NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A REPRESENTATION AND SOME MECHANISMS FOR A PROBLEM SOLVING CHESS PROGRAM *

By
Hans J. Berliner
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213
May, 1975.

I. INTRODUCTION

This paper is a condensation of a recent Ph. D. dissertation, [Berliner, 1974]. We describe a program, CAPS-II, and present both its form and some of the results obtained in testing it. The rationale which led us to the design of this program may be found in [Berliner, 1973]. Of necessity, we must omit much of the philosophy behind the program, many of the implementation details, and multiple examples of how it operates. However, we treat in detail what we feel are the major accomplishments of the work.

The domain of the work is chess tactics, and the emphasis is on recognizing situations and dealing with them explicitly. As such, we will be discussing 1) Recognition predicates, 2) Methods of stating specific problems so that their solution is easier than the general problems that include them, and 3) Ways that results of dynamic analysis (tree search) can be made available throughout a search tree for various purposes.

II. THE RECOGNITION MACHINERY

We first discuss our representation of chess, as it is necessary to understand the remainder of the program. We represent a position as a vector of 1040 words of information. Positions in the variation being currently analyzed are represented as 1040 word segments in a stack which allows analysis to a depth of 20 ply. Within the vector representing a single position there is a hierarchy of complexity of information.

The most primitive elements in this hierarchy of information are the pseudo-legal moves which define the possible transitions from position to position that could be allowable under the rules of chess. Upon this structure is erected the notion of a bearing relationship. This is a relationship of a piece to a square, and in our usage tells under what conditions the piece could pseudo-legally move to that square. The basic bearing relations used in this program are defined below.

DIR(PC,SQ) - A piece, PC, bears DIR on square SQ if, were SQ occupied by a king of the opposite color as PC, this king would be in check by PC.

---

* Presented at the 2nd Computer Chess Conference, Oxford, England, March 1975. This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defence (Contract F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

OTHRU(PC,SQ) - A piece, PC, bears OTHRU on square SQ if PC would be bearing DIR on square SQ, if it were not for another (intervening) piece of the same color as PC which has a DIR relation on SQ.

ETHRU(PC,SQ) - A piece, PC, bears ETHRU on square SQ if PC would be bearing DIR on square SQ, if it were not for one (intervening) piece of the opposite color which has a DIR relation to SQ.

DSC(PC,SQ) - A piece, PC, bears DSC on square SQ if PC would be bearing DIR on square SQ, if it were not for a piece of its own color, which is NOT bearing DIR on SQ.

OBJ(PC,SQ) - A piece, PC, bears OBJ on square SQ if PC would be bearing DIR on square SQ, if it were not for a piece of the opposite color, that is NOT bearing DIR on SQ. Intuitively, this corresponds to a pin ray by the bearing piece through the intervening piece (subject of the pin) and looking for an object to pin it to further down the line.

BLOK(PC1,PC2) - A piece, PC1, has a BLOK relation to a sliding piece, PC2, if, were all other pieces removed from the board, PC2 would then bear DIR on PC1.

BEH(PC,PWN) - A piece, PC, has the relation BEH to a pawn, PWN, if it is a rook or a queen and is behind PWN (as it would advance), and bears DIR, or OTHRO, or ETHRU on the square on which PWN is located.

These relations are sufficient to be able to determine whether a piece can: 1) Participate in a capture with or without an intervening piece participating, 2) Be a pinner of an opposing piece, 3) Be the source of a discovered attack if one of its own pieces were to clear the line, 4) Be in the path of a sliding piece if all intervening pieces were removed, and 5) Help or retard the advance of a pawn.

Once the above information has been gathered, the program embarks on a square by square analysis of the board. By using the above relational information it is able to decide which pieces affect the struggle on a certain square. For occupied squares, the safety of each piece is analyzed to be in one of five mutually exclusive categories. These are:

COMPLETELY EN PRISE - The full value of the piece on this square is subject to loss on the opponent's next move.

PARTLY EN PRISE - Only part of the value of the piece on this square may be captured with gain, but not the full value of the piece will be lost (i.e. a rook attacked by a bishop and defended a pawn).

BARELY DEFENDED - No capture with gain is possible now, but attacking this square with any one more unit of force will make this piece at least partly en prise.

COMPLETELY OVERPROTECTED - The piece is safe against any further single attack by a piece of equal or greater value (i.e. a pawn which is attacked by one pawn and defended by two pawns).

PARTLY OVERPROTECTED -The piece is safe against a set of single attack types, but not safe against the complementary set (i.e. a knight which is attacked by a rook and defended by a king and queen is safe against attacks by queen and king, but not against attacks by any lesser piece).

In the process of making these calculations, the action of pieces of both sides is invoked. Pieces are invoked in reverse order of value, the program also being able to avoid the use of pieces that are known to be pinned, until no more effective piece is available. The computation determines within excellent accuracy limits what the outcome of the struggle on a square will be. During this computation, pieces that are invoked in the safety analysis are assigned functions. By a function we mean a triple (PC, DUTY, SQ). A piece, PC, is said to have duty, DUTY, on square, SQ, if it was invoked for the purpose, DUTY, during the analysis of square, SQ. Typical duties are attacking, defending, over-protecting, and pinning, although blocking and supporting duties also exist in contexts which are discussed later.

Partitioning all pieces into categories which show how readily they are attackable, allows a further computation: noting which pieces can be usefully attacked by each class of enemy piece. For instance, it is now possible to determine the set of all pieces that can be usefully attacked by a White queen. This information is in turn used to determine squares where multiple attacks by a certain piece can take place. Other status information has also been gathered about individual pieces. Those that are pinned or considered low in mobility are considered to be worthwhile single targets. Squares where worthwhile attacks can take place are also noted. From the above, it is possible to list the set of all "attacking" moves for each side. The safety of each attacking move is then determined in the same way as was done for the safety of pieces, and functions are assigned which deal with the execution of the threatened attack and the defence against it. In this way a picture is built up of all the duties of every piece on the board, and this makes it possible to later ascend still further up the ladder of abstraction to find those moves which perturb the existing fabric of the position (as defined by the functions) in interesting ways.

Assigned functions are saved in a cross-referenced format that makes it easy to determine which functions a particular piece has, and what the set of all functions on a particular square are. Squares that contain en prise pieces are saved in a vector called BEST which keeps track of the threats against material by each side. In computing the above information, CAPS-II begins to put an interpretation on what is going on on the board.

III. GOAL STATES

Goal states play a major role in directing the activity of CAPS-II. They constitute a scheme for partitioning the moves that may be looked at at each node. A goal state defines intuitively a condition and explicitly a set of moves that are appropriate to the problem as perceived by the program. Only these moves may be searched as long as this goal state is in charge. This produces the desired economy of not having to search all moves or all "attractive" moves, but only those that are deemed pertinent to the problem at hand. This produces a significant

reduction in the branching factor of the over-all search. A node is always in one and only one of the following goal states.

