An introduction to computer chess

3

Peter W. Frey
Northwestern University

Chess has been an intriguing problem for individuals interested in machine intelligence for many years. Claude Shannon, the American mathematician, first proposed a plan for computer chess in 1949 [81]. The literature on mechanical chess-playing prior to this time reveals that the early automatons were merely facades which concealed a skilled human player [50]. Shannon believed that chess was an ideal problem for experimentation with machine intelligence since the game is clearly defined in terms of allowed operations (the legal moves) and in the ultimate goal (mate). At the same time, chess is neither so simple as to be trivial nor so complex as to be impossible. Shannon felt that the development of a credible chess program would demonstrate that "mechanized thinking" was feasible.

Shannon's prescription for machine-chess was not modeled on human chess play. As Charness describes in Chapter 2, humans master chess by capitalizing on their vast memory capacity and by organizing information in terms of meaningful piece configurations, plausible moves, and likely consequences. Several million years of evolution have provided the human with a tremendously complex visual pattern-recognition system and a large memory capacity. These permit man to assimilate newly acquired information in such a way that it may be subsequently recalled by using any one of many different retrieval cues. Human problem solving is critically dependent upon recognizing similarities among patterns and recalling information relevant to the specific situation at hand. The modern high-speed computer lacks both of these skills. Pattern-recognition skill in computers is still at a primitive level. Computer memories are large and fast but are organized simplistically in such a way that retrieval strategies based on structural or conceptual similarities are difficult to implement.

The computers that were available to Shannon in 1949 were much less powerful than our present machines. Shannon also had an additional handicap in that much less was known about human memory or human information processing at that time. For these reasons it is surprising that Shannon's proposal for machine chess is remarkably similar to the methods widely used today. Two of the strongest programs, KAISSA (the Institute for Control Science, Moscow) and CHESS 4.5 (Northwestern University, Evanston, Illinois), use sophisticated implementations of the basic strategy suggested by Shannon.

Machine representation of the chess board

A modern computer consists of a central processing unit, a memory unit, and multiple devices for the input (e.g., a card reader) and the output (e.g., a high-speed printer) of information. The central processor can perform simple operations such as addition, subtraction, and conditional branching on numbers that are transmitted to it from memory. After the designated calculations have been performed, the "results" are transmitted back to memory. Input and output operations, because of their slow speed, generally involve auxiliary devices that interface directly with the machine's memory. Most computer memories are quite large, ranging in size from several thousand computer "words" to hundreds of thousands of computer "words." A "word" is a binary string (e.g., 01101001011. . .) in which each bit[1] can take on only two values, generally represented as 0 and 1. A typical computer word consists of 8 or 12 bits in minicomputers and 48 or 64 bits in the high-speed giants. A 64 bit word is especially convenient for chess programming for a reason that most chess players can easily guess.

Shannon suggested that the chess board be represented by having 64 computer words each represent one of the squares of the chess board. Each word in memory can be thought of as a simple mailbox that will hold one piece (i.e., word) of information. Just like a mailbox, each word in memory has a specific address such that the central processor can gather information (e.g., several numbers) from specific memory addresses and —after performing several calculations with these numbers—store the result in one of these same addresses or into a new address according to its instructions.

Shannon suggested that each piece be designated by a number (+1 for a White pawn, +2 for a White knight, +3 for a White bishop, etc.; −1 for a Black pawn, −2 for a Black knight, etc.). Each of these numbers would be stored in the memory address that represented the square on which the piece resided. An empty square would be represented by having a zero stored at the address representing that square. More recent programs have followed this procedure, except a 10×12 board is used rather

[1] "Bit" is an abbreviation for "binary digit", either 0 or 1.

than an 8×8 board with a unique number, such as 99, stored at the address of all squares which are "off-the-board." In this way, the edges of the board can be easily detected. Using this system, the central processor can examine the contents of each memory address and determine if a piece exists on that square and if so, what its type and color is. If the central processor "examined" the address which represents KB4 (f4) and noted a −3 stored at that address, it would "know" that a Black bishop resided on KB4.

Legal moves from any given position can be determined quite easily by simply noting the mathematical relationship among the squares. Assume an assignment of memory addresses to the squares of the board as depicted in Figure 3.1. QR1 is assigned address 22, QN1 becomes address 23, QR2 becomes 32, QR3 becomes 42, QR8 becomes 92, and KR8 becomes 99. All addresses between 1 and 120 that are not depicted in Figure 3.1 would be assigned a value such as 99 because each represents a "square" that is off the board. Now if a knight is located on any square, the 8 potential squares to which it might move can be calculated by adding the following offsets to its present address: +8, +19, +21, +12, −8, −19, −21, and −12. For example if a White knight resided on KB3 (square 47) its potential move squares are 47+8 or 55, 47+19 or 66, 47+21 or 68, 47+12 or 59, etc. Potential king moves can be calculated in a similar manner by using the offsets of −1, +9, +10, +11, +1, −9, −10, and −11. After calculating the address of each potential move square, the machine must check the present contents of the new address to determine if the move is legal. If the address contains the number 99, the proposed move is illegal since the new square would be off the board. If the address contains a positive number, the move would be illegal since a positive number indicates that a white piece is already occupying the square. If the new address contains a negative number, the White piece can legally move to the new square[2] and capture the Black piece that occupies the

92	93	94	95	96	97	98	99
82	83	84	85	86	87	88	89
72	73	74	75	76	77	78	79
62	63	64	65	66	67	68	69
52	53	54	55	56	57	58	59
42	43	44	45	46	47	48	49
32	33	34	35	36	37	38	39
22	23	24	25	26	27	28	29

Figure 3.1 "Mailbox" representation of the chess board.

[2] If the White piece were a king, the machine would also have to determine if the square were attacked by an enemy piece.

square. Finally if the new address contained the number zero, it would represent an empty square to which the White piece could legally move.

Calculation of legal moves for a sliding piece such as a bishop, rook, or queen is slightly more complicated. Assume a White bishop is located on square XY (e.g., 35, where $X=3$ and $Y=5$). Examine address $X+1$, $Y+1$ (i.e., 46); if this address contains a positive number the bishop cannot move to this square; if this address contains a negative number, the White bishop can move to the square (capturing a Black piece) but can move no further along this diagonal; if the address contains a zero, the bishop can move to this square, and address $X+2$, $Y+2$ (i.e., 57) should be examined next. In this manner, each of the four potential directions for a bishop move can be checked until an edge (99) or another piece is encountered on each. After $X+1$, $Y+1$ (46), $X+2$, $Y+2$ (57), $X+3$, $Y+3$ (68), etc. are examined, the machine then needs to look at $X+1$, $Y-1$ (44), $X+2$, $Y-2$ (53), etc. and then $X-1$, $Y+1$ (26), $X-2$, $Y+2$ (17), etc. and finally $X-1$, $Y-1$ (24), $X-2$, $Y-2$ (13), etc. In this way, the machine can determine legal moves in all four potential directions for the bishop.

The legal moves for a rook can be determined in a similar fashion. Assume a rook on square ZW (e.g., 67, where $Z=6$ and $W=7$). Examine address Z, $W-1$ (66), Z, $W-2$ (65), Z, $W-3$ (64), etc.; then examine address Z, $W+1$ (68), Z, $W+2$ (69), etc.; next look at address $Z+10$, W (77), $Z+20$, W (87), etc.; finally examine address $Z-10$, W (57), $Z-20$, W (47), etc. In this way, the legal moves for a rook can be determined. For a queen, the legal moves for bishop and rook need to be considered in conjunction. Thus Shannon provided a relatively neat prescription for making pseudolegal moves in chess by machine. I use the term pseudolegal because I have failed to discuss the intricacies involved in determining the legality of moving a pinned piece or considering castling or an *en passant* capture. These additions make our mechanical move generator more complex but do not threaten the feasibility of Shannon's approach.

