OFFPRINT FROM

COMPUTER ORIENTED LEARNING PROCESSES

edited by

J.C. SIMON

Professeur à l'Institut de Programmation
Université Pierre et Marie Curie
Paris VI, France

NATO ADVANCED STUDY INSTITUTES SERIES

Series E: Applied Science - No. 14

NOORDHOFF - LEYDEN - 1976

REALIZATION OF A PROGRAM LEARNING TO FIND COMBINATIONS AT CHESS

J.Pitrat

Structures de l'Information, Centre National de la Recherche Scientifique, France

1. LEARNING IN GAMES

1.1. Some methods of learning

1.1.1. Rote learning.We can keep all the situations already found. With each situation we store an indication on its interest or the move which has to be played. Samuel[1] gives an example of such an application. This can be done if there are not too many possible situations. For instance in tic tac toe. Even in games where there are many possible situations, this method can be useful for the beginning or the end of the games. For instance, at chess, for learning the openings. We can improve this method :

if the rules are the same for all the players, we can standardize the situations : we assume that it is always the same player who has to play ; for instance, at chess, white. We just keep half of the possible situations.

if there are geometrical symetries, we can decrease the number of different situations. At Go Moku where there are two axes of symetry, we just keep a quarter of the situations.

if the game is invariant by a translation (cylindrical or toroidal board for instance) we keep only one instance of each equivalent situation.

But even with these improvements, there are many cases where this method is not useful because there are too many situations. It is doubtful that we can have good results in the middle game

at chess with such methods.

1.1.2. Learning by clustering the possible situations. We can try to generalize what has been done in a situation to another similar situation. For example in the second situation we play the same move than in the first one. Waterman[2] has written an interesting program playing poker. Let us describe it roughly.

A situation is described by the value of seven variables : value of the program's hand, amount of money in the pot, measure of conservative style by the opponent ... The program defines a partition of the set of possible values of these variables. For each subset, it associates one action. For instance : If our hand is excellent, bet low if the opponent tends to be a conservative player and has just bet low. The problem is to define wisely these subsets. This can be done by the program which improves progressively the quality of the partition.

This method is good for poker and it obtained very good results. But it is difficult to see how we can use it in a game like chess. How could we evaluate the similarity between situations, such as in similar situations we have to play the same move ? A different position of a pawn can destroy a combination.

1.1.3. Optimization of a set of weighting factors.This is interesting for the determination of an evaluation function. Let us give to a situation the value $\Sigma C_i.P_i$, P_i being some parameter: we measure its value on the board. For instance, the number of friendly men surrounded by at least three empty squares. We can try to improve the value of the weighting factors C_i .

This optimization may be progressive. After each move we modify the value of some factors. There are problems with instabilities if the changes are too important, mainly in the initial stages of this idea.

The optimization may also be done when we have gathered all the moves necessary for learning; then we estimate the best values for the weighting factors. We have no problem of instability, but, at the beginning we have not any function. Samuel[3] has also given an interesting method for this case. But such statistical methods cannot find anything from a type of combination which occurs only once.

1.1.4. Learning with definition of the parameters. If, in the preceding method, the programmer does not give in the evaluation function an important parameter, the method fails. We are also obliged for each game to find the good parameters. If we want a general program, we have to find a method which discovers a good set of parameters for each games.

Newman and Uhr[4] proposed a method for finding such set of parameters. It is good for games such as Tic tac toe, Go Moku, where we put men on a board. The program extracts useful patterns from the games that it plays. Some are desirable, others are undesirable. We understand that it is very interesting to find the pattern of figure 1 on a board at Go Moku. As the pattern is extracted from the situations, the program adapts itself to any game.

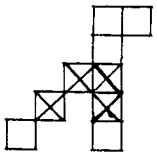


fig. 1

But there are some difficulties. We have to select useful patterns. It is also necessary that there are not too many patterns. In the case of games such as Go Moku, all the men are of the same kind. There is also no connection between men which are at a great distance from each other. But in games like chess, the situation is not so good. There are six kinds of men, so there is a great variety of patterns. And there are connections between men very distant.

T				T			
I						S	I
	I		D	I		I	C
			I			N	
			P				
		P					Q
P	P				P	P	P
R				R		K	

White to play

Qh3 × h6 Kg7 × h6
Ng5 - f7 Kh6 - g7
Nf7 × d6

T				S		C	T
I	I	I			I	I	I
	F		I				
	P						
			P	P	D		
		B		C	B		
P			Q			P	P
R	N				R	K	

Black to play

Qf4 × h2
Kg1 × h2 Ne3 × f1
Kh2 - g1 Nf1 × d2

fig. 2*

There is also another reason which explains that the geometrical patterns are good for Go Moku and not for chess. In Go Moku,

* See the appendix for the conventions of representation of the board and of the moves.

the logical properties of the game are very simple : we can only put a man in an empty square. At chess, the rules are more complicated, and the logical relations between men are difficult to give in geometrical terms. It is frequent that situations which are similar in a geometrical sense give combinations completely different. Opposite, very different situations give combinations which are similar.

The two combinations of figure 2 seem intuitively very similar. But there is no similarity between the boards, even if we change Black and White in one of the boards. The reason is that they have in common a logical pattern, not a geometrical one. If we write $A \rightarrow B$ for indicating that move A creates move B and $C \dashv\!\rightarrow D$ for indicating that move C destroys move D , we have the patterns of figure 3 . We have exactly the same logical structure. It would be interesting to apply the ideas of Newman and Uhr for finding logical patterns. I tried to develop this idea.

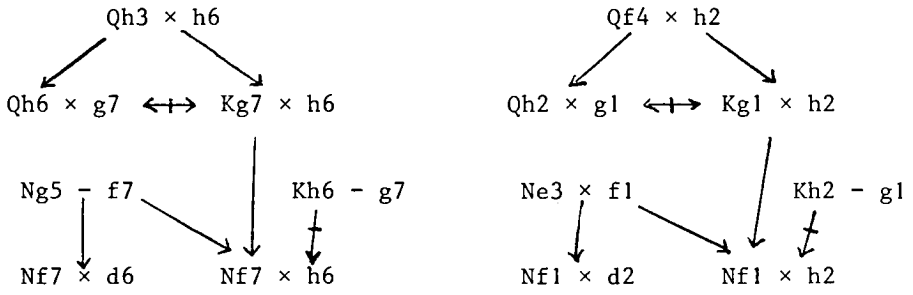


fig.3

In conclusion some interesting programs have been written. But their methods are difficult to apply to a game such as chess: there are too many situations and it is not easy to cluster them. A purely geometrical description of the situations does not seem convenient . The optimization of a set of weighting factors is probably good for positional playing. But it is not useful for finding combinations. Also, with statistical methods we cannot learn with just only one example.

