A Chess Combination Program Which Uses Plans

Jacques Pitrat

Centre National de la Recherche Scientifique 75007 Paris, France

Recommended by Hans Berliner

ABSTRACT

The program analyses carefully the initial situation. It creates some plans and tries to execute them. It analyses the situations deeper in the tree only if the plan fails. In that case it generates new plans correcting what is wrong in the old one. So, the program considers only natural branches of the tree. It can find combinations for which it is necessary to look more than twenty ply ahead. The paper describes the methods used for analyzing a situation and for modifying unsuccessful plans. Then we examine some results found by the program.

1. Introduction

1.1. Utility of plans

Artificial Intelligence programs generate generally a very large tree. Heuristics eliminate many branches, but they select when a node is developed for the first time; therefore if some branch is eliminated, it is often impossible to consider it again later. But it is very difficult to choose branches in that way: either we eliminate important branches or we keep all the branches and the tree is too large. It is frequently unnatural to consider many branches when we find them for the first time; but, after developing some other subtree, it becomes evident that we must consider them for correcting what is wrong in this new subtree.

It is also very expensive to perform for each node of the tree the same analysis that is done at the root of the tree. It is better to analyse carefully the initial situation, and to generate plans, i.e. sequences of actions. We carry out these plans when we generate new nodes deeper in the tree. If all is going well, no analysis is necessary: we execute the next step of the plan. A new analysis is useful only if something goes wrong; but this analysis is short, we try mainly to see how we can correct the anomaly. It is cheaper than the initial analysis. It is well known (De Groot [4]) that a chess player perceives the essential properties of his position during the first moments. He is not engaged in a search of move sequences from the beginning like many chess playing programs. The analysis of the given position may be very lengthy, it is done only once.

Artificial Intelligence **8** (1977), 275–321

It was necessary to use some specific problem for showing the interest of this method. We choose to write a program finding combinations in chess. It is a well known problem, which is difficult even for a good player. It is necessary to develop very long tactical sequences. The player must be able to look 14 ply or more ahead (Berliner [2]). It is very difficult for present chess playing programs to develop such trees.

1.2. An expository example

Let us see the first steps of the program when it finds a combination for Fig. 1 (M39).[1] The goal is to win a knight.

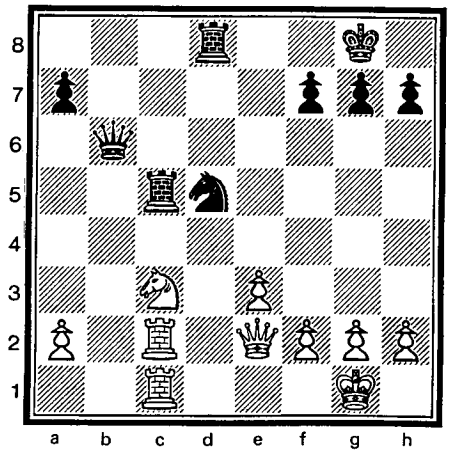

FIG. 1. M39: Black to play.

The initial analysis finds some ten plans. Many of them result from the following observation: the white king is on the first rank and it cannot move out of this rank.

So, if we can move a rook or the queen on this rank, we win. We have particularly the plan P1.[2]

P1. Remove the black knight from d5[3]

[1] The position is named by the combination of the letter referring to a book and the number of the diagram in the book. E stands for Euwe [6], L for Le Lionnais [11], M for du Mont [5], T for Tarrasch [16].

[2] Plans are named by the letter P followed by a number.

[3] We use the algebraic notation. The eight files counting from left to right are lettered consecutively a to h. The eight ranks counting from the side of the board initially occupied by the white men are numbered consecutively 1 to 8. Each square is named by the combination of the letter of the file and the number of the rank in which it occurs. (See Fig. 1). A move will be recorded by the designation of the man moved (not being a pawn: K for King, Q for Queen, R for Rook, B for Bishop, N for Knight) followed by the designation of the square it occupied and then the square to which it has been moved. × indicates a capture and + a check.

Rd8d1
Rd1 × g1.

Several methods can be used by Black for removing his knight from d5. He may capture an enemy man with this knight, for instance the white knight. So we generate a new plan:

P2. Nd5 × c3
Verify that d5 is empty
Rd8d1
Rd1 × g1.

Now we must look for what the opponent can play after Nd5 × c3, the first move of our plan. The most natural move is to recapture the knight with the rook. So after 1 . . ., Nd5 × c3; 2Rc2 × c3, we consider the following step of plan P2. As d5 is now empty, it succeeds and we play 2 . . ., Rd8d1 which is the third step of the plan. This move threatens Rd1 × g1; if the opponent does not play, it loses. King's moves: Kg1h1 and Kg1f1 are futile. But two capture moves are interesting: Rc1 × d1 and Qe2 × d1. After these moves we cannot execute the fourth step of plan P2. The program chooses one of these moves, for instance Rc1 × d1, and tries to destroy it. It is essential to play the black rook to d1; so we must first remove the white rook from c1. After 1 . . ., Nd5 × c3; 2Rc2 × c3, a new plan is considered:

P3. Remove the white rook from c1
Rd8d1
Rd1 × g1.

There are several ways for removing an enemy man *M* from a square. We analyse the position only for achieving this goal. One of the possibilities is to capture a man protected by *M*. Here the rook in c1 protects only the rook in c3. We look for a friendly man which can capture the rook in c3. There is only one move: Rc5 × c3. P3 becomes

P4. Rc5 × c3
Verify that there is no white rook in c1.
Rd8d1
Rd1 × g1.

After 2 . . ., Rc5 × c3, the most natural move is 3.Rc1 × c3. After the two exchanges, we execute the second step of P4: there is no white rook in c1. We can execute the third step and we generate 3 . . ., Rd8d1. For this, no analysis of the position is necessary: it is sufficient to verify that the move is legal. This move threatens Rd1 × g1. White is obliged to play Qe2 × d1 and Black cannot play the last step of P4. It has to destroy the move Qe2 × d1. The only way is to remove the

queen from e2. After 1 . . ., Nd5 × c3;2 Rc2 × c3,Rc5 × c3;3 Rc1 × c3, we generate the plan:

P5. Remove the white queen from e2
Rd8d1
Rd1 × g1.

For removing the white queen, we may threaten her. There are several moves, for instance 3 . . ., Qb6b2, which threatens Qb2 × e2. If white plays 4Qe2 × b2, Black plays the following step of the plan: Rd8d1 and wins.

This is only the main variation of the combination. At each level, white, using the same method, finds the black moves which are dangerous and generates plans for destroying the combination. For instance it finds that after 3 . . ., Qb6b2, it must move its queen (Qb2 × e2 is a dangerous move), while always controling the square d1 (Rd1 × g1 is a dangerous move). So it finds the move 4.Qe2c2; thus Black has to find another plan, because if it plays 4 . . ., Rd8d1, white plays 5Qc2 × d1.

So we develop a tree including only natural moves. The tree may be very large if the problem is difficult: we must consider all the natural replies of the opponent. If they are not good, generally the execution of the following steps of the plan gives a combination and no time is wasted for analyzing unrelated positions.

1.3. Related work

Chess programs develop generally very large trees (Greenblatt [10], Gillogly [9]). For instance for Gillogly, all moves are searched to a fixed depth (usually five ply). After which, it investigates all sequences of captures. The tree is used for finding combinations. If there is no combination, the program uses a very interesting positional analysis for finding the best move. But this analysis does not use the tree. This program develops a tree for the same reason that our program. But there are many combinations that cannot be found in using this large tree: frequently it is necessary to consider a move which is not a capture at a depth greater than five. It is not easy to see how this method can be improved. The computer time increases exponentially with the depth. With this method, it is possible to write programs which will play reasonably well, but we doubt that they could reach some day the level of a grandmaster even with faster computers. The majority of the combinations found by our program cannot be found by Gillogly's program. In these cases, it may play the good move, but by chance, without analyzing some important variation.

Baylor and Simon [1] have written a chess mating combination program. Their method is different and they are interested mainly in mating combinations. They can use moves threatening mate. As we see later, these are not considered by our program. So, the positions in which these programs find combinations are very different.

Berliner [2, 3] has realized a very interesting program. This program and ours have several common features, although they are written independently. It tries to understand the consequences of a move and to generate moves preventing what happens if it is wrong. But the program does not use plans. His goal is more ambitious, since his program plays and not only finds combinations.

Plans were used in other areas than chess. For instance by the robot of the S.R.I. (Fikes [8], Sacerdoti [14]) or by Lawaly (Siklossy [15]). But these plans are very different from those used in our program. First, they find them in defining abstracted spaces where some details are omitted. These details are always the same in their environment. For instance the condition "an object is pushable" is always important, while "a door is open" is not: a door can always be opened if necessary in their world, but if an object is not pushable, it is impossible to change that fact. At chess the problem is quite different and it is not easy to define abstracted spaces. The existence of the queen may be a mere detail, the existence of some pawn may be essential.

Another difference is that plans generated for robots become more specified, but do not alter the original order. At chess, we are frequently obliged to modify drastically the initial plan. For instance we discover that some enemy man destroys our plan, and we have to correct this trouble; for this, we may have to change the beginning of the plan. In another example, the initial plan leads to a mate with a rook, however we end up with a combination where a knight captures a pawn. But the only natural way for finding this combination is to try the first plan.

Fahlman [7] has also used plans for robots. But usually Fahlman's plan is the wanted result: it has to be executed in the real world, and the author does not study what the program has to do if some unexpected event occurs. In that sense, his plans have nothing to do with our plans. But in some cases, a subplan already found is manipulated as an object as in our work. If it has to build a movable subassembly, it keeps all the steps that occur after the placement of the base block. If it succeeds in building the subassembly, it tries to make as much use as possible of the old plan. Naturally it must verify each step of this plan fragment. But Fahlman's program does not generate at the beginning a plan for building some structure. It tries only to find what block may be placed first. It creates a plan which could be tried by a robot in the real world. But it does not plan for creating the plan.

Another important aspect of this work is the ability to modify already generated plans by placing the modification into the middle of the planning sequence. If a plan fails and if one reason for this failure is some enemy move M, the program tries to destroy this move M before achieving the old plan. It generates a subplan for destroying M; this subplan is executed instead of the friendly move just before move M. After the subplan, the program will execute the end of the initial plan. The beginning of the plan will also be unchanged: the program does not change the first moves of the sequence, except the move before M.

2. The Program

Rules of chess have been simplified; there is no castling, nor capture en passant. This is not important, these moves are rather infrequent in a combination. There is also no stalemate. This is a more serious limitation: some beautiful combinations are based on the search for a stalemate, if a player has a hopeless position.

The program receives the description of a chess board, the player (Black or White) who has to play and the value of the combination which it has to find. For instance it knows that it has to checkmate the opponent's king or to win at least one rook or only one pawn. This is an important difference from a chess playing program, which does not know what it has to win. It would be necessary to give to the program some method for evaluating what it is possible to win in a given position. Berliner [3] uses EXPCT for doing this. This problem is not easy, even for a human chess player. In chess books, the author gives only the main variations, where some important enemy man is captured. But in other variations it is possible that we cannot win more than a pawn, or in some cases only a positional advantage. It happens also that there are several combinations in one position, and it is interesting to find the one where we capture the most important man. In that case, we indicate the value of this man. If the program has to win one pawn, it generates all the possible plans, even those for mating the king. If such a plan succeeds, the program has won more than one pawn; if the plan fails, the program may win some enemy man that the opponent must sacrifice for avoiding the mate. In contrast, if the program must mate, it does not try a plan winning only one pawn; if it has not checkmated its opponent, the program fails, even if it has captured some enemy piece.

