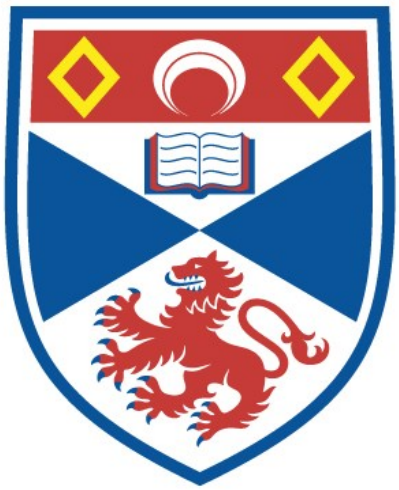
University of St Andrews



Full metadata for this thesis is available in
St Andrews Research Repository
at:
http://research-repository.st-andrews.ac.uk/

This thesis is protected by original copyright

AN EXPERIMENT IN MACHINE LEARNING USING THE GAME OF THREE DIMENSIONAL NOUGHTS AND CROSSES

Thesis presented for the Degree of M.Sc.

by M. J. Doake, M.A.(Hons.), St. Andrews, 1972

DECLARATION

I hereby declare that this thesis has been composed by myself; that the work of which it is a record has been done by myself; and, that it has not been accepted in any previous application for any higher degree. This research concerning machine learning was undertaken from October 1969, the date of my admission as a research student for the degree of Master of Science (M.Sc.).

CERTIFICATE

I hereby declare that the conditions of the Ordinance and Regulations for the degree of Master of Science (M.Sc.) at the University of St. Andrews have been fulfilled by the candidate, M. J. Doake.

A. J. T. Davie

SUMMARY

A program was written to play the game of three dimensional noughts and crosses on a computer. The program was written in such a way that it would 'learn from experience'. The emphasis therefore was not on producing an unbeatable program, but one which would improve its own play. This is achieved by using a backtrack analysis, which is called every time a game is won or lost. It works by starting from the winning board position and backtracking, i.e., unplaying the last move played, then the second last etc., until it reaches the board position where it thinks the critical move was made. This board position is analysed and stored in a generalised form, so that next time that board position, or one essentially similar, is found, the program will recognise it. The analysis is done in terms of the pattern formed by the counters on the board. It is important that the generalised description of the pattern should capture the essential, and only the essential elements of a pattern, and this the program is only partially successful in doing. Alternative methods of backtracking are discussed, and suggestions made as to how the program might be developed.

The program also has a selective look-ahead procedure which is heuristic in method. The look-ahead uses the list of patterns produced by the backtrack analysis. Initially, therefore, it will only look for those patterns given in the rule-book, viz., four in a row, which constitutes a win. The list of patterns will be built up with experience. The program looks ahead at several different levels. It looks to see :

a) if a pattern exists in the actual board position

b) if a pattern can be formed by either side in one move

c) if a pattern still exists if (b) is true (a pattern can be formed in one move) and the opponent does his best to stop this pattern.

Only one square is selected as the best one to stop a particular pattern, and this square will be dictated by the nature of the pattern found. The original analysis of a pattern during the backtrack analysis indicates which square this is.

The program has the facility to play both sides in a game, or to play against a human opponent. Games of both kinds are listed and discussed.

INDEX

I INTRODUCTION Why play games?

Machines which play games have a long and varied history. Amongst the first was the "chess playing automaton" constructed by Baron Kempelen in 1796 which "computed" its moves thanks to the efforts of a dwarf inside it, gaining its inventor considerable ill-gotten fame and fortune. More recently the motives for mechanising game-playing have become, we hope, less suspect. Interest is now focussed not on machines specially constructed to play a game, but on programming already existing digital computers to do so.

Programming computers to play games seems to provide a special sort of challenge to those who do it, especially where the game involved is an "intellectual" game such as chess or draughts. A game like chess is thought of as involving real intellectual effort, the use of intelligence. If a machine could be successfully programmed to play this sort of game it would prove that machines could be used for work other than the dull slavish routine which is thought of as "mechanical". This seems to have been sufficient motivation for some; to prove that a machine could be said to be displaying "intelligence". For others a successful game-playing program would mean "one would seem to have penetrated to the core of human intellectual endeavour" (NEW 58). Similarly "man can solve problems without knowing how he solves them"; a successful game-playing program would capture and mechanize human decision making and problem solving and so "add to his kit of tools for controlling and manipulating his environment" (NEW 58).

A more straightforward motive is that of improving programming techniques and learning new ones. Games provide complicated problems, but have a certain regularity because of their rules, which one would

not find in a problem taken from "real life". Hopefully the techniques learnt in the solving of a complicated game problem can later be used to solve an economics or business problem. Games it is thought, retain many of the characteristics of real life problems while eliminating many of the worrisome complications; thus providing an opportunity to isolate in pure form the logical structure underlying real-life problems.

A specific example of this is the technique of "learning from experience." A program which could learn could be used to do a lot of trivial and detailed work which otherwise would have to be done by hand, and would be very time-consuming.

II BRIEF HISTORY OF GAME PLAYING PROGRAMS

We are concerned here only with programming computers to play games for which there exists no known algorithm which can guarantee a win, as can be very successfully done with such games as Nim or the normal game of noughts and crosses. However, programming a computer to play such games is essentially trivial, unless the game is played without reference to the algorithm.

The names mentioned below are only those of pioneers in the field of game-playing programs, or those whose work is directly connected with our own research.

(i) Claude Shannon

One of the first important names in the history of mechanized game playing is that of Claude Shannon (SHAN 50a, SHAN 50b). Although he did not actually write a program for a digital computer, he discussed the problems involved in doing so. The answers he gave built up a frame-work on which almost all subsequent chess playing programs have been based.

The rules of chess (and most other games), ensure that it is a finite game, which must end in a win, loss or draw. It can therefore be completely described as a branching tree, the nodes corresponding to the positions and the branches to the alternative moves from each position.

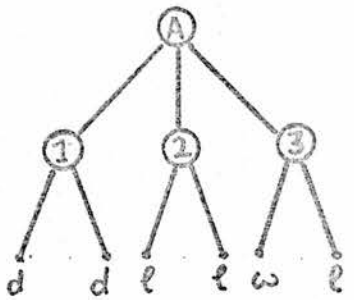


Figure 1

This simplified diagram illustrates the point. A is the starting board position. White has 3 alternative moves, 1, 2, or 3. White's move will be followed by Black's, and for the purposes of this example it is assumed that all of Black's moves lead to board positions with known results.

Move 1 will lead to a draw, no matter what Black does. Move 2 can only lead to a loss for White. Move 3 could lead to a win or a loss, and the outcome will depend on what Black does. So far as White is concerned Move 1 is his best move as it ensures a draw, whereas Move 2 ensures a loss, and Move 3, since the opponent has the issue of the game in his power, is almost certain to result in a loss for White.

In a game of chess one could in principle examine the whole tree comprising all the alternative moves and their continuations, until one had worked out all the possible endings to the game. It would then be relatively simple to work out the best move. The procedure would be to work backwards from the terminal positions, as we did in the above example, and at each branching point decide which was the best branch to take from the point of view of the side making the move. If White

is looking ahead. he will choose the best branch for himself when it is his turn. and assume that Black will choose the branch which is worst for White when it is his turn. That is, looking back up the tree, White choses his path by alternatively maximising and minimising. This procedure is known, for obvious reasons, as minimaxing.

However from any given board position there are, in chess, about 30 possible alternative moves and this means that the number of alternatives to be considered looking ahead to the end of the game would be astronomical, and could not be contemplated, even with the use of the fastest modern computers.

Shannon's suggestion was that one should look ahead to a certain depth examining all alternative moves. At this depth all the resulting board positions should be evaluated in some way. Then one could minimax back from the board positions reached in the look ahead to find the best move.

To evaluate a board position Shannon suggested a numerical measure formed by assigning values and weights to various factors which are considered important by chess experts, e.g., the number and nature of the pieces on the board.

Shannon also pointed out the importance of evaluating a board position only if it had a certain amount of stability. If in the next move something is going to happen which will drastically alter the value of a board position an evaluation at this point will be misleading.

He also suggested that instead of examining all possible alternative legal moves to a fixed depth, one should only examine promising moves and not explore the other at all.

(ii) A. M. Turing

Turing (TUR 50, 53) is the next important name. He too was concerned with chess playing. He wrote a program which was not implemented on a computer but which could be simulated by hand. The program fitted into the Shannon framework. His main contribution was a clear definition of what he called a "dead position". He would only evaluate a board position if it was dead. A dead position was one which was more than 2 moves ahead of the actual board position, and from which the next move was not a capture, recapture, or mate. The 1st condition of course was arbitrary - Turing chose to consider all alternatives to a depth or ply of two, (a ply being a move by 1 person, 2 plys 1 move by each side), then evaluate those which were dead. The rest of the alternatives were explored until they were dead.

(iii) A. Bernstein

Bernstein's (BER 58a, BER 58b) program made another major contribution to game playing technique. Instead of examining all legal alternatives to a certain depth, Bernstein's program used drastic selection and only explored a fraction of the possible paths in the tree. A series of subroutines suggested which moves were worth considering. Each subroutine corresponded to some feature of the game e.g., king safety, development, defending own men etc. The subroutines functioned in order of priority, each one in turn suggesting plausible moves until 7 moves had been suggested. These subroutines were used at every stage of the look ahead, so never more than 7 moves were considered from any one board position. The program looked ahead to a depth of 2 moves. This meant that out of a possible 800,000 (approximately) alternative board positions the program evaluated only 2400.

This selection was too drastic in that the program tended to overlook simple moves which had important consequences and it made bad blunders. This was hardly surprising with so much selectivity. What was surprising was that in spite of this drastic cut in the number of moves evaluated, the program played a reasonable game at all.

The introduction of selectivity made the program more complicated and it took longer to examine each position than earlier programs. However, if selection could be properly implemented it was obviously more profitable to consider fewer moves to a greater depth.

(iv) Newall, Shaw and Simon (NEW 58)

The aim of this team in writing a chess program is not only to play good chess, but to analyse chess situations in much the same way as human beings. They are interested in mechanical simulation of human thought processes and feel that in any case the best chess program will be the one which most closely simulates human chess player's analysis. The complex nature of a human chess player's thought processes is dictated by the game, and a good chess program should have a similar complexity.

Newall, Shaw and Simon (NSS) are convinced that careful but drastic selectivity is essential, and this is the dominant theme in their program. They use Bernstein's idea of a move generator, but take it a stage further and are rather more careful.

Given a board position, the first thing the NSS program does is to decide which goals are appropriate to this board position. These goals represent different features of chess, such as King safety, material balance, centre control, denying stalemate etc. It can easily

be appreciated that all these goals are not relevant to every stage of the game - the last, for example, will only be relevant towards the end. This list of goals is important because the goals which the program decides are relevant to a particular board position then control the rest of the process of choosing the next move - i.e., which moves to examine, which continuations to explore, the evaluation when a dead position is found, etc. This is an important new step which recognises and allows for the fact that a different sort of game is played at the beginning, middle and end, requiring different techniques and different emphases.

Each goal in the selected list proposes alternative next moves each selecting only moves which further its own purposes. E.g., only the material balance goal will propose preventing a piece being captured, only king safety protecting king, etc.

Having selected a list of proposed alternative moves, separate analysis generators decide which continuations should be explored to give a correct analysis of each move. The exploration of continuations is based on Turing's definition of a dead position. A position has to be considered dead or static before it is evaluated, but NSS introduce an extention of Turing's idea. Before a position is considered dead it is examined from the point of view of each of the goals relevant, and judged to be dead by all of them before it is evaluated. If a goal judges the position in question to be static, it gives the position a provisional value. However, this value is only valid if all the other goals find the position static too. Otherwise the program generates the moves which will drastically effect the position, i.e., the moves which prevented the position from being static, and the process is repeated until a position is found which is considered static from all points of view. Thus the selection of continuations is dictated by the search for a dead position.

(v) A. L. Samuel

Samuel (SAM 59, SAM 60, SAM 67) chooses the game of checkers for his experiments. His main interest in writing this program is the development of new programming techniques, especially those involved in programming a machine to learn from experience.

Samuel's program looks ahead in a normal way with no selection up to a ply of three, at which point all positions are evaluated if they are dead. Otherwise the program looks ahead until the positions are dead. Since the positions which are not dead are usually the most profitable, this is a useful form of selection - it is the one used by NSS. The best move is then found by minimaxing.

Samuel's program learns by two quite different methods, although they can be combined. The simplest is what he calls the "rote learning method". The program stores all the board positions it encounters with the score they were given by the look ahead procedure. In most cases these scores will represent a look ahead of three plys. When the program, while, for example, looking ahead 3 plys, comes across a board position which is already stored with its backed up score, it can give the board position it is currently evaluating a score based on a look ahead of effectively 6 plys. If this process is repeated often enough, the score should eventually become quite accurate since the further one looks ahead the more accurately one can assess a position. Because of the enormous number of possible board positions the program also has a facility for forgetting board positions. It forgets those which it uses least.

Samuel's other learning process involves "generalization on the basis of experience". This works best when the program is playing both sides, i.e., selecting both Black's and White's moves.

The score of any given board position is worked out by using an evaluation polynomial, i.e., an equation whose terms represent the various features of checkers, such as the material balance, etc. Each term is given a numeric value which represents its current state in the board position, and this is multiplied by a coefficient which represents the importance of that term (i.e., feature) to the game as a whole. The program learns by experimenting with different terms in its evaluation polynomial and by varying the sign and magnitude of the coefficients for these terms. One side, Black, plays with a fixed evaluation function. The other side experiments continually. After every move White generalizes on the basis of experience and adjusts the terms and their coefficients accordingly. If White wins, Black is given its latest evaluation polynomial. If Black consistently wins a drastic change is made in White's evaluation polynomial.

To judge its evaluation polynomial, White compares the score it gives a certain board position with the score given to it by the rote learning method. The difference between the 2 scores is reflected in a variable, Delta, which is then used to make appropriate adjustments to the coefficients. A term in the evaluation polynomial will be replaced if it consistently has the lowest coefficient.

(vi) D. Michie and J. E. Doran

Doran and Michie's (DOR 66) work with the Graph Traverser program provides a more general method for the solution of a whole family of problems. It can be used to solve any problem which can be translated into terms of graph theory, as that of finding a path between two specified nodes of a specified graph. The approach used initially is that of "state evaluation". A problem state is given a value which reflects the extent to which it has features in common with the goal state, or which is related to its "distance" from the goal state.

The program is general in that it can be applied to different problems if it is provided with a rule book and an evaluation function. The rule book must enable the program to generate from an initial state all neighbouring states. The evaluation function will be used to discover to what extent a state approaches the goal state.

A quantity, described as "penetrance", is discovered to be a useful measure of the efficiency of an evaluation function. Penetrance is defined as $\frac{\text{length of the path produced}}{\text{total number of nodes developed}}$, or more informally as "the degree to which the search tree is 'elongated' rather than 'bushy'." Using this quantity the program can improve its own evaluation function in the midst of attempting to solve a problem, and thus be said to be learning.

(vii) E. W. Elcock and A. M. Murray

Elcock and Murray (ELC 68, MUR 67, MUR 68) are primarily interested in the problems involved in writing a learning program. The game they choose for their experiment is Go-Moku - a "simple but not trivial game". They choose a simple game so that they can study their particular learning technique in some depth.

The aim in Go-Moku is to make up a straight line of five of one's own counters while preventing the opponent from doing so first. The essence of the game is to recognise and try to make up certain patterns which are unbeatable. Elcock and Murray's program is like Samuel's in that it learns from experience, but the learning process is completely different from either of Samuel's methods. The Go-Moku program learns by using a "deductive backtrack analysis". When a game has been won the program looks back to the point where it thinks a win was inevitable. The backtrack analysis, like the rest of the program, works in terms of pattern recognition, so that when the critical board position has been found it is described in terms of the pattern it forms, and if

the descriptive language is good enough, that pattern will be recognised next time it occurs, and marked as a winning one. The descriptive language is such that when a board position is described it is automatically generalised, so as to capture the essential and only the essential elements of the pattern. This is much more economic than storing all the board positions.

Move selection is governed by a list of subgoals, each describing a pattern from which a win should be inevitable. These subgoals are listed in order of the number of moves away from a win. The list of subgoals is added to every time the program loses. When this happens the program unplays each move in turn starting with the last move and working backwards. As soon as it comes across a board position which is not described in the current list of subgoals, a generalised description of that board position is added to the list, on the assumption that this position is a necessary and sufficient step in the formation of the board position which immediately succeeded it in the game, which the program did recognise. For example, 4 counters in a row is a necessary and sufficient step to making up 5 in a row.

This assumption is only justifiable if the descriptive language has attained the ideal of capturing the essential, and only the essential elements of a pattern. Elcock and Murray had a certain amount of difficulty with their language because it tended to pick up some superfluous information which meant that the achievement of a listed subgoal did not necessarily lead to the subgoal above it, or bring about a win. It also meant that what was essentially the same pattern would be listed several different times in slightly different forms.

Elcock and Murray also found that their language was not powerful enough to describe certain situations. The opponent would win by a pattern which was listed, but whose description had missed some vital factor, had not been flexible enough to capture all the essential features of this pattern.

III THE GAME OF THREE-DIMENSIONAL NOUGHTS AND CROSSES

The game of 3 dimensional noughts and crosses was chosen for our own experiment. The object of the game is to place four counters in a straight line. The board has four layers, each of which is four squares by four: i.e., there are sixteen squares on each layer (or plane) and therefore sixty four squares in all:-

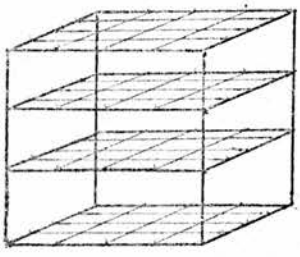


Figure 2

A line of four can go in any direction provided it is straight. That is, instead of making up lines on one plane as in ordinary noughts and crosses one can make up a line which has a square on each plane, e.g., going straight down from a square on the top plane, or diagonally from the top near corner to the bottom far corner. This can be more easily demonstrated if we represent the board thus:-

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

33	34	35	36
37	38	39	40
41	42	43	44
45	46	47	48

49	50	51	52
53	54	55	56
57	58	59	60
61	62	63	64

Figure 3

A line could be made up by playing in squares 1,2,3,4 or 1,17,33,49, or 1,18,35,52, or 1,22,43,64. The opponent of course does his best to prevent one from getting 4 in a row. This is done, as in ordinary noughts and crosses by placing one of his own counters in the line you are trying to make up. The skill in this game consists in recognising certain patterns. There is no point in merely forming lines of three which the opponent can stop with one counter. One must try to form a pattern where by placing a counter in a particular square, one can simultaneously make up two different lines of three,

e.g.,

X		X	
	X	X	
		1	
		3	2

Figure 4

Play in square 1 makes up a forcing pattern - the opponent must defend (unless he can win in one move himself), by stopping one of the lines of 3, but whether he plays in square 2 or square 3, he cannot stop both lines: whichever square he plays in the attacking side will win by playing in the other.

There are several of these forcing patterns which a player must try and form and at the same time prevent his opponent from forming. This is in fact the essence of the game.

It is worth noting that from the point of view of forming patterns, certain squares on the board have more potential value than others. There are altogether 76 straight lines on the board - 10 on each plane, 16 vertical lines, 16 semi-diagonal lines, contained in planes parallel to an edge of the board, and 4 true diagonals passing from one corner

of the board at the top level to the diagonally opposite corner of the board at the bottom level. Some squares appear on more of these lines than do other squares. The 8 corner squares on the top and bottom layers have 7 lines passing through them, and so do the 8 squares in the very centre of the board. All the other squares have only 4 lines passing through them. Squares with 7 lines passing through them are referred to as "strong" squares, and the rest as "weak" squares.

IV INFORMAL DESCRIPTION OF PROGRAM

The current version of our program, "Score 4", is similar in many respects to the Go-Moku program of Elcock and Murray. Like the Go-Moku program, "Score 4" plays to a win, then backtracks and analyses the board position at the stage where it thinks the critical move was made. The analysis is made in terms of the pattern formed by the winning combination of counters. This analysis is then stored and used to recognise that pattern if it occurs again. The analysis is generalised so that its description applies to both sides and to any board position which has essentially the same features. The program also has a selective look-ahead procedure which looks for any of the patterns analysed by the backtrack analysis. Move selection is, for the most part, governed by this look ahead procedure.

In the course of writing the program, several different techniques were tried and discarded and new techniques adopted as a direct result of some of the program's deficiencies as demonstrated by the sort of game it played. In order to understand the present form of the program, and the nature of the problem itself it is worth discussing briefly some of the techniques which were tried and rejected.

The first technique tried was a look ahead which examined every possible alternative and continuation to a ply of 3. The resulting board positions were evaluated simply in terms of how many counters had been played. However, as has been explained, the important thing in this game is not how many counters there are in any one line, but what patterns there are. Therefore this look ahead proved to be singularly unhelpful, mainly because of its evaluation function. It was also rather slow.

Various different forms of look aheads were tried, but it soon became evident that some form of pattern recognition would have to be used to evaluate board positions.

