The first version of a mating program described here—MATER I—is the work of G. Baylord, H. Simon and P. Simon, using the Newell-Prasad chessboard. The program was revised during 1964 by Baylor into a second version, MATER II.

3. ORGANIZATION OF THE ANALYSIS TREE

The Problem

As stated in Section 1 the mating program analyzes chess positions. An analysis of a position—as the term is used here*—consists of the set of moves and evaluations made in the course of resolving the choice-of-move problem. Taken together, moves and positions make up a tree of possibilities in which moves operate on positions to produce new positions (see Fig. 1) and on which evaluations of positions and of moves in achieving desired positions can be hung as desired.

The dots or nodes in Fig. 1 denote positions (static states) and the lines between dots denote moves (operators) that transform one position into another.†

*Chess players would probably prefer to define “analysis” as the finished product rather than the process of search, laying stress on the “right” moves and continuations rather than emphasizing how these were arrived at.

†Simon and Newell have often drawn this difference equation analogy to the problem-solving process: given an initial state description and a desired state, the problem is to find a process description that operates on the initial state to produce the desired one. In discussion of the Logic Theorist, for example, the logic expressions correspond to the static states, and the rules of inference, to the operators.

How should the analysis of a position be conducted? Figure 2 presents one simple scheme. Would this scheme be workable if it were made operational? For example, let:

1. “X” be defined as checkmate and the program be given the capability of generating moves in its service;
2. the criteria for deciding among moves be specified by certain rules of selection;
3. the program be given the capability of making moves and updating board positions;
4. a test be provided so that the program “knew” if it had achieved “X”; and
5–8. the corresponding provisions be made for “Y”, defined as escaping check, and for choosing among replies.

The answer is no, not quite. The scheme lacks a means for recovering from false starts, for retracing its steps when it runs into blind alleys. Indeed, what is lacking can be seen by considering the difference between actually playing a game of chess and analyzing a chess position: the course of analysis is fickle and reversible, whereas in an actual game a move once made cannot be unmade. In other words, the scheme outlined above needs provisions for unmaking moves and for abandoning seemingly unpromising positions as much as it needs the capability of making moves and pursuing promising positions. Ideally, one should like to be able to
enter and reenter the move tree at any node (position) at any time and from there to proceed down any branch, old or new. Providing such capabilities for reinstating the right position at the right time is probably the central problem of organization at this level of the program, while making operational and making sense out of steps 1–8 above is probably the central problem at the next higher conceptual level. This section reports on the former problem: implementation of a flexible move tree. Section 4 is devoted to the latter: defining the problem and the heuristics of search.

**Building the Tree**

The notion of analysis as a tree search is misleading to the extent that it implies that each step consists solely of selecting a move from the many available alternatives. Actually, the process is more one of generating moves as one goes along, of building one’s own tree. This is the very distinction Maier* (p. 218) has drawn between decision making under conditions of uncertainty* and problem solving: “Decision making implies a given number of alternatives, whereas in problem solving the alternatives must be created. Thus, problem solving involves both choice behavior and the finding or creating of alternatives.”

In every chess position, of course, the rules of the game place an upper limit on the number of possibilities that can be created; de Groot* found that, averaged across the course of a game, the mean number of legal move possibilities lies somewhere between 30 and 35. This is a full-grown tree; the one the searcher actually builds is much smaller: on the average, four or five branches at the top node and smaller thereafter.

The first question addressed in this section is the technical one: How does one build a tree? How are limbs and nodes—moves and positions, respectively—structured into a tree of possibilities? (See the Appendix for the details of construction of moves and positions.) Second, and pursuing the metaphor, think of the chess player climbing the tree as he builds it. Crawling along a branch in one direction corresponds to making a move in the current position (node) while traversing it in the opposite direction corresponds to unmaking a move and restoring the previous position (node). This ability to back up the tree is what enables a player to abandon unpromising lines of investigation and start afresh. In starting afresh, moreover, the player may either reinvestigate branches he has previously built or build new ones.

Third, as the player builds and climbs he also accrues and retains information. The information garnered en route and the use to which it is put are in large part what Denkpsychologists have called the development of the problem. That is, the searcher’s conception of the problem at any one time consists of the information he has about the problem, how he has evaluated this information, and even how it has shaped his definition of what the problem is (cf. Duncker* and de Groot*). Provisions for gathering information are considered in both this section and the next; the use to which information is put and the matter of problem development are more properly treated in the subsequent sections.

Most of the organizational problems are solved via the description lists of moves. For convenience of reference the entire list of possible attributes a move can take on is set forth in Table 1.
With respect to the first question—provisions for holding the tree together—a signal cell and attributes A46 and A47 do the job. The signal cell contains the name of the most recent move made on the board—the contemporary—while attributes A46 and A47 are its ancestor and its list of descendants, respectively. The log of the analysis is preserved by defining a dummy move, L31, which has on its description list the list of descendants attribute A47. Thus the course of analysis is linked together as a chain of moves with contemporaries linked by ancestor and descendant relations, as in the example of an analysis tree in Fig. 3.

Second, because of the strong family ties just described, one can eventually crawl one's way down any branch of any node and then back up. That is to say, one can make or unmake any move. Two routines, E65 and E66, make and unmake moves, respectively. The procedure for unmaking a move is exactly the reverse. With the help of routine E11 the position list is restored, that is, the description lists of the pieces and squares affected are restored and the signal cells reset.

At any node a new limb may also be constructed (by routines E51 and E52; see Appendix) simply by specifying the "from square" and the "to square" (and special move status, if any) whereupon the move is added to the list of descendants, V(A47), and assigned an ancestor, V(A46).

Third, information gathered in the search for mate is stored on the description list of the move that gathers it. (See Table 1.) When a move is constructed, its ancestor is always assigned as a value of A46 and the man moved in always assigned as a value of A51. Conditionally, a move is assigned a value of A43 if a Rook or King is removed from the castle list, a value of A44 if a man is captured, a value of A45 if a Pawn is captured en passant, a value of A42 if it is a double check, a value of A43 if it is a discovered check, and a value of A44 if it is a checking move at all.

