Allen Newell
J. C. Shaw
H. A. Simon

Chess-Playing Programs and the Problem of Complexity

Abstract: This paper traces the development of digital computer programs that play chess. The work of Shannon, Turing, the Los Alamos group, Bernstein, and the authors is treated in turn. The efforts to program chess provide an indication of current progress in understanding and constructing complex and intelligent mechanisms.

Man can solve problems without knowing how he solves them. This simple fact sets the conditions for all attempts to rationalize and understand human decision making and problem solving. Let us simply assume that it is good to know how to do mechanically what man can do naturally—both to add to man's knowledge of man, and to add to his kit of tools for controlling and manipulating his environment. We shall try to assess recent progress in understanding and mechanizing man's intellectual attainments by considering a single line of attack—the attempts to construct digital computer programs that play chess.

Chess is the intellectual game *par excellence*. Without a chance device to obscure the contest, it pits two intellects against each other in a situation so complex that neither can hope to understand it completely, but sufficiently amenable to analysis that each can hope to outthink his opponent. The game is sufficiently deep and subtle in its implications to have supported the rise of professional players, and to have allowed a deepening analysis through 200 years of intensive study and play without becoming exhausted or barren. Such characteristics mark chess as a natural arena for attempts at mechanization. If one could devise a successful chess machine, one would seem to have penetrated to the core of human intellectual endeavor.

The history of chess programs is an example of the attempt to conceive and cope with complex mechanisms. Now there might have been a trick — one might have discovered something that was as the wheel to the human leg: a device quite different from humans in its methods, but supremely effective in its way, and perhaps very simple. Such a device might play excellent chess, but would fail to further our understanding of human intellectual processes. Such a prize, of course, would be worthy of discovery in its own right, but there appears to be nothing of this sort in sight.

We return to the original orientation: Humans play chess, and when they do they engage in behavior that seems extremely complex, intricate, and successful. Consider, for example, a scrap of a player's (White's) running comment as he analyzes the position in Fig. 1:

"... Are there any other threats? Black also has a threat of Knight to Bishop 5 threatening the Queen, and also putting more pressure on the King's side because his Queen's Bishop can come over after he moves his Knight at Queen 2; however, that is not the immediate threat. Otherwise, his Pawn at King 4 is threatening my Pawn ..."

Notice that his analysis is qualitative and functional. He wanders from one feature to another, accumulating various bits of information that will be available from time to time throughout the rest of the analysis. He makes evaluations in terms of pressures and immediacies of threat, and gradually creates order out of the situation.

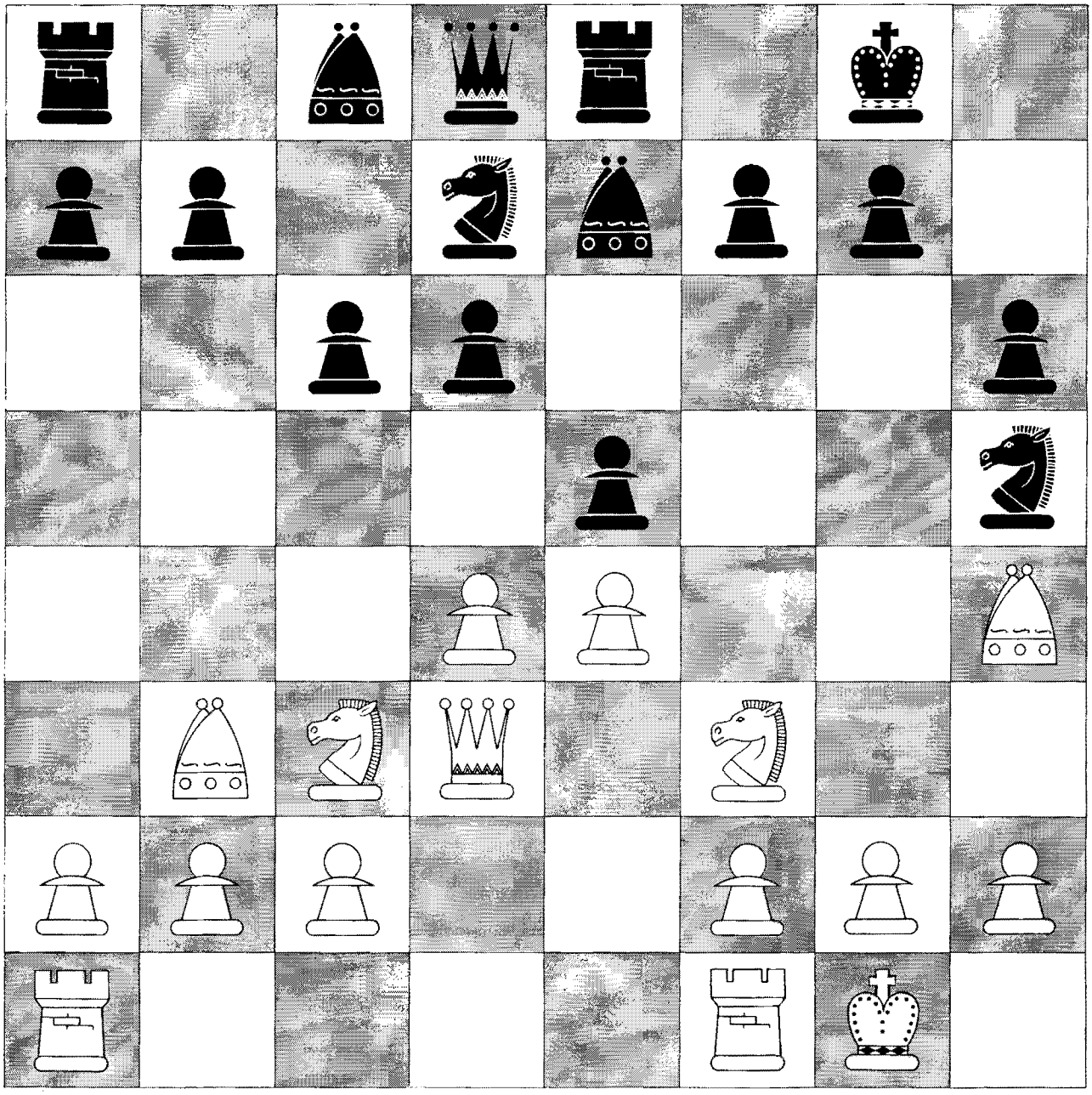
How can we construct mechanisms that will show comparable complexity in their behavior? They need not play in exactly the same way; close simulation of the human is not the immediate issue. But we do assert that complexity of behavior is essential to an intelligent per-

formance—that the complexity of a successful chess program will approach the complexity of the thought processes of a successful human chess player. Complexity of response is dictated by the task, not by idiosyncrasies of the human response mechanism.

There is a close and reciprocal relation between complexity and communication. On the one hand, the complexity of the systems we can specify depends on the language in which we must specify them. Being human, we have only limited capacities for processing information. Given a more powerful language, we can specify greater complexity with limited processing powers.

Let us illustrate this side of the relation between complexity and communication. No one considers building chess machines in the literal sense—fashioning pieces of electronic gear into automatons that will play chess. We think instead of chess programs: specifications written in a language, called machine code, that will instruct a digital computer of standard design how to play chess. There is a reason for choosing this latter course — in addition to any aversion we may have to constructing a large piece of special-purpose machinery. Machine code is a more powerful language than the block diagrams of the electronics engineer. Each symbol in machine code

Figure 1



specifies a larger unit of processing than a symbol in the block diagram. Even a moderately complicated program becomes hopelessly complex if thought of in terms of gates and pulses.

But there is another side to the relation between communication and complexity. We cannot use any old language we please. We must be understood by the person or machine to whom we are communicating. English will not do to specify chess programs because there are no English-understanding computers. A specification in English is a specification to another human who then has the task of creating the machine. Machine code is an advance precisely because there are machines that understand it—because a chess program in machine code is operationally equivalent to a machine that plays chess.

If the machine could understand even more powerful languages, we could use these to write chess programs—and thus get more complex and intelligent programs from our limited human processing capacity. But communication is limited by the intelligence of the least participant, and at present a computer has only passive capability. The language it understands is one of simple commands—it must be told very much about what to do.

