CHESS AS PROBLEM SOLVING: THE DEVELOPMENT OF A TACTICS ANALYZER

Hans J. Berliner

Carnegie-Mellon University

Prepared for:

Defense Advanced Research Projects Agency
Air Force Office of Scientific Research

March 1974

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-784881

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 74 - 1489	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CHESS AS PROBLEM SOLVING: THE DEVELOPMENT OF A TACTICS ANALYZER		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Hans J. Berliner		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A02466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE March, 1974 13. NUMBER OF PAGES 175
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research /NM 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis concerns itself with progress that has been made in the development of a better model of computer chess. We examine some faults of the popular model, including the Horizon Effect which is shown to cause arbitrary errors in program performance. Our effort is concentrated on 1) Developing a representation of chess knowledge which allows descriptions of board situations to be generated on demand for a variety of uses, 2) Limiting the use of tree searching to deal with well specified problems. We examine how to make a tree search converge naturally, without any externally specified limits.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (abstract cont.)

Several new tree searching heuristics are introduced. A program (CAPS-II) has been developed. It does not play well, but is able to do well certain tasks that current programs could never be extended to do. Among the more significant developments is a Causality Facility which is able to deal in a general manner with cause and effect issues and use results to narrow the search.

CHESS AS PROBLEM SOLVING:
THE DEVELOPMENT OF A TACTICS ANALYZER

Hans J. Berliner

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

March, 1974

This report reproduces a dissertation, titled as above, submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

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. This document has been approved for public release and sale; its distribution is unlimited.

ABSTRACT

This thesis concerns itself with progress that has been made in the development of a better model of computer chess. We consider the fact that chess programs have made almost no gain in strength, as measured on the human scale, in the period 1968 - 1973, as indicative that the popular model of computer chess is near the limits of its exploitability.

Some indication of why this could be so is provided in a chapter which discusses some very basic flaws in the current popular model of computer chess. Most serious of these is the Horizon Effect which is shown to cause arbitrary errors in the performance of any program employing a maximum depth in conjunction with a quiescence procedure.

We see the problem of computer chess as two-fold. There is the representation of chess as a knowledge domain. Such a representation allows:

1) Situations to be recognized

2) Problems to be stated

3) Results to be expressed, and

4) Intermediate knowledge to be saved.

A major portion of the thesis addresses itself to the development of a representation detailed enough to support the above requirements.

The second aspect of the problem is that a verification method is required to validate any proposed course of action. This is because it is seldom possible to establish a move (part of a course of action) as correct, based solely on the properties that the move may have. The usual verification method employed in computer chess is tree searching, and we do not differ here. However, in today's most successful programs tree searching is used both for discovery and verification, and we object to the former use. When tree searches are used for discovery, they tend to produce bushy (high branching factor) trees. This is what creates the exponential explosion which severely limits the depths to which current programs can effectively probe.

The thesis investigates the problems of how a more effective use of the representation avoids the need for much discovery tree searching. It also examines the problem of how to make a tree search converge naturally, without any externally specified limits on the depth and width of the search. Several new tree searching heuristics are introduced.

A program (CAPS-II) has been developed during the research reported herein. It is only concerned with the tactical (conservation of material) aspects of chess. It does not play very good chess by current computer chess standards. However, it is able to do well certain tasks that current programs can not do and could never be extended to do. This includes circumnavigating very large search spaces. CAPS-II regularly is allowed to go to a depth of 10 ply compared with the 5 ply for today's best programs. This is an increase of about three orders of magnitude in the size of the virtual search space, and the research points to ways for further extending this.

We have tested CAPS-II for its tactical ability, and it does quite well on sets of problems that are used to measure tactical chess know-how in humans. Since CAPS-II depends more on concepts and less on searching than today's programs, it is able to solve some problems with long principal variations while failing to solve others which could have been solved by shallow exhaustive searches.

Among the more significant developments in the research is a Causality Facility which is able to deal in a general manner with cause and effect issues and thus produce a radical narrowing of searches aimed at discovery of the remedy for a certain cause. Both causes and effects are described in the representation, and these descriptions can be used by move generators and for making deductions.

ACKNOWLEDGEMENT

The author wishes to acknowledge his indebtedness to the following persons:

To Allen Newell for his patient supervision of this thesis. His guidance proved invaluable both in formulating issues and in later appraising their outcomes.

To Herbert Simon for the many useful and thought provoking conversations that took place during the time that this thesis was being done.

To the other members of my committee, Raj Reddy and William Chase for reading the draft and making valuable suggestions.

To the members of the Computer Science Department for making CMU a place where doing research is a pleasure.

And last but far from least to my wife Araxie, whose never-failing support and understanding made difficult things much easier.

All the world's a stage, and all the men and women merely players --

Shakespeare -- As You Like It

TABLE OF CONTENTS

INTRODUCTION

Even before computers were commercially available, scientists were concerning themselves with the problems of computer chess. The first serious attempt to treat the problem of how a computer could play chess was made by Claude Shannon [Shannon, (1950)]. However, no programs that could play complete games of chess appeared until the late fifties [Kister, et. al. (1957), Newell, et. al. (1958), Bernstein, et.al. (1959)]. This was also the time period during which the field of Artificial Intelligence was beginning to emerge. However, these early programs played very weak chess and could only hope to beat beginners.

A historic correspondence chess match was held in the middle sixties between a computer program at Stanford University and a program at the Moscow Institute of Physics. Although this was the first occasion of computers competing against each other, the quality of play did not attract much attention. The first program to attract wide attention was that of Greenblatt [Greenblatt, (1967)]. It played in human tournaments and soon achieved a Class "C" rating, which is just below the median of the human scale for tournament chess players. However, in the intervening years no program has been able to achieve a stable rating above the "C" class.

This thesis concerns itself with a framework within which a Master level chess program could be built. We consider the fact that little advance in the strength of chess programs has been achieved since the 1968 Greeenblatt program as indicative that there is something wrong with the model of chess that is currently in popular use. In Chapter I we examine some of these problems. We show the need for programs to probe considerably deeper than they do now. We also show that the Horizon Effect is a limiting factor on the accuracy of evaluations, which can only be overcome by quiescing all parameters used in any terminal evaluation function.

These considerations lead us to aim for a model of chess which has a naturally converging tree search of unlimited depth. Our research is aimed at achieving this, and has two major directions. The first is to find additional tree control methods for converging the search. These include partitioning the search problem into several goal states, using new level of aspiration methods, and making comparisons among nodes in the tree in order to achieve termination criteria. Among the latter is a Causality Facility, which can make cause/effect decisions about a set of consequences that occurred in a given sub-tree.

The second method, on which the first partly depends, is to develop better representations of chess. This involves developing constructs that can reasonably project a future board state some three to seven ply away from the current position. We achieve this by noticing certain critical relations between pieces and squares. Pieces are then bound to those relations that must be maintained in order for the current situation to retain its stability.

We apply the above notions to the development of a Tactics Analyzer. A Tactics Analyzer is a program that is able to determine if in a given position a sequence of forcing moves exists that can change the material balance of the position. Such a program should be able to determine the material quiescence of any position. In our ultimate program, the Tactics Analyzer would be a basic unit for determining the tactical quiescence of any node. Thus it could be used for determining the tactical feasibility of any idea (move) suggested by a strategical agency.

A chess playing program that corresponds to the above notions has been developed during this research. Its name is CAPS-II (Chess As Problem Solving, Version 2). It is quite good at solving tactical middle-game problems, and can solve many deep problems that are out of reach of today's programs. It produces trees with a branching factor of 1.5 to 3.0, which diverge two to three times more slowly than today's standard trees. The program does not play a good complete game of chess, because there are still several tactical perceptual mechanisms which have not been programmed, and because it does not have the positional knowledge needed for evaluating positions and guiding end-game play. Since a chess program, like a chain, hangs by its weakest link, these factors currently limit the program's performance. However, the basic approach appears successful in that the above weaknesses can be overcome by additional programming without significantly affecting the program's branching factor. In fact several unimplemented ideas give promise of further significant reduction in the branching factor.

In reading this thesis, it may be useful to consider the following: Chapter I gives some basic insight into the generic problems of today's generation of chess programs and what would have to be done to overcome these in a significant way. Chapter II explains the structure of our program, and is essential for a deep understanding of the work reported herein. Chapter III explains the representation and gives examples of its use. Chapter IV performs a similar function for the tree searching method used here. Chapter V gives data deriving from large scale tests of the program. Chapter VI treats some meta issues and speculates about how certain issues raised by the research could be resolved.

Since the vocabulary of chess is somewhat specialized, and since we define many new constructs in this work, we have provided a glossary to aid the reader.

CHAPTER I

SOME NECESSARY CONDITIONS FOR A MASTER CHESS PROGRAM *

In this chapter we derive the motivation for the work reported in this thesis. But this is not a philosophical chapter. The Horizon Effect is examined for the first time in its full complexity. Other chronic problems plaguing the current generation of chess programs are also investigated. Deductions made about the necessary properties of a program that would not be afflicted with these problems lead to the formulation of the model, for which the work reported herein is a first implementation step.

A. THE HORIZON EFFECT

When branches in a tree search must be terminated prior to a legal termination point (according to the rules of the game), it is necessary to assign a value (an interim value other than win, lose, or draw) to the terminal node, which then allows comparison with other terminal nodes. This is usually done by invoking a static evaluation function. In games where a search to legal termination is not possible, no other recourse appears possible. An interesting phenomenon arises from the interaction of the artificial termination of the search and the fact that all the terms in the static evaluation function are evaluated at this terminal point. The result of this combination is that for the game playing program, reality exists in terms of the output of the static evaluation function, and ***anything that is not detectable at evaluation time does not exist*** as far as the program is concerned. This interesting fact is present in all tree searches in any chess program that we know of, and causes aberrations in program behavior.

First let us consider some salient aspects of a terminal evaluation. Who has the advantage in Figure 1.1? Merely counting the material on the board would result in an evaluation of "equal". However, one side is about to capture the other's queen and thus win the game. Since dramatic changes in the status quo are possible, we clearly cannot evaluate this position unless we can estimate the effect of these changes. This involves trying to construct a likely sequence of moves that would lead to a more stable situation. For this we also must know who moves first. What we are dealing with here is the concept of quiescence. A non-quiescent position cannot be evaluated properly without resorting to special approximations in order to attempt to find its quiescent value. Thus in Figure 1.1, it would be enough to know that the side-on-move can make a winning capture. This can be done by either extending the analysis one more ply or by doing a static analysis of the position considering whose turn to play it is. This example establishes the need for quiescence analysis.

However, the problem reaches much deeper than this. Let us say that a program does not do quiescence analysis, or does it improperly, so that it regards the position in Figure 1.1 as equal. Then a logical consequence of this would be that White to play in Figure 1.2 would play QxR in a depth one search. This is so since QxR results in an "equal" position at evaluation time, whereas other moves would not have such a "beneficial" effect. This is an elementary instance of the Horizon Effect. The Horizon Effect causes programs to make moves which are inferior, due to the fact that in some branch there exists a terminal position which was not quiesced or improperly quiesced.

(*) An earlier version of this chapter [Berliner, (1973)] constituted the major portion of a paper presented at the 3rd International Joint Conference on Artificial Intelligence, Stanford University, 1973.

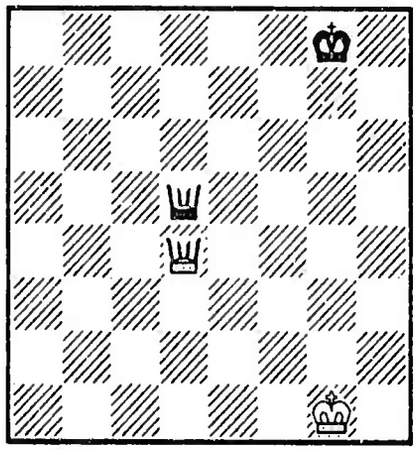

Figure 1.1

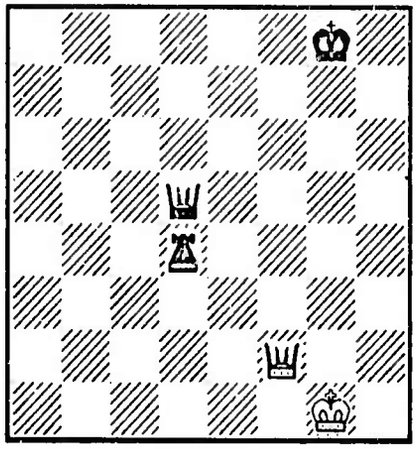

Figure 1.2

White to Play

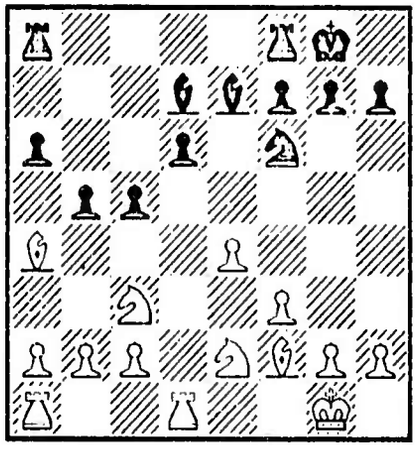

Figure 1.3

White to Play

The above example is due to lack of quiescence of the final position.

Examples of the The Horizon Effect have been observed by several researchers [Strachey (1952), Greenblatt (1967), Berliner (1970), Atkin et. al. (1971)] in game playing programs. However the complete phenomemon has never received a name in the literature nor have its causes and effects been properly cataloged. The regimen of insisting on a quiescence analysis (which as we have seen is necessary) results in today's programs in a ***two domain*** search. One domain is the regular search, and the other the quiescence search. The criteria for admitting moves to the search are different in the two domains. It is sometimes possible to shift the analysis of a particular problem in a position from the search domain into the quiescence domain. If this can be done in such a way that it will produce a superior evaluation while also producing an incorrect analysis (as seen by a chess player superior in strength to the program), the program will prefer the incorrect analysis and possibly make a bad move because of this. When the Horizon Effect results in creating diversions which ineffectively delay an unavoidable consequence or make an unachievable one appear achievable, we call it an instance of the Negative Horizon Effect. This is the phenomenon previously reported in the literature. It can best be shown by a typical example.

