Offprint from

Artificial Intelligence and Heuristic Programming

Edinburgh University Press

1971

A General Game-Playing Program

J. PITRAT

INTRODUCTION

A general game-playing program must know the rules of the particular playing game. These rules are:
(1) an algorithm indicating the winning state;
(2) an algorithm enumerating legal moves. A move gives a set of changes from the present situation.
There are two means of giving these rules:
(1) We can write a subroutine which recognizes if we have won and another which enumerates legal moves. Such a subroutine is a black box giving to the calling program the answer: 'you win' or 'you do not win', or the list of legal moves. But it cannot know what is in that subroutine.
(2) We can also define a language in which we describe the rules of a game. The program investigates the rules written with this language and finds some indications to improve its play.

With this method the program can find the different possible kinds of moves; for instance the possibilities of pawn promotion at chess. It can find also that, at chess, pawns always move forward; that in all cases, except capture *en passant*, the capturing piece goes to the square of the captured piece; that we can castle at most once, and so on.

With the first description of the rules, we observe some of these facts, but we cannot be sure that they are laws. We can wait a long time before observing a promotion at chess. It does not occur very often in a play, although this possibility has a great importance for the players' strategy.

We have already seen that with the second method of description, the program can find some important characteristics of the game before the first play. Another aspect of the second method is that it is more convenient to write the rules in a special language than in a general programming language.

LANGUAGE USED TO DESCRIBE THE RULES OF A GAME

We choose the second method. The language is suitable to describe games on a rectangular board. But it can describe other games. It has some limitations and cannot describe any game, even on a board.

My aim is not to define the language accurately, but to give a quick idea of it.

The initial board is read at the beginning of each play. It is not described by the rules.

Variables. They are a sequence of alphanumeric characters. They have two characteristics.

(1) Their value may be an integer or a pair of integers. In the second case, the value of the variable can represent a square of the board. The value of the first element of the pair is the abscissa of the square, the value of the second, the ordinate. We can use the operations + and − with the variables.

If V is (3, 2) and HP is (0, 1) the value of 'V + HP' is (3, 3). We can multiply a variable by an integer. If k is 2, (k * V) is (6, 4).

We can apply four functions to these variables, the value of the functions being an integer.

(a) OCCUPY(V) – 3 values are possible: empty, friend, enemy.

(b) NATURE(V) – The value of the function is the type of the piece which occupies the square V. The function is undefined if the square is empty. At chess, there may be six values: *King, Queen, Rook, Bishop, Knight, Pawn.* For some games, like go-moku, only one value is possible.

(c) ABSCISSA(V) – Its value is the first coordinate of square V.

(d) ORDINATE(V) – Its value is the second coordinate.

The program finds the type of the variables with the context. There are no constants, but the values of a variable may be defined at the beginning.

(2) Variables have another characteristic, but only in the case where their value is given initially. Some have only one initial value (an integer or a pair of integers), others have two initial values: one of these values is for the first player, the other for the second player. These variables are useful for describing the rules for the two players with only one algorithm. For instance the value of FRIEND is 1 for the first player and 2 for the second. The value of CHP is (0, 1) for the first player and (0, −1) for the second. This is very useful to describe the pawn moves at chess: we add CHP to the value of the pawn square. We obtain the value of a square where the pawn can go if it is empty.

It is also useful to introduce the variable KA whose values are (6, 1) for the first player and (6, 8) for the second, and KB: (7, 1) or (7, 8). If we want to see if short castling is possible, we must check that OCCUPY(KA) and OCCUPY(KB) have the value EMPTY.

Statements. Each statement may begin by a label. When the label is optional, it is placed between parentheses in the description of the statement. We do not describe statements in BNF., but we give the general form of each statement.

(1) ARITHMETIC.

$$(\textit{Label}).\ \text{ARITHMETIC}*\begin{Bmatrix}\textit{variable}\\ \textit{pseudo function}\end{Bmatrix}=\textit{arithmetic expression.}$$

The operators may be: $+$, $-$, $*$ in the arithmetic expression. We may also use the four functions. Their value is an integer or a pair of integers. The variable of the first statement must be of the same type as the result of the arithmetic expression.

A pseudo function may be:

ABSCISSA(V)

ORDINATE(V)

where V is a variable, its value is a pair of integers.

In that case the value of the arithmetic expression must be an integer. The effect of the statement is to give the value of the expression to the first or the second element of the pair V.

After execution of an arithmetic statement, the following statement is executed.

(2) Unconditional GO TO.

.GO TO*L.

The next statement to be executed is the statement labelled L.

(3) Computed GO TO.

(*Label*) . CPTD GO TO * *arithmetic expression* $=\text{I}_1,\ \text{L}_1 . \text{I}_2,\ \text{L}_2 \ldots \text{I}_n,\ \text{L}_n.$

I_j are variables, L_j are labels. We compute the arithmetic expression. If its value is I_k, the next statement to be executed is the statement labelled L_k.
For example:

.CPTD GO TO*NATURE(SQ)=KING, TA. QUEEN, TB. ROOK, TC.BISHOP, TD.KNIGHT, TE.PAWN, TF.

If the square SQ is occupied by a bishop, the next statement will be the statement labelled TD.

(4) IF.

(*Label*) . IF**arithmetic expression* 1 *relation arithmetic expression* 2. L1, L2.

L1 and L2 are labels. The relation may be: $=$, $\neq$, $>$, $\geqslant$, $<$, $\leqslant$. The arithmetic expressions are computed. If the relation is true, the next statement is L1, if false, L2.

(5) END.

It indicates the static end of the algorithm: it is the last statement.

(6) FINISHED.

(*Label*) . FINISHED *

It indicates the dynamical end of the algorithm. The execution stops if we go to this statement.

(7) SCANNING.

L1 . SCANNING * V . L2.

V is a variable; its value is a pair of integers.

We define an auxiliary label by concatenation of the letter X with the symbols of L1. We represent concatenation by ||. If L1 is AB, X||L1 is XAB. We can call this new label elsewhere in the algorithm.

The result of this statement is given by figure 1.

With this statement we may scan the board. A call to L1 initializes the scanning, and the first value of V is (1, 1). At each call to X||L1, we give

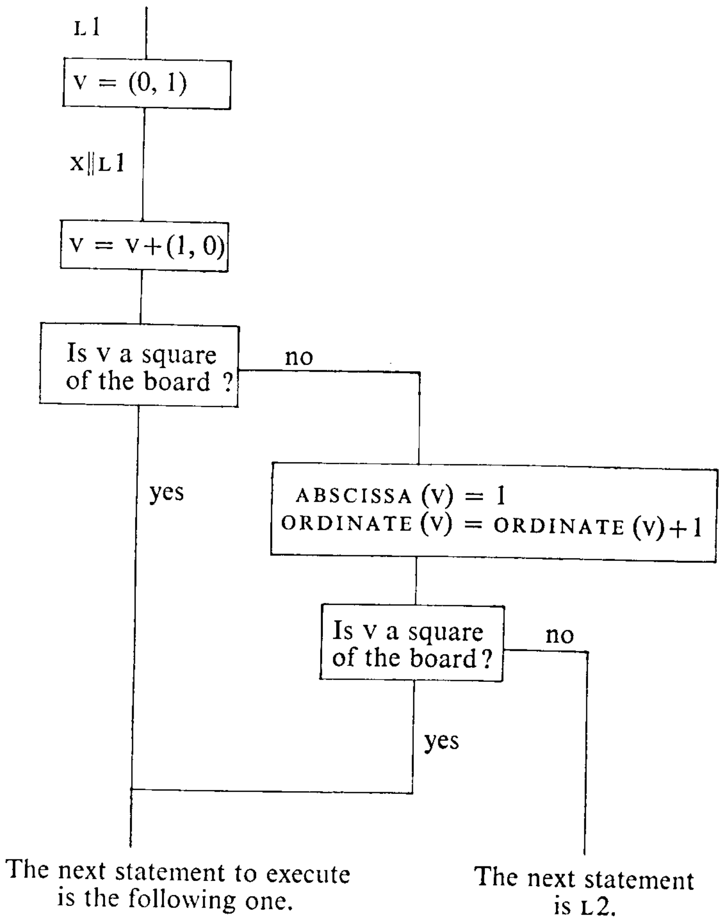

L1
V = (0, 1)
X||L1
V = V+(1, 0)
Is V a square of the board ?
no
yes
ABSCISSA (V) = 1
ORDINATE (V) = ORDINATE (V)+1
Is V a square of the board?
no
yes
The next statement to execute is the following one.
The next statement is L2.