AGGRESSIVE - This state consists of discovering and proposing moves that 1) Produce double attacks, 2) Cause attacks on low mobility pieces, 3) Produce discovered attacks, and 4) Vacate a square that another piece would like to occupy.

PREVENTIVE DEFENCE - This state is invoked when the side on move finds itself significantly ahead of expectation in material. The state then generates moves which attempt to consolidate the material plus by defending against any apparent threats.

NOMINAL DEFENCE - This state is invoked only when the position is deemed worth defending and no previously tried goal state has produced a good move, nor has a clear enemy threat been noticed in the process. This state defends against apparent threats in the hope that this will satisfy the needs of this node.

DYNAMIC DEFENCE - This state is actuated when a search has backed up to a node with an unsatisfactory value, and the last move tried at this node was blameless. In this case the CAUSALITY FACILITY (see Section VII) is invoked and this state is a part of its operation and is discussed in that context later in this paper.

STRATEGY - This state is invoked only at depths which are specified at the start of the game (and has always been equal to 1 thus far). The strategy implemented here is to call the legal move generator of TECH [Gillogly, 1972], which does a positional sort on the legal moves. These are then tried in the given order, with the proviso that moves that were already searched are not searched again. Since this in effect means opening up the search completely, STRATEGY is invoked only at the root of the tree in the current program. This is a very primitive way of looking at strategy. However, the module is completely independent of everything else, and could be replaced incrementally by more sophisticated procedures. Within the present framework, it allows the program to play complete games by making a move selection agency available in situations where no tactical move is preferred.

KING IN CHECK - When the king is in check, this state is invoked directly, since the set of legal moves is usually small and can be sorted effectively based upon knowlege of the safety of squares for the pieces.

How the program passes from one goal state to another is discussed in Section VIII.

IV. MOVE GENERATION

In CAPS-II move generation is very dependent on the recognition machinery. Move generation is done under control of a Goal State, which decides the type of move we should be looking for. The following simple move generators are then used to try to find moves that match the description of the desired set of moves.

OCCUPY(SQ) - is a move generator which generates all pseudo-legal moves to square SQ for the side on move. One of the uses of OCCUPY is in generating captures.

MOVEAWAY(SQ) - is a move generator which generates all pseudo-legal moves for the piece on square SQ. It is useful in defence considerations and for generating discovery moves.

INTERPOSE(SQ1,SQ2) - SQ1 and SQ2 define a straight line with SQ1 being the name of the square on which an attacker resides and SQ2 being the square on which a target resides. INTERPOSE finds all intervening squares, and then repeatedly calls on OCCUPY to provide the complete set of interposing moves. Before doing this, INTERPOSE first finds the value of the attacker, and sets a global constant which lets OCCUPY know that moves that counter-attack the attacker along the specified line are to get special heuristic credit.

MOVTOCON(SQ) - generates the set of all pseudo-legal moves which bring a piece of the moving side which at present does not have DIR control on square SQ, into a position where it does; i.e. brings a new piece to bear on SQ.

V. MOVE EVALUATION

Moves are then evaluated by the procedure EVALUATE. Our information environment is detailed enough to allow detecting and scoring a proposed tactical move in the environment of the old position, rather than by setting up the new position. This results in a definite saving in computing time. Further, the assignment of functions results in binding certain pieces to certain important duties on the chess board. These duties are considered to be essential if the existing stability of the current position is to be maintained. Therefore, when a move results in perturbing these functions, it is possible to gauge the effect of such a disturbance. Detecting perturbation effects involves noticing whether a piece is moving en prise, whether any piece that is set to capture another is committed to other important duties, and whether a moving piece clears or blocks any important squares.

To understand perturbation of a position, it is important to be aware of the various types of moves that can occur and how their merits can be determined. For instance, many programs that use forward pruning of legal moves include all moves that are checks or captures, even if they involve loss of material, just on the chance that some other factors exist which may make such a move successful. Those "other factors" would, however, have to be discovered at the exponential cost of tree searching. The present program can, because of its refined analytical methods, rule out many checks and captures as completely worthless. This would occur, for instance, when a capture will result in loss of material without causing commensurate "disturbing effects" on the position. The ability to optimistically gauge such disturbing effects and thus dismiss certain "sacrificial" moves as having absolutely no sacrificial merit, is one of the things that our evaluation procedure is be able to deal with effectively.

EVALUATE looks for the main perturbing effects that can occur. These are known in the chess literature as: 1) Guard destruction, 2) Piece overloading, 3) Decoying, 4) Line Blocking, 5) Unblocking, and 6) Desperados. We discuss each briefly.

Guard destruction occurs when a piece that has a defensive function is captured. Piece overloading occurs when a move is made that requires the exercise of a defensive function by a piece which has another defensive function to fulfill, and cannot fulfill it from the new square. Decoying occurs when a piece has a defensive function, but the exercise of it will bring the piece to a new square. The defending piece is "decoyed" to this new square, the only necessary condition being that the cost of decoying the piece not be greater than the value of the piece itself. Of course, the ultimate success of the decoy depends on whether it can be successfully attacked at its new location. Line blocking occurs when the action line of an enemy piece with a function is blocked. This can result in either the severing of a defensive tie, or the blocking of an attacking move. Likewise, unblocking occurs when the moving piece makes possible a future attack by one of own pieces either on the square in question or by travelling across the square. Desperados are the sacrificing of men that are already attacked, because an enemy piece of similar value is also attacked and can be expected to be captured immediately after the sacrificed piece is captured. All but the last of these effects depend directly on recognizing the functional bindings that are involved. It should be noted how easy it is to describe the perturbing effects, given the notion of functions, while without this it would be rather difficult.

EVALUATE scores the effect of a proposed move on an existing position. The resulting score is an integer, which represents an optimistic view of what the proposed move could accomplish. To facilitate this process, any move which is proposed gets a quantitative recommendation from the agency which proposes it. The amount of the recommendation is a function only of the effect that the move could have (according to the proposing agency) if it were successful. How much of this value is actually given to the move depends on EVALUATE.

EVALUATE first examines the safety of the new square for the moving piece. If the piece cannot be captured without loss on the destination square, then it gets credit for whatever value its recommending agency gave it. However, if it can be captured advantageously, the following procedure is performed: Each opposing piece that has been assigned a defensive function on that square has its function list examined to see what other functions it has. The following quantities are then computed:

DEFENSIVE OVERLOAD = Maximum [across all squares on which this piece has defensive functions] (The number of material units defended on this square + the value of enemy threats associated with this square).

DECOY VALUE = Maximum value (any opposing piece that has a defensive function on this square).

DISTRACT VALUE = Maximum [across all opposing pieces that have a defensive

function on this square] (Value of any en prise piece on which this defensive piece has an attacking function).

REDEEMING VALUE = Maximum (DEFENSIVE OVERLOAD, DECOY VALUE/2, DISTRACT VALUE/2).

