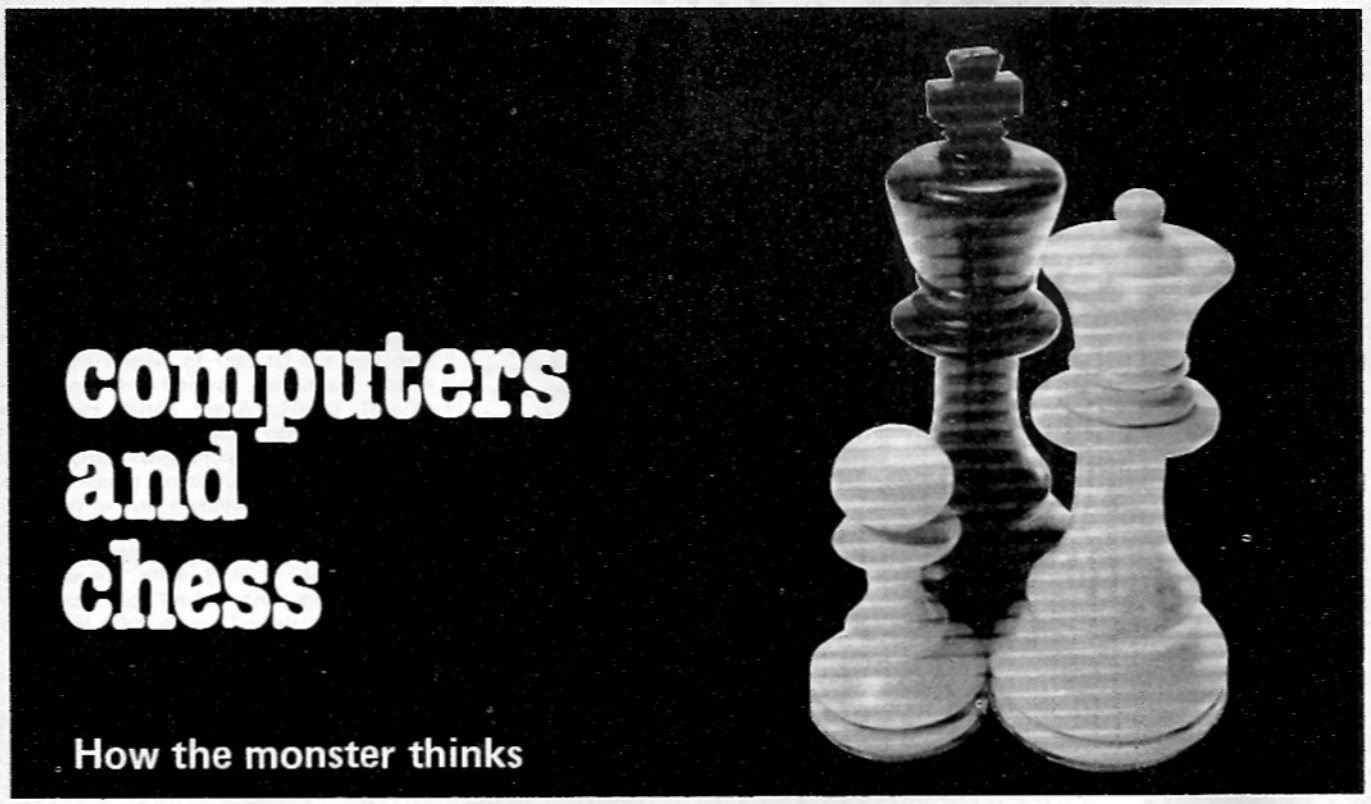


# computers and chess

## How the monster thinks

**The game of chess has long been regarded as a symbol of man's intellectual prowess. Until recently, the prospect of a chess-playing computer defeating a master-strength human opponent seemed remote. A few months ago, however, in a much publicised match an International Chess Master, David Levy, actually lost a game to a program from North America. The story of the match is recounted by Mr. Levy himself elsewhere in this issue. The following article takes a look at the background to computers and chess: how they play, their weaknesses and strong points, and speculates on the chances of Karpov being the last flesh-and-blood World chess champion!**

Thirty years ago, with the electronic computer still in its infancy and illustrating above all else the First Law of Thermodynamics ('Work is Heat'), the game of chess attracted the interest of a number of researchers in the field of artificial intelligence. The first person to actually propose how a computer might be programmed to play chess was the English Mathematician Claude Shannon. In 1949 he presented a paper entitled 'Programming a computer for playing chess', which was significant, both for the fact that it was the first paper expressly on this subject, and more importantly since many of Shannon's ideas are still employed in the strongest chess-playing programs of today.

Shannon's interest in chess programming stemmed from his belief that the game represented an ideal test of machine intelligence. Chess is clearly defined in terms of legal operations (moves of the pieces) and ultimate goal (checkmate), whilst being neither so simple as to be trivial nor too complex to be unsusceptible to analysis.