Figure 1. Flowchart of the scanning statement

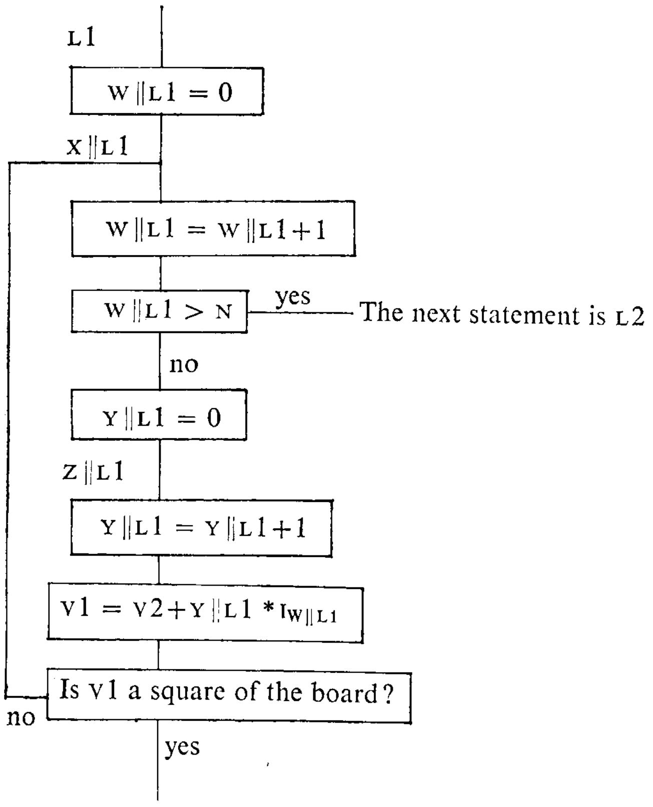
to V the value of the next square. When the scanning is completed, we go to L2.

(8) LOOP.

L1 . LOOP * V1 = V2 . $I_1, I_2, \ldots I_N$. L2. $\quad N > 0$

V1, V2, $I_1, I_2, \ldots I_N$ are variables. Their value is a pair of integers.

We define two auxiliary labels: X‖L1 and Z‖L1 and two auxiliary variables: W‖L1 and Y‖L1. We can use these labels and these variables in the algorithm. Their meaning and the result of the statement is given by figure 2.



The next statement to execute is the following one.

Figure 2. Flowchart of the loop statement

If we transfer to L1, we initialize the loop, and we give to V1 the value of V2 plus the value of I_1. If we transfer to Z‖L1, we add once more the same increment I_j: we make one step more in the same direction. If we are off the board, we add to V2 the next increment I_{j+1}.

If we transfer to X‖L1, we add to V2 the next increment. We make one step in a new direction. If we have used the N increments, the next statement is L2.

The value of W||L1 shows which increment has to be used and the value of Y||L1 how many steps we have taken in one direction.

(9) RESULT.

(*Label*). RESULT * *Nature of the result.*

The nature of the result may be:

victory
no victory
draw
loss
...

This statement is used one or several times in the algorithm indicating the winning state: No victory is not necessarily a loss: that is, we have not yet won, but we do not lose.

Then, we execute the next statement.

(10) MOVE.

(*Label*). MOVE * *Submove* 1 *Submove* 2 ... *Submove* N   N > 0

There are five kinds of submoves.

V, V1, V2 are variables, their value is a pair of integers, P is a variable, its value is an integer.

(1) TRAVEL, V1, V2. The piece in square V1 goes to square V2. If V1 is empty or if V2 is not empty, there is an error.

(2) CAPTURE, V. The piece in square V is captured. If there is not an enemy piece in V, there is an error.

(3) PUT, P, V. The player puts a piece of type P in square V. If V is not empty, there is an error.

(4) PROMOTION, P, V. The friendly piece in square V gets a new type P. If there is not a friendly piece in V, there is an error.

(5) INDIC, M, P. It is often necessary to use 'indicatifs' to describe the game when the knowledge of the board is not sufficient. For instance, in chess, if we want to castle or to capture *en passant*, we introduce three indicatifs:

(1) PASSANT Its value is 0 after a move, except when a pawn has been moved forward two squares. In that case the value of PASSANT is the abscissa of the pawn.

(2) CAS1 and CAS2 for short and long castling. Each indicatif has two values, one for each player. For a player the value of CAS1 is 0 if short castling is still possible, because there has been no move of the King or of the Rook in column 8. If not, its value is 1.

We have the possibility of changing their value in playing a move. If we move our King, we indicate in the MOVE statement that the values of CAS1 and CAS2 will be 1. We use the submoves: INDIC, CAS1, ONE.INDIC, CAS2, ONE. which give to the indicatifs CAS1 and CAS2 the value of the variable ONE which has been defined at the beginning as 1.

We shall give a part of the algorithm enumerating legal moves at chess, with some comments.

```
        A.SCANNING*V.M.
```

M is the label of the first statement of the part of the algorithm which examines if castling is possible. If we were playing a variety of chess where there is no castling, we would write the statement: M.FINISHED*

```
         .IF*OCCUPY(V)=FRIEND.B, XA.
```

If there is not a friendly piece in square V, we look at the following square.

```
        B.CPTD GO TO*NATURE(V)=KING, CK.QUEEN, CQ.ROOK,
       CR.BISHOP, CB.KNIGHT, CN.PAWN, CP.
```

We give the algorithm only for two cases: Queen and Knight.

```
       CQ.LOOP*W=V.HP, HN, VP, VN, BP1, BP2, BN1, BN2.XA.
```

We have a Queen in square V. The value of HP is (0, 1), HN: (0, −1). VP:(1, 0) . . . BN2:(−1, −1). When we know the moves of the Queen in the eight directions, we look at the following square.

```
         .CPTD GO TO*OCCUPY(W)=EMPTY, E.FRIEND, XCQ.
         ENEMY, F.
```

If there is a friendly piece in square W, we try the following direction.

```
        E.MOVE*TRAVEL, V, W.
         .GO TO*ZCQ.
```

The square W is empty. We go further in the same direction.

```
        F.MOVE*CAPTURE, W.TRAVEL, V, W.
```

When we execute a move statement, we output the submoves with the actual value of its variables. In other words, if V is (2, 2) and W is (7, 7), we output:

```
                 CAPTURE(7, 7).TRAVEL, (2, 2), (7, 7).
         .GO TO*XCQ.
```

There is an enemy piece in W. We cannot go further in this direction. We try the next direction.

```
       CN.LOOP*Y=V.OT, OMT, MOT, MOMT, TO, TMO, MTO, MTMO.XA.
```

We have a Knight in square V. The value of OT is (1, 2), OMT: (1, −2), MOT: (−1, 2) . . . MTMO: (−2, −1).

```
         .CPTD GO TO*OCCUPY(Y)=EMPTY, H.FRIEND, XCN.ENEMY, I.
```

If there is a friendly piece in square Y, there is no move.

```
        H.MOVE*TRAVEL, V, Y.
```

```
 . GO TO*XCN.
I . MOVE*CAPTURE, Y . MOVE, V, Y.
 . GO TO*XCN.
 . . .
```

We describe the moves of King (CK), Rook (CR), Bishop (CB), Pawn (CP) and castling (M).

```
 . END.
```

For chess, the difficulty of the algorithm is in the description of castling and of the pawn moves.

Conclusion. There are several ways of improving this language.

(1) It would be convenient to take the moves as values of a new type of variables and use statements to modify these variables. For instance, if Q and R are such variables, we could write:

Q = R + *submove.*

This is useful if we do not know at one time all the submoves of a move – for instance at checkers when we capture several Pawns.