(i) Descriptions

As Elcock and Murray (MUR 67, MUR 68) realised, the method of describing a pattern in a board position is the most essential thing in this sort of program. If a pattern is a winning one, the description must capture those features which make it a winning one. A good description must not miss out any essential feature, nor must it include any inessential feature. At the same time, the description must be general enough to apply to any board position which has essentially the same features, even though it is not exactly the same board position. These specifications are very difficult to meet, and in fact our program is not entirely successful in doing so, nor was the Go-Moku program discussed in "Machine Intelligence 1" (1967). (MUR 67).

To illustrate these points:-

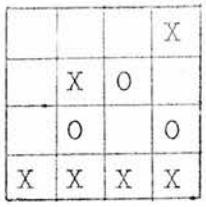


Figure 5

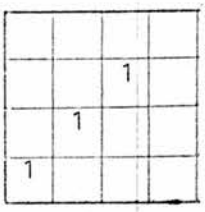
The essential thing about this board position is that there are four counters in a row - this is all the description of this board position should be concerned with. The other counters on the board are irrelevant to the pattern, and should be ignored in the description. The actual position of the counters on the board is also irrelevant, i.e., that they are (for example) in the front four squares of the top plane. The only relevant fact in the positioning is that they are in a straight line. All that the description need include is "a line (by which we always mean a straight line) with 4 (red) counters in it."

In fact, from a programming point of view, we found the simplest method of describing board positions was in terms of what the relevant lines add up to. The computer's move is always indicated by a 5, the opponent's by 1 and an empty square by 0. This means that by summing a line we always know exactly what counters have been played in it. A sum of 10 for example means 2 computer counters and 2 blank squares, 11 means 2 computer counters and an opponent one etc.

Therefore the actual description of the above board position would be:- "a line which adds up to 20".

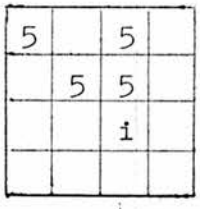
"A line which adds up to 3", would describe the board position below, making it quite clear that it is the opponents line, that he can win in one move because there is a blank square in the row.

Figure 6



This board position can be converted into a forcing pattern by playing in the square marked (i). This square is referred to as the "key" square in the pattern, as is a blank square in any pattern if by playing in it, one can create a forcing pattern.

Figure 7



The essential features here are that there are "two lines, both adding up to 10, which have a blank square (i) in common". This description captures the essential features and is general enough to apply to the board positions:

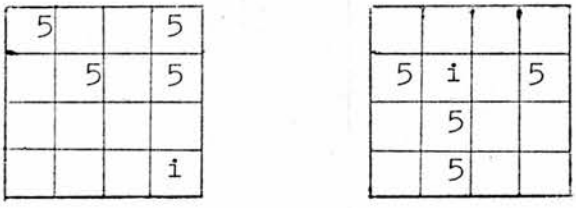


Figure 8

which are essentially the same. (This pattern is referred to in Appendix 1 as Pattern 1.)

Referred to as Pattern 2.

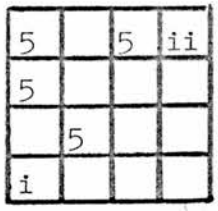


Figure 9

The description of this pattern involves 3 lines and there are 2 "key" squares. In this instance it does not matter in what order they are played. Play in either one forces the opponent to defend by stopping a row of 3. Play in the

second "key" square then makes up 2 separate rows of 3, only one of which the opponent can stop. The essential features are covered by the description:-

"a line adding up to 10 with a blank square in common with a line which adds up to 5, such that the line of 5 has a blank square in common with a different line adding up to 10."

This description captures the essential elements and is general enough to describe also the board positions:-

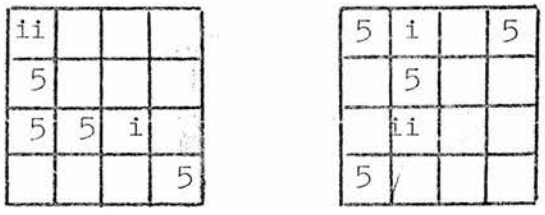


Figure 10

The board position:-

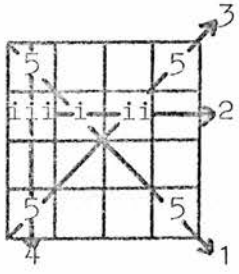


Figure 11

involves 4 lines. Here there are 3 "key" squares and the order they are played in is more important than in the last example. However the order of (i) and (ii) could be interchanged. Play in square (i) makes up 3 in a row and forces the opponent to defend. Play in square (ii) makes up another 3 and the pattern is now the same as that in Figure 7, and therefore play in square (iii) makes up 2 separate lines of 3. The description of this pattern is:-

"a row adding up to 10 with a blank square (i) in common with a line adding up to 0 which has a blank square in common with another line adding up to 10 and also has a blank square in common with a 3rd line adding up to 10."

i.e., 3 rows of 2 are involved in such a way that by making up 2 of them into rows of 3, one at the same time makes up another row of 2 which converges with the 3rd row of 2.

The description also fits this board position:-

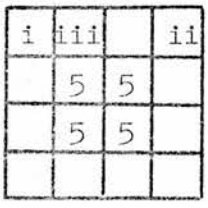


Figure 12

Thus the program describes a pattern in terms of the lines which are involved in playing out the pattern and what they add up to, and the intersections of the lines which take place on the "key" squares. The lines are carefully listed in the right order, so that the square which must be played in 1st is in the line which is mentioned first, and so on.

The more lines mentioned in the description of a pattern, the more moves have to be made before a win is made.

This method of description has the advantage over Elcock and Murray's 1st method, that only the lines which are actually involved in a win are described. By "involved", we mean lines which are built up to have 3 in a row and thus force the opponent to defend. A win almost always involves 2 lines being simultaneously built up to have 3 in a row, and both these are "involved".

Elcock and Murray describe a pattern in terms of the line which has the most counters in it and the line crossing it which has the most counters in it. This is arbitrary in that the 2nd line mentioned is sometimes irrelevant. The result was that more than the essential features were sometimes described, which meant that the description would not fit another board position which had the same essential features, but not the irrelevant ones which were also picked up in the description. The description is not what they call a minimal one.

(ii) Backtrack analysis

Like the Go-Moku program, the "Score Four" program generates its own list of patterns by using a backtrack analysis to find and analyse the board position which made a win inevitable. A game is played to a win, then the program backtracks until it finds the critical board position which is then analysed in terms of its pattern and listed.

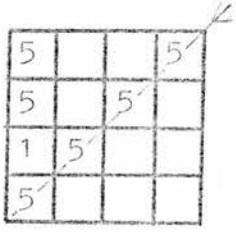
The backtrack analysis (b.t.a.) is called whenever a game is won or lost, as it has the facility to analyse either a computer or an opponent win. The b.t.a. unplays in turn each of the winner's moves and the corresponding loser's moves, taking the moves in reverse order, i.e., unplaying the last move 1st and working backwards.

The backtrack analysis first takes a note of the winning line and then unplays the winner's last move. Let us assume that the computer playing Red, won. The opponent (White)'s last move is also unplayed, so that we recreate the board position two plys before the end of the game. The b.t.a. then tests to see if Red could still have won in one move had White played his last move where Red did. If so, a note is taken of the relevant line (i.e., the line in which Red

could have won had White not played his last move in it) and the next pair of moves is unplayed and the process repeated until Red can no longer win in one move.

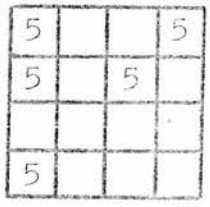
e.g.,

Figure 13



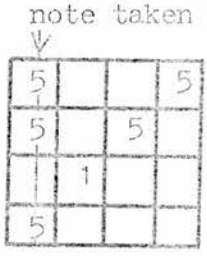
If this is the winning board position, the last pair of moves will be unplayed to give the board position:-

Figure 14



The opponent's move is then played where the winner's last move was:-

Figure 15



It is obvious that Red can still win in one move, by playing where White's original last move was.

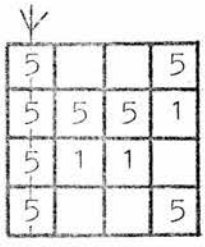
The object of this exercise is to discover at what stage the pattern started to be a forcing one. By the time the pattern in Figure 14 has been formed, Red is virtually unbeatable. The program must be able to recognise patterns before they become forcing, so one more pair of moves is unplayed to give the position:-

Figure 16

This board position is then analysed by simply adding up the counters in the lines noted during the b.t.a. and listing them as one pattern. In the above example two lines were noted; these lines are summed as they are in the final position reached by the b.t.a., i.e., at the stage where the position is one move away from being forcing. In this case, as they are in Figure 16.

The pattern would be listed simply as 10, 10. A winning pattern always involves lines intersecting at some stage and it is important to know and record, when analysing a board position, which lines intersect with which. In this program the order in which the lines of a pattern are listed, reflects which lines intersect with which. In the above example the order does not matter, because only 2 lines are involved and they must intersect with each other. However in a more complicated example:-

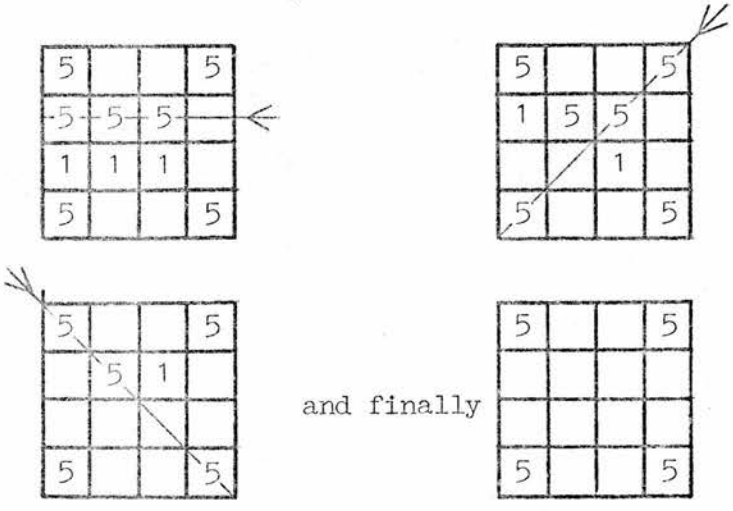
Figure 17



the order in which the lines are listed is very important.

The sequence of the backtracking is as follows:-

Figure 18



The lines with arrows are the ones noted during the course of the backtrack. Four lines are noted including the winning one, which add up in the final position, to 10,0,10,10. It is important to know that the 1st line intersects with the 2nd, the 2nd with the 3rd and the 4th also with the 2nd.

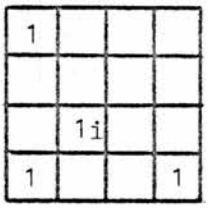
The b.t.a. is also used to compile a list of the patterns as they were one move previously, i.e., two moves before they are forcing. This list is used for move selection in certain cases (see page 27).

(iii) Limitations of backtrack analysis and description

This method of backtracking unfortunately has certain limitations. For one thing it is limited in the amount of learning it can do, and this limitation leads to a limitation in recognising the potential of certain board situations. This can be quite simply illustrated. Given the board position:-

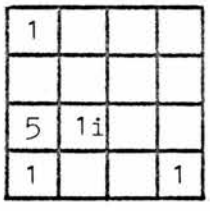
Figure 19

a



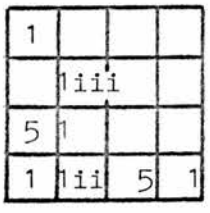
the program will recognise this pattern (see Figure 9) and will play:-

b



which stops one pattern. However the strength of this position is that playing in square (i) makes up 2 separate patterns

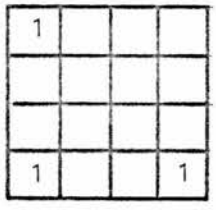
c



and the opponent can go on to win by playing in square (ii) which forces the computer to defend, then square (iii) which makes up 2 rows of 3.

In fact the board position:-

Figure 20



should be listed as a forcing pattern, but because of the way the b.t.a. works, it never will be. If the b.t.a. worked like the Go-Moku one, i.e., by working backwards from the winning board position until it finds a board position which is not listed as a pattern; and then adding the new board position to the list, then this position would be added. The difference between these two methods of backtracking lies in the criteria used to find the board position to be analysed - the board position which made the win inevitable. Our criterion is whether the side attacking can win in one move if his opponent does not defend. As soon as a position is reached by backtracking where the attacker can no longer win in one move, this position is analysed. The position in Figure 19a satisfies this criterion. The Go-Moku program builds up its list of patterns by backtracking until it finds a position it has not yet listed, analysing this position and adding it to the list - the b.t.a. is only used when the program loses. This method may take several games to completely analyse a fairly complicated pattern, but in theory it should get there in the end. In fact Elcock and Murray complain in their 1967 article (MUR 67) that their program was limited in the amount it could learn and that it could not realise the potential danger of some positions - which is precisely what we are complaining of. However, their method of backtracking seems to be a much stronger one and would have been used but for certain programming difficulties - see **Chapter VII.**

(iv) Look ahead

The limitation described above makes the program vulnerable to certain opponent attacks, and also means that it does not realise the strength of certain patterns it forms, or could form, itself. For this reason there are two separate look ahead procedures, one which checks on the strength of the opponent's position, and one for the computer. The results of both look aheads are then compared and the decision made whether to defend or attack, due allowance being made for the fact that it is the computer's turn to play.

The program starts the look ahead by playing one of its own moves in the 1st blank square on the board. It then looks to see if this makes up any of the listed patterns. If no pattern is formed by playing in that square, the move is unplayed and the next blank square is tried and so on until each blank square in turn has been tried. When a pattern is formed, the square played in to form it is given a score which will depend on how good that pattern is considered to be. An opponent's move is then played in the "key" square, and the program looks to see if a pattern still exists and also to see if forcing the opponent to play there makes up a pattern for him; i.e., forces him to help himself. If so, i.e., if an opponent patterndoes exist, then the original square played in during the look ahead is given no score. However, if a pattern still exists for the computer, after playing an opponent move in the "key" square, then the rest of the look ahead is skipped and the program automatically plays in the original square being tested by the look ahead. This is because if this situation does exist it means that playing in one square has made up simultaneously 2 patterns, only one of which can be stopped by the opponent. This situation is referred to as a double pattern. This forms a highly

selective look ahead procedure. Unless a double pattern is found, all legal next moves are tried in turn, the look ahead continuing with a square only if playing in that square forms one of the listed patterns. If not, the square being tried is given a score but the look ahead goes no further with it and the next square is tried. The program plays an opponent move in what it reckons to be the worst possible place for the computer, i.e., in the "key" square of the pattern found by the look ahead, taking each pattern in turn. Having done this the look ahead continues by looking to see what patterns, if any, are left. If it finds no pattern, it means that the computer cannot create a forcing pattern in one move. If there is a computer pattern, the program knows that by playing in a certain square (and it knows exactly which one) it can create a forcing pattern and almost certainly win. Thus the program can be said to look ahead to a depth of 3 plys - it has worked out what will happen if Red plays there, White there and then Red there. In a sense, if a pattern does exist after the opponent's move has been played (in the look ahead), the program can be said to look ahead to the end of the game because once it looks ahead to a forcing pattern the course of the game is predictable - the opponent will be forced to keep defending until the computer wins. This at least is what ideally should happen. However, as in most selective look aheads, the computer's prediction of where the opponent will play may easily be wrong. Sometimes playing the opponent's move in the "key" square is not the best place for him. The "key" square is where the attacking side would need to play to make his pattern forcing, but sometimes the opponent can stop 2 patterns by playing in one place, but only one by playing in the "key" square. If the look ahead procedure tried playing the opponent's move in every possible square, it would inevitably discover the most effective place for the opponent to play, but this would be too time consuming.

The look ahead for the opponent is exactly the same, except that if the program realises that it will not be able to stop the opponent winning, because he has a double pattern, the "key" square in the forming of the double pattern is given a large enough score to ensure that it will be the one selected for the machine to play in unless it finds a double pattern for itself.

If no pattern can be formed in one move, the program looks to see if by playing in any of the blank squares, it can make up a pattern which can be converted in one move into one of the listed patterns; i.e., it looks to see if there is any pattern of counters in the board position which in 3 moves can become a forcing pattern. This is very often the case as soon as there are 2 counters on the board.

However, sometimes even this situation does not exist - e.g., when there is only one counter on the board. In this case another method of selection must be used. The method used is simply that of choosing a "strong" square in a line which (preferably) already has a counter in it. One point is added to the score of each square in a line with 1 (computer) counter in it, or 2 points if there are 2 counters. These points are added to the initial (see below) score of the relevant squares. The highest scoring square is then chosen - strong squares are given more points than weak ones.

(v) Scoring

A detailed account of the scoring system.

At the beginning of a game, each square is given an initial score which supposedly reflects its positional strength. There are 2 categories of squares, as we have seen 16 "strong" squares and the rest "weak" squares.

The initial score of a strong square is 7, corresponding exactly to the number of lines which pass through it. A weak square accordingly has an initial score of 4.

In an earlier version of the program, each square was then subjected to various tests (such as what pattern, if any, would be formed if the computer played in it, and if the opponent played in it etc.). The score of the square was added to, or subtracted from, as it passed through each test, so that its final score reflected its worth from all points of view. However, it was not carefully enough balanced, and what tended to happen was that a mediocre attacking square which was also a mediocre defensive square would score more highly than a good attacking square. The result was that the program played a dull game with no good attacking moves. A good attacking move does not need to worry about defending.

It was found to be simpler to decide first whether to defend or attack, and once that decision had been made, to find the best attacking move, or the best defending move if the decision was to defend.

To do this, each square is given a score which reflects its value to the opponent, and its value as an attacking square is separately assessed. If the highest scoring attacking square scores less than the highest scoring square for the opponent, then the computer defends.

The first scoring test is the look ahead for the opponent. The program looks to see whether any of the listed patterns will be formed if the opponent plays in a certain square. If so, the square has points added to its score - the exact number of points will depend on which pattern is formed. Four points are added if the pattern is "three in

a row", and usually 9 points for any other pattern. There is a mechanism for raising the number of points given for any pattern to which the computer consistently loses. If the computer consistently loses to a particular pattern, it is because it has not fully realised the strength of this pattern. This is a fault in the description of the pattern. One way of compensating for this fault is to simply allot more points to this pattern, so that it is more likely to be stopped if spotted during the look ahead.

Having discovered an opponent pattern, the look ahead proceeds to play a computer move in it's key square, and looks to see what pattern, if any, is left. If no pattern is left, then no more points are added to the score of the square being assessed, and none are subtracted. If the original opponent pattern is stopped, and instead a computer pattern has been formed, then the square has all its points taken away. If despite the computer's defensive move, there still exists an opponent pattern, then 30 points are added to the score of the original square being tested by the look ahead. The assumption is that if the opponent plays in that particular square, then the computer will not be able to stop him making up a forcing pattern. The points given are sufficient to ensure that unless the computer can make up a similar pattern, it will play in that square.

A note is taken of the highest scoring opponent square and the score of each square is then set back to its original score, i.e., either 4 or 7, and the look ahead for the computer is done. This works in exactly the same way as the opponent look ahead, except that 5 points are allotted to a "three in a row" pattern and 10 to any other pattern. Also as has been mentioned, if a pattern is found which the opponent cannot stop in one move, then the computer immediately plays in the appropriate square and skips the rest of the look ahead procedure and assessment.

If after these two look aheads, the highest scoring square has less than 15 points, which would mean in fact that no pattern other than "three in a row" had been found for the computer during the look ahead, then the next test is applied. This is another look ahead to see if any of the listed patterns (other than three in a row) can be formed in two moves, i.e., whether by playing in a certain square the computer can create a board position which can be converted into one of the listed board patterns in one move. If so, the square has 8 points added to its initial score.

However, if no square satisfies this test there will still be no way of discriminating between squares, apart from their initial positional strength.

The final test discriminates between squares on the basis of how many counters have been played. One point is added to the initial score of every square in a line in which there is a computer counter (and no opponent one). Two points are added if there are 2 counters. If this leaves several squares with the same score, the 1st one found is chosen. This is referred to as the "counter scoring method" (CSS).

V THE PROGRAM (formal description)

(i) Nomenclature

MOVE(64) The board is represented by the array MOVE(64), which contains an integer variable for each of the 64 squares on the board.

SUM(76) is an array used to hold the sums of each of the 76 lines.

ROW(304) is an array used to keep an account of which squares are contained in each line, i.e., the first 4 values of ROW are the squares which make up line 1. The squares are numbered in the order shown in

Figure 3, and the 1st line is made up of the squares 1,2,3,4 and these four numbers are therefore the first four values of ROW. In the same way line 76, which is one of the main diagonals, is made up of squares 4,23,42,61 and these numbers are the last four values of ROW.

ROWDY(304) is a similar array which keeps account of the various lines passing through each square. The first 7 values of ROWDY are therefore the 7 lines which pass through square 1, the next four values are the four lines which pass through square 2 etc.

ROWCNT(65) is an array used for referencing ROWDY. It holds an account of where in ROWDY to find which lines pass through any square. Thus ROWCNT(1) is 1, ROWCNT(2) is 8 - the beginning in ROWDY of the lines for square 2. The number of the subscript corresponds to the number of the square.