Evaluative information is also gathered: attribute A49 records the win-loss value of a move: mate, no mate, or no value. If a checking move has no descendants, that move mates; consequently, attributes A70, A71, and A72 record the kinds of replies to check. Attribute A47 lists the descendants in toto and the value of A50 is a count of them.

The point here is to illustrate how information is hung on the move tree as it is gathered. How the information is retrieved and utilized is a topic for the next section.

Table 1. Move Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A40</td>
<td>From square</td>
</tr>
<tr>
<td>A41</td>
<td>To square</td>
</tr>
<tr>
<td>A42</td>
<td>Special move (yes or no)</td>
</tr>
<tr>
<td>A43</td>
<td>Man removed from castle list</td>
</tr>
<tr>
<td>A44</td>
<td>Man captured</td>
</tr>
<tr>
<td>A45</td>
<td>Square of captured en passant Pawn</td>
</tr>
<tr>
<td>A46</td>
<td>Ancestor</td>
</tr>
<tr>
<td>A47</td>
<td>List of descendants</td>
</tr>
<tr>
<td>A48</td>
<td>Irreversibility of move (reversible; or irreversible)</td>
</tr>
<tr>
<td>A49</td>
<td>Value of move (mate, no mate, no value)</td>
</tr>
<tr>
<td>A50</td>
<td>Number of descendants</td>
</tr>
<tr>
<td>A51</td>
<td>Man moved</td>
</tr>
<tr>
<td>A52</td>
<td>Double check (yes or no)</td>
</tr>
<tr>
<td>A53</td>
<td>Discovered check (yes or no)</td>
</tr>
<tr>
<td>A54</td>
<td>Checking move (yes or no)</td>
</tr>
<tr>
<td>A55</td>
<td>Descendant's list of mate threats</td>
</tr>
<tr>
<td>A56</td>
<td>Threatened mating square</td>
</tr>
<tr>
<td>A58</td>
<td>Mating piece on V(A56) square</td>
</tr>
<tr>
<td>A63</td>
<td>Reply NOMV (no move)</td>
</tr>
<tr>
<td>A65</td>
<td>Number of checking moves generated to date</td>
</tr>
<tr>
<td>A66</td>
<td>Number of replies generated to date</td>
</tr>
<tr>
<td>A70</td>
<td>List of King replies</td>
</tr>
<tr>
<td>A71</td>
<td>List of capturing replies</td>
</tr>
<tr>
<td>A72</td>
<td>List of interposing replies</td>
</tr>
</tbody>
</table>

respectively. The log of the analysis is preserved by defining a dummy move, L31, which has on its description list the list of descendants attribute A47. Thus the course of analysis is linked together as a chain of moves with contemporaries linked by ancestor and descendant relations, as in the example of an analysis tree in Fig. 3.
4. THE EXECUTIVE AND HEURISTICS OF SEARCH

Introduction

In a given position, what moves should be considered and in what order? Human chess players are known to be highly selective in the moves they look at, a selectivity based on their heuristics of search. Computers must also incorporate such selectivity. What follows then is a discussion of the search heuristics incorporated into the early version of the mating program, some measures on its search behavior, a brief description of the routines that effect the move generation, and, finally, later developments added to a second version of the program, MATER II.

MATER I

Restricting the mobility of the opponent's pieces is a recognized principle of chess strategy. It is particularly important in checkmating combinations since checkmate is defined as an unopposed attack on an enemy King whose mobility has been reduced to zero. Strategically this means the attacker strives to gain control over 1) the square the enemy King occupies, as well as 2) all the squares contiguous to it that do not contain an enemy piece. If just condition (1) obtains, the enemy King is simply in check; if just condition (2) holds, the enemy King is stale-mated; while if both (1) and (2) hold, he is checkmated. Viewed in this light, checkmate is a process of acquiring controls, or more and more restricting the enemy King's mobility. This principle is the cornerstone of the mating program.

The restriction of mobility principle applies to the generation and selection of moves as well as to decisions about when to abandon search in certain directions. Thus in the mating program: Only checking moves (in MATER I) and moves that threaten mate in one move (in MATER II) are generated for the attacker; the move selected for investigation is the one that most restricts the opponent's mobility; and search is continued down a chosen path only so long as the opponent's mobility is on the decline: when the number of replies at some node in the current line of investigation is equal to or greater than the number of replies at some prior node, the current line is abandoned, the prior node restored, and the alternative that had once been passed over is tried. This nips in the bud unpromising proliferations in the move tree.

In addition to the notions just described—rate of growth serving to terminate search in a particular direction, the set of considerable moves serving to restrict the set of applicable operators in the given position, the try-list ordered by the fewest-replies heuristic serving to stipulate the application of offensive operators—some heuristics of chess strategy serving to stipulate the order of application of defensive operators can also be seen more clearly from the following illustrative position taken from Fine and MATER I's performance on it.

The following layout is adapted to the simple recursive scheme of Fig. 2.

---

Since only one move can be tried out at a time in any particular position, other "eligible" checking moves must wait their turn on a list, L35—the try-list. This list has two noteworthy properties: first of all, it is an ordered list, and second it is independent of a move's level.* Such independence proves powerful in directing search.

The list is ordered by a fewest-replies heuristic: highest priority goes to moves with the fewest number of legal replies, while checking moves with more than four legal replies are discarded entirely. Ties are broken by giving priority to double checks, then to checks that have no capturing responses, then to the order in which the checks were generated. The second property—that checks from all levels are mixed—effects the evaluative principle that search is continued down a particular path only so long as the opponent's mobility is on the decline: when the number of replies at some node in the current line of investigation is equal to or greater than the number of replies at some prior node, the current line is abandoned, the prior node restored, and the alternative that had once been passed over is tried. This nips in the bud unpromising proliferations in the move tree.

In addition to the notions just described—rate of growth serving to terminate search in a particular direction, the set of considerable moves serving to restrict the set of applicable operators in the given position, the try-list ordered by the fewest-replies heuristic serving to stipulate the application of offensive operators—some heuristics of chess strategy serving to stipulate the order of application of defensive operators can also be seen more clearly from the following illustrative position taken from Fine and MATER I's performance on it.

The following layout is adapted to the simple recursive scheme of Fig. 2.

