PANEL ON COMPUTER GAME PLAYING

Chaired by

Hanr> Berliner
Carnegie-Mellon University
Pittburgh, Pa. 15213

Objectives

Historically, programs that use big search have been distinctly superior to those that rely more on Knowledge than search. We wish to discuss here why this is, and what can be done to produce more successful programs in the AI mold, i.e. using Knowledge to guide actions, rather than discovering useful actions as the result of search.

Participants and Background

Hans Berliner Cornegie Mellon Univ. Pittsburgh, Pa.	CAPS-II, BKG
Richard Greenblatt MIT Cambridge, Mass.	Greenblatt Chess Program, CHEOPS
Jacques Pitrat C.N.R.S. Paris, France	Planning in Chess
Arthur Samuel Stanford University Palo Alto, Calif.	Checkers, Signature Tables
David Slate Northuestern Univ. Evanston, Illnois	CHESS X.X

SEARCH AND KNOWLEDGE[1]

Hans Berliner
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Abstract

We discuss some important properties of various search techniques, and some properties that Knowledge must have in order to support adequate game playing behavior.

I. Some Points on the Continuum

It seems that everything in game-playing (and possibly in all of AI) ultimately comes down to Knowledge and search. This is really another case of a space/time tradeoff. All state-space problems can be solved by searching, if there were enough time to do a complete search of the problem space. Also all problems can be solved by enough Knowledge, by merely having all pertinent information available for every node in the problem space; provided it is available and there is enough space to store it. TasKs that can be solved by either of these methods are considered not to be interesting.

To search without Knowledge when a complete search is not possible does not maKe sense. To apply partial Knowledge without search has not been very successful in games such as chess. Turing achieved only mediocre play with his (hand-simulated) chess program Turochamp [Tur53], and other attempts since then have not realized much improvement.

Thus interesting problems appear to require a combination of search and Knowledge. However, there are many possible methods, both for controlling the search and for organizing the Knowledge. Knowledge must be able to detect interim advantages. In non-terminal states such interim advantages can be considered indicators of how the game may go in the future.

A search with such Knowledge will be able to find leaf nodes at the limit of its searching capability which maximize its understanding of the universe. However, such a search only understands "ends"; it does not understand "means". Thus if it reaches a leaf node in a search to an arbitrary depth, it does not understand how it achieved its present success (or lacK of it). This creates additional problems when a greater success can be achieved in another branch by the same means, but at a slightly greater depth. This is the Horizon Effect [Ber73].

II. Some Tractable Search Techniques

It is convenient here to distinguish goal-directed searches which develop and prune nodes based on properties of a node (rather than the terminal value of its branch), and "mindless" techniques which search all of a predefined search space to find the optimal path, basing decisions solely on the value of terminal nodes. To date the latter techniques have been eminently more successful in game-playing. This is no doubt due to the tremendous amount of Knowledge that is required

[1] This worK was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (contract F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

to adequately guide a goal-directed search in a large domain. We now discuss advantages and limitations of each approach.

A. Techniques that search a uniform space

It is now rather clear that the old controversy of depth-first vs. breadth-first search (if it was ever really controversial) has been resolved in favor of a new technique called depth-first search with iterative deepening. This technique was discovered independently by Slate & Atkin [Sla77] of the Northwestern University chess group, and James Gillogly, then at Carnegie-Mellon University. The technique searches depth-first fashion to depth N, and reports its results. The search is then itereated to depth N+I, using information gained in the earlier searches. This information allows the guiding of the search so as to produce a better ordering of alternatives, which makes the alpha-beta mechanism more effective. This makes an iterated depth-n search cheaper than a depth-n search without the signposts (so to speak). These results indicate that search to one additional ply maintains much of the relative value structure of the branches, at least in a domain such as chess and almost certainly for any domain having structure. This technique is also very effective in allocating time to a search as the time for one additinal iteration can be well estimated from the previously iterations. Thus this technique which combines the best features of depth-first alpha-beta (efficiency) and breadth-first (uniform coverage) appears vastly superior to any other technique that searches a uniform space.