TEST(5,32) is a 2 dimensional array which describes the patterns which have been found and analysed by the program.

SUTEST(3,20) performs a similar function to TEST but describes the same patterns as they were one move before the stage they are described in TEST.

SUB(64) is an array used for scoring. Its values correspond to the initial value of the squares, which depends on how many lines go through it.

SUB1(64) and SU1(64) are also used for scoring.

LAST(32) is used to list the moves made by the computer.

NEXT(32) is used to list the moves made by the opponent.

START(16) is used for the initial move; it holds the 16 initially "strong" (v.P.14) squares from which the computer's first move is selected at random.

M11(2,16) is a 2 dimensional array used during the look aheads to hold the different patterns found in the current board position with the appropriate square to play in.

ROWNO(5) is used during the backtrack analysis to hold the numbers of the lines which go into the making up of a pattern.

MOST(3) is used during the backtrack analysis for putting the lines into the right order in a pattern.

X(3) is used for the same thing as MOST.

AJUST(32) is used in a self adjusting scoring system which adjusts the value given to any pattern.

WIN(4) is an array used to hold the 4 squares which are in the winning line. These are printed out at the end of a game.

PCOUNT is a variable used to keep track of how many patterns are listed in TEST. It must be updated each time a new pattern is found.

SUCNT keeps count of the number of patterns in SUTEST, as PCOUNT does for TEST.

GCNT is used as a switch which is off when a pattern is to be listed in TEST and on when it is to be listed in SUTEST. It is switched on the 2nd time the program goes through the b.t.a.

BIG is a variable used to hold the highest score of the squares examined during the opponent look ahead.

BIGOPP holds the number of this square.

NOCNT keeps count of how many times subroutine TESTT is called during the backtrack analysis, and is used as a test to ensure that a new pattern is listed both as a computer and an opponent pattern.

Before each new game the following are set to zero:-

MOVE i.e. the board

SUM - the sums of the lines

LAST and NEXT - the lists of moves

COUNT and SCOUNT which are used to keep count of the number of moves made by each side.

(ii) Input data

ROWDY, ROW, ROWCNT, TEST, SUTEST, SUB, START, ADJUST are read in as data. A complete list of the input data follows on the next page.

Had there been time to tidy up this program, the data statement would have been used, as a more efficient method of inputing this data.

TEST and SUTEST are shown here with several patterns listed. Originally they would be input with SUTEST completely set to zero, and only 20 and 4 (i.e., four in a row for both sides) in TEST.

```
ROWDY   1 17 33 49 57 69 73  1 18 34 70  1 19 35 71  1 20 36 50 68 72 74  2 17 37 58
        2 18 38 49  2 19 39 50  2 20 40 67  3 17 41 59  3 18 42 50  3 19 43 49  3 20
       44 66  4 17 45 50 60 61 76  4 18 46 62  4 19 47 63  4 20 48 49 64 65 75  5 21
       33 51  5 22 34 57  5 23 35 68  5 24 36 52  6 21 37 69  6 22 38 51 58 70 73  6
       23 39 52 67 71 74  6 24 40 72  7 21 41 61  7 22 42 52 59 62 76  7 23 43 51 63
       66 75  7 24 44 64  8 21 45 52  8 22 46 60  8 23 47 65  8 24 48 51  9 25 33 53
        9 26 34 68  9 27 35 57  9 28 36 54 10 25 37 61 10 26 38 53 62 67 75 10 27 39
       54 58 63 76 10 28 40 64 11 25 41 69 11 26 42 54 66 70 74 11 27 43 53 59 71 73
       11 28 44 72 12 25 45 54 12 26 46 65 12 27 47 60 12 28 48 53 13 29 33 55 61 68
       75 13 30 34 62 13 31 35 63 13 32 36 56 57 64 76 14 29 37 67 14 30 38 55 14 31
       39 56 14 32 40 58 15 29 41 66 15 30 42 56 15 31 43 55 15 32 44 59 16 29 45 56
       65 69 74 16 30 46 70 16 31 47 71 16 32 48 55 60 72 73
ROWCNT  1  8 12 16 23 27 31 35 39 43 47 51 55 62 66 70 77 81 85 89 93 97104111115119
      126133137141145149153157161165169173180187191195202209213217221225229236240244
      251255259263267271275279283290294298305
ADJUST  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
        0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
ROW     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
       25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
       49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64  1  5  9 13  2  6 10 14
        3  7 11 15  4  8 12 16 17 21 25 29 18 22 26 30 19 23 27 31 20 24 28 32
       33 37 41 45 34 38 42 46 35 39 43 47 36 40 44 48 49 53 57 61 50 54 58 62
       51 55 59 63 52 56 60 64  1 17 33 49  2 18 34 50  3 19 35 51  4 20 36 52
        5 21 37 53  6 22 38 54  7 23 39 55  8 24 40 56  9 25 41 57 10 26 42 58
       11 27 43 59 12 28 44 60 13 29 45 61 14 30 46 62 15 31 47 63 16 32 48 64
        1  6 11 16  4  7 10 13 17 22 27 32 20 23 26 29 33 38 43 48 36 39 42 45
       49 54 59 64 52 55 58 61  1 18 35 52  5 22 39 56  9 26 43 60 13 30 47 64
       13 25 37 49 14 26 38 50 15 27 39 51 16 28 40 52 16 31 46 61 12 27 42 57
        8 23 38 53  4 19 34 49  1 21 41 61  2 22 42 62  3 23 43 63  4 24 44 64
        1 22 43 64  4 23 42 61 16 27 38 49 13 26 39 52
TEST   20  4 15  3 10  2 10  2 10  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 10  2  5  1  0  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 10  2 10  2 -1 -1 -1 -1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
       10  2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
SUB     7 4 4 7 4 4 4 4 4 4 4 4 7 4 4 7
        4 4 4 4 4 7 7 4 4 7 7 4 4 4 4 4
        4 4 4 4 4 7 7 4 4 7 7 4 4 4 4 4
        7 4 4 7 4 4 4 4 4 4 4 4 7 4 4 7
SUTEST 10  5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  5  5 -1 -1 -1 -1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 10 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
       -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
START   1  4 13 16 22 23 26 27 38 39 42 43 49 52 61 64
```

(iii) Program listing

```
C       JILL DOAKE
C       4SCORE
        IMPLICIT INTEGER(A-Z)
        LOGICAL M4
        DIMENSION MOVE(64),SUM(76),ROW(304),ROWDY(304),ROWCNT(65),TEST(5,32),
       1SUTEST(3,20),SUB(64),SUB1(64),SU1(64),LAST(32),NEXT(32),START(16),
       2M11(2,16),ROWNO(5),MOST(3),X(3),AJUST(32),WIN(4)
        COMMON ROW,TEST,M1,M2,M3,M4,M11,SN,CNT,ROWDY,ROWCNT,XYZ
C       READ ALL DATA
        READ(5,101)ROW
        READ(5,114)ROWCNT
        READ(5,114)ROWDY
        READ(5,114)((TEST(I,J),J=1,32),I=1,5)
        READ(5,114)((SUTEST(I,J),J=1,20),I=1,3)
        READ(5,104)SUB
        READ(5,114)START
        READ(5,114)AJUST
C       SET COUNTS
C       PCOUNT IS SET TO THE NUMBER OF PATTERNS CURRENTLY LISTED IN TEST
        PCOUNT=0
C       SUCNT IS SET TO THE NUMBER OF PATTERNS IN SUTEST
        SUCNT=0
        CNT=0
        COUNT=0
        SCOUNT=0
        Y=0
```

```
C       CLEAR THE BOARD
    1 DO 2 I=1,64
    2 MOVE(I)=0
      DO 933 I=1,76
  933 SUM(I)=0
C       SET LIST OF COMPUTER AND OPPONENT MOVES TO ZERO
      M2=0
      DO 16 I=1,32
      LAST(I)=0
   16 NEXT(I)=0
C       READ OPPONENT'S MOVE, CHECK FOR NO MOVE(0), OR END OF GAME(-)
    4 READ(5,100)NEXT1
      WRITE(6,100)NEXT1
```

Read in the opponent's move and write it out (simply for the opponent to check that he has typed in the move correctly).
The move is now tested to see if it is any of a series of conventional signals - a minus number indicates the end of the game and start of a new one, 0 is the conventional sign that the opponent wants to skip his move and let the computer have the 1st move, 99 is the signal to end the whole session.

```
      IF(NEXT1)14,12,6
    6 IF(NEXT1-99)8,7,10
    7 CALL EXIT
C       IS IT A BLANK SQUARE
    8 IF(MOVE(NEXT1))11,11,10
C       IF NOT WRITE ERROR MESSAGE AND TRY AGAIN
   10 WRITE(6,105)
```

```
      GO TO 4
C     OTHERWISE MOVE ARRAY IS UPDATED TO TAKE ACCOUNT OF NEW MOVE, AND
C     SO IS COUNT
   11 MOVE(NEXT1)=1
      Y=NEXT1
      WRITE(6,109)MOVE
      COUNT=COUNT+1
      NEXT(COUNT)=NEXT1
      WRITE(6,102)NEXT
      IF(COUNT.EQ.32.AND.SCOUNT.EQ.32)GO TO 14
```

This last test ensures that if all the squares on the board are filled, the game is automatically terminated.

M4 is a logical variable used by Search to distinguish between a real board position and a look ahead procedure - it is set to TRUE for a real board position.

```
   12 M4=.TRUE.
      CALL SEARCH(MOVE,SUM,Y)
      IF(M3.EQ.100)GO TO 15
      IF(M3.EQ.98)GO TO 13
      IF(M1.GT.0)GO TO 13
```

This is the start of the calculation of the computer's move. Search is a subroutine which has a note of the sums of the 76 lines, which it updates with each new move made. The new move is noted in the variable Y. Using these sums, Search looks to see if there exists anywhere on the board a situation which forms one of the patterns

listed in TEST. M3 will be 100 if someone has 4 in a row, in which case a winning message is printed and the backtrack analysis starts. M3 will be 98 if someone has 3 in a row in which case the computer must immediately play in the blank square left, (either to make up 4 or to stop the opponent's 4) and the number of the square will be the value of M1. If any other pattern exists, M1 will be set to the number of the "key" square (v.P.17) in this pattern, and the computer will automatically play there.

```
      IF(SCOUNT.GT.0)GO TO 40
      L=KLOCK(J,K)
  130 L=KLRAND(L)
      M=L*.465661287E-9*16+1
      M1=START(M)
      IF(MOVE(M1).GT.0)GO TO 130
      GO TO 13
```

This part of the program is only used for the very 1st move - it is skipped as soon as SCOUNT, which keeps count of the number of computer moves played, is greater than 0. The values of START are the numbers of the 16 "strong" (v.P.14) squares on the board. The computer's 1st move is taken from amongst these at random. "L=KLOCK(J,K)" is the calling sequence of a program which sets L to the time of day in 50ths of a second. "L=KLRAND" is the calling sequence for a program which, given a number (here the time of day, i.e.L), generates a new pseudo random number in the range 1 - 2^{31-1}. Multiplying by .465661287E-9 produces numbers between 0 and 1, multiplying again by 16 and adding one produces numbers between 1 and 16.

The next section is a look ahead for the opponent, i.e., the program looks to see if the opponent has any dangerous situations coming up. The board situation is assessed in terms of the potential danger from the opponent, and this is reflected in a score for each square, held in SUB1. The score of the most dangerous (i.e., highest scoring) square is compared with the computer's after its own look ahead.

M4 is set to false to indicate that this is a look ahead. This is important because it means that instead of returning from Search as soon as any pattern is discovered, all the patterns found when testing a particular square during the look ahead, are stored in the array M11 together with their respective "key" squares. Sometimes two or three different patterns will be formed by playing in one square.

```
40 M4=.FALSE.
   BIG=0
   DO 91 I1=1,64
91 SUB1(I1)=SUB(I1)
```

This stops the score from accumulating. A single move can make such a difference to the score that it is worth while recomputing the whole lot rather than just updating the squares affected.

The score always starts from SUB, not zero. The SUB scores correspond to the initial values (v.P.27) of the squares which are based on the positional value of the squares. This means that patterns formed by playing in one of the 16 "strong" squares are automatically given a better score than the others.

```
      DO 90 I=1,64
      IF(MOVE(I).GT.0)GO TO 90
      MOVE(I)=1
      CALL SEARCH(MOVE,SUM,I)
```

The opponent look ahead works by playing an opponent move in every blank square in turn and then examining the board position.

```
      DO 954 I1=1,16
      IF(M11(2,I1).LE.0)GO TO 920
      IF(M11(2,I1).NE.98)GO TO 911
      SUB1(I)=SUB1(I)+4
      IF(M11(2,I1+1))954,954,905
  911 IF(I1.EQ.1)GO TO 310
      IF(M11(1,I1).EQ.M11(1,I1-1))GO TO 905
  310 SUB1(I)=SUB1(I)+9+AJUST(M11(2,I1))
```

For each blank square played in during the opponent look ahead, Search is called and the patterns found in this board position (if any) listed in M11. The contents of M11 are examined. M11 lists both the number of the pattern and the number of the "key" square - the 1st column listing the square number and the 2nd the pattern number. If there are no patterns at all M11(2,1). i.e., the place for the 1st pattern number, will be 0, in which case control is transferred to statement 920 where the move is unplayed and SUM is readjusted to take account of this. The next square will then be tested in the same way, i.e., an opponent move played in it Search called etc.

If there is a pattern listed in M11, tests are made to find which pattern it is. A distinction is made between a pattern of 3-in-a-row (in which case M11(2,1) will be 98) and the rest of the patterns. Because 3-in-a-row is not a very valuable pattern, it has a low score attached to it. The score of the square being tested is adjusted according to which pattern is found, 4 for 3-in-a-row, 9 for any other. The score in AJUST (v.P.68) for this particular pattern is also added to the score of the square. Normally this will be zero, but if the pattern is one to which the computer has lost more than once, the score of the pattern is raised.

If the pattern is 3-in-a-row, a test is made to see if any other pattern exists. This is done simply by looking to see if the next value of M11 is greater than zero; pattern 98 will always be listed first. Statement 911 and the one immediately after it are to stop the score of a square being added to in certain circumstances. It sometimes happens that although playing in one square forms 2 different patterns, they can both be stopped by playing in one square. If this is the case, the "key" square for both patterns will usually be the same one. In this case, statement 310, which adds to the score of the square will be skipped, and the square will score points for only one pattern. Normally if more than one pattern is formed, the square scores points for each pattern, i.e., if 3 patterns are found, 3 lots of points will be added to the square's original score.

```
905 Z=M11(1,I1)
    MOVE(Z)=5
    M4=.TRUE.
    CALL SEARCH(MOVE,SUM,Z)
    M4=.FALSE.
    J=ROWCNT(Z)
```

```
      JJ=ROWCNT(Z+1)-1
      DO 906 YZ=J,JJ
      R=ROWDY(YZ)
  906 SUM(R)=SUM(R)-MOVE(Z)
      MOVE(Z)=0
      IF(M1)954,954,925
  925 IF(M2.NE.98)GO TO 934
  908 SUB1(I)=0
      GO TO 920
  934 IF(M3.NE.98)GO TO 907
      SUB1(I)=SUB1(I)+30
      GO TO 920
  907 IF(TEST(1,M3).GT.4.OR.TEST(2,M3).GT.4)GO TO 908
  910 M1=I
      SUB(I)=SUB1(I)+30
      GO TO 920
  954 CONTINUE
```

This section looks ahead one stage further. Once a pattern has been found during the look ahead, the program looks to see what happens if the computer tries to stop it - whether another pattern still exists, or whether forcing the computer to play there actually helps it by making it form a pattern.

Z is set to the no. of the "key" square for the pattern, and MOVE(Z) set to 5 - i.e., to a computer move. Search is then called, this time with M4 set to TRUE because it is enough to know if any one pattern exists without knowing how many.

SUM is adjusted in Search, and readjusted on return from Search when the computer move is unplayed. If there is no pattern M1 will be

zero and the next pattern in M11 undergoes the same process.

If there is a pattern we must know whether it is a computer or an opponent one.

M2 will be 98 only if the computer has 3-in-a-row, which would mean that forcing the computer to stop that particular pattern has made up a 3 for it. The score of the square being played in to form the 1st pattern is therefore given no points at all - playing there forces the computer to defend by playing in a square which in fact is advantageous to it. The trial move (opponent) is unplayed, SUM readjusted and the next square tried.

If M3 is 98 (but M2 is not) then the opponent has 3-in-a-row, i.e., playing in the trial square forms at least 2 patterns, one of which is 3-in-a-row and the computer cannot stop them both. This is obviously a very strong position and 30 points are added to the score of that square.

If the pattern is other than 98, it is a simple matter to discover whether or not it is a computer one, since the number of the pattern will be in M3 and this corresponds to the position of the pattern in the TEST array. If the 1st or 2nd part of the pattern in TEST is greater than 4, then it must be a computer pattern. In this case the score of the trial square is set to zero, otherwise it must be an opponent pattern and the situation is the same as if M3 is 98, i.e., 30 points are added to the score of the square. This is the end of the loop which examines M11.

```
920 J=ROWCNT(I)
    JJ=ROWCNT(I+1)-1
    DO 912 YZ=J,JJ
    R=ROWDY(YZ)
```

```
912 SUM(R)=SUM(R)-MOVE(I)
    MOVE(I)=0
```

This unplays the trial move and sets SUM back to what it was before the move was made.

```
    IF(SUB1(I).LE.BIG)GO TO 90
    BIG=SUB1(1)
    BIGOPP=I
 90 CONTINUE
```

This section is used to find which square has the biggest score. This is stored in BIG and the number of the square in BIGOPP. The score of each square is examined after SEARCH has been called and M11 examined. This score is compared with the current value of BIG and if it is bigger, then BIG takes this value and BIGOPP the number of the square. This ends the opponent look ahead loop.

The computer now looks ahead for itself.

```
    LARGE=0
    DO 50 I=1,64
 50 SUB1(I)=SUB(I)
    DO 52 I=1,64
    IF(MOVE(I).GT.0)GO TO 52
    MOVE(I)=5
    CALL SEARCH(MOVE,SUM,I)
    DO 54 I1=1,16
    IF(M11(2,I1).LE.0)GO TO 720
    IF(M11(2,I1).NE.98)GO TO 811
    SUB1(I)=SUB1(I)+5
```

```
      IF(M11(2,I1+1))54,54,805
  811 IF(I1.EQ.1)GO TO 801
      IF(M11(1,I1)-M11(1,I1-1))801,805,801
  801 SUB1(I)=SUB1(I)+10
  805 Z=M11(1,I1)
      MOVE(Z)=1
      M4=.TRUE.
      CALL SEARCH(MOVE,SUM,Z)
      M4=.FALSE.
      J=ROWCNT(Z)
      JJ=ROWCNT(Z+1)-1
      DO 806 YZ=J,JJ
      R=ROWDY(YZ)
  806 SUM(R)=SUM(R)-MOVE(Z)
      MOVE(Z)=0
      IF(M1)54,54,825
  825 IF(M2.EQ.98)GO TO 810
      IF(M3.NE.98)GO TO 807
  808 SUB1(I)=0
      GO TO 720
  807 IF(TEST(1,M3).LT.5.OR.TEST(2,M3).LT.5)GO TO 808
  810 M1=I
      J=ROWCNT(I)
      JJ=ROWCNT(I+1)-1
      DO 809 YZ=J,JJ
      R=ROWDY(YZ)
  809 SUM(R)=SUM(R)-MOVE(I)
      MOVE(I)=0
      GO TO 13
```

```
 54 CONTINUE
720 J=ROWCNT(I)
    JJ=ROWCNT(I+1)-1
    DO 812 YZ=J,JJ
    R=ROWDY(YZ)
812 SUM(R)=SUM(R)-MOVE(I)
    MOVE(I)=0
    IF(SUB1(I).LE.LARGE)GO TO 52
    LARGE=SUB1(I)
    Q=I
 52 CONTINUE
```

This works in almost exactly the same way as the opponent look ahead, appropriate adjustments being made to allow for it being the computer look ahead. The main difference is, that instead of adding 30 to the score of a square which forms a pattern which cannot be stopped, that square is automatically played in. Thus no matter how large the opponent's best score is, if the computer thinks it can win, it does not defend. This is justifiable because the computer is one move ahead.

```
    IF(LARGE.GE.14)GO TO 715
    IF(BIG.GT.15)GO TO 715
```

If either of these situations occur it means that a pattern other than 3 in a row has been discovered during the look ahead. In this case the section of programming which follows is skipped because it is redundant. Also if BIG is greater than 15 it will be bigger than CHARGE (v.P.48) can ever be, therefore there is no point in working out CHARGE.