The Redeeming Value is the evaluation for moves that apparently "don't work", while the evaluation of moves that appear to work is the sum of their recommendation from the proposing agency and the beneficial side effects that they cause.

If a move is selected for searching, two evaluations are developed for the corresponding node in the tree. They are the nominal value which counts the material on the board and gives credit to each side for all their material threats as cataloged in BEST, and the pessimistic value (from the point of view of the side that is to move) which counts the material on the board and then adds in only the side-which-just-moved's best capture threat.

VI. TREE CONTROL

The basic tree search used by CAPS-II is a depth first, mini-maxed search with Alpha-Beta pruning. This has been supplemented by algorithms which make risk decisions. During the tree search the basic emphasis is on: 1) Trying to find properties of the current node which allow termination of the search at that point, 2) Making deductions about the current goal state which may lead to abandoning the state or the node, and 3) Forward pruning of proposed moves which fail to have certain necessary properties, in order to limit the number of descendants of any parent node.

CAPS-II uses the following level of aspiration scheme. Two limits of aspiration (Alpha and Beta) are needed in order for each side to know what is the maximum it can hope to achieve at any point in the tree search. An expectation (EXPCT) is needed, that is the best estimate of the value of the position at the root of the tree. Around EXPCT, a margin (MARG) is defined. If a value, differs from EXPCT by more than MARG, we say it DIFFERS SIGNIFICANTLY from EXPCT. If such a value is ever backed up to the root of the tree, EXPCT is changed to that value, and the search is redone.

Alpha and Beta are set initially at plus and minus infinity. The value of MARG is set permanently at 68% of the value of a pawn. EXPCT is provided on input of a position and retained from one tree search to the next as the minimaxed value of the last tree search. If a value that is greater than EXPCT plus MARG is ever backed up to the root of the tree, the following occurs: EXPCT is set to the value that has just been backed up. The new Alpha-Beta limits become plus infinity and EXPCT minus a very small quantity. The search is repeated unless the value returned is sufficiently high to guarantee a completely winning position for the side making the gain. An algebraically opposite adjustment is made if the value returned to the root of the tree is EXPCT minus MARG.

In order to prevent oscillations in the value of EXPCT, a very conservative view of position evaluation must be taken. This means that wherever there is doubt about what the terminal value of a position is, the value closest to EXPCT must be chosen. The reason for this is that when the search at a node is terminated as described above, this is almost invariably a non-quiescent value. If such a value should survive all the way to the root of the tree, then this will become the new EXPCT for the next tree search. So if the estimate at the terminated node exceeds that which can in reality be achieved from the root node position, the program could well be in a state on the next search in which it cannot fulfill the new EXPCT. This would then result in EXPCT being reset to a lower value, and oscillation could result from this. Therefore, whenever an estimate is made for backing-up purposes, it should be conservative with respect to EXPCT. This means that if a value is significantly greater than EXPCT then only the pessimistic value of this position may be backed up. Likewise, if a value is significantly below EXPCT, the optimistic value of the position must be backed up. Clearly, if the new estimate does not remain significantly different from EXPCT then the conditions for node termination have not been met.

VII. THE CAUSALITY FACILITY

Whenever the search backs up to a node with an unsatisfactory value, the CAUSALITY FACILITY is invoked. The CAUSALITY FACILITY allows determining whether a set of consequences can be definitely dissociated from the last move tried at a node. Only the detection of this condition allows fixing the blame for a set of consequences on something that existed before the search came to this node. Once it is known that the node has inherited a problem, the necessary mechanisms can be set in motion for trying to solve it. Causality is established by comparing a description of a set of consequences (the Refutation Description) with a description of a move. The CAUSALITY FACILITY then decides whether the consequences could have in any way been made possible by the move made. We first describe the data used by the CAUSALITY FACILITY. Then we take up how the CAUSALITY FACILITY gathers this data and uses it for comparisons and decision making.

A. The Refutation Description

During the backup process, whenever a result is acceptable to Alpha-Beta, the following data are collected at that node for use by the CAUSALITY FACILITY. These data constitute the Refutation Description. While the simple notation of specifying an origin square and a destination square is enough to describe a move or to play out a game, this is not enough to describe a sequence of move without having recourse to updating. For the task we are about to describe, we need a description that is rich enough so that properties of a move can be understood without having recourse to updating the board. To do this, we need to know not only the squares of origin and destination, but also the name of the moving piece (since the origin square could at some other time be occupied by a different piece), and the squares which had to be unoccupied in order for the move to be made. These things are essential to a description that can examine a sequence of moves without actually going through the process of setting up the position after each move in the sequence.

RPCS - is a bit-vector which has bits representing names of pieces. The name bit of the piece that moved to produce this node is set in this vector.

RSQS - is a bit-vector with bits representing squares on the board. The bit corresponding to the destination square of the move that produced this node is set in RSQS.

RPATH - is a bit-vector with bits representing squares on the board. The bit for any square across which a sliding piece moved in making the made move is set in RPATH. If the move was a non-capture pawn move, then all squares over which it passed including the destination also have bits set for them.

RTGTS - is a bit-vector with bits representing names of targets. A comparison is made of BEST for this node with BEST one ply previously. Any squares which are now named as containing material targets, but were not mentioned in the previous BEST, have bits set for the name of the piece on this square to indicate that this threat was created by the last move.

TGTSQS - is a bit-vector with bits representing squares on the board. For any RTGTS detected as above, bits are set in TGTSQS for the corresponding squares.

TPATH - is a bit-vector with bits representing squares on the board. For any TGTSQS detected as above, if a piece that has an ATTACKING function on this square is a sliding piece, then all the intervening squares have bits set for them in TPATH.

Once this information is generated, it is accumulated during the backing-up process of the tree search. This is done by forming the union of the current description and the previously existing description whenever a node's value is accepted. Thus when returning to a node, a complete description of all that each side has accomplished in the immediate sub-tree and how, is available. If the results of the last move tried at this node were not satisfactory, the CAUSALITY FACILITY consults the Refutation Description in order to decide what can be done about it.

B. The Rationale for the CAUSALITY FACILITY

To understand the value of having a rich representation for this, consider what is possible without it. For instance, assume a sequence of moves resulted in a loss of material. Best current practice would be to remember the first move of the backed-up variation as the "killer", and then try it first on every move that is served up from here on in the generate and test mode. If the representation was richer and we could get a complete description of all moves in the backed-up variation, then it would be possible to determine the sequence of moves that produced this result. We would then be limited to doing things about this sequence only. This would include such things as suggestions to move or defend any captured piece, capture or pin any capturer, or block the path of any moving piece. However,such a scheme is incomplete since it does not deal with threats that were adequately met.