---

*The level of a move refers to its depth in the move tree, i.e., how many moves out it is from the starting position.

†They can also be seen more clearly within the picture of heuristic search in general in Newell and Ernst.
(1) Generate checking moves: 1.N-K6ch; B-K7ch; B-B7ch; Q-B6ch.

(2) Select one move for further analysis:
   (a) 1.N-K6ch 1.B-K7ch 1.B-B7ch 1.Q-B6ch
       1...K-K1 1...NxQ
       1...NxB 1...NxB
       1...QPxN
       1...B-PxN 1
       1...NxQ 2
       1...N-K2

   (b) Transfer checks to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.B-B7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis.

(4) Test if checkmate has been achieved: No.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select best reply: 1...NxQ.

(7) Make that reply (1...NxQ) in the analysis.

(8) Test if check has been relieved: Yes.


(2) Select one move for further analysis.
   (a) For each check generate and count replies:
       2.N-K6ch
       2...K-K1
       2...QPxN
       2...B-PxN

   (b) Transfer check to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis.

(4) Test if checkmate has been achieved: Yes.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select next best reply: 1...N-K2.

(7) Make that reply (1...N-K2) in the analysis, restoring the board to the appropriate position.

(8) Test if check has been relieved: Yes.

(1) Generate checking moves: 2.N-K6ch; BxNch; B-B7ch; Q-Nch.

(2) Select one move for further analysis.
   (a) For each check generate and count replies:
       2...NxQ
       2...B-PxN
       2...QPxN

   (b) Transfer checks to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.B-B7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3
       2.BxNch 0
       2.B-B7ch 0
       2.Q-Nch 0
       2...QPxN
       2...B-PxN

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis, restoring the board to the initial position.

(4) Test if checkmate has been achieved: Yes.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select best reply: 1...NxQ.

(7) Test if check has been relieved: Yes.

(1) Generate checking moves: 2.N-K6ch; BxNch; B-B7ch; Q-Nch.

(2) Select one move for further analysis.
   (a) For each check generate and count replies:
       2.K-K1 0
       2...NxQ
       2...B-PxN
       2...QPxN

   (b) Transfer checks to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.B-B7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3
       2.BxNch 0
       2.B-B7ch 0
       2.B-Qch 1
       2...B-PxN

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis, restoring the board to the initial position.

(4) Test if checkmate has been achieved: Yes.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select best reply: 1...NxQ.

(7) Test if check has been relieved: Yes.

(1) Generate checking moves: 2.N-K6ch; BxNch; B-B7ch; Q-Nch.

(2) Select one move for further analysis.
   (a) For each check generate and count replies:
       2.K-K1 0
       2...NxQ
       2...B-PxN
       2...QPxN

   (b) Transfer checks to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.B-B7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3
       2.BxNch 0
       2.B-B7ch 0
       2.B-Qch 1
       2...B-PxN

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis, restoring the board to the initial position.

(4) Test if checkmate has been achieved: Yes.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select best reply: 1...NxQ.

(7) Test if check has been relieved: Yes.

(1) Generate checking moves: 2.N-K6ch; BxNch; B-B7ch; Q-Nch.

(2) Select one move for further analysis.
   (a) For each check generate and count replies:
       2.K-K1 0
       2...NxQ
       2...B-PxN
       2...QPxN

   (b) Transfer checks to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.B-B7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3
       2.BxNch 0
       2.B-B7ch 0
       2.B-Qch 1
       2...B-PxN

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis, restoring the board to the initial position.

(4) Test if checkmate has been achieved: Yes.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select best reply: 1...NxQ.

(7) Test if check has been relieved: Yes.

(1) Generate checking moves: 2.N-K6ch; BxNch; B-B7ch; Q-Nch.

(2) Select one move for further analysis.
   (a) For each check generate and count replies:
       2.K-K1 0
       2...NxQ
       2...B-PxN
       2...QPxN

   (b) Transfer checks to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.B-B7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3
       2.BxNch 0
       2.B-B7ch 0
       2.B-Qch 1
       2...B-PxN

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis, restoring the board to the initial position.

(4) Test if checkmate has been achieved: Yes.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select best reply: 1...NxQ.

(7) Test if check has been relieved: Yes.

(1) Generate checking moves: 2.N-K6ch; BxNch; B-B7ch; Q-Nch.

(2) Select one move for further analysis.
   (a) For each check generate and count replies:
       2.K-K1 0
       2...NxQ
       2...B-PxN
       2...QPxN

   (b) Transfer checks to try-list; order them by their “u” values:
       L35 1
       1.B-K7ch 1
       1.B-B7ch 1
       1.Q-B6ch 2
       1.N-K6ch 3
       2.BxNch 0
       2.B-B7ch 0
       2.B-Qch 1
       2...B-PxN

   (c) Select and delete top move from try-list L35: 1.B-K7ch.

(3) Make that move (1.B-K7ch) in the analysis, restoring the board to the initial position.

(4) Test if checkmate has been achieved: Yes.

(5) Generate replies to relieve check: 1...NxQ; N-K2.

(6) Select best reply: 1...NxQ.

(7) Test if check has been relieved: Yes.
A CHESS MATING COMBINATIONS PROGRAM

Diagram 36

1/6/10/15

Figure 4. MATER I's analysis tree of Diagram 36, Fine 4.
(The u's are a tally of the number of replies by which the priority of moves on the try-list is established. The square boxes represent positions and the numbers in them trace the course of the investigation—the order in which the positions were taken up. The crosshatched branches trace the mating path.)

This example also illustrates the criteria by which the order of application of defensive moves is accomplished: by "best" reply is meant that reply that seems most likely to give the attacker trouble. Thus the priority of defensive moves Black tries is, first, the capture of the most valuable White pieces by the least valuable Black pieces followed by King moves, interpositions, then order of generation. Again this is an attempt to clip unnecessary proliferations in the move tree: if there is a "killing" reply to a checking move, further analysis of that checking move would seem futile.*

*This is the minimax assumption; namely, that the opponent will make his strongest reply at every opportunity. McCarthy's killer heuristic (see Kotok) assumes that a killing reply to one checking move may be a killing reply to other checking moves and thus should be looked at first.