There is a more modern way to represent the chess board that was first suggested in the late sixties by a Russian computer-chess group [2] and discovered, apparently independently, by the Northwestern computer-chess group (see Chapter 4) and by Berliner [12] at Carnegie–Mellon. Assume, for example, that one is programming a large computer that has a 64-bit computer word. Now instead of assigning one computer word for each square, we will assign 1 bit of each 64 bit computer word to a square. Next we will represent the chess board in terms of 12 words. One word will represent all White pawns by setting a bit to 1 if a White pawn resides on that square and to zero otherwise. A second computer word will represent all Black pawns in the same manner (i.e., 1= piece present, 0= piece absent). A third word will represent all White knights, a fourth for all White bishops, and so forth for all the pieces such that six words are used for the White pieces and six words are used for the Black pieces. In

addition to pieces, we can use this procedure (called "bit maps" or "bit boards") to represent other information about the chessboard. Thus, we can have one word that represents all White pieces, one word for all Black pieces, one word for all squares attacked by White pieces, one word for all squares from which a bishop or queen could pin a particular piece against the king, etc. The power of this bit map procedure for expressing chess relationships is limited only by the programmer's skill in selecting important patterns.

The advantage of this approach becomes more apparent when one considers the instruction set for a modern computer. In addition to the common operations of addition and subtraction, the computer can perform Boolean instructions such as "logical-and" and "logical-or." For example, if we have two 8 bit words, say 01011100 and 11010011 and perform an "and" operation with the two words, the result would be 01010000. The new word would have a bit set (i.e., a 1) only at those positions where a 1 appeared in both of the original words. The "or" operation sets a bit in the new word at each position where a 1 appears in *either* of the two original words. For our example case, an "or" operation would produce 11011111. These operations can be applied to chess programming with highly desirable results.

Consider generating all legal moves for a White knight on KB3. First, the central processor "fetches" a bit map from memory that has a bit set for each position representing the squares to which a knight on KB3 could move. In this schema, central memory would contain 64 bit maps representing potential moves for a knight from each of the 64 squares on the board. Secondly, the central processor would fetch a bit map representing the positions of all White pieces presently on the board. The central processor would then negate (change 1's to 0's and 0's to 1's) the White piece map and then "and" this new map with the knight move map. The resulting bit map would represent all pseudolegal moves for the knight. Notice that move generation in this case involved two "fetch" instructions and two "Boolean" instructions. The Shannon "mailbox" procedure requires many more computer operations to determine the same result.

Consider a simple problem in chess. During the middle game, White decides to examine his attacking possibilities. He poses the question, "can I fork Black's queen and king with one of my knights on this next move?" Answering this question requires an assessment of several subgoals such as

1. Does there exist a square that is a knight's jump away from both the Black king and Black queen?
2. Is there a White knight positioned such that it can jump to the forking square in one move?
3. Is the forking square undefended by Black?

For a human, the answers to these three questions can be determined at a glance. For a machine, the answer is more difficult to determine.

If we start at the goal and work backward using Shannon's mailbox method, we first need to add the eight Knight offsets (+8, +19, +21, etc.) to the Black king square and then store these eight locations. Next we add the eight knight offsets to the Black queen square and compare the resultant of each addition to the eight stored values. If we find a match, then we must test the square's contents to determine if it is on the board (i.e., not 99) and if no White piece presently occupies the square (i.e., not a positive number between 1 and 6). If we find a match and the address does not contain a positive number, we have determined that a forking square exists.

Next we must determine if one of our White knights is located a knight's jump away from this square. This can be done by adding the eight knight offsets to the address of the forking square and then checking to see if any of these contain a +2 (a White knight). Finally, we need to determine if the forking square is attacked by a Black piece. This would be a tedious operation if we needed to check the legal moves of each Black piece to see if it can attack the forking square. Generally the attack squares for each piece are usually calculated and stored at the beginning of each move calculation. These data can then be used repetitively during the subsequent analyses of potential moves. In any event, the machine must execute at least 50 computer instructions in order to answer the question in an affirmative manner.

On the other hand, let us examine a bit map approach to this same question. First, we would "fetch" bit maps from memory for the potential knight moves from the Black king's square, from the Black queen's square, and from the White knight's present square. Next, the machine would fetch a bit map from memory representing the location of all white pieces. This map would be negated (a Boolean "not" operation, all 1's becomes 0's, all 0's become 1's) and the resultant map would be compared using an "and" operation with the resultant map produced by "and"ing the other three bit maps. If this final map were nonzero, a forking square exists. Finally, a bit map representing all squares attacked by Black would be "and"ed with the previous result. If this last operation produced a map containing all zeroes, one would know that the forking square was not being attacked by Black. Note that the bit map approach answers the same question, but uses only 5 "fetch" instructions, 1 "not" instruction, and 4 "and" instructions. A very significant increase in machine efficiency.

At present, many computer programs use bit maps which are oriented toward move generation and basic relationships among the pieces. There is no reason, however, why these bit maps cannot represent complex relationships such as "all squares accessible to a particular bishop in three moves given the constraints of the present board configuration." The advice-taking program developed by Zobrist and Carlson [101], which has been "tutored" by Charles Kalme, makes extensive use of bit maps as part of its pattern recognition scheme.

Static evaluation functions

Shannon [81] proposed that a move be selected by considering potential moves by White, replies by Black, counter-replies by White, etc. until relatively static terminal positions were reached. This examination of move sequences is commonly referred to as the look-ahead procedure and will be discussed at length shortly. Shannon proposed that each terminal position be evaluated in a mechanical way. He suggested a crude evaluation function which examined the material balance (9,5,3,3,1 for queen, rook, bishop, knight, and pawn, respectively), the relative mobility for each side (number of available legal moves), and pawn structure (penalties for doubled, backward, and isolated pawns). In his appendix he suggested additional factors for consideration such as control of the center, pawn structure adjacent to the king, passed pawns, centralized knights, knights or bishops in a hole, rooks on open files, semiopen files, or on the seventh rank, doubled rooks, attacks on squares adjacent to the enemy king, pins, and other factors. The idea was to weigh each of these factors according to its importance and then add all items together to determine the value of the terminal position. Once these values have been determined, the machine can select a move which would "lead toward" the most desirable terminal position.

Other factors for a general evaluation function have been suggested by Shannon's successors. Greenblatt and his associates at MIT [48] have included a "piece-ratio change" term that encourages piece exchanges when the machine is ahead in material and discourages piece exchanges when it is behind. The MIT group also included a king-safety term that encourages the king to remain on the back rank when queens are on the board. Church (see Chapter 6) has suggested that the king safety term should include the number of moves required before the king can castle. Turing [96] suggested a king-safety factor that "imagines" a queen on the king's square and subtracts points for the number of legal moves which the queen would have from that square. It is important to realize that king safety becomes less important as the number of pieces on the board diminishes and that in the end game the king-safety term becomes a hindrance. In the end game, for example, the Northwestern program (see Chapter 4) adds evaluation points if the opponent's king is close to the edge of the board.

In writing an evaluation function for a chess program, it is essential that efficient computer instructions be employed because this aspect of the program is used repetitively (i.e., ten thousand to 100 thousand times during each move selection). For this reason it is probably best not to include every conceivable evaluation term in the function since each new term means a small increment in the time which is required to evaluate each terminal position. A good evaluation function is one that assesses the critical aspects of the position in question and does this as efficiently as possible. For each new term which is added to the evaluation function one must ask if the chess information gained is worth the cost in computation

time that this additional assessment will require. The time requirement is important, of course, because computers play chess using the same tournament rules as humans, such as requiring 40 moves in the first two hours of play.

The structure of the evaluation function is dependent on the type of look-ahead procedure which is employed. In a program such as Berliner's [12], the emphasis is on evaluating a small number of terminal positions (e.g., 500) in a very thorough manner and therefore a highly complex evaluation function is necessary. In programs in which a very large number of terminal positions are examined, the evaluation function must be very fast and thus simplistic. For example, the "Technology" chess program by Gillogly at Carnegie–Mellon [46] examines only the material advantage (or disadvantage) for each terminal position and looks at as many as 500,000 terminal positions before selecting a move in tournament play. At present it is not clear which strategy will eventually lead to the best machine chess although our present best model, the human, clearly uses the former approach rather than the latter.

Our experience in computer chess over the past few years seems to indicate that future chess programs will probably benefit from evaluation functions that alter as the general chess environment changes. Such "conditional" evaluation functions will consider the type of opening, the stage of the game, the pawn structure, and the king defenses and then construct an evaluation function appropriate to the particular position. Computer programming techniques that use a hierarchical structure involving "discrimination nets" and "decision tables" are useful for this purpose. This modification would make the machine's evaluation procedure more similar to human analysis.