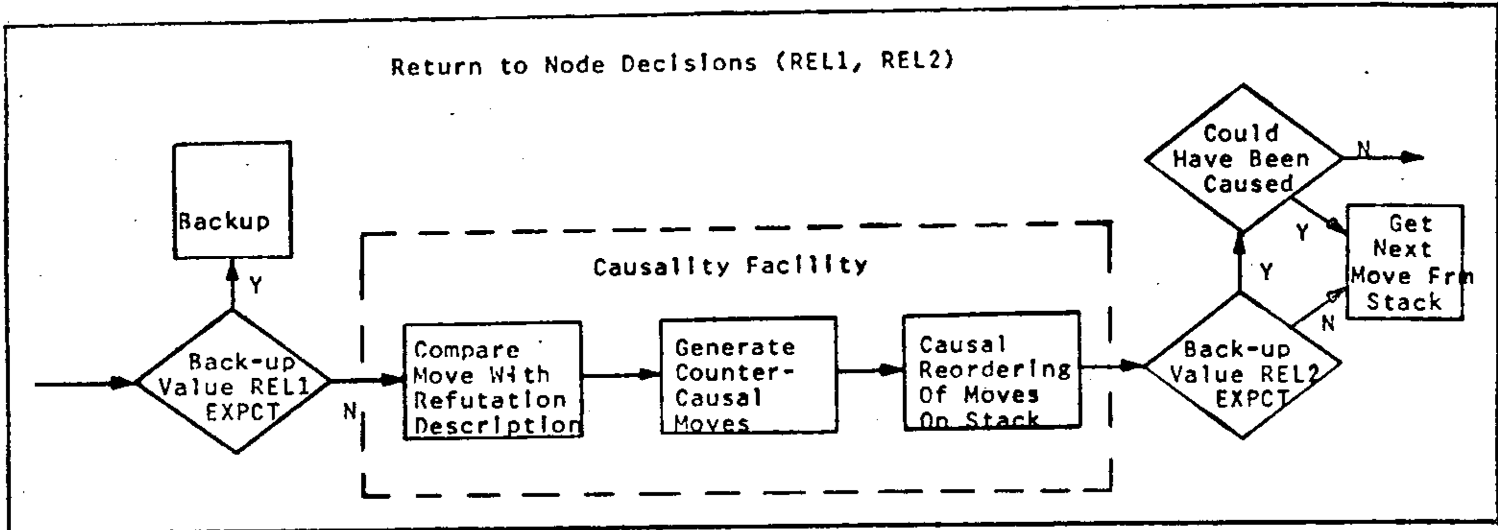
The present program has a much more complete understanding of a set of consequences. The set of data that the program abstracts from a position and sends back up the tree was discussed earlier. This includes a knowledge of all squares critical to the transportation of pieces that moved, squares on which pieces became targets, and squares over which threats passed. It also includes the names of all pieces that moved or became targets.

When returning to a node, the CAUSALITY FACILITY correlates this description with changes that occurred in the data structure as a result of the move tried at this node. This includes noticing changes in control of critical squares by the losing side, changes in threats as noted in BEST, and whether any unblocking of critical paths occurred as a result of the last move. Making comparisons of these quantities with the Refutation Description makes it possible to decide whether this move could be to blame for what happened. Whenever this is not the case, the search for a direct method of preventing what happened can begin. For instance, assume a knight was lost as a result of a double attack which also involved the king. Then moving the king away, or blocking the threat path to the king are validated as goals for meeting the threat, as well as doing things about the knight and trying to capture the attacking piece or guard the squares on which attacks occurred. The first two goals of this set would not show up in the principal variation, since the major threat is usually avoided. Thus, the present method gets directly at the whole set of consequences, not merely those which were executed in the principal variation.

The CAUSALITY FACILITY, does very well at generating defences to deep threats, as is demonstrated in Section IX. As a consequence, it is not necessary to make a priori decisions about the goodness of certain moves for "defensive purposes". Rather, it is possible to wait to see if a defensive problem occurs and then generate the moves that do something about this description. While this is a major advance in the state of the art, it is still considerably short of human performance. First, there are situations in which many defensive moves are suggested, and the program is unable to assign accurate enough values to these moves to prevent a certain amount of hit-or-miss searching. Second, the problem of indirect defences is not treated at all. An indirect defence occurs when a threat is met by playing a move that would allow the execution of a desirable move sequence ONLY if the opponent tried to realize his threat. This is quite different from a counter-attack, since the indirect defence is intended to produce its result only when the opponent persists in his attack. A typical indirect defence would involve preparing to move a piece through a square that would be vacated in the process of attempting to execute the threat. The method for detecting indirect defences is to make a null move and then execute the opponent's detected threat sequence. In the final position, the indirect defender now tries to find two moves in succession which would produce a favorable result. One of these moves would then have to be substituted for the null move in order to make the indirect defence work. However, implementing schemes such as this is beyond the scope of the effort reported herein.

VIII. Goal State Transitions

The way the program progresses from goal state to goal state, once having



START

KING IN CHECK STATE

Y

King In Check

N

MAT >> EXPCT

Y

Pess. Eval. >> EXPCT

Y

Backup

N

N

AGGRESSIVE STATE

When Stack Empty

PREVENTIVE DEFENCE STATE

NOTE:
>> means
Significantly
Greater

Return To Node Decisions (>> , <)

Delete Insufficient Counter-Atk Moves

If Not Caused

Return To Node Decisions (>> , <)

If Not Caused

Return to Node Decisions (>> , <)

When Stack Empty

Add Counter Causal Moves To Stack

Nominal Eval. ≥ EXPCT

Y

Elig. For Strategy

Y

STRATEGY STATE

N

N

DYNAMIC DEFENCE STATE

Backup

Return To Node Decisions (≥ ,)

Optim'ic Eval. ≥ EXPCT

N

Backup

Y

Back-up Value >> EXPCT

N

Y

N

N

Y

When Stack Empty

Y

N

Back-up Value ≥ EXPCT

Y

Worth-while Posit.

NOMINAL DEFENCE STATE

Last Move In Prin. Var.

Y

Back-up Value ≥ EXPCT

N

FIGURE 1 - Goal State Transition Diagram

reached a given node, is shown in Figure 1. The goal state at each active node in the tree is remembered in the representation of the node. This means that goal states do not change because of departure and return to a node, but only because of overt decisions made in the course of problem solving at the node. Similarly, the move stack at each node is available for inspection until the node is finally quitted. Thus, any move examined in one goal state will not be tried again if suggested by another.

The "Return to Node" block at the top of Figure 1 shows the decision structure that pertains when returning to a node in most goal states. It is invoked as a sub-routine, with arguments REL1 and REL2, by the main transition diagram in the lower part of Figure 1. REL1 is the relation between the backed-up value and EXPCT that when true allows exiting the node immediately. Otherwise, the CAUSALITY FACILITY is invoked. The CAUSALITY FACILITY compares the description of what is best play below this node (the Refutation Description), with the description of the move made at the node. Based on this comparison, it makes the decision as to whether the consequences could have been caused by the present move. In either case, a list of counter-causal moves is generated. These moves attempt to do something about the Refutation Description that has been backed up. The exact methods are described in Section IX.

