INFORMATION PROCESSING 68 - NORTH-HOLLAND PUBLISHING COMPANY - AMSTERDAM (1969)

REALIZATION OF A GENERAL GAME-PLAYING PROGRAM

JACQUES PITRAT
Institut Blaise Pascal, C.N.R.S.,
23, *Rue du Maroc*, 75, *Paris XIX*, *France*

We study some aspects of a general game-playing program. Such a program receives as data the rules of a game: an algorithm enumerating the moves and an algorithm indicating how to win. The program associates to each move the conditions necessary for this move to occur. It must find how to avoid a dangerous move.

We describe the part of the program playing the combinatorial game in order to win: how it can find the moves which lead to victory and what are the only opponent's moves with which he does not lose. This program has been tried with various games: chess, tic-tac-too, etc.

1. INTRODUCTION

My aim was to realize a program playing several games. The rules of the particular game which it must play are given as data. If we want to have a performing program, it must be capable of studying these rules.

The program is not completely general. It has limitations of three kinds:

a. It can only play games on a bidimensional board.
b. The rules of a game are written in a language which cannot describe every game, but which, however, covers a very large ground.
c. The more severe restrictions arise from heuristics which can be used in various games, but with very weak performances for some games.

I cannot describe the whole program, which is very large. I shall describe the combinatorial play which happens when we try to win, whatever the opponent may do.

The program can also play a positional game: this comes about when the opponent can play many moves without serious threats. We shall not discuss this part of the program.

2. LANGUAGE USED TO DESCRIBE THE RULES OF A GAME

There are two parameters for each square of the board:

a. One giving the occupation of the square: empty - friend - enemy,
b. One giving the type of man if the square is not empty.

For example, if the game is chess, the piece may be: king, rook, pawn . . .

For some games, all men are of the same kind: tic-tac-toe, Go-Moku.

There are variables. They can represent a square or a number.

There are statements such as those of FORTRAN, ALGOL: arithmetic, test, go to statements.

Some are very specific to games:

a. Result statement. This statement gives information about winning in a particular state of the board. This may be: victory, loss, draw, no victory . . .
b. Move statement. This statement describes a move, which can be made up of several parts (partial moves). The parts of a move fall into four types:
    i. The man in square A goes to square B
    ii. The man in square A is captured
    iii. A man of type T is put in square A
    iv. The man in square A becomes a new type T.

To sum up, the rules of a game are given as algorithms written in the language described above: an algorithm enumerating legal moves and an algorithm indicating how to win.

3. STUDY OF AN ALGORITHM

First, the program must find, for each move or for each indication of victory, the conditions necessary to obtain it. This is useful if we want to destroy an opponent's move or if we want to try to make a move possible, or to win, or to escape a danger.

These conditions are not given by algorithms,

which merely enumerate moves or indicate whether we win or not. To obtain these conditions, the program must study these algorithms.

An algorithm is put into the computer as a graph. It has many branch points, corresponding to tests, computed go to, loops. At each such point, there is a condition. For instance: "If a square K is empty, go to L1, otherwise to L2".

To each branch point, we associate its "conjugate". This is the first statement where we are sure to arrive, whatever branch we choose and whatever the answers of the following tests may be. There is always a conjugate: we gather in one statement all the possible terminal points of the algorithm. This is done in a first step: each branch point has its conjugate.

While executing an algorithm, at each branch point, we put the condition which has been satisfied in a push down. We remove it when we execute the conjugate: this condition now has no importance. Whether this condition is true or false, we are sure to arrive at this statement.

If we have a statement of result or of move, the necessary conditions are in the push down. Hence, we output the contents of the push down at each such statement.

For instance, if the game is chess, and if we have the state of the board shown in fig. 1, if we enumerate the opponent's moves we have:

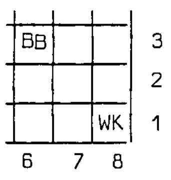


Fig. 1.

The man in the square (8,1) is captured. The man in (6,3) goes to (8,1).

With this move, the program gives the conditions:

a. Enemy man in (6,3)
b. Bishop in (6,3)
c. Square (7,2) empty
d. Friendly man in (8,1).