1. The Negative Horizon Effect

In Figure 1.3 it is White's turn to play, and for expository simplicity let us suppose the search is to be limited to three ply (it is relatively easy to construct examples at any given depth). What will happen in the above position is that the program will try to play 1. B-QN3 and after P-B5, 2. Anything, it is time to do a static evaluation. This usually consists of a material quiescence analysis, together with a calculation of other coefficients of an evaluation function. The material quiescence analysis could consist of trying all capture sequences and accepting the minimax value if it is an improvement for the side starting the sequence. Other quiescence procedures are also possible, but in essence they should yield the same value. Now at the end of the above 3-ply sequence, the program will come to the conclusion that it will lose the Bishop on N3, and will continue its search for something better. It will eventually come upon 1. P-K5 and recognize that if now PxB, then 2. PxN is good for White. Therefore it will consider as best for Black to play PxP, after which White plays 2. B-QN3. Since we are now at maximum depth of the regular search (the search horizon), this position will be evaluated using the standard procedure. The quiescence analysis will show that White has saved his Bishop since there is ***no sequence of captures*** which will win the Bishop. Alas, it is only after the next move that the program finds out that the ***non-capture*** threat of P-B5 has not been met by this diversion.It then looks for other ways of parting with material worth less than a Bishop in order to postpone the inevitable day when the Bishop will finally be trapped and captured. In this case 2. RxB would no doubt be tried next since after NxB, 3. B-QN3, "saving" the Bishop by giving up the Rook for the Black Bishop is preferred to losing it. We have seen programs indulge in this type of folly for five to six successive moves, resulting in going from a position in which they are well ahead to one in which they are hopelessly behind.

In this case the Horizon Effect is due to improper quiescence. The quiescence procedure used only includes captures but not moves that attack trapped pieces. Thus the improper quiescence leads the program to believe something exists that really does not.

A clever device to prevent this behavior was invented by Greeenblatt [Greenblatt (1967)] and is also used by the Northwestern University group [Atkin et. al. 1971]. This consists essentially of extending a new principal variation another two ply, to see whether the reason it was considered superior, will continue to obtain. In the above example, this will result in finding that the threat of P-B5 does not go away, and thus a potential sequence of blunders is averted. However extending a principal variation two ply can only discover whether a one move threat has or has not been dissapated. Threats requiring two or more moves can not be dealt with effectively in this manner. This is usually not noticed, since today's best programs perform at a level of skill where two move threats are rare and far from the major cause of concern for their developers. However, it is clear that the Horizon Effect can not be dealt with adequately by merely shifting the horizon.

2. The Positive Horizon Effect

The Positive Horizon Effect is different in that instead of trying to avert some unavoidable consequences, the program prematurely grabs at a consequence that can be imposed on an opponent later in a more effective form, or else tries to adjust the timing of moves so that some unachievable effect appears to have been achieved because of improper quiescence. This phenomenon has been largely overlooked in the literature, but is reported in a previous paper [Berliner (1970)]. Figure 1.4 shows an example of the Positive Horizon Effect where the program adjusts the timing of moves in order to take advantage of a lack of proper quiescence at maximum depth.

It is White to play and the search is being conducted to a depth of five ply. Since nothing tactical can be achieved; the program will try to find some way of improving its position. A natural candidate would be the move N-K5, which places the knight on an important central square and also increases White's control of space significantly. Unfortunately, this maneuver does not work as Black's reply, P-Q3 forces the knight back whenever it moves to K5. However, the program will find the following (ingenious!) way of getting the knight there anyway. It will play 1. PxP and after 1.--PxP, 2. RxR, BxR, 3. N-K5 maximum depth will have been reached. Since no captures result in any basic change in the position and since nothing is done to see whether the position of the knight on K5 is relatively permanent, White will have "achieved" his goal of significantly improving his position. Of course, after Black's next move when the horizon moves forward two ply, White will discover his error. However, in the mean time he has made a commiting decision (1. PxP) in the belief that it would lead somewhere, and it in fact did not. In this way, today's better programs teeter and totter along in a game.

An example of the Positive Horizon Effect that illustrates throwing away a positional advantage is shown in Figure 1.5. Here, if the evaluation function is aware of the beneficial effect of controlling an open file, and if the search is again being conducted to three ply, the most likely continuation will be 1. PxP ch, PxP, 2. R-KR1 with control of the open file and "some advantage". The fact that on the next move Black can answer R-KR1, after which White's advantage has largely evaporated is not recognized. Neither is the key fact that Black can do absolutely nothing to prevent White from opening the file whenever he likes (for human players there is the dictum "do not open a file until you are ready to use it"). However today's programs would almost certainly reject the correct 1. R-KR1 since

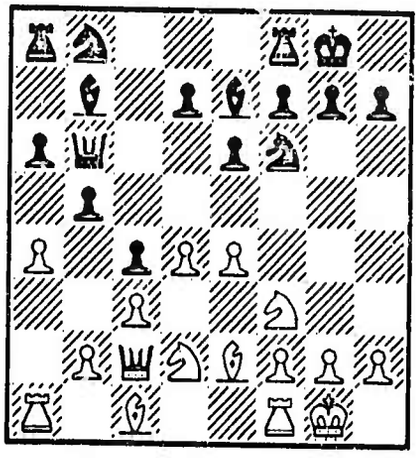

Figure 1.4

White to Play

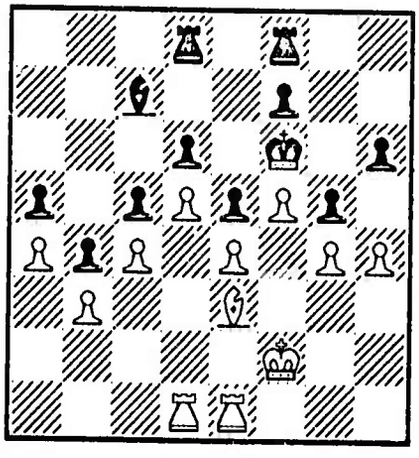

Figure 1.5

White to Play

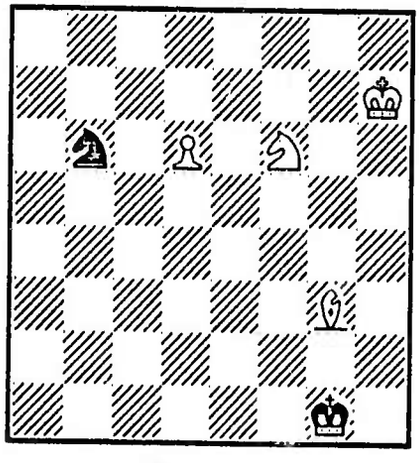

Figure 1.6

White to Play

after Black plays R-KR1 and White plays 2. PxPch, it is time to invoke the quiescence procedure which produces PxP. Now in contrast to the earlier variation, control of the open file is disputed and the evaluation will not be as favorable for White. Nor would it be if any 2nd move other than 2. PxPch were played. Clearly, a program could recognize the value of playing 1. R-KR1 before 1. PxP ch, only if it were secure in the knowledge that the file can be opened at a later time by PxP and that if Black plays PxP, he will merely incur an equally difficult problem in defence of the KRP as he has now in defence of the KR-file. In fact having once played 1. R-KR1 and getting the answer R-KR1, a program that has reasoned thus far should have little difficulty in now playing 2. R-R2 since opening the file at the present moment is not advantageous and making room for the other Rook could help. It should be noted that incorporating the human players' dictum appears extremely difficult as the issue of "ready to use it" is one requiring dynamic judgements, in which even good human players make mistakes at times.

Another example of grabbing too soon at an advantage (this time a material advantage) can be seen in Figure 1.6. Here it is White to play and the search is again to a depth of three ply. The program notices that it can play 1. P-Q7 and if Black does not now play NxP, 2. NxN, then it would get a new Queen. It sees that in this way it can increase it's material superiority. It may or may not notice that it will then have to face the formidable task of mating with a Bishop and Knight. The interesting thing about this position is that the maneuver 1. B-K5 followed by 2. B-Q4 cannot be prevented and results in forcing the Pawn through to a Queen without letting Black give up the Knight for it, thus simplifying the win greatly. Here the important point is that there is a consequence on the horizon, and the program insists on realizing it within the horizon of the search as otherwise ***it does not exist.*** As a result, a consequence which could have turned out to be very beneficial, turns out to only have a small benefit. If the issue of whether or not the pawn is promotable and what the largest penalty that can be exacted for it, were separated from the function that counts material in all terminal positions then it would be possible to treat this problem separately and possibly come to the correct conclusion with respect to it. Thus the promotable pawn could be regarded like money in the bank, and would not be spent until a survey of various ways of spending it was complete. By concentrating on detected issues it should be possible to avoid the premature throwing away of winning advantages that one frequently finds in today's programs. In fact, the main reason for the demonstrated lack of tolerance of complexity of today's chess programs is that their evaluation function insists on maximizing, in terms of a preconceived set of evaluation terms, anything that it detects within the search horizon, and thus all too frequently destroys an advantageous situation before it really has a chance to bloom.

3. What can be done about the Horizon Effect

It is important to distinguish between the Positive and Negative Horizon Effects. In both cases, a phenomenon is detected during the regular search, and an inappropriate response to this phenomenon is then made because of the analysis in the quiescence or terminal evaluation phase. In the Negative Horizon Effect, the response consists of the program believing that the undesirable effect has gone away or been gotten rid of at a lower cost than the original impact of the effect. This illusion is caused by the program manipulating the timing of moves so that a certain position will occur at the search horizon. The Negative Horizon Effect can

be gotten rid of if the regimen for the regular search were the same as the regimen for the quiescence search. This is the approach used in this research. Another possibility would be to maintain descriptions of all undesirable effects encountered, and then include investigations of these described effects in the quiescence analysis.

In the Positive Horizon Effect, two things can happen. A desirable consequence can be detected, and the program then fools itself into thinking that this consequence is achievable (again by manipulating the timing of moves), even though it really is not. This was the case in Figure 1.4. The other situation is where the detected effect is realizable, but in a much better form than what the program discovers (Figures 1.5 and 1.6). In such cases, it is usually not possible to blame the tree search. Rather, this is a function of knowledge; knowing that the detected phenomenon may be realizable in a better form, and then applying techniques (which may not be tree searching) to determining if this is possible. For example in Figure 1.6, a five ply search could not be blamed for not finding the right solution. However, once the problem of finding the maximum gain for promoting the passed pawn is detected, a method must exist for dealing with this. When tree searching is not adequate, some general knowledge may suffice. For instance, knowing that the Black knight in Figure 1.6 can be attacked could be sufficient. In this last example, we do not wish to imply that such a paradigm will always react optimally to every position. Rather it has the facilities for reacting optimally to phenomena detected during the search. The phenomena must be detected,and describable. Further the methods for finding the optimum are subject to the same risks that all problem solving processes are; however, such methods must be made part of any proper quiescence analysis or terminal evaluation.

B. NEED FOR A GLOBAL STRATEGY

Another basic problem, the need for a global strategy, is shown in Figure 1.7. Here White is to play and every one of today's programs would conduct a 5-ply search and then play 1. K-K3. A summary of its findings during this tree search might run a follows: it decided that P-B7 would lose the pawn to K-K2, and therefore decided to move the King to the most central location available (this is a quantity recognized by the evaluation function). On the next move, having already achieved its "optimum" position, the program would be faced with a problem that all hill climbers face when they reach the top: How to back down as little as possible? Accordingly there would occur either K-Q2 or K-B3. The point of this whole example is to show the hopeless hill-climbing characteristics of current program designs. In the given position, even a poor human player would recognize that there is nothing to be gained by the above maneuvers. The real problem is that today's programs mix their strategical and tactical objectives during the search. Thus the above position could be handled effectively if a tactical search were first done and this came to the conclusion that P-B7 only resulted in losing the Pawn. There being no other tactical tries, control would then revert to a strategical module which would try to improve the position of any and all pieces. Since, in this simplified situation, we only have the King as a candidate, the next step would be to try to find an optimum or near optimum position for the King and determine if it could get there. Here we must not rely solely on a static, preconceived notion of centrality, although that certainly is a part of the picture, but more importantly we seek a functional optimum. This can be found by noting that the Black KNP and KP are not defended by Pawns and could possibly be attacked by the King, and also that our own KBP could possibly benefit from having our King near it. Next, a null move analysis

could be carried out, consisting of moving the White King around without looking at intervening moves, to see if we can find a path to any of the desired squares. This will eventually yield the correct idea of infiltrating with the White King via QR3, which wins easily. Admittedly the control structure that could evoke such behavior would present some problems. Most of the problems in chess are tactical (immediate) problems and for this reason, the lack of global ideas is frequently obscured in today's programs. However, it is absolutely necessary to be able to generate global goals in order to avoid hill climbing behavior.

We have touched above only on the relatively simple problem of finding the correct way to proceed. A far more difficult problem, which would also have to be faced by the Master strength program, is to judge whether the position can be won or is a draw. A simple "pawn ahead" judgement is not enough. There may be other endgames from which to choose, in which the program is also a pawn ahead. In the position being discussed, for instance, if a further White Pawn were at QN4, and a Black Pawn at its QN4, the position would be a draw. Clearly dynamic judgements of this type are absolutely necessary.

C. CALCULATION OF LONG TACTICAL SEQUENCES

In Figure 1.8, we see a much better understood problem than any of the above. It is the problem of calculating in depth. Here White to play can execute a mating combination requiring an initial Queen sacrifice and nine further moves, a total of 19 ply as follows: 1. Q-R5ch, NxQ, 2. PxPch, K-N3, 3. B-B2ch, K-N4, 4. R-B5ch, K-N3, 5. R-B6ch, K-N4, 6. R-N6ch, K-R5, 7. R-K4ch, N-B5, 8. RxNch, K-R4, 9. P-N3, Any, 10. R-R4 mate. This combination was played by a former World Champion while playing a total of 20 games simultaneously. The reason no program that looks at 10 or more alternatives at every node can play the correct move is that the principal variation to justify the initial queen sacrifice extends much, much further than the 5-ply depth that is about all that is possible with a program that gets buried in the exponential explosion of investigating 10 sprouts from every node. Now it is quite possible to play Master level chess without playing such long combinations, but it is necessary to be able to defend oneself against such long sequences. In the author's experience, one must at least once a game be able to look 14 or more ply ahead. As far as the above example goes, we believe that 99% of all Masters would solve it as well as a high percentage of Experts and Class A players. What is really difficult about the example is not the simple unravelling of the main line, which having few branches is fairly linear, but the conception of the position, and that such a solution involving chasing the King up the board might exist in it.