REL2 specifies a relationship between the backed-up value and EXPCT. This relates to whether the backed-up value is satisfactory with respect to the aims of the goal state that the node is currently in. If an unsatisfactory value has been backed up to this node, and the causal analysis reveals that this could not have been caused by the last move tried at this node, it means that a problem has been inherited from higher in the tree. In that case, a transition to a new goal state occurs. If, on the other hand, the result is not deemed to have been caused by the last move or if REL2 does not obtain, then the program merely does a reordering of the untried moves on the move stack, moving those mentioned most often in the counter-causal list to the top of the untried stack. This Causal Reordering will result in their being tried earlier, but does not change their value.

Following now the flow chart in the lower part of Figure 1, we see that the first determination made is whether the king is in check. If so we go to the KING IN CHECK state in which all legal moves are generated. These are tested in order of decreasing evaluation. If a value is ever backed up to this node which is significantly greater than EXPCT, the search backs up. If the CAUSALITY FACILITY detects a consequence which could not have been caused by the last move tried, the state is changed to DYNAMIC DEFENCE, but the move stack remains intact. If neither of the above occur, causal reordering of the untried moves takes place.

If the king is not in check and if the side on move is significantly ahead of EXPCT in material then if the pessimistic evaluation of this node is also significantly greater than EXPCT the conditions for backing up have been met. If the latter condition has not been fulfilled, then it means that the opponent still has some important threats (else the pessimistic evaluation would be better). In this case, the PREVENTIVE DEFENCE state is entered. Here all moves that move a threatened piece, capture an attacker, block an attacking line, or defend a threatened piece are generated. If this set of moves, when submitted to tree searching fails to

maintain the significantly greater than EXPCT advantage or if any dynamic problem not caused by the tested move is detected then the AGGRESSIVE state is entered. This means that the attempt at consolidating the gains has failed, and the program resorts to the more usual method of dealing with a node.

It is possible to get to the AGGRESSIVE state as above, or if the material significantly ahead of EXPCT test fails initially. This means that we know of no reasons at the moment why the side on move should not try to make a successful attacking move. As explained earlier, the AGGRESSIVE state is really a set of move generating states. The states are arranged so as to generate moves in order of forcefulness. EVALUATE gives each move a likelihood of success measure. A sorting routine then arranges the moves according to likelihood of success within forcefulness. This results in generating all moves that have aggressive potential (with the exception of moves that only involve an attack on a single non-low-mobility man, this feature not having been implemented as yet).

After an AGGRESSIVE move has been selected for tree searching, and the search has returned to this node, several things can happen. If the program has found a move that is significantly better than EXPCT, it will back up from this node. If the backed-up value is not significantly better than EXPCT, then the CAUSALITY FACILITY is invoked to do causal analysis and reordering of untried moves. If the backed-up value is less than EXPCT, and if the consequences could not have been caused by the last move tried, then the DYNAMIC DEFENCE state is entered. Otherwise, the program selects the next move from the move stack until it exhausts the proposed moves in the AGGRESSIVE state. In that case, if the nominal evaluation of the position is not less than EXPCT, a check is made to see if this node is at a depth that makes it eligible for STRATEGY. If so, this state is entered. Otherwise, the node is exited (i.e. BACKUP). If no successful AGGRESSIVE move was found then if the nominal evaluation of the position were less than EXPCT and the optimistic evaluation greater or equal to EXPCT, then it means that the opponent must have a threat. Since the position has the potential to produce a satisfactory value, the NOMINAL DEFENCE state is entered.

The NOMINAL DEFENCE state is charged with producing defences against statically recognized threats. It is only invoked when no AGGRESSIVE moves have succeeded and the position is considered worthwhile. If in any prior processing of this node, a backed-up threat had been recognized, then this state would be by-passed in favor of DYNAMIC DEFENCE. This is very logical, since NOMINAL DEFENCE deals only with statically recognized threats, and one can never be sure that such a "threat" is really a threat. The move generators of the NOMINAL DEFENCE state aproduce moves that defend threatened points, move away the pieces on these squares, capture their attackers and block attacking lines. The NOMINAL DEFENCE state is exited as soon as a move which produces a backed-up value greater or equal to EXPCT is found. If in the process of testing moves, the CAUSALITY FACILITY finds a threat that could not have been caused by the last move tested at this node, then the DYNAMIC DEFENCE state is entered.

The DYNAMIC DEFENCE state is invoked whenever a deep problem has been detected during back up, which was clearly not made possible by the last move tried.

The counter-causal moves which are deemed to be the only ones that can do something about this description, have already been generated by the CAUSALITY FACILITY. Now all moves on the stack which are not mentioned in the counter-causal list, or which do not have a counter punch at least equal and opposite to the caused value (with respect to EXPCT) are deleted. The remaining counter-causal moves are pushed onto the stack in order of evaluation. These moves are now tested until a value is backed up which is greater or equal to EXPCT. Then if the material plus the best threat of the side to move are not larger than the backed-up value, the node is exited. Otherwise, the search continues until all proposed moves have been tried.

In the present program, the STRATEGY state performs the function of parading all legal moves for testing. It does this only at the root node of the tree (in the current program), and only when the AGGRESSIVE state has failed to produce anything worthwhile. The move generator of the STRATEGY state emphasizes the centralizing and mobilizing effect of each move. It is, in fact, the TECH [Gillogly, 1972] move generator which is available to this program as a sub-routine. If a value is ever backed up which is significantly greater than EXPCT, the node is exited. Otherwise, when the move that is now in the principal variation is proposed, if the current Alpha at the node is greater or equal to EXPCT, the node is exited. Otherwise, the search continues as long as there are moves to try.

IX. THE CAUSALITY FACILITY AT WORK

A. An Expository Example

Figure 2 shows a position in which Black to play has a defensive task. White is threatening mate in two beginning with 1. Q-K8ch. When CAPS-II is presented this position, it finds no particularly inviting offensive moves since all the Black queen checks are adequately guarded and there are no double attack moves. It therefore asks the STRATEGY routine for a move and starts out with 1.--P-R7.