We see the interest of this method. If we want to avoid losing, we must destroy one of these conditions. The number of moves to consider is thus greatly reduced.

This method is not entirely exact. Under certain circumstances, some conditions are not taken into account. But this case does not arise in any of the games for which I have written rules. It is possible to write a more complex program which always gives all the conditions.

It is also useful to enumerate the moves or the wins that would be possible if the board were changed. If we force an empty square to be occupied by an enemy man, what are the new moves? If we force a square to be occupied by a friendly man, do we win? In these cases, we may say that we are forcing.

To see what happens with forcings, at each modifiable branch point, we store the state of the variables and the references of the statements to which the other branches lead. This is done only if this forcing is possible for the game.

When its normal work is finished, the program is reset to the state which has been stored and takes another branch. It stops when it arrives at the conjugate: the rest of the algorithm has already been done. At each result or move, it also outputs the conditions, but indicates those which have been forced.

In the figure shown above for chess, the following move should be recognized as a forcing: The man in (7,2) is captured. The man in (6,3) goes to (7,2).

Under the conditions:

a. The man in (6,3) is an enemy
b. A bishop is in (6,3)
c. The man in (7,2) is a friend. This is the forcing condition.

4. THE SEARCH FOR A WIN

First, let us show how we can destroy or fulfil a condition.

If the condition is:

the square A is empty

we can destroy it by bringing a man there. To destroy: the man in the square A is a friend, we can take away the man.

To destroy: the man in the square A is an enemy, we can capture the man in A.

The methods to fulfil a condition are similar.

This method is entirely general and good for every game. Of course, if a particular game has no capture move, a condition like: "enemy is in square A", cannot be destroyed by the player.

We count as a single condition two conditions on the same square.

Now, we can see how to win against all defences. In a first stage, we shall get pairs: move dangerous for the opponent - list of conditions one of which must be destroyed by the opponent. In a second stage, we shall get pairs: move dangerous for the opponent - list of the opponent's counterstrokes. Then we shall see how the program can choose among its moves.

Suppose the algorithm indicates when we win. In parenthesis, we write what to do when it indicates when we do not win.

There are two cases:

a. There is a win with one forcing (only one condition prevents us from winning). There are conditions $E_1, \ldots, E_p$, already fulfilled (for win only) and one condition k to be fulfilled (destroyed). If a move fulfils (destroys) k without destroying $E_1, \ldots, E_p$, we win. But if no move fulfils (destroys) k, we try to see if we can fulfil (destroy) k in two moves. We enumerate all the moves which can fulfil (destroy) it with one forcing, and we remove all those which destroy one or several of the E_j. Let $q_1, \ldots, q_n$ be these moves.
Let q_i be one of these. For the others, we will proceed in the same way. We know the conditions $C_1, \ldots, C_t$ necessary for this move, and there is a condition D to fulfil since there is one forcing. We look for the move fulfilling D and we remove those destroying one of the E_j or one of the C_j. Let $r_1, \ldots, r_s$ be these moves.
If we play one of these (which fulfils D), for instance r_m, the opponent is obliged to destroy the move q_i, if he does not, he loses after q_i, unless he destroys one already fulfilled condition for a win. Thus he must destroy one of the conditions C_j or D, or one of the E_j, winning conditions already fulfilled.
We have a list of pairs:

$$r_m - E_1, \ldots, E_p, C_1, \ldots, C_t, D,$$

b. There is a win with two forcings (two conditions prevent us from winning).
Suppose we are in the first case. Conditions $E_1, \ldots, E_p$ are already fulfilled and two conditions, k_1 and k_2, must be fulfilled. There may be many possibilities of this type. We will proceed in the same way for each of them.
Let us take k_1 (when this is finished, we do it again, swapping k_1 and k_2). We enumerate the moves fulfilling k_1 and we eliminate those which destroy one or several E_j. Let $q_1, \ldots, q_n$ be these moves.
After playing q_i, if we fulfil the condition k_2, we win. Let $t_1, \ldots, t_s$ be the moves fulfilling k_2. Suppose for the sake of simplicity that there is only one move M and let $C_1, \ldots, C_t$ be the conditions necessary for this move.
If we play q_i, the opponent must prevent us from playing M or destroy one of the conditions already fulfilled; he must destroy one of the C_j to prevent us from playing M next, which thus enables us to win, or one of the E_j or k_1 to destroy a condition already fulfilled. If we play M next, there will always be a condition to fulfil.
Thus we have the list of pairs:

$$q_i - C_1, \ldots, C_t, E_1, \ldots, E_p, k_1.$$

Now, for each condition to destroy, we look for the opponent's moves destroying it. Thus we have a new list of pairs, the first element being a player's move and the second, the list of the opponent's counterstrokes. He must choose among them if we play the first element and if he does not want to lose. If there is no move in the second element, we win.

Let us examine the application of this method to chess. The win algorithm tells us we do not win. By our method, the program sees that it is because there is an opponent's king in the square A. We are in the first case and $p = 0$. Condition k is: enemy king is in the square A. If there is a move which destroys it: a move which captures the man in A, we win. If not, we look for the moves which destroy it with one forcing. For instance, the condition to fulfil may be:

a. Friendly rook in square B, or
b. Square C empty (discovered check).

Let us take the first case: there must be a friendly rook in B. The win move: "Rook captures king" needs other conditions which are fulfilled, for instance:

C_1 : enemy man in A
C_2 : square E empty.

We look for the moves bringing a friendly rook in B. Let a move be r_i: Rook in F goes to B.

If we play this move, the condition: "Friendly rook is in square B" is then fulfilled; the opponent must destroy one of the conditions of "Rook captures king":

a. Friendly rook is in B
b. Enemy man is in A
c. Square E is empty.

For each condition, he looks for the moves destroying it. If, for instance, he cannot move his king, without a new check, nor capture the rook, he may have only two moves:

a. The man in H goes to E
b. The man in I goes to E.

Then we know that if we play:
Rook in F goes to B,
the opponent has only two counterstrokes:

a. The man in H goes to E
b. The man in I goes to E.

It is essential to see that this method is entirely general. I gave an instance for chess, but we can apply it to Go-Moku or to tic-tac-toe.

We can make two remarks:

a. A move may produce many threats (for instance double check). In this case, we will find twice or more the same move as first element of a pair. Then, we remove all these pairs and we create a new pair: its first element is the move, and the second a list of conditions obtained by a "and" of the lists of conditions. If there is no condition, we win.

b. When we play the move which is the first element of the pair, we must verify that the opponent does not win. If he does, we must remove the pair: it is useless for the opponent to destroy the threat, he wins before it occurs. We must also verify for each opponent's counterstrokes that there is no new threat for him. For instance, if the game is chess, it is useless to move his king from one check towards another check.

When the program has the following list of pairs: threatening moves - list of opponent's counterstrokes, it must choose one of the first elements. The opponent is then free to choose among the corresponding list of counterstrokes. We have a tree.

We must use heuristics in order to find quickly if we can win. One of them is to try first the moves where the opponent has few counterstrokes. We restrict his possibilities and we see more easily all the possible cases.

When we reach a win, it does not mean that we win, because the opponent may attempt a different move previously. We must prune the tree, working towards the beginning. If the opponent has only one possibility and loses, we climb up two levels higher: we choose the move which leads to a win. If there are more than one branch, we only cut the branch, if there is only one branch, we resume the procedure.

If we return to the beginning, we win whatever move the opponent chooses. We stop if we have no further possibility of threatening the opponent or if he has too many ways of escape. This measure is heuristic.

In chess, this method leads to a sequence of checks. It is very close to Mater I of Baylor and Simon [1].

5. SOME RESULTS OF THE PROGRAM

It has been written for the CDC 3600. We describe a move by:

a. A man is added in square A : A

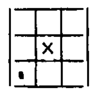
b. A man is moved from A to B : A-B

c. The man in square A is captured : × A.

A square is characterized by its two coordinates.

Tic-tac-toe.

```
cross     noughts
 2,3        2,1
 3,1        1,3
 2,2 - 3,2  and victory after the next move.
```



```
cross     noughts
 3,3        1,1
 1,3        2,3
 3,3 - 3,2  and victory after the next move.
                                Time: 6 seconds
```

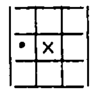


Fig. 2.