The look-ahead procedure

The most obvious way to examine future moves which might occur in a chess game is to generate all legal moves for the side on-the-move, all legal replies for his opponent, all legal counters, etc. until the possible sequences of moves and counter-moves seem to be sufficiently deep to make a terminal evaluation appropriate. As Shannon [81] pointed out in his classic paper, this "type-A strategy" has a serious drawback. The number of legal moves at each position (on the average, about 38) and the depth which seems necessary for reasonable play (6–10 plies), generate an enormous number of terminal positions. For example, a 2-ply (i.e., one move for each side) analysis of all legal moves, assuming 38 moves at each position, would generate 1444 terminal positions. A 4-ply analysis would generate 2,085,136 terminal positions. A 6-ply analysis would generate 3,010,936,-389 terminal positions! This difficulty with the exhaustive look-ahead procedure is referred to as the "exponential explosion" since the number of terminal positions increases exponentially with depth.

The look-ahead procedure is often discussed in terms of a "game tree" since a diagramatic depiction of the possible sequences of moves leads to a structure that branches out much like a tree. The original position is like the trunk of a tree which leads to several moves (the major limbs) which in turn lead to counter-moves (the large branches) which lead to counter-counter-moves (the small branches), etc. The point at which one branch subdivides into many smaller branches is called a "node" of the tree and represents an intermediate board position in the game tree.

Shannon suggested that the exponential-explosion problem be solved by examining only a small subset of the potential legal moves at each node. He labeled this approach as a "type-B strategy." If one makes a 6-ply analysis examining only 5 continuations at each node instead of 38, the number of terminal positions would be 5^6 rather than 38^6 or "only" 15,625 positions. Since there are never more than 5 reasonable moves in any given position [31], this suggestion appears to have great merit. One merely needs to select 5 "plausible" moves at each node and examine these continuations while ignoring all others. To implement this approach, the machine must be able to determine which of the potential moves at each node are most reasonable. The machine subroutine which is designed for this purpose is called the "plausible-move generator."

Human chess players have a great facility for selecting reasonable moves and reasonable replies at each node. The ability of skilled players to play a respectable game of "speed" chess (e.g., 5 seconds per move) demands this. This human ability seems to be perceptual (see Chapter 2) since it is unlikely that the consequences of each move can be examined thoroughly in 5 seconds. When de Groot [31] studied the protocols of skilled players, he reported that the experts failed to select the best move because it was not even considered in their verbal analysis of the position. Apparently, their "plausible-move generators" were not as discerning as the ones used by the grandmasters. The success of Shannon's "type B" strategy depends upon our ability to develop a plausible move generator for the computer. History has shown that this is a very difficult problem. Most computer chess programs contain fatal blind spots in this regard and this severely limits the quality of their play.

Let us assume for the moment, however, that it is possible to develop a reasonable static evaluation function and a reasonable plausible-move generator for our machine. Given these two basic functions, how can we use them to decide which move to select when confronted with a specific game position? How do we construct a game tree and then use it to select our move? There are actually several procedures for doing this.

The method initially suggested by Shannon involves the "minimax" procedure proposed by von Neuman and Morgenstern [97] in their classic book. A move is selected by "looking ahead" in the game tree from the base position to some predetermined depth. At each node the machine assumes that the player who has the move will select that alternative that is "best" for him. "Best" would be defined in this case by the static evalua-

tion function. If White has the move from the base position and if the static evaluation function is represented such that large numbers reflect an advantage for White (and small numbers favor Black), then the machine will maximize (choose the pathway leading to the largest evaluation) at the odd nodes (i.e., White's turn to move) and will minimize (choose the pathway leading to the smallest evaluation) at the even nodes (i.e., Black's turn to move). This process of alternately choosing the maximum value and minimum value leads to the descriptive name of "minimaxing."

The machine's procedure for analyzing a game tree can best be explained by using an example. Let us assume a starting board position with all pieces and pawns in their original position. White, as always, has the first move. Assume that the machine analyzes to a depth of 4 half-moves (4 plies) and uses its plausible move generator to develop the game tree depicted in Figure 3.2. The squares represent a position in which it is White's turn to move and the circles represent a position in which Black has the move. The pathways between successive positions each represent a potential move which seems plausible to the machine. The nodes of the game tree can be generated in many different orders, but generally a "depth-first" procedure is used. That is, the first branch is explored to its maximum depth before the machine starts on a second branch. An alternate procedure would be a "breadth-first" search in which all the nodes at one level are explored prior to examining nodes at a deeper level. The nodes in Figure 3.2 are numbered in the order in which they would be generated if a "depth-first" search had been employed. The machine would first examine (1) P–K4, P–K4; (2) N–KB3, N–QB3, evaluate the terminal board configuration and then store this value. Next, the machine would examine (1) P–K4, P–K4; (2) N–KB3, P–Q3 apply the static evaluation function, and store the result. Next, the computer would explore (1)

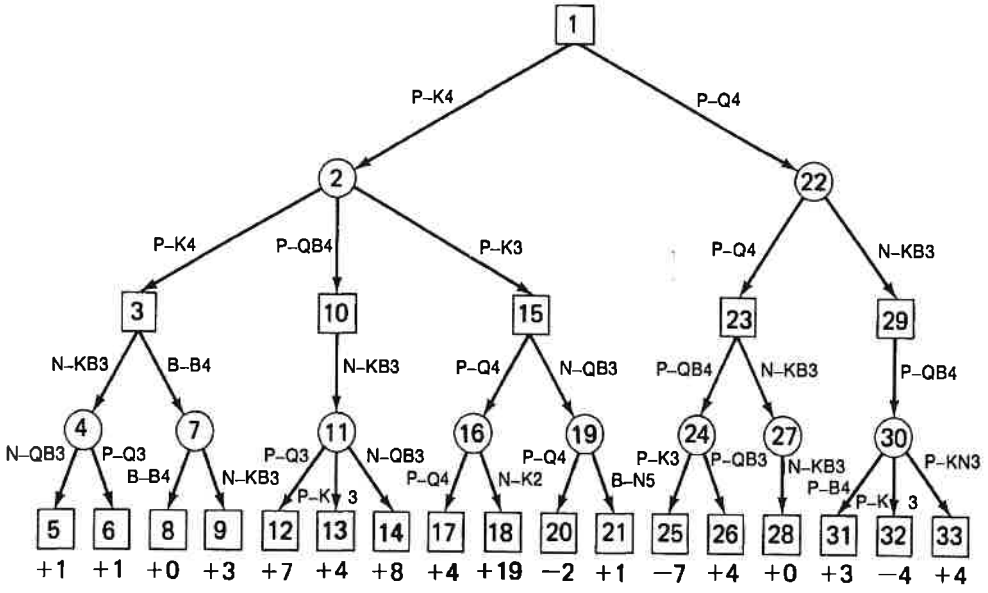

Figure 3.2 Sample game tree for the opening position in chess.

P–K4, P–K4; (2) B–B4, B–B4, evaluate, and store the result. This procedure would continue until the entire game tree in Figure 3.2 had been generated and an evaluation calculated for each of the 17 terminal positions.

After each position is evaluated, the value is returned to the level above (the immediate parent) and it becomes the "best value so far" for that node. Each additional descendent of this same parent node is examined in the same fashion, and when a value is obtained for each terminal position, the value is compared to the best value found so far. If the value associated with the most recent evaluation is better for the side to move than the best value so far, the new move is stored in memory and its value becomes the new "best value so far." After all of the descendents of each node have been examined, the best value so far and the move associated with that value are "backed-up" to the level above. This process is continued until a value and a move for the current position are determined.

For expository purposes, let us assume a simple evaluation function for the opening which is computed as follows. Let LW and LB equal the number of legal moves for White and Black respectively, let CW and CB equal the minimum number of moves required for White and Black to castle respectively, and let SW and SB equal the number of central squares (K4, K5, Q4, Q5) attacked by White and Black respectively. Our evaluation function is then defined as: (LW–LB)+[3 × (CB–CW)]+ [3 × (SW–SB)]. The larger the number, the better the position from White's point of view. The numbers which appear underneath each terminal position in Figure 3.2 represent the result of applying this evaluation function to each of these positions. The intermediate calculations are presented in Table 3.1. Note the laborious calculations necessary for even a very simple evaluation function such as this one.