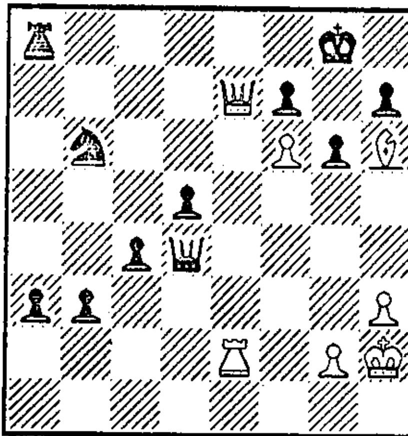


Figure 2

Black to Play

Play now proceeds 2. Q-K8ch, RxQ, 3. RxRmate. This result causes the backing up process to begin, and with it the accumulation of the Refutation Description. Here, we will only follow the process associated with the White moves, since the process associated with the Black moves, even though it also goes on, yields no meaningful results in this case. After the moves 1.--P-R7, 2. Q-K8ch, RxQ, 3. RxRmate, the search begins to back up. When the move 3. RxRmate becomes part of the local principal variation during the backup process, a description of the change in environment that it produced is generated. This description consists of putting the name of the moving rook into RPCS (refutation pieces), putting its destination into RSQS (refutation squares), and putting the squares on its path (K3,K4,K5,K6, and K7) into RPATH. Since the move resulted in a capture, the name of the captured piece is noted in RTGTS (pieces that became target during the refutation). The move resulted in a change in the threat picture in so far as the Black king is now attacked when it wasn't one ply earlier. This fact is incorporated by noting the square of the threatened piece (the Black king) in TGTSQS, its name in RTGTS, and putting the path squares (KB8) associated with the threat into TPATH. The above entries describe the essential points of interest in the current position and the important changes from the previous one. As the new principal variation continues to survive during backup, this Refutation Description is backed up too.

The first place where this Refutation Description can be used is one ply further up the tree, at the point where Black played 2.--RxR. Here, a causal test is performed which shows that the move 2.--RxR could have caused the consequences described in the Refutation Description since it moved to a square mentioned in RSQS. The exact nature of other tests performed as part of the causal test are described later in this example. Since the consequences could have been caused by the last move played, the search at this node continues. But first a set of counter-causal moves are generated, which could be tried in an effort to avoid the consequences anyway. However, here they are useless since there was only one legal move, and that has already been tried.

As backing up continues and the move 2. Q-K8ch becomes part of the new principal variation, a description of it is generated. This consists of putting the name of the queen into RPCS, putting K8 into RSQS, and (since the queen did not cross any squares) putting no path squares into RPATH. The noting of the new threat to the Black king (as against the lack of this threat one ply previously) causes its square to go into TGTSQS, its name to go into RTGTS, and the name of the square on the threat path (KB8) goes into TPATH.

When this move is backed up, the union of the new description and the existing Refutation Description is produced. When the backing up process reaches the point where Black originally played 1.--P-R7 this description is examined. The following tests are made by the CAUSALITY FACILITY to determine whether the move 1.--P-R7 could have brought on the consequences described in the Refutation Description. First a test is performed to see whether the move resulted in moving onto an RSQS square. This is not so. Then a check is made to see whether the name of the moving piece is mentioned in RTGTS (became a later target). This is also not so. Then a check is made to see whether the move vacated a square mentioned in RPATH

or TPATH (making a refutation move or threat across this square possible). This, too, is not so. Then each square mentioned in RSQS or TGTSQS is checked in the representation before and after the move 1.--P-R7 to see if something about the move caused either fewer of own pieces to bear DIR on such a square, or more of the opponent's pieces to bear DIR on such a square. We are interested here both in whether the move resulted in unprotecting such a point, and whether it could have permitted a new enemy piece to bear on the square. Here this involves only K8 and KN8, and no change in the control of those squares occurred. The final test involves noting the pin status of all pieces mentioned in RPCS to see if any such piece was pinned before the made move, and unpinned immediately afterwards. This, too, is not so. Therefore, the conclusion is reached that 1.--P-R7 could not have caused the consequences, and these must therefore have been inherited from above.

The counter-causal move generator is now invoked in order to generate those moves that can directly counter this description. The counter-causal move generator calls MOVTOCON to generate moves which add new DIR bears on all squares mentioned in RSQS. Here there is only one square (K8) and there is no new way to defend it. Next it calls OCCUPY with the squares of any piece mentioned in RPCS, in order to generate moves which capture pieces involved in the refutation. These pieces are the White queen and rook, and here neither of them are capturable. An additional facility which is not yet in the program could impede the movement of such action pieces by trying to pin them against something of greater or equal value to the actual consequences in the principal variation. Next, OCCUPY is called with every square mentioned in RPATH and TPATH, to generate moves which block such paths. This yields Q-K4ch, Q-K5, Q-K6 and R-KB1. Then MOVTOCON is called with the names of squares in TPATH, with the idea that putting a piece in position to occupy such a threat path may defend the threat. Here the only square in TPATH is KB1, and thus the move N-Q2 is generated. Finally, an attempt is made to remove targets by calling MOVEAWAY with the name of any square mentioned in TGTSQS which is occupied by a piece mentioned in RTGTS. This yields the move K-R1. A check is then made to see if any piece mentioned in RTGTS is a low mobility piece which is not presently attacked. Here, the Black king qualifies and MOVEAWAY is called with the names of squares that are presently occupied by any king's own piece and to which the king could otherwise have access. However, here neither the KBP nor the KRP can be moved. Thus the counter-causal move generator ends up with suggesting six moves: Q-K4ch, Q-K5, Q-K6, R-KB1, N-Q2, and K-R1. After a little tree searching, the program decides that the optimum variation for both sides is: 1.--Q-K5, 2. RxQ, PxR. It does not recognize that Black now has a winning position (all of White's threats have been met and there is no effective method of prevent the queening of the Black QRP), but it does find this only defence very quickly.

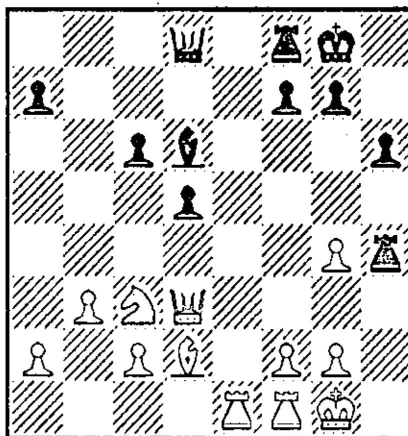


Figure 3

Black to Play

B. Causality Reordering

An example of how the program uses causality in order to improve its attacking processes can be seen in Figure 3. Here it is Black to play. After spending some time on non-productive issues the program finds the perpetual check: 1.--B-R7ch, 2. K-R1, B-Q3ch, 3. K-N1 with repetition of position. It then raises EXPCT to equality (Black was down in material in the original position), and looks to see if there is something better. The next thing tried is 2.--B-B2ch, 3. K-N1, (here there is no repetition of position and the functional similarity is not discernable to the program), B-R7ch and now a repetition is again noted. Next the program backs up one ply and tries 3.--R-R8ch, 4. KxR, Q-R5ch, 5. Q-R3 and decides this position is not good for Black. It then begins to back up, generating a Refutation Description of all of White's (the refuting side because there was a cut-off) moves. The first point where something can be done about the description is at the point where Black played 2.--B-B2ch. Here the Refutation Description is used to generate the set of counter-causal moves. This set is then matched with the moves already on the move stack. Any matching move is promoted to a place higher in the move stack. The move that matches the counter-causal set most frequently is promoted to the highest place. In this case the Refutation Description mentions the king and queen as RPCS, and mentions the path of the queen in blocking the check in RPATH. Nothing can be done about capturing the king or queen, but among the discovered checks with the bishop that have already been proposed is B-N6ch, which matches a move in the counter-causal set proposed for the purpose of blocking the queen's path. The program then tries 2.--B-N6ch, 3. K-R1, B-R7ch and again finds the repetition of position. Backing up one ply, it tries 3.--R-R8ch, 4.KxR, Q-R5ch, 5. K-R1, Q-R7mate. This variation is forced and nothing can be done about it so the search is exited and the progam announces mate in five moves. It should be pointed out that things do not always work out so favorably when causal reordering is invoked. If the initial idea tried is unworkable, then moves that help a hopeless cause are promoted. However, by and large, the mechanism helps considerably more than it hinders.