Thus, the program wins if it is playing first and can put its first man in the center.

Chess. K = King; Q = Queen; R = Rook; B = Bishop; N = Knight; P = Pawn; W = White; B = Black.

BR	BN				BR	BK	
BP	BB	BP	BP	BQ		BP	BP
	BP			BP	BB		
				WN			WQ
			WP	WN			
			WB				
WP	WP	WP			WP	WP	WP
WR				WK			WR

```
x  8,7;8,5-8,7          x 8,7;7,8-8,7
x  6,6;5,4-6,6          if    8,7-8,8

                  Then  5,5-7,6  and victory
                  Thus  8,7-8,6

   5,5 - 7,4            8,6 - 7,5
   8,2 - 8,4            7,5 - 6,4
   7,2 - 7,3            6,4 - 6,3
   4,3 - 5,2            6,3 - 7,2
   8,1 - 8,2            7,2 - 7,1
   5,1 - 4,2  and win         Time: 47 seconds
```

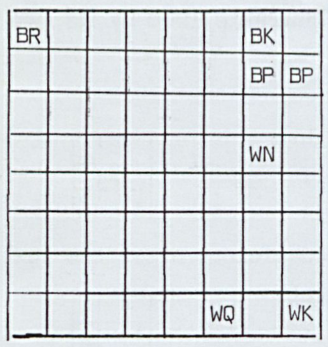
Fig. 3.

Edward Lasker played exactly the same sequence of moves. It is not the quickest mate. There is a mate in seven moves (see fig. 4).

The sequence found by the program is the sequence given by Tarrasch.

6. CONCLUSION

We must not compare the performance of a general program with that of a program playing



```
6,1 - 3,4      If  7,8 - 6,8

               Then  3,4 - 6,7 and win
               Thus  7,8 - 8,8

7,5 - 6,7            8,8 - 7,8
6,7 - 8,6       If  7,8 - 6,8
               Then  3,4 - 6,7 and win
               Thus  7,8 - 8-8

3,4 - 7,8         x  7,8 ; 1,8 - 7,8

8,6 - 6,7      and win

               Time : 77 seconds
```

Fig. 4.

only one game. Artificial intelligence aims to get general programs. If they are too particular, we often have progress in the theory of the particular problem, but no progress in the theory of artificial intelligence. Also, a particular program cannot study new problems.

This program has been tried on games as varied as tic-tac-toe, Go-Moku, chess and cylindrical chess.

It is important to see that nowhere does it use the particular properties of the game. If it is playing chess, it does not know, for example, that it is advisable to protect its men.

REFERENCE

[1] Baylor and Simon, A chess mating combinations program, AFIPS, Vol. 28 p. 341.

DISCUSSION

Question by K. Paton

In your conclusion, you state that in playing chess, the program does not know that it is protecting men. Does the program ever find out that it is protecting men?

Answer

No, it cannot learn. The general method enables it to perform in each specific case, but it is not formally capable of discovering the notion of protection. Incidentally, the idea of protection in chess does not apply in all cases, and makes the formal discovery of the program difficult.

Question by D. Levy

How does the program decide what to do in a non-tactical position?

Answer

The program takes into account the possibility of winning and forcing, e.g. in Go-Molar, it takes into account relationships of moves and opponent's moves, and tries to increase its mobility (i.e. the number of possible moves) and lessen the mobility of the opponent. The definition of mobility is included in the program, and is completely general.

Question by P. Braffort

I can understand the advantage of a general approach to the problem of artificial intelligence treating games of a different nature on the same level and with the same formalism for the establishment of rules. But have you also made an independent evaluation or comparison of your program with others, and if so how do you find it?

Answer

The aim of the program is to play several board games, but not specifically any one game, and I would not expect it to be better than a specific program for a specific game.

Question by G. Nagy

Can you compare the performance of your program with the Simon-Baylor results?

Answer

A general purpose game playing program will obviously be less efficient than a special program for a game. The combinational program is finished, and I am now writing the positional program.