### Board, pieces and moves

Shannon suggested that the machine represent the chess board by assigning a location in computer memory to each square of the board. Each piece is then designated as a numerical value: +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, and so on. These numbers are stored in the memory location which represents the square occupied by the corresponding piece. An empty square is represented by storing a zero in the appropriate location. A number of more recent programs also adopt this method, with the exception that a 10 x 12 board is used instead of 8 x 8, and that a unique number (such as 99) is stored in all the off-board locations, thereby allowing the program to detect the edge of the board. This is illustrated in figure 1, where the addresses for each square are given in the top left-hand corner and the contents (before the game starts) of the memory locations are also shown.

The program generates legal moves simply by noting the mathematical relationship between the addresses of the different squares. For example, the addresses for each square may be assigned as shown in figure 1. Then, to calculate the possible legal moves of, say, a king standing on e1 (square 25) one adds the offsets +1, +9, +10, +11, −1, −9, −10 and −11 to that address. The program then checks the contents of these new addresses to determine the legality of the move. If the location contains the number 99, the square is off the board and the move illegal. If the location contains a positive number, the square is already occupied by a white piece. If the contents of the location are negative, on the other hand, the king can legally move to that square capturing an enemy piece (always assuming that the piece is not defended). Finally, a location containing a zero also represents a legal move assuming that the corresponding square is not attacked by an enemy piece.

Calculating the legal moves for a sliding piece such as a bishop, is only slightly more complicated. With a white bishop situated on square XY (e.g. 54, where X = 5 and Y = 4), the program examines

address [X + 1, Y + 1] (i.e. 65), checks to see whether the contents of that location are zero, and if so proceeds to examine address [X + 2, Y + 2] and so on (if [X + 1, Y + 1] turns out to contain a negative number, then the bishop can move to that square, capturing a piece, but obviously can move no further along that diagonal). The machine repeats the above procedure for [X – 1, Y – 1], [X – 2, Y – 2] etc, then does the same for [X – 1, Y + 1], [X – 2, Y + 2] etc., and finally for [X + 1, Y – 1], [X + 2, Y – 2] etc. In this way the program can generate legal moves along all four diagonals of the bishop.

Similar operations can be performed to determine the legal moves of all the pieces, although one must bear in mind that certain moves have to be checked for the existence of pins (is the piece pinned against the king, for instance) and the procedure is complicated when considering castling and capturing *en passant*.

## A more 'logical' approach

The above approach is still adopted by many modern programs, although an alternative method which is particularly suited for use with large computers has subsequently been developed. This utilises the fact that certain large computers operate with 64-bit words. If one bit is assigned to each square, only 12 64-bit words suffice to represent the position of all the pieces on the board. For example, one word will provide information on the position of all white pawns on the board by setting a bit to '1' for each pawn that resides on the corresponding square. If a square is empty the bit remains unset ('0). A second word will represent the position of all the black pawns, a third the position of both white knights, and so on.

In addition to the position of pieces, these 'bit maps' or 'bit boards' as they are called can be used to represent other information. For example, one 64-bit word might represent all the squares attacked by white's pieces, another all squares which are a knight's move away from the black king, and so on.

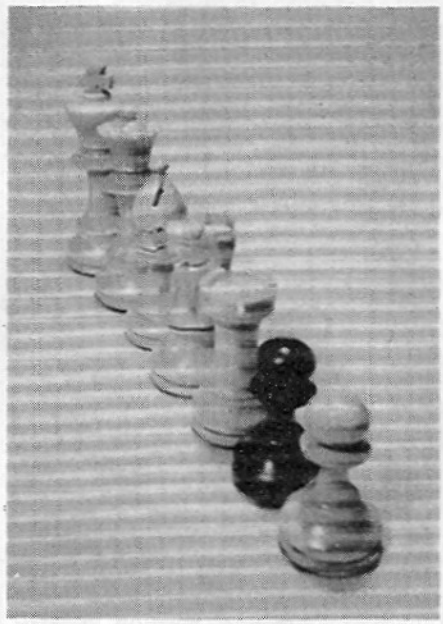
The real advantage of this alternative approach can be seen if one considers the instruction set of a modern computer, containing as it does a number of 'Boolean Logic' operations. These can be used to combine considerable amounts of information stored in bit maps. For example, assume that we wish to know whether white has a knight's move that will fork black's king and queen. One simply fetches two bit maps of potential knight's moves from the black king and queen respectively, and a bit map of knight moves from their present squares. Since the square may not be occupied by a white piece, a map of all white pieces is inverted and then ANDed with the first three maps. If the result is non-zero, then a forking square exists. Finally, this map is ANDed with a map representing all squares attacked by black pieces to determine whether the forking square is defended. It can be seen that the above operation takes very few program steps.