Thus it seems that the rise of effective communication between man and computer will coincide with the rise in the intelligence of the computer—so that the human can say more while thinking less. But at this point in history, the only way we can obtain more intelligent machines is to design them—we cannot yet grow them, or breed them, or train them by the blind procedures that work with humans. We are caught at the wrong equilibrium of a bistable system: we could design more intelligent machines if we could communicate to them better; we could communicate to them better if they were more intelligent. Limited both in our capabilities for design and communication, every advance in either separately requires a momentous effort. Each success, however, allows a corresponding effort on the other side to reach a little further. At some point the reaction will "go," and we will find ourselves at the favorable equilibrium point of the system, possessing mechanisms that are both highly intelligent and communicative.

With this view of the task and its setting, we can turn to the substance of the paper: the development of chess programs. We will proceed historically, since this arrangement of the material will show most clearly what progress is being made in obtaining systems of increasing complexity and intelligence.

Shannon's Proposal

The relevant history begins with a paper by Claude Shannon in 1949.[11] He did not present a particular chess program, but discussed many of the basic problems involved. The framework he introduced has guided most of the subsequent analysis of the problem.

As Shannon observed, chess is a finite game. There is only a finite number of positions, each of which admits a finite number of alternative moves. The rules of chess assure that any play will terminate: that eventually a

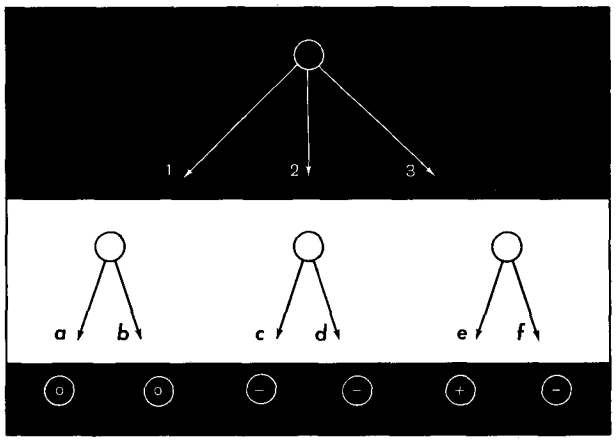


Figure 2 **The game tree and minimaxing.**

position will be reached that is a win, loss, or draw. Thus chess can be completely described as a branching tree (as in Fig. 2), the nodes corresponding to positions and the branches corresponding to the alternative moves from each position. It is intuitively clear, and easily proved, that for a player who can view the entire tree and see all the ultimate consequences of each alternative, chess becomes a simple game. Starting with the terminal positions, which have determinate payoffs, he can work backwards, determining at each node which branch is best for him or his opponent as the case may be, until he arrives at the alternative for his next move.

This inferential procedure—called *minimaxing* in the theory of games—is basic to all the attempts so far to program computers for chess. Let us be sure we understand it. Figure 2 shows a situation where White is to move and has three choices, (1), (2), and (3). White's move will be followed by Black's: (a) or (b) in case move (1) is made; (c) or (d) if move (2) is made; and (e) or (f) if move (3) is made. To keep the example simple, we have assumed that all of Black's moves lead to positions with known payoffs: (+) meaning a win for White, (0) meaning a draw, and (−) meaning a loss for White. How should White decide what to do—what inference procedure allows him to determine which of the three moves is to be preferred? Clearly, no matter what Black does, move (1) leads to a draw. Similarly, no matter what Black does, move (2) leads to a loss for White. White should clearly prefer move (1) to move (2). But what about move (3)? It offers the possibility of a win, but also contains the possibility of a loss; and furthermore, the outcome is in Black's control. If White is willing to impute any analytic ability to his opponent, he must conclude that move (3) will end as a loss for White, and hence that move (1) is the preferred move. The win from move (3) is completely insubstantial, since it can never be realized. Thus White can impute a value to a position—in this case draw—by reasoning backwards from known values.

To repeat: If the entire tree can be scanned, the best move can be determined simply by the minimaxing pro-

cedure. Now minimaxing might have been the "wheel" of chess—with the adventure ended almost before it had started—if the tree were not so large that even current computers can discover only the minutest fraction of it in years of computing. Shannon's estimate, for instance, is that there are something like 10^{120} continuations to be explored, with less than 10^{16} microseconds available in a century to explore them.

Shannon then suggested the following framework. Playing chess consists of considering the alternative moves, obtaining some effective evaluation of them by means of analysis, and choosing the preferred alternative on the basis of the evaluation. The analysis—which is the hard part—could be factored into three parts. First, one would explore the continuations to a certain depth. Second, since it is clear that the explorations cannot be deep enough to reach terminal positions, one would evaluate the positions reached at the end of each exploration in terms of the pattern of men on the chess board. These static evaluations would then be combined by means of the minimaxing procedure to form the effective value of the alternative. One would then choose the move with the highest effective value. The rationale behind this factorization was the reasonableness that, for a given evaluation function, the greater the depth of analysis, the better the chess that would be played. In the limit, of course, such a process would play perfect chess by finding terminal positions for all continuations. Thus a metric was provided that measured all programs along the single dimension of their depth of analysis.

To complete the scheme, a procedure was needed to evaluate positions statically — that is, without making further moves. Shannon proposed a numerical measure formed by summing, with weights, a number of factors or scores that could be computed for any position. These scores would correspond to the various features that chess experts assert are important. This approach gains plausibility from the existence of a few natural quantities in chess, such as the values of pieces, and the mobility of men. It also gains plausibility, of course, from the general use in science and engineering of linearizing assumptions as first approximations.

To summarize: the basic framework introduced by Shannon for thinking about chess programs consists of a series of questions:

1. Alternatives
   Which alternative moves are to be considered?
2. Analysis
   a) Which continuations are to be explored and to what depth?
   b) How are positions to be evaluated statically—in terms of their patterns?
   c) How are the static evaluations to be integrated into a single value for an alternative?
3. Final choice procedure
   What procedure is to be used to select the final preferred move?

We would hazard that Shannon's paper is chiefly remembered for the specific answers he proposed to these questions: consider all alternatives; search all continuations to fixed depth, *n;* evaluate with a numerical sum; minimax to get the effective value for an alternative; and then pick the best one. His article goes beyond these specifics, however, and discusses the possibility of selecting only a small number of alternatives and continuations. It also discusses the possibility of analysis in terms of the functions that chess men perform—blocking, attacking, defending. At this stage, however, it was possible to think of chess programs only in terms of extremely systematic procedures. Shannon's specific proposals have gradually been realized in actual programs, whereas the rest of his discussion has been largely ignored. And when proposals for more complex computations enter the research picture again, it is through a different route.

Turing's Program

Shannon did not present a particular program. His specifications still require large amounts of computing for even such modest depths of analysis as two or three moves. It remained for A. M. Turing[3] to describe a program along these lines that was sufficiently simple to be simulated by hand, without the aid of a digital computer.

In Table 1 we have characterized Turing's program in terms of the framework just defined. There are some additional categories which will become clear as we proceed. The Table also provides similar information for each of the other three programs we will consider.

Turing's program considered all alternatives—that is, all legal moves. In order to limit computation, however, he was very circumspect about the continuations the program considered. Turing introduced the notion of a "dead" position: one that *in* some sense was stable, hence could be evaluated. For example, there is no sense in counting material on the board in the middle of an exchange of Queens: one should explore the continuations until the exchange has been carried through — to the point where the material is not going to change with the next move. So Turing's program evaluated material at dead positions only. He made the value of material dominant in his static evaluation, so that a decision problem remained only if minimaxing revealed several alternatives that were equal in material. In these cases, he applied a supplementary additive evaluation to the positions reached by making the alternative moves. This evaluation included a large number of factors—mobility, backward pawns, defense of men, and so on—points being assigned for each.

Thus Turing's program is a good instance of a chess-playing system as envisaged by Shannon, although a small-scale one in terms of computational requirements. Only one published game, as far as we know, was played with the program. It proved to be rather weak, for it lost against a weak human player (who did not know the program, by the way), although it was not entirely a pushover. In general its play was rather aimless, and it was capable of gross blunders, one of which cost it the game. As one might have expected, the subtleties of the evaluation function were lost upon it. Most of the numer-

ous factors included in the function rarely had any influence on the move chosen. In summary: Turing's program was not a very good chess player, but it reached the bottom rung of the human ladder.

There is no *a priori* objection to hand simulation of a program, although experience has shown that it is almost always inexact for programs of this complexity. For example, there is an error in Turing's play of his program, because he—the human simulator—was unwilling to consider all the alternatives. He failed to explore the ones he "knew" would be eliminated anyway, and was wrong once. The main objection to hand simulation is the amount of effort required to do it. The computer is really the enabling condition for exploring the behavior of a complex program. One cannot even realize the potentialities of the Shannon scheme without programming it for a computer.