The next section is a look ahead procedure which looks for the patterns listed in TEST as they would be one move before, i.e., it looks for a situation which in one move could form a pattern listed in TEST. For example, instead of looking for two converging twos, it will look for a row of two converging with a row of one.

```
  500 CHARGE=0
      DO 5555 I=1,64
 5555 SUB1(I)=SUB(I)
      DO 502 I=1,64
      IF(MOVE(I).GT.0)GO TO 502
      MOVE(I)=5
      CALL SUGOAL(MOVE,SUM,SUTEST,I)
      J=ROWCNT(I)
      JJ=ROWCNT(I+1)-1
      DO 18 II=J,JJ
      R=ROWDY(II)
   18 SUM(R)=SUM(R)-MOVE(I)
 5301 MOVE(I)=0
      IF(M1.GT.0)SUB1(I)=SUB1(I)+8
      IF(SUB1(I).LE.CHARGE)GO TO 502
      CHARGE=SUB1(I)
      QQ=I
  502 CONTINUE
      IF(CHARGE.GT.LARGE)GO TO 510
```

This look ahead is basically a simplified version of the other two. The scoring starts from the SUB values. Each blank square is played in in turn - it is used only as a computer look ahead. The subroutine SUGOAL is then called. This subroutine is basically similar to SEARCH.

The patterns are generated in the same way as those in TEST by simply unplaying one move more in the backtrack analysis. SUM is then readjusted and the move unplayed. If a pattern has been found M1 will be greater than 0. In that case a score of 8 is added to the score of the trial square. The last section finds the largest scoring square which is stored in QQ while its score is kept in CHARGE. If there is a pattern at all, the next part of the program will be redundant and is skipped.

What follows is a very simple method of scoring, based on how many counters have been played (known as the "counter scoring system" - CSS). It is a simple way of selecting a square to play in when no patterns can be found during the look aheads.

```
      DO 27 I=1,64
   27 SU1(I)=SUB(I)
      DO 17 I=1,76
      IF(SUM(I).NE.5)GO TO 820
      G=4*I
      H=G-3
      DO 779 I1=H,G
      R=ROW(I1)
  779 SU1(R)=SU1(R)+1
      GO TO 17
  820 IF(SUM(I).NE.10)GO TO 17
      G=4*I
      H=G-3
      DO 778 I1=H,G
      R=ROW(I1)
  778 SUI(R)=SU1(R)+2
   17 CONTINUE
```

As with the other scoring systems, the counter scoring system starts with the initial values of the squares. Each line is then tested in turn to see if its sum adds up to 5 or 10. If it is 5 one point is added to each square in the row, if it is 10 two points are added.

To find which squares are in a certain row, we multiply the number of the line by 4 and set G equal to the result. We then subtract 3 and set H equal to that number. The ROW array is so arranged that these two subscripts of ROW - i.e., ROW(H) and ROW(G) have as their values the beginning and end squares of the line in question, and the ROW subscripts in between have as their values the squares in between.

```
47 MARGE=0
   DO 42 I1=1,64
   IF(SU1(I1).LE.MARGE)GO TO 42
   IF(MOVE(I1).GT.0)GO TO 42
   MARGE=SU1(I1)
   LL=I1
42 CONTINUE
```

This loop finds the square which, using the above system, has the highest score, checking that the square has not been played in already. The method used is the same as for finding BIG.

```
510 IF(CHARGE.LT.BIG)GO TO 56
    IF(CHARGE.LE.MARGE)GO TO 57
    LARGE=CHARGE
    M1=QQ
```

```
      GO TO 13
  715 IF(BIG.GT.LARGE)GO TO 56
   57 IF(LARGE.GT.MARGE)GO TO 53
   58 LARGE=MARGE
      M1=LL
      GO TO 13
   53 M1=Q
      GO TO 13
   56 IF(MARGE.GE.BIG)GO TO 58
      LARGE=BIG
      M1=BIGOPP
```

The scores of the squares selected by different methods are compared. M1 is set to the square with the highest score.

```
   13 SCOUNT=SCOUNT+1
      MOVE(M1)=5
      LAST(SCOUNT)=M1
C     WRITE OUT COMPUTER'S MOVE
      WRITE (6,108)M1
      WRITE (6,112)LAST
```

The selected square is played, i.e., MOVE(M1) is set to 5. The move is added to the list of computer moves in LAST and SCOUNT updated.

```
      XYZ=1
      Y=M1
      M4=.TRUE.
      CALL SEARCH (MOVE,SUM,Y)
      XYZ=0
      IF(M3.EQ.100)GO TO 15
      GO TO 4
```

SEARCH is called to update SUM. At this stage a check is made only to see if the computer has 4 in a row. XYZ is used as a switch to limit the searching in the subroutine. Control is then passed to statement 4, where the opponent's next move is read in.

```
   14 WRITE(6,107)
      GO TO 1
```

This starts a new game. Control is passed to this statement if a minus number is read in as the opponent's move.

```
   15 H=SN*4
      G=H-3
      J=0
      DO 134 I=G,H
      J=J+1
  134 WIN(J)=ROW(I)
      IF(M2.NE.100)GO TO 135
      WRITE(6,106)WIN
      GO TO 64
  135 WRITE(6,231)WIN
```

This simply writes out a winning message and the numbers of the squares in the winning line. SN holds the number of the winning line. To find which squares are in this line the same method is used as in the CSS. These four squares are put in the array WIN which is then written out with an appropriate message - if M2 is 100 it is a computer win.

The rest of the main program forms the backtrack analysis. ROWNO is used to keep track of which rows (or lines) were involved in the winning pattern.

```
C     BACKTRACK ANALYSIS
   64 DO 77 I=1,5
   77 ROWNO (I)=-1
      NOCNT=0
      GCNT=0
      PATT 1=-1
      PATT 2=-1
      PATT 3=-1
      PATT 4=-1
      PATT 5=-1
```

All these variables have to be set before each new backtrack.

```
      IF(M2.EQ.100)GO TO 220
```

There is a separate analysis for computer and opponent wins. The first is for an opponent win - if M2 is 100 the computer has won and the first bit is skipped.

```
      J=1
      N=1
94    GCNT=GCNT+1
      DO 72 I1=J,N
      MOVE(NEXT(COUNT))=5
      MOVE(LAST(SCOUNT))=0
      DO 73 I2=1,64
      IF(MOVE(I2).GT.0)GO TO 73
      MOVE (I2)=1
      DO 75 I4=1,76
      SUM(I4)=0
      H=4*I4
      G=H-3
      DO 76 I3=G,H
      R=ROW(I3)
76    SUM(I4)=SUM(I4)+MOVE(R)
      IF(SUM(I4).NE.4)GO TO 75
      MOVE(NEXT(COUNT))=0
      MOVE(I2)=0
      ROWNO(N)=I4
      COUNT=COUNT-1
      SOUNT=SCOUNT-1
      N=N+1
      GO TO 72
75    CONTINUE
      MOVE(I2)=0
73    CONTINUE
72    CONTINUE
      MOVE(NEXT(COUNT))=0
      GO TO 200
```

The actual backtracking is done in this loop. The program unplays the last computer move by setting it to zero, and sets the last opponent move to a computer move by setting it to 5, i.e., the program looks to see if the opponent would still have won if the computer had played its last move where the opponent did instead of playing where it did. To do this an opponent move is played in each vacant square in turn and the lines added up to see if one comes to four. If such a line is found this means that even if the computer had played its last move where the opponent did, the opponent could still have won, i.e., the opponent's second last move made up at least two separate lines of three and the computer could not stop them both; its last move stopped one line of three but left another one for the opponent to make up to four.

The object of the backtrack analysis is to analyse a board position one move before it is unbeatable (or forcing). There is no point in recognising an unbeatable board position once it has been formed, the program must recognise it before it is formed. With this aim in mind the program unplays the moves on a winning board position until the opponent can no longer win in one move.

Let us suppose the opponent has won by forming this board position.

	1		1
	1	1	
	1		
1	5		

Figure 21

The program has a list of the opponent's and computer's moves in the order they were played in the arrays NEXT and LAST. SCOUNT and COUNT are counts of how many opponent and computer moves have been played so that

NEXT(COUNT) contains the latest opponent move. If the last opponent move is set to a computer move, and the last computer move to zero, this gives the position:

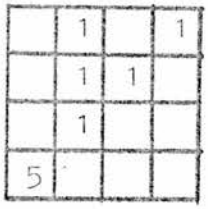


Figure 22

Obviously the opponent still can win by playing in the square which originally contained the computer's last move. Therefore the board position

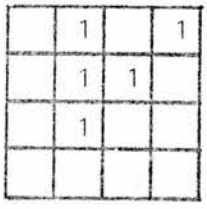


Figure 23

is unbeatable - wherever the computer plays the opponent can still win. The program must therefore unplay another move to achieve its goal of storing a board position one move before it is unbeatable. The program sets to zero the square it has just set to 5, i.e., the square which was the opponent's last move, and similarly with the square which contained the computer's original last move.

A note has already been taken of the number of the winning line in the variable SN. ROWNO takes note of the other lines involved in the winning pattern. In the above example this is the line in which the opponent could still win after the computer's last move was changed. This will nearly always be the line the computer stopped with its last move. In this case there are only two lines involved in the winning pattern. If there are more lines involved they are stored one at a time in ROWNO, each time a move is unplayed.

To get the second last moves, we subtract one from SCOUNT and COUNT, and in order to go through the loop once more we add 1 to N. The length of the loop depends on N, which is set to 1 initially and added to each time it is found necessary to unplay another move. After going through the loop again (unplaying another move) the board position will be as in Figure 24.

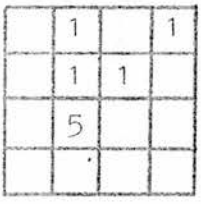


Figure 24

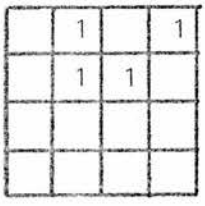


Figure 25

In this case the opponent can no longer win in one move and the board position in Figure 25 is considered unbeatable in one move, i.e., unless it is stopped by playing where the five is in Figure 24.

Let us take a more complicated example and look at the sequence of board positions during the backtracking analysis

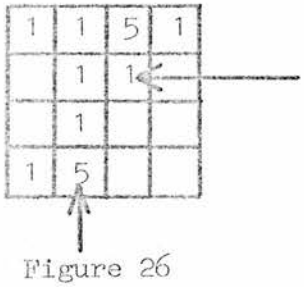


Figure 26

the arrows indicate the latest two moves, i.e., the ones which will be unplayed if necessary.

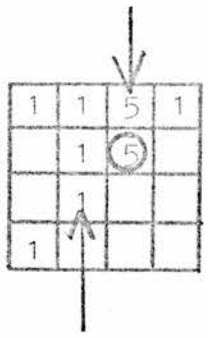


Figure 27

The altered move is ringed. The computer's last move is played where the opponent's last move was, its original last move being left blank. The opponent can still win in one move so the next two moves are unplayed and the altered move set to zero.

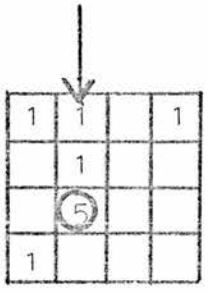


Figure 28

The opponent can still win in one, so another set of moves must be unplayed.

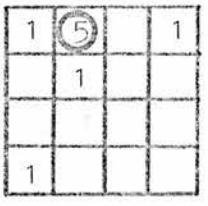


Figure 29

The computer's last move did not influence this pattern and so is not shown. The opponent can no longer win in one, therefore this board position without the five is the one that will be analysed and stored as one move before being unbeatable (or forcing).

The same process with appropriate adjustments is gone through if the computer wins, i.e., the opponent's move is played where the computer's last move was, and tests made to see if the computer can still win in one. This section is skipped if the opponent won.

```
220 N=1
    J=1
 95 GCNT=GCNT+1
    DO 222 I=J,N
    MOVE(LAST(SCOUNT)=1
    MOVE(NEXT(COUNT)=0
    DO 223 I1=1,64
    IF(MOVE(I1).GT.0)GO TO 223
    MOVE(I1)=5
    DO 225 I2=1,76
    SUM(I2)=0
    H=4*I2
    G=H-3
    DO 226 I3=G,H
    R=ROW(I3)
226 SUM(I2)=SUM(I2)+MOVE(R)
    IF(SUM(I2).NE.20)GO TO 225
    MOVE(LAST(SCOUNT))=0
    MOVE(I1)=0
    ROWNO(N)=I2
    COUNT=COUNT-1
    SCOUNT=SCOUNT-1
    N=N+1
225 CONTINUE
    MOVE(I1)=0
223 CONTINUE
222 CONTINUE
    MOVE(LAST(SCOUNT))=0
```

The next section of the program forms the analysis of the board position to find the winning pattern. Firstly the lines of the final board position reached during the backtrack analysis are summed, i.e., the lines of the board position which is one move before being unbeatable.

```
200 DO 43 I=1,76
    SUM(I)=0
    H=4*I
    G=H-3
    DO 43 I1=G,H
    R=ROW(I1)
    SUM(I)=SUM(I)+MOVE(R)
 43 CONTINUE
```

The sums of the lines involved in the winning pattern, i.e., SUM(SN) and the sums of the lines stored in ROWNO, will be the numbers listed in the TEST array as a new pattern. However, it is very important that the lines are listed in TEST in the right order. This is because of the way SEARCH works. SEARCH takes the sums of the lines of a given board position and looks for each of the patterns in TEST in a specific way. It looks first for a line adding up to the first number of the pattern in TEST, then for a line adding up to the second number, which has a blank square in common with the first line, etc. What is important about the ordering of the lines, is that SEARCH looks for the right line intersection. With two lines it does not matter which line it finds first. However, with any greater number it is important; e.g., for the board position:-

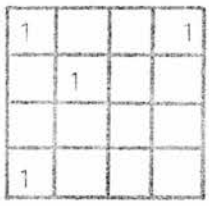


Figure 30

it is important to look for a line of two with a blank square in common with a line of one, which has a blank square in common with another line of two, i.e., it is the line of one which must have a blank square in common with the other two. Similarly with the board position in Figure 31, line 1 must have a blank square with three separate rows adding up to two.

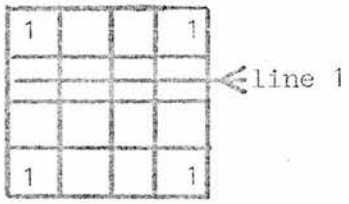


Figure 31

In the analysis of a new board position it is important to find which line it is that must intersect with the others. This line is one which is built up by making up threes in the other lines and is always either the winning line or the last line stopped by the opponent before a win. Because of the way SEARCH works it must always be listed as the second line in TEST, i.e., it must always be the second line SEARCH looks for. This is the only thing that matters in the ordering, the rest of the lines can be in any order.

To find this line it is necessary simply to know which line has most blank squares in common with all the other lines involved, e.g.:

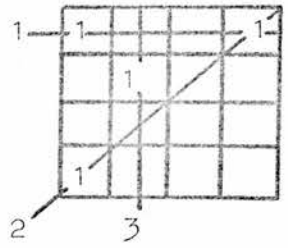


Figure 32

Line 1 has one blank square in common with any other line involved; similarly with line 2. However line 3 has two blank squares in common with the other lines involved, one in common with line 1 and one in common with line 2. This therefore is the line which must go second in the pattern.

```
IF(N.LT.3.OR.GCNT.GT.1)GO TO 88
```

i.e., if less than three lines are involved skip the next section which does the ordering.

```
G=4*SN
H=G-3
G1=4*ROWNO(I)
H1=G1-3
G2=4*ROWNO(2)
H2=G2-3
```

This takes the three lines to be ordered, the winning line and the first two found in the backtrack analysis and sets the variables G, H, G1, H1, G2, H2 to the ROW subscripts which will contain the first and last squares making up these lines.

```
      DO 403 I=1,3
  403 X(I)=0
```

The array X is used to count the intersecting squares in each line.

X(1) keeps count for SN, the winning line.

X(2) for ROWNO(I)

and X(3) for ROWNO(2)

This array is set to zero.

All three lines are now compared with each other in respect of intersecting blank squares.

```
      DO 80 I=H,G
      R=ROW(I)
      IF(MOVE(R).GT.0)GO TO 80
      DO 81 I1=H1,G1
      R1=ROW(I1)
      IF(MOVE(R1).GT.0)GO TO 81
      IF(R.NE.R1)GO TO 81
      X(1)=X(1)+1
      X(2)=X(2)+1
   81 CONTINUE
   80 CONTINUE
```

The first loop takes each square in SN in turn checks that it is a blank square, as played in squares do not count, and each square is compared with all of the blank squares in ROWNO(I). If any square in SN is also in ROWNO(I) then one is added to the counters of both lines, i.e., X(1) and X(2). This process is repeated until every line has been compared with all the others.

```
      DO 82 I=H,G
      R=ROW(I)
      IF(MOVE(R).GT.0)GO TO 82
      DO 83 I1=H2,G2
      R1=ROW(I1)
      IF(MOVE(R1).GT.0)GO TO 83
      IF(R.NE.R1)GO TO 83
      X(1)=X(1)+1
      X(3)=X(3)+1
   83 CONTINUE
   82 CONTINUE
      DO 84 I=H1,G1
      R = ROW(I)
      IF(MOVE(R).GT.0)GO TO 84
      DO 85 I1=H2,G2
      R1=ROW(I1)
      IF(MOVE(R1).GT.0)GO TO 85
      IF(R.NE.R1)GO TO 85
      X(2)=X(2)+1
      X(3)=X(3)+1
   85 CONTINUE
   84 CONTINUE
```

Now X(1), X(2) and X(3) are compared and put in MOST in descending order so that MOST(1) contains the subscript of X which held the score of the line with the most intersecting squares.

In the above example (Figure 32) let us suppose line 2 is the winning line (SN), ROWNO(I) is line 3 and ROWNO(2) is line 1. In this case X(1), keeping count of the intersecting lines in SN will be 1; X(2) for ROWNO(I) will be 2, and X(3) will be 1. When these scores are compared MOST(1) will be 2 as X(2) has the biggest score and since the other two are equal they will be put in MOST in the order this bit of the program comes to them.

```
    DO 400 I=1,3
    LARGE=0
    DO 401 I1=1,3
    IF(X(I1).LE.LARGE)GO TO 401
    LARGE=X(I1)
    Q=I1
401 CONTINUE
    X(Q)=0
    MOST(I)=Q
400 CONTINUE
```

Having found the row with the most intersecting squares it only remains to put it and the other rows in the right order in the pattern. The variables, PATT 1-5, are set to the sums of the lines involved, in the order they will be in the TEST array. Each pattern in the TEST array has five elements which allows room for a pattern involving up to five lines, each element of the pattern containing the sum of one line. If a pattern involves less than five lines the remaining elements are set to -1.

```
      IF(MOST(1).NE.1)GO TO 410
      PATT 2=SUM(SN)
      IF(MOST(2).NE.2)GO TO 411
      PATT 1=SUM(ROWNO(1))
      PATT 3=SUM(ROWNO(2))
      GO TO 87
  411 PATT 1=SUM(ROWNO(2))
      PATT 3=SUM(ROWNO(1))
      GO TO 87
  410 IF(MOST(1).NE.2)GO TO 420
      PATT 2=SUM(ROWNO(1))
      IF(MOST(2).NE.1)GO TO 412
      PATT 1=SUM(SN)
      PATT 3=SUM(ROWNO(2))
      GO TO 87
  412 PATT 1=SUM(ROWNO(2))
      PATT 3=SUM(SN)
      GO TO 87
  420 PATT 2=SUM(ROWNO(2)
      IF(MOST(2).NE.1)GO TO 413
      PATT 1=SUM(SN)
      PATT 3=SUM(ROWNO(1))
      GO TO 87
  413 PATT 1=SUM(ROWNO(1))
      PATT 3=SUM(SN)
```

The program tests to see which of the three lines is the one which must go second in the pattern, i.e., which line has the most blank squares in common with the other two lines.

If MOST(1) is 1 then the winning line (SN) is the line which must go second. PATT 2, i.e., the second element in the pattern, is set to SN. The program then tests to see which line has the next most intersecting squares. This line will be in MOST(2). It is not really important in what order the rest of the lines are put in the pattern so long as the first element of the pattern is set to the sum of one and the third element to the sum of the other (and not the first and third elements set to the sum of the same line).

If MOST(1) is not 1 the program tests to see if it is 2. If so the same process is gone through with PATT 2 set to sum of ROWNO(2). If not, MOST(1) must be 3 and the process is gone through with PATT 2 as the sum of ROWNO(2).

```
 87 IF(N.GT.2)GO TO 89
 88 PATT 1=SUM(SN)
    IF(N.EQ.1)GO TO 44
    PATT 2=SUM(ROWNO(1))
    IF(N.EQ.2)GO TO 44
    PATT 3=SUM(ROWNO(2))
 89 IF(N.EQ.3)GO TO 44
    PATT 4=SUM(ROWNO(3))
    IF(N.EQ.4)GO TO 44
    PATT 5=SUM(ROWNO(4))
```

The programming between statement 88 and instruction before statement 89 are relevant only if the analysed board position involved less than three lines (in which case N will be less than 3) because this part of the program sets PATT 1 and PATT 2, which are already set if more than two lines are involved.