## Looking for good moves

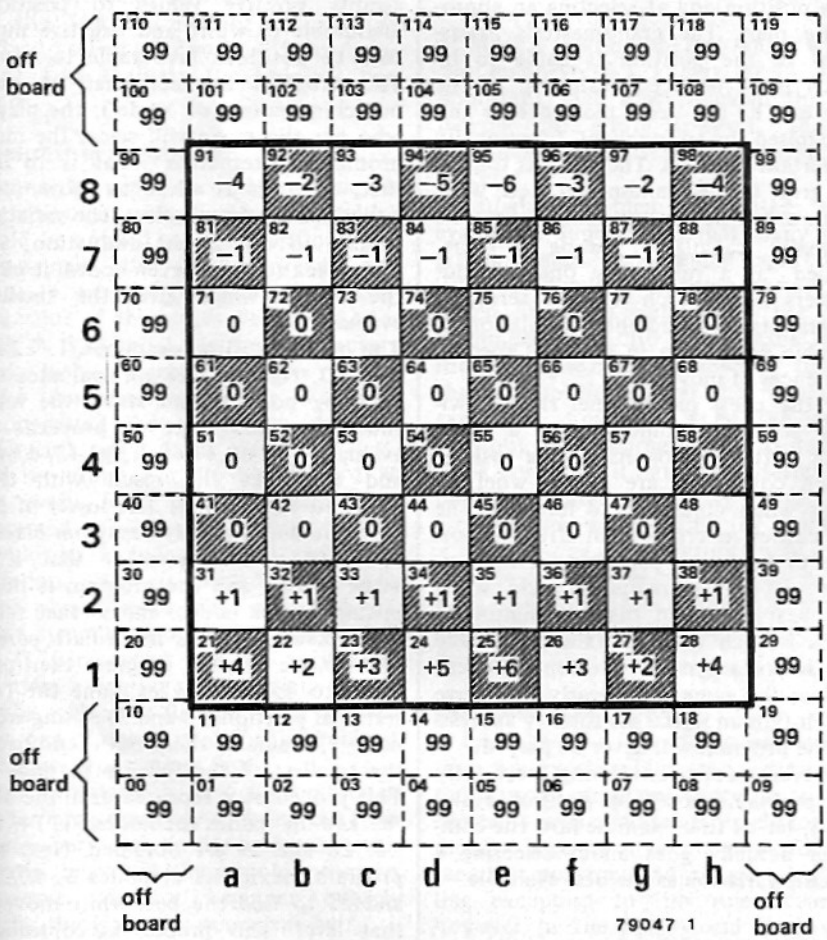
Having enabled the program to generate legal moves, there comes the problem of selecting the good from the bad. This is where the difficulties start to accumulate. The most obvious approach is to have the program examine all legal moves by white, followed by all legal replies by black, all legal counter-replies, and so on to a fixed depth.