B. Techniques that Search Selectively

Selective search techniques *are* generally classified as best-first searches. It is very probably that the reason that these techniques have not been very successful is that at every node it is necessary to apply knowledge to ciecide how to continue the search. The decision can range from stopping, to jumping to another branch of the tree, to selectively expanding the current node. The knowledge that could be used for such purposes is extensive, and it appears quite likely that no one has thus far been able to ciccumulate a data base of sufficient size and detail in any complex domain. However, it appears likely that there is another reason too; one-pass evaluation which we take up in section V. Of the present techniques, Harris' bandwidth .search (an adaptation of the A* procedure) appears very effective [Har74].

I have recently invented a new procedure called the B* tree search procedure which combines the best facets of best-first searching with branch and bound [Ber77]. This procedure requires that an optimistic and pessimistic value be assigned to every node at the time it is generated. When the most optimistic (as seen from the opponent's side) successor of node N is better for the side on move at node N than the previously estimated pessimistic value, then this pessimistic value is adjusted to correspond with the most optimistic value of the opponent. In the same way the optimistic value at node N must be adjusted if the pessimistic value of the most pessimistic successor to node N (as seen by the opponent) is worse than the previously estimated optimistic value. In this way values are backed up from newly expanded nodes. To find the next node to expand, it is only necessary to trace the most optimistic (for both sides) path through the tree and expand that node, although other search strategies are possible. The search terminates when it can be shown that the pessimistic value of a successor node to the root is at least as good a:, the best optimistic value of the remaining successors to the root.

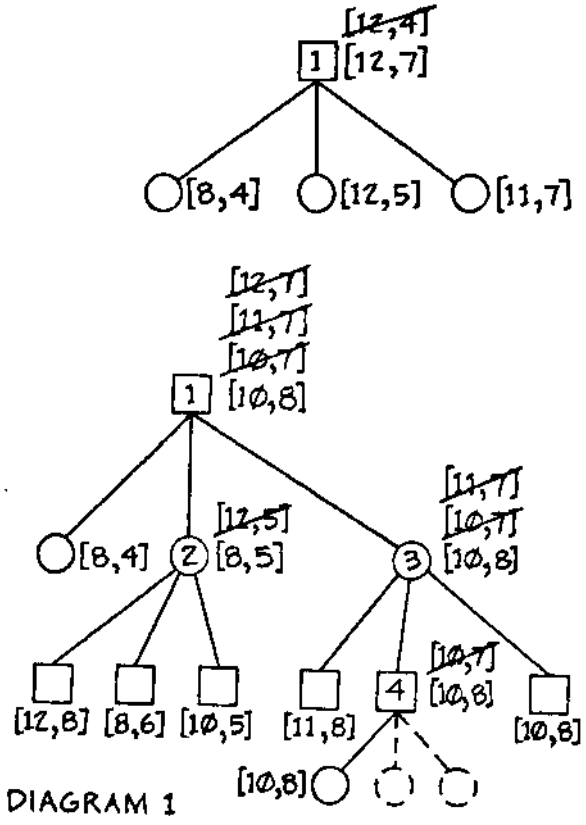

DIAGRAM 1

In Diagram 1, we illustrate a simple search with B*. The upper part shows an initial tree configuration, and the lower shows the tree upon completion of the search. Nodes in boxes have MAX to play; nodes in circles have MIN to play. The numbers inside the node symbols indicate the order in which nodes were expanded. The pair in brackets are the optimistic and pessimistic values at a node (updated by being crossed out). The search terminates because the pessimistic value of the right-most descendant of the root is no worse than the optimistic value of any other descendant.