Measures of Search Behavior. Many measures of search behavior can be picked off an analysis tree like MATER I's of Diagram 36 (Fig. 4). For example, the tree can be characterized by counting the number of positions or the number of moves. Simon and Simon call the total number of positions examined the "exploration tree"; in Diagram 36, above, the position count yields an exploration tree of size 16. In general, however, more moves are seen than positions are investigated, which is to say that some moves remain unexplored, such as the replies to 1.N-K6ch in the example above. The count of moves seen—the uninvestigated as well as the investigated ones—will be called the discovery tree; in Diagram 36, above, this move tally yields a discovery tree of size 36 (14 checks and 22 responses). One further refinement can be carved out of the exploration and discovery trees; namely, the "verification tree," which Simon and Simon define as the total number of positions required to prove the combination—the positions resulting from the single best move at each node for the attacker and from every legal move at each node for the defender; respectively, the positive and negative parts of the proof schema. The verification tree "is precisely analogous to the correct path in a maze. It is a tree instead of a single path because all alternatives allowed to the defender must be tested" (Simon and Simon, p. 427). In Diagram 36, above, the
branches of the verification tree are crosshatched, yielding a position count of size 6, or alternatively, a move count of size 5.

These measures do not reveal the time order in which the tree was generated. Human chess players are fickle tree climbers, “progressive deepeners,” to use de Groot’s11 term for the phenomenon: “The investigation not only broadens itself progressively by growing new branches, counter moves, or considerable own-moves, but also literally deepens itself: the same variant is taken up anew and is calculated further than before” (de Groot,11 p. 266). In other words, the search strategy is an important structural characteristic of the thought process. In Fig. 4 the order in which positions are taken up is captured by numbering the nodes (positions) in the analysis. These measures will be used for comparative purposes in Section 5.

**Routines.** How are checking moves and replies actually generated in any given position? There would seem to be two tacks, corresponding to a one-many approach and a many-one approach. In trying to find all the checks in a given position, for example, one could either radiate out from the enemy King and from each square, search for a piece that can get there and give check (the one-many approach), or converge from the squares along the move directions of each attacking piece onto the enemy King’s square (the many-one approach). If there are many pieces on the board, the former is the more efficient; if few, the latter.

G1 is a master routine that procures all checks in a given position. It employs the many-one approach, calling subroutine G11 for Queen, Bishop, and Rook checking moves; G12 for Knight and regular Pawn checks; and G13 for double Pawn moves that administer check. Similarly, R21 procures all replies to a given checking move: R11 generates all the King moves that get out of check, R12, all the captures of the checking piece, and R13, all the interpositions. In this way the mating program is able to enumerate all checks and all replies in a particular position.

**MATER II**

In designing search programs it is useful to distinguish the strategy of search from the information that is gathered during the search. The search strategy tells where to go next, and what information must be kept so that the search can be carried out. It does not tell what other information to obtain while at the various positions, nor what to do with the information after it is obtained. There may be strong interaction between the search itself and the information found, as in the decision to stop searching, but we can often view this as occurring within the confines of a fixed search strategy.

(Newell and Simon,16 pp. 24-25)

MATER II adds a modification to MATER I’s search strategy by bypassing the fixedness in the order of application of operators inherent in the try-list. The new search rule states: in the given position pursue immediately and in depth all checking moves that keep the enemy King stalemated (or nearly so), i.e., moves that can only be answered by captures and/or interpositions or, in the absence of both, by one and only one King move (the “nearly so” condition). In addition to altering the program’s search strategy by telling it “where to go next,” this procedure also gathers information about the position. In this respect it resembles what de Groot11 has called a “sample variation,” a kind of trial balloon sent up for the express purpose of gathering information to direct subsequent investigation; in this sense it is orientative. Before turning to what information is gathered and how it is used, it should be mentioned that sometimes a sample variation pays off directly—the “sample moves” may be a path to a quick mate.

Specifically, a routine G10 conducts the preliminary search and, if no “easy mate” is found, records the sequence of moves investigated on a list B4. A routine G17 makes use of this information later in drawing up a plan of attack. Just how these routines operate can best be seen by considering in three parts MATER II’s analysis for a particular position, Diagram 97 from Fine.4

The first part has to do with the preliminary search; the second with the use to which the recorded information is put in drawing up a plan; and the third with the exploration and verification of that plan.

The first 10 moves of the discovery tree are those in Fig. 5a. (Note that both the 1.Q-N7ch and 1.QxRPch sample variations are recorded on list B4.)
A CHESS MATING COMBINATIONS PROGRAM  

Clearly, the "wishful thinking" goal of the sample variations went unfulfilled; the preliminary excursions did not yield mate. They do yield two sequences of forcing moves, however, that may be useful in constructing a plan of attack. Indeed, the routine G17 searches list B4 for the first move candidates and finds, in this example, 1.PxR and 1.N-N5. The former is rejected as illegal in the current position while the latter is deemed considerable. Note that the set of operators that may be applied in the initial position is expanded in MATER II to include moves other than just checking moves, yet the means by which these are generated continues to ensure a high degree of selectivity.

Routine G17 asks if a proposed move, in this case 1.N-N5, threatens mate in one move. It determines the answer by assuming that Black does nothing on his turn, that is, by playing a "No Move" and then seeing if White can enforce an immediate checkmate. And, indeed, 2.QxRP is mate. In other words, White leaves the actual problem space to seek a mate in a simplified planning space (see Newell, Shaw and Simon's) and, in fact, the second part of the move tree is given over to solving the problem in the planning space (Fig. 5b).*

Finally, the third stage is devoted to testing the soundness of the plan; that is, suppose Black tries to avert 1.N-N5 and 2.QxRPmate. Can he? It happens that in this position he cannot so that the exploration tree and the verification tree are identical in this stage of the analysis. (The rather lengthy third stage is omitted here.)