The Los Alamos Program

In 1956 a group at Los Alamos programmed MANIAC I to play chess.[5] The Los Alamos program is an almost perfect example of the type of system specified by Shannon. As shown in the Table, all alternatives were considered; all continuations were explored to a depth of two moves (i.e., two moves for Black and two for White); the static evaluation function consisted of a sum of material and mobility measures; the values were integrated by a minimax procedure,* and the best alternative in terms of the effective value was chosen for the move.

In order to carry out the computation within reasonable time limits, a major concession was required. Instead of the normal chess board of eight squares by eight squares, they used a reduced board, six squares by six squares. They eliminated the Bishops and all special chess moves: castling, two-square Pawn moves in the opening, and *en passant* captures.

The result? Again the program is a weak player, but now one that is capable of beating a weak human player, as the machine demonstrated in one of its three games. It is capable of serious blunders, a common characteristic, also, of weak human play.

Since this is our first example of actual play on a computer, it is worth looking a bit at the programming and machine problems. In a normal 8×8 game of chess there are about 30 legal alternatives at each move, on the average, thus looking two moves ahead brings 30^4 continuations, about 800,000, into consideration. In the reduced 6×6 game, the designers estimate the average number of alternatives at about 20, giving a total of about 160,000 continuations per move. Even with this reduction of five to one, there are still a lot of positions to be looked at. By comparison, the best evidence suggests that a human player considers considerably less than 100 positions in the analysis of a move.[4] The Los Alamos program was able to make a move in about 12 minutes on the average. To do this the code had to be very simple and straightforward. This can be seen by the size of the program—only 600 words. In a sense, the machine barely glanced at each position it evaluated. The two measures in the evaluation function are obtained directly from the process of looking at continuations: changes in material are noticed if the moves are captures, and the mobility score for a position is equal to the number of new positions to which it leads—hence is computed almost without effort when exploring all continuations.

The Los Alamos program tests the limits of simplification in the direction of minimizing the amount of information required for each position evaluated, just as Turing's program tests the limits in the direction of minimizing the amount of exploration of continuations. These programs, especially the Los Alamos one, provide real anchor points. They show that, with very little in the way of complexity, we have at least entered the arena of human play—we can beat a beginner.

Bernstein's Program

Over the last two years Alex Bernstein, a chess player and programmer at IBM, has constructed a chess-playing program for the IBM 704 (for the full 8×8 board).[1,2] This program has been in partial operation for the last six months, and has now played one full game plus a number of shorter sequences. It, too, is in the Shannon tradition, but it takes an extremely important step in the direction of greater sophistication: only a fraction of the legal alternatives and continuations are considered. There is a series of subroutines, which we can call plausible move generators, that propose the moves to be considered. Each of these generators is related to some feature of the game: King safety, development, defending own men, attacking opponent's men, and so on. The program considers at most seven alternatives, which are obtained by operating the generators in priority order, the most important first, until the seven are accumulated.

The program explores continuations two moves ahead, just as the Los Alamos program did. However, it uses the plausible move generators at each stage, so that, at most, 7 direct continuations are considered from any given position. For its evaluation function it uses the ratio of two sums, one for White and one for Black. Each sum consists of four weighted factors: material, King defense, area control, and mobility. The program minimaxes and chooses the alternative with the greatest effective value.

The program's play is uneven. Blind spots occur that are very striking; on the other hand it sometimes plays very well for a series of moves. It has never beaten anyone, as far as we know; in the one full game it played it was beaten by a good player,[1] and it has never been pitted against weak players to establish how good it is.

Bernstein's program gives us our first information about radical selectivity, in move generation and analysis. At 7 moves per position, it examines only 2,500 final positions two moves deep, out of about 800,000 legal continuations. That it still plays at all tolerably with a reduction in search by a factor of 300 implies that the selection mechanism is fairly effective. Of course, the

*The minimax procedure was a slight modification of the one described earlier, in that the mobility score for each of the intermediate positions was added in.

Table 1 **Comparison of current chess programs.**

	TURING	LOS ALAMOS Kister, Stein, Ulam, Walden, Wells	BERNSTEIN Roberts, Arbuckle, Belsky	NSS Newell, Shaw, Simon
• Vital statistics				
Date	1951	1956	1957	1958
Board	8 × 8	6 × 6	8 × 8	8 × 8
Computer	Hand simulation	MANIAC-I 11,000 ops/sec	IBM 704 42,000 ops/sec	RAND JOHNNIAC 20,000 ops/sec
• Chess program				
Alternatives	All moves	All moves	7 plausible moves Sequence of move generators	Variable Sequence of move generators
Depth of analysis	Until dead (exchanges only)	All moves 2 moves deep	7 plausible moves 2 moves deep	Until dead Each goal generates moves
Static evaluation	Numerical Many factors	Numerical Material, Mobility	Numerical Material, Mobility, Area control, King defense	Non-numerical Vector of values Acceptance by goals
Integration of values	Minimax	Minimax (modified)	Minimax	Minimax
Final choice	Material dominates Otherwise, best value	Best value	Best value	1. First acceptable 2. Double function
• Programming				
Language		Machine code	Machine code	IPL-IV, interpretive
Data scheme		Single board No records	Single board Centralized tables Recompute	Single board Decentralized List structure Recompute
Time	Minutes	12 min/move	8 min/move	1-10 hrs/move (est.)
Space		600 words	7000 words	Now 6000 words, est. 16,000
• Results				
Experience	1 game	3 games (no longer exists)	2 games	0 games Some hand simulation
Description	Loses to weak player Aimless Subtleties of evaluation lost	Beats weak player Equivalent to human with 20 games experience	Passable amateur Blind spots Positional	Good in spots (opening) No aggressive goals yet

selections follow the common and tested lore of the chess world; so that the significance of the reduction lies in showing that this lore is being successfully captured in mechanism. On the other hand, such radical selection should give the program a strong proclivity to overlook moves and consequences. The selective mechanisms in Bernstein's program have none of the checks and balances that exist in human selection on the chess board. And this is what we find. For example, in one situation a Bishop was successively attacked by three Pawns, each time retreating one square to a post where the next Pawn could attack it. The program remained oblivious to this possibility since the successive Pawn pushes that attacked the Bishop were never proposed as plausible moves by the generators. But this is nothing to be unhappy about. Any particular difficulty is removable: in the case of the Bishop, by adding another move generator responsive to another feature of the board. This kind of error correction is precisely how the body of practical knowledge about chess programs and chess play will accumulate, gradually teaching us the right kinds of selectivity.

Every increase in sophistication of performance is paid for by an increase in the complexity of the program. The move generators and the components of the static evaluation require varied and diverse information about each position. This implies both more program and more computing time per position than with the Los Alamos program. From Table 1, we observe that Bernstein's program takes 7,000 words, the Los Alamos program only 600 words: a factor of about 10. As for time per position, both programs take about the same time to produce a move—8 and 12 minutes respectively. Since the increase in problem size of the 8×8 board over the 6×6 board (about 5 to 1) is approximately canceled by the increase in speed of the IBM 704 over the MANIAC (also about 5 to 1, counting the increased power of the 704 order code), we can say they would both produce moves in the same 8×8 game in the same time. Hence the increase in amount of processing per move in Bernstein's program approximately cancels the gain of 300 to 1 in selectivity that this more complex processing achieves. This is so, even though Bernstein's program is coded to attain maximum speed by the use of fixed tables, direct machine coding, and so on.

We have introduced the comparison in order to focus on computing speed versus selectivity as sources of improvement in complex programs. It is not possible, unfortunately, to compare the two programs in performance level except very crudely. We should compare an 8×8 version of the Los Alamos program with the Bernstein program, and we also need more games with each to provide reliable estimates of performance. Since the 8×8 version of the Los Alamos program will be better than the 6×6, compared to human play, let us assume for purposes of argument that the Los Alamos and Bernstein programs are roughly comparable in performance. To a rough approximation, then, we have two programs that achieve the same quality of performance with the same total effort by two different routes: the Los Alamos program by using no selectivity and being very fast, and the Bernstein program by using a large amount of selectivity and taking much more effort per position examined in order to make the selection.