To select a move, the computer would apply the minimax procedure to this game tree. The two terminal positions "sprouting" from node 4 would be examined and the minimum evaluation (it is Black's turn to move) would be "backed-up" to node 4 (+1). The two terminals "sprouting" from node 7 would be examined next and the minimum evaluation would be "backed-up" to node 7 (+0). In a similar manner, "backed-up" values for nodes 11, 16, 19, 24, 27, and 30 of +4, +4, −2, −7, +0, and −4 respectively would be selected. Next, the machine would maximize at node 3 (it is White's move), "backing-up" that value from its descendents, nodes 4 and 7, which is largest (+1). Maximizing at nodes 10, 15, 23, and 29 would produce "backed-up" values of +4, +4, +0, and −4 respectively. Moving toward the base node again, "backed-up" values for nodes 2 and 22 would be selected by minimizing since it would be Black's turn to move. Thus node 2 would have a value of +1 and node 22 would have a value of −4. Finally, the machine would select a move at the base node by maximizing between nodes 2 and 22 and this would lead to the selection of node 2 (i.e., P–K4). Although this minimax procedure can lead to some confusion when it is applied by humans, it is easily

TABLE 3.1 Preliminary calculations used by a primitive evaluation function

Terminal position	Legal moves		Moves to castle		Center squares attacked	
	White	Black	White	Black	White	Black
5	27	29	2	3	3	3
6	27	32	2	3	3	2
8	33	33	2	2	2	2
9	33	27	2	2	2	3
12	27	29	2	4	3	2
13	27	29	2	3	3	2
14	27	25	2	4	3	3
17	38	34	3	3	3	3
18	38	22	3	3	3	2
20	33	35	3	3	3	3
21	31	33	3	2	3	1
25	30	34	4	3	3	3
26	30	29	4	4	3	2
28	29	29	3	3	4	4
31	30	24	4	3	3	3
32	30	28	4	2	3	3
33	30	23	4	2	3	2

followed by a machine. The choice rule at each node is precise and therefore the machine only needs to remember whether to minimize or maximize at odd-ply levels and vice-versa at even ply levels. The operation of selecting the largest or smallest value for the potential descendents at each node is trivial for the machine. Thus the minimax strategy provides a workable procedure for the machine to use in selecting a move.

Backward pruning

Although Shannon apparently never implemented his plan for playing chess by computer, Newell, Shaw, and Simon [74] did and soon discovered that much of the laborious tree-searching involved in the minimax procedure can be avoided. The reason can be easily demonstrated by examining Figure 3.2. Assume that the computer generates nodes sequentially in the depth-first order depicted in Figure 3.2 and evaluates each terminal node as soon as it has been generated. After all of the descendents of node 4 have been generated and the resulting terminal positions have been statically evaluated, the machine can "back-up" the appropriate evaluation to node 4. In the present case, this would be a value of $+1$. Now at node 3, it is White's turn to move so that the larger of the values backed-up at nodes 4 and 7 will be selected. For this reason, the machine can determine in advance that node 3 will eventually have a value which is $+1$ or larger. The value would never be less than $+1$ since White could always select the pathway leading to node 4. Given this information, the computer

can deduce that it is a waste of time to generate and evaluate node 9. The reason for this is that the evaluation value for node 8 is less than +1 and black will choose the descendent of node 7 with the smallest value. Therefore node 7 will have a final backed-up value which is 0 or less independent of the value eventually computed for node 9. Since 0 is less than +1, White will choose the pathway to node 4 and not the pathway to node 7. Therefore it is senseless to examine any further descendents of node 7 once a value has been found which is equal to or less than +1. This reasoning can be applied throughout the tree. When the branching factor (i.e., the average number of descendents for each node) is high, this procedure will lead to a substantial reduction in the number of nodes in the game tree which need to be generated and evaluated. In the example presented in Figure 3.2, this "pruning" procedure would eliminate nodes 9, 19, 20, 21, 29, 30, 31, 32, and 33 from the minimax search. When the game tree involves more descendents from each node (say 25 instead of only about 2 in our example) and the tree extends to 5 or 6 plies in depth, this "backward-pruning" method can lead to a tremendous saving in search time.

The technical name for this procedure is the α–β algorithm because the value for the move which is currently best for White during the tree search is labeled α while the value for Black's best move so far is called β. This α–β algorithm has made the minimax procedure feasible in many problem-solving tasks where it would otherwise be totally unrealistic because of time constraints. The beauty of this modification is that it always produces the same result as the full minimax procedure [59]. One factor that is very important to the efficiency of an α–β minimax tree search is the order in which the moves are generated and tested. If White's best moves and Black's best replies are considered first, a great many weaker alternatives need not be examined. For example, if White considers QxP and Black can reply BxQ, it is wasteful for Black to generate and evaluate 34 other moves which do not involve the capture of White's queen. If the queen capture is examined first, White "knows" that QxP is a "losing" move and therefore should not be attempted. Moves such as BxQ are called "refutation moves" since they refute the opponent's previous move(s). By examining refutation moves first, the machine can reduce the number of nodes in the tree by a substantial amount. The efficiency produced by proper ordering of the moves is enormous since each pathway eliminated early in the tree also eliminates all the potential descendents that would "sprout" from that pathway.

Present computer-chess programs place great emphasis on ordering the moves at each node such that strong moves will be examined first. To do this, the program uses "heuristics" (i.e., general rules-of-thumb) that provide useful information concerning what kinds of moves in a particular situation will lead to rapid pruning. One important heuristic is that of examining all capturing moves first. Captures are common pruning moves since they refute any previous move by the opponent which has left a

piece *en prise*. Captures also reduce the size of the game tree by eliminating pieces from the board. This usually has the effect of decreasing the number of legal moves that can be generated from each node. A decrease of a few moves in a tree growing exponentially has a surprisingly large effect.

Another commonly used heuristic is to have the program store previous refutation moves in some other part of the game tree and to try these moves again in each new position of the tree. In some situations Black, for example, may have a move that is strong against a variety of moves for White. In this type of situation, poor moves by White can be most efficiently "pruned" from the game tree by always examining first this one move for black each time it is Black's turn to move in the tree search. Because this procedure involves remembering previous refutation or "killer" moves, it is often referred to as the "killer heuristic."

It is interesting to note that this look-ahead minimax procedure is essentially a blind search. It is only after the sequence of moves has been generated and the terminal position evaluated that the machine can determine if this prior activity was worthwhile. This process bears some resemblance to Darwinian evolution. Reproduction in biological organisms produces new variations in a somewhat haphazard fashion. These variations are acted upon by natural selection (i.e., nature's evaluation function). Selection among the new variations occurs by a retrospective process. Only those variations that survive the rigors of the natural environment persist long enough to reproduce. The minimax procedure works in a similar retrospective manner, selecting those variations that "survive" the scrutiny of the evaluation function. The success of this "blind" search process in biological evolution indicates the feasibility of the procedure. However, biological evolution has taken millions of years while the selection of chess moves must occur in 3 minutes or less. The weakness of this process, therefore, lies in its inefficiency rather than in its ultimate feasibility.

In a recent article in *Artificial Intelligence*, a Russian group [3] has suggested a technique for making the tree search more efficient. Their idea is based upon the observation that the machine's calculations in the game tree are highly repetitive. For example, there might be a position on the board in which (1) NxP, PxN; (2) BxP, BxB; (3) RxB, RxR; (4) QxR, QxQ or some similar exchange is possible. In order for White to determine that these exchanges are not worth initiating, the machine must examine all captures on this square in all possible orders. Unfortunately, at each new node in the tree where it is White's turn to move, this same potential set of captures may exist and thus need to be refuted each time by generating all possible capture sequences. Most programs would refute NxP several thousand times in this way at various locations in the game tree. Obviously this is highly wasteful since most of the positions in the game tree are highly similar and the outcome of this exchange will be the same in almost all of these cases.