1.2. How can a program hypothesize training information for learning without teacher.

It is easier to learn with a teacher who gives us in every situation the best move. But we can also learn without teacher. We always know who wins or loses a game. Some trials have been done using this information: we decrease the interest of the moves of the loser and increase the interest of the moves of the winner. But this

method is good only if there are not too many moves in a game. In the other case, the winner can play some bad moves. Similarly the loser can play several good moves and one very bad move : we will punish good moves. If we learn only with this information, the learning will be very slow. In some cases the program assumes that the opponent's move is the best one. This is dangerous if the opponent is not a good player.

Our program may find a combination in the record of the moves of a game. It looks if, for some succession of moves, the balance of the captures is good for one player. We compute the balance in using the usual values of the men (10 for Pawn, 30 for Knight...). We may object than in a real game between two good players there are not many combinations which succeed. It is true, but even in this case, the program can find many ideas of combinations.

Let us suppose that in the situation of figure 4 (there are only the interesting men) we have the exchanges :

Nf2 × g4 Nf6 × g4
Qa1 × h8 Qc7 × a5

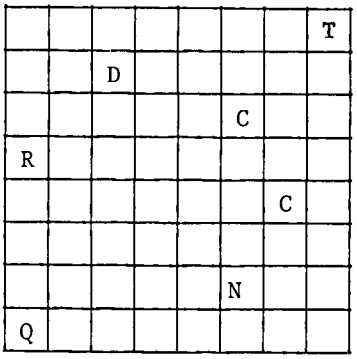


FIG. 4

The value of the balance is zero ; and however we can find two ideas of combination.

If there was no black queen in c 7 , we could win a knight with the combination :

Nf2 × g4 Nf6 × g4
Qa1 × h8

The signification of this combination is : a pinned man (the

Knight in f6) cannot protect.

After the move Nf2 × g4 there is a combination for Black where they win a knight :

	Nf6 × g4
Qa1 × h8	Qc7 × a5

The signification is : a pinned man M can capture if the man which is pinning M has also to protect another man.

From these exchanges, we can find two ideas of combinations. One because the opponent's reply could not exist in another situation. The other because there is really a combination after the first capture so that the opponent can reduce the balance to zero.

The knowledge of relations between moves is important for understanding what happened. Some relations are easy to find: a move creates another move. A move is characterized by changes on the board. But it is important to know the conditions by which the move is legal. In Pitrat [5,6] , I indicated how a program could find these conditions when we know an algorithm giving the legal moves.

Let the move : white queen in a1 goes to d4 . There are two changes : a1 becomes empty and d4 becomes occupied by the white queen. There are four conditions necessary for this move : the white queen is in a1 and the squares b2, c3, d4 are empty.

So, with each move, we can give the changes and the conditions. If a condition of move Q has been created by a change of move R, we indicate :

Q creates R	near move Q
R is created by Q	near move R

A move can create several moves, and a move may be created also by several moves. Let us give the beginning of the game G :

number	move	conditions	changes	created by	creates
1	e2-e4	e2,e3,e4	e2,e4		5,12
2	e7-e5	e7,e6,e5	e7,e5		11
3	Ng1-f3	g1,f3	g1,f3		11,15
4	Nb8-c6	b8,c6	b8,c6		9
5	Bf1-b5	f1,e2,d3,c4,b5	f1,b5	1	9,15
6	Ng8-f6	g8,f6	g8,f6		12
7	Nb1-c3	b1,c3	b1,c3		13
8	a7-a6	a7,a6	a7,a6		
9	Bb5×c6	b5,c6	b5,c6	4,5	10
10	d7×c6	d7,c6	d7,c6	9	14,20

number	move	conditions	changes	created by	creates
11	Nf3×e5	f3,e5	f3,e5	2,3	16
12	Nf6×e4	f6,e4	f6,e4	1,6	13
13	Nc3×e4	c3,e4	c3,e4	7,12	
14	Qd8-d5	d8,d7,d6,d5	d8,d5	10	16
15	0-0	e1,f1,g1,h1	e1,f1,g1,h1	3,5	19
16	Qd5×e5	d5,e5	d5,e5	11,14	18
17	d2-d4	d2,d3,d4	d2,d4		21,23
18	Qe5-f5	e5,f5	e5,f5	16	
19	Rf1-e1	f1,e1	f1,e1	15	
20	Bc8-e6	c8,d7,e6	c8,e6	10	
21	Bc1-g5	c1,d2,e3,f4,g5	c1,g5	17	
22	h7-h6	h7,h6	h7,h6		
23	Qd1-d3	d1,d2,d3	d1,d3	17	

We see that move 9 is created by two moves: 4 and 5 . Also move 17 creates two moves : 21 and 23. These connections may be useful for understanding what happened. If there is a connection between moves n and n+1 , move n+1 may be a reply to move n. For instance moves 9 and 10 or moves 12 and 13.

If there is a connection between moves n and n+2 (and more generally between n and n+2p), there is perhaps a plan.

1 creates 5 : after playing the pawn in e2, the bishop can play.

3 and 5 create 15 : we can castle if we play the knight and the bishop.

14 creates 16 : the queen creates a double attack at move 14 and captures one of threatened knights at move 16.

7 creates 13 : the knight moved at 7 protects the pawn e4.

So we have a precious help for understanding a game. But very often we have a succession of threats which are countered by the opponent. Nothing appears in the creations of moves :

21 : Bc1-g5 destroys for the Black a possibility of castling. So long castling does not occur.

20 : Bc8-e6 counters the threat of a discovered check by the knight. So, white cannot play this check.

9 : Bb5×c6 has several reasons. One of them is to destroy the threat : a6×b5 which no longer exists after 9 .

For finding that, it is not sufficient to study the connections

between moves such as move Q creates move R . We have to see : is a move a threat ? If it is, how the opponent destroys this threat ? For finding the threats, we have to find what we would play if we had the possibility of playing again. If there is something interesting , we try to play this combination after the move really played by the opponent. If it is not possible or if it is no longer interesting, we have understood what happened.