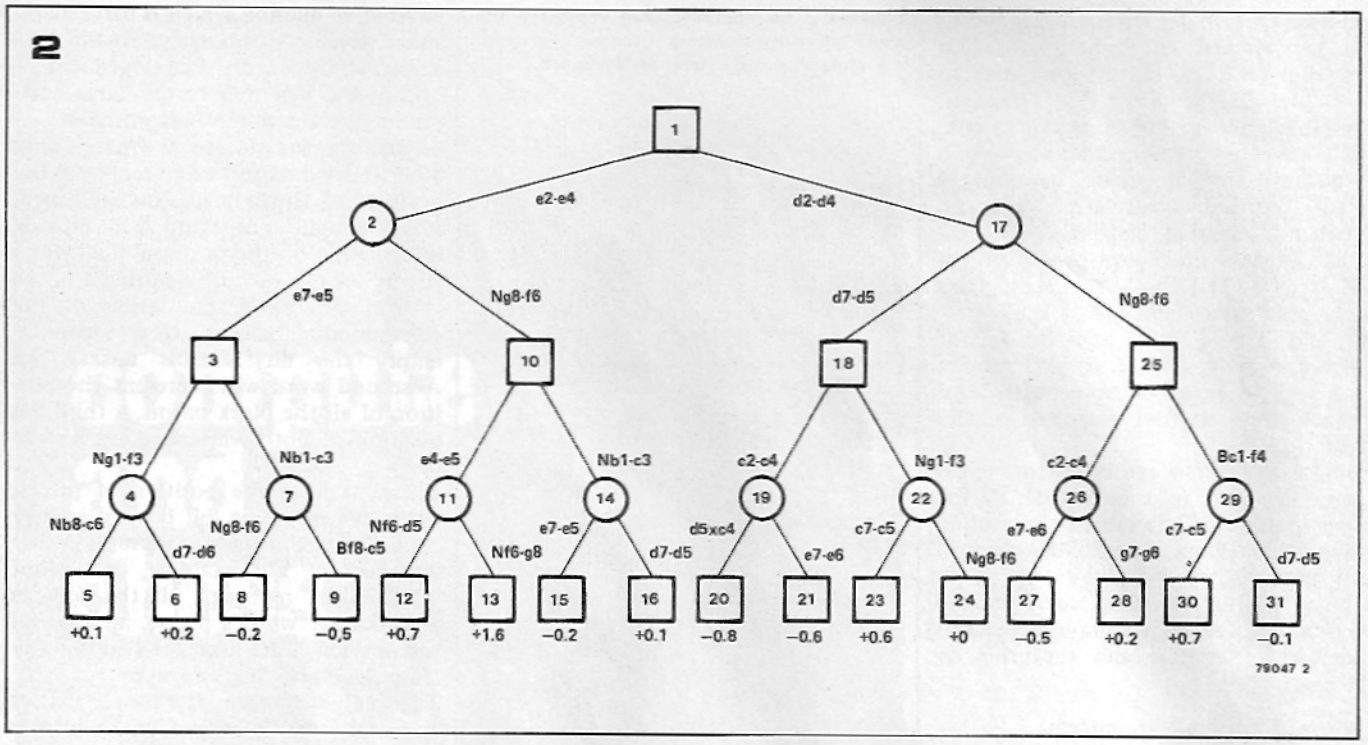
This procedure, which Shannon called the 'type-A strategy' does however suffer from a number of serious drawbacks. The number of legal moves in each position is on average around 38. This means that a 2-ply analysis of all legal moves (i.e., one move each side; ply = 1/2 move) would produce $38^2 = 1444$ terminal positions to be evaluated. An analysis only 4-ply deep would yield 2,085,136 terminal positions, and a mere 6-ply look-ahead would involve evaluating some

Figure 1. The computer can represent the chess board by assigning a location in memory to each of the squares on the board.



1






3,010,936,389 positions! Because of this 'exponential explosion' as it is called, an exhaustive look-ahead of this type rapidly becomes unmanageable.

A second disadvantage of a fixed-depth exhaustive look-ahead is that the machine may well terminate its search in the middle of a series of exchanges – with the result that its evaluation of the position will be hopelessly wrong. It may, for instance, be deceived into thinking it is a piece ahead, when in actual fact it is about to loose a piece back – or even worse. A fascinating example of this type of 'computer blindness' will be given later – in the game COKO v. GENIE.

Shannon was well aware of the inherent problems of a type-A strategy, and therefore proposed an alternative model which he called type-B strategy. The latter is characterised by the notion of 'quiescent' positions, i.e. the program is encouraged to continue its search until all forcing variations are exhausted and the position for evaluation is 'static'. More importantly, a B-type strategy will not attempt to generate all legal moves in a given position, but rather will select a small number of 'plausible' moves for subsequent analysis. This approach obviously requires that the program have some criteria by which it can select the more promising moves from those which are plainly irrelevant, i.e. that the program have a 'plausible-move generator'.

The interesting feature of the type-B strategy is that it attemps to simulate the approach of the most efficient chess model we know of, namely the human. Contrary to uninformed opinion, the chess master does not analyse dozens of moves deep and investigate hundreds of different variations before selecting a move. Quite the reverse is true. Research carried out by a Dutch psychologist, de Groot, revealed that, in a fairly typical middlegame position, Grandmasters tended to look at only three or four different possible moves, and that the maximum depth to which they calculated was not much more than 7-ply! However, the grandmaster is adept at perceiving the critical features of a position and at selecting an appropriate plan. The grandmaster's assessment of the position is liable to be much more nuanced than that of the amateur; he has 'seen' more deeply and recognised the truly salient, functionally important features. The story is told of the great Czech grandmaster Reti, who, when asked how many moves ahead he normally calculated during a game, replied 'as a rule, only one'. Grandmasters think much more in terms of general strategy and the formulation of suitable plans than in terms of specific sequences of moves.

For the chess programmer this knowledge comes as something of a blow, since pattern recognition is a task at which computers are as yet woefully inept when compared to humans. The difficulties in creating an effective position evaluator and plausible-move generator are enormous, particularly when one bears in mind that the nature of chess is such that the failure to make one important move is often sufficient to lose the game, and clearly any move which fails an initial plausibility analysis by the program will never be played.

However before considering more fully the problems posed by position evaluation, let us first examine how the computer actually goes about selecting a specific variation as the best available.