Every time an element of the pattern is set, a test is made to see how many lines are involved in this particular pattern. As many elements of the patterns are set as there are lines involved, the rest are left as -1.

```
44 IF(GCNT.GT.1)GO TO 97
```

GCNT is a count used to skip those sections of the backtrack analysis not relevant to the setting of SUTEST. To work out the pattern to be put in SUTEST the relevant backtracking loop has to be gone through once more, so that one more move is unplayed. When this has been done it is inapplicable to call subroutine TESTT, and the new pattern must be listed in SUTEST not TEST. To skip the relevant part of the program the count GCNT is used. It is set to 0 at the beginning of the backtrack analysis. Each time the program comes to the beginning of one of the backtracking loops, one is added to GCNT. This will happen once during the original backtracking and once when working out the pattern for SUTEST. At this stage GCNT will be 1 and the following program instructions are not skipped.

```
C     CALL SUBROUTINE TESTT TO SEE IF THIS PATTERN HAS ALREADY BEEN LISTED
      IF NOT ADD IT TO THE TEST ARRAY
   45 CALL TESTT (PATT 1, PATT 2, PATT 3, PATT 4, PATT 5, ANS, SAME)
```

TESTT is a subroutine which checks whether the pattern worked out by backtrack analysis is already listed in TEST. ANS will be set to -1 if this is the case. If a pattern is found to be already listed in TEST, the variable SAME is set to the number of the pattern, i.e., to the relevant subscript of TEST. This is used to update AJUST.

```
      NOCNT=NOCNT+1
```

NOCNT keeps count of how many times TESTT is called during the backtrack analysis.

```
      IF(M2.EQ.100)GO TO 251
      IF(SAME.GT.0)AJUST(SAME)=AJUST(SAME)+1
```

The array AJUST is used to adjust the scoring system. It has as many subscripts as there are patterns in TEST and every pattern has a score. Originally they all score zero but every time the computer loses to a pattern already listed, the relevant subscript in AJUST has one point added to it. During the lookaheads the score of a pattern in AJUST is added to the marks allotted to that particular pattern (v.P.41).

```
  251 IF(ANS.EQ.-1)GO TO 46
      PCOUNT=PCOUNT+1
      TEST (1,PCOUNT)=PATT 1
      TEST (2,PCOUNT)=PATT 2
      TEST (3,PCOUNT)=PATT 3
      TEST (4,PCOUNT)=PATT 4
      TEST (5,PCOUNT)=PATT 5
```

This adds the new pattern to the TEST array. If the pattern is already listed this section is skipped. A new pattern is always listed both as an attacking and as a defending pattern; i.e., it is listed as multiples of 5 and multiples of 1. If the new pattern was derived from an opponent win, the elements of the pattern are multiplied by 5 to make it a computer pattern, and if it was a computer win the elements are divided by 5.

When the second version of the pattern is generated subroutine TESTT

is called again and the pattern added to the TEST array. Therefore once TEST has been called twice and NOCNT is greater than 1, the new pattern is adequately represented in TEST.

```
      IF(NOCNT.GT.1)GO TO 33
      IF(MC.EQ.100)GO TO 254
```

The next section is relevant only if the new pattern came from an opponent win and is first listed in terms of an opponent board position, i.e., as multiples of 1.

```
      IF(PATT 1.GT.0)PATT 1=PATT 1*5
      IF(PATT 2.GT.0)PATT 2=PATT 2*5
      IF(PATT 3.GT.0)PATT 3=PATT 3*5
      IF(PATT 4.GT.0)PATT 4=PATT 4*5
      IF(PATT 5.GT.0)PATT 5=PATT 5*5
      GO TO 45
```

This multiplies the elements of the pattern by 5 to list it in terms of a computer board position, and goes back to call TESTT.

```
  245 IF(NOCNT.GT.1)GO TO 33
      PATT 1=PATT 1/5
      IF(PATT 2.GT.0)PATT 2=PA1. 2/5
      IF(PATT 3.GT.0)PATT 3=PATT 3/5
      IF(PATT 4.GT.0)PATT 4=PATT 4/5
      IF(PATT 5.GT.0)PATT 5=PATT 5/5
      GO TO 45
```

This does the reverse procedure if the pattern came from a computer win.

```
 97 IF(PATT 4.GE.0)GO TO 96
    IF(PATT 2.LT.0)GO TO 96
```

The next section further analyses the winning board position for inclusion in SUTEST. SUTEST only lists patterns involving less than 4 and more than 1 lines as these are generally the most useful. Therefore if PATT 4 is set, i.e., greater than -1, the pattern is not listed. Similarly if PATT 2 is not set.

```
    IF(M2.EQ.100)GO TO 250
    PATT 1=PATT 1*5
    PATT 2=PATT 2*5
    PATT 3=PATT 3*5
250 SUCNT=SUCNT+1
    SUTEST (1,SUCNT)=PATT 1
    SUTEST (2,SUCNT)=PATT 2
    SUTEST (3,SUCNT)=PATT 3
```

SUTEST lists only computer patterns therefore if the new pattern came from a computer win it is immediately listed, otherwise the elements are multiplied by 5 and listed.

In fact this part of the program is not reached until the following section of the program has been completed. The section following goes back to the appropriate computer or opponent backtracking loop, unplays one more move, analyses the resulting board position to form a pattern which is then processed by the above section of program.

```
33 IF(GCNT.NE.1)GO TO 96
```

The next section is only relevant if the appropriate backtracking loop has only been used once. Once the computer has gone through the loop again to analyse the board position for SUTEST, GCNT will be 2 and when this statement is reached the computer will skip to the end of the whole backtrack analysis.

```
N=N+1
J=N
```

These two variables control the number of times the backtracking loop is gone through (v.P.53). To unplay 1 move it need only be done once.

```
COUNT=COUNT-1
SCOUNT=SCOUNT-1
IF(M2.EQ.100)GO TO 95
GO TO 94
```

1 is subtracted from SCOUNT and COUNT to point to the move before the last one unplayed.

The beginning of the computer backtrack loop is at statement 95 and the opponent one at statement 94.

```
46 WRITE(6,213)
96 GO TO 1
```

This begins a new game.

```
100 FORMAT(I3)
101 FORMAT(24I3)
102 FORMAT(' NEXT ARRAY IS',1X,4I3,/,3(15X,4I3,/),4(15X,4I3,/))
104 FORMAT(16I2)
105 FORMAT(' ILLEGAL MOVE,TRY AGAIN')
106 FORMAT(' COMPUTER WINS ON',4I4)
107 FORMAT(' RESTART GAME')
108 FORMAT(' COMPUTER'S MOVE IS SQUARE NUMBER',I3)
109 FORMAT(4I2)
112 FORMAT(' LAST ARRAY IS',1X,4I3,/,3(15X,4I3,/),4(15X,4I3,/))
114 FORMAT(26I3)
213 FORMAT(' THIS PATTERN IS ALREADY LISTED')
231 FORMAT(' YOU WIN ON',4I4)
850 FORMAT(' M1',I3)
    END
```

SUBROUTINE TESTT is used to check that the pattern found by the analysis of a winning board position is not already listed. If the pattern is listed the variable ANS is set to -1 and the variable SAME to the number of the pattern in the TEST array. If not they are both left as 0.

```
    SUBROUTINE TESTT(A,B,C,D,E,ANS,SAME)
    IMPLICIT INTEGER(A-Z)
    DIMENSION ROW(304),ROWDY(304),ROWCNT(65)
   1 TEST(5,32),SUB(64),LAST(32),NEXT(32),M11(2,16)
    COMMON ROW,TEST,M1,M2,M3,M4,M11,SN,CNT,ROWDY,ROWCNT
    SAME=0
    ANS=0
    IF(A.LT.0)GO TO 16
    TEST1=A
```

```
 2 TEST2=B
   TEST3=C
   TEST4=D
   TEST5=E
```

The dummy variables A,B,C,D,E hold the values of the 5 elements of the pattern, e.g., for the pattern 2,1,2; A would be 2, B would be 1, C would be 2, and D and E would be -1.

```
   DO 22 I1=1,32
   IF(TEST(1,I1).EQ.-1)GO TO 16
   IF(TEST(1,I1).NE.TEST1)GO TO 22
   IF(TEST(2,I1).NE.TEST2)GO TO 22
   IF(TEST3,I1).NE.TEST3)GO TO 22
   IF(TEST(4,I1).NE.TEST4)GO TO 22
   IF(TEST(5,I1).NE.TEST5)GO TO 22
   ANS=-1
   SAME=I1
22 CONTINUE
16 RETURN
   END
```

If the program finds the first element in one of the patterns listed in TEST to be equal to the first element of this new pattern, it looks to see if the second element of the same pattern in TEST is equal to the second element of the new pattern. If so it checks the third, fourth and fifth elements and if they all tally, then ANS is set to -1, and SAME is set to the number reached in the first loop, which corresponds to the number of the pattern as it is listed in the TEST array.

If at any stage of the search an element in the new pattern differs from the corresponding element in the TEST array, then the next pattern in the TEST array is tried.

If the first element of a TEST pattern equals -1 this means that all the patterns listed in TEST have been tried and the new pattern is not amongst them and control is returned to the main program with ANS set to 0.

SUBROUTINE SEARCH

This subroutine does the bulk of the work in the selection of a move. Basically, given a board position it looks to see if it contains any of the patterns listed in TEST. The patterns are all listed as combinations of sums of the lines, therefore the SUBROUTINE must keep an up to date account of these sums. SEARCH is only called after a new move has been made either in reality or in a look-ahead. Therefore the first thing it does is to update the relevant line sums to take account of the new move.

It then takes each pattern in turn and checks if any pattern can be formed by any combination of the sums of the existing lines. It is not enough that the appropriate lines should exist, e.g., that three lines should exist which add up to the three elements of a pattern. These three lines must have a certain relationship to each other. The relationship is always the same. The first line (which corresponds to the first element of the pattern) must have a blank square in common with a line adding up to the second element in the pattern and (if a third line is involved) the second line must have a blank square in common with the third line etc.

```
      SUBROUTINE SEARCH(MOVE,SUM,Y)
      IMPLICIT INTEGER(A-Z)
      LOGICAL M4
      DIMENSION MOVE(64),SUM(76),ROW(304),ROWDY(304),
     1 ROWCNT(65),TEST(5,32),SUTEST(3,20),SUB(64),LAST(32),NEXT(32),
     2 M11(2,16)
      COMMON ROW, TEST,M1,M2,M3,M4,M11,SN,CNT,ROWDY,ROWCNT,
     1 XYZ
      M3=0
      M2=0
```

M3 and M2 are set if a pattern is found, and must always be set back to zero for a new board position, as there may be no pattern in the new board position. The same applies to M11CNT and M11.

```
   68 M11CNT=0
      DO 43 I=1,2
      DO 43 I1=1,16
   43 M11(I,I1)=0
      IF(Y.EQ.0)GO TO 74
```

Y will only ever be zero if the opponent lets the computer move first by setting his move to 0. In this case there is no updating of the sums of the lines and the following section is skipped.

```
      J=ROWCNT(Y)
      JJ=ROWCNT(Y+1)-1
      DO 80 I=J,JJ
      R=ROWDY(I)
   80 SUM(R)=SUM(R)+MOVE(Y)
```

The dummy variable Y is always set to the number of the new move in the MOVE array. The updating of the lines is done in exactly the same way as in the main program. For the sake of clarity the first four patterns in TEST are tested for separately. The first 4 patterns are 4-in-a-row, for both sides, and 3-in-a-row for both sides, i.e., 20,4,15,3. These situations are treated in a different way from other patterns. Looking for these four patterns in any case is very quickly done as each pattern involves only one line. If any of these patterns exist (unless it is during a look ahead) there is no point in looking for any other pattern as the game is either finished or about to be finished, unless the computer can stop the opponent making a 4, in which case it must do this immediately.

```
74 DO 20 I=1,4
   DO 20 I1=1,76
   IF(SUM(I1).NE.TEST(1,I))GO TO 20
   SN=I1
   IF(I.GT.2)GO TO 15
   M3=100
   IF(SUM(I1).EQ.20)M2=100
   RETURN
15 M3=98
   IF(SUM(I1).EQ.15)M2=98
   GO TO 19
20 CONTINUE
```

Each pattern in turn is compared to all the sums of the lines.
The first two patterns are 20 and 4. If one of these exists M3 is set to 100. SN takes note of the number of the line with the pattern in it.

A separate test is made to see if in fact it is a computer win, in which case the line will add up to 20.

If so M2 is set to 100. Control is returned to the main program. If the third or fourth pattern exists M3 is set to 98, and M2 is set to 98 if it is a computer pattern, and the rest of the loop is skipped.

```
19 IF(M3.NE.98)GO TO 32
   G=4*SN
   H=G-3
   DO 30 I=H,G
   R=ROW(I)
   IF(MOVE(R))31,31,30
30 CONTINUE
31 M1=R
```

The familiar process is used to find out which squares are in this line and which one is blank. When the blank square is found M1 takes a note of it.

```
   IF(M4)RETURN
   M11CNT=1
   M11(2,M11CNT)=98
   M11(1,M11CNT)=M1
```

M4 is a logical variable which is TRUE if all that is wanted is to know if a pattern exists at all. It is FALSE if a list of all the patterns existing is required. In most cases this will be the distinction between a real board position and a look ahead. If M4 is TRUE, control is returned to the main program. Otherwise M11CNT is set to 1

(this will always be the first pattern) and M11 takes note of the pattern number (98) and the key square (M1). The program then tests for any other patterns.

```
32 IF(XYZ.EQ.1)RETURN
   DO 21 I=5,32
   M3=I
   TEST1=TEST(1,I)
   IF(TEST1)29,2,2
 2 TEST2=TEST(2,I)
71 TEST3=TEST(3,I)
   TEST4=TEST(4,I)
   TEST5=TEST(5,I)
```

As the first four patterns have been dealt with, the loop starts at 5. If TEST1 is ever less than 0 it means that all the listed patterns have been tried and the computer skips to the end of the subroutine. The program now looks for a line whose SUM=TEST1.

```
   DO 22 I1=1,76
   IF(SUM(I1)-TEST1)22,4,22
```

Having found it, it looks for a blank square on that line.

```
 4 G=4*I1
   H=G-3
   DO 23 I2=H,G
   M1=ROW(I2)
   IF(MOVE(M1))23,10,23
```

```
C     M1 TAKES NOTE OF THE BLANK SQUARE
C     PROGRAM LOOKS FOR A LINE WHOSE SUM EQUALS TEST2
   10 DO 24 I3=1,76
      IF(SUM(I3)-TEST3)24,6,24
```

Having found it, the program looks to see if the blank square in the first line is also in this line and checks that this line is not the same line as the first line.

```
    6 IF(I3-I1)7,24,7
    7 G1=4*I3
      H1=G1-3
      DO 25 I4=H1,G1
      IF(M1 ROW(I4))25,8,25
```

If this condition is satisfied the program tests whether the pattern involves any more lines. If not TEST3 will be -1.

```
    8 IF(TEST3)18,9,9
   18 IF(M4)RETURN
      M11CNT=M11CNT+1
      M11(1,M11CNT)=M1
      M11(2,M11CNT)=M3
      GO TO 21
```

If all the conditions of the pattern are satisfied either control is returned to the main program (if M4 is TRUE), or the pattern is listed in M11 and the program goes on to test whether the next pattern listed in TEST also exists in the given board position. However if another line is involved there are more conditions to fulfill.

```
C     PROGRAM FINDS ANY BLANK SQUARE,M2,ON THE LINE WITH SUM=TEST2
    9 DO 26 I5=H1,G1
      M2=ROW(I5)
      IF(MOVE(M2))26,11,26
C     FIND A ROW WHOSE SUM EQUALS TEST3
   11 DO 27 I6=1,76
      IF(SUM(I6)-TEST3)27,12,27
C     CHECK THAT IT IS A NEW LINE
   12 IF(I6-I1)13,27,13
   13 IF(I6-I3)14,27,14
C     FIND THE BLANK SQUARE M2 ON THE ROW WITH SUM=TEST3
   14 G2=4*I6
      H2=G2-3
      DO 28 I7=H2,G2
      IF(M2-ROW(I7))28,42,28
C     ARE ANY MORE LINES INVOLVED
   42 IF(TEST4)45,47,47
C     IF SO RETURN OR LIST PATTERN
   45 IF(M4)RETURN
      M11CNT=M11CNT+1
      M11(1,M11CNT)=M1
      M11(2,M11CNT)=M3
      GO TO 21
C     IF NOT LOOK FOR A LINE WITH SUM=TEST4
   47 DO 50 K3=1,76
      IF(SUM(K3)-TEST4)50,51,50
C     CHECK IT IS A DIFFERENT LINE
   51 IF(K3.EQ.I6)GO TO 50
      IF(K3.EQ.I3)GO TO 50
      IF(K3.EQ.I1)GO TO 50
```

```
C       FIND A BLANK SQUARE ON THIS ROW WHICH IS ALSO IN THE LINE
C       WITH SUM=TEST2
        G3=4*K3
        H3=G3-3
        DO 60 I8=H3,G3
        R4=ROW(I8)
        IF(MOVE(R4))60,69,60
   69   DO 61 I9=H1,G1
        R5=ROW(I9)
        IF(R4-R5)61,70,61
   70   IF(TEST5)49,65,65
C       IF ANOTHER LINE IS INVOLVED,FIND IT OTHERWISE RETURN OR LIST PATTERN
   49   IF(M4)RETURN
        M11CNT=M11CNT+1
        M11(1,M11CNT)=M1
        M11(2,M11CNT)=M3
        GO TO 21
   65   DO 52 K4=1,76
        IF(SUM(K4)-TEST5)52,53,52
   53   IF(K4.EQ.K3)GO TO 52
        IF(K4.EQ.I6)GO TO 52
        IF(K4.EQ.I3)GO TO 52
        IF(K4.EQ.I1)GO TO 52
        IF(M4)RETURN
        M11CNT=M11CNT+1
        M11(1,M11CNT)=M1
        M11(2,M11CNT)=M3
        GO TO 21
```

```
   52 CONTINUE
   61 CONTINUE
   60 CONTINUE
   50 CONTINUE
   28 CONTINUE
   27 CONTINUE
   26 CONTINUE
   25 CONTINUE
   24 CONTINUE
   23 CONTINUE
   22 CONTINUE
   21 CONTINUE
C     IF THIS POINT IS REACHED NO PATTERN EXISTS
   29 M1=0
      M2=0
      M3=0
```

These three variables (M1, M2 and M3) are all set at some stage in the subroutine, but if they are set on return to main program it is taken to mean that a pattern exists, therefore they must be set back to 0 here, to indicate that no pattern exists.

```
   16 RETURN
      END
```

SUBROUTINE SUGOAL

The subroutine SUGOAL performs more or less the same function for SUTEST as subroutine SEARCH does for TEST; i.e., given a board position it looks to see if there exists in it any of the patterns listed in SUGOAL. (v.P.46)

```
      SUBROUTINE SUGOAL (MOVE,SUM,SUTEST,Y)
      IMPLICIT INTEGER(A-Z)
      DIMENSION MOVE(64),SUM(76),ROW(304),ROWDY(304),ROWCNT(65),TEST(5,32),
     1 SUTEST(3,20),M11(2,16)
      COMMON ROW,TEST,M1,M2,M3,M4,M11,SN,CNT,ROWDY,ROWCNT.
      M1=0
      J=ROWCNT(Y)
      JJ=ROWCNT(Y+1)-1
      DO 1 I=J,JJ
      R=ROWDY(I)
    1 SUM(R)=SUM(R)+MOVE(Y)
```

Using exactly the same method as in subroutine SEARCH, the sums of the lines affected by the new move are updated by the addition of the new move Y.

```
  523 DO 50I I=1,20
      M3=I
      GOAL1=SUTEST(1,I)
      IF(GOAL1)3,502,502
  502 GOAL2=SUTEST(2,I)
      GOAL3=SUTEST(3,I)
```

GOAL1, 2 and 3 are set in turn to the three elements of the patterns in SUTEST. If GOAL1 is ever -1 it means that all the listed patterns have been tried.

```
      DO 503 I1=1,76
      IF(SUM(I1)-GOAL1)503,504,503
```

Using exactly the same method as in subroutine SEARCH, the program looks for a line whose sum equals GOAL1.

```
  504 G=4*I1
      H=G-3
      DO 505 I2=H,G
      M1=ROW(I2)
      IF(MOVE(M1))505,506,505
C     LOOKS FOR A BLANK SQUARE (M1) IN THAT LINE
  506 DO 507 I3=1,76
      IF(SUM(I3)-GOAL2)507,508,507
  508 IF(I3-I1)509,507,509
C     LOOKS FOR A LINE WHOSE SUM EQUALS GOAL2 AND CHECKS THAT THIS
C     LINE IS A DIFFERENT ONE FROM THE FIRST ONE.
  509 G1=4*I3
      H1=G1-3
      DO 510 I4=H1,G1
      IF(M1-ROW(I4))510,511,510
```

```
C     FINDS THE BLANK SQUARE M1 IN THIS ROW
  511 IF(GOAL3)512,513,513
  512 RETURN
C     IF GOAL3 IS -1 THEN NO MORE LINES ARE INVOLVED IN THE PATTERN AND
C     CONTROL RETURNS TO MAIN PROGRAM.
  513 DO 514 I5=H1,G1
      M2=ROW(I5)
      IF(MOVE(M2))514,515,514
C     OTHERWISE LOOK FOR THE BLANK SQUARE (M2) ON THE LINE WITH SUM EQUAL
C     TO GOAL2
  515 DO 516 I6=1,76
      IF(SUM(I6)-GOAL3)516,517,516
  517 IF(I6-I1)518,516,518
  518 IF(16-I3)519,516,519
C     FIND LINE WITH SUM EQUAL TO GOAL3 AND CHECK IT IS A DIFFERENT LINE
C     FROM EITHER OF THE TWO ALREADY FOUND
  519 G2=4*I6
      H2=G2-3
      DO 520 I7=H2,G2
      IF(M2-ROW(I7))520,521,520
  521 RETURN
C     LOOK FOR M2 ON THE LINE WITH SUM EQUAL TO GOAL3 AND RETURN
  520 CONTINUE
  516 CONTINUE
```

```
514 CONTINUE
510 CONTINUE
507 CONTINUE
505 CONTINUE
503 CONTINUE
501 CONTINUE
```