The value of a combination is the material balance. It is computed with the following values:

King: 1000; Queen: 95; Rook: 45; Bishop: 30; Knight: 30; Pawn: 10.

These values are read, and it is easy to give another set of values. Perhaps this set was not the best one. But a set such as 1000:90:50:30:30:10 would give similar results in most cases. But the values given above are used for all the positions in this paper. The plans are used for generating the moves which have to be considered in a position. They add new nodes to the tree and do nothing else. The program finds a combination, if, when it uses the minimax procedure, the backed up evaluation is greater or equal to the wanted value. The evaluation function for the leaves of the tree is the value of the material of the program minus the opponent's material.

2.1. A language for writing plans

A plan is a sequence of statements. There are four types of statements.

*2.1.1. Move statement: x y z t **

The asterisk is optional. x and z stand for letters a to h, y and t for numbers 1 to 8. This statement means that we have to consider the move: the man in square xy

There are four kinds of modification statement.

(1) Remove a friendly man. First we may want that a square becomes empty: the friendly man obstructs a line; if forbids an interesting move of another man. In this case, we write F → V. But we may also want that the square is no longer occupied by a friendly man, because an enemy man threatens our man. In that case, we write F → NF. NF represents the state where there is no friendly man in the square. We may have an enemy man, which is not good for F → V. In both cases, it is good if the square becomes empty.

We have an asterisk if the friendly man not only clears the space, but also has to capture or threaten an enemy man. If there is no asterisk, it is sufficient to play our man freely.

EXAMPLE We write the initial plan P1 of M39 (Fig. 1)

P7. d5 F → V*
d8d1
d1g1*

The square d5 must be empty. It is not sufficient to play our knight: White would play for instance g2–g3. We want to capture or threaten a white piece with a move of the knight.

(2) Remove an enemy man. There are three reasons for doing this.

(i) We want the square to become empty. We write E → V. The enemy man obstructs a line of one of our pieces.

(ii) We want that the square is no longer occupied by the enemy man which is there now. The square may be occupied by a friendly man, another enemy man or empty. We write En → NEn. n stands for the initial of the man in the square. For instance, the plan P5 of M39 (Fig. 1) is:

P8. e2 EQ → NEQ
d8d1
d1g1*

The reason of this modification is that the enemy man is dangerous or forbids a combination. We prefer it on another square.

(iii) We want that the square remains occupied by an enemy man, but not of the same nature that the one which stands now in this square. We write En → ENn, n stands for the initial of the enemy man. This kind of modification statement is useful when the enemy man can play a dangerous move, but the presence of an enemy man on this square is essential, because we intend to capture later the man which is on this square.

(3) Move a friendly man to some square. We want that this square is no longer empty. We obstruct a line of an enemy man. We write V → NV. It is always good if our man is captured by an enemy man which obstructs also the line.

moves to square *zt*. If there is no asterisk and if the move is legal, we put this move in the tree. If it is not legal, the plan fails and we do not consider its following steps.

If there is an asterisk, we consider always the immediate next step. But we put the move in the tree only if it is a legal capture move. Intuitively a move with an asterisk is the execution of a threat. It is conceivable that the move is no longer possible because the opponent has just captured our man or moved his man. It may also occur that this move has no interest, because the opponent has protected the threatened man. But these possibilities were foreseen by the program, and the program must execute the end of the plan.

For M39 (Fig. 1), remove the white queen from e2 in plan P5 gives the plan:

P6. b6b2
b2e2*
verify that the white queen is not in e2
d8d1
d1g1*

After Qb6b2 and the opponent's reply, we consider the statement after: b2e2, even if Qb2 × e2 is not legal (the black queen has been captured for instance) or is not a capture move because the white queen has moved from e2. It is just what we want. But if Qb6b2 or, later, Rd8d1 are not legal, we stop the execution of the plan.

2.1.2. Modification statement

This statement includes the coordinates of the square to modify, the state of this square before the modification, an arrow, then the wanted state of the square. Optionally there is an asterisk. Its meaning is given for each kind of modification.

In the description of the state of a square, F stands for Friend, E for Enemy, V for empty (Vacant). The nature of the man, if necessary, is indicated by K for King, Q for Queen. . . . If N is the first letter of a state, it indicates that the square must not have the following state.

EXAMPLE e2 EQ → NEQ.

There is the enemy queen in square e2 and we try to get a position where this queen is not in square e2. This square may be empty or occupied by a friendly man, or an enemy man other than the queen.

e2 E → V.

There is an enemy man in square e2 and we want a position where this square is empty.

When it executes such a statement, the program generates a subplan for realizing this modification. The modification statement is substituted by this subplan. After the subplan it creates a verification statement which verifies that the opponent has played what was expected and has not disturbed our plan. Then we go back to the old plan.

If there is an asterisk, we must create a threat while we are playing the move which realizes this modification. For instance, in its new square, our man can threaten the enemy queen.

(4) Induce some enemy man to come on the square. We write NE → E. We indicate eventually the nature of this man. For instance if we want the enemy king on square f8, we write

f8 NEK → EK.

In that case, f8 may be initially empty or occupied by a friendly man, or by an enemy man other than the king.

2.1.3. Verification statement

We have the coordinates of a square and the wanted state for this square. The wanted state is the part of a modification statement following the arrow. If we have NF, the program verifies that the square is not occupied by a friendly man. If we have EK, it verifies that the Enemy king is on the square.

If the verification fails, the plan is terminated. If the verification succeeds, the next statement of the plan is considered.

EXAMPLE The plan P3 of M39 (Fig. 1) is:

P9. c1 ER → NER
d8d1
d1g1*

The modification statement generates several new plans. Among them, there is:

P10. c5c3
c1 NER
d8d1
d1g1*

The capture move Rc5×c3 captures a piece protected by the rook c1. If c1 recaptures, the verification statement succeeds and we consider the following statement which generates the move Rd8d1. If c1 does not recapture the verification fails, so it is unnecessary to consider the following statements. If the balance of the captures is good after the enemy move following Rc5×c3, the program succeeded. If the balance is not good, for instance the opponent captured some other black piece, it will be necessary to find another plan.

In the same way, the execution of the modification statement of plan P8 of M39 (Fig. 1) will give several plans. Among them:

P11. b6b2
b2e2*
e2 NEQ
d8d1
d1g1*

After Qb6b2 and White's reply, we consider Rd8d1 only if the white queen is no longer in square e2.

2.1.4. Limited analysis statement

We indicate the coordinates of some enemy man M, then the word "analysis". When we execute this statement, we try to find some simple combination based on M. For instance a double attack on M and some other enemy man.

2.2. Generation of the plans for the initial position

This analysis is done only once, in the initial position. There are four main kinds of plans.

2.2.1. Capturing a man

For each enemy man, we look for the friendly men attacking it and for the enemy men protecting it. We consider also the potential attacks where some man obstructs a line. If it is a friendly man, we have a discovered attack.

EXAMPLE E42 (Fig. 2).

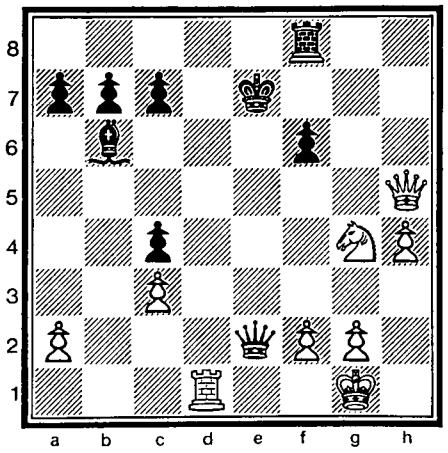

FIG. 2. E42: White to play.

White has the plan:

P12. g4 F → V*
h5e2

There is an asterisk in the modification statement: we must create a threat while removing the knight from g5. If not, Black would move his queen.

If the obstructing man is an enemy man, we use this pinned man. We generate a plan where we try to remove it from its square (modification E → V) and then we capture the man behind the pinned man.

EXAMPLE E42 (Fig. 2). If Black could play instead of White, he would have the plan:

P13. f2 E → V
b6g1

It may happen that it is possible to capture an enemy man without risk. In that case, the plan has only one statement: the move capturing the man.

EXAMPLE E42 (Fig. 2). If Black could play first, it would generate the plan:

P14. e2d1

It occurs frequently that a man could be captured if some protecting man is removed. We generate a plan with only one statement: the move capturing the man. After the program realizes that an enemy man recaptures our man, it generates various plans where it tries to destroy this enemy move before capturing the man.

EXAMPLE E42 (Fig. 2). If Black could play first, he would have:

P15. e2g4

But after Qe2 × g4, White would play Qh5 × g4. Then Black generates plans destroying this move—if possible—before playing Qe2 × g4.

We do not program the capture of men which can play only a few moves. In some cases we may capture a queen or a bishop in that way. It is possible to get a man in a position where there is no escape. But this situation is unfrequent, so we do not program this, although it would be possible to write a subroutine ad hoc. The program considers only the case of the king (cf. 2.2.3) and of the pawn with a man before it.

2.2.2. Double attacks

Pinning a man is a special type of double attack. The analysis made for 2.2.1 indicates which men could be captured if the program attacks them once more. For each couple of such men, P_i and P_j, we look for a friendly man M attacking them simultaneously if it moves on some square S. If the square S is occupied by a friendly man (or by an enemy man if M is a pawn), the program creates first a modification statement S F → V*. If some intermediary square I between M and S, S and P_i or S and P_j is obstructed, it generates modification statements: I E → V or I F → V*. Then it puts: the man M moves to S, and the two capture moves: the man in S captures P_i, and: the man in S captures P_j. These two capture moves have an asterisk: if the opponent destroys the threat for one man, it must consider the move which captures the other.

EXAMPLE M58 (Fig. 3). The pawn on a2 is not protected. It is always good to attack the king.

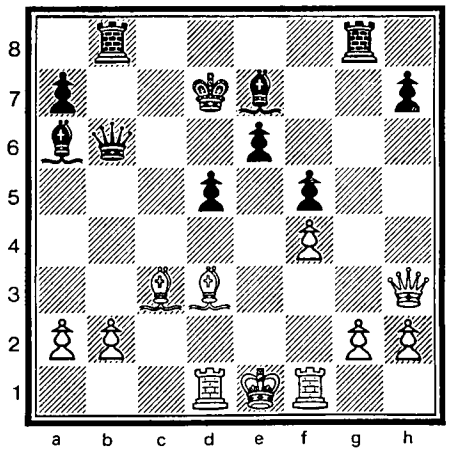

FIG. 3. M58: Black to play.

A queen on square e6 attacks both squares a2 and e1.