The point we wish to make is that this equality is an accident: that selectivity is a very powerful device and speed a very weak device for improving the performance of complex programs. For instance, suppose both the Los Alamos and the Bernstein programs were to explore three moves deep instead of two as they now do. Then the Los Alamos program would take about 1,000 times (30^2) as long as now to make a move, whereas Bernstein's program would take about 50 times as long (7^2), the latter gaining a factor of 20 in the total computing effort required per move. The significant feature of chess is the exponential growth of positions to be considered with depth of analysis. As analysis deepens, greater computing effort per position soon pays for itself, since it slows the growth in number of positions to be considered. The comparison of the two programs at a greater depth is relevant since the natural mode of improvement of the Los Alamos program is to increase the speed enough to allow explorations three moves deep. Furthermore, attempts to introduce selectivity in the Los Alamos program will be extremely costly relative to the cost of additional selectivity in the Bernstein program.

One more calculation might be useful to emphasize the value of heuristics that eliminate branches to be explored. Suppose we had a branching tree in which our program was exploring n moves deep, and let this tree have four branches at each node. If we could double the speed of the program—that is, consider twice as many positions for the same total effort—then this improvement would let us look half a move deeper ($n+\frac{1}{2}$). If, on the other hand, we could double the selectivity—that is, only consider two of the four branches at each node, then we could look twice as deep ($2n$). It is clear that we could afford to pay an apparently high computing cost per position to achieve this selectivity.

To summarize, Bernstein's program introduces both sophistication and complication to the chess program. Although in some respects—e.g., depth of analysis—it still uses simple uniform rules, in selecting moves to be considered it introduces a set of powerful heuristics which are taken from successful chess practice, and drastically reduce the number of moves considered at each position.

Newell, Shaw, and Simon Program

Although our own work on chess started in 1955,[6] it took a prolonged vacation during a period in which we were developing programs that discover proofs for theorems in symbolic logic.[8,10] In a fundamental sense, proving theorems and playing chess involve the same problem: reasoning with heuristics that select fruitful paths of exploration in a space of possibilities that grows exponentially. The same dilemmas of speed versus selection and uniformity versus sophistication exist in both problem domains. Likewise, the programming costs attendant upon complexity seem similar for both. So we

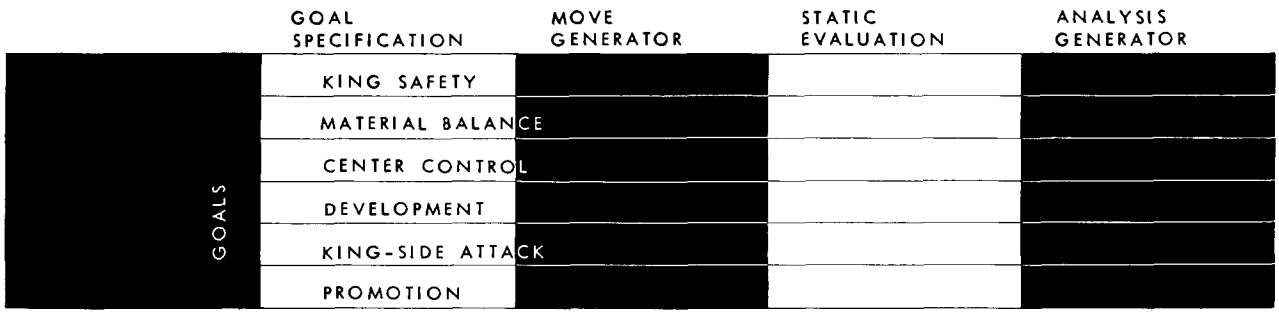


Figure 3 **Basic organization of NSS chess program.**

have recently returned to the chess programming problem equipped with ideas derived from the work on logic.

The historical antecedents of our own work are somewhat different from those of the other investigators we have mentioned. We have been primarily concerned with describing and understanding human thinking and decision processes.[9] However, both for chess players and for chess programmers, the structure of the task dictates in considerable part the approach taken, and our current program can be described in the same terms we have used for the others. Most of the positive features of the earlier programs are clearly discernible: The basic factorization introduced by Shannon; Turing's concept of a dead position; and the move generators, associated with features of the chess situation, used by Bernstein. Perhaps the only common characteristic of the other programs that is strikingly absent from ours — and from human thinking also, we believe—is the use of numerical additive evaluation functions to compare alternatives.

• *Basic organization*

Figure 3 shows the two-way classification in terms of which the program is organized. There is a set of goals, each of which corresponds to some feature of the chess situation—King safety, material balance, center control, and so on. Each goal has associated with it a collection of processes, corresponding to the categories outlined by Shannon: a move generator, a static evaluation routine, and a move generator for analysis. The routine for integrating the static evaluations into an effective value for a proposed move, and the final choice procedure are both common routines for the whole program, and therefore are not present in each separate component.

• *Goals*

The goals form a basic set of modules out of which the program is constructed. The goals are independent: any of them can be added to the program or removed without affecting the feasibility of the remaining goals. At the beginning of each move a preliminary analysis establishes that a given chess situation (a "state") obtains, and this chess situation evokes a set of goals appropriate to it. The goal specification routines shown for each goal in Fig. 3 provide information that is used in this initial selection of goals. The goals are put on a list with the most crucial ones first. This goal list then controls the remainder of the processing: the selection of alternatives, the continuations to be explored, the static evaluation, and the final choice procedure.

What kind of game the program will play clearly depends on what goals are available to it and chosen by it for any particular move. One purpose of this modular construction is to provide flexibility over the course of the game in the kinds of considerations the program spends its effort upon. For example, the goal of denying stalemate to the opponent is relevant only in certain endgame situations where the opponent is on the defensive and the King is in a constrained position. Another purpose of the modular construction is to give us a flexible tool for investigating chess programs—so that entirely new considerations can be added to an already complex but operational program.

• *Move generation*

The move generator associated with each goal proposes alternative moves relevant to that goal. These move generators carry the burden of finding positive reasons for doing things. Thus, only the center-control generator will propose P-Q4 as a good move in the opening; only the material-balance generator will propose moving out of danger a piece that is *en prise*. These move generators correspond to the move generators in Bernstein's program, except that here they are used exclusively to generate alternative moves and are not used to generate the continuations that are explored in the course of analyzing a move. In Bernstein's program—and *a fortiori* in the Los Alamos program—identical generators are used both to find a set of alternative moves from which the final choice of next move is made, and also to find the continuations that must be explored to assess the consequences of reaching a given position. In our program the latter function is performed by a separate set of analysis generators.

• *Evaluation*

Each move proposed by a move generator is assigned a value by an analysis procedure. We said above that the move generators have the responsibility for finding positive reasons for making moves. Correspondingly, the analysis procedure is concerned only with the acceptability of a move once it has been generated. A generator proposes; the analysis procedure disposes.

The value assigned to a move is obtained from a series of evaluations, one for each goal. The value is a vector, if you like to think of it that way, except that it does not necessarily have the same components throughout the chess game, since the components derive from the basic list of goals that is constructed from the position at the beginning of each move. Each component expresses acceptability or unacceptability of a position from the viewpoint of the goal corresponding to that component. Thus, the material-balance goal would assess only the loss or gain of material; the development goal, the relative gain or loss of *tempi;* the Pawn structure goal, the doubling and isolation of Pawns; and so on. The value for a component is in some cases a number—e.g., in the material-balance goal where we use conventional piece values: 9 for a Queen, 5 for a Rook, and so on. In other cases the component value is dichotomous, simply designating the presence or absence of some property, like the blocking of a move or the doubling of a Pawn.

As in the other chess programs, our analysis procedure consists of three parts: exploring continuations to some depth, forming static evaluations, and integrating these to establish an effective value for the move. By a process that we will describe later, the analysis move generators associated with the goals determine what branches will be explored from each position reached. At the final position of each continuation, a value is assigned using the static evaluation routines of each goal to provide the component values. The effective value for a proposed move is obtained by minimaxing on these final static values. Minimaxing seems especially appropriate for an analysis procedure that is inherently conservative, such as an acceptance test.

To be able to minimax, it must be possible to compare any two values and decide which is preferable, or whether they are equal in value. For values of the kind we are using, there must be a complete ordering on the vectors that determine them. Further, this ordering must allow variation in the size and composition of the goal list. We use a lexicographic ordering: Each component value is completely ordered within itself; and higher priority values completely dominate lower priority values, as determined by the order of goals on the goal list. To compare two values, then, the first components are compared. If one of these is preferable to the other, this determines the preference for the entire value. If the two components are equal, then the second pair of components is compared. If these are unequal in value, they determine the preference for the entire value, otherwise the next components are compared, and so on.

