulty, however: the algorithm would have to incorporate comprehensive knowledge in order to avoid overlooking important differences.

To summarize, a program to play Master-level chess might contain algorithms to

- Find the important features of the position.
- Determine the relevant goals and rate their importance.
- Compare two similar positions to determine whether one is clearly better than the other.
- Select a list of reasonable moves in a given position.

Each algorithm would use the results of the previous ones in the list. The program would contain much chess knowledge, which would best be represented in a form both compact and easily alterable.

Prerequisites

How good a chess player do you have to be to tackle some of these problems? Most people need only a basic understanding of chess strategy and the ability to find simple combinations. Far more important than chess knowledge is the ability to teach what you know to a very dull, nonhuman pupil.

You will have to be able to state explicitly the reasons for the choices you make while analyzing a chess position. It's not as easy as it sounds. Above all, it's important to keep in mind that writing a chess program is a big project. A methodical approach, using structured programming and careful documentation, is absolutely essential.

Concluding Remarks

In this article I have tried to cover the basic ideas of chess programming and indicate some new directions for experimentation. I hope that many of you will be stimulated to get involved in this growing field of research. ∎

BIBLIOGRAPHY

1. Frey, Peter W, editor, *Chess Skill in Man and Machine,* Springer-Verlag, New York, 1977. Contains an introduction to computer chess, a description of Chess 4.5 and much thought provoking material.

2. Gillogly, J J, "The Technology Chess Program," *Artificial Intelligence,* volume 3, pages 145 thru 163, 1972. A simple program using exhaustive search.

3. Horowitz, I A, and Mott-Smith, Geoffrey, *Point Count Chess,* David McKay, New York, 1970. An informal system for evaluating positions numerically.

4. Kozdrowicki, Edward W, and Cooper, Dennis W, "COKO III: The Cooper-Koz Chess Program," *Communications of the ACM,* volume 16, pages 411 thru 427, July 1973. A program that uses a small game tree.

5. Newborn, Monroe, *Computer Chess,* Academic Press, New York, 1975. Contains brief descriptions of some of the older programs, many games and a detailed description of the author's program, Ostrich.

6. Newell, A, Shaw, J C, and Simon, H A, "Chess Playing Programs and the Problem of Complexity," *Computers and Thought* (Feigenbaum, E, and Feldman, J, editors), McGraw-Hill, New York, 1963, pages 39 thru 70. A program that uses a small game tree.

7. Shannon, Claude E, "Programming a Computer for Playing Chess," *Philosophical Magazine,* volume 41, pages 256 thru 275, March 1950. The start of it all, and still a good introduction to the subject.

8. Slagle, James R, and Dixon, John K, "Experiments with Some Programs that Search Game Trees," *Journal of the ACM,* volume 16, pages 189 thru 207, April 1969. Contains the results of some experiments with the alpha-beta algorithm and a proof of the formulas used for table 1 in this article.

9. Tarrasch, Siegbert, *The Game of Chess,* David McKay, New York, 1976. Contains a good summary of the principles of the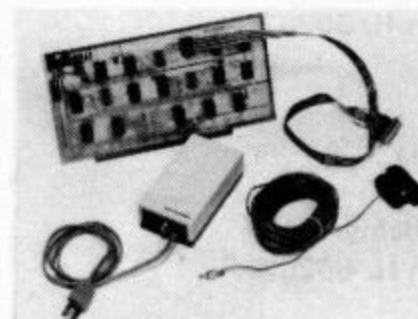 opening and a systematic survey of middle game tactics.