This search is guaranteed to converge as long as the estimates are consistent (though not necessarily correct). We believe that this algorithm will never expand more nodes than any other procedure having access to the same information. Thus it should be the best of the AND/OR tree searching procedures. It can be trivially demonstrated that in its OR/OR form, it can *never* do worse than the A* [Nil70] procedure which is supposed to be optimal. However, in all these procedures there are serious difficulties in finding reasonable functions for computing the bounding values.

C. Mechanism within the Search

We consider that the actual tree search procedure is strongly subordinate to the ability to reject searching of sub-trees based on the semantics of the search. This type of pruning is supported by mechanisms that collect information (other than terminal values) during the search and bacK this up as the search progresses. Decisions are then made, based on the backed-up information, which allow pruning of large sub-trees.

The first of these mechanisms to appear was the Causality Facility [Ber74] in my chess *program* CAPS-II. This facility collected descriptions from nodes when backing up in a tree search. When it reached a node where the side on move was not satisfied with the backed up value of the search (measured against a global expectation level), that side set in motion the causality facility which examined the backed up description (called the refutation description). Current versions of the causality facility are able to determine whether or not the problem described by the refutation description was caused by the current move at this node. If so, a lemma is posited which describes the partial position (out of the current position) which makes the refutation description possible. Lemmas are also posited about winning moves. Whenever a move is suggested at some future point in the search, the lemma file is first examined to determine if a lemma exists about this move and if the partial board description matches. If so, the result of the move is assumed known and the search can be foregone. A similar system has been described by Adelson-Velsky, et. al. [Ade75].

If the move in question did not cause the current problem, then we know it to have existed when the current node was reached. Thus the backed up refutation description, describes this problem. The causality facility has mechanisms for generating the set of all moves (counter-causal moves) that can do something about the refutation description. This can save a tremendous amount of effort in ad hoc searching to find a solution. Sometimes the counter-causal set is empty so it is known that the problem cannot be solved at this node and the search must back up further. Sometimes a proposed counter-causal move leads to another problem description which is also not caused by the move. Then it is known that there is more than one problem to solve at the current node, and any suggested move must be on the list of counter-causal moves for each such problem, a fact which radically reduces further search.

Another technique was recently demonstrated by Pitrat [Pit77]. He used plans to guide the search. Plans were formulated to satisfy goals of winning at least a certain amount of material. If such a plan failed, the reason for its failure was analyzed, and a plan for overcoming this difficulty was inserted into the appropriate place in the original plan. The description of the plan was pushed down the tree as the search progressed and only moves in accordance with the plan (as modified) were admitted to the search. This method was shown to develop small trees and solve many interesting chess problems.

The essence of techniques of this type is that a potential move is analyzed with respect to a specific purpose. If the move fails, then information is returned which can be used to improve the selectivity of the remaining search. It appears to this writer that these techniques are ripe for application to other areas, e.g. theorem proving, planning techniques..

III. Problems with certain Search Techniques

At times search is a very powerful technique, achieving things that humans may have difficulty in replicating, e.g. Samuel's checker program [Sam63"|, Dendral [Buc69], and CIESS 4.5 [Sla77]. However, certain artifacts appear to be associated with some well known search techniques.

Any piocedure which changes the mode of search (including termination) at an arbitrary depth creates the conditions necessary for the Horizon Effect [Ber73]. Thus searches to fixed depths must reconcile themselves to the possibility that the program will try to push unavoidable consequences over its search horizon, by conceding lesser (but avoidable) ones.

Techniques that search a large uniform space usually try to get the maximum value for each CPU second of evaluation. This is because there are many nodes to be evaluated and a few extra microseconds per node may *add* an intolerable amount of time to the whole search. Therefore evaluation must be lean. If a tree contains two or three reachable good nodes, the path to one of these will be selected depending upon *very* arbitrary factors. 1 his is fine when all these nodes are very favorable. However, in delicate situations it would be highly desirable to isolate the competing nodes and apply additional knowledge to them in an effort to find which is really best.