If this point is reached no pattern exists and to indicate this M1 and M3 are set to zero before returning to the main program.

```
  3 M1=0
    M3=0
    RETURN
    END
```

FLOWCHARTS

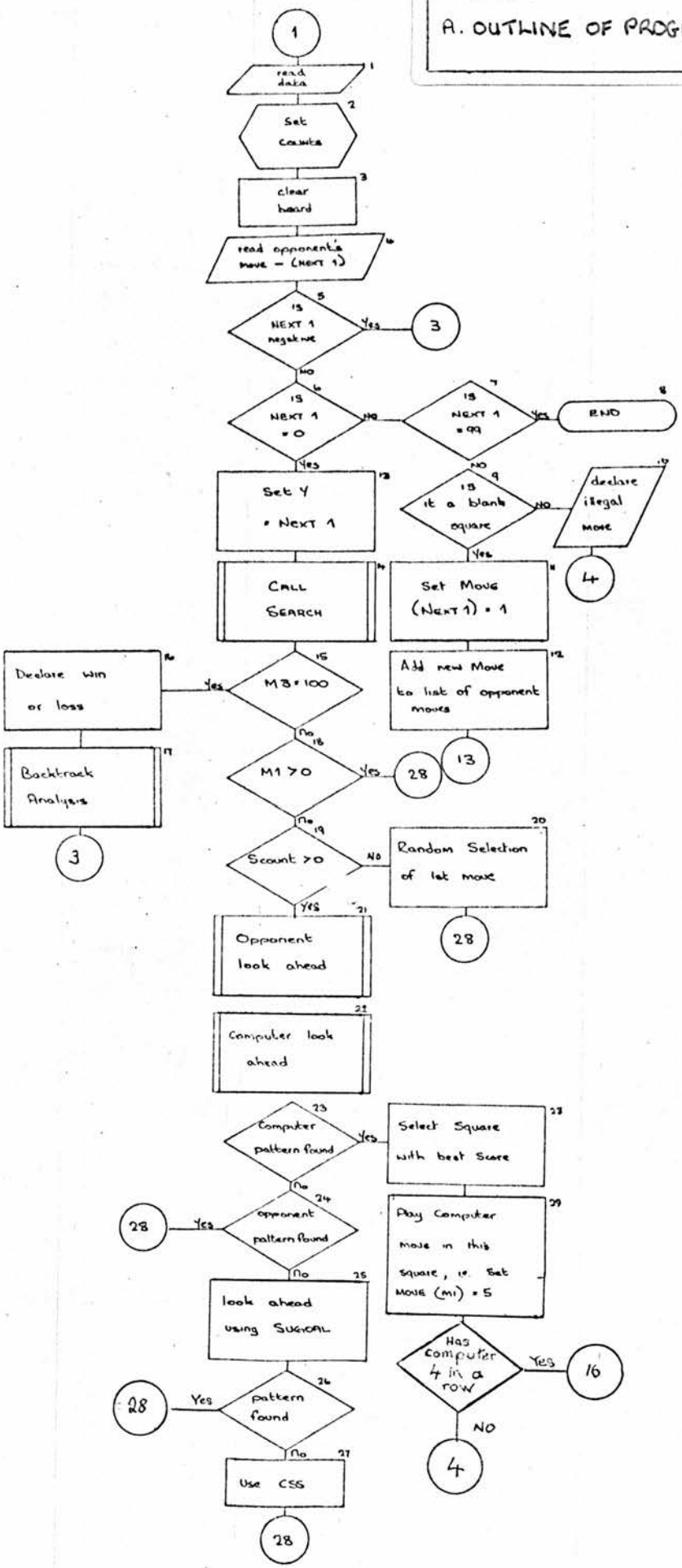
NOTATION.

Label (3) means 'go to box labelled 3 on the same flowchart'.

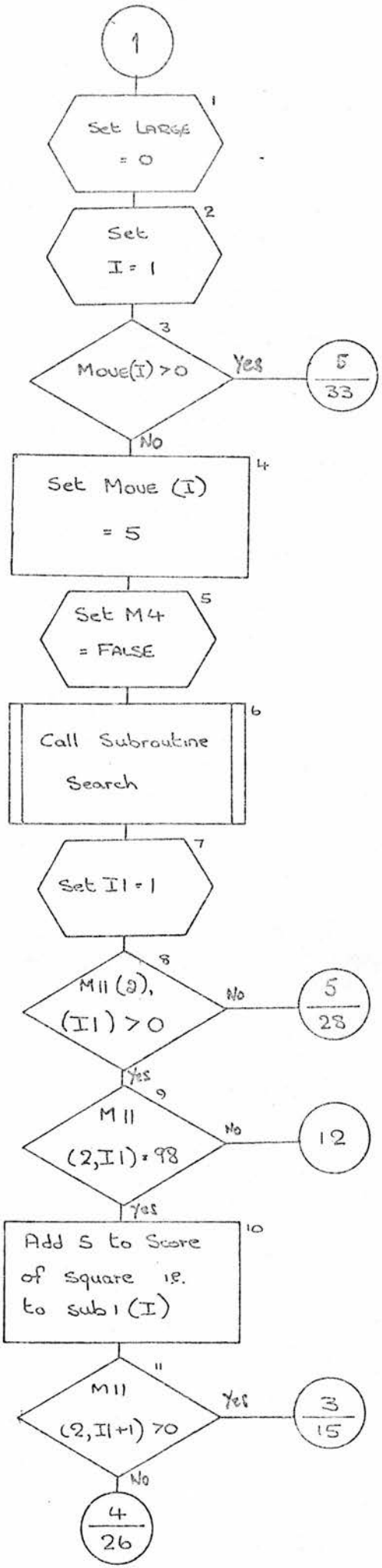
Label ($\frac{5}{33}$) means 'go to box labelled 33 on flowchart F5'.

FLOWCHARTS

A. OUTLINE OF PROGRAM



B. COMPUTER LOOK AHEAD LOO



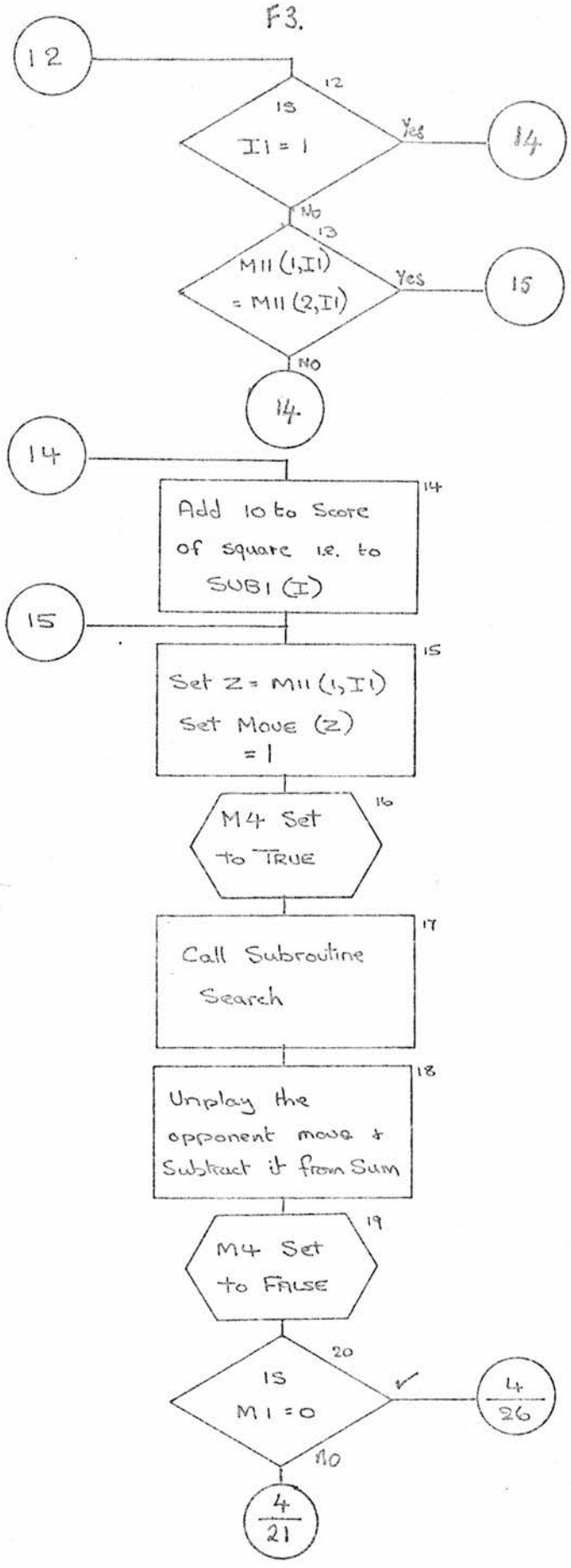
Play a computer move in each blank square in turn & examine the effect.

M4 set to FALSE to indicate to search that this is a look ahead, & that return should not be made until all existing patterns have been listed.

Examine list of patterns to see how many SEARCH has found.

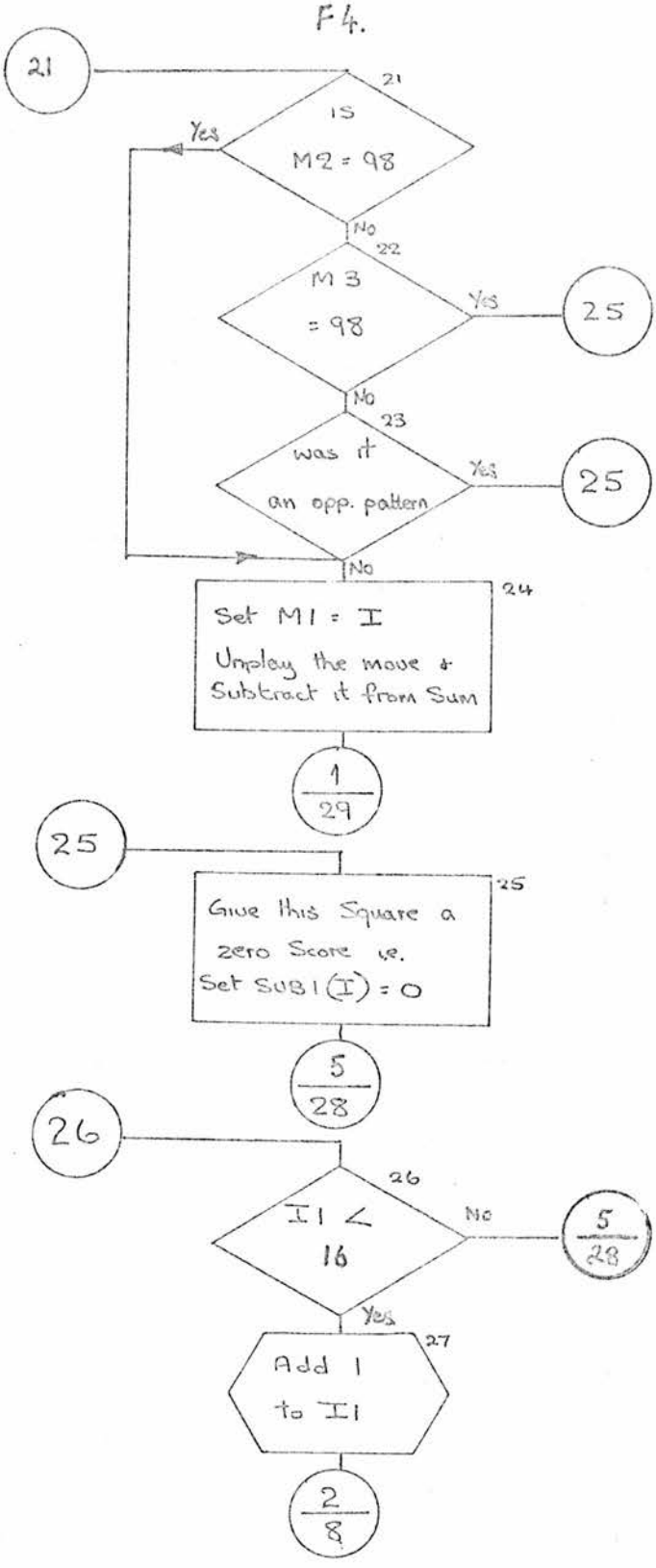
for finding a pattern of 3 in a row

Is there another pattern

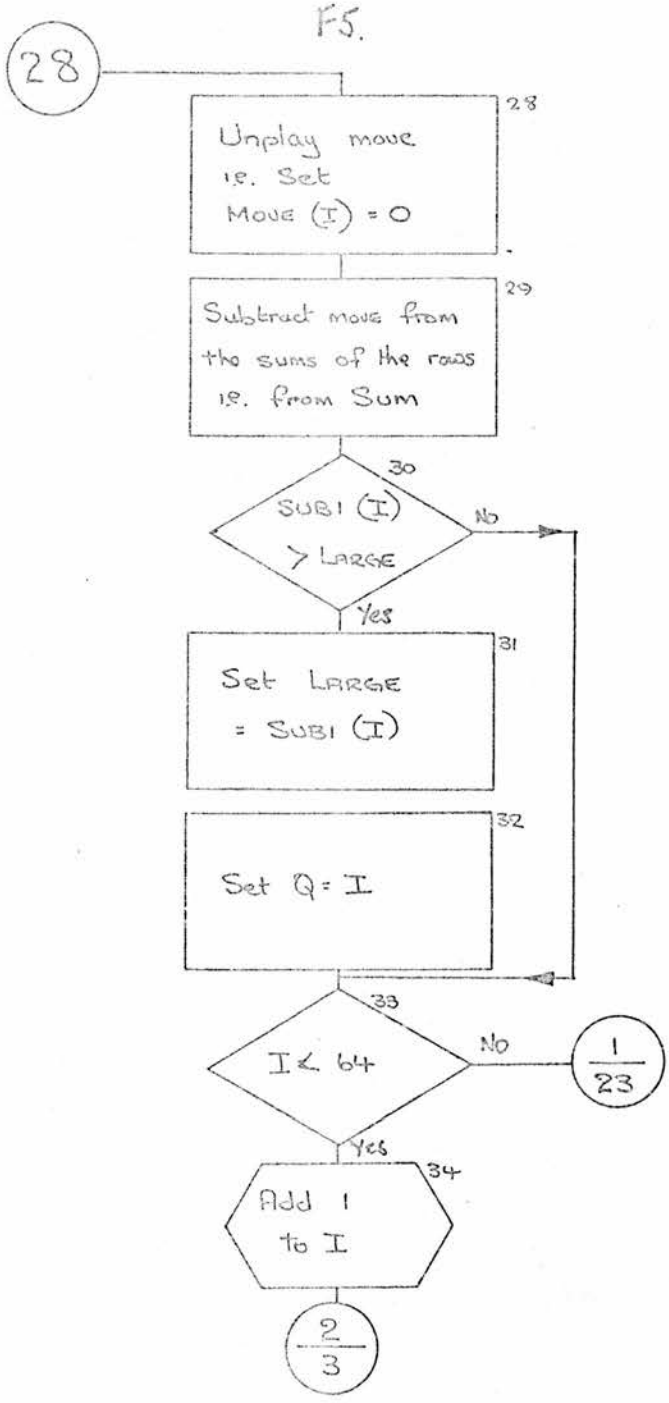


Is the M1 square of the pattern the same as the M1 square of the last pattern. If so it does not score.

If more than 1 pattern is found the 1st time search is called, an opponent move is played in the 'key' square (the M1 Square) & search is called to re-examine the situation.
M4 Set to TRUE because it is only necessary to know if any pattern exists



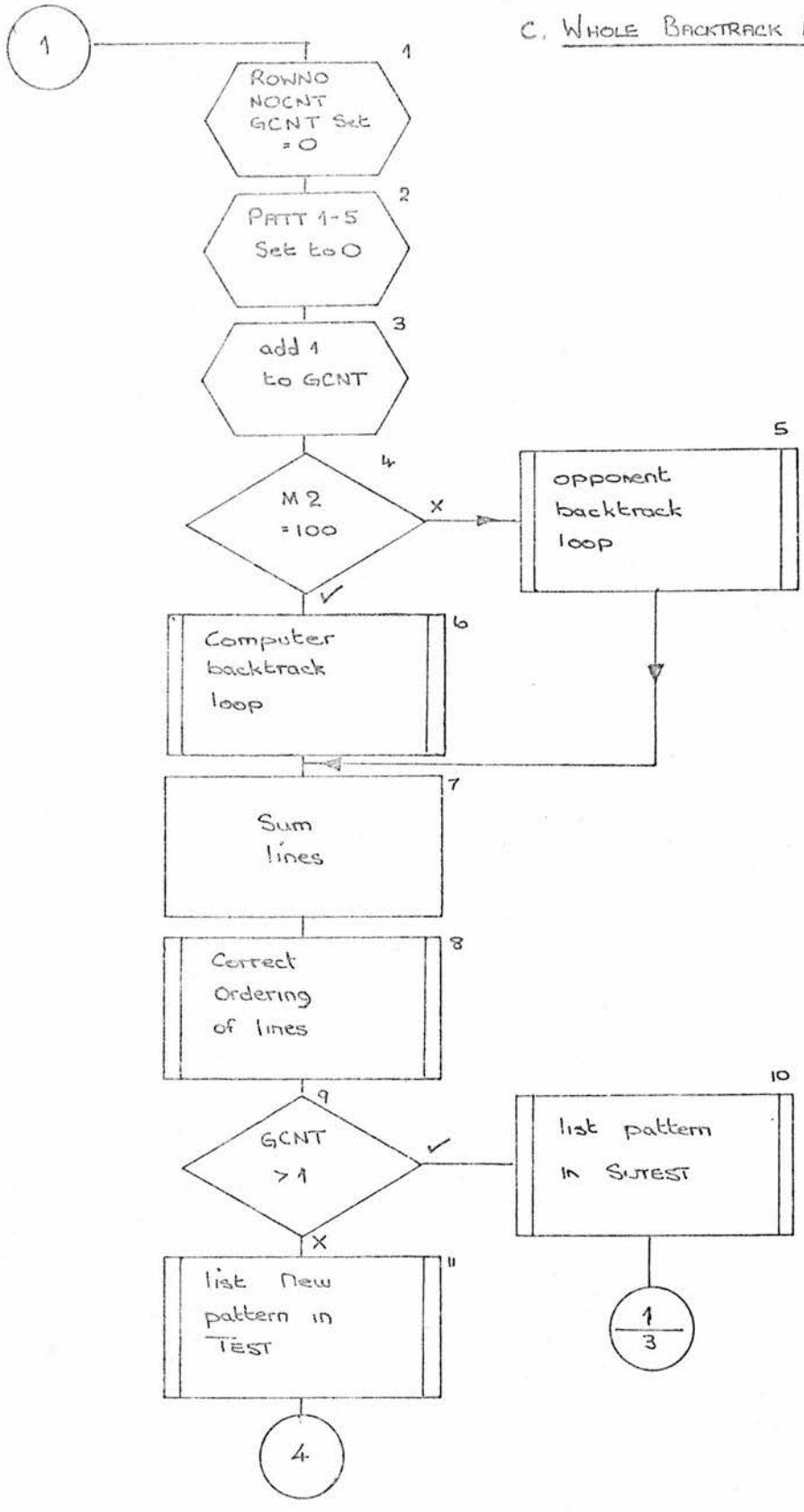
If M1 = 0, Search found no pattern. If M2 = 98, it found 3 in a row for the computer. If M3 = 98, it found 3 in a row for the opponent, in which case the square is given no score at all.