(2) It would be useful to describe the connections between one move and the others – for instance, if the existence of a friendly move follows from the existence of enemy moves. In chess we cannot castle if there is an enemy move capturing our King. I gave this possibility to the language. I do not describe it here because it is not essential.

(3) It would be convenient to have subscript variables. If we could use them, we could describe the moves of any rider like Queen, Rook, Bishop . . . with one subroutine, and with another the moves of any jumper like Knight, King . . .

Remark. I first wrote an interpreter for the language, then a compiler. With the second solution, the execution is approximately 20 times faster. As we shall see later, it is more sophisticated than an ordinary compiler, it is more than a simple translator.

NECESSARY CONDITIONS FOR A MOVE TO OCCUR

Introduction. The algorithm enumerating legal moves shows, for each move, how this move modifies the board. For example, if the move has only one submove: TRAVEL, (3, 2), (4, 4)., then there are two changes on the board after this move:

(3, 2) becomes empty

(4, 4) is occupied by the piece which was in (3, 2).

This information is necessary for playing the move. But it is not sufficient if we want to have an efficient game playing program. The program must know why this move exists. For example, if there is an enemy move Q which is dangerous because some conditions c_1, c_2, . . . c_n are true on the board, and if the program knows these conditions, it chooses only from the moves which change at least one of these conditions. In the same way, it is useful to know that some interesting move would appear if there were some change on the board.

The knowledge of these necessary conditions for a move to occur is important, but it is not given by the algorithm. The program must find them in studying the algorithm.

Construction of the flowchart. The program constructs the flowchart of the algorithm. ARITHMETIC, MOVE and RESULT statements have one successor. The IF statement has two and COMPUTED GO TO has $N>0$. We replace SCANNING and LOOP statements by the equivalent flowchart.

Unconditional GO TO and END statements disappear: they are a consequence of the linear description of the algorithm.

We gather in one statement all the FINISHED statements of the algorithm.

Principle of the method that finds the conditions. Suppose that we have the flowchart of figure 3, '*a*' being the condition of an IF statement, for instance:

OCCUPY(BR)=EMPTY.

Here BR is a variable, its value is a square; B, C, D are move statements.

If we execute the IF statement, we are sure to execute the move statement D, whatever '*a*' is: true or false. So, for the move described by D, the condition '*a*' is not necessary. But it is necessary that '*a*' be true if we want to execute statement B. If the value of BR is (5, 4), it is necessary that (5, 4) be empty. It is one of the necessary conditions for the move described by B. In the same way, it is necessary that '*a*' be false if we want to execute C.

We can see the method used: in a first step, which will be made once when we read the algorithm, we determine the 'conjugué' of each transfer

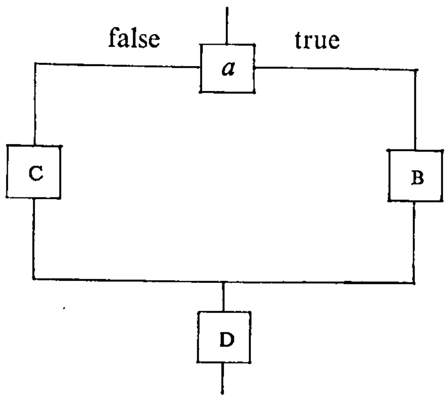


Figure 3. Flowchart showing the principle of the method that finds the conditions

statement: IF and COMPUTED GO TO. The 'conjugué' of the statement S is the first statement that we are sure to execute if we execute S, whatever path we take after S.

In a second step, when we execute the algorithm, at each transfer statement we store the condition used with the name of the conjugué. We erase this condition when we execute the conjugué. At each move statement, we output the conditions stored at this moment: these are the necessary conditions for the move to occur.

We shall develop these two points.

Determination of conjugués. The 'conjugué' of a transfer statement S (IF, COMPUTED GO TO) is the first statement T which we are sure to execute if we execute S, whatever path we take after S. There is always a conjugué: we have only one dynamical end, and we are sure to execute it if we have a correctly written algorithm.

We determine the conjugué for each transfer statement once after reading the algorithm. This search does not use the values of the variables and is the only function of the flowchart. This is done in two steps.

First step. We look for a path from the transfer statement S to the dynamical end of the algorithm. We get this path by looking at all the branches of the transfer statements encountered, till we obtain the dynamical end. When we arrive at a transfer statement, we develop one branch and we put the other unexplored branches in a push-down.

To avoid loops, we put a mark on the transfer statements already explored. If we find a statement with a mark, we stop developing this branch and we take another unexplored branch in the push-down.

We are sure to get such a path, because from any point of an algorithm, there is always one possibility, at least, to go to the end. If not, there is a programming error in the algorithm.

By this process we have a succession of statements (not all transfer statements):

$$S, S_1, S_2, \dots S_n.$$

the last statement S_n being the dynamical end. S_{i+1} is one of the possible successors of S_i. There may be several paths between S and the dynamical end, but this is not important, we only want to get one of them.

Second step. The 'conjugué' is on every path between S and the dynamical end, it is one of the statements of the path found by us: it is one of the S_i, $i=1, \dots n$. The conjugué of the statement S is the first S_i that we are sure to execute after S.

So we consider all the S_i, beginning with S_1 until we get the conjugué. We want to see if s_i belongs to all the paths issued from s. If such is the case, s_i is the conjugué:

We are sure to execute this statement after s.

It is the first statement with this property, because this property does not hold for $s_1, s_2, \ldots s_{i-1}$.

To find if all the paths issued from s run through s_i, we put at the beginning a mark on s_i, and we use the same method as in the first step: we try to go from s to the dynamical end, but in this case we cannot cross s_i which has a mark and is taken as already explored. If we can find a path, s_i is not the conjugué: a path exists where we do not execute s_i. We resume the procedure with s_{i+1}. But if there is no path, s_i is the conjugué: in every path from s to the dynamical end we execute s_i and it is the first statement with this property.

By this method, we associate with each transfer statement another statement (not always a transfer statement) called its 'conjugué'. The conjugué may be the dynamical end.

This is done only once, after reading the rules of a game. For example, in the chess algorithm, we see by this method that the conjugué of the computed GOTO following CQ is XCQ. This is trivial: if OCCUPY(W) is *enemy*, we go to F, then to XCQ. If it is *empty*, we go to E then to ZCQ, then to XCQ or to the preceding COMPUTED GOTO seen above (we replaced the loop by the equivalent flowchart). If it is *friend* we go to XCQ.

How to find the conditions. When, during the execution of an algorithm, we meet a COMPUTED GO TO or an IF statement, we put the condition associated with this statement in a push-down, with the name of the conjugué of the statement. In the condition we put the value of the variables. For example, if the condition of an IF statement is OCCUPY(BR)=EMPTY and if the value of BR is (4, 5) when we execute this statement, we put in the push-down:

OCCUPY(4, 5)=EMPTY.

if the condition is true. If the condition is false, for instance if (4, 5) is *friend*, we put:

OCCUPY(4, 5)=FRIEND.

When we execute a new statement S (not only a transfer statement) we see if it is the conjugué of conditions in the push-down (they are necessarily at the top). If such is the case, we erase these conditions. They are no longer necessary: we execute the statement S and the following one even if these conditions are false. As the statement may be the conjugué of several transfer statements, we can erase several conditions at the same time.

When we meet a result statement or a move statement, we output the conditions in the push-down with the description of the result or of the move. Thus we get the necessary conditions for this move or for this result.
For example: If we are at chess and we have:

White Rook in (3, 1)
White Bishop in (1, 2)
Black Knight in (4, 5).

The move: 'Bishop captures Knight' has with this method the conditions:

OCCUPY(1, 2) = FRIEND
NATURE(1, 2) = BISHOP
OCCUPY(2, 3) = EMPTY
OCCUPY(3, 4) = EMPTY
OCCUPY(4, 5) = ENEMY.

This move changes two squares:

OCCUPY(4, 5) becomes FRIEND
NATURE(4, 5) becomes BISHOP
OCCUPY(1, 2) becomes EMPTY.