Therefore, it considers the plan:

P16. d5 F → V*
e6 F → V*
b6e6
e6e1*
e6a2*

If P_i, P_j and S are on the same line, it adds a statement between the first capture statement and the second one. If the first man may be protected (its value is not greater than the value of the attacking man M), it places a modification statement E → V for its square. If the opponent protects the man which would be now pinned,

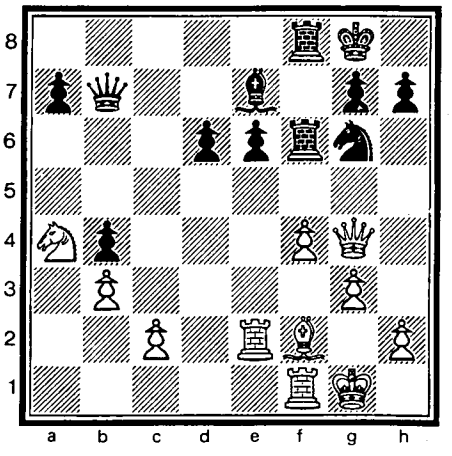

FIG. 4. T117: Black to play.

it will try some method for removing it. If the man cannot be protected, it is sufficient to insert a verification statement: if the first capture move is not possible because the opponent has played his man, the program tries the second capture move.

EXAMPLE T117 (Fig. 4). A rook on the file g attacks the king in g1 and the queen in g4. The move: Rf6g6, moves a rook on that file.
It would be legal if the program removes first its knight from g6. It must also remove the white pawn from g3: it prevents the move Rg6 × g1. So the program generates the plan:

P17. g3 E → V
g6 F → V*
f6g6
g6g4*
g4 V
g6g1*

The queen cannot be protected, as she is threatened by a rook. Therefore there is a verification statement on g4. If there were a knight on g4 instead of a queen, we would have instead: g4 E → V. If, in this case White protects his knight, the program may try to attack it again, with a pawn for instance.

When the attacking man M is a knight, the program examines if it can move to a square S where it attacks one enemy man P_i and a square T next to the opponent's king. If so, it generates a double attack, but the plan begins with a modification statement T NEK → EK, inducing first the enemy king to move to square T.

EXAMPLE M9 (Fig. 5). The program generates:

P18. e1 NEK → EK
f2d3
d3e1*
d3f4*

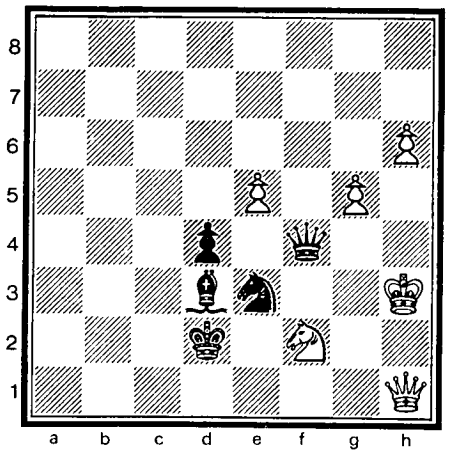

FIG. 5. M9: White to play.

EXAMPLE E2 (Fig. 6). It generates:

P19. f8 NEK → EK
d4e6
e6f8*
e6c7*

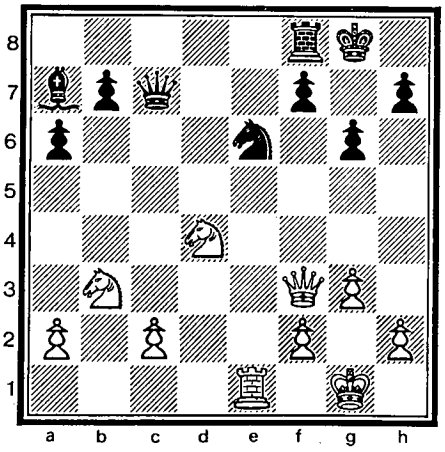

FIG. 6. E2: White to play.

2.2.3. Attacking the king

The program generates a plan including a move giving a check on square *S* followed by the move capturing the enemy king if:

(1) The square *S* is controled by its forces: the opponent cannot capture the man giving the check.

EXAMPLE E42 (Fig. 2).

P20. h5h7
h7e7*

The white queen in h7 cannot be captured.

(2) When its man is on square *S*, every square where the opponent's king can play is controled by its forces, except, eventually, the square *S*. In that case the program generates, if necessary, modification statements at the beginning of the plan if some man obstructs a line.

EXAMPLE E42 (Fig. 2).

P21. h5f7
f7e7*

This plan is generated although the black king may capture the queen on f7.

EXAMPLE M58 (Fig. 3).

P22. b6e3
e3e1*

EXAMPLE M39 (Fig. 1).

P23. d5 F → V
d8d1
d1g1*

EXAMPLE E2 (Fig. 6).

P24. f7 E → V
f3f8
f8g8*

But the program does not consider all the plans which create a check. For instance for E42 (Fig. 2) it does not generate at the beginning the check move: Rd1d7+: White does not control the square d7 when the rook is on d7 and the black king may escape to e6. In the same way, it does not consider for M9 (Fig. 5): Qh1e1+; the king can capture the queen or it can play on c2.

We do not program the generation of moves threatening mate without giving check. It would be possible to write a subroutine generating plans including such moves. But it would be a large subroutine and in many combinations it would not be used, mainly when we do not try to checkmate the opponent. Therefore the program does not receive combinations where it is necessary to consider moves threatening mate. This is not a problem of computer time: these moves do not occur frequently and for many positions, no move of this type would be generated. But for realizing the subroutine, a difficult analysis—from a chess point of view—is necessary. As our goal is to test a method and not to write a chess playing program, we did not try to implement this possibility.

2.2.4. Promoting a pawn

If there is a white (black) pawn on the ranks 6 or 7 (3 or 2), the program generates a plan for pushing this pawn. If there is a man before the pawn, the plan begins

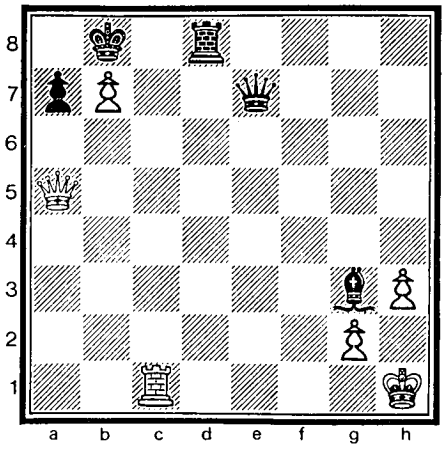

FIG. 7. M3: White to play.

with a modification statement removing this man: F → V* or E → V. If the nearest square forward of either of the diagonals is empty, the plan begins with a modification statement: NE → E; after this statement, the pawn captures.

EXAMPLE M3 (Fig. 7).

P25. b8 E → V
b7b8

P26. c8 NE → E
b7c8

The program considers two kinds of promotions: the pawn is exchanged for a queen or for a knight. The exchanges for a rook or a bishop have no interest: we do not consider stalemate.

2.3. Execution of a plan

The program adds moves to the tree only if some plan indicates doing this. Two lists of plans are associated with each node of the tree. Let us suppose that the move Q is a friendly move. The first list contains all the friendly plans which indicate to consider move Q. There is at least one plan in this list: a move can be generated only by a plan. The second list (which may be empty) includes the enemy plans which the opponent may execute after move Q. When a plan of this list is tried, it is deleted.

There is a tie between these two kinds of lists. Let R be the enemy move before our move Q, and $P_1, P_2, \ldots P_n$ the plans which indicate to consider R (elements of the first list tied to node R). P_i begins with a move statement generating R. The plan beginning after this move statement must be examined after our move Q. Therefore if P_i has more than one statement, it gives a plan for the second list tied to move Q. We examine them after each friendly move following R.

We may change the part played by friend and enemy.

EXAMPLE M39 (Fig. 1). Let R be the Black move 2..., Rc5×c3, following 1..., Nd5×c3 2·Rc2×c3. We have seen that the plan P10 indicated to consider it. After each White move Q following R, the program will consider the plan P27 beginning at the second statement of P10.

P27. c1 NER
d8d1
d1g1*

If there is no white rook on c1, this plan generates the move Rd8d1. Particularly, Q may be Rc1×c3. P27 will be in the second list of plans of this node.
We turn now to how the program executes the different kinds of statements.

2.3.1. Move statement

The program examines first if the move is legal. It is not always true. The opponent tries generally to destroy our plans; he may for instance obstruct a line or capture the man which the program wants to play.

If the move Q is legal (and if it is a capture move when there is an asterisk), the program adds it to the tree if it has not already been generated. The list of the friendly plans generating move Q has only one plan: the one which the program is executing. The second list of plans is created from the list of plans which create the enemy move just before Q.

If the move Q was already in the tree, it just adds the new plan to the first list of plans. It verifies however that this plan is not already in this list. Then it adds the end of the new plan to the second list of plans of all the enemy moves (if any) following Q.

EXAMPLE. Let us consider the plan P11 of M39 (Fig. 1); after the moves:

1		Nd5 × c3
2	Rc2 × c3	Rc5 × c3
3	Rc1 × c3	

This plan adds the move 3 . . ., Qb6b2. The plan P11 is put in the first list of this move. But after each enemy move following Qb6b2, for instance 4·Qe2c2, 4·Qe2 × b2, 4·Qe2e1; 4·g2g3 . . . the program puts in their second list of plans the end of plan P11:

P28. b6e2*
e2 NEQ
d8d1
d1g1*

If e2 is not empty, for instance after 4·g2g3, Qb2 × e2 is a capture move and it tries it. If e2 is empty, Qb2e2 is not a capture move, and the program does not consider it, even if it is legal. Then it verifies that e2 is no longer occupied by the enemy queen and, if the verification succeeds, it generates the move Rd8d1. For instance after 4·Qe2 × b2, it considers first 4 . . ., Rd8d1 ≠.

When the program adds a new plan to the second list of plans of a node, it verifies first that it is interesting to consider this plan. For this, the program simulates the execution of the plan. It supposes that the opponent will play exactly what it hopes. If there is a modification statement where the opponent is induced to move his king to some square, after the statement, the king is supposed to be in that wanted square.

If there is somewhere an impossibility: capturing an enemy man on an empty square, playing a friendly man from an empty square . . ., the plan is rejected. If it has not been rejected when this optimistic simulation is completed, the program verifies that the balance of the captures is sufficient: if it wants to win the queen, it eliminates a plan where it wins only one pawn. It verifies also that the plan is not absurd: a plan ending with a double attack on two pawns has no interest if, in the first steps, it wins a queen. If it is not possible to win the queen, the plan is bad, and if it is possible, the program will generate a special plan for this. In the same way, if the program's queen is lost in the first steps of the plan, threatening a

pawn is not a severe threat for the opponent. (When a man is sacrificed deliberately, the program considers in the balance that its man is captured).

Many plans without interest are eliminated with this simulation. Not much time is lost, it is very fast because the possible enemy moves are not considered. For the position at the root of the tree, all the plans are plausible, but after the opponent's replies many plans are impossible or very bad.

2.3.2. Modification statement

We suppose that the program executes the plan P beginning with a modification statement on square S, followed by a subplan Q, which may be empty. The execution of the modification statement generates subplans $R_1, R_2, \ldots, R_n$ (n may be 0). In that case, the program considers n plans $S_i.S_i$ begins with the subplan R_i; there is, after R_i, a verification statement, verifying that square S has the wanted state. Then the old subplan Q ends up the new plan.