• *Final choice*

It is still necessary to select the move to be played from the alternative moves, given the values assigned to them by the analysis procedure. In the other programs the final choice procedure was simply an extension of the minimax: choose the one with highest value. Its obviousness rests on the assumption that the set of alternatives to be considered is a fixed set. If this assumption is relaxed, by generating alternatives sequentially, then other procedures are possible. The simplest, and the one we are currently using, is to set an acceptance level as final criterion and simply take the first acceptable move. The executive routine proceeds down the goal list, activating the move generators of the goals in order of priority, so that important moves are considered first. The executive saves the best move that has been found up to any given moment, and if no moves reach the specified level of acceptability, it makes the best move that was found.

Another possible final choice procedure is to search for an acceptable move that has a double function—that is, a move that is proposed by more than one generator as having a positive effect. With this plan, the executive proceeds down the list of goals in order of priority. After finding an acceptable move, it activates the rest of the generators to see if the move will be proposed a second time. If not, it works from the list of unevaluated moves just obtained to see if any move proposed twice is acceptable. If not, it takes the first acceptable move or the best if none has proved acceptable. This type of executive has considerable plausibility, since the concept of multiple-function plays an important role in the chess literature.

Yet a third variation in the final choice procedure is to divide the goals into two lists. The first list contains all the features that should normally be attended to; the second list contains features that are rare in occurrence but either very good or very bad if they do occur. On this second list would be goals that relate to sacrificial combinations, hidden forks or pins that are two moves away, and so on. The executive finds an acceptable move with the first, normal list. Then the rest of the available time is spent looking for various rare consequences derived from the second list.

• *Analysis*

In describing the basic organization of the program we skipped over the detailed mechanism for exploring continuations, simply assuming that certain continuations were explored, the static values computed, and the effective value obtained by minimaxing. But it is clear that the exact mechanisms are very important. The analysis move generators are the main agents of selectivity in the program: They determine for each position arrived at in the analysis just which further branches must be explored, hence the average number of branches in the exploration tree and its average depth. The move generators for the alternatives and the final choice procedure also affect the amount of exploration by determining what moves are considered. But their selection operates only once per move, whereas the selectivity of the analysis generators operates at each step (half-move) of the exploration. Hence the selectivity of the analysis generators varies geometrically with the average depth of analysis.

The exploration of continuations is based on a generalization of Turing's concept of a dead position. Recall that Turing applied this notion to exchanges, arguing that it made no sense to count material on the board until all exchanges that were to take place had been carried out.

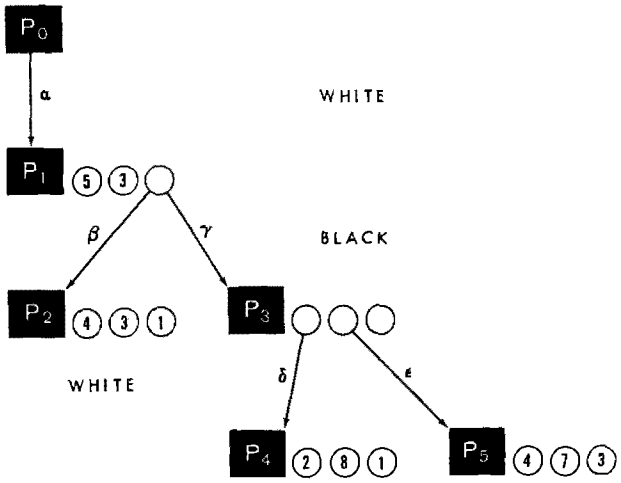


Figure 4 **Analysis.**

We apply the same notion to each feature of the board: The static evaluation of a goal is meaningful only if the position being evaluated is "dead" with respect to the feature associated with that goal — that is, only if no moves are likely to be made that could radically alter that component static value. The analysis-move generators for each goal determine for any position they are applied to whether the position is dead with respect to their goal; if not, they generate the moves that are both plausible and might seriously affect the static value of the goal. Thus the selection of continuations to be explored is dictated by the search for a position that is dead with respect to all the goals, so that, finally, a static evaluation can be made. Both the number of branches from each position and the depth of the exploration are controlled in this way. Placid situations will produce search trees containing only a handful of positions; complicated middle game situations will produce much larger ones.

To make the mechanics of the analysis clearer, Fig. 4 gives a schematic example of a situation. P_0 is the initial position from which White, the machine, must make a move. The arrow, α, leading to P_1 represents an alternative proposed by some move generator. The move is made internally (i.e., "considered"), yielding position P_1, and the analysis procedure must then obtain the value of P_1, which will become the value imputed to the proposed alternative, α. Taking each goal from the goal list in turn, an attempt is made to produce a static evaluation. For P_1 this attempt is successful for the first and second components, yielding values of 5 and 3 respectively. (Numbers are used for values throughout this example to keep the picture simple; in reality, various sets of ordered symbols are used, their exact structure depending on the nature of the computation.) However, the third component does not find the position dead, and generates two moves, β and γ. The first, β, is considered, leading to P_2, and an attempt is made to produce a static evaluation of it. This proceeds just as with P_1, except that this time all components find the position dead and the static value (4, 3, 1) is obtained. Then the second move, γ, from P_1 is considered, leading to P_3. The attempt to produce a static value for P_3 runs into difficulties with the first component, which generates one move, δ, to resolve the instability of P_3 with respect to its feature. This move leads to P_4 which is evaluable, having the value (2, 8, 1). However, the second component also finds P_3 not dead and generates a single move, ε, leading to P_5. This is also evaluable, having the value (4, 7, 3). The third component finds P_3 dead and therefore contributes no additional moves. Thus the exploration comes to an end with all terminal positions yielding complete static values. Since it is White's move at P_3, White will choose the move with the highest value. This is ε, the move to P_5, with a value of (4, 7, 3) (the first component dominates). The value of this move is the effective value assigned to P_3. Black now has a choice between the move, β, to P_2, yielding (4, 3, 1) and the move, γ, to P_3, yielding (4, 7, 3). Since Black is minimizing, he will choose β. This yields (4, 3, 1) as the effective value of the alternative, α, that leads to P_1, and the end of the analysis.

The minimaxing operation is conducted concurrently with the generation of branches. Thus if P_5, which has a value of (4, 7, 3), had been generated prior to P_4 no further moves would have been generated from P_3, since it is already apparent that Black will prefer P_2 to P_3. The value of P_3 is at least as great as the value of P_5, since it is White's move and he will maximize.

This analysis procedure is not a simple one, either conceptually or technically. There are a number of possible ways to terminate search and reach an effective evaluation. There is no built-in rule that guarantees that the search will converge; the success depends heavily on the ability to evaluate statically. The more numerous the situations that can be recognized as having a certain value without having to generate continuations, the more rapidly the search will terminate. The number of plausible moves that affect the value is also of consequence, as we discussed in connection with Bernstein's program, but there are limits beyond which this cannot be reduced. For example, suppose that a position is not dead with respect to Material Balance and that one of the machine's pieces is attacked. Then it can try to (a) take the attacker, (b) add a defender, (c) move the attacked piece, (d) pin the defender, (e) interpose a man between the attacker and the attacked, or (f) launch a counter-attack. Alternatives of each of these types must be sought and tried—they are all plausible and may radically affect the material balance.

As an example of the heuristics involved in achieving a static evaluation, imagine that the above situation occurred after several moves of an exploration, and that the machine was already a Pawn down from the early part of the continuation. Then, being on the defensive implies a very remote chance of recovering the Pawn. Consequently, a negative value of at least a Pawn can be assigned to the position statically. This is usually enough in connection with concurrent minimaxing to eliminate the continuation from further consideration.

• *Summary*

Let us summarize our entire program. It is organized in terms of a set of goals: these are conceptual units of chess —King safety, passed Pawns, and so on. Each goal has several routines associated with it:

1. A routine that specifies the goal in terms of the given position;
2. A move generator that finds moves positively related to carrying out the goal;
3. A procedure for making a static evaluation of any position with respect to the goal, which essentially measures acceptability;
4. An analysis move generator that finds the continuations required to resolve a situation into dead positions.