MATER II contains one other highly selective mechanism for finding moves that threaten mate in one. Controls exerted over the enemy King's square and the squares in this immediate vicinity are built into a list structure called the King's Sector. For a given square five kinds of control have been defined:

1. No control—the enemy King can move to the given square.
2. Attacking control—the attacker can move to or capture on the given square.
3. Occupation control—one of the attacker's pieces occupies the given square.
4. Block control—one of the defender's pieces occupies the given square.
5. X-ray control—the attacker can unmask an attacker control by removing

* A hybrid version of MATER I and II would first have re-investigated 1.Q-N7ch and 1.QxRPch, invoking the fewest-replies heuristic, transferring these two checks to the try-list, and elaborating them, before even considering moves which threaten mate in one. Unfortunately the statistics gathered on various versions of the program are too incomplete to say which search strategy is superior across positions, if in fact a correct strategy can be determined independent of position.

From the collection of the Computer History Museum (www.computerhistory.org)
one of his own pieces (corresponding to a "discovery" in chess jargon) or he could unmask an attacking control but for an enemy interposer (corresponding to a "pin").

The King's Sector, L40, is constructed by the four routines E91-E94. The complete structure of L40 and the information contained therein can best be seen in Fig. 6, the King's Sector for Diagram 70 from Fine. Attribute Y3 has data term XO as its value, a tally of the number of uncontrolled squares in the Sector (see Fig. 6).

How is all this information retrieved and used by the program? First, a routine G14 tries to generate mate-threatening moves by converting an X-ray control into a second attacking control. For example, in Diagram 70, routine G14, seizing on the White Bishop's X-ray control of KN8, proposes 1.BxBP but then rejects the move because 2.B-N8 does not administer check, let alone mate. Second, a routine G19, given one attacking control, tries to add a second. Routine G19, seeing an attacking control over KN7 in position 70, proposes to add another with the moves 1.Q-KB6, 1.Q-N6, and 1.Q-R6. Since all three produce mate if Black does nothing, all three are accepted as considerable moves in the plan.

In summary, MATER II contains several mechanisms for generating a selective set of considerable moves. Incorporating MATER I's ability to generate all checking moves and all replies in a given position, MATER II goes on to generate mate-threatening moves based either on their earlier appearance in forced sequences of checking moves or on the function they serve in controlling key squares around the enemy King. Moreover, MATER II has a set of routines, R15, R16, R22, for generating defensive replies to a threatened mate.

MATER II also contains three principal mechanisms in its search strategy specifying the order in which moves are to be considered. Defensively, in reply to checking moves, captures are preferred to King moves, which, in turn, are preferred to interpositions; while in reply to one-move mate threats, captures are preferred to moves that defend the mating square as well as to interpositions and King runs. Offensively, search is directed by pursuing particular moves in depth so long as the enemy King remains very highly constricted, and then later by pursuing the move that leaves the opponent with the fewest replies. Each of these search evaluators rests on a single criterion: sometimes a line of search is terminated because the defender is left with King moves in reply (nodes 4 and 7 in the move tree of position 97); sometimes a move is rejected because it does not produce immediate mate (nodes 11 and 12 in position 97); sometimes a move just never gets off the waiting list (node 2 in position 36); and

Figure 6. IPL-V List Structure of King's Sector Controls (arranged in attribute-value pairs) of R. Fine's Diagram 70, from a game Alekhine-Supico, 1942.
checking moves with more than four legal replies are always rejected out of hand. Indeed, it is the thesis here that these kinds of criteria, criteria based on features of the task area, are what regulate chess players’ choice-of-move decisions and form a good alternative representation to complicated weighting functions of the sort employed by Samuel in checkers and Bernstein and Roberts in chess. Even though mating combinations are the only facet of the game in which the final evaluators, MATE and NOMATE, are well defined—a degree of certainty attained nowhere else in the game—the search-directing decisions intermediate to the final choice of move must all be made on far less than certain criteria, just like the rest of the game.

Except for the sample variations recorded on the list B4, the information-gathering mechanisms rely on the description list of the move that gathered it, including the final evaluation, MATE or NOMATE, which are propagated back up the tree by the minimax inference procedure in an attempt to demonstrate the proof of a combination.

5. INTERPRETATION AND RESULTS

Introduction

With respect to the verification of simulation models in general, and problem-solving models in particular, two criteria for assessment seem to have emerged clearly: an achievement criterion and a process criterion. That is, can the model solve the class of problems it was designed to handle, and are its mechanisms for doing so equivalent to, or even comparable to, a human problem solver’s? The answer to the first question is relatively straightforward; not so to the second, however, since the requirements for equivalence or comparability of process are themselves open to question. In the present report, we shall not consider questions of human simulation but will confine ourselves to a discussion of the achievement of the programs.

MATER I solves combinations which consist of uninterrupted series of checking moves, given that the defender at no node in the verification tree has more than four legal replies; MATER II solves combinations that begin either with checks or with one-move mate threats and checking moves thereafter. This limitation on the class of moves the program can see restricts severely the class of combinations on which the model can be tested. Nevertheless, the program has been tested on material taken from Fine’s chapter on the mating attack. Solutions to one class of positions in the chapter call for an uninterrupted series of checking moves ending in mate (51/129 positions). Another class of positions is solved with one-move mate threats and checking moves thereafter (5/129 positions). In the residual class, mate can either be averted through a sacrifice of material or the mate is not “forced,” as the term was defined in Section 1 (73/129 positions).

MATER I’s Achievement

MATER I found solutions to 43 of the 51 mating positions. The machine missed one combination entirely by failing to move a Pawn that gave a discovered check and exhausted available space before finding the other seven.* Table 2 breaks these 43 positions down according to certain structural measures of search behavior: the depth of search to mate (D), the mean size of the verification tree necessary to prove the combination (VT measured in moves), and the mean size of the discovery tree generated in searching for mate (DT measured in moves). These latter two measures were defined in Section 4.