The move: 'Rook goes to (1, 1)' has the conditions:

OCCUPY(3, 1) = FRIEND
NATURE(3, 1) = ROOK
OCCUPY(2, 1) = EMPTY
OCCUPY(1, 1) = EMPTY.

The changes are:

OCCUPY(3, 1) becomes EMPTY
OCCUPY(1, 1) becomes FRIEND
NATURE(1, 1) becomes ROOK.

Actually we do not write all the conditions in the push-down. We are interested only in the conditions which can be changed when we play a move. The value of a loop counter has no interest, the player can do nothing with it. Also we only keep conditions which specify the occupation or the nature of a square or the value of an indicatif. When we (or the opponent) play, we can change these conditions and destroy the move or the result.
Remark. This method is simple, but there are some cases where we have difficulties.
(1) The algorithm is not well written; some transfer statements are not necessary. In that case the program can consider as necessary some conditions which are not. It is difficult to write a program which gives a better algorithm! This error is not frequent and it is not very important to have more conditions than necessary.

(2) There is another case where the method fails, and where we forget conditions, which is more unpleasant. This case occurs when the value of a variable is not the same if we take different paths and if this variable is used in a transfer statement. The value of the variable stores the result of a preceding condition.

For example, let us consider figure 4. '*a*' is a condition. The second test: 'A = 0' is equivalent to testing whether '*a*' is true. With our method we say that after statement S the condition '*a*' is no longer necessary. So we miss a condition.

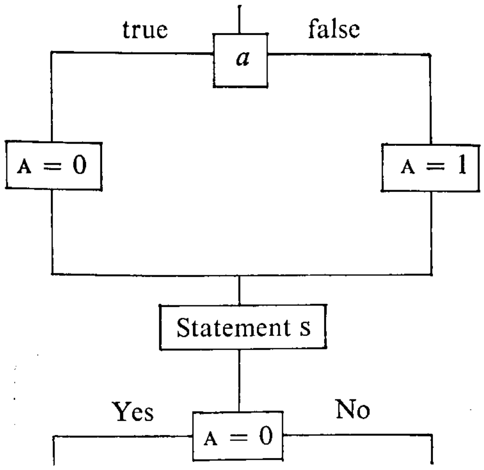


Figure 4. Flowchart showing an objection to the preceding method

But we can improve the method, and remove this difficulty. We associate with each variable a list of conditions, by the following method:
(a) initially this list is empty;
(b) when we define the value of a variable (by an arithmetic statement) we put in the associated list the conditions in the push-down and the conditions of the lists of the variables used to define the new value. We put each condition only once. But we erase the conditions associated with this variable before the statement.

At each transfer statement we add to the push-down the conditions which are in the lists associated with the variables used in the statement. The conjugué associated with all these conditions is the conjugué of the transfer statement.

In the example given above, when we execute the ARITHMETIC statement 'A = 0' we associate '*a true*' which is in the push-down at this time. Statement S erases '*a true*' from the push-down. But when we test 'A = 0', we put in the push-down (if A has not been defined again) the conditions associated with the variable A: we will have again the condition '*a true*'.

With this method we do not miss conditions, but we often have unnecessary conditions.

I have not written this new program, because the case of a variable which memorizes the result of a transfer statement does not occur in the games studied. There are not many ARITHMETIC statements in the algorithms, and generally their result is quickly used.

*Modifications.** It is important to see what moves could exist if there were one or several changes on the board. We could perform this change and execute the algorithm again. But this method wastes computer time. And we often do not know what modifications it is interesting to examine. So, I prefer to generate the 'moves which could be possible' at the same time as legal moves. We must improve the preceding method.

Let '*a*' be the condition of a transfer statement S. Suppose that this condition bears on the occupation or the nature of a square. In a first step we execute the algorithm normally: the next statement will be the statement designated by the real value of the condition '*a*'. But we keep:

the value of the variables;

the conjugué of the transfer statement S;

the other statements issuing from the transfer statement S;

the values taken by the condition '*a*' to get these other statements.

For example, suppose S is:

```
        AB . CPTD GO TO *OCCUPY(BR)=FRIEND, BC . ENEMY,
        BD . EMPTY, BE.
```

Suppose that the value of BR is (3, 2) and that there is an enemy man in (3, 2). We put in the push-down the condition: 'OCCUPY(3, 2)=ENEMY' and we execute BD. But we also store that if: 'OCCUPY(3, 2)=FRIEND' we must go to BC and if 'OCCUPY(3, 2)=EMPTY' we must go to BE.

When the normal execution is completed, we restore these situations. We put in the push-down the conditions not fulfilled, with a special indication, and we execute the algorithm until we are at the conjugué of the transfer statement S. The following statements have already been executed during the normal execution. If we meet a move or a result statement, we output the push-down with indication of the conditions not fulfilled.

If, during such an execution, we meet a new transfer statement, we can carry out the same process again. We can generate moves which require *n*

*This word is better than forcing, used in Pitrat (1968), which is confusing.

modifications on the board. We can stop this process if $n > N$, N being an important parameter of the program.

With this method we have the moves which would exist if there were one or several modifications on the board. It is not necessary to execute all the algorithm for each modification.

Example. In a game of chess there is a friendly Rook in (8, 5), a friendly Bishop in (7, 4), an enemy Bishop in (8, 6), an enemy Queen in (8, 7) and an enemy King in (8, 8). Among the moves with modifications, we shall get:

Rook in (8, 5) *captures* (8, 7).

Conditions: friendly Rook in (8, 5)
(8, 6) becomes empty
(8, 7) enemy.

The second condition is not fulfilled.

Rook in (8, 5) *captures* (8, 8).

Conditions: friendly Rook in (8, 5)
(8, 6) becomes empty
(8, 7) becomes empty.
(8, 8) enemy.

The second and the third conditions are not fulfilled.

Bishop in (7, 4) *captures* (8, 5).

Conditions: friendly Bishop in (7, 4)
enemy in (8, 5).

The second condition is not fulfilled.

Example. In a game of Go moku, (6, 5), (6, 6) and (6, 8) are empty, and there is a friendly piece in (6, 7) and (6, 9). The program will see that there is a possibility of winning with three modifications:

(6, 5) becomes FRIEND
(6, 6) becomes FRIEND
(6, 7) is FRIEND
(6, 8) becomes FRIEND
(6, 9) is FRIEND.

The program applies this process only to transfer statements of which the condition is on the occupation or the nature of a square, or the value of an indicatif: a move can change such conditions and we can hope that the move would become legal.

Applications. It is very useful to know what are the necessary conditions for a move to exist and the moves which could exist if there were some modifications. I used this in positional playing, and I am writing a general game-playing program which is able to learn combinations. I shall give one application to the search for a win.

THE SEARCH FOR A WIN

Introduction. In some games like chess, tic-tac-toe, Go moku . . . we are very often near winning. If the opponent did not play to avoid this threat, we should win. The reason is that there are only a few conditions to realize or to destroy which prevent us from winning. We can expect to win with a small number of moves.

In other games, like checkers, there is no danger of an immediate loss during a great part of the play. This danger occurs only at the end. In these games, we may be sure from the beginning that we will lose; for instance, if the opponent has twelve men and we have only eight. But there will be generally many moves before he will win.

The following method is possible for any game, but it is, in practice, useful only for the first class of games.

Blockade and threat. In any case we can always try to see if we can win whatever the opponent plays. We expand the tree of the moves and we see if there is a sequence of moves which leads to a winning state for all the opponent's replies.

But generally the expansion of the tree is so large that it is not practically possible, except for games where there is always a small number of legal moves. However it is sometimes possible when we manage to reduce drastically the number of the opponent's replies. There are two methods for this: (1) *Blockade* and (2) *Threat.*

(1) *Blockade*

We try to decrease the number of the opponent's legal moves. This is useful in games with a high number of moves when, for some reason, this number can decrease – for example, if one must play a capture when there is such a move. If we play a move which creates a capture for the opponent, he will have only one legal move. This is interesting if this move is bad for him.

We can also create situations where all the opponent's possible moves are bad for him. We use the obligation of playing, and the opponent has only a few moves to play, because the others have bad consequences for him. In many games a player must play when it is his turn.