The alternative moves come from the move generators, considered in the order of priority of their respective goals. Each move, when it is generated, is subjected to an analysis. This analysis generates an exploration of the continuations following from the move until dead positions are reached and static evaluations computed for them. The static evaluations are compared, using minimax as an inference procedure, so that an effective value is eventually produced for each alternative. The final choice procedure can rest on any of several criteria: for instance, choosing the first move generated that has an effective value greater than a given norm.

• *Examples of goals*

In this section we will give two examples of goals and their various components to illustrate the type of program we are constructing. The first example is the center-control goal:

Center control

Specification

Goal is always operative unless there are no more center Pawns to be moved to the fourth rank.

Move generator

1. Move P-Q4, P-K4 (primary moves).
2. Prevent the opponent from making his primary moves.
3. Prepare your own primary moves:
   a) Add a defender to Q4 or K4 square.
   b) Eliminate a block to moving QP or KP.

Static evaluation

Count the number of blocks to making the primary moves.

Analysis move generators

None; static evaluation is always possible.

To interpret this a little: Goals are proposed in terms of the general situation—e.g., for the opening game. The list of goals is made up for a position by applying, in turn, the specification of each of the potential goals. Whether any particular goal is declared relevant or irrelevant to the position depends on whether or not the position meets its specification. For Center Control, no special information need be gathered, but the goal is declared irrelevant if the center Pawns have already been moved to the fourth rank or beyond.

The most important part of the center-control program is its move generator. The generator is concerned with two primary moves: P-Q4 and P-K4. It will propose these moves, if they are legal, and it is the responsibility of the analysis procedures (for all the goals) to reject the moves if there is anything wrong with them—e.g., if the Pawns will be taken when moved. So, after 1. P-Q4, P-Q4, the center-control move generator will propose 2. P-K4, but (as we shall see) the evaluation routine of the material balance goal will reject this move because of the loss of material that would result from 2. . . . , PxP. The center-control generator will have nothing to do with tracing out these consequences.

If the primary moves cannot be made, the center-control move generator has two choices: to prepare them, or to prevent the opponent from making his primary moves. The program's style of play will depend very much on whether prevention has priority over preparation (as it does in our description of the generator above), or vice versa. The ordering we have proposed, which puts prevention first, probably produces more aggressive and slightly better opening play than the reverse ordering. Similarly, the style of play depends on whether the Queen's Pawn or the King's Pawn is considered first.

The move generator approaches the subgoal of preventing the opponent's primary moves (whenever this subgoal is evoked) in the following way. It first determines whether the opponent can make one of these moves by trying the move and then obtaining an evaluation of it from the opponent's viewpoint. If one or both of the primary moves are not rejected, preventive moves will serve some purpose. Under these conditions, the center-control move generator will generate them by finding moves that bring another attacker to bear on the opponent's K4 and Q4 squares or that pin a defender of one of these squares. Among the moves this generator will normally propose are N-B3 and BP-B4.

The move generator approaches the subgoal of preparing its own primary moves by first determining why the moves cannot be made without preparation—that is, whether the Pawn is blocked from moving by a friendly piece, or whether the fourth rank square is unsafe for the Pawn. In the former case, the generator proposes moves for the blocking piece; in the latter case, it finds moves that will add defenders to the fourth rank square, drive away or pin attackers, and so on.

So much for the center-control move generators. The task of the evaluation routine for the center-control goal is essentially negative—to assure that moves, proposed by some other goal, will not be made that jeopardize control of the center. The possibility is simply ignored that a move generator for some other goal will inadvertently provide a move that contributes to center control. Hence, the static evaluation for Center Control is only concerned that moves not be made that interfere with P-K4 and P-Q4. A typical example of a move that the center-control evaluation routine is prepared to reject is B-Q3 or B-K3 before the respective center Pawns have been moved.

The second example of a goal is Material Balance. This is a much more extensive and complicated goal than Center Control, and handles all questions about gain and loss of material in the immediate situation. It does not consider threats like pins and forks, where the actual exchange is still a move away; other goals must take care of these. Both the negative and positive aspects of material must be included in a single goal, since they compensate directly for each other, and material must often be spent to gain material.

Material balance

Specification

A list of exchanges on squares occupied by own men, and a list of exchanges on squares occupied by opponent's men. For each exchange square there is listed the target man, the list of attackers, and the list of defenders (including, e.g., both Rooks if they are doubled on the appropriate rank or file). For each exchange square a static exchange value is computed by playing out the exchange with all the attackers and defenders assuming no indirect consequences like pins, discovered attacks, etc. Exchange squares are listed in order of static exchange value, largest negative value first. Squares with positive values for the defender are dropped from the list. At the same time a list of all pinned men is generated.

Move generator

Starting with the exchange squares at the top of the list, appropriate moves are generated. If the most important exchange square is occupied by the opponent, captures by attacking pieces are proposed, the least valuable attacker being tried first. If the move is rejected because the attacker is pinned, the next attacker is tried. If the move is rejected for another reason, the possibility of exchange on this square is abandoned, and the next exchange square examined.

If the exchange square under examination is occupied by the program's own piece, a whole series of possible moves is generated:

a) Try "no move" to see if attack is damaging.
b) Capture the attacker.
c) Add a defender not employed in another defense.
d) Move the attacked piece.
e) Interpose a man between the attacker and the target; but not a man employed elsewhere, and not if the interposer will be captured.
f) Pin the attacker with a man not employed elsewhere and not capturable by the attacker.

Static evaluation

For each exchange square, add the values of own men and subtract the values of opponent's men. Use conventional values: Q-9, R-5, B-N-3, P-1.

Move generators toward dead positions

A position is dead for this goal only if there are no exchanges —that is, if the specification list defined above is empty. Then a static evaluation can be made. Otherwise, the various kinds of moves defined under the move generator are made to resolve the exchanges. However, various additional qualifications are introduced to reduce the number of continuations examined. For example, if in a particular exchange material has already been lost and a man is still under attack, the position is treated as dead, since it is unlikely that the loss will be recovered. When a dead position is reached, the static evaluation is used to find a value for the position.

It is impossible to provide here more than a sketchy picture of the heuristics contained in this one goal. It should be obvious from this brief description that there are a lot of them, and that they incorporate a number of implicit assumptions about what is important, and what isn't, on the chess board.

• *Performance of the program*

We cannot say very much about the behavior of the program. It was coded this spring and is not yet fully debugged. Only two goals have been coded: Material Balance and Center Control. Development is fully defined as well as a Pawn structure goal sufficient for the opening, where its role is primarily to prevent undesirable structures like doubled Pawns. These four goals—Material Balance, Center Control, Development and Pawn Structure—in this order seem an appropriate set for the first phase of the opening game. Several others—King safety, Serious Threats, and Gambits—need to be added for full opening play. The serious threats goal could be limited initially to forks and pins.