Table 2. MATER I’s Performance on 43 Positions from Fine

<table>
<thead>
<tr>
<th>N (positions)</th>
<th>D</th>
<th>VT</th>
<th>DT</th>
<th>VT/D</th>
<th>DT/D</th>
<th>VT/DT</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
<td>3.5</td>
<td>15.5</td>
<td>1.8</td>
<td>7.8</td>
<td>4.4</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>5.4</td>
<td>24.6</td>
<td>1.8</td>
<td>8.2</td>
<td>4.6</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>9.9</td>
<td>61.5</td>
<td>2.5</td>
<td>15.4</td>
<td>6.2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>11.0</td>
<td>56.0</td>
<td>2.2</td>
<td>11.2</td>
<td>5.1</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>17.0</td>
<td>108.0</td>
<td>2.1</td>
<td>13.5</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Simon and Simon (p. 428) suggest depth, number of positions in the exploration tree, and number of positions in the verification tree as measures of the difficulty of combinations. They remark that the four positions in their sample “are not ordered in the same way with respect to the different measures of difficulty.” Using depth, number of moves in the discovery tree (DT), and number of moves in the verification tree (VT) as equivalent measures, the

*In particular, MATER I failed to see 2.P-B6ch in position 148. The seven positions that exhausted available space did so because the fewest replies heuristic failed to discriminate among alternative checking moves: among six alternative discovered checks in positions 41 and 100; among a large number of initial checks available in positions 109 and 140; and among a large number of checks in depth involved in a King hunt on the open board in positions 111, 130, and 157.
data of Table 2 do show, with but one exception, the same ordinal relationship across measures, at least when averaged over the 43 positions.

Between depth and the size of the verification tree (VT/D) there is a fairly close correlation—around two moves in the verification tree per move in depth. This confirms the Simon and Simon observation: the tree varies linearly, not exponentially, with depth, and it probably is this property that makes deep analysis possible in combinations. Neither DT/D nor VT/DT shows any consistent relationship.

The only roughly constant ratio, VT/D, has more to do with the characteristics of mating positions than with characteristics of the mating program. The only measure on the program's search behavior is DT and there seems to be no consistent relationship between it and the two measures on the combinations (DT/D and VT/DT).

**MATER II's Achievement**

MATER II has been tested on all five positions from Fine that necessitated an initial threat of mate in one and checks thereafter. It solved three directly, the other two, because of a change in computer facilities, by hand simulation. The search tree for position 97 has already been described. In position 107 the initial Queen sacrifice as well as an unexpected Bishop sacrifice were easily spotted in the service of mate. The analysis tree of position 70 is given in Fig. 7. Note that the correct move and theme in position 70 derive from the celebrated game of Marshall's for which spectators showered the chessboard with gold coins!* Of the two hand

---

*The program also finds the correct sequence of moves from the immortal Lewitzky-Marshall game, played in Breslau in 1912 (Diagram 69 in Fine); it is excluded from the count here since Lewitzky, had he not chosen to resign, could have averted mate at the cost of a piece.
simulated runs position 95 required but the simple addition of a second control on the KN7 square via 1.Q-R6, while position 113 required the move 1.R-R5, which had been discovered in one of the exploratory sample variants.

**Conclusion**

In conclusion, MATER's power stems from its ability to generate a small selective set of moves that merit investigation. Since most of the earlier chess programs (see the review in Newell, Shaw and Simon and Kotok) spent their analysis time processing the wrong moves, it would seem that MATER II's two major mechanisms for generating relevant moves—its reliance on the sample variations and on the control of key squares—warrant further research. MATER II's major weakness, on the other hand, lies in its poorly organized search strategy for using its selectivity at all points in the analysis process.

On the horizon, proposals have been made for strengthening the program's perceptual capabilities as well as altering its search strategy.

**Appendix**

**THE BASIC REPRESENTATION**

**Statement of the Problem**

In this Appendix we describe the basic representation implemented by Newell and Prasad. Two interrelated questions guided the choice of representation:

1. What are the necessary components of a chess representation?
2. How should this information be organized?

In response to (1): The program should be able to "see" the same things a human sees when he looks at a chessboard. Thus the program requires an internal representation of the squares and pieces on a chessboard and the relations among them, and a set of processes that can pick off and make use of these relations as needed. The former requirement is called "setting up the chessboard," the latter "move-making and board processing capabilities."

In response to (2): The game of chess provides an inhomogeneous collection of information out of which moves must be forged. Thus there must be enough variety in the representation to discriminate all the different kinds of moves; given that, the information should be stored in such a way that little space is allotted to moves that seldom occur (such as Pawn promotions, castling, etc.), and the dependence and division of information between routines and data should remain flexible and open to change and never solidify into a resistant collection of conventions (Newell). List processing languages are specifically designed to cope with such problems.

A chessboard is made up of squares, which lodge pieces, which make moves from one square to another. Objects in chess, like the 64 squares and 32 men, can be represented as symbols on lists, and moves can be represented as names of description lists with certain prescribed associations (such as the square from which a piece comes, the square to which it moves, and the kind of move in question). A chess position, moreover, can be fully described as a list of pieces and squares and a chess game as a list of moves that originate from a standardized initial position and terminate in a well-defined checkmate position.

**Setting up the Chessboard**

A chessboard is made up of eight ranks and eight files which rule off 64 squares. The sequence of symbols S1...S64 is used to denote these 64 squares.

In the data section of the program there are a list of ranks, L1, containing members R1 through R8, and a list of files, L2, containing members F1 through F8. Each rank is itself the name of a list containing eight member squares; e.g., R1 contains S1, S2, ..., S8.

In the routines section of the program a super-routine, E1, sets up a chessboard; it calls nine routines E2-E7, E9, E10, and E12, which do the work. Routine E2 builds the eight file lists, F1 through F8, out of the rank lists, R1 through R8. Then routine E12 takes each of the 64 squares and assigns it rank and file (x,y) coordinates, which are later used to compute another set of relations among squares.

**Squares.** For each square on a chessboard it is essential to know: 1) the name of its occupant, if there is one, and 2) the name of all its neighboring squares in the chess-legal directions. The first desideratum is effected by defining an attribute MO, "Man on Square?" on the description list of every square and assigning as its value the name of the piece occupying it—if there is one.

The extensive network of relations among squares, constituting all legal move directions in chess, is captured by defining 16 directions on the chess-
board, beginning with D1 for the forward direction and continuing clockwise to D16 for forward and left (e.g., the Knight’s move KN1 to KB3).

Thus the even numbers (D2, D4, ..., D16) define the eight possible Knight move directions; half the odd numbers (D1, D5, D9, D13) define rank (horizontal) and file (vertical) directions; the other half (D3, D7, D11, D15), diagonal directions.