## Growing trees

Shannon suggested that the program adopt the 'minimax' procedure first proposed by Morgenstern and von Neumann in their work on game theory. Basically, the program grows a 'tree' of variations. An example of a simplified game tree is given in figure 2, which starts from the initial position with white to move and assumes that some form of static evaluation function awards positive values to positions favourable to white and negative numbers to positions favourable to black. The program assumes that, at each branching point (or 'node'), the player who has the move will select the most promising alternative. That is to say that, when it is white to move (odd nodes), the program selects the variation leading to the largest evaluation, and with black to move (even nodes) it picks the branch which gives the smallest evaluation.

The program first examines 1. e2-e4, e7-e5 2. Ng1-f3, Nb8-c6, evaluates the resulting position and stores the value thus obtained. Next it proceeds to evaluate 1. e2-e4, e7-e5 2. Ng1-f3, d7-d6, and compares the result with that obtained for node 5. The lower of the two values is obviously best *from black's point of view* (remember that it is black's move and the program is minimising at even nodes) and so that value is 'backed up' to its immediate parent node (node 4). The program then proceeds to successively examine the two terminal positions (8 and 9) arising from node 7, evaluates them both, and backs the smallest of the two up to node 7. This procedure is repeated until the best 'backed-up' values for nodes 11, 14, 19, 22, 26 and 29 are obtained. Next the program maximises at nodes 3, 10, 18 and 25 to find the best white move at that level. This process is continued, 'minimaxing' back up the tree, until the best move for the current position is determined.

Figure 2. A simplified game tree.

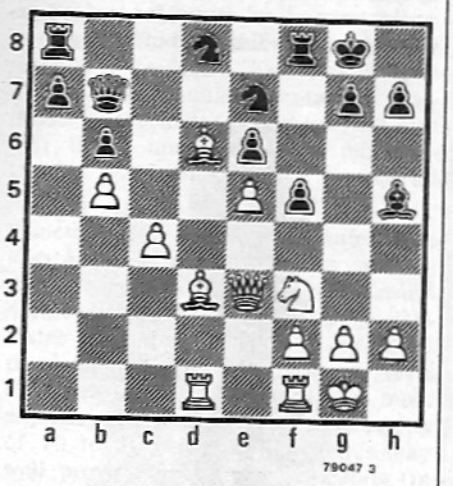
Although this procedure seems 'logical', a full-width search to the depth shown here (4-ply) would produce, on average, some two million terminal positions for evaluation. Fortunately, subsequent research showed that techniques can be employed which result in a substantial pruning of the game tree. A more fundamental problem, however, is presented by the 'bottom line' of the tree: in order to select good moves, the program must first evaluate the terminal positions.

## Evaluating positions

Shannon's paper provided a simple example of an evaluation function which could be applied to static positions. Not surprisingly, the greatest weight was given to material balance and the relative value of the pieces were assessed as 200, 9, 5, 3 and 1 for the king, queen, rook, bishop/knight and pawn respectively. Positional evaluation was then incorporated by penalising doubled, backward or isolated pawns (= –½) and rewarding mobility by adding 1/10 for every legal move. Shannon also suggested additional features which should be included in the evaluation function, such as control of centre, open or semi-open files, passed pawns, pawn structure around the king, and so on. It is important that one arrive at an accurate weighting of the various factors in the evaluation function, and this is in fact one of the most difficult problems the chess programmer faces.

Early programs in particular exhibited an alarming tendency to bring out their queens very early in the game, since this greatly increased their mobility score. However, as any beginner quickly learns, this is usually poor strategy . . .

The problem of writing an efficient evaluation function is compounded by the fact that the importance of certain positional features changes during the course of the game. Furthermore, a particularly thorny problem is the difficulty in assessing whether a 'terminal' position is truly 'quiescent', or whether it in fact occurs, say, half way through a series of exchanges. As mentioned, most programs attempt to resolve the latter problem by performing an additional search for all checks and captures until these are exhausted. However, this approach is at best a makeshift solution, since it fails to deal with purely positional manoeuvres which a strong human player would examine as part of his evaluation of the position. In the position in figure 3, for example, the most significant feature is the 'hole'




in Black's position at c6 to which White can manoevre his knight on f3. It is important that Black prevent this by playing Bh5 x f3. However, this is extremely difficult for a program to perceive.

A further problem associated with evaluation functions is that many programs contain an 'opening book' (i.e. lists of standard opening variations) which has been included by the programmers to ensure a reasonable position from the opening. However, due to the unsophistication of the program's evaluation function, when the book runs out and the program has to start to think for itself, it naturally assesses the position quite differently from the Grandmaster whose game (or analysis) it is following. It therefore spends the next few moves re-arranging its pieces until they correspond with the evaluation function's idea of where they should be!

## Computers are greedy

A typical fault of most programs is that they are excessively materialistic (even the Russian programs succumb to this capitalist evil) and are extremely loth to sacrifice a pawn or even a piece for less tangible, positional advantages. A startling exception to this rule occurred however in the first World Computer Chess Championships held in Stockholm in 1974.