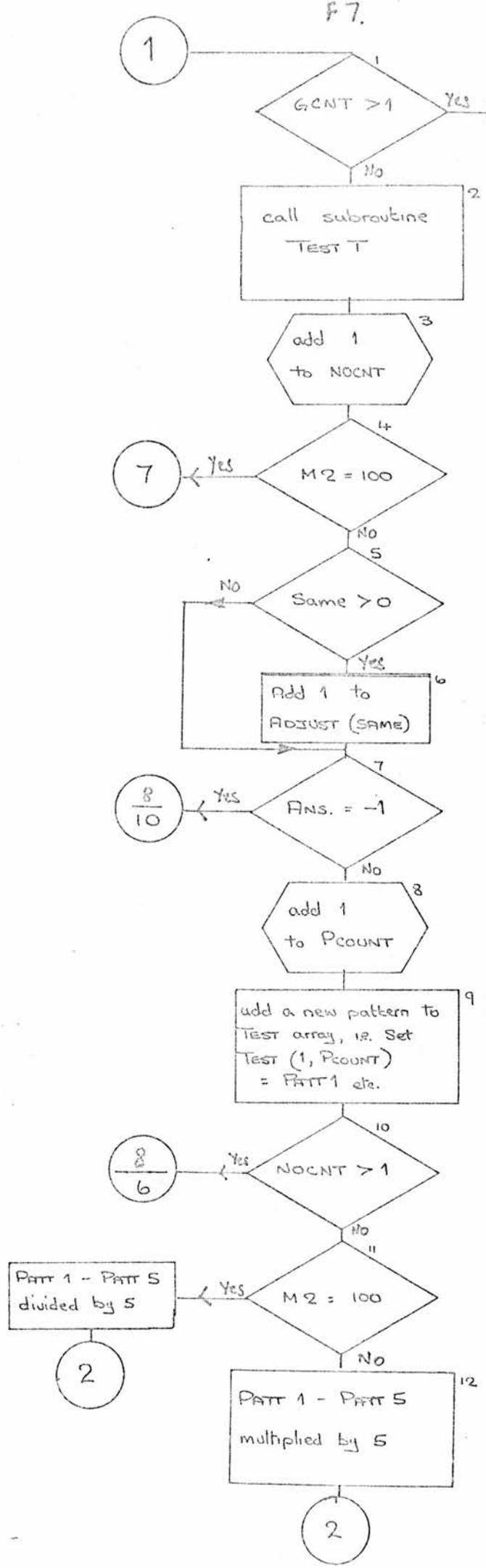


best Score
Stored in LARGE

number of best
Scoring Square
stored in Q.

C. WHOLE BACKTRACK ANALYSIS





D. Listing of New Patterns in Test

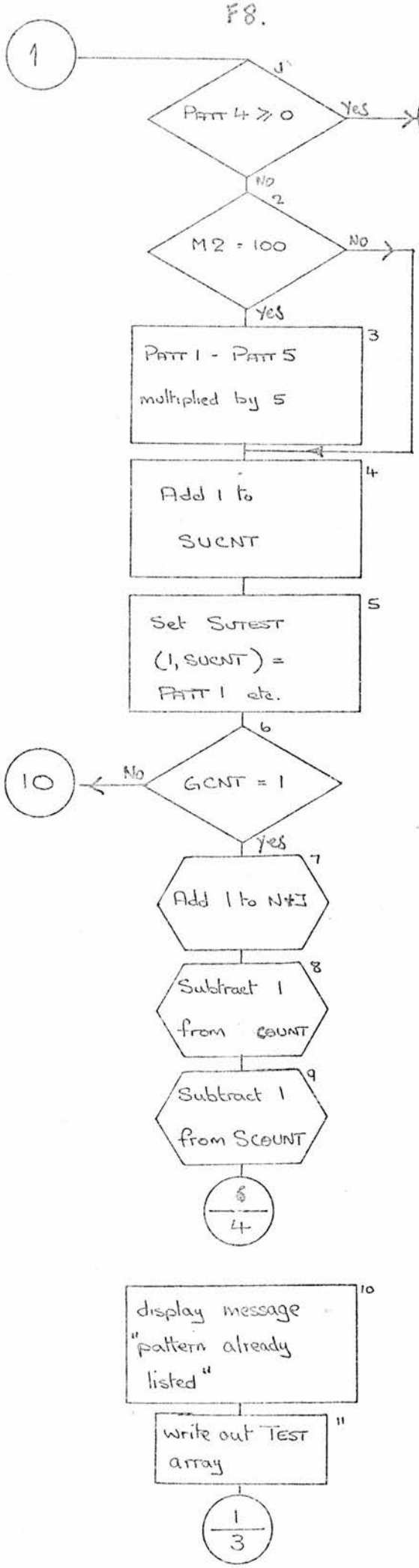
If GCNT is greater than 1, the program has been twice through the backtracking loop and the pattern is ready to be listed in SUBTEST

NOCNT is used as a switch to ensure a new pattern is listed once in computer and once in opponent terms.

ANS will be -1 if the pattern is already listed. It is set in Subroutine TEST T.

If the opponent won the game, the new pattern is converted to a computer pattern and vice-versa Both forms of the pattern are listed in TEST.

E. Listing of New Pattern in Sutest



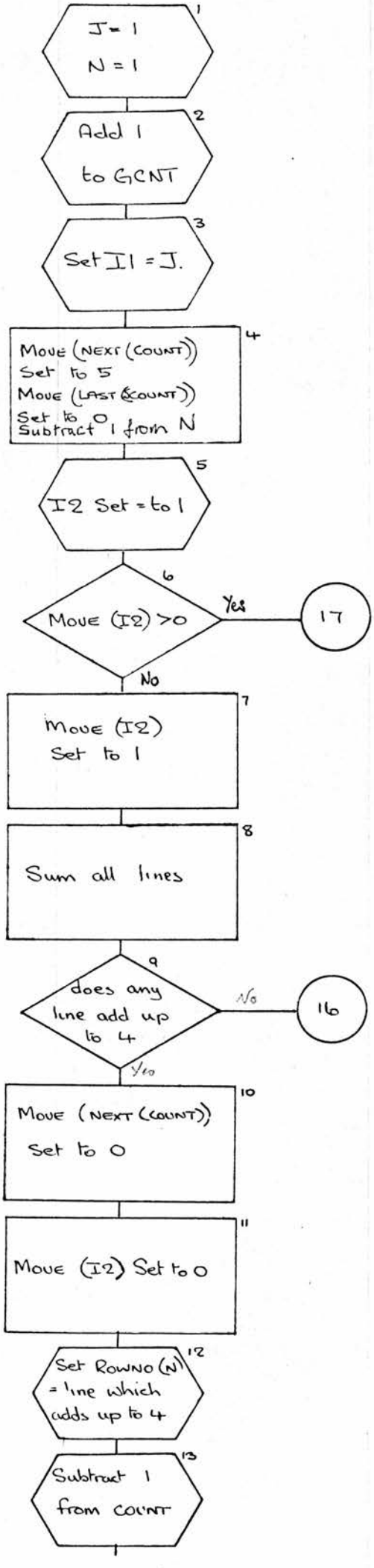
Sutest only lists patterns involving less than 3 lines.

Convert pattern to computer terms.

Sutest only lists patterns in computer terms.

New pattern added to Sutest.

F. Opponent Backtrack Loop



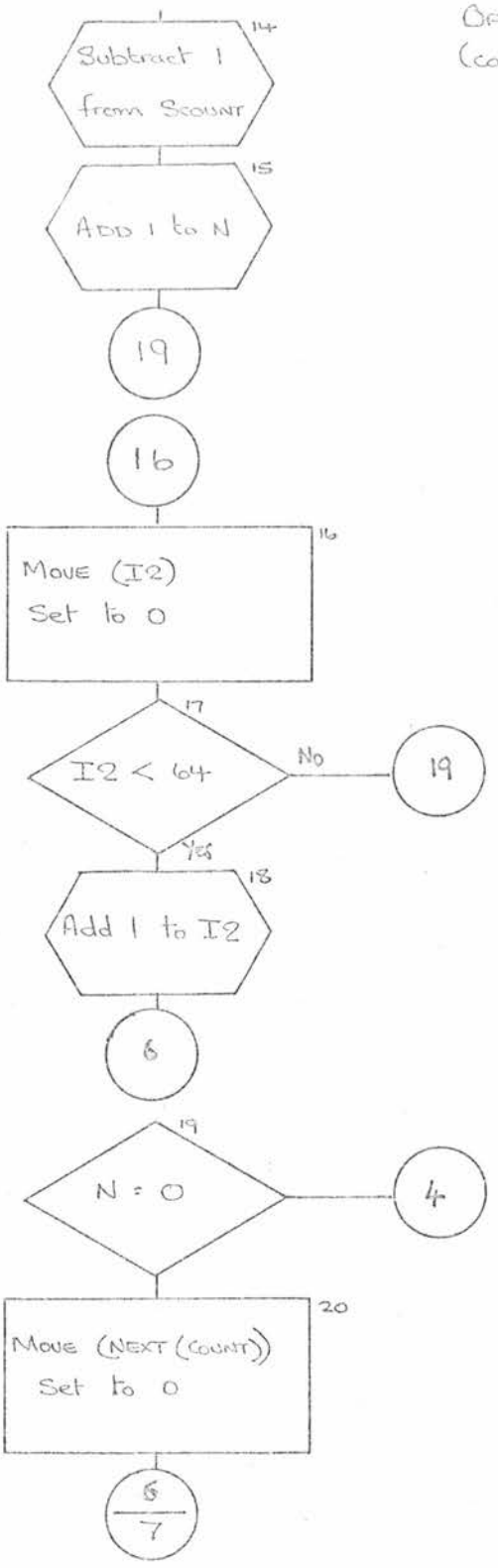
Computer move played where opponent played his last move. Computer's last move unplayed.

An opponent move is then played. in each remaining blank square to see if opponent could still win in 1 move.

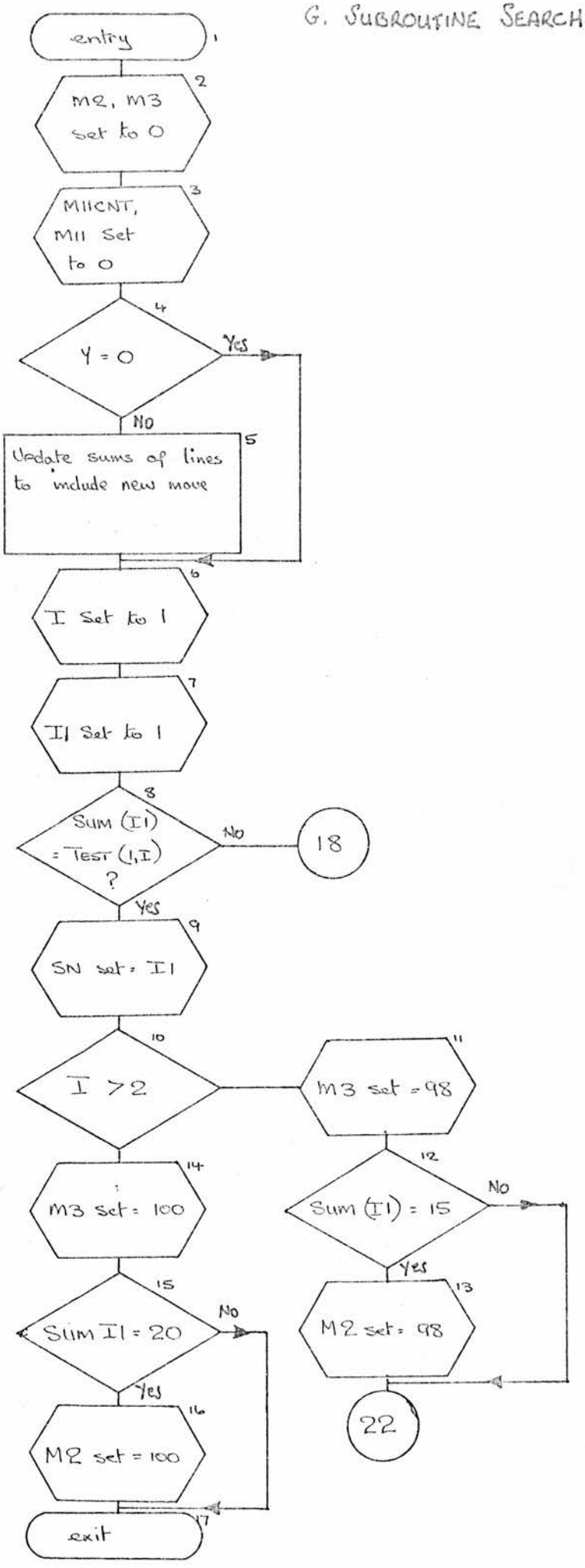
If so, unplay next pair of moves (1 computer and 1 opponent) and repeat process.

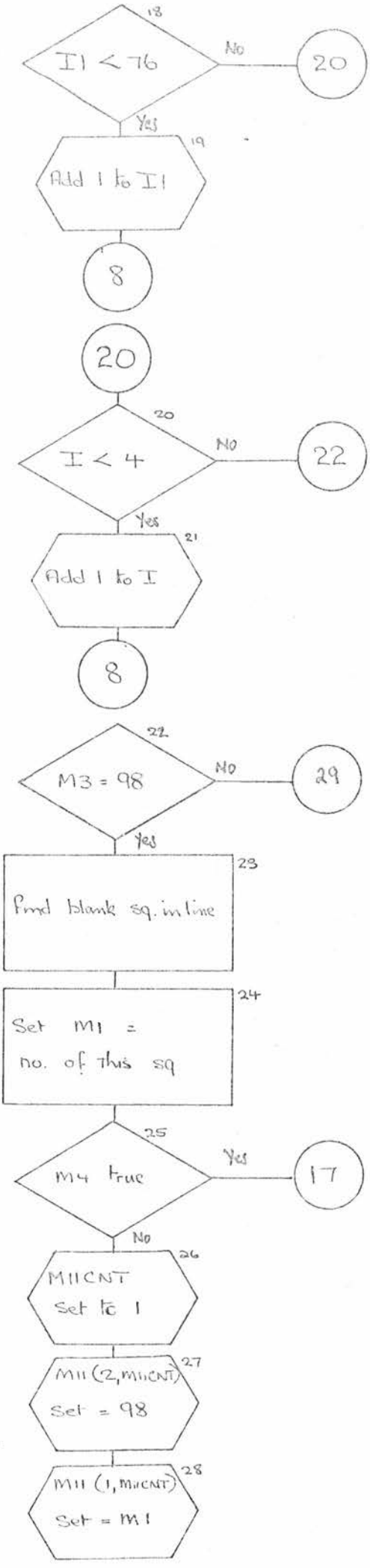
list line which adds up to 4 in ROWNO.

OPPONENT BACKTRACK LOOP (CONT.)



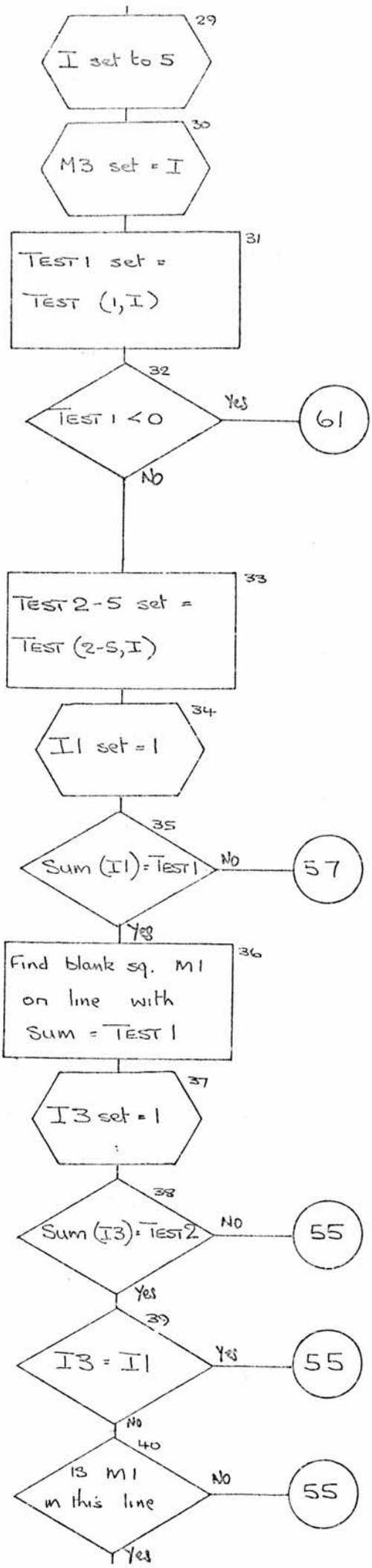
G. SUBROUTINE SEARCH

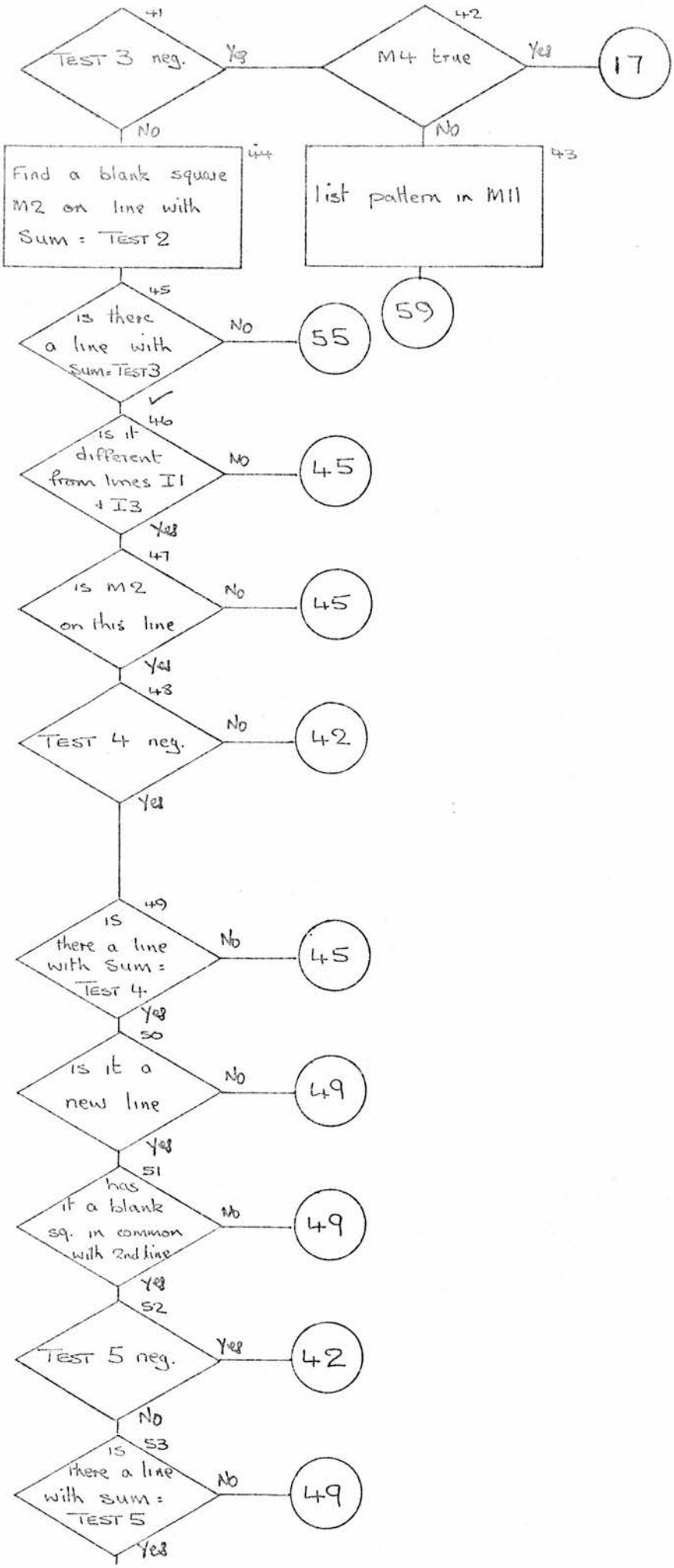




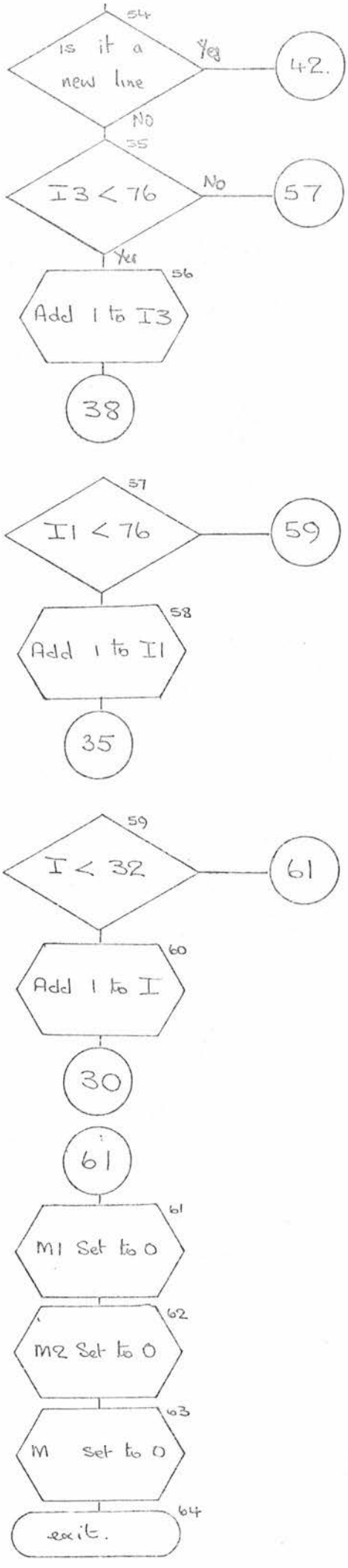
M4 will not be true if search has been called during a look ahead. If so any pattern found will be listed in M11 & program continues to look for more patterns

F13


29
I set to 5
30
M3 set = I
31
TEST 1 set =
TEST (1,I)
32
TEST 1 < 0
Yes
61
No
33
TEST 2-5 set =
TEST (2-5,I)
34
I1 set = 1
35
Sum (I1) = TEST 1
NO
57
Yes
36
Find blank sq. M1
on line with
Sum = TEST 1
37
I3 set = 1
38
Sum (I3) = TEST 2
NO
55
Yes
39
I3 = I1
Yes
55
No
40
I3 M1
in this line
NO
55
Yes


41
TEST 3 neg.
Yes
42
M4 true
Yes
17
No
No
44
Find a blank square M2 on line with Sum = TEST 2
43
list pattern in M11
59
45
is there a line with Sum = TEST 3
No
55
46
is it different from lines I1 & I3
No
45
Yes
47
is M2 on this line
No
45
Yes
48
TEST 4 neg.
No
42
Yes
49
IS there a line with Sum = TEST 4.
No
45
Yes
50
is it a new line
No
49
Yes
51
has it a blank sq. in common with 2nd line
No
49
Yes
52
TEST 5 neg.
Yes
42
No
53
IS there a line with sum = TEST 5
No
49
Yes

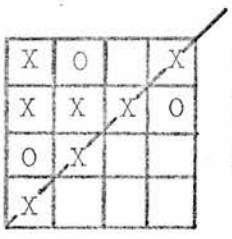
F15


54
Is it a new line
Yes
42.
No
55
I3 < 76
No
57
Yes
56
Add 1 to I3
38
57
I1 < 76
59
58
Add 1 to I1
35
59
I < 32
61
60
Add 1 to I
30
61
61
M1 Set to 0
62
M2 Set to 0
63
M Set to 0
64
exit.