Let us specify that. The program knows some patterns which indicate in each situation where they exist which moves it has to consider. It has to understand the move Q , which is not explicable by connections with other moves. The program plays again after its move Q . With its patterns, it may find a combination, the first move of it being R. If, after Q, the opponent has played S , the program tries to play R after S . If R is no longer legal, all is explained. If R is legal, using the patterns already found, the program generates the opponent's combinations after R . If there is a combination for the opponent, the program has understood. It generates a tree such as that of figure 5 (where we assumed that our combination is reduced to move R). Ø stands for no move. From this tree it generates a new pattern. The information useful for learning is the knowledge of the two moves that have been played in the game : Q and S. If the program has not many patterns, it may happen that it does not find the friendly combination beginning by R or the opponent's combination after R. In these cases it does not learn. It has not understood what happened. It is also possible that move Q is really not a threat. In that case there is nothing to learn.

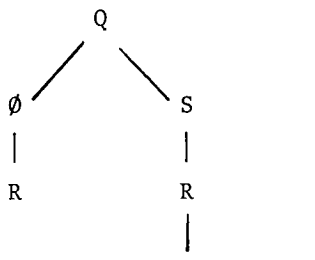
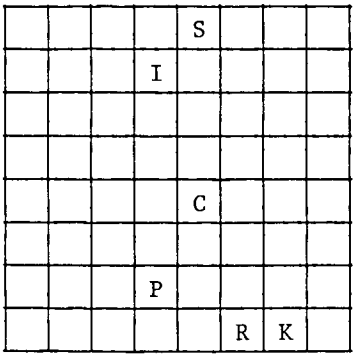


fig.5

We can find many other things. For instance it is possible that move S , which is a good reply in the situation, would not be legal in other situations. Does the opponent has other possibilities to escape ? So we generate with the patterns already known the moves that destroy the threat created by Q . After these moves we try to see if we have a combination. If it is true, we generate a new pattern. The threat would give a combination if S was not possible.

With the connections we see that d2-d3 creates d3×e4. There is no other connection of this kind. However many things happened. We have also to understand why the opponent hase losed his knight. It is not sufficient to attack a man for capturing it.



white to play

fig. 6

We have the following moves :

Rf1 - e1 d7 - d5

d2 - d3 Ke8 - f8

d3 × e4

Let us suppose that the program does not know many things: for generating a combination, it has only two patterns. The first one indicates to try captures. The second one indicates that, if a man is threatened, we can play it.

We have to understand the move Rf1-e1 (Q of figure 5). We play again and the program, finding a combination, generates the capture move : Re1×e4 (R of figure 5). The opponent has replied : d7 - d5 (S of figure 5). This move is not always possible. For instance if the pawn in d7 was in c7 . What happened if the opponent has no pawn in d7. It knows only one possibility for destroying R : play the threatened man. So the knight plays. But after these moves, the program looks if there is a combination and finds the capture move: Re1×e8 . So we generate a first tree given by figure 7.

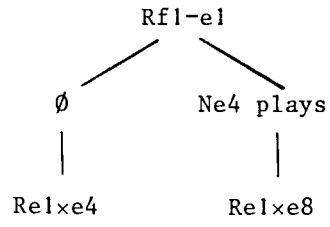


Fig.7

The tree of figure 7 will give a new pattern indicating how we can pin a man with a rook.

After Q the opponent plays S : d7-d5. Why ? After S , we try the move : Re1 × e4 (R of figure 5). Does the opponent has a combination ? Yes, it has the capture move : d5 × e4 . All is explained by the tree of figure 8.

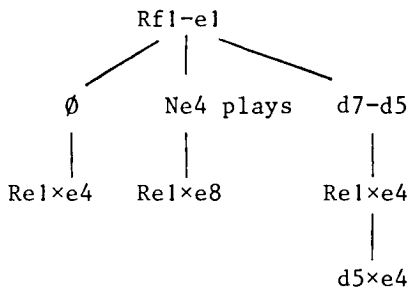


fig.8

The tree of figure 8 will give a new pattern which indicates how we can save a pinned man : we may protect it with a pawn.

The move d2-d3 gives a last pattern. It threatens d3×e4. If the opponent plays the knight, we find the combination : Re1×e8. The tree of figure 9 explains that

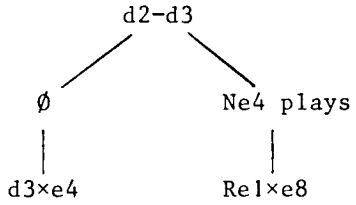


fig.9

The tree of figure 9 will give a new pattern which indicates that, if a opponent's man is pinned, we may try to attack it with a pawn.

We see that knowing few things, a program can understand more complicated combinations. These combinations create new patterns, and with these new patterns the program will understand more moves ... So, we have a program which improves its performances.

It is important to see that the information is in the record of the moves already played. This is useful, even if these moves are not good. Even a bad move can help. A move can be bad because another move would give a better combination : we have not found all that can be done, but we have however a combination; it is always good to keep it. So the program learns effectively in this case. The move can also be bad because, after it, the opponent has a combination. We can understand the opponent's combination, and it is a first result. But also in other situations the opponent, perhaps, could not play the move really played. So the move which was bad in the real situation could be good in another situation and we can keep another pattern. It was not bad to examine this move. It was bad to play it. The patterns are used to generate the moves which have to be examined. The selection of the good move is done by using the minimax procedure. This procedure is applied to the tree of the moves generated by the patterns.

For instance, in the game G , move 18 was Qe5-f5. If in the game, there was played instead: Qe5×e4 , it would be a bad move. If the opponent plays then Rf1-e1, we learn a way of capturing a queen. But the move Qe5×e4 would not be always bad. For instance if there was no white rook in f1 or if it was possible to bring a black man in e2 which would be protected in that square. So it is natural to create a pattern which generates such a move, although it was a bad move in game G .

With bad moves, learning is sometimes slower; but, in using them, the performances of the program do not decrease. We can generate a good set of patterns even if we give to the program only bad moves. Bad moves are troublesome when we are using statistical methods, not for our program.

We have to see more in detail : how the program generates the tree explaining what happened and how it translates this tree into a pattern which generalizes the situation.