Favourite to win was a North American program called Chess 4.0, written by three former students of Northwestern University: Larry Atkin, Keith Gorlen and David Slate. In the second round Chess 4. 0, which hitherto had been undefeated by another program reached the following position (as black) against another North American entry, CHAOS:


Black is a pawn ahead, having greedily consumed white's king pawn, however he is behind in development, and in particular is not yet castled. White seizes the opportunity to make a decisive piece sacrifice. What is surprising about this offer is that it must have been based on a purely positional evaluation of the resulting position, since the program could not possibly have seen sufficiently far ahead to ascertain that he would eventually more than recoup his investment

*16. Nd4 x e6!* . . .

This move has been praised as 'the finest ever played by a computer'

| | | |
|---|---|---|
| | | *f7 x e6* |
| *17.* | *Qe2 x e6 +* | *Bd6-e7* |
| *18.* | *Rd1-e1* | *Qb8-d8* |
| *19.* | *Bc1-f4* | . . . |

The threat is Bf4-c7

| | | |
|---|---|---|
| *19.* | . . . | *Ke8-f8* |
| *20.* | *Ra1-d1* | *Ra8-a7* |
| *21.* | *Rd1-c1* | *Nf6-g8* |
| *22.* | *Rc1-d1* | *a6-a5* |

Black has no good moves, white has a stranglehold on the position

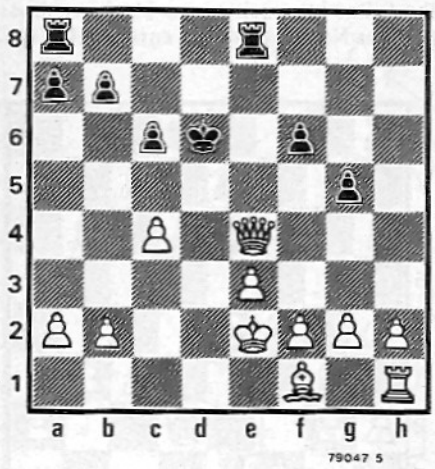
| | | |
|---|---|---|
| *23.* | *Bf4-d6* | *Be7 x d6* |
| *24.* | *Qe6 x d6 +* | *Ng8-e7* |
| *25.* | *Na4-c5* | *Bg6-f5* |
| *26.* | *g3-g4* | *Qd8-e8* |
| *27.* | *Bb3-a4* | *b4-b3* |
| *28.* | *g4 x f5* | |

And white eventually won, although it took another 47 moves to do so.

## Horizon effect v. snow-blindness

Selecting a suitable search depth also creates a number of problems when evaluating positions deep in the game tree. A particularly harrowing example of the fate that can befall a program when faced with a choice of equally promising continuations occurred in a –now notorious – game between two programs called COKO and GENIE

which was played during the second ACM tournament in 1971. After the first 27 moves the following position was reached with COKO (white) to play:




COKO thought for 120 seconds and offered a pawn to entice the black king out into the centre:

*28. c4-c5 + Kd6 x c5 ?*

Too greedy! COKO, having seen ahead the next 8½ moves, played:

*29. Qe4-d4 + .....*

And announced mate in 8!

*29. ... Kc5-b5*
*30. Ke2-d1 + Kb5-a5*
*31. b2-b4 + Ka5-a4*
*32. Qd4-c3 Re8-d8 +*
*33. Kd1-c2 Rd8-d2 +*
*34. Kc2 x d2 Ra8-d8 +*
*35. Kd2-c2 Rd8-d2 +*

Black's last four moves are a classic example of a typical computer weakness which is called 'the horizon effect'. Since the program's evaluation function scores the mate, i.e. the loss of its king as being very much worse than anything else that can happen to it, it indulges in a rather disarming piece of self-deception; it attempts to postpone the evil hour by sacrificing whatever comes to hand and so push the mate beyond the horizon of its look-ahead! Two rooks down is better than mate it thinks, however it doesn't 'realise' that it cannot prevent mate anyway.

The horizon effect is an extremely difficult problem inherent in all programs, particularly those with a short look-ahead. Any combination or manoevre which is longer than the look-ahead will be misperceived by the computer. However, just for once, maybe the program knew what it was doing, for there then followed:

*36. Qc3 x d2 Ka4-a3*

Still after the pawns!

*37. Qd2-c3 + Ka3 x a2*

White now has a choice of two mates in one (Bf1-c4, Qc3-b2) and a large number of mates in 2, 3, 4 etc. Unfortunately, COKO seemed unable to distinguish between the value of so many promising lines, and it made a random choice between the mating continuations:

*38. Kc2-c1 ....*

Black, not having much else to do played:

*38. ... f6-f5*
*39. Kc1-c2 f5-f4*
*40. Kc2-c1 g5-g4*
*41. Kc1-c2 f4-f3*
*42. Kc2-c1 f3 x g2*
*43. Kc1-c2 ...*

One can imagine the anguish of COKO's programmers

*43. ... g2 x h1 = Q*

This is now COKO's last chance, does he take it?