The Russians have suggested a procedure which they call "the method of analogies." The strategy is to examine refutation moves in detail and

determine all factors which are necessary for this refutation to remain true. This is really a case of theorem-proving since it attempts to establish a set of postulates that must be true for the refutation to remain effective. The actual proofs are quite complicated and involve such information as the number and types of pieces bearing upon a particular square, the absence of any pieces which might block the attack pathway of a sliding piece, and the pin status (relative and absolute) of pieces involved in the exchange. If such a plan is implemented, the move NxP could be rejected after checking a few board features rather than having to generate and test all sequences of captures each time. This strategy could produce a very significant improvement in middle-game play since it would produce rapid pruning in the game tree and would thereby eliminate many pathways from further search.

There are tree-searching methods that are different from the α–β minimax procedure. General discussion of these are provided by Nilsson [77] and by Slagle [88]. In Chapter 7 of this book, Harris describes one of the more powerful alternatives which is similar to progressive-deepening as described by de Groot [31].

Quiescence

In using the look-ahead game tree approach, it is often difficult to determine when a node should be considered terminal and ready for evaluation. A major source of error is the premature evaluation of a node giving an erroneous impression of the advantage or disadvantage of that position. The problem for the machine is to avoid applying the "static" evaluation function to a position that is not static or "quiescent." If an evaluation is made on an active node, such as one representing a position in the middle of a piece exchange, the calculated value may be grossly in error. It would be a blunder of major proportions to evaluate a position after RxQ if the next move for the opponent was BxQ. The notion of quiescence applies to material evaluations and also in a more subtle way to positional evaluations. For example, a knight or bishop may have to move to a relatively unfavorable square in order to eventually reach its "post" in a hole or in some other desirable position. If an evaluation is made while the knight or bishop is "en route," the machine may conclude that the necessary sequence of moves is not worth examining any further. Unfortunately, it has been very difficult to program a machine to "understand" whether a position is active or static.

Many programs try to compensate for this lack of understanding by always examining capture sequences to their end until no additional captures exist. Unfortunately even this approach is oblivious to pins or other positional moves that guarantee a gain in material for the opponent. Checks on the king can also lead to serious problems. In many positions a series of checking moves can proceed for 10, 12, or 14 plies without really accomplishing anything. Unfortunately, the machine is weak at discriminat-

ing between "aggressive" checks and "useless" checks. In order to prevent an exponential explosion of the game tree, most programs place a limit upon the number of checks which can be pursued without an immediate gain of material or mate. In most cases, this limit is about 2 extra plies. Thus, if a tree search were structured for a 5-ply search in general, checks might be pursued to 7 plies and captures out to 12 or 15 plies, whatever was necessary to exhaust the different sequences of captures. These procedures clearly improve the quality of play. It would be preferable, however, for the machine to be able to assess the quiescence of a position directly and terminate its search on this criterion rather than at some fixed depth which has no rationale other than the time limit for choosing a move. Harris amplifies this point in Chapter 7.

It might seem reasonable to solve this problem by searching the game tree to great depth and thereby looking far enough ahead to compensate for a less-than-perfect evaluation function or for a poor notion of quiescence. It has been easier to increase the search depth by acquiring faster machines and developing more efficient tree-search procedures than it has been to improve the machine's evaluation function or its notion of quiescence. Depth of search, in and of itself, produces a considerable increase in the machine's degree of "understanding." Many chess concepts are clearly depth dependent. With a fixed 2-ply search supplemented by a check and capture search, the machine will completely miss a simple forking maneuver. With a 4-ply search, however, the computer appears to develop an "understanding" of the fork and will select moves to avoid this simple trap.

Greater depth can be achieved in the middle game if a very primitive evaluation function (such as an analysis of material only) is used. Additional depth might also be obtained by developing a special central processing unit with an instruction set including chess operations. Even with the blinding speed available with modern hardware, this approach would still have an optimistic limit of around 8 ply for a full-width search during the middle game. This does not augur well for brute force solutions to machine chess since most good human players find it necessary to calculate to 12 or 14 plies at least once during the middle game. Time will tell whether hardware innovations will permit future machines to substitute brute strength for finesse. The mathematics of the situation, however, are highly unfavorable to the brute strength approach.

Plausible-move generators

For those who have concluded that a brute-force, full-width search (i.e., Shannon type-A strategy) will never be able to look ahead far enough to play acceptable chess, there is a second method, that of selecting only a few of the progeny at each node for further examination (i.e., Shannon type-B strategy). This can be dangerous since a move excluded from the game tree can never be made across the board. If the machine were

successful, however, in developing a good plausible move generator such that only 3 moves, on the average, sprouted from each node, the depth of search could be increased dramatically. A 12-ply search with 3 sprouts per node produces 531,441 terminal positions. With 30 sprouts per node, the number of terminal nodes becomes 810,000 after only 4 ply. This type of calculation has convinced many people, including Shannon, that emphasis should be placed on developing an accurate plausible-move generator.

The Greenblatt program [48] attempts to select the most promising moves at each node. The search width for tournament play is usually set at 15, 15, 9, 9, and 7 moves for the first, second, third, fourth, and fifth ply, respectively. Moves are selected that cause pieces to attack squares in the center and near the enemy king. Moves that block opponent's pieces or unblock friendly pieces are also considered. Priority is given to attacks on weak pawns, pinned pieces, pieces defending other pieces, etc. All checks are investigated within the first 5 ply. All captures are examined within the first 2 ply. Priority is given to considering a few moves of several different pieces rather than many moves of only one or two pieces. In all, Greenblatt uses about 50 heuristics for computing the plausibility of the different descendents at each node.

Berliner [12] also places great emphasis upon reducing the branching factor at each node. He feels that a successful move generator will average only 1.5 descendents from each node in the tree. Because of this belief, he is willing to accept very long computation times for determining plausible moves and thereby restrict his game tree to 500 nodes or less. This approach more nearly approximates human chess play than do the brute-force procedures. The success of Berliner's approach will depend very heavily upon his skill in transmitting a tremendous amount of chess knowledge to the machine.

The difficulty of building an "intelligent" plausible-move generator can best be illustrated by citing several examples. Consider the board position depicted in Figure 3.3, taken from a game between Kirdetzoff (White)

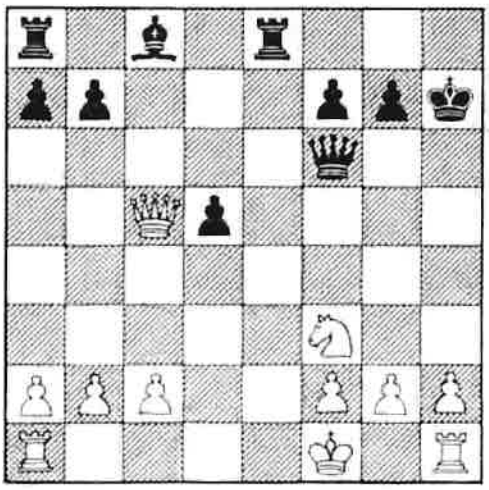

Figure 3.3 Position after 16 moves between Kirdetzoff (White) and Kahn (Black), Copenhagen, 1918.

and Kahn (Black) played in Copenhagen in 1918. Black, with the move, has several plausible lines of play. Developing the bishop seems reasonable since it would also double his rooks. The queen pawn is *en prise*, so moves that defend it are plausible. White's QN pawn can be captured with resultant pressure on White's queen rook. All of these moves are reasonable. The winning move for Black, however, was QxN! Play continued (17) PxQ, B–R6+; (18) K–N1, R–K3; (19) Q–B7, QR–K1; (20) R–KB1, R–K8; (21) White resigns. The move Q–B7 for White was necessary to prevent R–N3 mate. This is a nice example of Morphy's sacrifice, clearly demonstrating that a positional advantage can be worth much more than a material advantage. For a machine, however, losing a queen for a knight is an "unthinkable" exchange and would probably not survive a superficial plausibility analysis. If QxN is not even entered into the game tree, it can never be selected as the across-the-board move.

A second example of this same idea is taken from a recent game between Adorjan (White) and Tompa (Black) in the 1974 Hungarian championship. After Black's 27th move (N–R4), the position depicted in Figure 3.4 was reached. In this active position, White has a number of seemingly aggressive moves. Play continued (25) QxN!, PxQ; (26) R–N3+, B–N2; (27) N–B5, Q–B1; (28) NxB, PxP; (29) N–K6+, K–R1; (30) BxP+, P–B3; (31) RxP!, Black resigns. Again we observe a queen sacrifice that gains a positional advantage at a major cost in material.