One could argue that just because good players can solve such problems, this does not show the requirement for the program to see to such depths in order to play at the Master level. What this would mean is that the program would have to rely almost exclusively on static, non-tree-search computations for its moves. But we have already shown in Figures 1.4 and 1.5 that static notions must be combined with dynamic tests in depth in order to yield correct results. So a program that could not look 10 ply ahead would be subject to any 10-ply deep threat that comes along. Even though the main thrust of most such threats could no doubt be muted, it would be inevitable that some concession would have to be made. This type of thrust and parry is at the heart of Master play. Even more importantly, a program that cannot look 10 ply ahead could never conceive a five move threat of its own which is dependent on adverse action. The evidence is quite overwhelming.

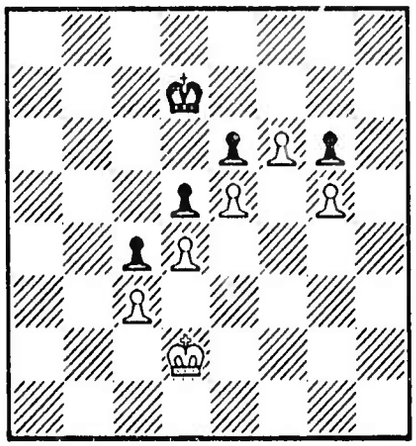

Figure 1.7

White to Play

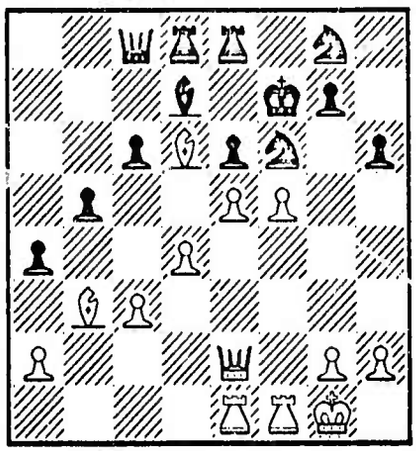

Figure 1.8

White to Play

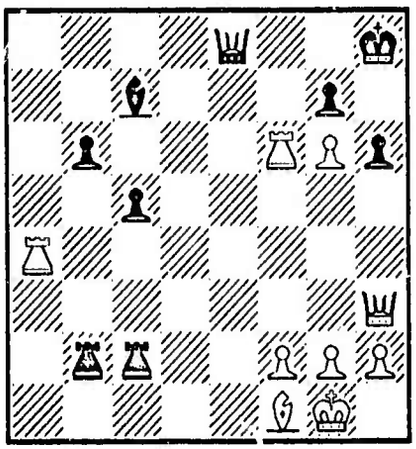

Figure 1.9

White to Play

D. THE UNRELIABILITY OF STATIC ANALYSIS

1. The Safety of Pieces

Another interesting phenomenon, that of reality or illusion, that afflicts all of today's best programs can be seen in Figure 1.9. Here it is White's turn to play. The first thing that the evaluation function will discover is that White has both of his Rooks "en prise" (capturable by the opponent under favorable conditions). If this position has occured at some node which is eligible for sprouting, then moves that move either of the rooks to a "safer" place will receive good recommendations. If the node is a terminal node, then it will be considered as not satisfactory for White, as it is presumed that at least one of the Rooks will be lost. In actuality, neither of the Rooks is in danger. If Black plays QxR then R-B8 mate, and if PxR, then QxPch, K-N1, Q-R7ch, K-B1, P-N7ch followed by P-N8=Qch wins quickly. Even stranger is the fact that if this position occurs somewhere in the tree below the top node, and if, say, two ply earlier White had played RxP(KB6) as a sacrifice which it turns out could not have been accepted, then in today's programs there would now be no knowledge of the sacrifice at KB6 when the position is tendered for evaluation two ply later. Rather the Rook would be considered en prise. Indirect defences of this type are seen all the time in Master chess. Clearly, if a program aspires to this level it must be able to handle such problems. Part of the solution consists in noting the functional overloading of the pieces that are thought of as doing the capturing. Here the Black Queen is guarding a check on the back rank apart from attacking the White Rook. Also the Black KNP is guarding a Pawn and a check, while attacking the White Rook. However this is not enough, since it is quite possible that the checks that are being defended against are quite harmless, and it would be folly to try to determine, without further searching, the exact potency of every check on the board.

2. The Value of Material can depend on Dynamic Considerations

Another problem, that of dynamic evaluation of material, is depicted in Figure 1.10. Here with either side to play, White's pawn cannot be stopped from queening, while Black's pawns are going nowhere fast. Yet there is no doubt that every one of today's programs, if playing Black would refuse a draw in this position, and it is also very clear that only a very weak human player would offer a draw with White. The programs' rationale is that three passed pawns are better than one. The problem here is one of recognizing the dynamic potential of the White passed pawn which cannot be caught. It is true that in this case the job can be done statically be merely noting the distances of the White Pawn and the Black King from the queening square. However, if the Black Pawns were all advanced three squares, the computation would have to be done dynamically, since there is a possibility they may arrive first. Similar dynamic ideas, which no program can at present handle well, are the notion of a defenceless King by reason of no surrounding men of his own to help defend him, and the notion of cooperation among various men rather than only assessing the goodness of their individual positions. Such notions require dynamic exploration to determine the degree of their applicability in a given position. However, in a program where terminal evaluation must be done very quickly because of the large number of nodes that must be evaluated, such luxuries are not possible. We are here directly confronted with a basic limitation of the generate and test approach. When it does not allow enough time to do a detailed evaluation of the nodes visited.

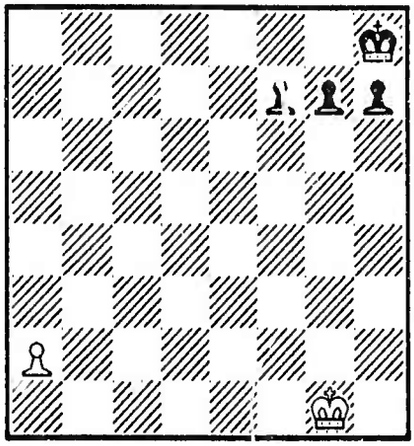

Figure 1.10

Either Side to Play

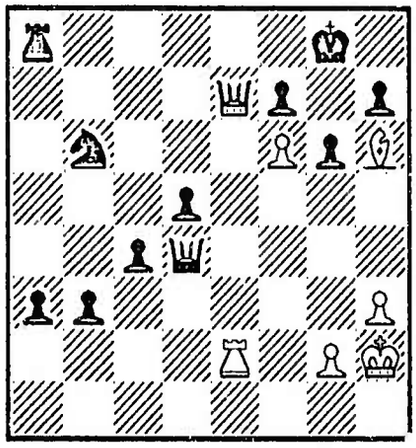

Figure 1.11

Black to Play

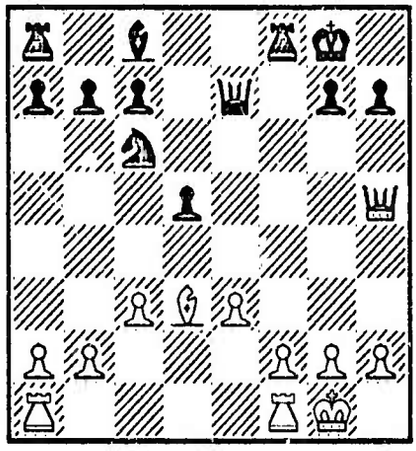

Figure 1.12

White to Play

E. EFFICIENCY ISSUES

Our last two examples deal with situations that present-day programs can handle. However, the method by which they do this is terribly inefficient and could not be used if one wanted to do tree searches which could extend even a little deeper than the current five ply. The first of these problems is the problem of defence. It is relatively easy to recognize attacks and develop criteria for judging the value of most attacks. However, this is not so with defence. The problem is that in order for a defence to exist, a threat must first be known. All threats are not of the simple type such as threatening a capture. It is the deep threat, the effect of which shows up only in the backed-up value of the current variation, that is not easy to counter. This would be especially true when the initial move of such a threat does not accomplish anything obvious. In such cases present technology is able to only determine the magnitude of the threat's effect. Figure 1.11 shows a position of this type. Here it is Black's turn to play and the search is being conducted to a depth of five ply. If Black plays a normal aggressive move such as 1.-- P-R7, he will find that after 2. Q-K8 ch, RxQ, 3. RxR he is mated. The search will then eventually revert to the point where Black played P-R7. Now in most of today's programs we would be armed with the killer heuristic (which says that against any new proposed move try the "killer" Q-K8ch first). This would indeed result in the efficient dismissal of the next 15 or so moves likely to be tested. However the fact remains that each of these alternatives is being served up in a generate and test mode, and the program can consider itself fortunate if it discovers the only defence (Q-K5) before it has exhausted half the legal moves. A much better way of handling this problem through the CAUSALITY FACILITY is demonstrated in Chapter III.

Our final example in Figure 1.12 shows another subtle consideration. In this position with White to play, programs that look five ply deep have an excellent chance of finding the mate in three moves: 1. BxPch, K-R1, 2. B-N6ch, K-N1, 3. Q-R7 mate. If such a program, due to the fact that White is behind in material, were only to look at captures of pieces of greater or equal value to the current deficit, and checks (an assumption which requires some preprocessing) and to stop at five ply depth (for which it would be difficult to establish a logical reason), there would still be about 100 bottom positions to examine before the mate is found. Here any tournament caliber human player would recognize the situation immediately as one of a set of Queen and Bishop mates. He would only have to determine the functional need to guard the King escape square at KB2, to determine what the correct sequence is and that it does lead to mate. The critical thing here is not that a program couldn't find the mate once the diagrammed position is reached, but that in advanced chess play such situations occur frequently in deep parts of a tree as a reason why some other move fails. If a program spends 100 nodes investigating such a well known pattern, then there is a definite limit on the amount of work the program can be expected to do. The answer here quite obviously is to have a repetoire of frequently occurring patterns available to the program together with some guidance to determine the exact applicability of any particular pattern. In the above case, recognition of the Queen and Bishop functionally bearing on the undefended KR7 square, together with the position of the Black King hemmed in by some of its own pieces is the basic pattern. The dynamic analysis reveals that the King could escape over KB2 if this were not kept under continued guard. With these constraints, the number of variations to be examined are very few.

F. SOME CONCLUSIONS

Let us examine some potential models of computer chess. All the complete models are clearly too time or space consuming. Therefore, the most reasonable course appears to be to rely upon models that construct trees of possibilities but with some limitations imposed upon the growth of the trees. Now depending upon how we define these limits, we have a tractable problem. The real question, and that addressed in this chapter, is how these limits can be defined and implemented in order to include the range of performance exhibited by chess masters while still keeping the problem tractable.

Let us summarize the requirements noted earlier:

1) In Figures 1.1 to 1.6 we saw the Horizon Effect in operation. We also saw that the two-ply extension of each new principal variation is only a stop-gap measure, which prevents one move debacles (anyone who doesn't believe this is invited to try Figure 1.13 out on his program). What can be done about the Horizon Effect? Clearly the problem is due to the fact that some term in the static evaluation function is evaluated "prematurely". Prematurely here means that a noticable change in the value of the term can be forced, without any compensatory change in any other term(s). From this, one can deduce that there can be no arbitrary depth limit imposed on the search. The decision as to whether to terminate the search at a node or continue, has to be a function of the information that exists at that node and how this relates to the quiescence of each and every term in the evaluation function. For instance, if we have an evaluation function that would consider it bad to have a bishop blocked in by its own pawns, then some effort must be expended to determine the permanency of such situations. In general what is required is a procedure to determine the quiescence of every term in the evaluation function and in cases of non-quiescence, a procedure for generating moves or applying some static means of reaching a quiescence decision. This should not be construed as meaning that perfect knowledge of the future status of each parameter in the evaluation function is required. In fact some practical maximum depth or time limits must exist. Thus only a finite set of resourses can be expended to determine the true future status, and some controlled error will no doubt have to be tolerated. However, the resulting error by this method should be orders of magnitude smaller (a so-called judgement error) than the errors produced currently by the Horizon Effect. In present day programs, quiescence is pursued only for the material parameter. And even this frequently does not work out satisfactorily, since usually only captures are considered, while forks, mate threats, etc. are ignored.

2) From Figure 1.7 we saw the need for having global goals and being able to determine something about the feasibility of such proposed goals. This may involve procedures of considerable complexity in order to answer basic questions about the value of any node. All of which adds to the potential evaluation time required at a node.

3) From Figure 1.8, we saw that the program must on occasion be able to calculate precise variations to a depth of 14-ply and possibly more. This in no way means that every move should be calculated to this depth nor that when a move is, that every branch would go to this depth also. However, the basic facility to allow probing to at least this depth must exist.

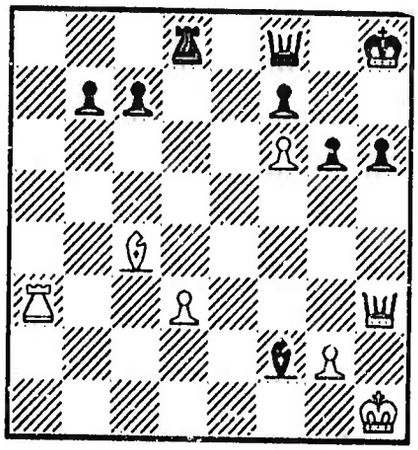

Figure 1.13

Black to Play

4) From Figure 1.9 we saw the need to diagnose certain dynamic properties of positions, and the requirement to communicate such data to other nodes in the tree. This need exists in order to avoid faulty interpretation and the necessity of otherwise "discovering America" over and over again.

One key to detecting that something may not be as it appears statically, is the use of a functional analysis. In Figure 1.9, the initial indication that neither of the Rooks is capturable is that each of their attackers is also defending something of importance. Sometimes it is possible to resolve such function conflicts statically by noting that another piece can assume the required functional role without itself becoming overburdened. When this is not possible, the validity of a potential function conflict must be established dynamically by tree searching. In later chapters of this thesis, we examine the notion of lemmas. Lemmas store intermediate information that has been discovered, and can be used to avoid the above problems.

5) From the defence problem in Figure 1.11, we see a need for communication within a search tree. A proper description of a set of undesirable consequences can save tremendous effort in finding problem solutions if such exist, or moving on to more fruitful endeavors if not. The adequacy of the descriptive language is important as it must be used to test whether the set of consequences were caused by the latest move, and to provide an input to move generators that could find an appropriate answer to the problem. For this purpose, functional relations which describe attacks that occured, and path information which describes paths traversed by moving pieces and paths over which threats occurred, appear to be among the required elements of the language.

