Chess Programming

Before You Begin

David Levy

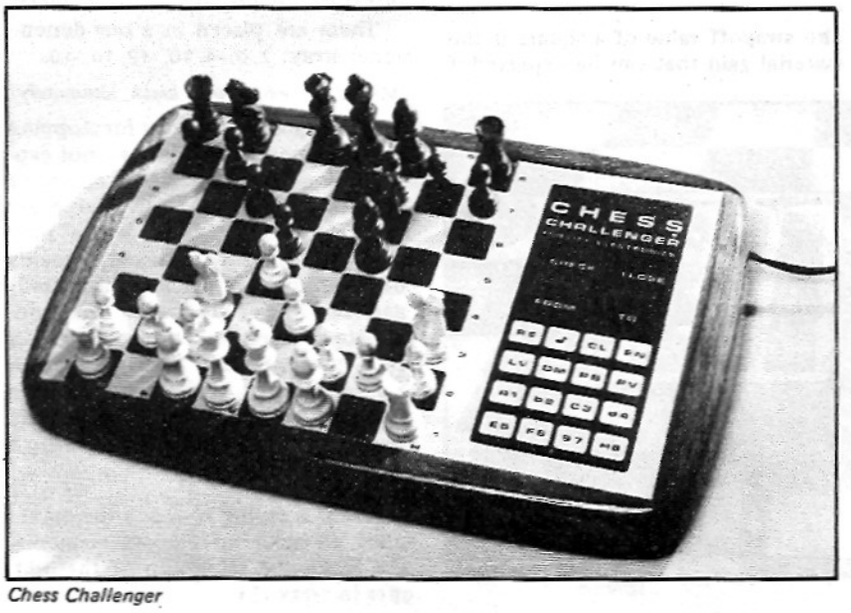
Until comparatively recently one essential prerequisite for any budding chess programmer was the odd £million or so needed to pick up a suitable mainframe. Nowadays it is possible for any home computer enthusiast to write his/her own program for a microprocessor based system costing less than £1,000.

Indeed, so many personal computer users are becoming interested in this field that I thought it worthwhile to pen a few pages of advice and warning so that PCW readers can be steered along the right track. The potential of micro-programs can be judged from the fact that at the 9th ACM chess championships, held in Washington last December, two such programs performed very creditably alongside many of the world's strongest mainframe programs: SARGON (written by Kathy and Dan Spracklen) scored 2½ out of 4 and MIKE (Mike Johnson) scored 1½. (Also see my article in November 78 PCW).

The Game of Chess

Chess is an extremely complex game to play yet its rules are clearly defined, and the way that the pieces move can be learned within a few minutes. Despite this, many programmers do not take the trouble to program all the rules properly. *En passant* captures and some of the rules pertaining to castling are often overlooked, even in commercially available chess machines costing up to £200. There is no excuse for such sloppy programming and I would therefore suggest that before you write so much as one instruction of code, do make sure that you know all the moves of chess and the circumstances under which the special moves can be made. This information is readily available in some elementary textbooks on the game but it would do no harm to consult the official rules of chess, published by Pitman in Britain and by David McKay in the USA.

Once you are confident that you know the moves and their rules, it is time to learn something about the game itself. If you are an inexperienced player you could do with some help, so try to get some hints on the basic principles of the game from a player who knows his onions. You can learn a certain amount from books but a few evenings at the local chess club will pay greater dividends and besides, the club can always do with your membership fee. Remember, if you do not possess some understanding of the problem (i.e. what chess is about) you will never be able to solve it (i.e. write a strong program).



Chess Challenger

Computer Chess Literature

There exists a wealth of literature on the subject, mostly papers in learned journals and in books on artificial intelligence. Before you rush off and invent the wheel again it will be worthwhile to study some of the more important items in the literature. As an optional introductory paper, and certainly one of great historic interest, I suggest Claude Shannon's "Programming a Computer for Playing Chess", Philosophical Magazine volume 41 (7th series), pp. 256-275.

Certainly more readily available are three useful books on the subject. "Chess and Computers" by David Levy, published in Britain by Batsford and in the USA by Computer Science Press, is an elementary introduction to the subject. "Chess Skill in Man

and Machine", edited by Peter Frey (Springer Verlag) is an outstanding work at a more advanced level. "Computer Chess" by Monroe Newborn (Academic Press) lies somewhere between the two. The first two titles contain extensive bibliographies.

Where to Begin Programming?

This is not an appropriate place to describe in detail how chess programs work. Suffice it to say that the three key items in all chess programs are:
(i) A move generator which lists the legal moves from any position and which can generate a tree structure representing the possible variations in play.
(ii) A position evaluator which assesses the merit of positions in the tree.
(iii) A tree searching mechanism which determines optimal play for both sides and thereby selects the move which the program makes.