2. THE LEARNING PROGRAM

The program has to understand what happened in a game and to generalize the combinations so that it can find them in another situation. For understanding what happened, it uses the patterns which were already generated. If it has more experience, it may use more patterns, it understands more moves and it stores more new patterns. The first time it studies a game, it generates some patterns. If it studies again this game, it is possible that it finds more patterns because the patterns generated the first time help it to understand more moves. If it finds the first time the interest of a discovered check, it can find the second time in another place the use of a threat of a discovered check.

Understanding a move is equivalent to build a tree. The program simplifies and generalizes the tree. Then it translates it into a pattern. If this pattern exists in some situation, the program generates a move similar to the one which was good in the original situation.

2.1. Generation of the tree.

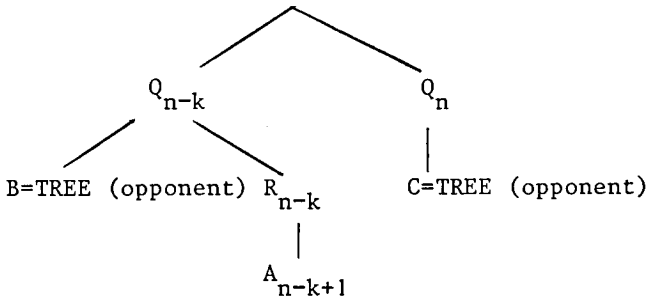
The program has a set of patterns (no pattern at the beginning). It uses them for understanding the moves of a game. We will explain in 2.5 how we use the patterns for generating moves. We introduce now the function TREE(P) : its value is a tree obtained by the search of a combination for player P in using the patterns. If we call this function in various situations, we have different trees. We take into account in this function that the opponent tries to destroy

our combinations. It generates also the moves which try to destroy a combination of the opponent. We have as data the record of a game:

$$\begin{array}{ll} Q_1 & R_1 \\ Q_2 & R_2 \\ .. & .. \\ Q_n & R_n \\ .. & .. \end{array}$$

The Q_i are moves of one player, the R_i moves of its opponent. We are trying to understand the move Q_n. Let B(Q) be the balance of move Q , computed with the value of the man captured (0 if there is no capture).

2.1.1. Case of a capture. Let Q_n be a capture move. If : $B(Q_n) - B(R_{n-1}) + B(Q_{n-1}) \ldots - B(R_{n-k}) + B(Q_{n-k}) + B(Q_{n-k})$ is positive (K=1,2,...) we consider the tree A_{n-k} of the figure 10 in the situation before Q_{n-k}. We put Q_n only if it is legal in this situation. A_n is a tree with only one move : Q_n .



Description of the tree A_{n-k}

fig.10

We consider the tree A_{n-k} only if the subtree beginning with Q_{n-k} gives a good result when we use the minimax procedure and if the subtree beginning with Q_n does not (if Q_n is legal). We will see in 2.2 in what cases we decide if this tree is interesting. If it is interesting, we resume the same procedure with Q_{n-k-1}. If it is not interesting we stop.

For instance, we are considering the move 11 of game G. We assume that we know only one pattern : generate capture moves. A_n is

the tree with one node : Nf3×e5 . With k=1 we have the tree A_{n-1} of figure 11. Q_{n-1} is Bb5×c6 , R_{n-1} is d7×c6. This tree will be considered as interesting. The program generates a pattern: if we want to capture a protected man, we have to make first, if possible, an exchange for capturing the man which protects.

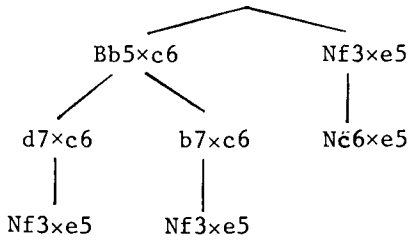


Fig.11

So we take k=2 and we have A_{n-2} of figure 12. The program will consider that it is not interesting (there are no connections between moves Nb1-c3, a7-a6 and the other moves). So we stop there.

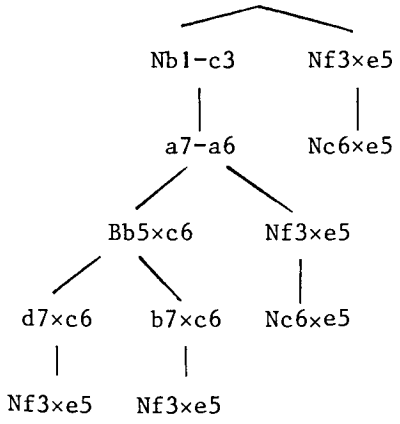


Fig.12

In the example of figure 4, we have the first combination when we study : Qa1×h8 and the second one when we study: Qc7×a5 with k=1 in these two cases.

2.1.2. Case of threats. We have just seen what we can find when there was a capture. We have also to discover the hidden

struggle, succession of threats countered by the opponent. After each move, we look if there were a combination if we could play again. If so, we generate the tree of figure 13.

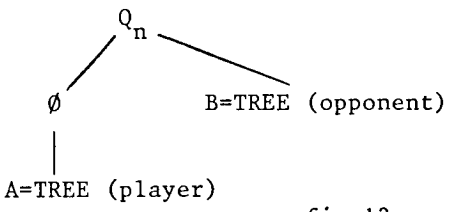


fig.13

We eliminate the opponent's moves at the second level if it is possible that these moves were not there. For instance if in the tree A of figure 13 there is the move Qa5×c5 and there is the opponent's queen in c5 , we cannot eliminate the move Qc5×a5 if it is generated in the tree B . If Qa5×c5 is legal, Qc5×a5 is always legal. We cannot eliminate also the move : Qc5 plays somewhere. But we can eliminate d7-d6 which protects the queen : it is possible that in another situation there is no pawn in d7. We can do this if the fact that there is an opponent's pawn in d7 is not necessary for the threat. For instance, in the tree A , the player must not capture d7. If the minimax procedure indicates that the tree of figure 13 is interesting, we try to generate a pattern. With the board of figure 6, when we try to understand the move Rf1-e1, we generate the tree of figure 7 .

But we can generate another tree. The opponent destroys probably the threat with move R_n. If it was already generated in the tree B, we have nothing to learn: we knew already this possibility. But if it was not generated, we build the tree of figure 14.