Routines E3, E4, E5, and E8 use lists L1, L2, L5, and L6 to build the network of relations among squares by assigning to each of the 64 squares as values of each of the 16 directions that obtain. In the initial position, for example, the list structure of the square S8—White’s KR1 in standard American chess notation—would look like this:

<table>
<thead>
<tr>
<th>Name of list</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square KR1 (S8)</td>
<td>Man on Square (NO)?</td>
<td>White King Rook (M8)</td>
</tr>
<tr>
<td></td>
<td>Square to the North (D1)?</td>
<td>Square KR2 (S16)</td>
</tr>
<tr>
<td></td>
<td>Square to the West (D13)?</td>
<td>Square KN1 (S7)</td>
</tr>
<tr>
<td></td>
<td>Square to the WNW (D14)?</td>
<td>Square KB2 (S14)</td>
</tr>
<tr>
<td></td>
<td>Square to the NW (D15)?</td>
<td>Square KN2 (S15)</td>
</tr>
<tr>
<td></td>
<td>Square to the NNW (D16)?</td>
<td>Square KN3 (S23).</td>
</tr>
</tbody>
</table>

Note that in such a list structure representation only those five of the 16 legal directions that are needed are defined; space in memory is not consumed by providing the “information” that the other 11 directions have no values, as it would seem to be in a matrix representation of this kind of data. (Cf the argument in Newell,21 pp. 411-412, for a fuller statement of this view.)

**Men.** The 32 men take and retain their designations from their placement in the initial position; they are denoted by the sequence of symbols M1 ... M32.

For each piece, it is essential to know:
1. the square he occupies (attribute S0),
2. his type (attribute A1),
3. his color or side (attribute A2),
4. his legally permissible move directions (attribute A20), and
5. his legally permissible capture directions (attribute A21).

The first of these is effected by defining an attribute SO, “Square on?” on the description list of every piece and assigning as its value the name of the square the piece currently occupies. The other four attributes assume the complete range of values in accord with the rules of chess.

In the data structure there are 11 lists that group the chess men by types or otherwise useful categories: White Pawns; Black Pawns; Bishops; Rooks; Knights; Queens; Kings; White Rooks, Bishops and Queen; Black Rooks, Bishops, and Queen; White Knights; and Black Knights.

For each side, moreover, there is a list giving the type of each man on that side, and another list giving the move directions of each type of man.

Routines E6 and E7 assign to each man his type, color, move directions, and capture directions. In the initial position, for instance, the list structure of M8 would look like this:

<table>
<thead>
<tr>
<th>Name of list</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>White King Rook (M8)</td>
<td>Man on what square (S0)?</td>
<td>a Rook</td>
</tr>
<tr>
<td></td>
<td>Type of man (A1)?</td>
<td>a list of directions D1, D5, D9, D13</td>
</tr>
<tr>
<td></td>
<td>Color of man (A2)?</td>
<td>White (K10)</td>
</tr>
<tr>
<td></td>
<td>Move directions (A20)?</td>
<td>a list of directions D1, D5, D9, D13</td>
</tr>
<tr>
<td></td>
<td>Capture directions (A21)?</td>
<td></td>
</tr>
</tbody>
</table>

**Positions.** A chess position can be described fully in terms of squares and pieces. Since MATER is supposed to find a checkmate in any given position, obviously some representation is necessary for encoding a particular position. This representation is a describable list called the position list, L10. Its main list consists simply of the name of each man present on the board and the name of the square each man occupies, arranged in attribute-value pairs. Its description list contains a set of special attributes pertinent to the characterization of the position; in particular, S65, the “Whose move is it?” attribute that flip-flops between K10 (White on move) and K11 (Black on move); S66, the name of the castle list that contains the Kings and Rooks still “eligible” for castling; S67, the signal cell that gets set when an en passant capture is in the offing; and S69, the name of the most recent move made on the board.

Routine E10 takes as input any position list—either the initial position or the mating position, which is read in from cards by routine E90—and
converts it into a set of associations between pieces and squares such that every piece has an attribute SO ("Square on?") with the square that piece occupies as value, and every square has an attribute MO ("Man on?") with the name of the chess piece—if there is one—occupying that square as its value.

This ends what might be called the "static" perceptual relations on the chessboard. What follows is a bundle of basic routines that attempts to provide "dynamic" perceptual relations to the program.

**Move-Making Capabilities**

Moves are the operators that transform one chess position into another. What are the common properties of chess moves? Each involves a piece, or sometimes two, going from one square to another. If the "to square" is already occupied, the move is called a capture. If the "to square" is on the eighth rank and the piece a Pawn, it is called a promotion. But in all cases the common "from-to" property holds.

This permits a move to be represented as the name of a description list containing a "from square" (as the value of an attribute A40) and a "to square" (as the value of an attribute A41). This information is sufficient to specify most moves. There is a special class of moves, however, which, while adhering to this "from-to" pattern, introduce some idiosyncratic properties of pieces. Each of the following five members in this class is assigned a different value to the special move attribute A42: King's side castling, Queen's side castling, a double Pawn move, an en passant response thereto, and a Pawn promotion.

Five steps are required to make a move: first, the move must be constructed; second, it must be tested for legality; third, for repetition of position; fourth, it must actually be made on the board; and fifth, it should be printed.