In both of these examples, a sacrifice in material was followed by a series of moves ultimately leading to victory. However, these subsequent moves were not an uninterrupted sequence of checks or captures and therefore the value of the queen sacrifice could not be easily analyzed with a deep but narrow tree search. Most programs place heavy emphasis on material factors simply because this strategy is necessary to maintain reasonable play. If the machine were to give positional factors equal weight, it would routinely throw away its material in unsound sacrifices. In the great majority of cases, a loss of a piece is tantamount to losing the game.

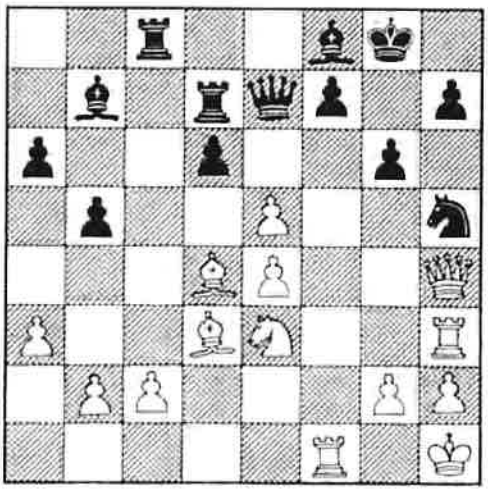

Figure 3.4 Position after Black's 27th move, Adorjan (White) vs. Tompa (Black), Hungarian Championship, 1974.

The use of a reasonable plausibility function seems to automatically exclude the brilliant sacrificial moves which add flair and excitement to chess.

In addition to material sacrifices, there are other excellent moves often missed by a plausible move generator. Figure 3.5 depicts a tactical position with White to move. Black has a pawn advantage and White's queen is presently *en prise*. White can win a pawn by (1) QxQ+, RxQ; (2) BxP or can simply hold his position by moving the queen to defend the bishop such as Q–N5 or Q–N8+. Each of these moves would be highly plausible for a machine since they win or "save" material. The correct move, however, is one which leaves both the queen and bishop *en prise*, namely B–Q6. If Black responds with QxQ, then R–B8 is mate. If Black responds with RxB, then Q–N8+ leads to mate. If Black captures the bishop with his knight, then QxQ+ wins. Only a dynamic analysis indicates that B–Q6 is correct and thus a plausible move generator that selects moves on a static basis would most certainly exclude B–Q6.

A final example is depicted in Figure 3.6. In this position, Black has a material advantage and it is White to move. There are a large number of potentially plausible moves but the correct move is one that seems, at least superficially, to be unsound. The proper move is Q–R6, virtually

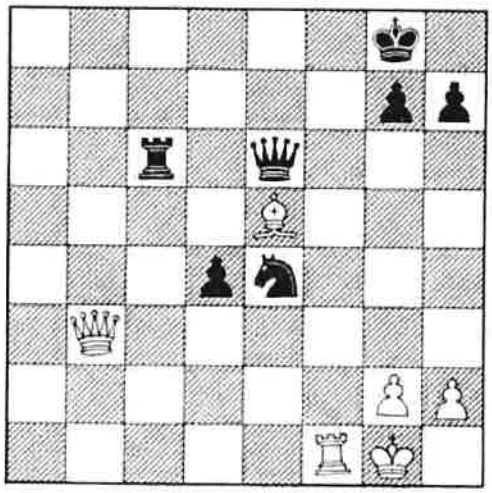

Figure 3.5 The best move for White seems plausible only after considerable analysis.

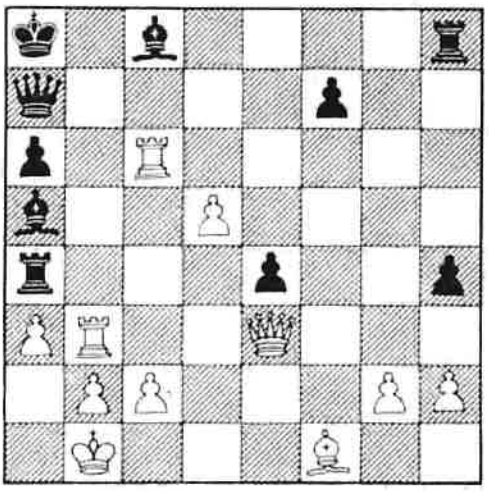

Figure 3.6 White can launch a winning attack with a move that, on the surface, appears to be unsound.

destroying Black's game. Few, if any, computer programs would consider this move plausible on the basis of a static analysis of the position. Only dynamic analysis would show that the rook is "riveted" to the back rank and therefore cannot capture the queen.

One of the characteristics of chess as a research environment for analyzing heuristic search is that the most promising continuation is often not discernible from a superficial analysis of the position. This is clearly evident in the examples just cited. Techniques based on a hill-climbing strategy, such as forward pruning, face serious problems in this type of search environment. Berliner has recognized this inadequacy of traditional forward-pruning search procedures and has developed a coping strategy he calls the "causality facility" [12]. This technique permits his program to discover an important problem deep in the search tree and to pass this information down to lower levels of the tree. The new insight is then used to restructure the search at shallow levels. This approach more nearly approximates a human search process and provides a viable alternative to the full-width search strategy.

Selective searching, deep in the game tree, can lead to additional problems whether it results from forward pruning or is part of a quiescence analysis following a limited-depth full-width search. If the tree-search is not tightly structured and firmly controlled, the machine can get lost at some deep level of the tree and spend hours trying to select the proper variation in a chess environment in which each pathway leads to an exploding number of potential continuations. This problem plagued the TECH II program from MIT at the 1974 ACM tournament in San Diego and has been discussed in some detail by the authors of COKO [61].

Full-width searching

The difficult problems associated with forward pruning have encouraged several programmers to employ a full-width search strategy. Both the Northwestern group and the Russian group [2] develop game trees in which all legal continuations are examined from each node out to a fixed depth. This full-width search is supplemented at the terminal nodes by a narrow search of potential exchange sequences and a few checking moves. The obvious weakness of this strategy is that deep searches are not possible. Berliner [12] has discussed the blunders that inevitably accompany shallow searching. Two of the most common problems are the absence of long-range planning and a special phenomenon Berliner has labeled the "horizon effect."

These problems can be demonstrated by reference to Figure 3.7, which is a slight variation of Figure 1.10 of Berliner's thesis [12]. In this position, with White to move, there is a clear and obvious win that even a chess novice can recognize after a moment's thought and a bit of counting. The White rook pawn is closer (in moves) to its promotion square than the Black king and its advance cannot be impeded in any way.

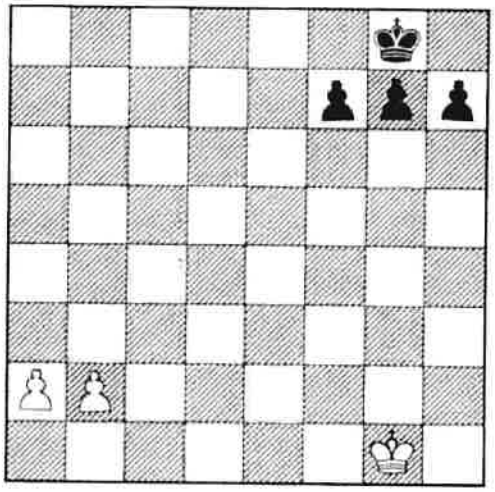

Figure 3.7 White to move and win.

This observation, however, involves long-range planning (being aware that pawn promotion is desirable and that this goal is highly relevant to the present situation) as well as some dynamic computations to determine if the desired goal can be attained. Unfortunately for the machine, a shallow search recognizes only those outcomes that exist within the search horizon so that the promotion of the pawn is not discovered until the event becomes part of the game tree.

On the basis of a static analysis of this position, the machine notes that Black has three pawns and White has only two and thus assumes that Black has the advantage. If the machine were White and were offered a draw, it would readily accept even though this would be utter folly. Static evaluation can be very misleading.