6) The functional relations mentioned in the previous examples are in a sense patterns involving two pieces or a piece and a square. Certain clues can be gained by searching these patterns when they focus about a common square or piece. However, from the example of Figure 1.12 we can see the need for a still higher level of pattern abstraction. Here we are looking for groups of pieces which form a pattern around some interesting focus. In the example cited, the KR7 square with the White Queen and Bishop attacking it, and the Black King are the focal points which should suffice to index into the correct pattern, which will then produce a pointer to a routine for deciding if we are confronted with an exploitable instance of the pattern in question.

Above, we have assembled the beginning of a set of requirements for a program that could have the power to play Master level chess. It does not take long to dismiss the possibility of extending the current generation of chess program to meet the above requirements. It is quite enough to realize that such a program requires about a factor of 20 of additional time for each additional two ply of depth that it searches.

G. STRUCTURAL REQUIREMENTS FOR A PROGRAM

In 1958 Newell, Simon, and Shaw [Newell, et. al. (1958)], argued that "As analysis deepens, greater computing effort per position soon pays for itself, since it slows the growth in number of positions to be considered". This is well substantiated in the ACM tournaments which have convincingly shown the superiority of programs that search a subset of legal moves and evaluate a moderate amount, over programs that search all

legal moves and evaluate little. Clearly it is time to move again, and more substantially, in the direction of more evaluation and less search. The requirements demonstrated here show a need to do possibly ten or more times as much processing at a node than is currently done. This means that, for equivalent computing power, we are faced with generating trees of at most 5000 nodes distributed throughout the search space. The Greenblatt and Northwestern University programs have an effective branching factor of 5 to 6 (where number of bottom nodes = BF^{DEPTH}). If it is assumed that the search is limited to 14-ply, then the branching factor must be less than 1.9, if we are to stay within 5000 nodes.

Actually this is a meaningful measure only for trees which have a maximum depth. In order for a tree of no maximum depth to converge, a necessary and sufficient condition is that for any arbitrary node Σ (i P[i]) < 1 (where P[i] is the probability of i sprouts). Clearly the less Σ (i P[i]) is, the more rapidly the tree will converge. It appears resonable that tree convergence could be achieved without arbitrary bounds, if it were possible to do meaningful comparisons of the state of any node with the states of earlier nodes in the tree branch being investigated. Comparisons could involve how earlier expectations are holding up, and whether moves that are eligible for testing have appropriate thematic relationships to what has gone before. The number of such comparisons grows linearly with depth thus providing ever more conditions for stopping the search or not investigating an arbitrary move.

To guide the search mechanisms are needed which can at linear cost provide analysis at a node so that the exponential cost of discovery and/or verification due to tree searching is drastically reduced. It appears reasonable that the more powerful (in the sense of greater depth) the prediction mechanism, the better the effect on program performance. Here the functional analysis and pattern recognition mentioned earlier clearly are destined to play a part, with the former being an essential element of the latter. Also the communication of defensive requirements appears vitally necessary. In fact since dissatisfaction with a result is a relative matter, one might consider using backed-up descriptions to discover ways of heightening the success of whatever is being attempted at present.

Lastly, one can see the overriding importance of quiescence of concepts being used in the evaluation procedure. The evidence is quite overwhelming that the attempt to drive all evaluations into a quiescent state should be the major force that determines the shape of the tree. Thus, while today's programs use up nearly all their time trying to assure tactical quiescence, this will now have to be done by less complete methods in order to make way for the additional facilities required. It is interesting to compare this derived role of quiescence as the main guiding force, with the control structure of the 1958 Newell, Simon and Shaw program [Newell, et.al. (1958)] which was apparently derived from a concern with human behavior.

The requirements derived in this chpater appear to be necessary for Master chess. However, they are almost certainly not sufficient. Masters know a great deal of chess knowledge which has as yet not been encoded in any program, and will probably have to be placed in a long-term memory for occasional reference. We have avoided discussing what a minimum quantity of such data might look like, since until the necessary mechanisms for its use are in place, so that it would be possible to experiment, there would be little scientific validity in such speculation. There is also the problem of doing at least some learning in order to avoid repeating obvious errors in

identical situations. However, an organization which takes account of the conditions noted here is almost certainly necessary to make significant progress beyond the present state of the art, and its implementation should strive to be extendable to the problems of learning and further pattern encodings, as these prove necessary. In the immediate future, the major problem appears to be how to produce a search of the economy of that proposed, while retaining at least the same reliability as evidenced by today's programs which use a more complete search strategy.

CHAPTER II

STRUCTURE OF THE PROGRAM

A. DESIGN CONSIDERATIONS

In Chapter I we have shown that programs that pay costs that are an exponential function of depth for discovering interesting things about a chess position will never attain the Master level of play. Therefore, if a program is ultimately to aspire to this level of play it must use more effective methods. These methods must be essentially linear in cost as a function of depth. By operating on a given position they should give clues as to what are the most worthwhile things to consider doing and what are the essential features that will be the key to evaluating this position. The type of complex inter-relations that we mean here are usually refered to as patterns. But patterns can be composed of very few elements or of many, and even very simple patterns have their information to deliver. Consider the domain of written language. Here a vertical line could be considered a pattern element. This element is encompassed in some of the letters, and letters are essential elements of words which are essential elements of phrases. Thus, it is to be expected that a viable pattern scheme for chess should have basic elements that can be combined recursively to make more powerful patterns.

The fact that patterns can be composed suggests that best results are to be obtained by designing the information structure for a chess program from the bottom up. Thus what are needed are some basic elements of patterns which can be used to form ever higher level patterns. Based upon insights gained as a subject in chess perception experiments [Chase and Simon, (1973a, 1973b)] in the Psychology Department at CMU, as well as introspective evidence, the following basic elements were chosen. The most basic element is the legal move since this is really what chess is all about and cannot be dismissed. Legal moves and moves that would be legal if one or two things about the current position were changed can be represented as bearing relations. By considering certain points on the board more important than others, it is possible to consider legal moves dealing with those points as more important than legal moves which do not deal with any such point. Thus arises the notion of a functional relation. A functional relation assigns a given piece a specific meaningful role with respect to another piece or square. This is the second level building block in our hierarchy of information structures. Relations of this type show up again and again in chess perceptual experiments to which we had access. Apart from this, our earlier program [Berliner, (1970)] was continually confronted with questions of which piece was responsible for which actions on the board. Since actions were usually only summarized, there was no longer any way of knowing the agencies responsible for the final output. This made it impossible to differentiate such situations as two pieces being defended by different pieces or by the same piece. Clearly, when both defensive tasks are being performed by a single piece there is a possibility of overburdening that piece, which would not exist if the two defences were independent. The need to be able to get at such information, long after the initial evaluation that a piece is defended has taken place, was also a major consideration for a data structure that preserves such relations.

Another important aspect in the design is the accuracy issue. In chess the only sure way to establish truth is to do a COMPLETE tree search. Short of that, one can attempt many different methods of ascertaining the truth about a set of features on the

chess board, but one must bear in mind that any such method is approximate and subject to error. However, even in the domain of approximations there are methods that are more approximate than others. Based upon the 1958 observation of Newell, et. al. that good exploitation of information at a node slows down the exponential growth of searching, it appeared very worthwhile to expend the maximum effort in trying to ascertain the truth about each issue dealt with. Such an effort would be expended even when this meant considerably increasing the complexity of the program. Furthermore, even very great static accuracy is no guarantee that these results would not be later upset by something that a more knowlegable agency in the program finds out. There certainly exist situations where almost any static analysis will fail. In such cases, there could be a series of inversions of truth, where each successive court of appeal to the real truth reverses the findings of the previous court. Finally, the actual truth could only be found out by a complete tree search, which would in most instances be unrealizable.

Another consideration is the executability of the ultimate chess program. To determine the capability of a program, it is more or less necessary to test it in an environment similar to that in which humans play chess. The playing demonstration is usually done in a tournament of either other programs or a mix of humans and programs. Therefore it is important in choosing the representation to consider that a program will be asked to produce chess moves at a rate of approximately three minutes per move. One of our design objectives has been to generate trees with between 200 to 500 nodes, which means that it would be possible to spend approximately one-half second per node. Since the basic design philosophy also called for computing as much as is reasonably possible before leaving a node, the efficiency of performing certain basic operations becomes paramount in the light of the three-minutes-per-move constraint.

For this reason the language Lisp [McCarthy, et. al., (1965)] which is frequently chosen for projects in artificial intelligence was ruled out, since the computing cost of searching lists for every datum is prohibitive at this stage of the hardware art. This does not, however, rule out the need for certain higher level data to exist in list form. But, there appeared to be many lower level operations for which a more hard and fast approach would work, and it was here that great efficiency was desired. Therefore, a language was required which could take full advantage of the set manipulation instructions in the hardware (PDP-10). The language chosen was Bliss [Wulf, et. al., (1971)], which is an ALGOL-like implementation language, developed at Carnegie-Mellon University, which makes use of almost all machine commands and produces highly optimized code. Thus the members of a particular set can be represented as "on" bits in a bit vector. It is then possible to find the set intersection of two sets by simply "anding" their vectors. The other set theoretic operations are also simply performed. It is easy to see that this method of data analysis for frequently used data, is much faster than the similar list processing technique. These techniques are used for all lower level data.

Given the above considerations, we can begin to lay out a concrete design. First consider a set of bearing relationships (defined in detail below) which specify certain attributes about moves and potential captures that are possible on the board at present or would be if some minimal change occured. These bearing relations between pieces and squares are stored in tables for quick reference. As the analysis of a given position proceeds, certain of these relations are picked out as being more important

than others -- those that are essential to the stability or instability that characterizes the current chess position on the board. Among the relations that are interesting are those that defend or attack a piece, defend or attack an important square, deny access to a square for a useful move, etc.

The restriction for information to be abstracted above the bearing relationship level is for it to be "meaningful". For this an arbitrating process must exist to decide which relations are meaningful and which are not. Such a process must decide, for instance, what is needed to defend a piece. If this piece is not attacked, then clearly nothing is needed to defend it, while if it is multiply attacked, several pieces may be needed. Or it may be possible to determine that a major fraction of the piece will be lost regardless of how many times it is defended, as would be the case if a pawn attacked a queen. We call the relations that have meaning, in the above sense, "functions", after Newell and Simon [Newell and Simon, (1972)]. In this way functions represent the fabric of the chess position.

The board analysis process can be extended to include other things besides the safety of pieces. This involves such things as unoccupied squares from which double attacks can be made. When the function assignment phase is completed, a description of the board noting certain basic situations (the piece on square X is en prise) and inter-dependencies (piece Y defends both square A and B) exists. From this description, it is then possible to detect potential moves that have utility for the moving side. The next step is to evaluate these moves in the description environment, which allows detecting not only useful properties of the suggested move, but any function conflicts which it raises. These function conflicts could result in a move, which at first glance appears foolish (it puts a piece en prise), being evaluated as potentially very good. This is when a piece that apparently could capture it, is otherwise committed.

Having found a set of potentially useful moves, we want to try them in some order according to their potential. This is done in the usual tree search paradigm. However, we wish to be more sensitive to issues that come up than the current generation of programs. Specifically, if something unfavorable happens, we want to be able to decide if this was due to some blunder just tried in the analysis, or if it was unavoidable at this node. This requires a facility which can make cause/effect decisions by comparing descriptions of a move with descriptions of what happened. This in turn requires facilities which can accumulate deep descriptions of consequences. When a causality decision is made, we want to address all the resources of the program to solving the detected problem, and abandon the searching of moves which may at one time have appeared appropriate, but no longer appear so. Finally, we want to have a level of aspiration for the program. This level is to define the expected value of the top position in the search tree. When a result that is clearly better or worse than this is found, we want to be able to readjust the expectation and possibly re-examine the analysis to see if it can be further improved.

The following sections describe how this program structure is implemented. We take up first the various levels of the data structure, how its elements are computed, and how elements at higher levels are dependent on those at lower levels. Next, we discuss move generation and the static evaluation procedure for proposed moves and for positions. The last portion of the chapter deals with the control structure for the program. In following the descriptions in this chapter it will be useful to refer to Figure 2.1 which is a rough flow diagram of the computational sequence. Any new position

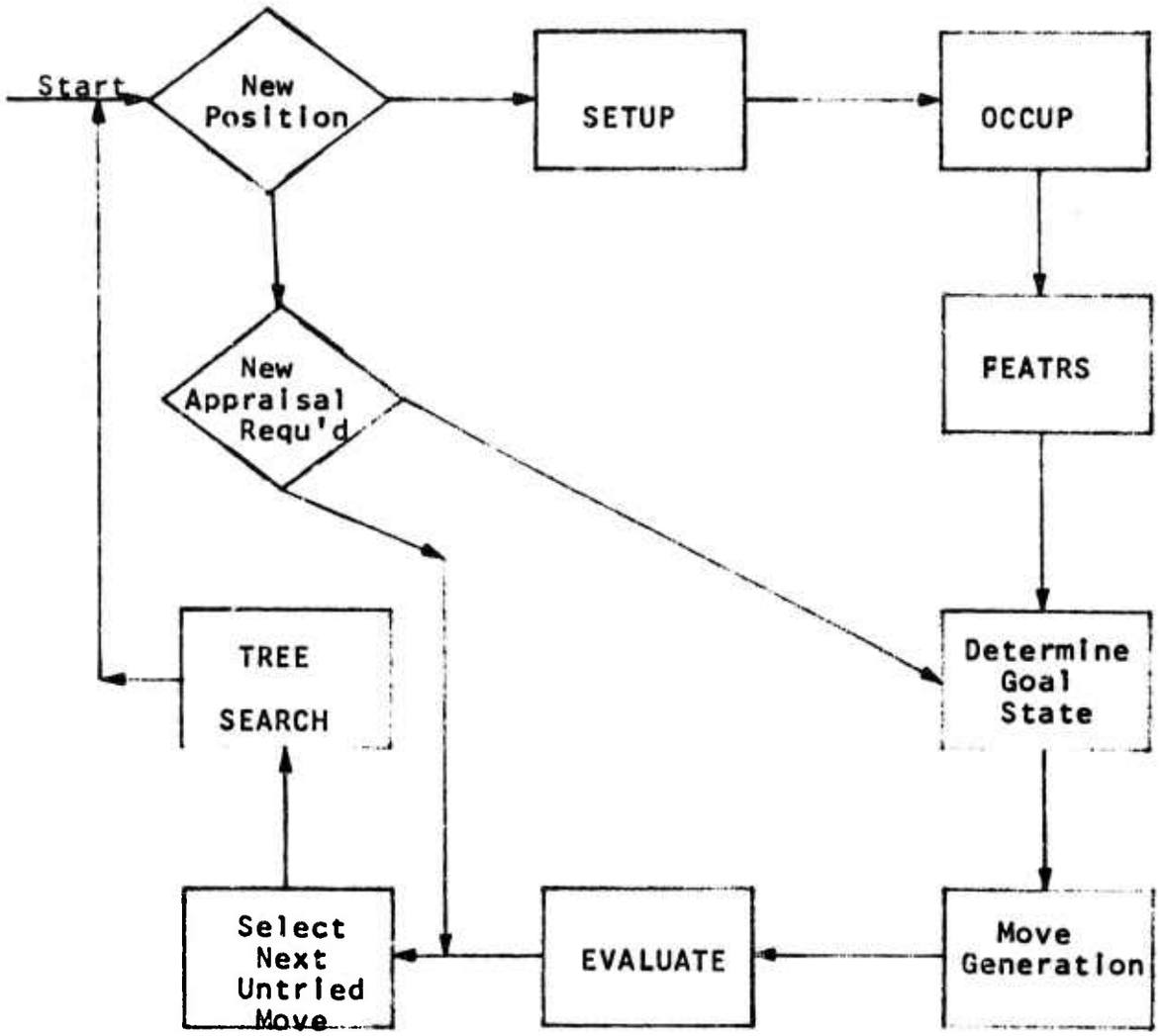

FIGURE 2.1 - Flow Chart of Program

(node) is processed by SETUP, which computes the lowest level of data, then OCCUP, which computes what is going on on a square, and then FEATRS, which computes across-the-board effects. Then a goal state determination is made and move generation proceeds. The proposed moves are then evaluated by EVALUATE and the best is selected for tree searching. If the node is ever returned to during the search, the first decision made is whether the current goal state is still considered adequate. The program then proceeds to find a new goal state or select the next move for testing accordingly.