Routines E51 and E52 construct regular moves and special moves, respectively. Both create a new cell or symbol, which becomes the name of the move. Both take as input the square from which the piece is to move and the square to which it is to move and assign these as values of attributes A40 and A41, respectively. For the special move routine, E52, the type of move must also be specified as input; it is assigned as the value of attribute A42. The name of the man moved is also received as the value of another attribute A51, for reasons that will appear under step 3 below. The move 1.P-K4, for example, would be represented as follows (where a1 and a2 are internal cell names):

<table>
<thead>
<tr>
<th>Name of list</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>From Square (A40)?</td>
<td>Square K2 (S13)</td>
</tr>
<tr>
<td></td>
<td>To Square (A41)?</td>
<td>Square K4 (S29)</td>
</tr>
<tr>
<td></td>
<td>Man moved (A51)?</td>
<td>White King Pawn (M13)</td>
</tr>
</tbody>
</table>

Similarly, the special move P-K8 = Q would be represented in the following format:

<table>
<thead>
<tr>
<th>Name of list</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a2</td>
<td>From Square (A40)?</td>
<td>Square K7 (S53)</td>
</tr>
<tr>
<td></td>
<td>To Square (A41)?</td>
<td>Square K8 (S61)</td>
</tr>
<tr>
<td></td>
<td>Special Move (A42)?</td>
<td>Promotion to Queen</td>
</tr>
<tr>
<td></td>
<td>Man moved (A51)?</td>
<td>White King Pawn (M13)</td>
</tr>
</tbody>
</table>

Second, a routine E18 checks to insure that a newly constructed move is legal. The routine tests, for example, whether a Bishop is moving through a Pawn, whether a Rook is making a Bishop move, whether a player is castling through check, and the like. The output is a simple "+" ("yes, the move is legal") or "-" ("no, the move is illegal").

Third, according to the laws of chess a threefold repetition of position constitutes a draw, and, according to the laws of computers, a loop. Before a move is executed, therefore, a routine E55 tests if the move under consideration has been played before in this same position. The position could not have occurred before if the move is irreversible, that is, if once the move is made on the board no subsequent set of legal moves can ever regain the exact same position. Captures, Pawn moves, and castling are all irreversible. Thus when a capturing move is constructed (step 1), it is given an attribute A44 with the man captured as its value. When a castling move is constructed, its status as a special move is recorded as the value of attribute A42. And for a Pawn move, a record is kept via the man moved attribute, A51. Routine E56, called by E55, tests for any of these three conditions to declare a move irreversible. If none of them obtains, E55 must take some further comparisons between the A40 and A41 values of the proposed move and earlier moves.

Fourth, a routine E65 makes a regular move on the board; routines E71–E75 and E81–E85 execute the special moves. A move is made by updating the position list, which is done in two steps: first, with the assistance of routine E11, the description lists of the pieces and squares affected by the move are updated. Second, the signal cells—S65, S66, S67, and S69—are affected by the move are reset.
Since different routines are needed to make each of the five special moves, these routines are simply associated with their respective special move values in the data section of the program.

Fifth, there is a print routine, E16, which prints out the name of the move, the "from square," the "to square," and the man captured, if any.

Additional Board Processing Capabilities

In addition to the move-making capabilities just described, there is a second group of routines intended to provide the machine with some more of the perceptual capabilities a human possesses; these are the board processing routines which provide answers to questions asked of the board. Routine E13 finds the direction, if one exists, between two given squares. Routine E14 tests to see if there is a piece between two squares in a given direction, and E24, if there is one and only one piece between two squares in a given direction. E15 tests if a piece is under attack, and routine E26 asks specifically if that piece is the enemy King. Routine E33 tests whether a given square is under attack, while E34 builds up the list of men of a particular color attacking a given square. Routine E36 tests if a given square is defended. These are some of the more important "building block" routines used in constructing the move tree.

Processing Speed

It might be supposed that since the program was written in an interpretive language, IPL-V, without any attempt to provide a special machine-language representation for primitive board manipulations, that it would be a very slow player. This has not proved to be the case—a tribute to the advantages of selectivity over machine brute force. The most difficult mates, requiring the examination of about 100 positions, were achieved in about 10 minutes on a CDC G-20—which would be equivalent to about three minutes for the IPL-V system on the IBM 7090. An excellent human player might be expected to take ten minutes or more to discover and verify the mate in a position of this difficulty.

ACKNOWLEDGMENTS

We would like to acknowledge the assistance of G. A. Forehand, B. F. Green, A. Newell, M. R. Quillian, and R. F. Simmons.

REFERENCES

A CHESS MATING COMBINATIONS PROGRAM


MULTIDIMENSIONAL CORRELATION LATTICES AS AN AID TO
THREE-DIMENSIONAL PATTERN RECONSTRUCTION*

Samuel J. Penny and James H. Burkhard
Lawrence Radiation Laboratory, University of California
Berkeley, California

INTRODUCTION

Spatial reconstruction of a three-dimensional object from a set of stereo photographs must begin with the matching of the topological characteristics in one view with those in the other views. A human is usually good at this form of pattern recognition, but he is too slow for some applications. For high-speed processing of bubble chamber photographs it has been necessary to design a computer code that will match the images seen in the various views.

If we have $n$ stereo views of an object, where each view contains images of the $m$ characteristics to be correlated, a total of $m^n$ combinations are possible when one image is used from each view. These combinations are known as "$n$-tuplets." There is then some set containing $m$ of the $n$-tuplets that describes the true matching of the images. With the physical limitation that an image in one view should not match more than one image in any other view, this solution set must be chosen from a total of $(m!)^{n-1}$ possible sets. This is in the class of combinatorial problems that involve the construction of an ordered set $S = \{s_1, \ldots, s_m\}$ where the $s_i$ are elements of a finite set $U$ and the elements of the set $S$ must be chosen subject to certain restrictions.

One approach for finding the correct solution set would be to compute the likelihood for each possible set of $m$ $n$-tuplets, and then choose that set with the maximum likelihood. This approach can require a prohibitive amount of computation and is unreliable when there are large errors in the data. An alternative approach would be to use some type of elimination process, but even this can be expensive unless a technique is used to simplify the bookkeeping and to provide a rapid correlation among the various steps in the process. The "correlation lattice" and methods for scanning it for solution sets (by means of an iterative scheme of backtrack programming\(^1\)) supply the simplicity and speed to make this approach economical on a computer.

THE THREE-DIMENSIONAL LATTICE

At the Lawrence Radiation Laboratory in Berkeley, much of the high-energy-physics research deals with nuclear-particle events occurring in a bubble chamber. Three stereo views of the event are photographed, and points along the track images are measured. The images must then be matched so that the spatial trajectories of the particles involved in the interaction can be mathematically reconstructed. Using the parameters of the trajectories, we do a kinematic analysis of the interaction to produce the information required by the physicist for his experiments.

Figure 1 is a photograph and sketches of a simple nuclear event that occurred in the bubble chamber.

---

*This work was performed under the auspices of the U.S. Atomic Energy Commission.