Example. Tic-tac-toe (*see* figure 5). If we play: man in (3, 3) goes to (2, 3) this move is not dangerous. But the opponent must play. He must move the center pawn and he will lose.

This situation is infrequent in chess (we can see some in plays by Nim-

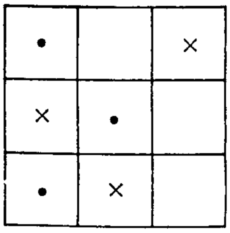


Figure 5. Blockade in tic–tac–toe

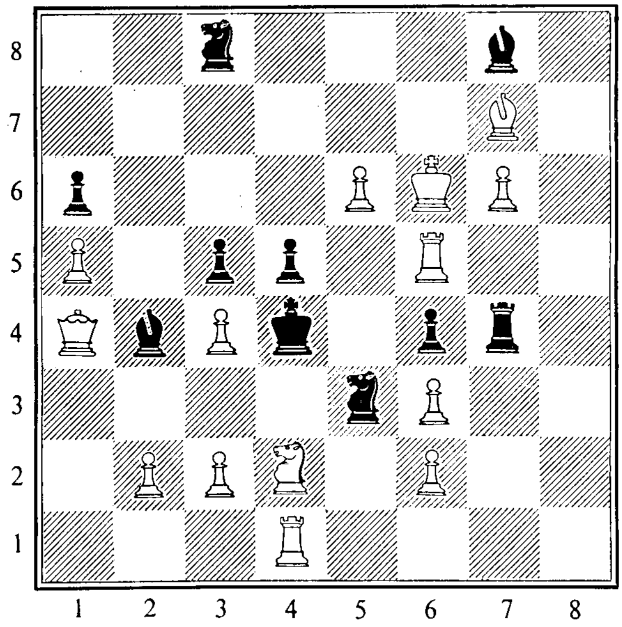


Figure 6. Blockade in chess

zovich), but very frequent in chess problems, where situations are specially constructed.

Example. [H. d'Ogly *v* Bernard given in Le Lionnais and Maget (1967)]. The situation is shown in figure 6:

Mate in two moves. The key move is: Queen in (1, 4) goes to (1, 1). It does not threaten a mate in one move. But Black must play, and each move creates a possibility of mate.

For instance:

Rook	in (7,4) goes to (8,4)	is followed by:	King	in (6,6)	goes to	(7,5)
Knight	in (5,3) moves	is followed by:	Rook	in (6,5)	captures	(4,5)
Bishop	in (2,4) captures (4,2)	is followed by:	Pawn	in (2,2)	goes to	(2,3)
Bishop	in (7,8) goes to (8,7)	is followed by:	King	in (6,6)	goes to	(6,7).

There are 25 legal moves for Black, and each of them is bad. Such a situation rarely occurs in a real game!

(2) *Threat*

A threat is a move such that, if we could play again, we would be sure to win, whatever the opponent does. In a simpler case, we have a winning move if we could play again. For instance a check at chess: such a move creates a winning move: a move capturing the opponent's King. In Go moku, if we have pieces in (6, 1), (6, 2), (6, 3) and if (6, 4), (6, 5) are empty, if we put a piece in (6, 4) we threaten to play in (6, 5). If the opponent does not put a piece here, we win. We call such a threat an *immediate* threat. There are some threats which are not immediate threats: we are sure to win if we could play again, but we do not win immediately.

Example. In Go moku, if we have: friendly piece in (6, 5) and (6, 6); (6, 3), (6, 4), (6, 7), (6, 8) empty. Then the move: 'put a piece in (6, 4)' is a threat. If we could play again, we put a piece in (6, 7). If the opponent puts a piece in (6, 3), we put one in (6, 8) and we also win. We are sure to win, but after two moves.

In chess there are also non-immediate threats. But generally moves threatening mate are checks. In chess problems, the key move must not be a check. So it is often a non-immediate threat. The opponent must destroy this

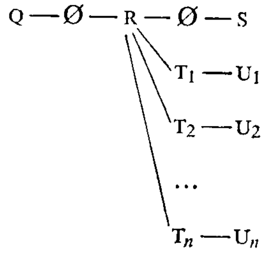


Figure 7. Description of a threat

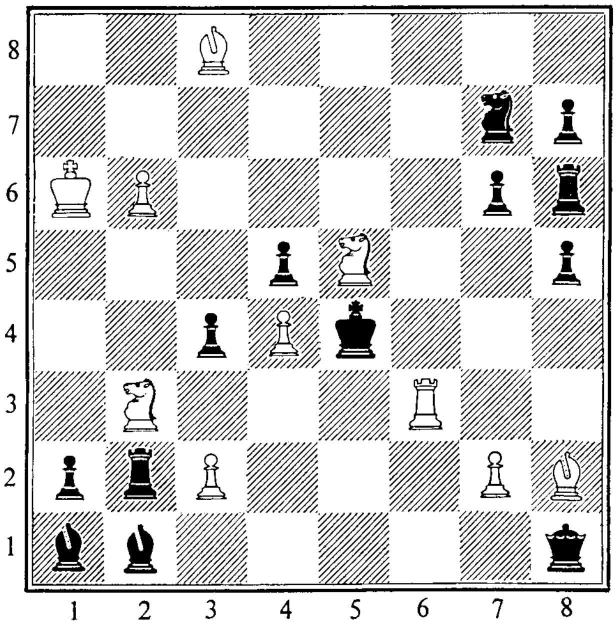


Figure 8. Threat in chess

threat. If Q is a threat to win in two moves, we have the situation shown in figure 7 (Ø indicates an opponent's no move).

If the opponent does not play after the key move Q, we should play R. If he does not play again, we play the winning move S. So the opponent tries to destroy S with the moves T_1, T_2, . . . T_n. But for each of them, there is a winning move U_i.

But the opponent can play the move X after Q. He will try to destroy the threat. He can do several things:

(a) destroy R with move X. After X, R is no longer legal.

(b) counter R. R is still a legal move after X, but if we play R, we lose before we could play the winning move S. In chess, for instance, X can pin the piece moved by R. If we move this piece to give a check, the opponent captures our King.

(c) create a new reply T_{n+1}, with no winning move U_{n+1} after it.

(d) destroy one of the U_i.

(e) X is an immediate threat. It creates the winning move Y for the opponent. If we play R, he will play Y before S.

Let us give an example [Hartong given in Le Lionnais and Maget (1967)]: *see* figure 8: Two mover.

Q is: Knight in (5, 5) captures (3, 4)
R is: Knight in (3, 4) goes to (4, 6)
S is: Knight in (4, 6) captures (5, 4)

T_1: King captures (4, 4)	U_1: Knight in (2, 3) captures King
T_2: King goes to (4, 3)	U_2: Rook in (6, 3) captures King
T_3: King goes to (5, 3)	U_3: Rook in (6, 3) captures King
T_4: King captures (6, 3)	U_4: Pawn in (7, 2) captures King
T_5: King goes to (6, 4)	U_5: Rook in (6, 3) captures King
T_6: King goes to (6, 5)	U_6: Rook in (6, 3) captures King
T_7: King goes to (5, 5)	U_7: Pawn in (4, 4) captures King.

Some opponent's possibilities after Q are:

(1) Knight in (7, 7) goes to (5, 8). He will have a new T_i: 'Knight in (5, 8) captures Knight in (4, 6)' without U_i.

(2) Pawn in (7, 6) goes to (7, 5). He creates a T_i: 'Rook in (8, 6) captures Knight in (4, 6)' without U_i.

(3) Queen in (8, 1) captures (8, 2). He creates a T_i: 'Queen in (8, 2) captures Knight in (4, 6)' without U_i.

(4) Queen in (8, 1) goes to (6, 1). The Knight in (3, 4) is pinned. If we play R, the opponent captures our King with his Queen.

(5) Pawn in (4, 5) captures Knight in (3, 4). The opponent destroys R.

(6) Rook in (2, 2) captures (2, 3). This move destroys U_1. T_1 is possible after R. . . .