For each kind of modification statement, we will see how the subplan R_i is generated.

(1) Removing a friendly man from square S. S F → V or S F → NF. These two cases are treated in the same way. The only difference is that the verification statements are different. In the first case, we verify that the square is empty, in the second one that there is no friendly man.

If there is no asterisk, we are not obliged to threaten. For the queen, the rook, the bishop and the knight, we generate a move statement: the piece in S plays. The program does not indicate precisely the square where this piece moves. If, deeper in the tree, the opponent has some combination, the program tries to find the squares where the piece can move for destroying that combination. For instance, if the opponent captures later a man M, and if it is possible to protect it, the program generates at the same level as the move: the piece in S plays, the moves where the piece in S can protect M. If, on S, there is the king or a pawn, all their moves are generated.

If there is an asterisk, the program must threaten. It determines the enemy men which can be threatened, and it looks for moves of its man on S attacking one of them, if necessary after removing some obstacle. The subplan R_i includes: the modification, if any, for removing the obstacles, the move from S and the move (followed by an asterisk) capturing the threatened man. If the man in S is a knight, the program tries also to attack a square next to the enemy king. But it puts first a modification for inducing the king to come to this square.

Instead of threatening, it is also possible to find moves where the man in S can capture an enemy man, if necessary after removing some obstacle.

EXAMPLE M39 (Fig. 1). For the execution of plan P23, the black knight must be removed from d5. It can capture the white knight on c3.

P29. d5c3
d5 V
d8d1
d1g1*.

EXAMPLE E42 (Fig. 2). For P12, the white knight must be removed from g4. It may attack the square d7 next to the black king on e7.

P30. d7 NEK → EK
g4f6
f6d7*
g4 V
h5e2.

(2) Remove an enemy man from square *S*. There are three cases: *S* E → V; *S* En → NEn; *S* En → ENn.

For the first two cases, the program can threaten the enemy man, for inducing it to move out of square *S*. If necessary, it may add some modifications for removing obstacles.

EXAMPLE. We have seen earlier for M39 (Fig. 1) that the plan

P8. e2 EQ → NEQ
d8d1
d1g1*

gave the plan:

P11. b6b2
b2e2*
e2 NEQ
d8d1
d1g1*

EXAMPLE M58 (Fig. 3). After 1 . . .: Qb6e3 generated by plan P22, the opponent captures the queen with 2·Qh3 × e3. This move must be destroyed. One possibility is to remove the white queen from h3:

P31. h3 EQ → NEQ
b6e3
e3e1*

The rook may threaten the queen.

P32. g8g3
g3h3*
h3 NEQ
b6e3
e3e1*

This plan is good if the opponent plays, after 1 . . ., Rg8g3, 2·Qh3 × g3. The program considers 2 . . ., Qb6e3. The move 3·Qg3 × e3 is still bad. It tries to remove the queen from g3. For this, it considers after 1 . . ., Rg8g3. 2·Qh3 × g3 the plan

P33. g3 EQ → NEQ
b6e3
e3e1*.

The bishop on e7 may threaten the queen:

P34. e7h4
h4g3*
g3 NEQ
b6e3
e3e1*

This plan gives the main variation of the combination. But, if, after 1 . . . Rg8g3, the opponent plays h2 × g3 instead of Qh3 × g3, the plan P32 cannot be used: the queen is always on h3 and the verification: h3 NEQ, fails. The winning move Qb6e3 is not considered. We will later see that another way for destroying the move: Qh3 × e3, gives the solution in both cases: instead of removing the queen from h3, the plan obstructs the third rank between e3 and h3.

Always for the first two cases of the modification statement: *S* E → V and *S* En → NEn, it is possible to capture a man protected by the enemy man on *S*. If it recaptures, the wanted result is obtained.

EXAMPLE E2 (Fig. 6). For the execution of plan P24 the program sees that the pawn f7 protects the knight on e6. Thus:

P35. e1e6
f7 V
f3f8
f8g8*

EXAMPLE T117 (Fig. 4). Black has to remove the white pawn on g3 for executing P17. But g3 protects f4.

P36. g6f4
g3 V
g6 F → V*
f6g6
g6g4*
g4 V
g6g1*

Naturally, before realizing a modification, it is necessary to verify that it has not already been done; If it is, we execute the following statement. With P36, if Black plays 1 . . ., Ng6 × f4, White can reply 2·g3 × f4. Then g3 is empty, but g6 also. The modification g6 F → V* has already been done. The move Rf6g6 is immediately generated and wins.

If the modifications of plans P17 were generated in a different order, g6 F → V* being the first statement of the plan, the same combination is however found.

Black can capture with the knight for removing it from g6, for instance the white pawn f4. This gives:

P37. g6f4
g6 V
g3 E → V
f6g6
g6g4*
g4 V
g6g1*

After 1 . . ., Ng6 × f4 2·g3 × f4 the modification g3 E → V is already done and Rf6g6 is generated.

EXAMPLE M39 (Fig. 1). In plan P9 the white rook must be removed from c1. The program can capture some man protected by this rook. There is only the rook on c3. So it obtains the plan P10.

P10. c5c3
c1 NER
d8d1
d1g1*

For the last two cases: *S* En → NEn and *S* En → ENn, it is possible to capture the enemy man on *S*.

EXAMPLE M39 (Fig. 1). After 1 *S* Nd5 × c3
2 Rc2 × c3 Rc5 × c3
3 Rc1 × c3 Qb6b2
4 Qe2c2 Rd8d1.

Black plays Qc2 × d1. A plan destroying this move and considered after 4 Qe2c2 is:

P38. c2 EQ → NEQ
d8d1
d1g1*

It gives:

P39. b2c2
c2 NEQ
d8d1
d1g1*

and Black wins with this plan.

Other methods may be used for removing an enemy man from its square. Some of them have been implemented, but they are used rarely and we do not describe them here.

(3) Move a friendly man to some square *S*. The program generates all the moves bringing a friendly man to *S*. Generally, there are not many moves doing this. If necessary it generates before a modification statement if a man obstructs the line. If there is an asterisk, it verifies that the friendly man threatens from *S* some enemy man.

EXAMPLE M58 (Fig. 3). After 1 . . ., Qb6e3, the opponent plays 2·Qh3×e3. One possibility for destroying this move is to remove the white queen from h3. We have seen that it does not give the solution if, after 1 . . ., Rg8g3, white plays 2·h2×g3: as the white queen is always on h3, the verification statement of P32 fails and the program does not consider the winning move: 2 . . ., Qb6e3.

But there are other methods for destroying 2·Qh3×e3. A friendly man can be moved on some intermediary square between e3 and h3. For instance in occupying g3, we have:

P40. g3 V → NV*
b6e3
e3e1*

Rg8g3 moves a man to the good square and it threatens the enemy queen. Thus we have:

P41. g8g3
g3h3*
g3 NV
b6e3
e3e1*

The program has now to verify that g3 is not empty.

If, after 1 . . ., Rg8g3, White plays 2·Qh3×g3, the program considers Qb6e3 and we have exactly the same solution than with P32. But if White plays 2·h2×g3, the verification succeeds now: g3 is not empty. So, 2 . . ., Qb6e3 is tried and wins after the interposition 3 Bd3e2, Qe3×e2≠.

(4) Induce some enemy man to come on the square *S*. *S* NE → E or *S* NEn → En. A friendly man creating a threat may move to that square. The opponent can capture this man for destroying the threat. If there is already an enemy man on *S*, but if this man has not the wanted nature (for example a rook instead of the king), it is possible to capture it, if it is protected by a man of the wanted nature. If the enemy king has to move to *S* and if it occupies a square next to *S*, the program looks for a man which can move to *S* (after, if necessary removing some obstacles) and which gives a check on *S*.

EXAMPLE M9 (Fig. 5). In P18, the black king on d2 has to move to e1.

P42. h1e1
e1d2*
e1 EK
f2d3
d3e1*
d3f4*

If the king captures the queen, the program executes the double attack and wins the bishop.

EXAMPLE E2 (Fig. 6). In P19 the black king must go to f8. If the white queen was on f8, she would give a check. But for this, we must remove the pawn f7 which obstructs the file f. We have:

P43. f7 E → V
f3f8
f8g8*
f8 EK
d4e6
e6f8*
e6c7*

A way for removing the pawn f7 is to capture the knight protected by this pawn. The plan where it is captured by the knight d4 is eliminated: this plan is impossible, because the knight is needed later for the double attack. It remains only the capture with the white rook.

P44. e1e6
f7 V
f3f8
f8g8*
f8 EK
d4e6
e6f8*
e6c7*

This plan gives the main variation of the combination:

1 Re1 × e6 f7 × e6
2 Qf3 × f8+ Kg8 × f8
3 Nd4 × e6+ Kf8 plays
4 Ne6 × c7.

White wins a knight and a pawn.

EXAMPLE M3 (Fig. 7). For P26, an enemy man must come to c8, so that the pawn b7 can be promoted. The program looks for a White man which can move to that square and threatens when it is there. It is possible with the white rook c1 which gives a check on c8.

P45. c1c8
c8b8*
c8 E
b7c8

After 1 · Rc1c8+, Black cannot play 1 . . ., Kb8 × c8. It must play 1 . . ., Rd8 × c8. The following move of the plan is not legal, but it is an optional move. Then the

verification succeeds: there is an enemy man on c8. So, after 1 Rc1c8—Rd8 × c8, 2 b7 × c8 is tried. But Black plays 2 . . ., Kb8 × c8. Before playing b7 × c8, white must destroy this move. The only way is to remove the king from b8. So, after 1 Rc1c8—Rd8 × c8 the following plan is generated:

P46. b8 EK → NEK
b7c8

A possibility is to threaten the king with the queen

P47. a5a7
a7b8*
b8 NEK
b7c8

If, after 2 Qa5 × a7, Black plays 2 . . ., Kb8 × a7, White captures the rook with the pawn, and when the pawn is exchanged for a knight, it creates a double attack on the king and the queen: White wins a knight.

EXAMPLE E42 (Fig. 2). For P30, the black king must be on d7. It is possible to move to this square the white rook which gives a check.

P48. d1d7
d7e7*
d7 EK
g4f6
f6d7*
g4 V
h5e2

This plan gives the main variation:

1 Rd1d7+ Ke7 × d7
2 Ng4 × f6+ K plays or R × N
3 Qh5 × e2

2.3.3. Verification statement

The program verifies that the square has the wanted state. If it is good, the program executes the following statement. If not, it leaves the plan.

2.3.4. Limited analysis statement

The analysis is similar to the analysis made for the initial position. But it is made only for one man. The program examines: if the man may be captured, eventually after removing some obstacle; if it is possible to create a double attack on this man and some other man. If the king is on the square, the program looks for possible mating attacks. Only one modification in a plan is accepted; for the analysis of the initial position, at most two modification statements were accepted in a plan. This limitation was made for avoiding exponential growth, and also because it was never necessary to consider plans with more than two modifications. Generally the plan which gave the initial idea of the combination had only one modification.