B. DATA STRUCTURE

1. Board Geometrical Primitives

The program has access to the following tables which give frequently required information dealing with the geometry of the chess board, without having to compute these each time.

VUE(SQ,TP) - Provides the set of all squares that would be legally accessible to a piece of type TP, located on square SQ on an otherwise empty board. The information for one SQ-TP pair is contained in a two-word bit vector.

CLR(SQ1,SQ2) - Provides the set of all squares on a straight line between squares SQ1 and SQ2 (two word bit-vector). If SQ1 and SQ2 are not on a straight line, then the value of the first word is -1. Since a PDP-10 word is 36 bits and only 32 are needed to represent one-half of the chess board, the last four bits in the first word of CLR give the direction between the two points which define the straight line. The last four bits of the second word are used to define the distance, as a king moves, between the pair of squares.

BEND(SQ,DTN) - Gives the name of the last square along the diagonal direction DTN, starting from square SQ.

2. Representation of an Actual Board Position

Each board position is represented as 1040 PDP-10 words. Table II-1 is a summary of this data structure.

TABLE II-1 - Data Structure for each Board Position

Datum	Size(Words)	Type	Purpose
NUM	64	Vector	Name of Piece on each Square
WHR	33	Vector	Location of Piece by Name
TYP	64	Vector	Type of Piece on each Square
TP	33	Vector	Type of Piece by Name
MOB	33	Vector(field)	Mobility of Piece by Name
MAT	2	Vector	Value of Material for each Side
BRD	2	Bit-vector	Occupied/Unoccupied for each Square
PCS	1	Fields	Status Information
SPACE	256	Vector	Free Space Area (lists and Movestack)
PTR	1	Word	Pointer into SPACE (next avail. word)
POSIT	1	Word	Positional Value of Position
VALUE	1	Word	Static Evaluation of Position
WTHRT	1	Field	List Header - White Threats
BTHRT	1	Field	List Header - Black Threats
WMISS	1	Field	List Header - White Unworkable Threats
BMISS	1	Field	List Header - Black Unworkable Threats
WIDEA	1	Field	List Header - White Vacating Moves
BIDEA	1	Field	List Header - Black Vacating Moves
FCTN	33	Vector(field)	List Headers - Piece Functions
BLOK	33	Vector(field)	List Headers - Blocking Relations
PINS	33	Vector(field)	List Headers - Piece Pin Status
FTRS	64	Vector(field)	List Headers - Functions by Squares
OCY	64	Vector(field)	Occupiability of Squares
KMOB	4	Bit-vector	Safe Squares for each King
DIR	64	Bit-Vector Array	Bearing Relation
OTHRU	64	Bit-Vector Array	Bearing Relation
ETHRU	64	Bit-Vector Array	Bearing Relation
OBJ	64	Bit-Vector Array	Bearing Relation
DSC	64	Bit-Vector Array	Bearing Relation
TSQ	64	Vector	Threats on each Square
BEH	16	Bit-vector Array	Names of Pieces Behind Pawns
SAME	14	Bit-Vector Array	Names of Piece of Like Movement
INT	24	Bit-vector Array	Targets for each Type
BEST	2	Vector(field)	Best Material Threat Squares
MVPTR	1	Word(fields)	Ptrs to Movestack (last tried, etc.)
PDIR	8	Bit-vector Array	Names of Pawns on each File
DPATH	2	Bit-vector	Squares on Defensive Paths
GN	1	Word	Current Goal State
TPATH	4	Bit-vector	Part of Refutation Description
RPATH	4	Bit-vector	Part of Refutation Description
RSQS	4	Bit-vector	Part of Refutation Description
TGTSQS	4	Bit-vector	Part of Refutation Description
RPCS	2	Bit-vector	Part of Refutation Description
RTGTS	2	Bit-vector	Part of Refutation Description

The above data structure does not add up to 1040 words since several data items are fields in the same vector.

a. Piece Identifying Primitives

This section concerns itself with the data that are generated at the lowest level of abstraction during the board analysis. The squares of the chess board are numbered from 0 to 63 starting in the lower left-hand corner on White's side of the board, and moving horizontally from left to right until the top right-hand corner is reached.

TYP - is a vector of 64 words representing the type of occupant of each square of the chess board. Table II-2 shows the legal types.

TABLE II-2 - Piece Types

TYPE	WHITE	BLACK
Empty	0	0
King	1	-1
Queen	2	-2
Rook	3	-3
Bishop	4	-4
Knight	5	-5
Pawn not on 7th rank	6	-6
Pawn on 7th rank	7	-7

NUM - is a vector of 64 words representing the name of each occupant of each square of the the chess board. Table II-3 gives the piece naming scheme. Names are assigned in order of increasing piece value, with knights coming before bishops.

TABLE II-3 - Piece Names

Color of Occupant	Range of Values
Empty Square	0
White	1 - 16
Black	17 - 32

TP - is a 33-word vector which gives the type of a piece when referenced by name.

WHR - is a 33-word vector which gives the square of a piece when referenced by name.

BRD - is a two-word bit-vector which has bits turned on for every square that is occupied.

b. Piece to Square Relations

Let us define a bearing relation R(PC,SQ) as a relation of a piece to a square. A piece, PC, is said to have bearing relation R on square SQ, if piece PC bears with relation R on square SQ. We can now define the following bearing relations:

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. Intuitively, the relation DIR defines the direct control of a piece over a square. Direct control is the ability to capture on a square.

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. Intuitively, the bearing relation OTHRU defines the ability of a piece to control a square through one of its own pieces. Such relations occur, for example, when two rooks of the same color are lined up on the same file.

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. Intuitively, this relation defines the ability of a piece to control a square through an enemy man that is also controlling the square.

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. Intuitively, this corresponds to the ability of PC to make a discovered attack on the square if the intervening piece were to be removed.

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.

The above are the basic bearing relations that are noted. They are computed by SETUP. Each relationship is preserved in its own table. Tables are laid out to have 64 words each, one per square on the board. Each word is then treated as a bit-vector with bits corresponding to the names of the pieces, and an "on" bit indicating the relationship exists.

c. Other Low Level Data

BEH(PC,PWN) - is a 16-word bit-vector. 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 OTHRU, or ETHRU on the square on which PWN is located. Bits are set for the name of each such piece in the word corresponding to PWN. Intuitively, this corresponds to knowing the names of the pieces that can affect from behind, the pawn's ability to advance.

PDIR(PWN,FIL) - is an 8-word bit-vector. If a pawn, PWN, is on file, FIL, its name bit is set in the word corresponding to FIL. This provides a quick overview of open files and other information about pawn structure.

BLOK - is a vector of lists hanging from list headers named BLOK and associated with the names of pieces. Each list gives the names of sliding pieces which would bear DIR on this piece if all other pieces were removed, and also gives

the direction between the two pieces. Intuitively, this corresponds to the BLOK list giving the names of all pieces whose action this piece could ultimately block, while neither piece moved.

MAT - This is a two-word vector which gives the value of material on the board for each side. Values are computed according to Table II-4, which are the usual values for material multiplied by 50 and made somewhat more precise in the case of knight and bishop.

TABLE II-4 - Value of Material

Piece Type	Value
Pawn	50
Knight	160
Bishop	165
Rook	250
Queen	450
King	1550

PCS - Is a word that contains the following subfields:

WPC - gives the number of pieces that White has.
BPC - gives the number of pieces that Black has.
WPP - gives the number of pawns that White has.
BPP - gives the number of pawns that Black has.
WCAS - is a two bit field that indicates whether White still has King-side and Queen-side castling rights.
BCAS - is a two bit field that indicates whether Black still has King-side and Queen-side castling rights.
FIL - is a field whose value is zero except when the previous move was a two square pawn advance, in which case it takes the value of the file on which this occurred, files being numbered in ascending order from 1 to 8 starting with the QR-file (needed for en passant captures).

All the data in this section are computed by SETUP.

d. Other Data

The remaining words of the 1040 which are associated with every position can be found in Table II-1 above. Only the following structure is worth describing at this point. The others will be described in detail when the computation that writes on them is described.

SPACE is a free space area of 256 words in which list structured data grows. The left-most nine bits of each list member in SPACE are reserved as a pointer to the next word in the list. The movestack for proposed moves is also kept in this area.

The data which represent each board position exist in a stack which has a 1040 word segment for each depth of the search tree.

3. Retaining Important Relations

PINS - While computing the bearing relationships during SETUP, the program also notices all obvious pins; that is, those that depend only on the relative values of the pinning piece and the object of the pin. Information about pins is stored in lists in SPACE with a list header for each piece existing as a nine-bit field in the 33-word vector FCTN. This field is called PINS. A list is used here since it is possible for a piece to be pinned in several directions. For each direction a word in the list gives the direction of the pin, the name of the object square (the one on which the pin object resides), the value that would be lost on the object square if the pinned piece were to move, and up to three names of pinners doing the pinning.

Once the program has meaningful pin information together with the above bearing data, it can embark on a meaningful square by square analysis of the board. The following data are computed for each square by the sub-routine OCCUP:

OCY(COLOR, SQ) - The value of the largest valued piece of side COLOR that can safely land on square SQ. These data are stored in two 8-bit fields in the vector NUM where they can be addressed by SQ and COLOR.

If a square is occupied then the safety of the piece that is there is computed. Since at a later stage, the knowledge of which objects are worth attacking by which men is crucial, the safety of a piece is determined to be in one of five categories:

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).

FTRS - It is interesting to note that the above categories were developed empirically. The initial notion applied here was the categories which one finds in chess books; namely en prise, safe, and overprotected. The need for splitting two of the categories will be explained later. The result of the piece safety computation is stored in the eight right-most bits of the vector FTRS. The result is stated in

terms of how many units of material (with pawn=1 and king=31) are at stake on this square. Six bits are used to indicate this number of units of material, and two bits indicate upon which side the onus rests to restore the balance.

OCCUP - The sub-routine OCCUP is used to compute the above data, but it can also be called by FEATRS to determine what is happening on a square. Its arguments are a square name, a task, and a bit-vector naming the pieces bearing on this square that should be considered in the computation. The most usual task is to calculate OCY and enpriseness for a square, but other tasks are mentioned below. OCCUP lines up the pieces in the passed argument for each side in the optimum order of employment. It then uses a minimax calculation to determine the actual value of each quantity. In ordering the pieces, OCCUP will use the value of a piece unless it is pinned or is itself a pinner of a piece in the passed argument. Each pinned piece goes onto a special internal list which gives the name of the piece and the value of the pin object. Each pinner goes on another internal list which has the name of the piece and the name of the pinnee. It is possible for a piece to be on both lists at the same time. OCCUP also has a mechanism which notices pieces bearing on the square in the OTHRU and ETHRU categories. These pieces are put into special tables according to the name of the intervening piece. When a piece is invoked in the calculation, a check of this table is made. If another piece was bearing "thru" this piece, that new piece is put into the appropriate bearing lists before the computation goes on. The rule for employing pinned pieces is not to employ a pinned piece until there is no longer any piece available which is of lesser value than the value of the object of the pin. The rule for employing pinners, where the pinned piece also bears on the square, is to release the pin only when the pinner is the least valuable piece available and is capturing a more valuable piece. As shown in Chapter V, OCCUP does an excellent job of computing occupiability. This is quite important as the program's outlook is very much conditioned by the accuracy of the information received at any level of understanding. If the correct value is underestimated, then the program could ignore important features, while if it is overestimated the program could spend time in useless activity. In situations where errors could occur, the latter type is of course preferred.

OCCUP is general enough to resolve a large set of problems dealing with a square. It can be called for various purposes with inputs differing from the usual bearing relationships. For instance, if there was a question as to whether a check on the rank from a given square was likely to succeed, OCCUP would be passed only the identities of horizontally moving pieces belonging to the checking side and bearing on that square, plus all defensive pieces. The resulting calculation would pronounce the square safe or unsafe in terms of the survival of the function, e.g. the check.

Whenever OCCUP is dealing with something of value, e.g. an occupied square, a pawn promotion, or a high-powered attacking move, it assigns functions to all pieces invoked during the computation. These functions are in the form of a relation from a piece to a square and can be of four different types:

1) ATTACKING - Involved in the aggressive purpose for which the computation was invoked.

2) DEFENDING - Defending against the purpose for which the computation was invoked.

3) OVERPROTECTING - Providing a presently unneeded defensive unit against the aggressive purpose of the computation.

4) SUPPORTING - Providing protective support to a piece when it carries out an attack on this square without itself participating in the attack.

FCTN, FTRS - When a function is assigned, it is put into each of two linked lists. One list is FCTN which has a header associated with the name of the function performing piece, while referencing the function type and the name of the square on which it is being performed. The other list is FTRS which is associated with the square on which the function is being performed, while referencing the name of the function performing piece and the function type. This makes cross-referencing of functional committments easier than if only one list existed.