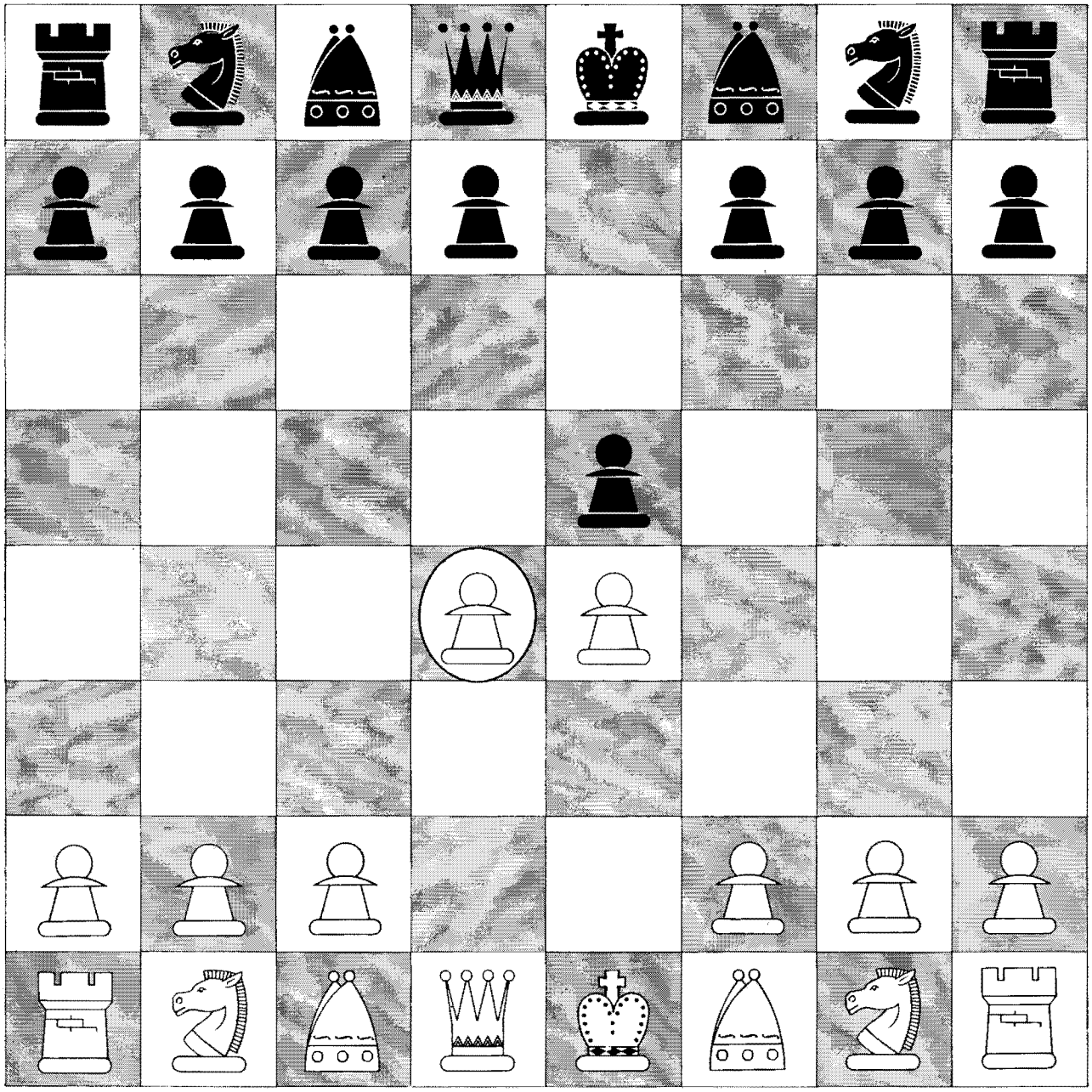
We have done considerable hand simulation with the program in typical positions. Two examples will show how the goals interact. In Fig. 5 the machine is White and the play has been 1. P-K4, P-K4. Assuming the goal list mentioned above, the material balance move generator will not propose any moves since there are no exchanges on the board. The center-control generator will propose P-Q4, which is the circled move in the figure. (In the illustration, we assume the center-control move generator has the order of the primary moves reversed from the order described earlier.) This move is rejected —as it should be—and it is instructive to see why. The move is proposed for analysis. Material Balance does not find the position dead, since there is an exchange, and generates Black's move, 2. . . . . , PxP. The resulting position is still not dead, and 3. QxP, is generated. The position is now dead for Material Balance, with no gain or loss in material. The first component of the static evaluation is "even." There are obviously no blocks to Pawn moves, so that the center control static value is acceptable. However, the third component, Development, finds the position not dead because there is now an exposed piece, the Queen. It generates replies that both attack the piece and develop—i.e., add a tempo. The move 3. . . . . , N-QB3 is generated. This forces a Queen move, resulting in loss of a tempo for White. Hence Development rejects the move, 2. P-Q4. (The move 3. . . . . , B-B4 would not have sufficed for rejection by Development, since the Bishop could be taken.)

The second example, shown in Fig. 6, is from a famous game of Morphy against Duke Karl of Brunswick and Count Isouard. Play had proceeded 1. P-K4, P-K4; 2. N-KB3, P-Q3; 3. P-Q4. Suppose the machine is Black in this position. The move 3. . . . , B-N5 is proposed by Material Balance to deal with the exchange that threatens Black with the loss of a Pawn. This is the move made by the Duke and Count. The analysis proceeds by 4. PxP, PxP. This opens up a new exchange possibility with the Queens, which is tried: 5. QxQ, KxQ; 6. NxP. Thus the Pawn is lost in this continuation. Hence, alternative moves

are considered at Black's nearest option, which is move 4, since there are no alternative ways of recapturing the Queen at move 5. The capture of White's Knight is possible, so we get: 4′. . . . , BxN; 5′. PxB, PxP; 6′. QxQ, KxQ. This position is rejected by Development since the forced King move loses Black his castling privilege, and this loss affects the tempo count. This is a sufficient reason to reject the move 3. . . . , B-N5, without even examining the stronger continuation, 5″. QxB, that Morphy as White chose. In our program, 5. PxB is generated before 5. QxB. Either reply shows that 3. . . . , B-N5 is unsound.

One purpose of these examples is to illustrate a heuristic for constructing chess programs that we incline to rather strongly. We wish not only to have the program make good moves, but to do so for the right reasons. The chess commentary above is not untypical of human analysis. It also represents rather closely the analysis made by the program. We think this is sound design philosophy in constructing complex programs. To take another example: the four-goal opening program will not make sacrifices, and conversely, will always accept gambits. The existing program is unable to balance material against

Figure 5

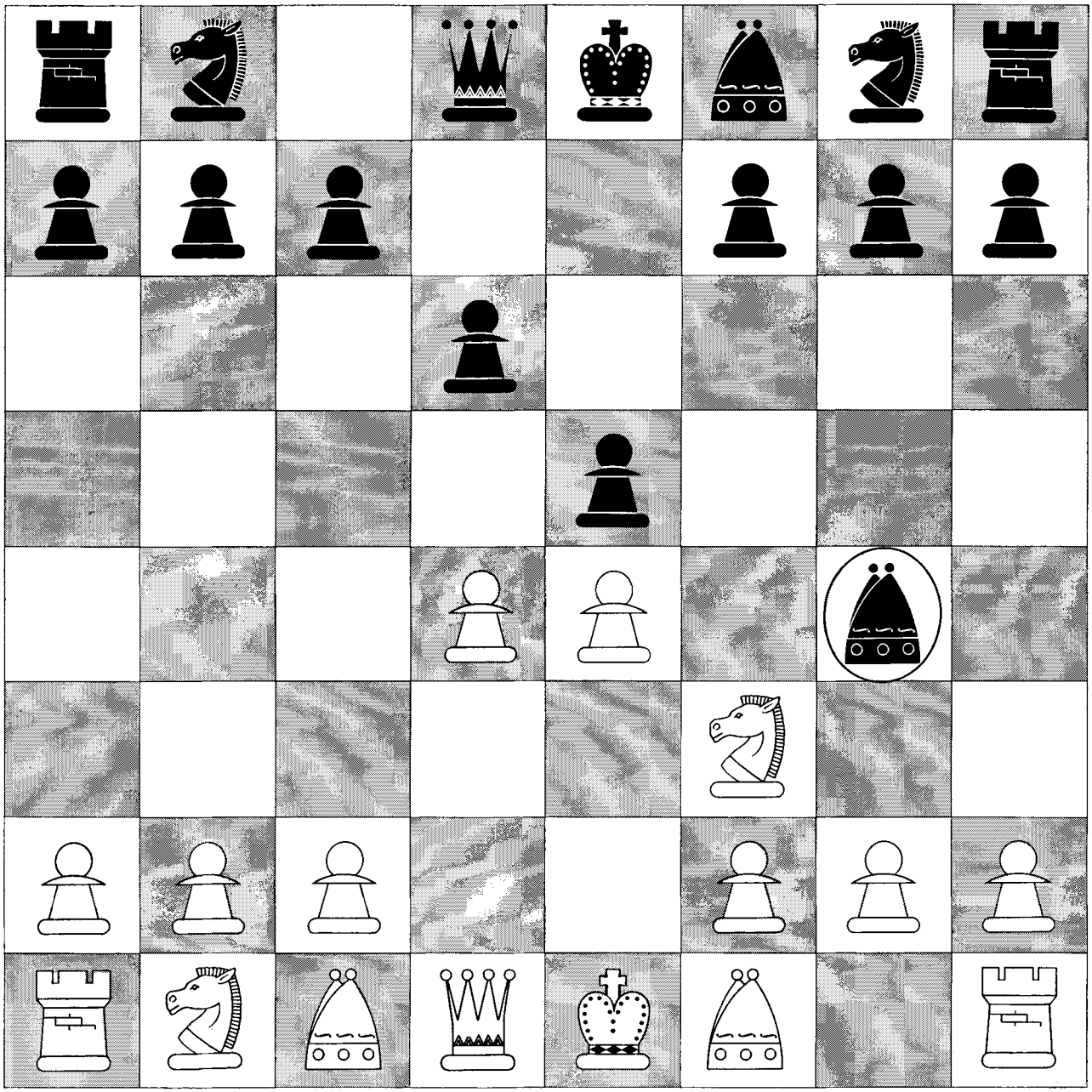


positional advantage. The way to make the program take account of sacrifices is to introduce an additional goal having to do with them explicitly. The corresponding heuristic for a human chess player is: don't make sacrifices until you understand what a sacrifice is. Stated in still another way, part of the success of human play depends on the emergence of appropriate concepts. One major theme in chess history, for example, is the emergence of the concept of the center and the notion of what it means to control the center. One should not expect the equivalent of such a concept simply to emerge from computation based on quite other features of the position.