To get a clear idea of the problem, let us analyze Figure 3.7 by having the Northwestern program (CHESS 4.4) conduct a full-width search on this position using the α–β minimax strategy. With a 3-ply search, the machine (Control Data Corporation 6400) examines 148 nodes in the game tree in 0.3 seconds and selects a principal variation of (1) P–N4, P–B4; (2) P–R4. Based on this 3-ply search, the machine evaluates the position as favorable to black with a rating of -107 (a pawn is worth 100 and the minus sign indicates that Black has the advantage). Obviously, the machine "misunderstands" the position since the correct move is P–R4 and White has a clear win.

By deepening the search, we can attempt to enlighten the machine. With a 5-ply search, the computer examines 990 nodes in 2.1 seconds and selects a principal variation of (1) P–N4, P–B4; (2) P–R4, P–N4; (3) K–B2. The machine is still blind to the promotion opportunity and now evaluates the position at -115. The selected first move, P–N4, could easily lose the game to a competent opponent.

With a 7-ply search, the computer examines 4523 nodes in 8.6 seconds and selects a principal variation of (1) P–N4, P–B4; (2) P–R4, P–N4; (3) K–B2, P–R4; (4) K–N3. The evaluation function now is set at -116. The machine is still totally blind to its clear opportunity. With a 9-ply

search, the machine finally discovers the clear advantage of pushing its rook pawn. This occurs because a 9-ply search involves 5 moves for White and the pawn can reach the eighth rank in 5 moves. In conducting this search, however, the machine makes use of the "horizon effect" to delay the pawn promotion. In the 9-ply search, the computer examines 48,273 nodes in 96.5 seconds and selects a principal variation of (1) P–R4, P–R4; (2) P–R5, P–R5; (3) P–R6, P–R6; (4) P–R7, P–R7+; (5) KxP. Its evaluation function improves by about 100 points since it believes that it can win a pawn. In fact, the machine selects P–R4 because it anticipates that this move will lead to a pawn capture! It does not understand that Black's sacrifice of a pawn does not prevent the ultimate promotion of White's rook pawn. Since the pawn sacrifice "pushes" the pawn promotion over the machine's horizon (i.e., beyond 9-plies) the computer assumes that the promotion has been permanently prevented. This ridiculous strategy of self-delusion plagues the machine in many different environments.

The position in Figure 3.7 demonstrates several major weaknesses of a full-width shallow search. The machine's evaluation indicating that Black has the advantage shows why a static evaluation emphasizing material is a poor substitute for a dynamic analysis of the position. Secondly, the pawn sacrifice by Black indicates the foolish behavior which is engendered by the horizon effect. Thirdly, the absence of long-range planning becomes painfully evident when the machine copes with this position by examining 48,273 positions in 96 seconds (at a million operations per second) instead of simply counting squares. At the end of all this labor, the computer picks the right move for the wrong reason. The machine would need an 11-ply search in order to select the right move for the right reason. Clearly, a conventional full-width search is not the right approach for positions such as the one in Figure 3.7.

The problem with long-range planning becomes painfully evident when a machine competes against a human. In the simultaneous exhibition between David Levy and twelve different machines (Minneapolis, October, 1975), CHESS 4.4 as White walked into a devastating mating attack developed by Levy over a sequence of many moves (see Figure 1.15). Levy, taking advantage of the weakened king side after Bird's Opening, carefully positioned his queen [(7) . . . Q–B2], his king bishop [(10) . . . B–Q3], and both knights [(2) . . . N–KB3 and (9) . . . NxP/K4] to bear down on the hapless White King.

Because it has no long-range planning, the machine had considerable difficulty "understanding" the purpose of Levy's opening strategy nor did it anticipate the piece sacrifices. From the machines perspective, these sacrifices involve the loss of material by Black without any immediate compensation. Without even a vague idea of Levy's long-range plan, the machine was quite happy to accept the material and thereby increment its evaluation function. After castling king-side, the machine had to take steps to thwart Black's attack. It made a fatal mistake with (10) B–K2, re-

treating the bishop to avoid losing the minor exchange. It could not afford this lost tempo because a developing move such as (10) N–QB3 was essential. The machine, however, lacking a long-range planning facility, had no idea that defensive maneuvers were necessary.

The "horizon effect" that appeared with the 9-ply search in the previous end-game example can also cause much more malevolent effects. Berliner has provided a clear example of one such disastrous situation in Figure 1.3 of his thesis [12]. Figure 3.8 presents this position with White to move. White's white-squared bishop is trapped and cannot be saved. Therefore White's strategy should be to get as much as possible for the bishop. The problem for the machine, however, is that the bishop capture can be delayed by sacrificing material worth less than the bishop. If these material sacrifices can push the eventual capture of the bishop over the search horizon, the machine will " believe" that it has saved the bishop.

Let us examine the way in which the Northwestern program deals with this position. With a 2-ply search, it examines 211 nodes in 0.6 seconds and selects a principal variation of (1) B–QN3, P–QB5. It's evaluation is 247 since it has a bishop advantage. It does not realize that the bishop is trapped because its 2-ply search examined only those capture sequences which might be initiated by White at the end of the principal variation. With a 3-ply search, it discovers that B–QN3 does not work and then "invents" a foolish plan to save the bishop. It examines 1137 nodes in 3.4 seconds and decides that (1) P–K5, PxP; (2) N–Q5 is its best strategy even though its evaluation function decreases to 176. The machine observes that if (2) . . . PxB, then (3) NxB+ or if (2) . . . NxN, then (3) RxN, PxB; (4) RxB. In each case, the machine's analysis past the third ply is based only on a direct sequence of captures. The problem here is that the machine's tactics merely delay the loss of the bishop by sacrificing an additional pawn and giving up a positional advantage. The machine fails to understand this position until it conducts a 6-ply search, examining 168,774 nodes in 442 seconds. At this depth, it selects a more promising principle variation of (1) B/2xP, PxB/5; (2) P–K5, N–R4; (3) PxP,

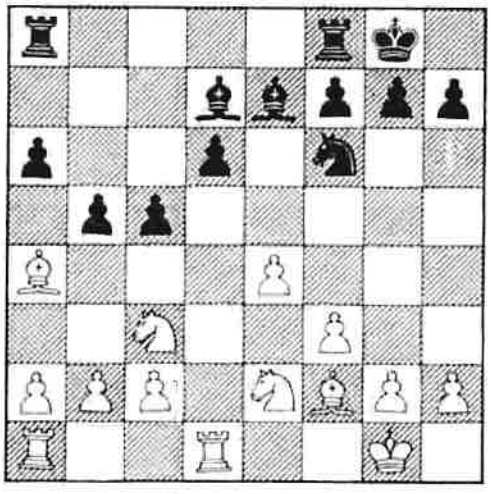

Figure 3.8 White to move [from Berliner (12)].

B–KN4. With this variation, White exchanges the bishop for two pawns and maintains a positional advantage.

Another instance of the horizon effect occurred at the 1975 ACM computer chess tournament in Minneapolis. CHESS 4.4 (White) and TREE FROG (Black) competed in the fourth round for the championship (see Figure 1.13). With its twelfth move, White placed a pawn on the seventh rank. Because CHESS 4.4 places a high value on a pawn in this position, it was unwilling to give up the pawn without a battle. Unfortunately for CHESS 4.4, it could not find a viable plan to save the pawn, so it used the horizon effect to engage in a bit of self-deception. Its next few moves served to delay the eventual capture of the pawn. The move, (13) P–KR3, provided a delay by forcing Black's queen bishop to retreat as did the move, (14) P–QR3, forcing Black's king bishop to move. The next move, (15) P–KN4, maintained this delay strategy by continuing these harassing tactics. Each of these moves served to delay Black's capture of the White pawn on the seventh rank and thus "successfully" pushed this capture beyond the search horizon. The price that CHESS 4.4 paid for this folly was that the positional advantage it had gained through its first twelve moves was virtually dissipated and its pawn structure on the king side was seriously weakened. Not until Black captured the aspiring pawn on (19) . . . RxP did CHESS 4.4 resume its usual solid game. The partisan observers from Northwestern gave a collective sign of relief when the pawn finally succumbed to the Black rook.

The horizon effect can lead to the meaningless sacrifice of material and the loss of positional advantage when the machine would otherwise have an excellent game. If a rook or queen becomes trapped and cannot be saved, the horizon effect will encourage the machine to offer up an endless procession of pawns and minor pieces to delay the eventual capture. In this instance, a tree search of 9 or 10 plies (not presently possible in the middle-game) might not find the correct continuation. This problem can only be solved by having the program terminate its search at each node only when the position is truly quiescent. Thus it is clearly necessary to improve the machine's ability to discriminate between active and non-active nodes.