2.4. The dangerous moves

The program must find a combination for one player. It supposes that its opponent tries to forbid it to do so. At each stage of development of the tree, one of the players succeeds and the other fails. The program is interested by the player who fails. It looks for the dangerous moves of the player who succeeds and tries to correct what goes wrong for the other by creating new plans which generate new moves. Then it determines again with the new tree which player succeeds and it resumes the procedure. It is possible that the same player fails consecutively several times: his plan was bad or there were several dangerous moves and the plan destroyed only one of them. The procedure stops when a player does not succeed and has tried everything that was possible for destroying the dangerous moves of his opponent.

This method for developing a tree seems similar to the method used by human players (De Groot [4]). The program tries first some promising combination; if it seems possible to refute this combination, it tries another one, but later it may return for a deeper analysis of the first combination if the second one fails also.

We give first a more precise definition of "a dangerous move" and a method for finding them. Then we will see what the program can do when it has found a dangerous move.

2.4.1. Definition and research of the dangerous moves

Let P be the player who does not succeed: either he has to find a combination and he has not found it or he wants that his opponent has no combination. For the sake of simplicity, we suppose that P is the first player, the one who wants to find a combination and fails.

There are many things which are not good in the tree. If we take any move Q of player P at the first level and if we apply to the subtree of root Q the minimax procedure, the value backed up is not sufficient. (We have seen earlier that the evaluation function at the leaves is the balance of the captures). From the definition of the minimax procedure, there is at least one enemy move following Q for which the value backed up is not sufficient. All the moves in that case are called dangerous moves. The program chooses, using heuristics, one of them R and it will try later to destroy it. If it destroys R and if there are other dangerous moves, it will have to destroy them later. If it fails, it is not necessary to study the other dangerous moves of the same level as R.

But there may be other dangerous moves deeper in the tree. The program resumes the procedure for each friendly move (if any) following R.

We give in Fig. 8 a possible state of development of the tree for M39 (Fig. 1). In that state White succeeds, Black fails and we circle the dangerous moves. $\emptyset$ stands for a null move. We can notice that a dangerous move is not for ever a dangerous move.

In Fig. 8, Rc2 × c3, Rc1 × c3 will not be dangerous moves when the combination has been found.

If *P*, the player who does not succeed, is not the first player, his opponent, the first player, has a combination: at least one move M at the first level has a sufficient value backed up by the minimax procedure. We use exactly the same method as before with the tree of root M which is unsuccessful for *P*.

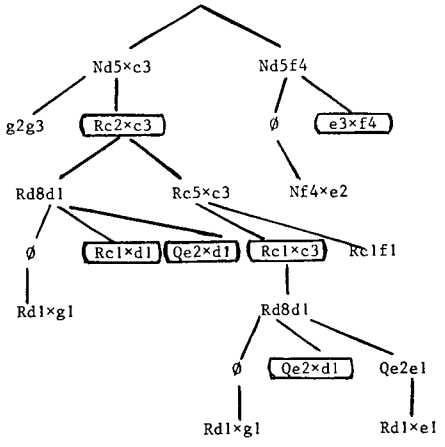

FIG. 8.

There is a little problem when a move is not good, but when there is no opponent's move after it: there is no dangerous move to destroy. In that case we examine if this move is a threat. A move M is a threat if a move with an asterisk follows it in at least one of the plans which generate M. If the move is a threat, we generate a null move (represented by ∅) for the opponent and we consider after this null move, the following move of the plan. If this threat is real, the balance of the captures will be good and this move will be dangerous for the opponent. So the opponent tries to destroy this new move and, if it succeeds, he will eventually add after M, moves which will be dangerous for the player.

EXAMPLE M58 (Fig. 3). When the program executes P32, the move Rg8g3 is not sufficient for Black: the balance of the captures is zero. But this move is a threat. The following move Rg3 × h3 is added to the tree after a null move. The balance of the captures (a queen) is good for Black. White must see what goes wrong. It finds that Rg3 × h3 is a dangerous move. It tries to destroy it, for instance it plays, after 1 . . ., Rg3 × h3 2·Qh3 × g3. Now Black fails. It looks for the dangerous moves and finds Qh3 × g3.

When a dangerous move is found, the program tries to correct its effects or to destroy it. There are many ways for doing this. The program does not try all these possibilities at once. They are sorted in six classes. It keeps with each move the number of the last class tried. If after a first attempt to correct it, the move is still dangerous, the program tries the following class. When the six classes have been tried, still not changing the outcome of the move, the program considers it a hopeless case and goes to another dangerous move. The idea is to try first the methods which succeed frequently.

In all cases the program executes the plans, if any, which are in the second list of plans of the dangerous move. They are the plans which must be executed after this move.

EXAMPLE M39 (Fig. 1). The plan P29 generates the move Nd5 × c3. If the opponent replies Rc2 × c3, the balance of the captures after this move is zero. It is not sufficient for Black, thus the move Rc2 × c3 is a dangerous move. But the end of plan P29 is in the second list of plans of the node corresponding to Rc2 × c3:

P49. d5 V
d8d1
d1g1*

The first thing to do in trying to correct the bad effects of the dangerous move is to execute the plans of its second list, and particularly P49. As d5 is empty, the verification succeeds. Thus Rd8d1 is generated. The balance of the captures is not positive for Black after this move, but the plan is not finished. A null move is generated for the opponent, then with the last statement Rd1 × g1. We have the following moves:

1		Nd5 × c3
2	Rc2 × c3	Rd8d1
3	∅	Rd1 × g1.

The balance is now good for Black. White determines why it is not good for him. The moves Nd5 × c3, Rd8d1 and Rd1 × g1 are dangerous and he tries to counter them, using the same methods.

2.4.2. The first class: generation of new capture moves

A new capture move is a move created by the dangerous move or by the friendly move played just before. A move M1 creates another move M2, if M2 is not a legal move before M1 is played. Each move changes two squares of the board (it is true because we do not program castling and capture en passant). The last two moves change at most four squares: it may happen that they change the same square. The program looks for the capture moves which may be created by each of these changes. If one of these changes is that the square becomes occupied by an enemy man, it examines if it is possible to capture this man. If the change is that the square becomes occupied by a friendly man, it examines if it is possible to capture with this man. If the change is that the square becomes empty, it examines if a friendly queen, rook or bishop is on a line including the square, and if there is an enemy man on the other side of this line.

Such moves are new capture moves. They are kept only if the capture is interesting: if the capturing man has a value greater or equal to the value of the captured man, the program sees if there is a possible sequence of captures where it wins some material advantage, whatever the opponent plays. If it cannot find such a sequence, the move is eliminated.

EXAMPLE M39 (Fig. 1). After 1 . . ., Nd5 × c3 two squares changed: d5 and c3. For instance c3 is now occupied by a black knight. This move is dangerous for white. But there is a new move capturing the white knight on c3: Rc2 × c3. If the rook c5 recaptures, it is recaptured by the rook on c1. For White the balance is good: one knight, and a plan with only the move: Rc2 × c3 is generated.

In the same way after 1 . . ., Nd5 × c3, 2 Rc2 × c3 Rd8d1, squares d1, d8, c2, c3 have changed in the last two moves. If Rd8d1 is a dangerous move for White (and it is if Rd1 × g1 is generated after a null move of White), the program looks for moves capturing the rook on d1. There are Qe2 × d1 and Rc1 × d1 which give a good result for White. The program has generated a part of the tree of Fig. 8.

For each new capture move, the program generates a plan with only one statement describing the move. But it generates other plans. Let Q be the dangerous move, R the friendly move just before Q and P a plan of the first list of R, i.e. one of the plans which created move R. Its first element is move R. Let P' the plan beginning at the second statement of P. The program generates a plan beginning with the new capture move followed by the end of the old plan: P'. This is important. For instance after an exchange the program must not forget its initial plan; if the opponent plays a capture move which has nothing to do with the program's plan, it may be necessary to recapture at once (with the new capture move) and then to go on with the execution of the old plan.

EXAMPLE E2 (Fig. 6). When executing P44, after 1 Re1 × e6, Black can play 1 . . ., Ba7 × d4 which destroys the dangerous move Nd4 × e6. After Ba7 × d4 which is dangerous for White, the end of P44 is first tried:

P50. f7 V
f3f8
f8g8*
f8 EK
d4e6
e6f8*
e6c7*

But the verification, f7 empty, fails. The program looks for a new capture move. There is a change on d4 and it can capture the black bishop with 2 Nb3 × d4. So a plan is generated, beginning with this move and followed by the end of P44: P50.

P51. b3d4
f7 V
f3f8
f8g8*
f8 EK
d4e6
e6f8*
e6c7*

If after 1 Re1 × e6 Ba7 × d4 2 Nb3 × d4, Black plays f7 × e6, White tries at once the plan beginning with the second statement of P51. As f7 is empty, the verification succeeds and the program plays without new analysis the good combination. The exchange: Ba7 × d4—Nb3 × d4 does not disturb the program which does not forget its initial plan.

2.4.3. The three following classes: destroying the dangerous move

If a friendly man on square S is captured in the dangerous move, a plan with the modification S F → NF is generated. If the balance of the captures is good at the level where the plan is executed, the man is first removed without threatening. This is the second class. In all cases, for the third class the man is removed and has to threaten at the same time. The modification statement is now S F → NF*.

EXAMPLE E42 (Fig. 2). If the following moves have been generated: 1 Rd1d7, no-move. 2 Rd7 × e7, the last move is dangerous for Black. But the balance after Rd1d7 is zero, so is good for Black—which has only to prevent a combination of White. Thus the following plan is found

P52. e7 F → NF

The king plays and it is not necessary that it threatens.

If the preceding steps fail or if the dangerous move is not a capture move, other methods are used in the fourth class of possibilities.

—If T is the square occupied by the enemy man which moves in the dangerous move, the modification statement T En → NEn is generated for removing this man. If later in the plan it is necessary to capture on square T, T En → ENn is generated instead: the man must be removed, but the square must be always occupied by an enemy man.

—If some intermediary squares D_i must be empty for the dangerous move, the statement D_i V → NV are generated. If the balance is not good, we indicate with an asterisk that a threat is necessary.

EXAMPLES M58 (Fig. 3). After 1 . . ., Qb6e3+, the opponent plays the dangerous move 2·Qh3 × e3. After this move, there is no new capture move and Black cannot move his queen from e3, since her presence is necessary on that square for realizing the end of the plan: Qe3 × e1. Then the program tries the fourth class: remove the queen from h3 and it obtains P31 which has already been seen.

P31. h3 Q → NEQ
b6e3
e3e1*

But it can also move a man to g3 or f3. We have seen P40 which gives the solution:

P40. g3 V → NV*
b6e3
e3e1*

It is necessary to threaten: at the beginning, the balance of captures is zero and it is bad for Black who wants to win some White man.

There are several other trials in the fourth class. For instance the program protects the square where the enemy man moves in the dangerous move. I do not detail them.

An important problem arises for these three classes. At what level of the tree, the new plan must be executed. Generally the plan is put in the second list of plans of the enemy move V preceding the friendly move R which precedes itself the dangerous move Q. The first move of the new plan is considered instead of move R which is unsuccessful.

But in some cases, the new plan is placed at another node. If, for instance, move V is a check. It is necessary to destroy this threat and it is not possible to realize a modification for destroying another threat at this level. Therefore the plan is placed two ply higher in the tree. The best solution in all cases would be to consider the plan after each enemy move preceding the dangerous move. But, as the tree is often very deep, there would be too many plans. There are other exceptions: if two enemy moves capture successfully on the same square, the program destroys also the second move at the place where it destroys the first one. Curiously enough, these heuristics which seem rough do not lead to trouble.