• *Programming*

The program we have been describing is extremely complicated. Almost all elements of the original framework put forward by Shannon, which were handled initially by simply uniform rules, have been made variable, and dependent on rather complicated considerations. Many special and highly particular heuristics are used to select moves and decide on evaluations. The program can be expected to be much larger, more intricate, and to require

Figure 6



much more processing per position considered than even the Bernstein program.

In the introduction to this paper we remarked on the close connection between complexity and communication. Processes as complex as the Los Alamos program are unthinkable without languages like current machine codes in which to specify them. The Bernstein program is already a very complicated program in machine code; it involved a great deal of coding effort and parts of it required very sophisticated coding techniques. Our own program is already beyond the reach of direct machine coding: It requires a more powerful language.

In connection with the work on theorem-proving programs we have been developing a series of languages, called information processing languages (IPL's).[7] The current chess program is coded in one of them, IPL-IV. An information processing language is an interpretive pseudo-code—that is, there exists a program in JOHNNIAC machine code that is capable of interpreting a program in IPL and executing it. When operating, JOHNNIAC contains both the machine code and the IPL code.

It is not possible to give in this paper a description of IPL-IV or of the programming techniques involved in constructing the chess program. Basically IPL is designed to manipulate lists, and to allow extremely complicated structures of lists to be built up during the execution of a program without incurring intolerable problems of memory assignment and program planning. It allows unlimited hierarchies of sub-routines to be easily defined, and permits recursive definition of routines. As it stands —that is, prior to coding a particular problem—it is independent of subject matter (although biased towards list manipulation in the same sense that algebraic compilers are biased towards numerical evaluation of algebraic expressions). To code chess, a complete "chess vocabulary" is built up from definitions in IPL. This vocabulary consists of a set of processes for expressing basic concepts in chess: tests of whether a man bears on another man, or whether two men are on the same diagonal; processes for finding the direction between two men, or the first man in a given direction from another; and processes that express iterations over all men of a given type, or over all squares of a given rank. There are about 100 terms in this basic process vocabulary. The final chess program, as we have been describing it in this paper, is largely coded in terms of the chess vocabulary. Thus there are four language "levels" in the chess program: JOHNNIAC machine code, general IPL, basic chess vocabulary, and finally the chess program itself.

We can now make a rough assessment of the size and complexity of this program in comparison with the other programs. The table indicates that the program now consists of 6000 words and will probably increase to 16,000. The upper bound is dictated by the size of the JOHNNIAC drum and the fact that JOHNNIAC has no tapes. In terms of the pyramiding structure described above, this program is already much larger than Bernstein's, although it is difficult to estimate the "expansion" factor involved in converting IPL to machine code. (For one thing, it is not clear how an "equivalent" machine coded program would be organized.) However, only about 1000 words of our program are in machine code, and 3000 words are IPL programs, some of which are as many as ten definitional steps removed from machine code. Further, all 12,000 words on the drum will be IPL program; no additional data or machine code are planned.

The estimated time per move, as shown in Table 1, is from one to ten hours, although moves in very placid situations like the opening will take only a few minutes. Even taking into account the difference in speed between the 704 and JOHNNIAC, our program still appears to be at least ten times slower than Bernstein's. This gap reflects partly the mismatch between current computers and computers constructed to do efficiently the kind of information processing required in chess.[12] To use an interpretive code, such as IPL, is in essence to simulate an "IPL computer" with a current computer. A large price has to be paid in computing effort for this simulation over and above the computing effort for the chess program itself. However, this gap also reflects the difficulty of specifying complex processes; we have not been able to write these programs and attend closely to the efficiency issue at the same time.

On both counts we have felt it important to explore the kind of languages and programming techniques appropriate to the task of specifying complex programs, and to ignore for the time being the costs we were incurring.

Conclusion

We have now completed our survey of attempts to program computers to play chess. There is clearly evident in this succession of efforts a steady development toward the use of more and more complex programs and more and more selective heuristics; and toward the use of principles of play similar to those used by human players. Partly, this trend represents—at least in our case—a deliberate attempt to simulate human thought processes. In even larger part, however, it reflects the constraints that the task itself imposes upon any information processing system that undertakes to perform it. We believe that any information processing system—a human, a computer, or any other—that plays chess successfully will use heuristics generically similar to those used by humans.

We are not unmindful of the radical differences between men and machines at the level of componentry. Rather, we are arguing that for tasks that could not be performed at all without very great selectivity—and chess is certainly one of these—the main goal of the program must be to achieve this selection. The higher-level programs involved in accomplishing this will look very much the same whatever processes are going on at more microscopic levels. Nor are we saying that programs will not be adapted to the powerful features of the computing systems that are used—e.g., the high speed and precision of current digital computers, which seems to favor exploring substantial numbers of continuations. However, none of the differences known to us—in speed, memory, and so

on—affect the essential nature of the task: search in a space of exponentially growing possibilities. Hence the adaptations to the idiosyncrasies of particular computers will all be secondary in importance, although they will certainly exist and may be worth while.

The complexity of heuristic programs requires a more powerful language for communicating with the computer than the language of elementary machine instructions. We have seen that this necessity has already mothered the creation of new information processing languages. But even with these powerful interpretive languages, communication with the machine is difficult and cumbersome. The next step that must be taken is to write programs that will give computers a problem-solving ability in understanding and interpreting instructions that is commensurable with their problem-solving ability in playing chess and proving theorems.

The interpreter that will transform the machine into an adequate student for a human instructor will not be a passive, algorithmic translator — as even the most advanced interpreters and compilers are today — but an active, complex, heuristic problem-solving program. As our explorations of heuristic programs for chess playing and other tasks teach us how to build such an interpreter, they will at last enable us to make the transition from the low-level equilibrium at which man-machine communication now rests to the high-level equilibrium that is certainly attainable.

References and footnotes

1. A. Bernstein and M. deV. Roberts, "Computer vs. Chess-player," *Scientific American,* **198,** 6, June 1958.
2. A. Bernstein, *et al.,* "A Chess-Playing Program for the IBM 704," *Proceedings of the 1958 Western Joint Computer Conference,* May 1958.
3. B. V. Bowden, *Faster Than Thought,* Chapter 25, Pitman, 1953.
4. A. D. De Groot, *Het Denken van den Schaker,* Amsterdam, 1946.
5. J. Kister, *et al.,* "Experiments in Chess," *J. Assoc. for Computing Machinery,* **4,** 2, April 1957.
   There are two other explorations between 1951 and 1956 of which we are aware—a hand simulation by F. Mosteller and a Russian program for BESM. Unfortunately, not enough information is available on either to talk about them, so we must leave a gap in the history between 1951 and 1956.
6. A. Newell, "The Chess Machine," *Proceedings of the 1955 Western Joint Computer Conference,* March 1955.
7. A. Newell and J. C. Shaw, "Programming the Logic Theory Machine," *Proceedings of the 1957 Western Joint Computer Conference,* February 1957.
8. A. Newell, J. C. Shaw, and H. A. Simon, "Empirical Explorations of the Logic Theory Machine, *Proceedings of the 1957 Western Joint Computer Conference,* February 1957.
9. A. Newell, J. C. Shaw, and H. A. Simon, "Elements of a Theory of Human Problem Solving," *Psych. Rev.,* **65,** May 1958.
10. A. Newell and H. A. Simon, "The Logic Theory Machine," *Transactions on Information Theory,* **IT-2,** No. 3, September 1956.
11. C. E. Shannon, "Programming a Computer for Playing Chess," *Phil. Mag.,* **41,** March 1950.
12. J. C. Shaw, *et al.,* "A Command Structure for Complex Information Processing," *Proceedings of the 1958 Western Joint Computer Conference,* May 1958.

Received May 27, 1958