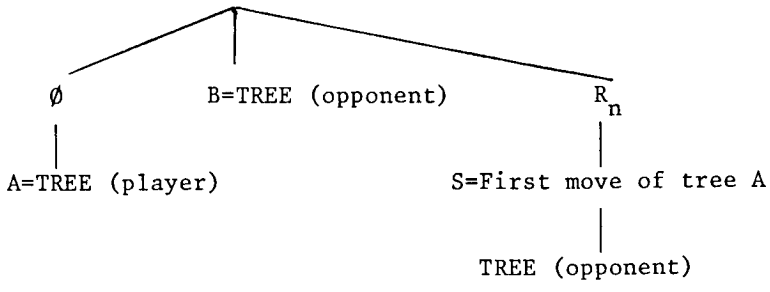


fig.14

We compute with the minimax procedure if this tree is not wrong for the opponent. If so, the opponent can destroy our threat ; we keep this for looking if it is possible to generate a new pattern indicating how we can counter this threat. With the board of figure 6, when we try to understand how the threat : Re1×e4 has been countered by the move : d7-d5, we generate the tree of figure 8, which gives a second pattern. The tree of figure 9 is generated when we try to understand the capture move d3×e4 , in using the methods of section 2.1.1.

So, we can find ideas of combinations in parts of a game where there was no exchange. We can also find new ways of destroying threats which are efficient, since no capture occurs. Naturally we can also find ideas of combination when some capture occurs. With the moves given for figure 6 , we can find three new patterns : it is interesting to pin a man ; if an opponent's man is pinned, try to attack it; if one of our men is pinned, try to protect it. However we knew only two patterns : generate capture moves; if one of our men is attacked, try to play it.

2.2. Simplification and generalization of a tree.

When we have built a tree with the preceding method, we try to generate a pattern which finds the combination in similar situations. But before, we have to eliminate many uninteresting moves and to generalize the other ones. Then we can create the pattern.

2.2.1. Simplification of a tree. We use two criteria. First, in a subtree generated by the function TREE , we keep only the moves useful for a good result of the minimax procedure. If the subtree itself is not good, we eliminate it.

In the board of figure 6 , after the moves : Rf1-e1 ; d7-d5 ; Re1×e4; we do not keep the move Ke8-f8 which can be generated for destroying the check : Re4×e8. The move Ke8-f8 is not good since the opponent (White in this case) wins a knight if Black plays it.

Always in the board of figure 6 , if we have the sequence of moves : Rf1-e1 ; d7-d5 ; d2-d3 ; ∅ ; d3×e4 ; d5×e4 ; we eliminate d5×e4 : the balance is not good for Black even if it plays it. It is not sufficient to capture a pawn. We keep, after d3×e4, only black moves which are necessary for winning at least a knight. In this case, there is no such move.

Secondly, we do not keep the moves which have no connection with any other move. More accurately, we replace by a "no move" the moves which do not create capture moves deeper in the tree and which do not destroy opponent's moves in other branches (move Q destroys move R if at least one of the changes of move Q destroys

one condition of move R) and which are not created by moves higher in the tree.

In the tree of figure 8 , we keep all the moves :

Ne4 plays : destroys Re1×e4 in other branches.
Re1×e8 : is created by Ne4 plays.
d7-d5 : creates the capture move d5×e4.
Re1×e4 : creates also the capture move d5×e4.
d5×e4 : is created by the moves d7-d5 and Re1×e4.

In the tree of figure 12, we eliminate Nb1-c3 and a7-a6 which have no connection with other moves. In the same way, when we study the move 16 of game G : Qd5×e5, we generate a tree with moves 14 and 15, since move 16 is a capture move (see 2.1.1.). But we eliminate move 15 (short castling) which has no connection with the other moves.

With this method our tree has some cohesion. If there are two no moves in sequence, we remove from the tree these moves and the following subtree. In figure 12, we are in this case after replacing Nb1-c3 and a7-a6 by no moves. So we erase the left part of the tree. After removing moves, we verify if the tree is always good when we use the minimax procedure. If not we eliminate the tree. This occurs with the tree of figure 12.

2.2.2. Generalization of the tree. There are several generalizations.

On the player. We suppose that the same player is always trying to find a combination. We are calling it Friend (F). Its opponent is Enemy (E). If we want to try a pattern, we have to know if White is Friend or Enemy

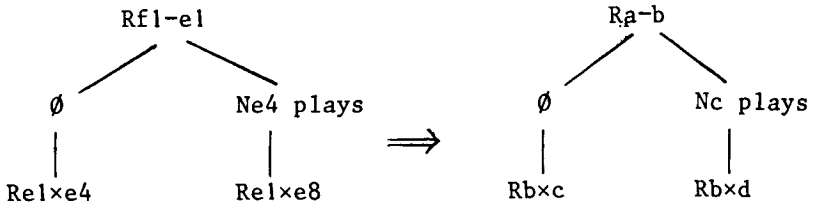


fig.15

We replace in the moves of the tree the real squares by variables. The tree at the left of figure 15 gives the tree at the right. With the moves, we keep the conditions bearing on squares which occurs in some other place of the tree. In figure 15, we keep with

move : Rb×d , that square c empty is a condition of this move. We do not keep the other conditions.

If a man of nature Y occurs only once in the tree in a move such as Ya×b or Ya goes to b, and if this move has no condition on a square which occurs somewhere else in the tree, we consider that the real nature of Y has no interest. We keep only the fact that this man is Friend or Enemy. For instance when we study the move Qc7×a5 of the game given with figure 4, we generate the tree of figure 16. It is essential that there is a queen in square a1 : it occurs in two moves: Qa1×a5 and Qa1×h8. In the same way, the queen in c7 occurs in two moves: Qc7×a5 and Qa1×a5 where it is captured (the two occurences of the same move Qc7×a5 are counted only once). But the knight in f6 occurs only in one move : Nf6×g4. We accept any nature for the man in this square. Instead of having a pattern on a pinned knight, we have a pattern on a pinned man.

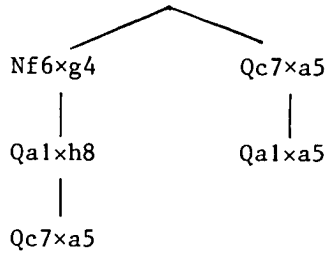


fig.16

If a man occurs at most in one move where it plays (without condition on an intermediary square which occurs elsewhere) and if this man is captured (in any number of moves), we replace it by a man such as its value may be greater (lower) if the man is an enemy (friendly) man. The minimax procedure, which gave a good result, would give a better one. We indicate this by $\geq$ or $\leq$ preceding the nature of the man. For instance in figure 16, we can have in the initial situation in square a5 : $\geq$ RE instead of an enemy rook. We may have there an enemy queen.