The minimax alpha-beta procedure is undoubtedly the most efficient of all known search procedures. However, a minimax procedure does very poorly at defending losing positions. In such cases, it will almost always choose the path which postpones the opponent's win the longest. However, this is seldom the best way. In bad to hopeless situations it appears necessary to have an opponent model, no matter how primitive, and select moves based on the hope that the opponent is fallible and will thus not always find the best way. Thus, for instance, it would be emminently reasonable for a program to avoid losing a rook in an otherwise even position, at the cost of the opponent finding a difficult mate in five moves. However, any program using the minimax approach and discovering the opponent's mating possibility will defend against it and leave the rook to its fate. Further, in near equal positions it is sometimes wise to take small risks in the hope of gaining an advantage. This, too, is impossible with the minimax model.

IV. Knowledge Organization

When the search does not go to terminal nodes as defined by the domain, knowledge is needed to evaluate leaf nodes. In goal directed searches knowledge is also needed to make guiding decisions, if any kind of means-ends methods are used.

The results of Samuel[$am63, Sam67] have always been interpreted to mean that a linear polynomial cannot adequately span a large domain such as checkers. However, recently Slate & Atkin [Sla77] have had great success with their chess program CHESS 4.5. This program uses essentially one evaluation function for the whole domain. The coefficients of the terms are designed so that they change very slowly over the domain, and thus provide a sort of rolling landscape in the evaluation space. With this type of design, the program appears to always find something constructive to do. It seldom gets stuck on a hill, since there is always a somewhat higher hill within searching distance from the present one.

Howevei, it is not difficult to show that it is impossible to evaluate all states in a state-space correctly with a single (reasonable) function. In chess, for instance, there are positions where changing the location of one man by one square, or changing whose turn it is, can make the difference between a win, loss, or draw. For many of these positions, even expert players will require some time to fully comprehend the value of the situation. Therefore, the single function approach appears to have definite limitations, and it may he necessary to partition the state-space and make evaluation functions which are expert in certain state classes only.

Such expert functions could show their worth when branches lead to leaf nodes representing very different kinds of positions. Consider Figure 1.

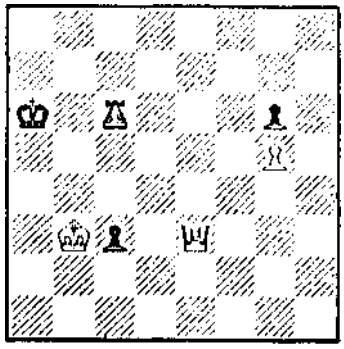

Figure 1

White to play

White can win by playing *QXP* because the K&P ending is won. But if the Black king were one square nearer to the center of the board, the *K&P* ending is a draw. Therefore, QxP is only effective now, and it is inconceiveable that White can win any other way. In the single function approach, it is almost inevitable that Q vs. R+P will be considered better than the materially even K&P ending. It would be simple to design functions which would judge both positions as superior for White. However, for the state-class approach to succeed it is necessary for the *K&P* endgame function to recognize that endgame as a win. Otherwise, this opportunity will be missed. This shows that at times it is necessary to have excellent goodness ordering across state-class boundaries. However, merely recognizing the goodness of a position may not be sufficient. Consider Figure 2.