This is only a study of the key move threat and of the possibilities of escape for the opponent. This is not a study of two movers which, for each opponent's reply, give a new winning move.

We see that if a move threatens to win in many moves, the opponent will have many possibilities of destroying this threat, so we do not limit the number of his moves very much; on the other hand, if there is an immediate threat, he can only destroy the winning move, and generally there are not many moves which do it. For this reason, immediate threats (like a check in chess) are very useful. In chess we find other threats, but not very often. But in Go moku, moves threatening to win in two moves occur very frequently. In that case the opponent can destroy the dangerous moves by playing in the squares where we want to play and destroying R or S, or he can play an immediate threat that we must destroy, because, if not, he would win before us.

The search for a win with immediate threats. We shall consider only immediate threats. We neglect blockade, but it is not very important because such cases are infrequent. We neglect also non-immediate threats, and this is a more serious limitation of the program. In some games, like Go moku, such threats are very frequent. The method which I shall describe can be complicated for such threats.

The method sees whether we can win with a succession of checks in chess, and with patterns where there are already three of the five necessary men in Go moku.

When we try to find a possibility of winning, there are two problems:
(a) Finding threatening moves and the opponent's replies.
(b) Pruning and developing the *and/or* tree.

We shall study these two points, but deal quickly with the second, which is well known.

How the program finds threatening moves and opponent's replies

(1) *Finding immediate threats.* We want to find a couple of moves of the player: Q — R such that, if we play Q, then R is a winning move. So the first thing to do is to execute the algorithm indicating how to win. There are two cases:

(a) the algorithm indicates that we do not win, and that p conditions: c_1, c_2, . . . c_p, are necessary for this no-winning situation. These p conditions prevent us winning, and we shall try to destroy them. We have this case in chess: $p = 1$; one condition prevents us winning: there is an enemy man in the square S, if the opponent's King is in square S.

(b) the algorithm indicates that there is a win with p modifications. Let $m_1, m_2, \ldots m_p$ be the conditions which must be fulfilled and $d_1, d_2, \ldots d_r$ the existing necessary conditions. We must fulfil the m_i without destroying the d_j.

We do not examine the case where one move Q can destroy (fulfil) the p conditions. It is a winning move, we play it and there is no problem. Also we do not examine the case where more than two moves are necessary: there is not an immediate threat.

If there is an immediate threat, there are four basic possibilities.

(a) $p=1$: only one condition has to be destroyed (fulfilled). Let this condition be on square S. But no move can do it, we must have an intermediate state for square S. Q changes S at this state and then after R we have the desired condition destroyed.

I know no game where this occurs, but it is theoretically possible. Suppose that we are playing a game where the moves are those of chess, but where we win if square (5, 8) becomes empty. Suppose that there is an enemy piece in (5, 8). We cannot win with one move. We must play at least two moves: the first move Q will capture the piece in (5, 8) and the second move R will move our piece out of (5, 8). The reason is that no friendly move can change: square S = enemy, into the state: square S = empty. We must pass by the intermediate state: square S = friendly.

(b) We must fulfil one or several conditions. The legal move Q fulfils all these conditions, but it destroys some conditions which are necessary and were already fulfilled. We must play R for correcting the bad effect of Q.

Example. In tic-tac-toe, *see* figure 9.

There is a possibility of winning with one modification. Two conditions are already fulfilled: (1, 3) and (3, 3) friendly, but (2, 3) must become friendly. Move Q: (3, 3) goes to (2, 3) fulfilling this condition. But it destroys the

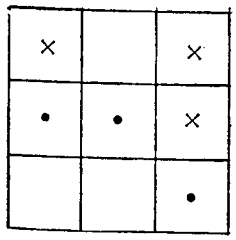


Figure 9. Second kind of threat in tic-tac-toe

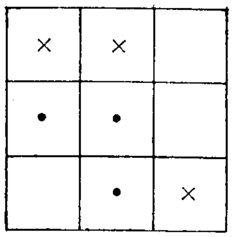


Figure 10. Third kind of threat in tic-tac-toe

condition: (3, 3) friendly, which was fulfilled. So we must play R: (3, 2) goes to (3, 3) and we correct the effect of Q.

(c) One or several conditions must be fulfilled (destroyed), but no move can do this. Q will create a situation where such a move exists; after Q there will be a winning move R which was not legal before playing Q. We are always in this case at chess. R is a move capturing the opponent's King: Q is a move creating R: a check.

We have also this case in tic-tac-toe, in figure 10. There is a winning state with one modification: (3, 3) becomes friendly. Q is: (3, 1) goes to (3, 2) and creates R: (3, 2) goes to (3, 3).

Two methods enable us to find such couples.

(1) For each friendly move Q, we generate all the friendly moves R_i created by Q. The number of R_i is smaller than the number of legal moves. We play Q, then R_i and we examine if we have won.

(2) We generate the moves R_i which fulfil (destroy) the necessary winning (non-winning) conditions. There is no legal move of this type, but we generate the moves which could exist if there were some modifications on the board. Then, for each R_i, we generate the moves Q_{ij} which give changes such that R_i becomes legal.

I choose the second method which is more directed toward the winning state. It is more interesting if there are many legal moves.

(d) Q fulfils (destroys) some conditions and R the others. In that case we can swap Q and R. We have two couples:

Q followed by R

R followed by Q.

Example. In Go moku: friendly piece in (6, 1), (6, 2) and (6, 4); (6, 3) and (6, 5) empty. There are two conditions to fulfil: (6, 3) and (6, 5) become occupied by a friendly piece. Q is: put a piece in (6, 3) and R is: put a piece in (6, 5).

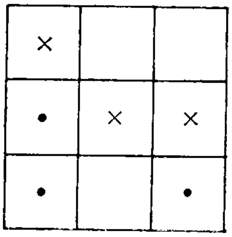


Figure 11. Fourth kind of threat in tic-tac-toe

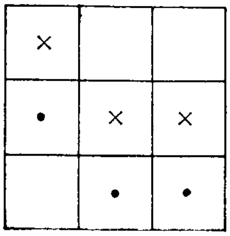


Figure 12. Reply to a threat in tic-tac-toe

Example. In tic-tac-toe, figure 11. Q is: (2, 2) goes to (2, 3); R is: (3, 2) goes to (3, 3).

In chess we never have this case: there is only one condition which prevents us from winning. But in fairy chess we can have this case. If each player has two Kings and if to win we must capture these two Kings, we can have:

Q: move capturing the first King

R: move capturing the second King.

There can be combinations of the four basic situations: Q fulfils some conditions, but destroys one condition already fulfilled. R reinstates this condition and fulfils the others. This is a combination of cases (b) and (c).

In all the cases we have some couples of friendly moves: Q — R such that if we could play anew after Q, we should win in playing R. We are not obliged to generate all the moves, but only the moves which destroy or fulfil a well-defined condition.

(2) *Finding the opponent's replies.* If we play move Q, the opponent can try two things if he does not want to lose:

(a) If there is a winning move for him, he plays this move and wins before we can play R. If such is the case, we eliminate the couple Q — R.

Example. Tic-tac-toe, figure 12.

Q is: (2, 2) goes to (2, 3)

R is: (3, 2) goes to (3, 3).

But, after Q, the opponent plays: (1, 2) goes to (1, 1) and wins. So we eliminate Q.

In the same manner, a piece which is pinned at chess in front of our King cannot make a check.

(b) The opponent has no winning move. He must destroy R if he does not want to lose. For this, he must destroy, at least, one of the necessary conditions for R to occur.

For each necessary condition of R, we enumerate the opponent's moves which destroy it. If there is a condition:

friendly piece in square K;

The opponent will try to capture it.

enemy piece in square K;

The opponent will try to move it.

square K empty;

The opponent will try to put a piece here.

If an opponent's move destroys several conditions of R, we consider it only once.

Example. R is: friendly Bishop in (7, 7) captures enemy King in (8, 8). There are two conditions: friendly Bishop in (7, 7) and: enemy in (8, 8). The move: enemy King captures Bishop – destroys these two conditions. But we keep it only once.