X. TESTS ON THE PROGRAM

CAPS-II was tested on many middle-game chess tactics problems from standard textbooks on chess. We also had it play a few complete games of chess. In both these modes it ran with a maximum depth of 10 ply. CAPS-II did not do too well in the games, since it has little positional knowledge, and more importantly the tactics mechanisms that it has are still not complete, causing it to make occasional blunders which would wipe out whatever good it had achieved earlier. However, it did quite well on the tactics problems as reported below.

It is interesting to note that the branching factor (average number of successors to a node) of the trees grown while solving textbook problems is 1.5, while the branching factor for games is about 3.0. The curve for the expected number of successors for a node is in the form of an exponential decay in both cases. Approximately 70% of all nodes have zero or one successor. This compares extremely well with standard programs which have a branching factor of about 5 to 6. Since the program does as yet not play as well as these nor treat positional issues, some caution should be exercised in evaluation this fact. However, from our experience in developing the program, we do feel that branching factors of this low order can be maintained by continuing the building process in the same vein as what has been reported herein. The reason that the branching factor is higher for games than for problems appears to be that the activity in problems is better focussed. If the program is able to follow this focus it solves the problem in a reasonably effective manner; if not, it usually does not find too much to waste its time on. In games, the situation is frequently not so clear, and the program spends more time in exploring non-productive issues.

CAPS-II was presented with the first 200 problems from Reinfeld's "Win at Chess" [Reinfeld, 1958], a book that teaches chess tactics by examples. Since these problems were also presented to the program TECH and a Class "A" player there were good comparative data available for evaluating CAPS-II's performance. For all three performers the performance criterion was that the problem had to be solved in five minutes of (CPU) time to be counted correct. For the programs, supporting output was required to show that the correct answer was not selected spuriously. Depth is defined as the depth of the deepest non-capture in any branch of the principal variation. This definition is being used mainly because of the structure of TECH, in which all capture sequences are examined as part of the quiescence process. Thus if a principal variation ends with one or more captures, these would be included as part of the quiescence analysis, if the search went to the depth of the previous non-capture move in that variation. This is not an unreasonable definition of depth, since in most positions there exist sequences of captures which either do not disturb the status quo or reap the fruits of the previous moves. Both these situations can be considered to be "self-evident" extrapolations of the current position; e.g. not related to any additional depth of search.

TABLE I

Depth	Both Right	TECH Only	CAPS-II Only	Both Wrong	Total
1	1	1	0	0	2
2	6	4	0	0	10
3	24	7	0	2	33
4	23	16	1	2	42
5	1	3	13	11	28
6	0	0	11	20	31
7	0	0	5	10	15
8	0	0	1	9	10
9	0	0	6	3	9
10	0	0	1	4	5
>10	0	0	0	11	11
TOTAL*	55	31	38	72	196

(*) These totals do not sum to 200, because problems on which partial credit was given are not included.

Table I shows the performance of TECH versus CAPS-II on individual problems as a function of the depth of the principal variation. The interesting thing about this table is the very pronounced skewing of results as a function of depth. TECH because of its exhaustive search does not miss any problems of depth 1 or 2. Then as the amount of work increases, the probability of TECH failing to solve a problem goes up steadily, until it can no longer solve any problems of depth 6 or greater in the five minutes allowed. On the other hand, CAPS-II misses a certain number of problems, at every depth. The percentage increases slightly as a function of depth, but the most important point to note is that CAPS-II, because of its approach, is able to solve some problems at every depth because the exponential explosion does not hurt it as much as a more conventionally designed program. It is reasonable to assume that as its perceptual facilities improve, CAPS will continue to increase the percentage it solves correctly at any depth. The conclusions associated with this table are probably the single most important ones in this paper.

TABLE II

Depth	Total	% Right/CAPS-II	% Right/CLass "A"
1	3	50	100
2	10	60	90
3	33	73	91
4	42	57	79
5	29	50	69
6	31	35	61
7	15	33	67
8	11	13	73
9	10	60	70
10	5	20	40
>10	11	0	18

It is interesting to contrast the results of CAPS-II versus TECH with a comparison of CAPS-II versus the Class "A" player as shown in Table II. Here the Class "A" human player very clearly excells the program in every category. This is, in our judgement, indicative of his greater understanding and flexibility of approach. However, the Class "A" player does not completely dominate CAPS-II's performance.

TABLE III

Depth	Both Right	Class "A" Only	CAPS-II Only	Both Wrong
1	1	1	0	0
2	6	3	0	1
3	21	9	3	0
4	20	13	4	5
5	8	12	6	2
6	6	13	5	7
7	2	8	3	2
8	1	6	0	3
9	5	2	1	2
10	0	2	1	3
>10	0	2	0	8

This can be seen in Table III which shows the comparative performance of the two on individual problems. Again the Class "A" player has the far superior performance. However, the next to last column shows that there were quite a few instances where CAPS-II was able to solve problems that the Class "A" player did not solve. This, in any case, serves to encourage us into believing that the basic approach has considerable potential, and will allow producing ever better programs as more and more of the details of tactical perception and analysis are built in.

An interesting test which helps to reveal some of CAPS-II's perceptual ability was performed on a sequence of 14 positions all of which were short mates. These positions had been selected because they could be solved rapidly, and the

time to solution varied inversely with the playing strength of human players who had solved them earlier. Using standard settings, CAPS-II solved 5 of the 14 positions correctly in 5 seconds or less. This appears to speak highly for CAPS-II's ability to diagnose and carry out simple attacks on the king. This in turn is due to the perceptual processing that the program engages in, which does a very good job of noticing powerful attacking moves. On the problem which was the single greatest discriminator of playing strength, CAPS-II achieved the highest possible rating, a grandmaster rating, by solving the problem in 5 seconds. Significantly, in solving this problem, the program made a wrong start on the correct idea, used the CAUSALITY FACILITY to find the correct implementation of the idea, raised the level of aspiration as an intermediate gain of a pawn was found, and deepened the solution to find the mate in three moves (all in five seconds).