The program can notice that it is not possible to destroy a dangerous move played in the initial position for these three classes. In that case, the possibilities of the fifth class are immediately considered.

Another important problem is to find the subplan which follows the modification statement. We have found that generally the modification must be made after the enemy move V preceding the move R which precedes the dangerous move Q. A new plan generated by the program is composed of the modification statement followed by one plan of the first list of plans of move R: plans which indicate to consider this move. Thus the program generates as many new plans as there were in the list. If the balance of captures is good for the player after the enemy move V, the plan composed with only the modification statement is also considered. If the modification is tried at another level (for instance V is a check), the program uses the plans of the first list of its move preceding V. There is an exception if the enemy move preceding W is still a check. We use the same method two ply higher.

EXAMPLE E2 (Fig. 6). We suppose that White has found the main variation

1	Re1 × e6	f7 × e6
2	Qf3 × f8+	Kg8 × f8
3	Nd4 × e6+	K plays
4	Ne6 × c7	

Black looks for the dangerous White moves. Among them, there is Ne6 × c7. Black cannot destroy this move after Nd4 × e6 or Qf3 × f8 which gives a check. Therefore he tries to destroy Ne6 × c7 after the white move Re1 × e6.

It is not possible to generate a modification statement for e6 because it is necessary

for destroying Ne6 × c7 to remove the white knight from e6 and, in the position after Re1 × e6, there is not a white knight, but a white rook on e6. It is impossible to remove a man which is not on the square. But it is possible to generate a modification for square c7: F → NF*. It is necessary to threaten because White has already won a knight.

Among the plans generating f7 × e6, there is certainly the plan including only this move: this is a new capture move. So, after 1 Re1 × e6, Black will try the plan:

P53. c7 F → NF*
f7e6*

Black does not forget that, after removing his queen, he has to capture the rook on e6 with his pawn f7.

P53 generates several plans: the queen has many threatening moves. One of them is:

P54. c7d8
d8d4*
c7 NF
f7e6*

If white moves his knight, Black remembers that he can also capture the rook, although this move is no longer a new capture move. It is a new capture move only after 1 Re1 × e6, not after 1 . . ., Qc7d8, 2 Nd4 plays. Let us give an example of a variation found by the program:

1	Re1 × e6	Ba7 × d4
2	Nb3 × d4	Rf8d8
3	Re6e4	f7f5
4	Re4f4	g6g5
5	Rf4 × f5	Rd8 × d4
6	Rf5 × g5+	Kg8h8

Ba7 × d4 is generated for destroying the move Nd4 × e6 of the main variation. Rf8d8 destroys the move Qf3 × f8 and attacks the knight on d4. As this knight is protected by the rook, the program tries to remove this piece with f7f5 and g6g5. The White moves Rf4 × f5 and Rf5 × g5 are new capture moves.

We can notice that the program stops because it has won what was wanted: two pawns. It does not generate the mating move 7·Qf3f8≠. It is not necessary to try to mate on some variation if it is impossible for the other variations. If it was required to mate, the program would naturally find the good move at the sixth step (2.4.5).

2.4.4. The fifth class: the interferences between combinations

The program generates the plan in the initial position for the player who has to find a combination. But it generates them also as if the opponent could play first. It examines if some difficulty prevents the immediate realization of a plan:

obstacle, enemy man which is protected. . . If, deeper in the tree, after a dangerous move, this difficulty has disappeared, the initial plan is tried. If there is no difficulty, the plan is always tried, if it is possible.

EXAMPLE T169 (Fig. 9). Black tries to win at least a pawn. The program generates also the white plans as if White could play first.

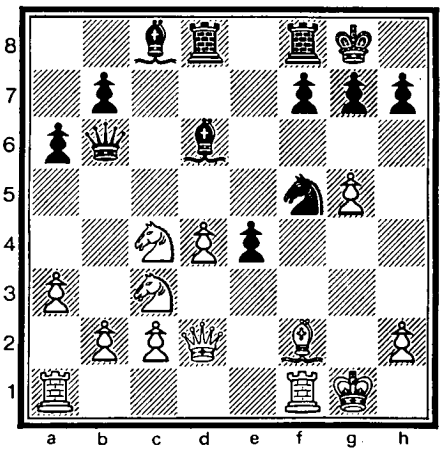

FIG. 9. T169: Black to play.

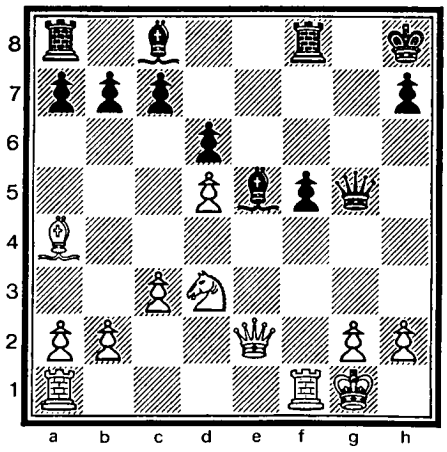

FIG. 10. T149: Black to play.

The queen on b6 and the pawn on e4 are attacked and not protected. Thus White will try, after black's dangerous moves, these two following plans:

P55. c4b6*

and

P56. c3e4*

After the sequence

1 Bd6×h2+
2 Kg1g2 Bh2 plays

Bh2 plays is a dangerous move for White. With the interferences, the moves Nc4×b6 and Nc3×e4 will be considered, although they are not new capture moves. If, instead of moving his bishop, Black would protect it with the queen, White generates only the move Nc3×e4. Plan P55 fails as the queen is not on b6.

EXAMPLE T149 (Fig. 10). Black tries to win at least a white pawn. If White could play first, plan P57 would be interesting.

P57. e5 E → V
e2e7
e7c7*
e7f8*

The queen attacks two unprotected men. But it is necessary that e5 becomes empty. If later, after a dangerous black move, White finds the square e5 empty, it tries the plan P57. For instance after the sequence

1 Be5×h2+
2 Kg1×h2 Qg5h4+
3 Kh2g1 Qh4×a4

the move Qh4×a4 is a dangerous move. As e5 is empty, White tries the plan P57 and generates 4 Qe2e7.

2.4.5. The sixth class: a limited analysis

In the dangerous move, an enemy man moves to some square *S*. The program tries to use that fact for creating a plan attacking *S*. For instance it looks for a double attack on *S* and some other man. If the king is on *S*, it tries to mate it. It uses for this the limited analysis statement defined in Section 2.1.4. Its execution is described in Section 2.3.4.

EXAMPLE E2 (Fig. 6). When the program executes P35, it gets:

1 Re1×e6 f7×e6
2 Qf3×f8+ Kg8×f8

Black's moves are new capture moves. The move Kg8×f8 is dangerous for White. The limited analysis on square f8 indicates that it is possible to attack the king and the queen on c7 with the knight. So, the following plan is considered

P58. d4e6
e6f8*
e6c7*

This is another way for finding the combination. It is not so fast as with P50, which gives it directly. Generally the program has several ways for finding a combination. Some way is more natural and is found first.

EXAMPLE M9 (Fig. 5). After 1 Qh1e1+, Black can play Kd2c2 and White cannot continue the execution of P42: the verification, e1 EK, fails. Kd2c2 is a dangerous move; the program examines if it is possible to attack the man which moved (here the king) on its new square. It is possible to attack the queen and a square next to the new square of the king.

P59. c1 NEK → EK
f2d3
d3c1*
d3f4*

which gives:

P60. e1c1
c1c2*
c1 EK
f2d3
d3c1*
d3f4

We have now the sequence

1 Qh1e1+ Kd2c2
2 Qe1c1+

If Black captures the queen, the bishop is won immediately with the double attack 3·Nf2×d3+.

Some other possibilities than the limited analysis are tried. If the friendly man, which moved during the move before the dangerous move, has not been captured, the program looks for a double attack with this man. If the friendly move before the dangerous move was a check, and if the opponent secured his king without moving it, the program looks however for a new attack of the king: something around the enemy king has changed. It is good to examine the new position.

Some heuristics, which were already used for a general game playing program (Pitrat [13]), are used by the program. Their goal is to study first the dangerous moves which are the more interesting to correct.

For instance, in the tree of Fig. 8, it is sufficient to correct one of the following dangerous moves: e3×f4, Rc2×c3, Rc1×c3, Qe2×d1. But it is not sufficient to correct Rc1×d1, because it would be also necessary to correct Qe2×d1.

I do not describe again the method given in Pitrat [13]; the important thing is to remember that the program orders the dangerous moves and tries to correct first moves such that, if it succeeds, there are as few other dangerous moves as possible to correct afterwards.

Because it is frequently easy to correct a dangerous move in generating new capture moves, however we try quickly this possibility for all the dangerous moves. If ten dangerous moves must be corrected, and if they are corrected easily with new capture moves, it would be bad not to consider this possibility.

3. Results

The program was written in FORTRAN IV for the IBM 370–168 of the C.N.R.S. It has more than 7000 instructions. List processing is done by using subscripted variables. There is a garbage collector.

The values of all the parameters were the same for all the combinations given to the program: the value of the men used for computing the balance of the captures, the number of modifications accepted in a plan (at most two for the analysis of the initial position, one in the other analyses).

One hundred and eighty-four positions were proposed to the program. They were taken in chess books (generally Tarrasch [16] and du Mont [5]). No move threatening mate was necessary for finding these combinations. We considered 100 positions given by Tarrasch in which there was a combination giving a material advantage. Many positions of this book could not be given: for some variations, the player won only a positional advantage, or the opponent played a move which was not the best one. In the others books, the positions were selected out of a larger set; there were no laws for selecting them, except we always considered a position if the author specified that the combination had not been found by the player in the real game.

The notion of branching factor has no sense for the program. It studies all the moves that are natural, for some reason, to consider. If there is no such move, it generates nothing. But in one case it has considered 26 enemy moves at depth 12. There was a reason for considering each of them. There is no limitation in depth, nor in width for the tree. If there is really a combination, it is easy to refute quickly the greatest part of the enemy moves. After most of them a plan generated higher in the tree can be executed or there is a new capture move. In both cases, the good move is quickly found.

The difficulty for the program is not to find the main variation of the combination. The program finds it generally very quickly. For instance for E2 (Fig. 6), the plan P43 which gives it was the 11th plan generated after the initial analysis. And the program had to consider more than 12 700 plans! The reason is that the opponent has many ways for escaping from a threat, and the program has to refute all of them. In a difficult position, the program may be obliged to find hundreds of small combinations. If it does not see only one of them, it fails.

The more difficult combinations are those where it wins only a pawn. It is easier to find a mating combination, and generally combinations where there are many checks: the opponent has only a few replies. If the program must win a pawn, the opponent has many ways for capturing another pawn and the tree quickly grows very large.

It is difficult to be sure that the program does not forget some interesting variation. In the books, there are only a few moves and the program, usually, finds them quickly. The problem begins with the other moves. Generally, when there is a sacrifice, the author of a book indicates only what happens if the opponent captures the man; for the program, the main difficulty is to see what happens if the sacrifice is not accepted.

3.1. Some examples

First we give two examples where the program succeeds and finds the solution given in the book.