Since the tree of possibilities in chess is enormous it is necessary to use various "tricks" to speed up the search process. By far the most powerful trick is a device known as the alpha-beta algorithm which can plough through a tree with N terminal nodes (or positions), and select a move after evaluating something of the order of 6 $x\sqrt{N}$ positions (optimally 2 $x\sqrt{N}$ but optimality in chess is a pipe-dream). Since the tree is enormous, so is the saving achieved by employing the alpha-beta algorithm.

This algorithm is described in many parts of the literature but possibly the best description, which comes complete with an Algol-like version of the algorithm, is that found in an article by Knuth and Moore, "An analysis of alpha-beta pruning", in "Artificial Intelligence", volume 6 (1975), pp. 293-326. Do not worry about overflowing memory during the tree search. The alpha-beta algorithm needs only to retain one node at each level of the tree.

Of crucial importance in tree searching is your choice of the quiescence criteria. Prune off too few moves and the combinatorial effects on the growth of the tree will cause your program to think forever over its moves. Prune off too many and it will overlook some important tactical possibilities. Most programs perform an exhaustive search to some fixed depth then a selective search along tactical paths, but why not try to grow intelligent trees, just as strong human players do?

Your position evaluator may be as primitive or sophisticated as desired, but remember that there is a trade off between sophisticated evaluation (which takes a lot of time) and the reduction in the size of the tree that can be examined (because of the time taken to evaluate each node). It has yet to be determined whether sophisticated evaluation and a small tree search is better than a primitive evaluation function and a larger search. You should experiment before making your final decision — begin with an evaluation function containing only one or two terms (material and mobility are the most important) and build it up from there until you are satisfied with the results. Remember, your position evaluator is the most frequently used part of the program so it must execute quickly.

One useful device in the evaluation mechanism is what Donald Michie calls "swap-off". This is described incorrectly on pages 45-47 of "Chess and Computers", the errors having been copied over from Michie's original article on the subject which was published in 1966. I am indebted to Helmut Richter of Hamburg for the following, more accurate, description.

Swapoff Values

The swapoff value of a square is the material gain that can be expected if the side on the move makes the most of a capturing or exchanging sequence on that square. The purpose of using swapoff values is to decide whether or not a particular capture is worthwhile without the necessity for lookahead. It can also determine the safety of a square.

The basis for calculating swapoff values is that the side on the move should either make no capture on a particular square or should capture with his least valuable unit so that any recapture by his opponent will have minimal value.

Swapoff values are calculated in the following way. Assume that a black piece of value v_0 is defended by n black pieces of values $v_1, v_2, \ldots v_n$ in ascending order of value, and attacked by N white pieces of values $u_1, u_2, \ldots u_N$ in ascending order of value.

(w = white; b = black)

$w_1 = v_0$

$w_2 = v_0 - u_1 + v_1$

$w_3 = v_0 - u_1 + v_1 - u_2 + v_2$

$w_4 = v_0 - u_1 + v_1 - u_2 + v_2 - u_3 + v_3$ etc.

and

$b_1 = v_0 - u_1$

$b_2 = v_0 - u_1 + v_1 - u_2$

$b_3 = v_0 - u_1 + v_1 - u_2 + v_2 - u_3$

$b_4 = v_0 - u_1 + v_1 - u_2 + v_2 - u_3 + v_3 - u_4$ etc.

These two series are calculated until one side or the other runs out of pieces with which to capture on the square occupied by v_0.

Let us assume that:

$w_1 = 2$ $b_1 = 0$

$w_2 = -4$ $b_2 = 10$

$w_3 = -12$ $b_3 = 10$

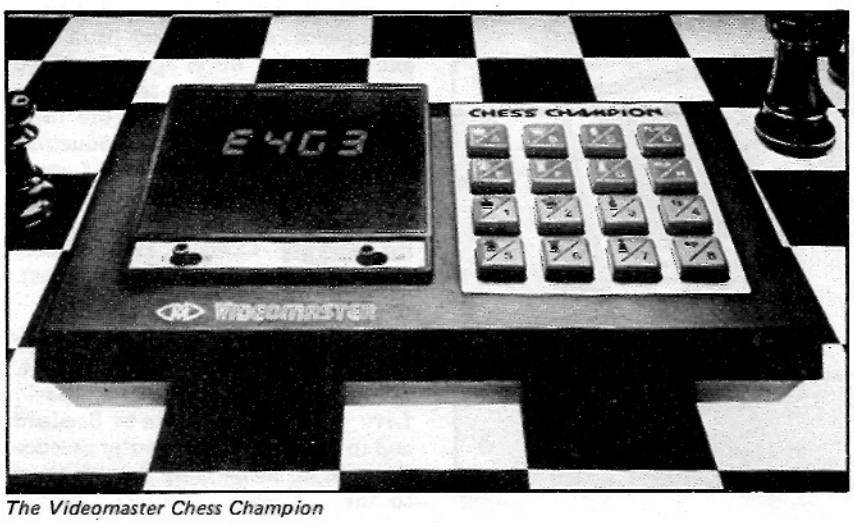
$w_4 = -10$ ($b_4 = 13$) . . . which is ignored, since the last player to capture does not lose the capturing piece.