The opening

The look-ahead procedure is relatively weak for selecting good moves in the opening. The opening emphasizes the development of pieces to squares where they can effectively do battle some 20 or 30 plies later. Since these future battles are beyond the machine's look-ahead horizon, it must develop its pieces purely on heuristic grounds (e.g., control the center, knights before bishops, prepare castling, etc.). Because these rules of thumb often lead to imprecise play, many programmers have decided to use a common human strategy, i.e., memorize many of the standard openings and play them by rote. The computer can easily be programmed to play "book"

openings. There is no reason, other than the work involved, why a machine could not have an opening library which covered as many as 100,000 positions. Most tournament programs now have between 3000 and 10,000 positions in their libraries.

Since moves can be accessed from this library very quickly, machines play the opening at blinding speed and it is often difficult for human observers to move the pieces about the board fast enough to keep up with the play. A library of openings not only insures the machine a decent level of performance in the early going but also conserves valuable clock time for the middle game where some lengthy calculations may be necessary. In science, however, we have a law (called Murphy's law) which states that if something can go wrong, it will and in a way which will cause the most damage. Sadly enough, this principle applies with a vengeance to computer play with a memorized opening library.

Sooner or later, regardless of the size of the library, the machine will exhaust its store of predigested moves and will have to think (i.e., calculate) on its own. The problem arising is that the board position it encounters when leaving its opening library is not of its own making. The grandmaster's "plan" or "idea" in selecting the opening moves is not available to the machine so that it must rely upon its own static evaluation function to determine whether its pieces are in their "proper" positions. Of course, the machine decides that they are not because its evaluation function is much less sophisticated than that of a grandmaster. The end result is that the machine devotes its first several moves to rearranging its pieces into a new configuration more compatible with its evaluation function. In doing this, it usually loses several tempos and often gets itself into an inferior position. It is interesting that the machine's failing in this regard is not that dissimilar to the difficulty that some novice players have when they laboriously memorize book after book of openings without learning the theme or idea behind each opening.

It is probably unrealistic to expect that computer chess will be played without a library of openings. If improvements do occur in opening play, therefore, they will probably result from organizing the library to store more than just a sequence of moves. For each position the library should catalog both a move and thematic information concerning the type of strategy which is appropriate to this position. This could be done by giving the library the capability of modifying the static evaluation function when the machine leaves the library. The modified function might encourage the movement of certain pieces to particular squares or to control certain squares. Even minor revisions along these lines would produce a major improvement in machine play during the late opening and during the early middle game. Since the machine must survive the opening and the middle game in order to reach the end game without a lost position, it is probably appropriate to devote considerable effort to this problem.

An interim solution which has been adopted by most tournament chess-programming teams is to carefully select "book" openings compatible with

their machine's style of play. If the machine's evaluation function emphasizes material rather than positional factors, the program will appear strongest in tactical positions. Thus the programmers can select library moves that quickly lead to active positions. If the evaluation function carefully considers pawn structure, the opening library can be structured to produce positions which permit pawn doubling or pawn isolation. Considerable skill is required to artfully select those openings which make the most of the machine's playing style.

The end game

Levy [67] and other observers have felt that the end game is the most difficult problem for machine chess. The end game requires highly specialized knowledge that is used relatively infrequently. The winning plan may involve an exact sequence of 20 or more moves. A full-width search to this depth is totally impossible. For this reason, it may be necessary to abandon the α–β minimax procedure in the end game and adopt an alternate strategy.

One promising plan is to structure the heuristic band-width search (see Chapter 7) to examine only those portions of the game tree relevant to some specific goal, such as pawn promotion. This strategy requires the machine be able to recognize the beginning of the end game and then determine which goals are feasible. The pattern recognition these decisions require is not an easy problem for a machine but it is one that must be solved if this approach is to be successful.

A second strategy is to analyze specific types of end games in some detail and then to develop specific programs to deal with each of these in an algorithmic fashion. Thus the machine will have a detailed set of instructions for every position it might encounter in a king and rook versus king ending or a king and pawn versus king ending. This approach is discussed in detail in Chapter 5. If this strategy can be generalized to more complex end games, it may provide a partial solution to this problem.

Improvement through competition

One of the advantages of research on machine intelligence in a chess environment is that new ideas, once implemented, can be easily evaluated in tournament play. Academicians have many ideas that persist in the literature simply because their truth or falsity can not be easily tested. This is unfortunate because it slows the pace of progress. Chess programming has the advantage that new ideas can be implemented and tested within a period of a few weeks. Several innovations in Northwestern's program were tested in a few early morning sessions simply by playing the new version of the program against the old version and noting the outcomes.

The annual computer chess tournaments held by the Association for Computing Machinery provide a national testing ground for chess pro-

grammers. This annual competition is useful for uncovering the important weaknesses which most programs inevitably have. It is true that these tournaments are expensive but their value could be maintained by restricting each machine to 20 or 30 seconds per move instead of 3 minutes. A "blitz" tournament would still provide each programming team with an opportunity to test their innovations but at a much reduced cost in machine time and in long-distance telephone expenses. At the same time, the audience enthusiasm that Mittman discusses in Chapter 1 would probably be augmented because the games would proceed at an exciting pace.

Greenblatt [48] has avoided the ACM tournaments and has entered his program exclusively in human tournaments. In addition to developing a USCF rating for his program, this competition has provided a broader test of the program's skill then would be the case for machine tournaments. Humans show much greater diversity in playing style than do present computer programs and this characteristic may be an important aspect of the testing environment. Tournament play against humans also provides some special opportunities not present in machine tournaments. With human opponents, the opening library can be used more effectively than with machine opponents. With two machines, book-after-book of openings can be read into each with neither gaining an advantage. With play against a human, however, the enlargement of the machine's library should have beneficial results. Secondly, the human's propensity for an occasional blunder (such as inadvertently leaving a piece *en prise*) should work to the machine's advantage. Such events never occur in computer tournaments. Thirdly, most human players are unaware of the common weaknesses of Shannon-type chess programs and thus would not be able to capitalize so easily on these shortcomings. It is interesting that the Northwestern program has competed very favorably in the few human tournaments it has entered. Slate and Atkin have been surprised at the relatively poor play exhibited by the machine's opponents. It is often difficult for humans to adjust to the machine's unnatural style of play and many face it with unjustified trepidation; their play suffers accordingly.

For example, CHESS 4.5 recently entered the Paul Masson American Class Tournament in California (July, 1976). Playing on a large scale Control Data Cyber 170 system, the program was matched against 5 human opponents in the B section who had an average rating of 1735. The machine won all five matches. This result was quite surprising because the program at a conceptual level is clearly in the C class or below (see Hearst's chapter for a more thorough discussion of this). Apparently the machine's abilities to avoid miscalculation and to never miss an opportunity seem to provide a certain amount of compensation for its conceptual inadequacies.

Future prospects

Chess, as a problem environment, is representative of a large class of problems that machines have been unable to master. The "cycle time" for

the human brain is relatively slow (4 or 5 operations per second as compared to several million per second for modern computers). Despite this, humans clearly outclass machines as problem solvers. Through evolution, man has developed specialized skills to compensate for the slow speed of biological information processing. Two major developments are prominent in this regard. One is the large degree of parallel processing that occurs very early in the visual and auditory systems. This permits complex pattern recognition, an essential ingredient in human problem solving. A second important development is a sophisticated storage and retrieval mechanism for accessing information. Machine representations of knowledge lack the richness and semantic meaningfulness of human memory. The computer needs to emulate the human brain in being able to recognize key features of a position, to know which actions are appropriate to these features, and to implement tactics that are thematic with an appropriate long-term strategy. Developments in these areas are essential for "intelligent" machine chess.

David Levy [67] has a bet with several academicians that no computer chess program will be able to beat him by 1978. His prospects for winning this bet are quite good considering that no present program has yet attained even an expert level of play. Given the tremendously difficult conceptual problems involved, Levy may even have an outside chance of winning such a bet if the deadline were extended for another decade.