DPATH - When OCCUP assigns a DEFENSIVE function to a sliding piece of the side that just moved, it puts the names of all the squares on this defensive path into the bit-vector DPATH. DPATH thus records all the squares on which the moving side can block a defensive path.

4. Analysis of Board Features Using Knowledge from Several Squares

Having obtained quite accurate information about each square, it is now possible to examine the full board for interesting tactical features. This is done by the sub-routine FEATRS. First certain square problems which could not be determined accurately on the first pass, are rectified. FEATRS checks all occupied squares to see if any piece could be a pin object, and if such a pin has not been detected previously. Such pins could be complicated enough so as not to have been picked up on the first pass; e.g. they could depend upon the safety of the pin object, or involve more than one pinner acting on the same pin line. Initially, information on the safety of each piece was not available, nor was it clear whether there was more than one piece acting along the same line. However, the first pass of OCCUP across all squares has clarified these problems. Now, if any such new pins are discovered, they are cataloged, and all functions of the newly pinned piece are re-examined, and reassigned if necessary. The process of discovering undiscovered pins scans the board only once. It is clear that situations exist in which the order of discovery makes a difference on the final interpretation of all the pins on the board. However, it is also clear that repeating the discovery process can lead to cycles from which some arbitrary escape must be arranged anyway. Therefore, we have opted for the single extra scan of the board. We have never noticed a problem in evaluation due to misinterpretation of pins.

POSIT - FEATRS next gets an estimate of which side is controlling the most space in the given position. This computation is a summation over the whole board of (OCY(White) - OCY(Black)) x centrality factor. This quantity is not used at present, but is intended to yield categories of space control (great, medium, even) for future use. It is stored in the word POSIT.

SAME - The vector TP is now examined and pieces of like types are put together

into elements of the bit-vector SAME. SAME has bits set for names of pieces. Sameness here means functional similarity. Thus a queen is the same as a bishop, but not vice versa.

MOB - Next the effective mobility of each piece is computed. This consists of first noting the squares to which a piece could pseudo-legally move. (A pseudo-legal move is one that would be legal if it were allowed to move a piece in such a way that the king of the side on move would be in check after the move, or if castling while in check were allowed.) Then credit is given for each square which can safely be occupied by the piece in question. Safety is here determined using the OCY values previously computed for each square. The mobility count for each piece is stored in a five-bit field named MOB which is in the vector FCTN, where it can be accessed by name of piece. Pieces that are found to have sub-standard mobility have the LMP (low mobility piece) bit in this vector set to preserve this fact. Table II-5 gives the minimum mobility requirements for each piece type. Any piece with mobility less than or equal to the given values is a low mobility piece. The king is always considered to be a low mobility piece. Any pinned piece is also a low mobility piece.

TABLE II-5 - Minimum Mobility of Types

Piece Type	Minimum Standard Mobility
Queen	3
Rook	1
Bishop	2
Knight	2
Pawn	0

KMOB - The mobility of each king receives special attention. For each square that is in the VUE of the king from its present location, the following determination is made. If the square is occupied by a piece of the king's color, no action is taken. Else, if any opposite colored piece bears DIR on the square, the lowest valued such piece is assigned the function of guarding this king escape square. Else, the bit corresponding to this square in the vector KMOB is set to indicate that the king has mobility to this square.

INT - The next analysis task that FEATRS undertakes is to identify for each piece type, all worthwhile targets. A worthwhile target is one which would change its en prise status if a piece of this type were to attack it. This information is then stored in bit-vectors in the array INT, which is accessed by piece type. The bit-vectors contains "on" bits for each square that has an interesting target on it. Since this information is frequently used later for evaluation of suggested moves, it is very desirable that it be as accurate as possible. Here is where the issue of, for instance, COMPLETELY OVERPROTECTED versus PARTLY OVERPROTECTED becomes important. A pawn which is defended by king and queen and attacked by a rook is PARTLY OVERPROTECTED. This causes the program to decide that this is a target of interest to pieces below the value of a rook, but not to those of value of a rook and above. If the pawn were merely regarded as OVERPROTECTED, then this interest of the lower valued pieces in attacking it would not be noticed statically.

WTHRT, BTHRT - FEATRS now scans all squares. For each piece that can presently move to a given square, it determines whether the piece can make a double attack or an attack on a low mobility piece or a capture on this square. Moves which are deemed effective, in the sense that there appear to be no reasons for their non-success, are put into lists of effective attacks. These lists hang from the list headers WTHRT and BTHRT and accumulate White threats and Black threats respectively.

WMISS, BMISS - Moves which are not effective are put into lists of moves which could be resurrected by considering across-board effects. These lists hang from the headers WMISS and BMISS.

The method for determining effectiveness is to look at the OCY of the square, and if it indicates that the moving piece cannot be captured on this square without loss, to assume the attack has merit. In this case the appropriate attacking function is assigned to the threatening piece, and no defensive functions are assigned. In cases where the piece can be captured without loss, OCCUP is called with the element of the array SAME corresponding to the TYPE of the attacking piece. OCCUP then assigns all necessary attacking, supporting and defensive functions and delivers the verdict which assigns the attack to one of the THRT or MISS lists.

WIDEA, BIDEA - When a square in the INT vector of a piece type is found to have a piece of that type bearing DSC on it, FEATRS finds the intervening piece that could make this discovered attack possible. This piece is then put on one of the idea lists hanging from the headers WIDEA and BIDEA for White and Black ideas respectively. When squares, on which double attacks and attacks on low mobility pieces can take place, are occupied by a piece of the same color as the piece that would like to make the attack there, these blocking piece names are also put on the appropriate idea list.

TSQ - Whenever a square is found on which a piece of the moving side has a threat (as distinct from a capture on that square) the value of the threat is added into the element of the vector TSQ corresponding to that square. TSQ thus keeps track of the threat potential of each square. This is done regardless of whether the threat was deemed to be successful or not, as long as the target square is not occupied by a piece of the attacking piece's color. The values in TSQ allow the identification of key squares, and sometimes make possible the occupation of such a square by another piece, considering that if it were captured the recapturing piece would be able to execute a threat there. Values are also added into TSQ for squares on which pieces reside that have discovery threats. In this way if a move involving this piece is being evaluated, the discovered threat potential will always be known regardless of any other reason for moving the piece. Thus TSQ serves two functions. For occupied squares, a value indicates discovery potential, while for unoccupied squares it indicates threat potential on that square.

BEST - Finally, the lists WTHRT and BTHRT are scanned, and capture threats found there are put into the vector BEST which contains the names of up to five squares for each side in order of the amount threatened on that square.

C. MOVE GENERATION

There have been two basic types of move generating procedures in previous chess programs. One consists of passing all legal moves in review and applying some evaluation process to each of them. The other method for generating moves is evocative, relying on certain goal oriented processes that exist in the program to suggest moves that should be investigated. Clearly the latter approach has certain risks, since it is possible to reach a position in which no move generator will suggest the right move and this would result in a performance failure. In a pass-in-review type of approach, all legal moves would at least be known, whether or not the merits of the correct one were recognized. This would allow certain "fall-back" procedures that could ultimately locate such a move. However in the current program the evocative approach was chosen, since it clearly fits in better with the philosophy of getting away from the generate and test, bushy tree searching method.

All move generation is done in response to conditions detected during program operation. For instance on first seeing a position at any node in the tree, FEATRS and OCCUP have put interesting tactical moves into the various lists available for this purpose as explained above. These lists then serve as input to the simple move generators which are discussed below. Likewise, on return to a node from deeper in the tree, certain information gained during the search produces specifications for moves that would result in defending against a detected "deep" threat, or make an attempted tactic possibly work better. The nature of this deep information is discussed later in this chapter under the CAUSALITY FACILITY.

A move stack for generated moves exists in SPACE. It grows from the opposite end as the free space area for lists. The two halves of the word MVPTR keep track of the location of the last move put on the stack and the last move searched. All generated moves are searched for duplication before being put on the move stack.

The basic move generators consist of the following:

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 then finds all intervening squares using the geometrical primitive CLR, 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 of this square into a position where it does.

These move generators are invoked at various points in the program where intelligence has been gathered that makes it desirable to find moves of a certain type. For instance if a discovered attack possiblity has been found, then MOVEAWAY is called with the square name of the masking piece. If a defensive condition has been found that makes it desirable to guard a certain square on which an opponents strong move lands, then MOVTOCON is called with that square name. The full details of when and in response to what conditions move generation occurs are presented later in this chapter under goal states.

D. THE STATIC EVALUATION PROCEDURE FOR PROPOSED MOVES

There are two basic methods of evaluating a proposed move, and there are good examples in the literature of each method. One way is to execute the proposed move and score the resulting position, as is done in J. BIIT [Berliner, (1970)], CHESS 3.5 [Atkin, et. al., (1971)] and others. The other method is to score the transition to a new position implied by the proposed move, using all properties of the old position and the differential properties of the move. This method is used by Greenblatt [Greenblatt, et. al., (1967)]. The second method is more economical, since it avoids the work of setting up the derived position (which would have to be done for every proposed move before one is selected). However this latter method involves more special code to score the transition, rather than using the existing code for scoring a position. Also, there is difficulty in developing a good scoring procedure for transitions, since it is desired to have the properties of the old position plus the differential properties of the move equal the properties of the new position.

We chose the second method not only for reasons of efficiency, but also since a major purpose of the whole program is to be selective about potentially good moves. The evaluation procedure uses factors which regularly involve predicting consequences five ply in advance (as for instance in an exchange calculation on a square). Since looking at a position after a move is made, only advances the knowledge one ply whereas scoring the transition advances it no ply, this gain seemed rather trivial in terms of the depth of prediction involved. It was also desired to have a handle on the actual properties of a move since this information can be used to determine thematic relations between moves. However, this use is not implemented in the present program.

The evaluation procedure used involves scoring the effect of a proposed move on an existing position. The resulting score after the evaluation 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 the evaluation procedure.

EVALUATE (the evaluation sub-routine) first determines the safety of the new square for the moving piece. This is done using the OCY values explained above. 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 without loss, the following procedure is performed: Each opposing piece that has been assigned a defensive function on that square has its FCTN 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 at stake on this square (as recorded in FTRS) + the TSQ value for that 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 coefficients used above have been developed over the life of the program, and the present values merely reflect a reasonable state of things. No claim for optimality is made. In addition there are still several lesser effects not yet programmed because the adjustment process on the above factors is still going on. Among these lesser effects are: 1) Whether a piece that has a defensive function would have to relinquish a blocking function (other than pinned which is noted); and 2) Whether a piece that has an attacking or supporting function on this square could unblock some favorable line.

Based on the above, an adjustment to the threat value associated with a proposed move is made. If the gain (or loss) associated with capturing the moving piece, plus the redeeming value, are less than the threat value, then the threat is reduced to that value, otherwise it is left the same.

Next, all moves that retain threat credit for giving a check through the above procedure are evaluated for the effectiveness of the check. This consists of determining whether the checking piece would be giving up any king escape guarding functions which it has been assigned, and which can not be taken over by another piece. Also every king escape square in the vector KMOB that can be controlled by this check is noted. After this the total number of escape squares available to the king are counted and the check is evaluated according to the degree of mobility left to the king, with a completely immobilizing check getting a large bonus.

EVALUATE then determines if the moving piece is pinned, and if so, subtracts the expected loss from the value of the move. If the move was recommended by an agency in charge of finding defensive moves (see discussion of GOAL STATES below) it probably received a heuristic value for its defensive potential. Such terms are now added in. Any change in the centrality of the moving piece causes a relatively small addition or subtraction from the value of the move. This tends to cause the program to prefer centralizing moves when all else is equal.

If the move is a capture, its evaluation is adjusted upward for any defensive function the captured piece may have had on a BARELY DEFENDED or PARTLY EN PRISE piece. Upward evaluation also occurs for any attacking functions that the captured piece had on any EN PRISE pieces. If the moving piece had functions on other squares than the one to which it is moving, an adjustment must be made for these. Here any defensive functions on pieces that were barely defended result in a debit, as do attacking functions on pieces that were en prise. In all this, if the captured piece had a

function on a square, and the moving piece had a complimentary function on the same square (i.e. one has an attacking or supporting function and the other a defensive one) then no adjustment is made.

Finally, any pieces wich are still known to be en prise are evaluated in such a way as to give slightly more credit to the side which is about to move. This evaluation is far from optimum since it fails to account for two pieces which are mutually en prise, or the possiblity that the side which is enduring the strongest threat may counter it rather than going ahead with its own best capture. During the early stages of program development this caused little trouble. It is, however, a major source of error now, and will be corrected during the next program revision.

E. THE STATIC EVALUATION PROCEDURE FOR POSITIONS

Two evaluations are developed for each node in the tree. They are:

NOMINAL VALUE - This evaluation counts the material on the board and gives credit to each side for all their material threats as cataloged in BEST, the side that it to move getting slightly more credit for its threats. This value is stored in the word VALUE.

PESSIMISTIC VALUE (from the point of view of the side that is to move) - This evaluation counts the material on the board and then adds in only the side-which-just-moved's best capture threat. If this result is more favorable than the Nominal Value, then the Nominal Value is used. The CLAIM SYSTEM (see discussion in Section F3b below) determines whether the Pessimistic Value will be preserved for use during tree pruning.

Since these evaluations only deal with imminent captures (as distinct from forking threats, for instance) they are less than ideal in their estimating capacity. However, at present they are reasonably good estimators, and because of their simple structure it would be easy to up-grade them when required.

F. CONTROL STRUCTURE

1. On the Applicability of Effort Expended During Tree Searching

Consider the following paradigm: Two domains A and B are given. An optimum solution to a given problem can be obtained by taking the correct element from A and matching it with the correct element from B. If the correct solution can only be recognized by comparing it with any other possible solution, the amount of work that needs to be done to find such a solution is proportional to $|A| \times |B|$. Now, assume domains A and B could be sub-divided into domains A1 and A2, and B1 and B2. If it can be ascertained that no solution could exist in A1 x B2 nor in A2 x B1, then clearly much work in finding a solution could be saved. Even if we were not sure that no solutions existed in A1 x B2, but merely that the likelihood was very low, then much work could still be saved on the average by searching A1 x B2 later than domains which have higher likelihoods. This basic notion is behind the following discussion.