In chess, the move R captures the enemy King. The conditions of R are:
friendly piece in a
enemy in b (square of the King)
squares $c_1, c_2, \ldots c_n$ empty (n may be 0).

If there is a check, the opponent must destroy one of these conditions:
capture the piece which gives a check
move his King
put one of his pieces in a square c_i (if $n \neq 0$).

In each case the program finds this property, well-known to chess players.

In fairy chess, this theorem is not always true. Suppose that there is a check by a Grasshopper [the Grasshopper moves in the same directions as a Queen, but a man (white or black) must be on his line; the Grasshopper can go to the square immediately behind this man (called *sautoir*), if this square is empty or occupied by an enemy man. (*See* Le Lionnais and Maget 1967)] and that the sautoir is an enemy piece in square d. The conditions of the move R: Grasshopper captures King are of the same nature as in chess, except the condition: enemy piece in square d. The opponent can destroy the threat by taking his piece out from d.

With this method, for each threatening move Q, we have the replies: T_1, $T_2, \ldots T_n$. After each T_i we must verify that there is not another winning move R′. If such is the case, we delete T_i.

Example. Friendly Rook in (6, 8). Enemy King in (8, 8). (7, 8) empty.
R is: Rook captures King in (8, 8).
T_1: King in (8, 8) goes to (7, 8), destroys two conditions of R:
(8, 8) enemy and (7, 7) empty. But, after T_1, there is a winning moveR′: Rook captures King in (7, 8). We delete T_1.

If after doing this $n = 0$, we shall win in playing Q, because the opponent can destroy R only by creating a new winning move. In chess we say in that case that there is checkmate after Q.

Remark. Multiple attacks. We can have the move Q several times as first member of a couple Q — R. We can win if we play Q then R_1 *or* if we play Q then $R_2 \ldots$ *or* if we play Q then R_n. If we play Q, the opponent must destroy R_1 and $R_2 \ldots$ and R_n. This situation is very interesting because it is difficult to destroy n moves with only one move, and this decreases drastically the number of opponent's replies.

We could write a special procedure for this case. But it is difficult; the

opponent has two possibilities. Let us take the case of a double threat: $n=2$.
(a) The opponent destroys one condition common to R_1 and R_2, if there are such conditions. This is always the case in chess; in a double check there is always one condition common to R_1 and R_2: enemy piece in square A (if the enemy King is in A). The opponent destroys the two threats in destroying this condition. It is well known: if there is a double check, the opponent must move his King.
(b) There is a move which destroys one condition of R_1 and one different condition of R_2. We can see that this case never occurs in orthodox chess, but we can find such cases in fairy chess.
Example. Friendly Grasshopper in (6, 8). Enemy Bishop in (7, 8). Enemy King in (8, 8). Friendly Rook in (8, 5). (8, 6) and (8, 7) empty.
R_1 is: Grasshopper captures King.
Conditions: (6, 8) friendly Grasshopper
(7, 8) enemy
(8, 8) enemy.
R_2 is: Rook captures King.
Conditions: (8, 5) friendly Rook
(8, 6) empty
(8, 7) empty
(8, 8) enemy.
The enemy move: Bishop goes to (8, 7), destroys these two moves. It destroys the condition; (7, 8) enemy, of R_1 and the condition; (8, 7), empty of R_2. In this case the opponent is not obliged to move his King.

But it is difficult to write such a program for multiple threats. It is more convenient, when we have several couples (Q—R) with the same Q, to consider only the first and to eliminate the others. The program will try to destroy R_1 with T_1 or T_2 . . . or T_p. If the move T_i does not also destroy the other winning moves R_j, T_i is eliminated: We verify always after each T_i that there is no winning move. As multiple threats are not very frequent, this method does not waste much computer time and is easy to implement.

Finally, we have the threatening moves and for each one, the possible opponent's replies. We give in figure 13 an example of the result of this method. We are playing chess.
First step · List of couples Q—R of friendly moves, R being a winning move. In chess we are always in the third case, Q creates R.
(1) Knight in (4, 5) captures (6, 6) – Knight in (6, 6) captures King
(2) Knight in (4, 5) captures (6, 6) – Queen in (2, 3) captures King
(3) Knight in (4, 5) goes to (5, 7) – Knight in (5, 7) captures King

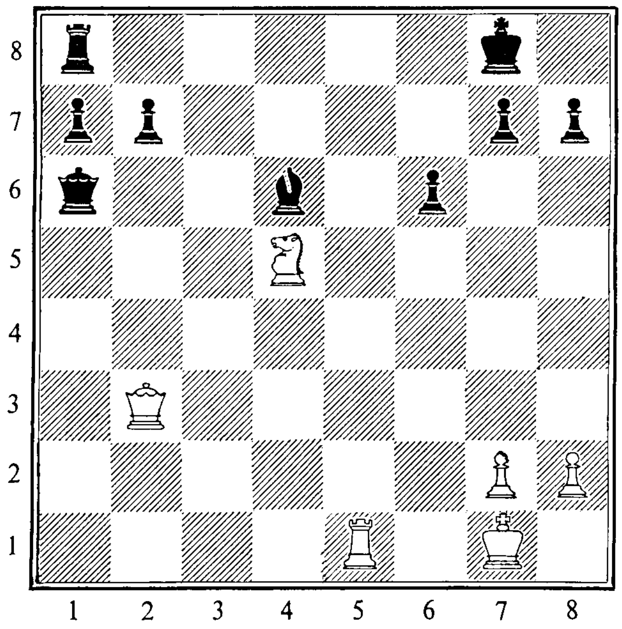


Figure 13. Threatening moves in a chess position

(4) Knight in (4, 5) goes to (5, 7) – Queen in (2, 3) captures King
(5) Knight in (4, 5) goes to (6, 4) – Queen in (2, 3) captures King
(6) Knight in (4, 5) goes to (5, 3) – Queen in (2, 3) captures King
(7) Knight in (4, 5) goes to (3, 3) – Queen in (2, 3) captures King
(8) Knight in (4, 5) goes to (2, 4) – Queen in (2, 3) captures King
(9) Knight in (4, 5) goes to (2, 6) – Queen in (2, 3) captures King
(10) Knight in (4, 5) goes to (3, 7) – Queen in (2, 3) captures King
(11) Rook in (5, 1) goes to (5, 8) – Rook in (5, 8) captures King.

There are two double checks. We eliminate, as we have said above, couples (2) and (4).

Second step · For each threatening move Q, we try to find the opponent's replies. We show this for three moves Q.

(a) *Couple* 1. The conditions of R are:

(6, 6) White Knight
(7, 8) Black.

Enemy moves destroying the first condition:

(1) Pawn in (7, 7) captures Knight.

The second condition:

(2) King goes to (8, 8)
(3) King goes to (6, 8)

(4) King goes to (6, 7).

But after move (1) there is a winning move: Queen in (2, 3) captures King in (7, 8). There is also a winning move after move (4): Queen in (2, 3) captures King in (6, 7).

So after: Knight captures (6, 6) – there are two replies:

King goes to (8, 8)

King goes to (6, 8).

We see that the program finds the good replies, although there was a double check.

(6) *Couple* 5. The conditions of R are:

(2, 3) White Queen

(3, 4), (4, 5), (5, 6), (6, 7) empty

(7, 8) Black.

No enemy move can destroy (2, 3) White and (4, 5) and (5, 6) empty.

(3, 4) empty:	(1) Queen in (1, 6) goes to (3, 4)
(6, 7) empty:	(2) King in (7, 8) goes to (6, 7)
(7, 8) Black:	King goes to (6, 7); already seen
	(3) King goes to (6, 8)
	(4) King goes to (8, 8).

But after move (2), there is a winning move: Queen captures King in (6, 7). We eliminate this reply.

After: Knight in (4, 5) goes to (6, 4), there are three replies:

Queen in (1, 6) goes to (3, 4)

King goes to (6, 8)

King goes to (8, 8).

(c) *Couple* 11. The conditions of R are:

(5, 8) White Rook

(6, 8) empty

(7, 8) Black.