Berliner [2] proposes the position B5 (Fig. 11) which shows the importance of developing a very deep tree. The program finds:

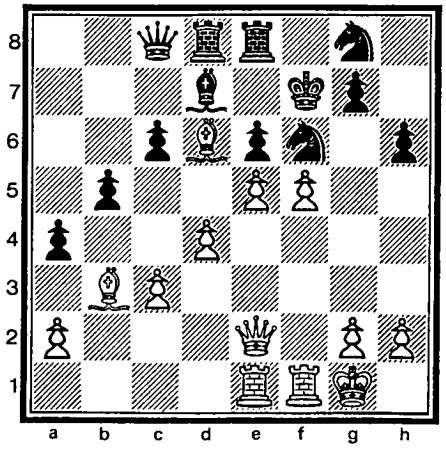

FIG. 11. B5: White to play.

1	Qe2h5+	Nf6×h5
2	f5×e6+	Kf7g6
3	Bb3c2+	Kg6g5
4	Rf1f5+	Kg5g6
5	Rf5f6++	Kg6g5
6	Rf6g6+	Kg5h4
7	Re1e4+	Nh5f4
8	Re4×f4+	Kh4h5
9	g2g3	any move
10	Rf4h4≠	

The initial plan generating Qe2h5 is found because the queen on h5 controls all the squares where the king could move and gives a check. The following moves are generated by limited analysis statements, except move g2g3. The program played first 9 Rf4h4. But Black plays the dangerous move 9 . . ., Kh5×h4. White

finds a plan controling the square h4, before playing Rf4h4. Particularly it controls the square with g2g3.

EXAMPLE M42 (Fig. 12). This was the most difficult combination found by the program. The basic idea came from the following fact: the black king can move only to squares on the eighth rank. So it is interesting to move a rook to that rank.

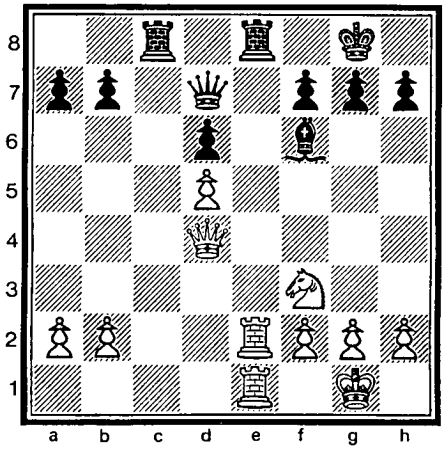

FIG. 12. M42: White to play.

P61. e2e8
e8g8*

But the rook on e8 is protected by the rook on c8 and the queen on d7. After the failure of P61, the program generates:

P62. d7 EQ → NEQ
e2e8
e8g8*

which gives, among other plans:

P63. d4g4
g4d7*
d7 NEQ
e2e8
e8g8*

If the black queen captures the white one, White executes the following steps of P63 and wins. Black can play his queen on squares where she controls e8 and where she cannot be captured. So it plays after 1·Qd4g4, Qd7b5. When White tries again 2·Re2×e8, the queen on b5 appears to be dangerous. After 1·Qd4g4—Qd7b5, it generates:

P64. b5 EQ → NEQ
e2e8
e8g8*

which gives

P65. g4c4
c4b5*
b5 NEQ
e2e8
e8g8*

If the black queen captures the white one, White wins in executing the following statements of P65.

Finally the program finds the main variation

1 Qd4g4 Qd7b5
2 Qg4c4 Qb5d7
3 Qc4c7 Qd7b5
4 a2a4 Qb5 × a4
5 Re2e4 Qa4b5
6 Qc7 × b7

either the black queen is captured or it loses the control of square e8. In that case the plan

P66. e4e8
e8g8*

can be executed and White wins.

There are many other variations. For instance

5 Re2e4 Re8 × e4
6 Qc7 × c8+ Re4e8
7 Qc8 × e8+

It is necessary to consider many other black moves. For instance after 3 Qc4c7, Qd7a4. Also White has many possibilities for threatening the black queen. Instead of 4 a2a4, it may play 4 Qc7 × b7 or instead of 5 Re2e4, 5 b2b3. It is not easy to see that these moves are not the good ones, and the program wastes much time in examining their consequences.

The program has a problem more difficult than the problem solved by the author of this combination. It is easy to see that, after 1·Qd4g4, White does not lose a man, Black can only play Qd7b5. This is sufficient for playing the move. But it is not so easy to be sure that, in the initial position, White wins something whatever Black plays. A player can play a move if it has proved that he wins a piece for some opponent's replies and that he loses nothing for the other replies. On the other hand, the program must prove that it wins a piece for all the opponent's replies.

We could give examples of combinations found by the program, although champions such as Smyslov, Euwe, Tshigorine, Reshevsky, Ed. Lasker, Pillsbury do not see them. But it is not very convincing because:

—even a grandmaster can play a bad move, especially in case of time pressure. It was the case of E42 (Fig. 2): master Prins did not see the combination because he had to play 12 moves in a few minutes.

— a program can always find anything if its programmer knows what it has to find.

Therefore we will see some other examples, of perhaps easier combinations, but where the solution found is different from the one given in the book. We did not know the solution found by the program.

— M9 (Fig. 5). The author of the book indicates:

1	Qh1e1+	Kd2c2
2	Qe1c1+	Kc2b3
3	Qc1b2+	Kb3c4
4	Qb2b4+	Kc4d5
5	Qb4d6+	Kd5c4
6	Qd6c5+	Kc4b3
7	Qc5b4+	Kb3c2
8	Qb4b2+	Kc2 × b2
9	Nf2 × d3+	K plays
10	Nd3 × f4	

The program finds a shorter combination; it wins a queen instead of a knight from the position obtained after the first eight ply.

The first four moves of the program are the same.

5	Qb4c5+	Kd5e6
6	Qc5d6+	Ke6f7 (if Ke6f5, Qd6f6≠)
7	e5e6+	K plays
8	Qd6 × f4	

There is a position very similar to M9 given by Le Lionnais [11]. We call it L180. There is a white pawn on f6 and the square g5 is empty. In this case the second combination is not possible and the program finds exactly the combination given by du Mont.

The program does not see that it is always doing the same maneuver when it finds the combination for L180. As the black king always moves, it makes a limited analysis on the new position of the king. It sees that a double attack would be possible if the enemy king moved to some square and it brings its queen on this square, prompting the king to capture her.

— M75 (Fig. 13).

The author gives a combination which seems good, beginning with 1 Re3 × e4— followed by 2 f5f6+.

But he says that the fork 1 f5f6+ is not possible because the black knight in e4 prevents it. However the program finds a combination beginning with the fork.

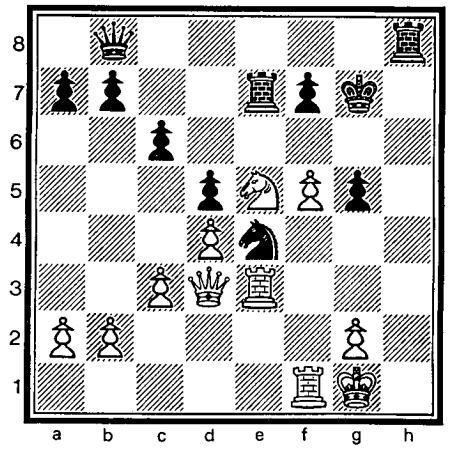

FIG. 13. M75: White to play.

In reality, during the experimentations of the program, it found two such combinations. We give here the more simple.

The initial plan is the double attack

P67. f5f6
f6e7*
f6g7*

After 1·f5f6+, Black has the new capture move. Ne4 × f6

But, the initial analysis gave other plans than P67:

P68. e4 E → V
e5 F → V*
e3e7*

This plan comes from the observation that the rook e7 is not protected.

When the program tries the interferences after the dangerous move Ne4 × f6, it sees that the modification e4 E → V has been realized. So it tries after 1 f5f6 Ne4 × f6 the plan

P69. e5 F → V*
e3e7*

There are several ways of threatening with the knight e5. Particularly it may attack a square next to the enemy king. This gives:

P70. f6 NEK → EK
e5g4
g4f6*
e5 V
e3e7*

White can capture the man on square f6 for inducing the black king to move to this square

P71. f1f6
f6 EK
e5g4
g4f6*
e5 V
e3e7*

This plan gives the main variation; the black moves are new capture moves or destroy a move capturing his king.

1 f5f6+ Ne4×f6
2 Rf1×f6 Kg7×f6
3 Ne5g4+ Kf6g7
4 Re3×e7

It seems that the combination given in the book does not hold for several situations. The program finds two cases in Du Mont [5] and three in Tarrasch [16]. We give one of them.

T149 (Fig. 10). It is easy to find the moves given in the book

1 Be5×h2+
2 Kg1×h2 Qg5h4+
3 Kh2g1 Qh4×a4.

Black wins a pawn. This combination is based on the possibility of a double attack on the pawn h2 and the bishop a4. But we have seen earlier (2.4.4) that White sees the interference with the initial plan

P57. e5 E → V
e2e7
e7c7*
e7f8*

As after 3 . . . Qh4×a4, e5 is now empty, the program considers Qe2e7 with the hope of recapturing a pawn. This is a very dangerous move. It is not evident that the more important thing for Black is to protect at once the rook f8 and the pawn c7. But, if he wants to do it, he can only play Bc8d7. Then the program plays Rf1f4 threatening the queen. If the queen moves, the program considers Rf4h4 threatening the pawn h7 and the mate. There are many variations, but it seems likely that there is no combination for Black in the initial position.

3.2. The failures of the program

The program does not find 3 of the 184 combinations.

The first failure was for T169 (Fig. 9). The tree is too large and the program stopped because the available storage was not sufficient. It had found the main

variations, but many moves may be considered if the sacrifice of the bishop is not accepted: after 1 . . . Bd6 × h2+ 2 the king moves to h1 or g2, the situation is very intricate. Three black men are en prise: the queen on b6, the bishop on h2 and the pawn on e4. White has many possibilities for trying to recapture at least a pawn. It seems that there is really a combination where Black wins a pawn, but it is necessary to develop a very large tree if we want to be sure of this.

The two other failures have another reason. Let us see one of them: T148 (Fig. 14)

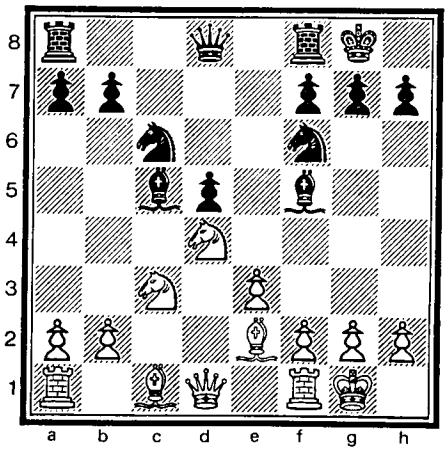

FIG. 14. T148: Black to play.

The program finds easily

1		Bc5 × d4
2	e3 × d4	Qd8b6

and sees that the pawn on d4 cannot be protected. If 3 Bc1e3, the pawn on b2 is captured.