*44. Kc2-c1 ...*

Alas no. GENIE now played:

*44. ... Qh1 x f1 +*

whereupon white's game started to fall apart. COKO's unfortunate masters could soon stand it no longer and resigned for their disgraced offspring.

## Pruning the game tree

As was mentioned, a full-width search strategy to a depth of even a few ply rapidly generates an enormous number of terminal positions. However, in 1958, three researchers at the Carnegie Institute of technology, Alan Newell, John Shaw and Herbert Simon, published a paper which demonstrated that the number of positions which it was necessary to evaluate could be drastically reduced with the aid of a relatively simple algorithm. To understand how this is done let us look at the simple game tree of figure 2. If we propose a crude evaluation function of material balance, mobility (+0.1 for each legal move), centre control (+0.2 for each centre square, i.e. e4, d4, e5, d5, attacked) and king safety (defined as the number of moves required to castle: subtract 0.5 per move), we obtain the values shown.

To select its move the program first examines terminal positions 5 and 6, then backs up the smaller value to node 4 (+0.1), as explained earlier. The two positions descended from node 7 could then be evaluated, and the best backed-up value for it obtained (−0.5). However, since the program will maximise node 3 (white to move), we know that the best backed-up value for it will never be less than +0.1, since the program would always select the branch leading to node 4. Thus, having found the value for node 8 (−0.2), the program can deduce that it is pointless to generate and evaluate node 9: the evaluation for node 8 is already smaller than that backed up to node 4. The same argument can be applied throughout the game tree, resulting in a substantial pruning in the number of nodes requiring evaluation.

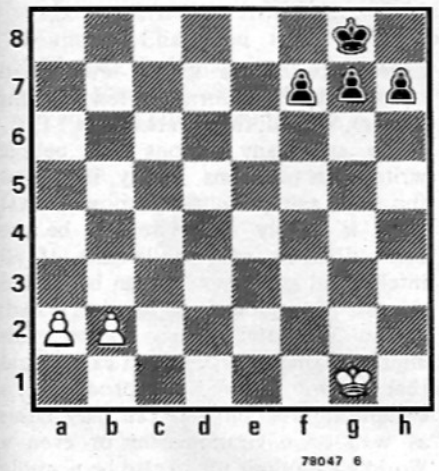
To obtain the full benefit of this procedure, it is important that the program examine the best moves first. Many moves fail to an immediate and obvious reply – the loss of one's queen for example. Clearly if the queen capture is generated and evaluated first, then it substantially reduces the number of nodes for subsequent evaluation. Modern programs all contain a number of 'heuristics' (rules of thumb) which provide the computer with information on the type of move it should examine first. One common heuristic involves the program storing refutation or 'killer' moves which were effective for positions earlier in the tree and testing to see whether they still work. Unfortunately, one effect of the capture heuristic is to make programs avid exchangers of pieces. Many programs dissipate an advantage by allowing the opponent to free a cramped game by trading off pieces. (See game 1 of Levy v. Chess 4.7.)

A number of additional techniques for speeding up the tree search have also been developed in recent years, and the full-width search has become a feasible proposition. However the fact remains that, with a type-A strategy, the program is searching blindly on a trial-and-error basis, generating and evaluating an enormous number of largely irrelevant positions. The computer is unable to formulate any sort of plan or, for that matter, recognise the plan of its opponent. It is at the mercy of the necessarily over-generalised positional factors of its evaluation function, and its vision is limited to a fixed-depth look-ahead. For this reason early programs in particular were atrocious at playing the endgame, often being unable to win such elementary positions as king and queen or rook against king. Unfortunately their human opponents were often unaware of this fact and resigned prematurely!

## How (not) to play the endgame

Most evaluation functions are oriented towards opening- and middle-game features (such as development, control of centre, etc.), whereas the endgame demands the ability to perceive a winning process. Often the winning plan will take twenty or thirty moves to execute, effectively ruling out the possi-

bility of the program stumbling across it in a full-width search. To the human this presents no problem because it simply becomes a question of implementing an idea, manoeuvring a piece to a key square, etc. The computer however does not have ideas, so it just sits there churning through tens of thousands of positions which all look pretty much the same to it anyway. A commonly cited example of the problems presented by the inability of programs to apprehend the salient features of a position, combined with a necessarily limited look-ahead, is shown in figure 6.