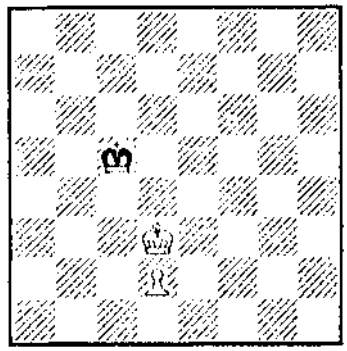

Figure 2

White to play

White has a winning position and can play K-B3, K-K3, or K-K4, all of which retain the win. However, if he chooses either of the first two alternatives, he will be back in the current position in another three ply. This is an instance of the "make-progress" problem which can frequently be resolved by the search, which notices that no progress has been made. Howevei, in more complicated examples, this may not suffice. Notice that if in Figure 7 the White king were at QB2 then it would be sufficient to recognize the position resulting from K-B3 as a win *because there are* no *real competitors to it.* Thus the make-progress issue can be separate from the goodness issue. The make progress issue was first treated by Huberman [Hub66] by having predicates which recognized nearness to a mate in simple winning chess endgames. However, the problem exists both in winning positions and also in positions that are not clearly won, but where a path toward progress exists. Another issue is "how to put up resistance", which arises in bad or losing positions. This is not quite the inverse of make-progress, because while there may be a guarantee of progress in superior positions, it is up to the losing side to find the method of resistance which requires the most knowledge (or calculation) from the superior side.

To investigate the above knowledge issues, we have for some time now been investigating backgammon, a game which due to its non-deterministic nature has a branching factor of about 800 at each node. Thus full width searches in this environment (as in GO) are not sensible. Our program achieved the ability to discriminate generally favorable and unfavorable factors together with appropriate coefficients for a single evaluation polynomial. This design reached a high beginner level, but there were great problems in having it understand more by merely adding more terms and tuning coefficients. In searching programs, some of these problems are subordinated by the search, since the program does not have to know about "means" up to the search depth, and a*ny* position potential up to that depth also does not have

to be inferred.

We have now implemented several state-classes which allow a great deal of context to be brought to bear on any evaluation. In some cases evaluation is iterative, so that additional Knowledge is only invoked when competitive moves exist. These steps have caused a great jump in performance of the program; however, it is still too early to evaluate the whole approach.

V. A modest Proposal

All these problems appear to be due to the fact that game playing programs indulge in one-pass evaluation of all nodes, i.e. the amount of knowledge that can be applied to a node has been predetermined. It is thus an efficient amount of knowledge, since applying all that is known would not be cost-effective most of the time. While this is fine for picking out the correct branch when there are no real competitors, this does not work when several nodes are very close in value or when there are other factors such as creating opportunities for opponent error. (Much of this was already pointed out in [New55]). Here a multi-stage process which weighs the pros and cons of each competitor for best node is required. It would seem that an ideal framework in which to do this search is the 8* algorithm. What is known about a node will increase as nodes in its sub tree are expanded, further, the amount of knowledge being applied can be a function of the degree of competitiveness between nodes. A first attempt at this type of problem solving mechanism is reported in [Per77].

References

[Ade75], Adelson-Velskiy, G. M., Alazarov, V. L, and Donskoy, M. V., "Some Methods of Controlling the Tree Search in Chess Programs", Artificial Intelligence", 1975, Vol. 6, No. 4, pp. 361-371.

[Ber73], Berliner, H. J., "Some Necessary Conditions for a Master Chess Program", Proceedings 3rd International Joint Conference on Artificial Intelllicnce, August 1973.

[Ber74], Berliner, H. J., "Chess as Problem Solving: The Development of a Tactics Analyzer", Ph. D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa., March, 1974.

[Ber77], Berliner, H. 1, "The B* Tree Search Procedure: Best-first Search Combined with Branch and Bound", Computer Science Dept., Carnegie-Mellon University, July 1977.

[Buc69], Buchanan, B., Sutherland, G., and Feigenbaum, E. A., "Heuristic DENDRAL: a Program for Generating Explanatory Hypothese in Organic Chemistry", in Machine Intelligence 4, (B. Meltzer, et. al., Eds), pp. 209-254, Amerkan Elsevier, 1969.

[Har74], Harris, L. R., "The Heuristic Search under Conditions of Error", Artificial Intelligence, 1974, Vol. 5, No. 3, pp. 217-234.