In all of the chess programs that we know to exist today, all moves are examined and then graded along a common dimension with subsequent selection of

the N best. We consider this practice as inadequate, since the selecting dimensions could vary considerably depending upon the position. This is particularly true of situations where defence is required. Present day programs have great difficulty with such situations, since scoring moves statically for their defensive potential is clearly more difficult than doing this for attacking moves, and may be nearly impossible. This is because it is impossible to be 100% sure of what is really attacked (e.g. can this attacked pawn really be captured safely, or is this check going to lead to mate or is it meaningless), nor can one be sure whether the right way to meet an attack is by retreating, or somehow interfering with the attack. The problem is further compounded by the fact that only a fraction of the moves with defensive potential can usually be included in the N best that are allowed to be searched at each node. Therefore, it was decided that defence should be a discrete activity in the analysis process, and should be activated when required, rather than have moves with "good defensive potential" mixed in with moves of other "good potential".

Similarly tactics, positional play, and long-term strategy should be separated from each other. Tactical moves are made in order to gain or avoid losing material and are usually of the highest degree of importance. Positional advantages are usually of lower order than the most minimal material advantage. They can be brought about as a result of tactics which leaves material unchanged, or without any tactics at all. We certainly want to react to both methods of obtaining a positional advantage. Positional gains at the end of a tactical variation can be noted for very minimal additional computational cost. In today's programs this virtue is made into a vice. Mixing positional and tactical moves in a given search tree with the hope of discovering both kinds of advantages results in much additional work. This is so, since tactical advantages almost always outweigh positional ones. Thus, the necessary work to find tactical stability is increades by one to two orders of magnitude for a 5-ply search. When positional factors are separated from tactics, these can be pursued after tactics has failed to find a clear-cut solution. In this case, positional moves can be investigated one after another for TACTICAL feasibility. The savings come from the fact that the search need not be concerned with both issues simultaneously, and positional stability (at the playing ability of today's programs) would not necessitate the deep rummaging that is done to assure tactical soundness.

In the same manner, long-term strategical issues should be separated from both tactics and position play. Long-term strategy need be invoked only rarely, to decide on a long-range goal of lower order than such things as material and positional advantages; e.g. move pieces into position for a king-side attack, or in this type of position a knight is better than a bishop. These rare invocations of strategy should be made only when these usually background issues become important, thus avoiding the need to re-examine strategical issues at every node in the tree.

Another division of labor is possible in the area of attacking moves. Some attacks appear more workable than others. Some attacks would be more potent should they succeed. By investigating moves that are most likely to succeed before moves that have greater potential but lesser likelihood, it is possible to establish firm Alpha and Beta values (see discussion in Section 3) which help in trimming branches containing unworkable ideas. Mixing likely-to-succeed moves in with

high-potential moves by giving some weight to each along a common dimension, reduces the ability to introduce the more efficient searching order.

By thus partitioning the problem and actively controlling which partition is being applied, it is possible to realize huge savings in tree size. Below, we describe the partitions of the present program, and how the analysis can proceed from partition to partition, occasionally making decisions to skip one or more partitions along the way.

2. CONTROL OF THE GOAL STRUCTURE AT A NODE

a. THE CAUSALITY FACILITY

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 describe here the data used by the CAUSALITY FACILITY. How the CAUSALITY FACILITY gathers this data and uses it for comparisons and decision making will be found in the examples of Chapter III.

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.

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 name 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. 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 them to indicate that this threat was created by the last move.

TPATH - is a bit-vector with bits representing squares on the board. For any TGTSQS detected 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 lower in the tree and how, is available. A discussion of what the CAUSALITY FACILITY contributes to the tree search may be found in later chapters.

b. GOAL STATES

GN - Goal states are 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 as explained above. A node is always in one and only one of the following goal states (which is preserved in the word GN).

AGGRESSIVE - This state consists of discovering and proposing moves in the following four categories:

STRONG WORKABLE ATTACKS - Double attacks and attacks on low mobility pieces that are deemed workable.

STRONG UNWORKABLE ATTACKS - Double attacks and attacks on low mobility pieces that are not deemed workable (but could be justified by other across the board factors).

SQUARE VACATING MOVES - Discovered attacks and moves which vacate a square that another piece would like to occupy.

SINGLE ATTACKS - Moves that only attack one piece or have one threat but have another beneficial aspect such as improving the position of the attacking piece or opening a line for another piece. (Not implemented thus far).

There are several DEFENSIVE states:

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

NOMINAL DEFENCE - 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 returned to a node with an unsatisfactory value, and the last move tried at this node was blameless. Information backed up during the tree search (the Refutation Description), is examined to determine what the nature of the problem is. Then the move generators are activated to try to

1) Capture pieces that participated in the refutation,

2) Block the paths of any such pieces,

3) Block the threat paths of any piece, against targets which were not actually captured,

4) Move away or protect target pieces, and

5) Protect squares to which refutation pieces moved.

These moves receive quantitative recommendations according to what they appear to be doing in helping with the solution of the problem. They then pass through the usual static evaluation, and are tried in order of decreasing value.

MISCELLANEOUS States:

STRATEGY - This state is invoked only at depths which were specified on input. 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 top 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 study, which is focussed on the tactical component of chess play, it permits a minimal completion to a total chess program capable of playing complete games. This facilitates comparison with other programs (see Chapter V).

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.

c. RULES FOR CHANGING GOAL STATES

The program is able to invoke and change goal state or abandon a node dependent upon:

1) What an initial analysis of the position indicates is the correct state.

2) Whether a satisfactory result has been obtained in this state.

3) Whether any more moves recommended by this state remain to be tested.

4) What the CAUSALITY FACILITY indicates about causes.

These relationships are explained in detail in Chapter IV.

3. CONTROL OF THE GROWTH OF THE TREE

The program employs a depth-first tree search to a maximum depth of 10 ply. It uses many standard tree control devices and several that are believed not to have been incorporated in any previous program. Many of these devices do not work on absolute reference points in making their decisions, but rather use relative values determined at a higher level of control. These relative values deal with level of aspiration, which is discussed after the individual mechanisms are explained.

a. The Problem of When and How to Stop

Ideally, to determine the tactical value of the top node in a tree, all branches that have to be searched should eventually terminate in a quiescent position (one in which neither side has a threat). However, such positions are very rare. Therefore, it is much more usual to stop searching because the current position is now outside of the bounds of reasonable consequences that could have derived from the top position. What this means is that we must have some idea of what is reasonable, e.g. an expectancy, which is represented in the program by the variable EXPCT. We also need to have a margin around this expectancy which will define the limits of reasonableness. This is the constant MARG. With these constructs, we can begin to make such tests. However, even here there are problems. In a non-quiescent position it can sometimes be very difficult to tell whether having lost material is detrimental (it could have been a sound sacrifice) or whether having gained material is advantageous (it could have been a trap).

For this reason, we have devised the following method of dealing with the reasonableness issue. Whenever a move is proposed, we develop an optimistic evaluation of its potential. That is we give credit for anything that may work in its favor and only debit it in a minimal way for any negative characteristics it may have. This tends to prevent moves that have any good features from being overlooked.

When a position is evaluated, we develop two values for the position: 1) A Pessimistic Value that is an estimate of the worst that could happen to the side on move if the opponent were able to carry out his strongest threat, and 2) A Nominal Value which is the best estimate (although in very non-quiescent positions this could be fraught with error) of the value of the position. In general, the search at a node continues as long as there are moves which have a potential greater or equal to that already attained. However, if the initial Pessimistic Evaluation of the node or any result that is returned by the tree search is MARG greater than EXPCT, the search at this node stops. The reasoning is that, if this value holds up, we will readjust the expectation for the

position and redo the search. Otherwise, we are only dealing with a spurious branch of the analysis.

b. Mechanisms that Deal with the Value of a Node

The following mechanisms operate on the value of a node in order to stop the search, and/or change the evaluation environment.

ALPHA-BETA - This device prevents the tree search from investigating branches of the virtual tree which have already been superceded based upon values found in another part of the tree. This is done by retaining two reference values (ALPHA for the side on move and BETA for the other side) for each node which show the best result that each side has achieved so far. Then it would be senseless for the program to investigate any branch which could not be minimaxed to yield a value in the range specified by the two values. Excellent treatments of the implementation and properties of the Alpha-Beta tree pruning algorithm can be found in Slagle and Dixon, (1969) and Nillson, (1971). The most advanced analysis to-date of the potential savings in the tree search achieved by the algorithm is Fuller, et. al., (1973).

CLAIM SYSTEM WITHIN ALPHA-BETA - This is a new concept used for the first time in this program. The idea is the following: Even if Alpha and Beta define a range of contention within which the program is currently trying to find the best solution, it is possible to help things along further. Suppose that the search reaches a point where a terminal evaluation can be made, and it turns out that White is a rook behind what he was at the top of the tree. The backing-up mechanisms of the tree search have no idea whether this was due to pressures on White which resulted in him losing a rook, or whether White tried an unsound sacrifice while under no pressure at all. Consequently, if the result is within the Alpha-Beta limits, the back-tracking process now concerns itself with how White can avoid losing the rook. This would be appropriate only if the rook were not voluntarily sacrificed, in which case it would be appropriate to look only at moves for White which could make the sacrifice sound.

In order to avoid spending time solving such unneccessary problems, the CLAIM SYSTEM does not allow any ALPHA value (best so far for the side on move) to remain lower than the Pessimistic Value for this position. For instance, suppose that in a given position Black's best threat is equal to EXPCT minus the value of a pawn (Black is always striving for the most negative values and White the most positive). However, the ALPHA value which describes the best White has achieved thus far is EXPCT minus the value of a rook. Before going on, ALPHA is adjusted to be EXPCT minus the value of a pawn, thus narrowing the range of contention. Then if White were later to sacrifice a rook unsoundly, the search could never become concerned with how to achieve anything less for White than EXPCT minus the value of a pawn (which he is considered to have achieved already at the node in question). This algorithm is especially effective when ALPHA and BETA are far apart. There is a certain risk in this method, as in the forward prune heuristic (described below), that a position's potential will not be properly appreciated. However, this is a normal risk associated with generating sparse trees and the risk should lessen as the program improves in its abilities.

POSITION REPETITION WITHIN A BRANCH OF THE TREE - This device checks the current position against previous positions in the same branch of the tree to see if a position repetition with the same side on move has occurred. This is necessary if perpetual checks and other forms of (forced) repetition are to be recognized as draws.

POSITION REPETITION WITHIN GAME - This device checks a move made by the opponent to see whether it results in a repetion of some earlier position in the game. If so, then a switch is set which causes every position generated during the tree search to be compared with hash-table entries representing all previous positions in the game. Any repetition that is found results in a value of "draw" being assigned to that node and causes the tree search to back up from that point.

STOP WHEN AHEAD - This heuristic causes backup at a node where the pessimistic evaluation is significantly higher in value than was expected at the top of the tree.

SUCCESS IN DEFENCE - This heuristic operates only when a node is in one of the DEFENSIVE goal states. Suppose in such a state, a value is backed up, which is greater or equal to EXPCT. Then if the material at this node plus the side-to-move's best threat are not greater than the backed up value, the search at this node stops. The rationale is that the defensive states are for the purpose of finding satisfactory solutions to defensive problems, and not for finding ways of improving prospects in excess of this.

SATISFIED - If at any node a value is reached which deviates from EXPCT by more than MARG, the search at that node is stopped. This is done, even though other moves may be available for testing, on the assumption that this will not be a principal variation in the final analysis. At a later stage of development it would probably be desirable to mark such values so that they can be identified as a certain type of estimate during backup, and then further action could be taken at the top of the tree if desired.

MAX. DEPTH - There is a maximum depth (currently 10 ply) beyond which the program can not search. When this depth is hit, we assign the nominal value of the bottom node to this branch.

TOTAL EFFORT LIMIT - If the top node is in the STRATEGY state and two minutes of CPU time have already been used for this move, then no additional searching is done if a satisfactory move has already been found. This is the only effort limit that exists at present. As the program becomes more involved in playing games, it is expected that others will be implemented.

c. Mechanisms that Deal with the Value of a Move

FORWARD PRUNE - This mechanism compares the Static Value of a proposed move with the ALPHA value at that node. If the move's value is worse than ALPHA, the proposed move is rejected without testing. Since the comparison involves taking the static value of a move and comparing it with the value of a position in the tree, some errors could be made here. Therefore, as explained

earlier, the static value for the proposed move is optimistic so as to minimize the likelihood of rejecting a move that may succeed.

CAUSALITY REORDERING - This mechanism is invoked whenever the search returns to a given node. In such a situation, the backed-up Refutation Description indicates why the move made was held to its current limit of effectiveness. If this was less than the maximum to which this player could aspire (as defined by Alpha-Beta), the program generates the set of moves that could do something about this description. It then looks on the move stack of as yet untried moves at that node, and promotes any untried move that was mentioned in the set of moves generated from the Refutation Description. This device thus reorders previously suggested moves because of information gained from the depth search. It does not however revalue the move, so that such a move could still be pruned by the forward prune device.

4. Level of Aspiration

After much experimentation, the following scheme for level of aspiration has emerged. 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. An expectation (EXPCT) is needed, that is the best estimate of the value of the position at the top of the tree. Around this expected value, a margin (MARG) is defined. If a value, that differs from EXPCT by more than MARG, is ever backed up to the top 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 top 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 top of the tree is EXPCT minus MARG. In order to prevent oscillations in the value of EXPCT, a very conservative view of 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. Otherwise, it is possible for a value to be returned to the top of the tree which cannot be substantiated in further searches. This matter is treated fully in Chapter IV.

CHAPTER III

THE REPRESENTATION AND ITS ADVANTAGES

This chapter presents a discussion of the organization of the data that make up the representation and how these data are used. It follows up on the definitions and constructions of Chapter II and uses these in providing a more commanding overview of the program.

In order for a chess program to be both a good player and efficient, it is necessary that:

1) Good moves be proposed for investigation

2) The goodness of proposed moves correspond somewhat to the order in which they are to be investigated.

3) The ultimate goodness of any proposed move be recognized

Whereas the tree search is charged with delivering an ultimate verdict about any proposed move, the data structure is charged with statically finding important moves and doing this economically. Thus we are interested in this chapter with the recognition equipment that must exist in order to support the latter effort. There are two main points to notice:

1) The recognition machinery must be capable of finding worthwhile moves even if their properties are somewhat hidden. We present examples of this and of how it is achieved.