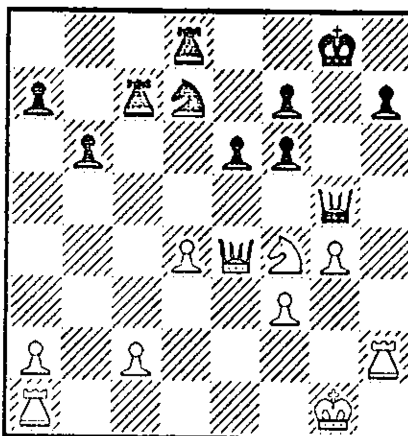


Figure 4

White to Play

The most difficult problem that CAPS-II has ever solved is shown in Figure 4. This is a famous combination stretching a full five moves for each side from the text position. The program looked at many possibilities, generating a tree of 897 nodes, but delivered the correct principal variation, letter-perfect as it is in the book. An investigation of the analysis tree showed it also correctly diagnosed all sub-variations. The correct move and essential sub-variations are:

1.#NxP!, PxN, 2.#QxKPch, K-R1, 3.#Q-K7!, Q-N1, 4.#RxPch!, QxR,
5.#QxRch winning

K-B1, 3.#Q-Q6ch, followed by QxR
K-N2, 3.#Q-K7ch, followed by QxR.

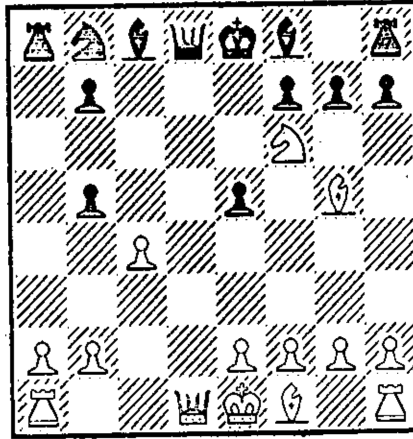


Figure 5

Black to Play

Another fine achievement was the solution of the problem in Figure 5. This is a famous opening trap, the correct move for Black being 1.-- QxN, because after 2. BxQ, B-N5ch wins. It took the program 12 CPU seconds and 49 nodes to discover the inadequacy of 1.-- PxN (because of 2. QxQch, ---, 3. BxPch, and 4. BxR) and then find the correct solution. Most of the credit for this must go to the CAUSALITY FACILITY which quickly pinpointed the cause of the loss after 1.-- PxB, and failing to find any meaningful alternatives at greater depth, returned the search to the top where the correct move was tried next and found to be good.

X. FUTURE CONSIDERATIONS

The following directions appear to be the most appropriate continuations of the research reported herein, and we are now engaged in developing these.

A. Complete Causality

The CAUSALITY FACILITY as it is presently constituted is only able to dissociate the blame for a set of consequences from the move made at the root of the sub-tree for which the consequences were collected. This means in effect that it can detect a "threat" that the move in question failed to meet and also did not cause. However, it is also important to be able to determine whether a move brought on a set of consequences that would not have been possible otherwise. This is the case of a move being "bad", and the Refutation Description gives the reason it is bad. To detect this case the following procedure is required.

When the CAUSALITY FACILITY cannot dissociate a set of consequences from a move, a null move is executed in place of the suspect move. Then the program tries to execute the Consequence Description in the sub-tree below this node. If the Consequence Description can be executed to produce a success value, then the suspect move was clearly not responsible. If the Consequence Description cannot be executed or a success value is not achievable, then the suspect move clearly

was responsible. The ability to make this decision correctly leads us directly to the next subject.

B. The Positing of Lemmas

There are certain invariants that tend to exist on the chess board from move to move. For instance, a certain check with the queen may continue to be bad because the checking square is adequately guarded. In their static analyses, programs manage to diagnose a high percentage of such cases correctly. However, because the static analysis will make some errors or interpret some unclear situation liberally (as it must), there will be some moves proposed which turn out to be bad. Further, since the static analysis always works in the same way (e.g. it does not learn), it will continue to propose these "bad" moves in similar board positions, and each time such a move is searched it will generate a sub-tree which in retrospect constitutes wasteful activity.

The first step toward preventing this activity is to detect that a particular move was in fact bad. This is what the completion of the CAUSALITY FACILITY offers. Once such a move has been detected, the essential environment in which its badness was detected must be specified. This consists of knowing the squares, pieces, and paths specified by the Consequence Description, and where each mentioned piece is in the position in which the bad move was made. It is now possible to posit a Lemma -- this being knowledge that a certain move will be bad as long as the described environment does not change. With this knowledge, any proposed move may be looked up in the Lemma file and if it has been previously cataloged, the program may determine if the current position contains any essential changes from the Lemma environment which might make the move succeed. It is important to note that should it be decided to try the move, and should it again fail, that it would now be possible to generalize the Lemma to include the union of the two environments, thus making it stronger. In a somewhat similar way, it is possible to generalize about the movements of a single piece, if more than one Lemma exists with respect to its moves.

Lest all this seem too beautiful and easy, we should point out that this implementation of Lemmas leaves out some conditions under which a bad move may become good. These conditions pertain to such things as pieces, which are presently defended and do not show up in the Refutation Description, becoming undefended and then becoming targets which would interfere with the successful execution of the refutation. Other possibilities also exist, but this subject is beyond the scope of the present paper.

C. Themes

It is possible that GOAL STATES are not a stringent enough concept to produce branching factors on the order of 1.5 or less in ordinary chess positions. The notion of Lemmas will help achieve this goal, however, further help is possible. When a move is selected for searching, it is because it was recommended by a Goal State and approved by EVALUATE. Once the searching of the sub-tree below this move begins in CAPS-II, this information is no longer used. It seems clear, that making

available to its sub-tree the reasons a move was selected for searching, will improve the utility of continuation moves that are to be searched. For instance, if a move was selected because it cleared a square for another piece, it seems reasonable that only moves which follow up on this clearance and otherwise new outstanding moves should be considered. It is possible for each reason that a move was generated, and for each redeeming feature that was noticed in EVALUATE, to provide a set of criteria for follow-up moves. These criteria should then be pushed down the tree, the union of the current set and the previously existing set being formed each time a move is selected for searching. While the set of meaningful follow-ups is an arbitrary notion, which can be expanded as a program matures, it is clear that this dissemination of information away from the root of a sub-tree will provide helpful guidance to a goal oriented problem solving mechanism.

REFERENCES

Berliner, H.J. (1973), "Some Neccessary Conditions for a Master Chess Program" **Proceedings of the 3rd International Joint Conference on Artificial Intelligence**, pp. 77-85, August 1973.

Berliner, H.J. (1974), "Chess as Problem Solving: The Development of a Tactics Analyzer", Ph. D. Dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., 1974.

Gillogly, J. J. (1972), "The Technology Chess Program", **Artificial Intelligence**, Vol. 3. (1972), 145-163.

Reinfeld, F. (1958), **Win at Chess**, Dover Books, 1958.