But White tries to capture the pawn on d5 and considers

3	Be2f3	Nc6 × d4
4	Nc3 × d5	Nf6 × d5
5	Bf3 × d5	

As the balance is zero, the program estimates that there is no combination for Black. But it has not seen the black move 5 . . ., Ra8d8 followed by Nd4c2, if the bishop on d5 moves. This last move threatens the rook on a1 and the queen on d1.

This is not a hopeless failure. The combination beginning with Ra8d8 succeeds because the white bishop on d5 is not protected. But white has just moved this bishop. So, a small modification of the limited analysis, which bares precisely on the square d5, would give the combination. I do not implement this because it is not good to modify a program for a failure which occurs only once.

The correction of the third failure would require also a slight modification of the limited analysis.

In the three cases, the failure came from a variation which was not considered in the book.

3.3. Some statistics

TABLE 1.

Position	*P*	*N*	*NR*	*DR*	*MER*	*F*	*FL*	*T*	*r*
M39	5290	2157	135	11	17	16	47	24.1	6.26
E42	769	305	35	7	8	12	35	2.73	11.48
M58	8437	2985	293	19	12	10	56	32.5	9.82
T117	529	218	69	7	12	7	36	2.91	31.65
M9	6474	3277	180	19	8	11	22	28.3	5.49
E2	12766	4800	790	17	26	17	47	71.4	16.45
M3	3173	1810	76	9	12	14	35	11.6	4.20
B5	17619	6439	117	21	4	5	44	68.8	1.82
M42	59748	22223	2454	23	26	5	44	448.9	11.04
M75	6511	2946	159	19	21	8	38	25.7	5.40
L180	16536	8110	189	19	8	13	22	74.7	2.33
INIT	9	0	0	0	0	0	20	0.42	—

The positions are given in the order where they appear in the paper. L180 is M9 (Fig. 5) with a white pawn on f6 instead of g5. INIT is the initial position in chess. Naturally there is no combination! This position was given to show that the program wastes no time when it is evident that there is nothing to find. It generates plans such as:

P72. a2 F → V*
a7 E → V
a1a8*

This plan does not generate a move: it is impossible to threaten some black man with the pawn a2. The computer time is mainly used for initializing the list storage.

P is the number of plans generated by both players.

N is the number of nodes of the tree. We have always $P \geqslant N$, since a node is generated by at least one plan, but a plan does not generate always a node, and several plans may generate the same node.

NR is the number of nodes of the "reduced" tree: When the first player plays, we keep only the winning move, and we keep all his opponent's moves after this move. *NR* indicates the minimum number of nodes necessary for justifying the combination.

DR is the maximum depth (in ply) of the reduced tree. If, in some variation, there is a mate, we count the move capturing the enemy king. This depth is often greater than the depth of the main variation. The maximum depth may occur for another variation. The program may also generate moves after the last move of

the main variation: it has to be sure that its opponent cannot recapture some friendly man.

MER is the maximum number of enemy moves considered after a friendly move in the reduced tree. This maximum may be reached at any level. For M42, 26 enemy moves are considered after a friendly move at depth 11. When there are many enemy moves, the greatest part is generally quickly refuted. For instance for E2 (Fig. 6), after 1 Rel × e6 the program considers 26 black moves (Black has 35 legal moves). But for 15 of them the tree following them has only one move.

F is the number of friendly moves considered at the first level of the tree.

FL is the number of legal moves in the initial position. So $F \leqslant FL$.

T is the C.P.U. time in seconds of IBM 370–168. The times given are large, because we give some difficult combinations. Generally they are smaller than 5 seconds.

r is the ratio $(100*NR)/N$. We have obviously $r \leqslant 100$. If $r = 10$, it does not mean that only 10% of the generated moves are well chosen. If a friendly move *M* is not very good, all the moves of the subtree following *R* are counted in *N* and not in *NR*. However many moves of this subtree are well chosen. It is also necessary to generate a friendly move which is not the best one for finding what goes wrong after this move. In generating another friendly move, the program corrects what was bad.

Thus it is normal to get values of *r* which are small.

4. Conclusion

4.1. Comparison with chess playing program

This program was not made for playing chess. There are four main things to change if we want to include it in a chess program.

In a chess program, it is not possible to give the wanted value of a combination. We could use some method such as the one described by Berliner [3] with his pessimistic values.

It would be necessary to program the complete rules of chess, including castling, capture en passant, stalemate. These rules complicate the program and are rarely useful in combinations.

In the analysis of the positions, we deliberately removed the possibilities of threatening mate and the attack of men (except king and pawns) with few moves. It is not a question of computer time, the analysis is done only once and if there is no possibility of threatening mate, no move is generated. The difficulty is to program the way of getting such plans and to experiment the program. It is possible to do this work, but another year is needed for achieving this.

The number of combinations with which the program has been tried is not sufficient. It would be necessary to modify some parts of the program, for instance the method for generating plans of double attacks. There are certainly very simple

combinations that the program would not find. But it is a very lengthy work to achieve this.

We think that a program of a grandmaster strength is possible in starting from this program and from the program developed by Berliner. But such a program would be very large, for instance 50 000 Fortran instructions. And it would be necessary to experiment with it for a long time.

Finding a combination would be the most important part of this program. This is the main reason for which programs develop large trees. For positional playing, methods presented by Gillogly [9] seem interesting. But it is also necessary to use the positional interest of a position in evaluating the tree.

It would be also necessary to find plans for endgames. These plans are rather different; all the moves will not be described, but only the main steps: try to promote pawns P_1 or P_2, for that the king is brought near them ... A very difficult study would be necessary.

It is natural that programs like Berliner's are beaten by programs which develop very large trees. It is difficult to write a program which reasons; this program must be a very big one, and it is sufficient that a small part is missing for playing a bad game. It is likely that, in developing the tree a part of the program, where there is a bug or where the chess analysis is not well made, is used. In consequence a very bad move may be chosen.

But it is the only way, in our opinion, for realizing a program which plays very well. It is unlikely that a simple program could do this. Chess is a very complicated game and very complicated programs are necessary.

4.2. Learning

As the program will be very complicated it would be interesting to write a learning program. The main difficulty is to find how we can analyze a position for finding

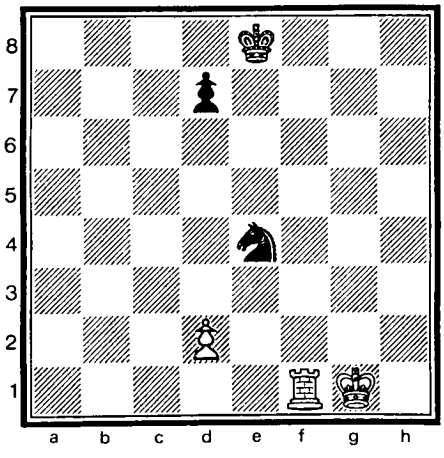

FIG. 15. T99: White to play.

good plans. In Pitrat [12, 1, 17], we described a program which could learn how to find plans for combinations. It learned when it received the moves of a game really played. It understood the combinations which occurred, and even those which were tried, but failed. It understood also how a combination had been refuted.

The program generalized the combination and generated a pattern: if some conditions are realized, try the following plan.

For instance, receiving for T99 (Fig. 15), the moves

1 Rf1e1 d7d5
2 d2d3 Ke8f8
3 d3×e4

the program learned the three following patterns:

(1) If the enemy king and a man of value greater or equal than the knight are on the same file or rank, consider the plan: try to bring a rook on the line, then capture the enemy man, or, if it moves, capture the king.

(2) If a friendly knight is pinned on the king by a rook, try to protect the knight with a pawn.

(3) If an enemy knight (or a man of greater value) is on the same file or rank than the enemy king, if our rook is on this line and if the knight is protected, attack it with a pawn.

The first two patterns do not succeed in the real game. However the program stores them. If there were no pawn on d2, the second one would also be good.

Such a program would certainly be useful. But the patterns found by this learning program were not as good as those that we found. There is still much work to do for experimenting learning programs.

4.3. Application to other areas than chess

The program is doing a very sophisticated analysis of the initial situation. It generates plans and executes them. If everything is good, it does not analyse the intermediary situations. If something goes wrong, it analyses what is wrong. It tries to correct it in several steps, beginning with the methods which have the greatest chance of success.

This method is general, and useful in other areas than chess. There are two main reasons for its success.

At each level of the tree, the program considers only natural branches. When it knows what happens after some node, other nodes at the same level become natural, because they correct what is wrong. For instance in games, a sacrifice is not natural, and must not be considered first. But if it sees in developing some other node, that an enemy man is troublesome, it considers a sacrifice which removes this man. The tree is not very large. This seems specially interesting for solving practical problems, where many actions are possible.

For many nodes, the program performs a very simple analysis, and even in some cases, no analysis. It has to do a complete analysis only in the initial situation. Later, the analysis is always directed toward a well-defined goal. It may be done very seriously, since it is limited.

REFERENCES

1. Baylor, G. W. and Simon, H. A., A chess mating combinations program, *Spring Joint Computer Conference* (1966) 431–447.
2. Berliner, H. J., Some necessary conditions for a master chess program. *Third Int. Joint Conf. Artificial Intelligence* (1973), 77–85.
3. Berliner, H. J., Chess as problem solving: the development of a tactics analyzer, Ph.D. Thesis, Carnegie Mellon University (March 1974).
4. De Groot, A. D., *Thought and choice in chess* (Mouton, The Hague, 1965).
5. Du Mont, J., *Les bases de la combinaison aux échecs* (Payot, Paris, 1970). In English: *The basis of combinaison in chess* (Routledge and Keagan Paul Ltd., London).
6. Euwe, M., *Les échecs. Position et combinaison* (Payot, Paris, 1966).
7. Fahlman, S. E., A planning system for robot construction tasks, *Artificial Intelligence* **1** (1974), 1–49.
8. Fikes, R. E., Hart, P. E. and Nilsson, N. J., Learning and executing generalized robot plans, *Artificial Intelligence* **4** (1972), 251–288.
9. Gillogly, J. J., The technology chess program, *Artificial Intelligence* **3** (1972), 145–163.
10. Greenblatt, R. D., Eastlake, D. E. and Crocker, S. D., The Greenblatt chess program, *Fall Joint Computer Conference* (1967) 801–810.
11. Le Lionnais, F., Maget, E., *Dictionnaire des échecs* (Presses Universitai de France, Paris, 1967).
12. Pitrat, J., Realization of a program learning to find combinations at chess, in: Simon, J. C., ed., *Computer oriented learning processes* (Noordhoff, Leyden, 1976) 397–424.
13. Pitrat, J., A general game playing program, in: Findler, N. V. and Meltzer, B. Eds., *Artificial Intelligence and heuristic programming* (Edinburgh University Press, Edinburgh (1971)), 125–155.
14. Sacerdoti, E. D., Planning in a hierarchy of abstraction spaces, *Artificial Intelligence* **2** (1974), 115–135.
15. Siklossy, L., Dreussi, J., A hierarchy driven robot planner which generates its own procedures, Report TR-6, University of Texas at Austin (February 1973).
16. Tarrasch, S., *Traité pratique du jeu d'échecs* (Payot, Paris, 1965).
17. Pitrat, J., A program for learning to play chess, in: Chen, ed., *Pattern recognition and artificial intelligence* (Academic Press, New York, 1976) 399–419.

Received March 1976; revised version received August, 1976