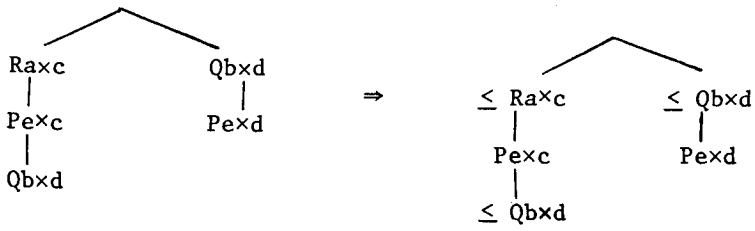


fig.17

In figure 17 we give a tree and its generalization. In the initial situation we have a bishop in d and a knight in c. The pawn protects a knight in c and a bishop in d. The knight in c can be captured by a rook in a. The bishop in d can be captured by a queen in b. After generalization we have $\geq$ N in c and $\geq$ B in d. The combination is still good if there is for instance a bishop in c, a rook in d, if the bishop in c can be captured by a bishop in a, and the rook in d by the queen in b.

If the first move of the tree is a capture move, we compute if the balance of the tree is always good if there were no capture at this level. If so, we put Na-b instead of Na×b.

If we have several moves at the same level which are similar, we keep only one of them. If we have : Ea×c and Eb×c, and a and b do not occur elsewhere in the tree, we keep only one of these moves. In the same way, if we have as sons of the same move : Na-b; Na-c; Na-d; and if b, c, d do not occur in other place, we put: Na plays.

At this stage, we have eliminated many of the uninteresting moves and generalized the remaining ones. We have now to see how we can generate a pattern from this tree. With this pattern, we must be able to find a combination in similar situations.

2.3. Generation of patterns.

A pattern is an ordered set of moves with a tree. If the moves are legal in a situation, the tree indicates when we have to generate these moves. In that case, it is natural to try the combination; but we are not sure that it will be always good, the combination can fail for some unexpected event.

For seeing if a pattern exists in some situation, we have to see if moves such as : $\leq$ NFa×b are legal; a and b may be unknown. In the pattern, we have the initial state of these squares. For instance $\geq$ RE in a and QF in b . We may have more precisions on a move : for a move such as : RFa×b , we may know that there is the condition : an intermediary square c is empty, and that the initial state of square c is $\leq$ BF .

It is important to distinguish :

- what is the state of a square when the move is played.
- what is the state of a square in the initial situation.

In the figure 17, we have the move Pe×d. In the initial situation, there is in square d an enemy man, its nature being a bishop, a rook, the queen or the king ($\geq$ BE). When the move is played, there is a friendly man which is the queen, a rook, a bishop, a knight or a pawn ($\leq$ QF).

Let us assume that we want to see if it is possible to find squares a and b such as : RFa×b is legal, when, in the initial situation we have $\leq$ NF in square a and $\geq$ RE in square b. Let A be the set of squares where the player Friend has a pawn or a knight in the initial situation. Let B be the set of squares where its opponent has a rook, the queen or the king. We can :

1. Check if a rook can go from $a_i \in A$ to $b_j \in B$.

2. Find the squares b_{ij} where a rook can go from $a_i \in A$ and see if in b_{ij} there are an enemy rook, queen or king.

3. Find the squares a_{ij} from which a rook can go to $b_j \in F$ and verify that in a_{ij} there are a friendly knight or pawn.

In the three cases, we have also to verify that the conditions, if any, on the intermediary squares are realized. After choosing one of these three methods, we have some couples (a,b) which can be good. We try to verify the end of the pattern with all these couples successively.

For generating moves by this way, the program has three subbroutines which :
- verify that the man of nature N can play from a to b .
- generate all the moves where a man of nature N plays from a.
- generate all the moves where a man of nature N can go to b.

It is important to order the moves of the pattern. We try to verify first the moves which generate the smallest number of couples (a,b). The best move in this respect is not always the move at the root of the tree.

For instance, in figure 15, the move Ra-b is at the root of the tree. In the initial situation there is a friendly rook in a ; b is empty. There are many couples (a,b), because there are two rooks, a rook can play many moves and there are plenty of empty squares. It is more interesting to take first Rb×d with the conditions : b is empty, the enemy king is in d and there is an enemy knight in some intermediary square c in the initial situation. There are only one king and two knights, and they have to be alined on a direction of the rook, with nothing between them. If this occurs, the empty squares beyond the square c may be the square b. We will see later if we can bring a rook in square b. It is likely that a human player uses such methods for finding a combination pinning an enemy man. If two important enemy men are alined, we try to find if we we can bring on the same line a friendly man (which can move along this line).

For ordering moves in the pattern, we compute their "indetermination". We take first the move which has the lowest indetermination.

Then some squares will be known. We compute again the indeterminations of the remaining moves ... and we stop when we have taken all the moves. When we have finished, we have the pattern : an ordered set of moves and the tree indicating when we have to consider them. The tree is the tree generated for understanding what happened. We replace in it the real moves by references to the corresponding moves in the pattern.

We have now to see how we compute the indetermination of a move. Let I(a) be the indetermination of a square a. It is the number of squares which have the wanted state in the situation at the commencement of a game :

King, Queen : 1
Rook, Bishop, Knight : 2
Pawn : 8
Empty : 32

If we know that we have a piece whose value is greater or lower than some man, we add the indeterminations of the possible men.

$\geq$ Knight : Knight or bishop or rook or queen or king.
$I(\geq N) = 2+2+2+1+1 = 8.$
$\leq$ Rook : Rook or bishop or knight or pawn.
$I(\leq R) = 2+2+2+8 = 14.$

For computing the indetermination of a square, we use the nature of the man which is in this square in the initial situation. If the square has been already defined by a preceding move of the pattern, its indetermination is one.

Let J(N) be the indetermination brought by a man of nature N. We choose the following numbers which are nearly the number of moves of the men:

King : 6	Queen : 15	Rook : 10
Bishop : 8	Knight : 6	Pawn : 2

We use the man which really plays the move, not the man which is in the starting square in the initial situation.

The indetermination of the move : Na-b is the lowest of the three values : $I(a)\times I(b)$; $I(a)\times J(N)$; $I(b)\times J(N)$. We have three ways for generating the move; we choose the one according the lowest value. If it is $I(a)\times I(b)$, we find the squares a and b and we verify that the man N can go from a to b. If it is $I(a)\times J(N)$, we find the square a, then the squares where a man of nature N can go from a and we verify than the square b is good. If it is $I(b)\times J(N)$, we find the square b, that the squares from where a man of nature N can go to b; we verify that the square a is good.