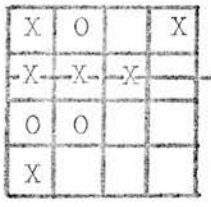
VII COMMENTS ON PLAY AND SUGGESTED IMPROVEMENTS

The standard of play could be considerably improved if the backtrack analysis were altered so that the criterion of when to stop the backing was to call SEARCH until it no longer recognised a pattern, instead of looking to see if either side could make up four in a row in one move. This (the former) is the criterion used by Elcock and Murray, and although it has certain disadvantages it is potentially a much stronger method for a learning program than the one we use. If it were implemented the backtracking would be carried out in the normal way, unplaying one move at a time, only each time it unplayed a pair of moves it would call SEARCH. If SEARCH recognised a pattern in this board position another pair of moves would be unplayed until a board position was reached where SEARCH recognised no pattern. A generalised description of this board

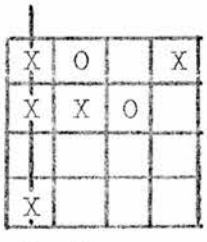
position would then be listed in TEST on the assumption that this position was a necessary and sufficient step in the formation of the board position immediately succeeding it, which the program did recognise. The description of the new pattern would be formed by taking note (in ROWNO) of the lines involved in the patterns seen by SEARCH at each stage of the backtracking, e.g.,

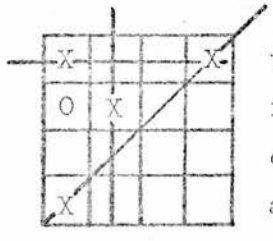
(i) 

the indicated line will go into SN

(ii) 

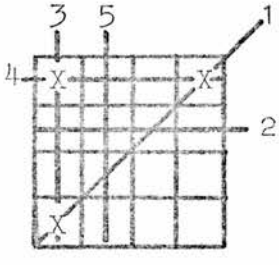
the rest of the lines with arrows will be stored in ROWNO

(iii) 

(iv) 

this pattern involves 3 lines, one of which is already listed

Figure 33

(v) 

this is the board position which will be analysed with the lines adding up to 2, 0, 2, 2, 0, which is the exact order they would be listed in TEST

The drawback to this is that it does not capture the essential feature of the situation, which is that even if the opponent stops one pattern by playing where he does, another pattern can still be made up. As it stands, SEARCH would not even recognise this pattern because it would look for a 4th line of 2 with a blank square in common with the 2nd line and in fact this is not the case.

The ordering part of the program could be altered to adapt to this by dealing with all five lines, not just the first three. It would find that the other line adding up to 0 in fact has a blank square in common

with three of the other lines involved, as does the first line adding up to zero. If the two lines adding up to zero, i.e., the two with most intersecting squares, were put in the 2nd and 3rd positions and a stipulation were made that the 5th line must have a blank square in common with the 3rd, this would go some way towards fitting the bill, and would not throw out any of the other patterns. The pattern would then go into TEST as 2,0,0,2,2. The procedure would then be to find a row of 2 with a blank square in common with a row of 0 which has a blank square in common with another row of zero; find a row of 2 which has a blank square in common with the 2nd row, and another row of 2 which has a blank square in common with the 3rd row.

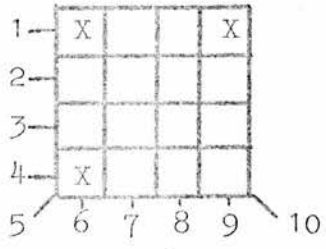


Figure 34

One could even specify that the last row had to have a blank square in common with the 2nd row of 2. However, although this would describe the situation it would not be specific enough to avoid confusion; e.g., the sequence of lines 6,2,8,1,5 would fit the description as well as the sequence 6,2,7,5,1 which it was meant to describe. Nor, as the program stands, would SEARCH indicate the right square to play in. With this pattern the first 'key' square in fact is the square on which lines 2 and 7 intersect. However as the pattern is listed above (2,0,0,2,2), SEARCH would indicate that the 'key' square was where the first two lines intersect, i.e., where lines 6 and 2 intersect. In fact continuing with this line of approach seemed more trouble than it was worth, and seemed to require major adjustments to the program before it could be made to work.

In any case the method of generating patterns which is actually used ensures that when a pattern is found it always means that when M1 is played, a forcing game will begin, i.e., whoever has made the pattern will start making up threes, forcing his opponent to defend until he makes up a winning line of 4. Unless in defending, the opponent accidentally makes up a three for himself, he is bound to lose once the 'key' square in a pattern has been played in by the attacker. This would not be the case if the alternative method of generating patterns discussed above were implemented.

The selectivity of the look ahead procedure means that some of the predictions made during the look ahead are not always true. It assumes that the opponent will play in a certain square which is not necessarily the case, and since the evaluations made by the computer are based on this assumption, if the opponent plays elsewhere, it throws out the computer's calculations.

For example:-

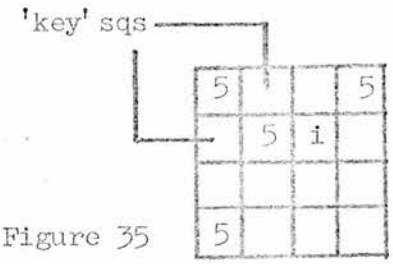


Figure 35

If the computer has this pattern it will assume that the opponent will play in one of the 'key' squares, and whichever he plays in will assume that it can still make up a forcing pattern by playing in the other. In fact if the look ahead procedure were less selective it would find that if the opponent played in sq. (i) he could stop both patterns at once. As it is the computer thinks it is bound to win.

VIII Appendix

A few games played by the computer. The squares are numbered as in Figure 3. The patterns referred to as 1, 2, 3 are described on page 17 v. figures 9, 10, 11, 12.

Game I The computer is playing both sides in this game.

Red starts :-

	RED	WHITE
1.	39	1
2.	13	26
3.	4	42
4.	23	7
5.	16	27
6.	43	15
7.	48	8
8.	47	35
9.	64	32
10.	30 (wins)	

1. 39 (red), 1 (white)

The 1st moves on both sides are 'randomly' chosen from amongst the 16 strong squares.

2. 13 (red), 26 (white)

At this stage no pattern can be formed by Red in 1 or even 2 moves, so the Counter Scoring Method (v.P.30) of selection is used to choose Red's next move. Square 13 is a strong square in the same line as Red's 1st move in square 39.

White cannot form a pattern either, but decides to play defensively. It chooses a square which is in the same line as Red's 2 counters and at the same time is a strong square. This ensures that he makes no

sacrifice by defending - the situation is not dangerous enough to demand this - square 26 is more or less as good a square as any for White to play in.

3. 4 (red), 42 (white)

Red has still not played enough moves to make up one of the patterns listed in TEST, in 1 move. However he can make up 1 of the patterns listed in SUTEST, which he does by playing in square 4 - this pattern can be converted into Pattern 1 (2 converging twos) in one move, using either square 23 or square 55. White realises this board position is potentially more dangerous than just the forming of Pattern 1. Red can make up Pattern 1 by playing in either 23 or 55, or Pattern 2 by playing in square 45. These patterns can be stopped in one move. However if Red plays in square 42, two separate realisations of Pattern 2 are formed, with separate 'key' squares - 7 and 45 respectively. This means that White could only stop one of these patterns. The same situation would be created if Red played in square 61. White stops this happening by playing in square 42. Although this is a defensive move, White loses nothing by it as it is a strong square. It is only when he is forced to defend in weak squares while Red monopolises the strong squares that White starts to lose strength. This in fact is what happens after Red's next attack.

4. 23 (red), 7 (white)

Red can now make up a 3-in-a-row by playing in squares 7 or 10, or form Pattern 1 by playing in squares 23 or 55. Since 23 is also a strong square this is what he does.

This pattern is easily stopped by White playing in square 7 but Red has gained a slight advantage in that he is now in occupation of 4 strong squares to White's 3.

5. 16 (red), 27 (white)

Red is now unable to form any TEST pattern in 1 move, so he uses the SUGOAL list to choose his next move in square 16. By playing in square 27 next move he will be able to make up Pattern 1.

However he is forestalled by White. White realises that again this is a potentially dangerous position. If Red plays in square 27 next move he will not only form Pattern 1 on square 15, but will simultaneously form Pattern 2 on square 8 (then square 38). White will not be able to stop both.

Apart from this, Red can form two patterns by playing in square 19 or square 38 or square 49 and can form various other single patterns. White plays in square 27, a strong square, and by so doing both stops a dangerous position and puts himself in a very strong position.

6. 43 (red), 15 (white)

Red is forced to worry about defense for the first time. White can form a double pattern (i.e., 2 patterns simultaneously which have different 'key' squares) by playing in square 9; i.e., by playing in square 9 forms Pattern 2 on square 25 (then square 57) and another realisation of the same pattern on square 10 (then square 12). White can also form a double pattern by playing in square 43 - pattern 3 on both square 25 and square 10. Play in square 60 will also give White a double pattern - Pattern 2 on square 28 and on square 58. Also various single patterns. Red must defend and does so by playing in square 43, which in fact is a strong square anyway.

As it happens this makes up a pattern for Red - this is something that Red realises, but the reason for playing there was purely defensive - at one stage the program did take this sort of thing into account when choosing a square - i.e., a square's value both as an attacking and a defensive move, but this played a very weak game. (See chapter on Scoring).

The pattern formed is an instance of Pattern 3 on square 15. White is forced to play there.

7. 48 (red), 8 (white)

Despite this White is still in quite a strong position. Playing in square 9 would still form a double pattern - Pattern 2 on squares 25 and 11. Playing in square 10 would make up 3-in-a-row and form Pattern 2 on square 11. Playing in square 12 makes up 3-in-a-row and forms Pattern 2 on square 11. Playing in square 60 makes up a double pattern - Pattern 2 on square 28 and on square 58. There are also various single patterns.

Red would be forced to defend if he could not form a double pattern of his own. However if he plays on square 48 he forms one instance of Pattern 2 on square 8 (then square 38) and another instance of Pattern 2 on square 47. White will only be able to stop one of these patterns. So Red plays on 48.

White defends by playing in square 8 to stop one of the patterns.

8. 47 (red), 35 (white)

Red plays in square 47, making up a forcing pattern.

9. 64 (red), 32 (white)

Red plays in square 64 making up 2 separate rows of three.

White stops one of them by playing in square 32.

10. 30 (red)

Red completes the other (makes it up to a row of 4) by playing in square 30, and so wins.

Game 2 Computer playing both sides.

White starts :-

	WHITE	RED
1.	23	13
2.	4	42
3.	1	22
4.	43	3
5.	16	27
6.	38	8
7.	49	33
8.	26	52
9.	2	6
10.	50	14
11.	34	18
12.	19 (wins)	

1. 23 (white), 13 (red)

The 1st move on both sides is picked 'randomly'.

2. 4 (white), 42 (red)

White's 2nd move is selected by the Counter Scoring Method, as there are no patterns.

Red's 2nd move is vaguely defensive as was White's 2nd move in the last game. However square 42 is also a strong square.

3. 1 (white), 22 (red)

White's third move is selected by the SUGOAL subroutine which applies the list of patterns in SUTEST to the current board position to see if any of the patterns on the list can be made up in one move, preferably on a 'strong square'. The pattern is one that can be converted into Pattern 1 in one move, by playing in square 43 or square 63.

Red realises this and also that Pattern 2 can be formed by playing in square 24. However Red also realises that by playing in square 22, White can form simultaneously two realisations of Pattern 2 with different 'key' squares, viz. square 3 and square 24. If this happens Red will only be able to stop one of the patterns - or at least that is the conclusion the computer comes to. So Red plays in square 22.

4. 43 (white), 3 (red)

White then goes on to form the pattern started in the last move by playing in square 43.

This forces Red to stop this pattern by playing in square 3.

5. 16 (white), 27 (red)

White can make up no patterns in one move so it uses SUGOAL to choose its next move which is square 16. By playing there he starts building up Pattern 1 (with square 38 in mind as its next move).

Red realises that White can now make up several different patterns. The most dangerous squares are 27, 33, 38 and 49. If White plays in square 27 he simultaneously forms 2 patterns, Pattern 1 converging on square 11, and pattern 2 with 'key' square as 8, followed by 38.

Play in square 33 forms Pattern 2 on square 8 (then 38) and Pattern 2 on square 49 (then 38).

Play in square 38 forms Pattern 1 on square 8 and Pattern 2 on square 11 (then square 27).

Play in square 49 forms Pattern 2 on square 8 (then 38) and Pattern 2 on square 33 (then 38).

In fact Red plays in square 27 as this is the first of the dangerous squares - they are all judged to be equally dangerous.

6. 38 (white), 8 (red)

White can now make up :-

Pattern 2 by playing in sq. 5

Pattern 2 by playing in sq. 7

Pattern 2 by playing in sq. 9 (this proposed move is immediately rejected as it would force Red to play in sq. 12 which would make up 3-in-a-row for Red.)

The same thing would happen if White played in sq. 10.

Pattern 2 by playing in sq. 33

" 1 " " 38

" 3 " " 39

" 2 " " 48

" 1 " " 53

" 3 " " 55

In no case are two patterns simultaneously formed, so White chooses square 38 which forms a pattern and is a 'strong' square.

This forces Red to play in square 8. (It is worth noticing that although all the patterns formed by White have been easily stopped, it usually happens that White choses 'strong' squares to build up its patterns and forces Red to play in weak squares, so White is steadily taking possession of all the strong squares on the board and building up a very strong general position for himself.)

It can be seen from play so far that there are always a considerable list of patterns which could be formed in the next move. In the remainder of the description of this game, only the important possible patterns are mentioned.

7. 49 (white), 33 (red)

White now plays in square 49 which makes up Pattern 1 on square 33. Red plays in square 33.

8. 26 (white), 52 (red)

White can now make up Pattern 3 by playing in square 52, but before this pattern could be made up to 4 in a row the opponent could make up 3 in a row and force White to defend; it is therefore abandoned. SUGOAL is used to select the next move which is square 26. This starts building up Pattern 1 on square 20.

Red realises that White is now in a very dangerous position and can form 2 patterns simultaneously by playing on square 2 or 18, 20, 28, 35, 40, 50, 52. He defends by playing in square 52 which scores more than the other squares as it is also a 'strong' square.

9. 2 (white), 6 (red)

White now sees the pattern which eventually leads him to a win four moves later. By playing in square 2 it will simultaneously form 2 patterns:- Pattern 2 on square 6 and Pattern 2 on square 50. Red will only be able to stop 1 of these patterns. So White plays on square 2. Red, recognising one realisation of Pattern 2 stops it by playing in sq. 6.

10. 50 (white), 14 (red)

White then plays in square 50 - the key square in the 2nd realisation of Pattern 2 - and so makes up 3 in a row forcing Red on to square 14.

11. 34 (white), 18 (red)

Playing in square 34 White then makes up 2 separate lines of 3 and Red can only stop 1, which he does by playing in square 18.

12. 19 (White)

White completes a line of 4 by playing in square 19.

Comment

Having the computer playing both sides has a certain weakness in that the computer will always play in the 'key' square of a pattern whereas sometimes it could stop 2 patterns by playing elsewhere. This is not discovered by the computer because of the selective nature of the look ahead procedure. If it checked every square it would find out if there was one which would stop 2 patterns at once.

Game 3 This game was played against a human opponent. White (the human) started, and the whole game was as follows :-

WHITE	RED
22	49
27	1
39	38
23	26
42	13
5	56
29	21
32	17
33	31
6	7
16	12
61	4
10	2
3	50
14	34
18	19 (wins)
8	

White's last move, 8, is played because the subroutine which detects a win, or any other pattern, is only called after an opponent's move. This saves a great deal of time although it does create the anomalous situation that the opponent has to play a move after a line of 4 has been made up by the computer.

Game 4 Computer playing both sides again.

WHITE	RED
42	43
4	23
1	22
13	16
6	2
5	9
7	8
10 (wins)	

Game 5

RED	WHITE
4	1
13	16
23	26
39	7
52	36
29	61
22	30
38	21
41	56
64	43
19	37
54	6
11	5
8	53 (wins)

IX REFERENCES

BEL 68 Bell A.G. (1968) "Kalah on Atlas". Machine Intelligence III, (ed. Michie D.)

BER 58a Bernstein A. et al. (1958) "A Chess Playing Program for the IBM 704 Computer". Proceedings of the Western Joint Computer Conference.

BER 58b Bernstein A. and Roberts (1958) "Computer versus Chess Player". Scientific American, June.

DOR 66 Doran J.E. and Michie D. (1966) "Experiments with the Graph Traverser Program". Proc. R. Soc. (A), 294, 235 - 259.

DOR 67 Doran J.E. (1967) "Graph Traverser". Machine Intelligence I, (ed. Collins N.L. and Michie D.)

ELC 68 Elcock E.W. (1968) "Descriptions". Machine Intelligence III, (ed. Michie D.)

GOO 68 Good I.J. (1968) "A Five Year Plan for Automatic Chess". Machine Intelligence II, (ed. Dale E. and Michie D.)

LOU 67 Louden R.K. (1967) Programming the IBM 1130 and 1800.

MAY 61 Maynard Smith J. and Michie D. (1961) "Machines that Play Games", New Scientist, Vol. 12, 367 - 369.

MIC 66 Michie D. (1966) "Game Playing and Game Learning Automata". Advances in Programming and Non-Numerical Computation, Chap. 8, (ed. Fox L.)

MIC 67 Michie D. (1967) "Strategy-Building with the Graph Traverser". Machine Intelligence I (ed. Collins N.L. and Michie D.)

MIC 70 Michie D. (1970) "The Intelligent Machine". Science Journal, Sept.

MUR 67 Murray A.M. and Elcock E.W. (1967) "Experiments with a Learning Component in a Go-Moku Playing Program". Machine Intelligence I, (ed. Collins N.L. and Michie D.)

MUR 68 Murray A.M and Elcock E.W. (1968) "Automatic Description and Recognition of Board Patterns in Go-Moku". Machine Intelligence II, (ed. Dale E. and Michie D.)

NEW 58 Newall A., Shaw J.C. and Simon H.A. (1958) "Chess Playing Programs and the Problem of Complexity". Computers and Thought, (ed. Feigenbaum E.A. and Feldman J.)

SAM 59 Samuel A.L. (1959) "Some Studies in Machine Learning using the Game of Checkers". Computers and Thought, (ed. Feigenbaum E.A. and Feldman J.)

SAM 60 Samuel A.L. (1960) "Programming Computers to Play Games". Advances in Computers, Vol. 1, (ed. Alt F.)

SAM 67 Samuel A.L. (1967) "Some Studies in Machine Learning using the Game of Checkers, II - Recent Progress". IBM Journal of Research and Development, Nov. pp 601 - 617.

SCO 69 Scott J.J. (1969) "A Chess Playing Program". Machine Intelligence IV, (ed. Meltzer B. and Michie D.)

SHAN 50a Shannon C.E. (1950) "Automatic Chess Player". Scientific American, Feb.

SHAN 50b Shannon C.E. (1950) "Programming a Digital Computer for Playing Chess". Philosophy Magazine, March.

TUR 50 Turing A.M. (1950) "Computing Machinery and Intelligence". Mind, Oct.

TUR 53 Turing A.M. (1953) "Digital Computers Applied to Games". Faster than Thought, (ed. Bowden B.V.)

TUR 58 Turing A.M. (1958) "Can a Machine Think". Computers and Thought, (ed. Feigenbaum E.A. and Feldman J.)

ACKNOWLEDGEMENTS

I should like to record my gratitude to Professor Cole and Mr. A.J.T. Davie for their help and encouragement in all aspects of this work.

I am indebted to the Science Research Council for the award of a Research Scholarship 1969-1970.

My thanks are also due to Dr. Robert Erskine and his staff for their invaluable assistance and to Mrs. Susan Weaver who typed this thesis.

Considerable use was made of a program written by R.K. Louden to play noughts and crosses.