(Hub68], Hubermann, B. J., "A Program to play Chess lind Games", Stanford Technical Memo Cs 106, 1968, Computer Science Department, Stanford University.

|New55], Newell, A., "The Chess Machine: An Example of dealing with a Complex Task by Adaptation", Proceedings of the 1955 Western Joint Computer Conference, March 1955, pp. 101 108.

|Nil/0], Nilsson, N., Problem Solving Methods in Artificial Intelligence, McGraw-Hill, 1971.

[Per//], Perdue, C, and Berliner, II., "EG, A Case Study in Problem Solving with King and Pawn Endings", Proceedings of the Sirth IJCAI, Boston, 19/7.

[Pil771, Pitt at, J. "A Chess Progtam which uses Plans", To appear in Artificial Intelligence.

[Sam63], Samuel, A. L, "Some Studies in Machine Learning Using the Game of Checkers", in Computers and Thought, (feigenbaum, I. A. and eldman, J., Eds.), pp. 71-105, McGiaw Hill, 1963.

[Sam67], Samuel, A. I., "Some Studies in Machine Learning Using the Game of Checkers, II Recent Piogress", IBM Journal of Research and Development, Nov. 1967, pp. 601 617.

|Sla77], Slate, D. J., and Atkin, I.R, "CHI SS 45) -- The Northwestern University Chess Program", in Chess Skill in Man and Machine, (P. Frey, Ed), Springer Ver lag, 1 977.

|lur53|, luring, A. M., "Digital Computers applied to Games", in Faster than Thought, (B V Bowden, Ed.), pp. 286 310, Pitman, London, 1953.

USING PLANS IN A CHESS PLAYING PROGRAM

Jacques Pitrat
C.N.R.S.
Paris, France

1 present a program which does not develop systematically a large tree; but it analyzes carefully the initial situation and generates plans which it then executes. The analyses deeper in the tree are made only when something goes wrong and are always directed towards a goal. With this method, it is possible to find combinations requiring many ply-

Today the chess playing programs do not play as well as the grandmasters. One reason is that there are serious problems for developing the tree. This tree is necessary for finding possible combinations. Programs develop very large trees, but it often occurs that important moves are not included in the tree; the minimax procedure backs up an incorrect value if somewhere in the tree the best move

is omitted As it is difficult to generate large trees, the solution seems to be to improve the choice of the moves at all the levels in the tree.

I have realized a program founded on this principle. This ptogtam cannot play a game, but only indicates if a combination exists in the given position. The program analyzes, this position very meticulously. It does not, as anany programs, generate all the legal moves and then eliminate some of them later. But it searches for the characteristics of the position and, using these chanactics, it generates plans, i.e., a sequence of actions that may be tried.

The program looks for various characteristics, one of the most important is finding men which could be attacked, cither because they are not protected or because their value is high. When these men have been found, the program checks if two of them can be attacked simultaneously (double attack or pinning one of them if they are on the same line). Eventually, if there are some obstacles which are in the way, it indicates that they first have to be removed. If a man has a low mobility (as the King), it looks for an attack on him alone. In some cases, it first prompts an enemy man to move to some square before realizing the combination.

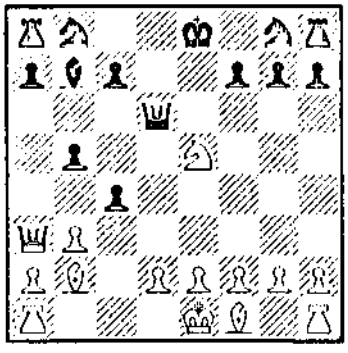

Figure 1

Black to Play

Fx. figure 1. This position was taken from a game IORRF Fdward Lasker. The combination was not seen by Laskrr who played f7 - f6.

I be knight on e5 is attacked once and protected once. The first plan is: Play Qd6xe5. It failed because White plays Bb2xe5. So the program modifies its plan and first adds a subplan for correcting this. For instance:

Move one of my men to c3 such that this man creates a threat.

Then play Qd6xe5.

[1]K for king. Q for queen. R for rook. B for bishop, N for knight. P for pawn. Each square is name by the combination of the letter of the file and the number of the rank.