These are placed in a one-dimensional array: 2, 0, -4, 10, -12, 10, -10.

(Values of white and black alternately)

There are two reasons for stopping the sequence. One side may not capture when it is possible to do so; or both sides capture until no more captures are possible. White may stop after the even indexed elements of the array (after Black has captured) and Black can stop after the odd-indexed elements. In order not to treat the last value as a special case a zero is added to the array and it is repeated, so both sides have the same last value: 2, 0, -4, 10, -12, 10, -10, 0.0 . . . (1).

White is trying to reach the maximum of the even-indexed elements and Black the minimum of the odd ones in array (1).



The Videomaster Chess Champion

BORIS — Chess Computer from Optimisation

Max (0, 10, 10, 0) = 10 index = 6
Min (2, -4, -12, -10, 0) = -12 index = 5

'Index is the position of max or min ement in array (1)).

Black cannot continue after -12, Therefore the sequence after -12 must be pruned off, leaving the new array:

2, 0, -4, 10, -12, -12
(this last value is a copy)

White now tries to find a new maximum and Black tries to find a new minimum. The process is repeated until White's maximum = Black's minimum. Here:

Max (0, 10, -12) = 10 index = 4
Min (2, -4, -12) = -12 index = 5

New array is 2, 0, -4, 10, 10 (copied)
'ax (0, 10) = 10 index = 4
Min (2, -4, 10) = -4 index = 3

ew array is 2, 0, -4, -4 (copied)
Max (0, -4) = 0 index = 2
Min (2, -4) = -4 index = 3

ew array is 2, 0, 0 (copied)
Max (0) = 0
Min (2,0) = 0. Finally we have:

Maximum = Minimum = value of exchanging sequence = 0.

Therefore white neither profits nor loses by making a capture on that square.

Swapoff values can be used to determine whether or not it is safe to move a piece to a particular (vacant) square. Simply make the first element of the array 0 (and remember that the piece which moves to this square no longer attacks it).

The purpose of using swap-off is to look out (without looking ahead!) for possibilities of gaining material by successively capturing on a particular square. The method used does not take into account pins, but I feel that it is still an extremely useful tool. A chess program which detected all elementary exchanging and capturing situations would already be a stronger player than 50% of the world's chess playing population.

Move generation should be carefully thought out. When 64-bit processors are readily available, move generation and position evaluation will become easier to code and quicker to execute (thank God chess is not played on a 9 x 9 board). Speed in move generation is almost as crucial as speed in position evaluation so any time that you invest in optimising these routines will be usefully spent.

The Three Phases of Chess

Chess is conventionally divided into the opening, the middle-game and the endgame. In the opening both sides seek to develop their pieces on sensible squares. A vast amount has been written about the openings but programmers should be wary of storing dozens or hundreds of opening moves as they will be useless against an opponent who chooses to follow a different variation. It is far better to teach your program some basic principles of opening play, such as encouraging it to develop its pieces and to castle.

The middle-game is usually witness to most of the cut and thrust fighting that goes on on the chess board. A good tactical analyzer is very useful here, so try to grow your trees in an intelligent way so that most CPU time is spent in looking at the critical variations. Use obvious heuristics to cut down the growth of your tree; for example, do not spend much time looking at moves that allow the opponent an immediate gain of material.

The endgame has always been and will always be the hardest part of chess to program. It is the part of the game that best sorts out the masters from the lesser mortals. Read the final two pages of Reuben Fine's "Basic Chess Endings" (Bell and Hyman) and try to implement some of his rules. In particular, your program should know about advancing passed pawns — an extra queen never did anybody any harm.

Since you are obviously interested in computer chess you should join the International Computer Chess Association. This organization publishes a news bulletin a few times each year and it will help you to keep up with latest trends. To join ICCA send $5 (US) to: *Professor B. Mittman*, Vogelback Computer Center, 2129 Sheridan Road, Evanston, I1, USA 60201.

Good luck in your programming, and if you write the first program that wins a match against me you will collect the $5,000 Levy/OMNI prize.

P.S. I regret that pressure of work prevents me from dealing with correspondence on this subject.

PCW *David Levy is setting up a company to develop intelligent software for sophisticated microprocessor applications. Any readers interested in writing assembler programs on a contract basis please write to David Levy, Box 123, Personal Computer World, 62a Westbourne Grove, London W2.*

Anyone actively interested in speech recognition, music composition or robotics is also invited to apply. **PCW**