Any beginner faced with this elementary position would instantly recognise that the black king is too far away to prevent the white rook's pawn from queening. Unfortunately, if the program has a look-ahead of less than 9-ply it will fail to appreciate this fact, and assessing the position on the merits of material inequality, will decide that black has the advantage! Even with a 9-ply search it calculates the following variation: 1. a2-a4, h7-h5 2. a4-a5 h5-h4 3. a5-a6, h4-h3 4. a6-a7, h3-h2 + 5. Kg1 x h2. Black, by sacrificing the h-pawn has succeeded in pushing the pawn promotion over the program's horizon. However white will nevertheless select this line because it wins a pawn! The inability of the program to form a respectable plan is painfully embarrassing.

It seems likely that an alternative approach will have to be adopted for the endgame, and indeed promising work has been done in Russia on writing programs for specific types of endgame. David Levy lost a less well-publicised bet of a case of Scotch that the programmers of KAISSA would be unable to write a program which played the ending of king, rook and pawn against king and rook correctly for both sides before the end of 1975.

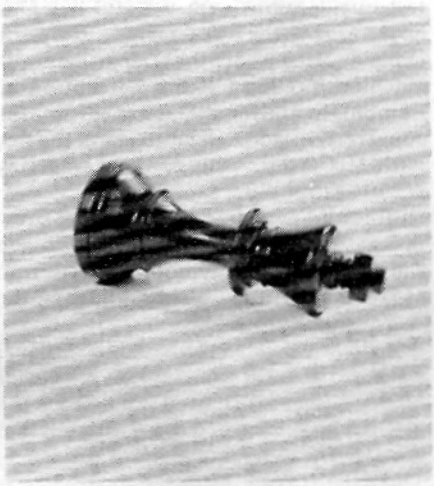
## What of the future?

Despite the inherent problems of a Shannon type-A strategy, there is no doubt that programs employing this technique have been making progress over recent years, whereas the difficulties involved in the development of a reliable plausible-move generator have remained largely intractable. In particular, Chess 4.7, the primary exponent of the type-A strategy, is gradually acquiring a rating near to that of an Expert of the US Chess Federation grading list, and more spectacularly has beaten an International Master under tournament conditions (as well as beating a Grandmaster in a blitz game) – see David Levy's article in this issue.

However these advances are to a certain extent due to progress in hardware – faster, more powerful computers – and more efficient programming techniques which increase the look-ahead of the program to a point where brute force exhaustive analysis is disguising its conceptual inadequacies.

In the endgame in particular much work remains to be done. The computer is at its weakest in quiet non-tactical positions where it cannot utilise the fact that, unlike humans, it never miscalculates a line, forgets about a piece *en prise* etc.

Nonetheless it cannot be denied that chess-playing computers are getting stronger and the best of them could defeat most average club-players. Estimates as to how long it will take before they play World Championship level chess are extremely difficult. The most popular guess is somewhere in the region of 10 to 20 years, although this may well prove to be wildly inaccurate (in 1958 Simon predicted that there would be a computer program as World Champion within 10 years).



*Literature:*

*Bell, Alex G., 'The machine plays chess'. Pergamon Press 1978.*

*Frey, Peter, 'Chess skill in man and machine'. Springer Verlag, New York 1977.*

*Levy, David, 'Chess and computers'. Batsford chess books, 1976.*

*Shannon, Claude, 'Programming a computer for playing chess'. Philosophical magazine, vol 41, pp 256-275, March 1950.*

# chess challenger 10

## plays like a human

The introduction of Chess Challenger 10 is an excellent 'state of the art' example of the space-age technology and almost human capabilities that can now be built into these computerised board games. It features no less than 10 levels of play:

| Level | Average Response Time |
|---|---|
| 1. Beginner | 5 seconds |
| 2. Intermediate | 15 seconds |
| 3. Experienced | 35 seconds |
| 4. Advanced | 1:20 minutes |
| 5. Superior | 2:20 minutes |
| 6. Mate in Two (two move puzzles and end games) | 1 hour |
| 7. Postal Chess | 24 hours |
| 8. Expert | 11 minutes |
| 9. Excellent | 6 minutes |
| 10. Tournament Practice | 3 minutes |


The levels are interchangeable at any time and during any move of a game and the player can select offense (light pieces) or defense (dark pieces) at the touch of a key.

A new feature of Chess Challenger 10 is its ability to play and follow a patterned classic book opening, e.g. Sicilian, French, Ruy Lopez, Queen Gambit Declined and so on.

Computer technology is growing so fast that the microprocessor 'brain' in Chess Challenger 10 can now analyse up to 3,024,000 board positions before making its move – a logic capability equal to the instinctive ability of even very experienced players.

An override key permits the player to make multiple moves before the computer responds and also permits the addition or subtraction of any piece to either side. Ideal for cheating!

Chess Challenger 10 sells for £ 200.