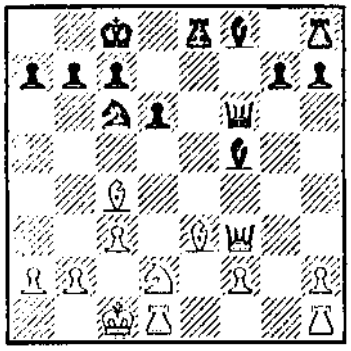

Figure 2

Black to Play

Fx. Figure 2. The white King cannot move. So the program tries to attack it. One possibility is to play a bishop on a3 after removing the pawn on b2. For playing a bishop on a3, it is necessary to first remove the pawn from d6. The following plan is generated:

> Remove the pawn from d6 so that it creates a threat. Remove the pawn from b2. Play Bf8~a3. Play Ba3xcJ.

In general, if the king and *an* unprotected knight are on the same file, the program tries to play a rook or the queen on this file. But it never considers a priori all the moves of the rooks,

This analysis is very slow, but it is done only once. The program usually produces several plans and then it tries to execute them.

Fx. Figure 1. For realizing its plan, Black can move a pawn to c3, threatening the bishop, therefore:

> Play t4-e.3 Then play c3xb2 or Qd6xe5.

Fx. Figure 2. Black removes the pawn on d6, if it plays d6-d5 threatening d5xc4. If after d6-d5, White plays Bc4xd5, the program tries the second element of the plan, i.e., remove the pawn from b2. The principal methods for removing an enemy man and leaving the square empty are: threatening it or capturing a man which it protects. Here the pawn protects the pawn on c3. The program looks for the moves capturing c3: Qf6xc3. If White replies with b2xc3, Black considers immediately the third element of the plan, i.e., Bf8-a3 mate.

We see that it is not necessary to analyze fully the other positions, when the plan is in action. It is sufficient to generate the moves which can realize some goal, for instance removing some enemy man or verifying that some move is always legal. There are two advantages: the analysis is fast and the program generates few moves: usually only a few moves satisfy a goal.

At each level of the tree, only "obvious" moves are considered. A move is added to the tree only if there is some reason to do so. If there is some problem, if the plan

cannot bo executed as it was foreseen, the program looks for the enemy moves which hinder the success of the plan. Then it modifies the original plan as follows: it adds a subplan which corrects the problem.

This entire method is applied for generating the moves of both players.

The program analyzes quickly the intermediate positions when all the plans fail. The main principle is to search for new possibilities to capture an opponent (moves which did not exist two ply before) such that this capture is advantageous (capture of an unprotected man or of a man whose value is greater than that of the capturing man).

Fx. Figure 1. After c4-c3, White has two new possibilities to capture: the first is d2xc3, but the second element of the Black's plan, i.e., Qdxe5, succeeds. The other is Bb2xc3 which also destroys the threat c3xb2. If now Black plays Qd6xeb, there is the new opportunity to capture: Bc3xe5 *and Bl*ack's plan fails.

Another way of using the initial analysis is to see if one of the initial plans cannot also be executed deeper. This can be done if the first condition of the plan is now fulfilled. In the case of Figure 1, the program generated several initial plans, and among them was:

Remove Bb2

Then play Qd6xa3

After *c4-c3* Bb2xc3, it sees that the first element of the preceding plan is satisfied. Then it considers the second element, Qd6xa3 and the combination succeeds.

If there is a new opportunity to capture, the program adds it in all its plans which are possible in this position. For instance, for Figure 1, if, after c4-c3, White plays Qa3xd6, the plan:

Play c3xb2 or Qd6xe5
becomes:

Play c3xb2 or c7xd6
since Qd6xe5 is no longer legal and c7xd6 becomes a new possibility to capture. After Black plays c7xd6, Black has a new chance to capture: d6xe5 and after c7xd6, it considers the plan:

Play c3xb2 or d6xe5

White cannot do anything against this plan, and the combination still succeeds. Naturally there *are* other variations.