For instance, let us consider the move : KFa×b. In the initial situation a is occupied by the friendly king and b by a friendly knight or pawn ($\leq$ NF). We have :

I(a)=1	I(b)=10	J(K)=6
I(a)×I(b)=10	I(a)×J(K)=6	I(b)×J(K)=60

The indetermination of the move will be 6. The method of operating is : find the square a occupied by the friendly king. Squares b are squares where a king can go from a. We keep only those which are occupied by a friendly knight or pawn. It is better than to find squares b occupied by a friendly knight or pawn, then to find the squares a from which a king can go to b and then verify that the friendly king is in a.

If we have one or several intermediary squares on which we have a condition, we decrease the value of the indetermination of the move according to the value of the indetermination of the intermediary squares.

Let us examine the generalized tree at the right of figure 15. The move : RFa×d has the condition : the square c is empty when we are playing this move. The initial situation of the squares is : a : RF ; b : empty ; c: $\geq$ NE ; d : $\geq$ KE. Let us compute the indetermination of the moves : Ra-b and Rb×d.

Move Ra-b . I(a)=2 . I(b)=32 . J(R)=10.
I(a)×I(b)=64 . I(a)×J(R)=20 . I(b)×J(R)=320.
So, the indetermination is 20.

Move Rb×d . I(b)=32. I(d)=1 . J(R)=10.
I(b)×I(d)=32 . I(b)×J(R)=320 . I(d)×J(R)=10.
The indetermination is 10. In reality this value is still decreased by the condition : there is an enemy piece in square c. So, we prefer to see if the king and an enemy piece are alined on a rank or a file. If so, we look then if a friendly rook can come on a square beyond the enemy piece.

We notice that after we have chosen the move Rb×d , the indetermination of Ra-b changes : b is known, so I(b)=1. The indetermination of Ra-b is : I(a)×I(b)=2×1=2. Even in the case where there would be many moves, this move would be selected quickly after Rb×d.

2.4. Storing the pattern.

When we generate a pattern, we have to verify if it has not already been stored identically or on a more general form. In this case, we do not store it again. If it is stored on a less general form, we eliminate the old pattern.

A pattern A is more general than a pattern B if we have the same tree and if all the moves of A are in B on the same form or on a more general form.

A move Q is more general than a move R if the same man is played and if the initial state of the squares are the same or more general.

An initial state I of a square S is more general than an initial state J of the same square, if the square is occupied by men of the same player, and if all the states of J are in I. For instance if I is $\geq$ NE and J is $\geq$ RE. I is knight or bishop or rook or queen or king which includes J which is rook or queen or king.

The patterns are stored in a tree. At each node of this tree, we put a move of the pattern. At the end of each path from the root, we put the tree which we develop if the pattern occurs. So, we have not to do several times the same generation of moves if several patterns begin in the same way.

If a move of pattern P_1 is more general than a move of pattern P_2 (but pattern P_1 is not more general than pattern P_2), we put the more general move. From this node, there are two paths describing the end of these two patterns; one describes the remainder of P_1, the other the remainder of P_2. But we indicate than we can take the second path only if some restrictions are true.

Let two patterns begin with move Qa×b. In a the initial situation is the same. But in b the first pattern has $\geq$ NE and the second $\geq$ RE. We have the tree of figure 18. If we find some possibilities for Qa×b , we examine always the continuation of the first pattern. But we examine the continuation of the second pattern only if there is a rook, the queen or the king in b.

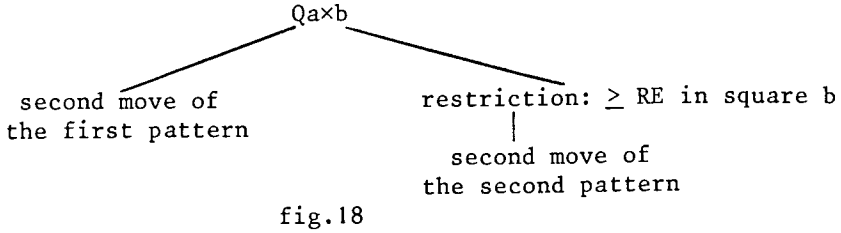


fig.18

2.5. Using the pattern.

Let S be a situation. We want to execute the function TREE which generates the tree of the eventually interesting moves. First we look what patterns can be applied in S. We have a tree. If there

is a combination, i.e. if the minimax procedure returns a positive value, we try to see what the opponent can reply. If there is a threat, we generate, using the patterns, the moves which counter this threat. After the friendly moves necessary for the combination, we generate TREE (enemy). If the opponent with the move Q destroys our combination, we generate after Q : TREE (friend). So the procedure TREE is called recursively. We execute this while we can do something. We stop if the opponent cannot destroy our combination or if all our combinations are refuted relentless.

This method limits the growth of the tree. Let us assume that we know only one pattern : this pattern indicates that we have to consider the capture moves. With this method, we do not generate all the possible arrangements of capture moves. If we have the succession of moves : Qa×b ; Pc×b ; Rd×e ; and if there was a rook in b and a bishop in e , we do not examine the situation after the third move, even if there is after a capture move. The third move is not a dangerous move for the opponent, since the balance after this move : B(R) - B(Q) + B(B) is good for him. If there was good moves for the friendly player after Rd×e and after the possible enemy moves, these good moves had to be generated in the tree of another pattern. If such pattern has not yet been discovered, the combination will not be found.

2.6. Results.

The program was written in FORTRAN for CDC 3600. Some possibilities of list processing had been added to FORTRAN.

Let us give for the game G the patterns found by the program when it knows at the beginning : capture a man ; for countering a capture move, play the threatened man. These patterns were generated during a very short part of a game, given previously and which showed their interest.

At move 4 : Nb8-c6 , we generate :
Initial state of squares :
a : NE ; b : $\leq$ PF ; c : NF ; d : empty or enemy.
If we have the moves :

1 NEa×b
2 NFd×b
3 NFc-d .

We consider the tree reduced to the sequence of moves: 3-1-2. If we have a pawn threatend by a knight, we can protect it with a knight.

At move 7 : Nb1-c3 , we generate exactly the same pattern than at move 4. We do not store it again.