These black moves destroy:

(5, 8) white:	(1) Rook in (1, 8) captures (5, 8)
(6, 8) empty:	(2) Bishop in (4, 6) goes to (6, 8)
	(3) King in (7, 8) goes to (6, 8)
(7, 8) black:	King in (7, 8) goes to (6, 8); already seen
	(4) King in (7, 8) goes to (8, 8)
	(5) King in (7, 8) goes to (6, 7).

The opponent has 5 possible moves. But after move (3) there is the winning move: Rook in (5, 8) captures (6, 8), and after move (4), the winning move: Rook in (5, 8) captures (8, 8). So there are only three possible replies.

After: Rook in (5, 1) goes to (5, 8), the opponent can play:

Rook in (1, 8) captures (5, 8)
Bishop in (4, 6) goes to (6, 8)
King in (7, 8) goes to (6, 7).

Pruning and expanding the tree

Tree pruning. We must prune the tree when we get a winning situation or a situation in which there is no winning possibility. We have a well-known *and/or* tree. I shall only indicate that a winning situation is a threat with no reply and that a non-winning situation is without possible threat.

Expanding the tree. We must choose the situation where we search the threats and the opponent's replies. We first try the situations where, if we win, the opponent has only a few possibilities to escape by playing other moves before our winning move. We shall describe this algorithm. There are two steps.

First step · We associate an integer with each node of the tree. We begin with the leaves. We put the weight 1 on each leaf. When we are in a situation after a friendly move, we associate with this node the sum of the weights of the nodes which are the opponent's replies: we must study all these replies. If we are after an enemy move, we associate with this node the value of the smallest weight of its successors: we can choose our move.

These weights give a good idea of the number of possibilities of escape for the opponent.

Second step · We choose the situation to study. We begin at the root of the tree. If we want to choose a friendly move, we take the node which has the smallest weight. If several nodes have the same value, we take the first. It is not important; if it is bad, we shall take another node the next time.

If we want to choose an enemy move, we take successively all the replies. For this, we indicate which node was taken the last time. This is important to avoid studying one reply a long time when we could see quickly that after another reply we cannot win.

By this process, when we get to a leaf, we have the situation to study. We apply the method described above, we generate a subtree which is connected to the main tree.

The weight of the root gives a good idea of the possibilities of escape for the opponent. If this weight is greater than a parameter given as data to the program, the program gives up. It is then unlikely we could win.

Results. I have given some results in Pitrat (1969). We shall take one and see the steps of the program. The board (chess) is given in figure 14. White can still play short or long castling. White to play. This situation occurred

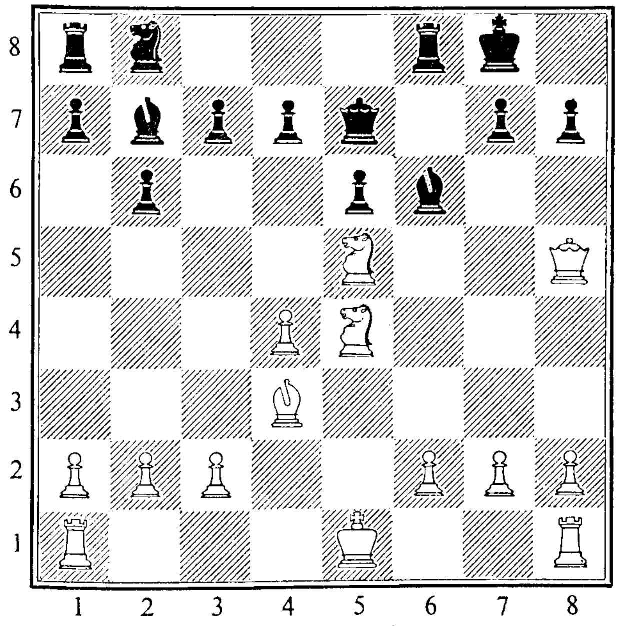


Figure 14. Chess position where there is a possibility of winning

in a real game played by Edward Lasker (1962).

The name of a situation is a letter put after each opponent's reply.

(1) There are 3 threats.

(8, 5) captures (8, 7)
one reply: (7, 8) captures (8, 7) – A

(8, 5) goes to (6, 7)
three replies: (5, 7) captures (6, 7) – B
(6, 8) captures (6, 7) – C
(7, 8) goes to (8, 8) – D

(5, 4) captures (6, 6)
four replies: (7, 7) captures (6, 6) – E
(6, 8) captures (6, 6) – F
(5, 7) captures (6, 6) – G
(7, 8) goes to (8, 8) – H

So the program considers A.

(2) After A there are 7 threats played by the Knight in (5, 4)

(5, 4) captures (6, 6)
two replies: (8, 7) goes to (8, 8) – I
(8, 7) goes to (8, 6) – J

(5, 4) goes to (7, 5) – there are four replies.
L

For the other threats there are four replies.
(3) The program considers I.
There are two threats: (5, 5) goes to (7, 6)
(5, 5) goes to (6, 7)
and no opponent's reply after the first. So we prune the tree.
It is sufficient to show that after J, we win.
(4) The program considers J.
There are 4 threats but only one with one opponent's reply:
(5, 5) goes to (7, 4)
one reply: (8, 6) goes to (7, 5) – K
(5) The program considers K.
There are 4 threats.
(8, 2) goes to (8, 4)
one reply: (7, 5) goes to (6, 4) – L
(6, 2) goes to (6, 4)
two replies: (7, 5) captures (6, 4) – M
(7, 5) goes to (8, 4) – N
(6, 6) goes to (8, 7). Four replies.
(6, 6) goes to (5, 4). Seven replies.

The second threat gives a quicker mate. But Edward Lasker did not see it. It was only several years later that it was discovered. It seems that human heuristics are, in this case, similar to those of the program.

(6) The program considers L.
There are 3 threats:
(7, 2) goes to (7, 3)
one reply: (6, 4) goes to (6, 3) – O
(6, 6) goes to (8, 5)
one reply: (6, 4) captures (7, 5) – P
(6, 6) goes to (4, 5) and four replies.
(7) The program chooses O.
There are 4 threats.
(7, 4) goes to (5, 5)
one reply: (6, 3) goes to (7, 2) – Q
(4, 3) goes to (5, 2)
one reply: (6, 3) goes to (7, 2) – R
(4, 3) goes to (5, 4)
one reply: (2, 7) captures (5, 4) – S
(7, 4) goes to (8, 2)
one reply: (6, 3) goes to (7, 2) – T

These four moves are equivalent. But if the program chooses Q, S or T, it will quickly stop in this direction, because there are only threats with several replies or because no threat is possible. If we have the first case, then it will prefer another situation with only one reply, so we are sure to examine R quickly, if the move, (4, 3) goes to (5, 2), is not generated first.

(8) The program chooses R.

There are 5 threats, but only two with one reply:

(8, 1) goes to (8, 2)

one reply: (7, 2) goes to (7, 1) – U

(7, 4) goes to (5, 3)

one reply: (7, 2) captures (8, 1) – V

(9) The program chooses U.

There is one threat without reply: (5, 1) goes to (4, 2). There is another threat of this type: long castling, but the program considered the first generated.

In pruning the tree, we see that we have won.

CONCLUSION

This method gives good results for games like chess and is good for many kinds of fairy chess. But for some other games, it is not so good for several reasons. Examples are:

(a) games where we are not constantly near winning. Such a procedure is interesting only if we can threaten the opponent;

(b) games where non-immediate threats are very frequent, such as Go moku. We must complicate the algorithm for these cases.

(c) Even if we have only immediate threats, the number of opponent's replies must not be always the same. The choice of the situation to study would not be efficient: all the moves would be equivalent. This case occurs in Go moku: there is always one reply for each immediate threat. In that case we should have other heuristics. General heuristics are not good for every game.

BIBLIOGRAPHY

Lasker, E. (1962) *Chess for fun and chess for blood.* New York: Dover publications.

Le Lionnais, F. & Maget, E. (1967) *Dictionnaire des échecs.* Paris: Presses Universitaires de France.

Pitrat, J. (1969) Realization of a general game playing program. *Information Processing 68*, pp. 1570–4 Amsterdam: North-Holland Publ. Co.