The initial analysis of a position generates a set of plans. These plans generate a set of moves at the first level of the tree. If one of these moves leads to a failure, the program analyzes the reasons for this failure and eventually creates some new plans. With these plans, we may generate moves which were not considered before.

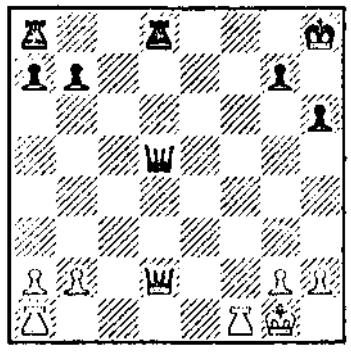

Figure 3

White to Play

x. Figure 3. The program finds only one useful characteristic: the queen on d5 is vulnerable. Only one plan is created:

Play Qd2xd5

After this move, Black plays Rd8xd5 and White has no advantage. So it tries to destroy the possibility of playing Rd8xd5. One way is to remove the rook. So a new plan is:

Remove the rook from d8

Play Qd2xd5

A method for removing a piece is to threaten it. It is possible with the rook fl:

Play Rf)-(8

Then play Rf8xd8 or Qd3xd5

So the program considers Rf1-f8 which was not first consiclered. It is not an obvious move, because the rook on f8 is not protected and is attacked twice. This is a check move, but the program does not consider it because the black king can move to h7 and because if the white rook is on f8, it *may be* captured. After Rfl-f8, Black may play Rd8xf8, and then White executes the following move in the plan: Qd2xd5. So Black considers Kh8-h7. If White plays the following move Rf8xd8, Black replies Ra8xd8 and the combination fails. But the alternative move of the plan, Qd2xd5 is good. Black plays Rd8xd5 and there is a new possibility to capture: Rf8xa8.

In the same way (see Figure 1), we have seen that c4-c3 was considered only because Black realized that it was interesting to obstruct the diagonal a3-d6.

The program does not develop the tree in depth first. It develops the nodes before or after the enemy moves, such that, if they were not legal, the balance backed up with the minimax procedure would be advantageous. If it succeeds, it applies the same method for the other player. Therefore it is necessary to represent the tree in the computer. But this is feasible, because the tree is not very large. It is

not possible to use the alpha-beta procedure, because after each node the program may later add new nodes. But the method used takes into account the provisory balance found with the minimax. It is difficult to compare my method with the alpha-beta procedure, but my method is also very selective: the program considers only the advantageous moves. If, for instance, some enemy move evaders an attack, it does not develop the other enemy moves at the same level.

The main difficulty in implementing this method is programming the analysis of the initial position. This is not a problem of computer time, because this analysis is made only once, but this program is large and difficult to define. For this reason, I do not program the detection of all the possible types of combinations. For this, it would be necessary to add some subroutines to the program. But I do not believe that it is possible to write a chess program which is simple and effective. Playing chess is a difficult problem and it would be necessary for several scientists to work several years to create a chess program playing as well as a grandmaster.

Berliner has written independently a very interesting program which has several similar features. For instance, if a node has been developed, and if the opponent has a combination after this move, then it is possible, with the causality facility, to generate new moves which destroy the combination. But most of the chess playing programs do not use such methods and systematically develop large tiees. Now they have better results because the authors use very clever methods to develop the tree. It is now necessary to work in another direction, using methods developed by Berliner and myself (and certainly some other methods which have not yet been found). The results are not always successful at this time, because it is difficult to write and check large programs, but, in my opinion it is possible to considerably improve the performance.

References

Berliner, H. J. "Chess as Problem Solving: The Development of a Tactics Analyzer", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University. March 1974.

Pitrat, J. "A Chess Program which uses Plans", To appear in *Artificial Intelligence,.*