At move 11 : Nf3 e5, we generate :

Initial state of squares :
a : $\leq$ BF ; b : $\geq$ NE ; c : $\geq$ PE ; d : $\leq$ NF ; e : E .
If we have the moves.

1 $\leq$ BFa×b
2 $\geq$ NEb×c
3 $\leq$ NFd×c
4 E e×b

We consider the tree of figure 19. In the nodes we indicate references to the moves of the pattern.

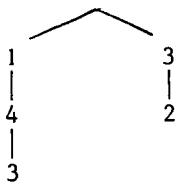


fig.19

This is a special case of the combination which indicates that it is interesting to exchange our man in a with the enemy man in b protecting another enemy man in c, before capturing this man in c. In this case the pattern is not so general. In c there may be a man of any nature. But we cannot see that the combination holds if there were rooks in a and b.

At move 16 : Qd5×e5 , we generate :
Initial state of the squares :
a : QF ; b : empty or enemy ; c : $\geq$ NE ; d : $\geq$ NE.
If we have the moves :

1 QFa-b
2 QFb×c
3 QFb×d
4 $\geq$ NEc plays
5 $\geq$ NEd plays

We consider the tree of figure 20.

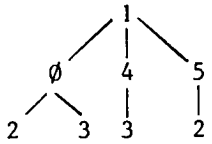


fig.20

It is necessary to indicate what patterns the program knows at the beginning. If it had more patterns, it could understand what

happens for instance at move 19. Rf1-e1 , where there is a threat of discovered check. As the program had no pattern for discovered check, it could not understand this at this time.

3. CONCLUSION

The program can learn some useful patterns from given games. But there are many ways to improve this program.

This method of learning is based on the possibility of finding in a game some indications of the presence of a combinations : in this case, the capture of men. There are games, other than chess, where a combination is not based on captures of men.

The generalization is not sufficient. We have to generate several patterns for describing what is for us only one kind of combination.

Sometimes , coincidences have for consequence to keep unnecessary moves. If the opponent plays on some square S for escaping a threat and if S is elsewhere a condition of a move, we may individualize this square and think that there is some connection between the two moves. For this reason, the generated pattern will not be as general as it could be.

The function TREE which generates a tree in using the patterns limits the growth of the tree. But it would be necessary to be more drastic. For instance, when we are deep in the tree, it would be good to consider only the simpler patterns. A human player is not able to see a very complicated combination which occurs many moves ahead. But we have to define what is a simple pattern. It would also be interesting to generate only the new combinations. It is not good to generate at each level always the same beginning of combination and see each time that this combination is not good, always for the same reason. This is desirable, but difficult to implement.

For understanding some part of a game, we have some troubles with superfluous moves. For instance if, in the middle of a combination, the opponent gives a check or exchanges one man, these moves having no connection with the preceding and the following moves. The program cannot connect what is before and what is after the superfluous move.

The pattern describes what moves are necessary. It would be important to include moves which must not exist if we want to try the combination. For instance, if we have not to capture a man which is protected, the pattern could verify that there is no enemy move recapturing our man. This has not been done.

There are many ways for improving our results. But this experiment showed that it is possible to learn to play a game in using another method than rote learning or optimization of a set of weighting factors. It is important to learn by understanding the reasons of the moves and by generalizing logically what happened. With this method we can learn with one example and even, in some cases, find an idea of combination from one example in which this idea does not succeed. We can also learn even from poor moves.

APPENDIX

Representation of a chess board. We use only one letter.

	White	Black
King	K	S
Queen	Q	D
Rook	R	T
Bishop	B	F
Knight	N	C
Pawn	P	I

Representation of a move in a game. We use the algebraic system. The men are designated by their initial, except the knight by N . The eight files from left to right are lettered consecutively a to h. The eight ranks are numbered consecutively 1 to 8. A move is recorded by the designation of the man moved (not being a pawn) followed by the designations respectively of the square it occupied and then the square to which it had been moved. 0-0 is an abreviation for short castling. × indicates a capture. Ø is the representation of a no move.

Move in a generalized pattern. We indicate the designation of the man which can be preceded by $\geq$ or $\leq$. It is omitted when any nature is possible. After the nature, we have, when necessary, F or E according as the player (Friend or Enemy) who plays the man. Then, the names of the square that the man occupied and the square to which it had been moved. We can have the second name replaced by the word "plays" if this square has no importance.

REFERENCES

1. Samuel, A.L., Some studies in machine learning using the game of checkers, in *Computers and thought* , Feigenbaum, E.A. and Feldman, J. Eds., Mc Graw Hill, New York, 1963, 71.
2. Waterman, D.A., Generalization learning techniques for automating the learning of heuristics, *Artificial intelligence* , 1,121,1970.
3. Samuel, A.L., Some studies in machine learning using the game

of checkers. II - Recent progress, *IBM Journal* , 11,601,1967.

4. Newman, C. and Uhr, L., BOGART : a discovery and induction program for games, in *ACM 20th National Conference*, 1965,176.
5. Pitrat, J., Realization of a general game playing program, in *Information processing 68* , North Holland, Amsterdam, 1969 , 1570.
6. Pitrat, J., A general game playing program, in *Artificial intelligence and heuristic programming*, Findler, N.V. and Meltzer, B., Eds., Edinburgh University Press, 1971, 125.

PITRAT : Discussion

MICHALSKI

As you don't mention any other work can we assume that this method is original ?

PITRAT

I have no knowledge of learning programs for chess. Samuel's program was for checkers and he is quite lucky to have a program that works. I think it would not apply to chess.

SIKLOSSY

I heard of two things. One, that in Samuel's case you could take only a feature and have as good results. Then I heard of a program being developed in Edinburgh which has a quite similar representation.

PITRAT

I know a paper by Griffith (Artificial Intelligence vol 5, n°2, Summer 74, p. 137-148) where there are many interesting experiments. In one of them the moves are classified in four sets. If there are several moves in the best set, the move to play is randomly chosen. However the results are as good as those of Samuel. As for Edinburgh I didn't read anything on it.

MICHALSKI

Your program learns combinations from human examples. Can it find patterns not found by humans ?

PITRAT

Of course it can give by chance, solutions which were not known, as it combines patterns. But it should be run on games played by better players.

SIROVITCH

What about the performances at play and not at learning ?

PITRAT

I have written a program using those patterns without learning. It gives good results. The program with learning is of course less efficient.