2) The recognition machinery must be able to reject worthless moves in order to avoid wasting the efforts of the tree search. Clearly this notion cannot be carried to the extreme, where the recognition machinery would be charged with producing just one move at the top node which would always be the best. Instead we take an intermediate attitude, and allow several moves to be proposed, but impose a stricture of reasonableness. This stricture is rather subjective, but must get tougher as the recognition machinery improves. Thus it is possible continuously to find new ways of improving the recognition machinery in order to notice a new facet of a move/position which makes that move unreasonable in that position environment, or makes the move important in that position environment.

We show examples of good tactical moves that our method finds, which current standard methods fail to find. We also show examples of moves that the recognition machinery effectively excludes (by giving them a very poor rating), which programs with less sensitive recognizers would include and thus waste tree searching time. Being sensitive to the goodness of moves narrows the width of the search tree, making searching to greater depth possible. The representation is primarily used in move generation, move evaluation, and the creation of descriptions. These facets are treated in turn. We present first a brief review of the pertinent parts of Chapter II.

A. HOW IS THE REPRESENTATION ORGANIZED

The most primitive elements in the representation 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 and explained in Chapter II. They are DIR, OTHRU, ETHRU, DSC, and OBJ. 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.

In addition, the relation BLOK indicates when a given piece is in the way of any sliding piece, between that piece's present location and the edge of the board. Another important relation is BEH which tells which pieces, not presently able to control the squares to which a pawn can advance, would be able to control these as the pawn advances. This involves rooks and queens behind the pawn. The totality of the above relations makes it possible to determine statically what squares are available to a given piece from its present location under a variety of simple conditions.

The next level of abstraction involves using the above information to rummage about the board, without doing tree searching. Certain points on the board are considered to be important, and analytic routines are invoked to discover the state that these important points are in. This involves such things as the apparent safety of every piece on the board. The program begins to thus put an interpretation on what is going on on the board. Functions, which specify a piece, a role it plays, and a square on which this role is played, serve to remember key roles of various pieces in the interpretation. The following types of functions have been found useful; it being a sufficient condition for one function type to differ from another, if it at any point requires a different analytical treatment: ATTACKING, DEFENSIVE, SUPPORTING, OVER-PROTECTING, PINNING, GUARDING THE ESCAPE SQUARE OF A LOW-MOBILITY-PIECE, and BLOCKING A SQUARE. Among the BLOCKING functions, there also exist dysfunctions, which serve against the best interests of the side whose piece this is. This occurs, for instance, when a piece block's an escape square for one of its own pieces. Functions are assigned to pieces only when a meaningful role for the piece-function combination is found. Special routines arbitrate over the meaningfulness issue, assigning, for instance, DEFENSIVE functions only when something needs to be defended.

B. HOW IS THE REPRESENTATION USED

1. Move Generation

The above described information environment is well suited to detecting meaningful tactical moves. It can be used for finding aggressive moves that are: captures, double attacks by one piece, attacks on a low mobility piece, moves that

cause a discovered attack, and moves that remove a piece that at present has a blocking dysfunction. This in turn creates new functional assignments associated with maintaining and defending against the threatening moves. It can also be used for generating moves that defend against the statically (without tree searching) noticed threats.

If the search ever returns to a node with a defensive problem, a Refutation Description will be available to describe the problem. This description is examined and appropriate defensive moves are generated from it. In all cases, the recommending agency has some idea of why a move is being proposed and assigns a heuristic quantitative recommendation to the move according to what it may be expected to accomplish should it be successful.

2. Move Evaluation

There are two general advantages to the use of this representation in the move evaluation process. The first is that it is detailed enough so that it is possible to evaluate the effect of a move simply by scoring the 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. The second advantage is that 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 a perturbation of these functions it is possible to examine the representation in an attempt to gauge the effect of such a disturbance. From the experience we have had, it is possible to both notice effects which would take up to seven ply to unfold in an actual game, and to estimate the maximum possible effect due to such a perturbation with good reliability. These effects involve 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. Optimism is introduced by giving full credit for any effect favorable to the moving side, and only a small partial debit for effects that are unfavorable.

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, it is quite usual in programs such as the Greenblatt and Northwestern programs, to 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 when, for instance, 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 function is be able to deal with effectively. It should be noted that the expedient of including all checks and captures is not a panacea, since many double attack moves or other forceful moves which do not fall into the class of checks and captures, appear to lose material but actually are correct sacrifices. Moves of this type are frequently not discovered by today's better programs, whereas our program makes the discrimination quite well. For example, in the

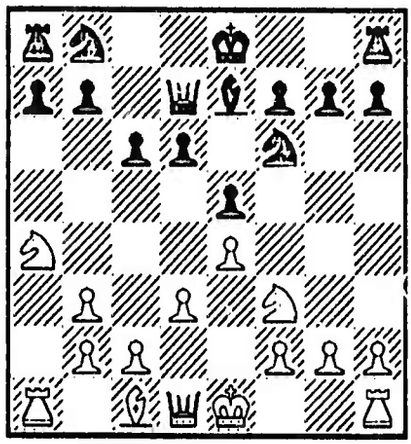

Figure 3.1

White to Play

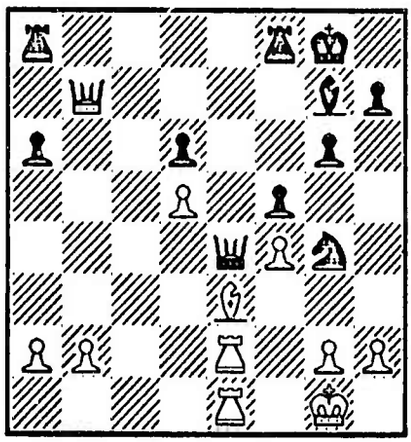

Figure 3.2

Black to Play

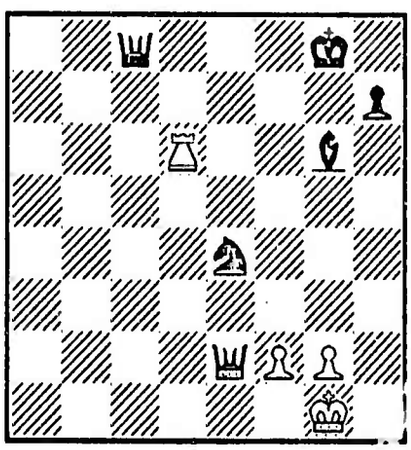

Figure 3.3

White to Play

position in Figure 3.1 the Greenblatt 1968 program (which had already achieved a class C rating) plays 0-0 for White while in the tournament mode, instead of N-N6 winning the exchange. In Figure 3.2 from the game TECH - CHESS 3.5 (Northwestern) ACM Tournament - 1971, it is Black to play. He can win a piece by 1.-- NxB because if 2. RxN then B-Q5, 3. Q-N3, QR-K1, 4. K-B2 ,QxBPch wins a whole rook. But instead CHESS 3.5 played 1.--KR-N1. It is clear that the wrong move was made because the program failed to understand that 2.--B-Q5 pinning the rook, also made the Black queen invulnerable. This is substantiated by the fact that Black did eventually play NxB winning the exchange, after he had previously played QxQP, getting the queen away from the tender spot. All this is rather strange as the Northwesten program sets up the position resulting from every move it wants to evaluate. Thus even after 2.--B-Q5 is set up, it still does not know that the queen is safe. While both these programs play very good tactical chess (for chess programs), these examples are typical of things that they do not do well, and that cannot be covered up by a catch-all such as "Try all captures and checks". Our program solves both problems correctly.

The evaluating procedure is programmed to look 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 take up each in turn.

a. Guard Destruction

Guard destruction results from a capture of a piece which has one or more functions to perform. It is possible to detect this effect by simply setting up the position resulting from the move. However, as previously explained this is a cost which can be avoided. With the use of functions, it is quite easy to note the disappearance of functions associated with a capture, and to also note their value. It would also be useful to know when the piece whose functions are being destroyed can be recaptured, and whether it is possible to do this so as to restore all functions without relinquishing any new ones. This feature has not yet been implemented.

Figure 3.3 shows a simple example of guard destruction. White to play can play 1. RxBch, PxR, 2. QxN and thus gain material. In this case, the simple rule of trying all captures finds the solution too. However, if the capture RxBch were not associated with some sacrificial theme, there would be no point in it. Such cases would just result in wasted tree searching effort.

In our program the move 1. RxBch is noticed because it is a capture. The captured bishop is known to have the function of guarding the knight on K5. Since that knight now becomes endangered whereas it was previously save, the REDEEMING VALUE of 1. RxBch becomes higher than the loss of the rook for a bishop that the move entailed. This results in a strong recommendation for the move.

b. Piece Overloading

Piece overloading occurs when a piece that has a specific function also has another function to perform which cannot be performed at the same time. The

piece will become overloaded when either function is demanded of it. It will be successfully overloaded, if the cost of demanding the first function is not greater than the reward to be gained as a result of the second function having to be relinquished. Some simple examples will help to clarify this: If a pawn is guarding each of two other pawns, then it would not be worthwhile for a bishop to capture one of the pawns at a net loss of 2 units, in order for some other piece to be able to capture the second pawn worth one unit. However, if a pawn is guarding each of two knights, then if a rook were to capture one of the knights (for a net loss of two units) in order for some other piece to capture the other knight for an ultimate net gain of one unit, this would be worthwhile.

Figure 3.4 shows a simple example of piece overloading. Here White to play can play 1. RxN, PxR, 2. QxN. It is the pawn which is overloaded by having two functions to fulfill and not being able to do both satisfactorily. Our program notices the move 1. RxN because it is a capture. Since the defending pawn has another defensive function on its FCTN list, it is found to be overloaded. Since the value of the overload is equal to the value of the knight on QB4, it creates a REDEEMING VALUE which makes the loss of the rook for a knight worthwhile. It is important to note the economics of such transactions; e.g. there is no point in starting with 1. QxN as the amount that can be recovered is not worth the investment made. As a matter of fact, 1. QxN is evaluated as below the value of the existing position.

c. Decoying

Piece decoying is a relatively simple idea which involves the sacrifice of some amount of material in order to bring to a new square an opponent's piece of greater value than the amount sacrificed. In a successful decoy sacrifice, the decoyed piece will then be attacked, and a net gain will result. The king is the piece that is most often decoyed; however, possibilities also exist with lesser valued pieces. The crucial information item here is again the function (defensive), which tells which piece will be decoyed. This information is absolutely essential, as simply knowing whether or not a piece is defended will not result in distinguishing between sacrifices that have decoy value and those that simply bring a very low valued piece to a new square.

Figure 3.5 shows the simplest type of decoy sacrifice. In the position it is White to play and he can mate in two moves by the sequence 1. QxPch, KxQ, 2. PxPmate. The present program finds this sequence without any back-tracking. This is due exclusively to the DECOY VALUE that EVALUATE associates with 1. QxPch. Many contemporary programs would investigate the moves QxPch, QxBch, PxP, and other attacking and capturing moves. In such programs the move QxBch would almost certainly be preferred to QxPch, since both moves give check, but the former captures a more valuable piece. However, the issue here is not capturing the most valuable piece, but luring the most valuable piece into a new position. In this connection, EVALUATE gives the move 1. QxBch an unsatisfactory static rating, since giving up the queen for a bishop and the luring of a rook to a new position does not get a satisfactory REDEEMING VALUE. The check gets no credit at all, since the recapturing rook has no other functions to perform. In fact the only sacrificial move besides 1. QxPch which gets an above average rating is 1. PxP which opens up a line on

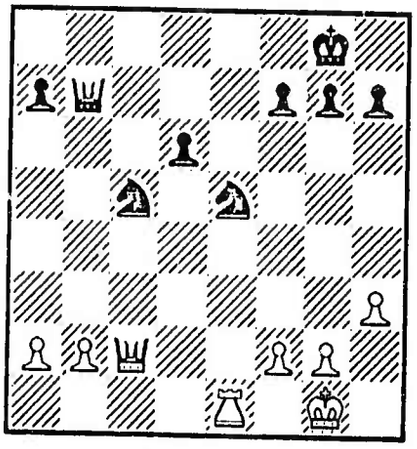

Figure 3.4

White to Play

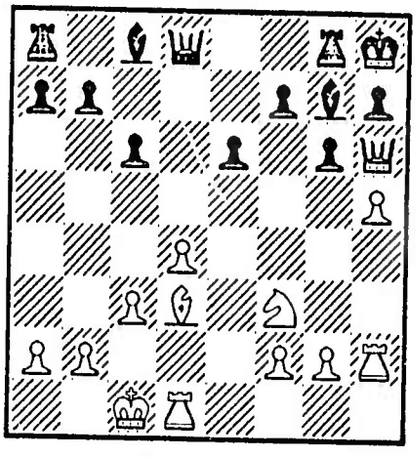

Figure 3.5

White to Play

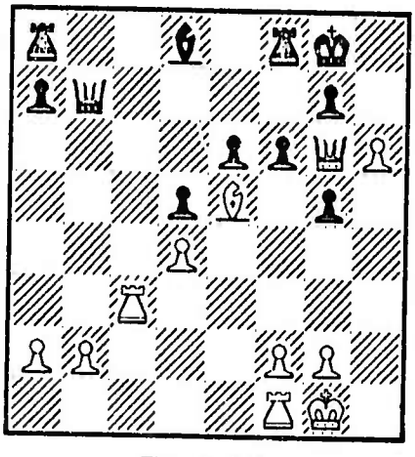

Figure 3.6

White to Play

the enemy king. All the other sacrificial moves do not have adequate properties to merit a high rating. It should however be pointed out that if the Black rook were not at KN1, both QxBch and QxPch would be tried in that order and neither found to be sucessful. However, the algorithms only help in detecting which violent moves have no potential, not which ones are guaranteed to succeed. For that, the tree search is required.

d. Blocking and Unblocking

Blocking and unblocking involve complicated problems, some of which are not addressed by this program. We turn first to those that are. It is possible to block the defensive function path of an enemy man, thus making the object of its protection undefended. This is noticed in the program by checking the destination square of a move against DPATH; a catalog of squares on which defensive paths can be intercepted. If the destination square of a move is mentioned in DPATH, EVALUATE finds all pieces that bear DIR on this square. It then checks the DEFENSIVE functions of each such opposing piece, to find which cross the given squaro. For this it uses CLR(function piece location, function square). For every defensive function that crosses this square, EVALUATE raises the evaluation of the proposed move according to the value of what this function defends. It is also possible to block an enemy attack or support path. However, this is a defensive gesture and is treated as part of the defensive move generation procedure.

An example of defensive path blocking can be seen in Figure 3.6. Here the correct move, 1. R-B7, is originally noticed by FEATRS since it forks the Black queen and the KNP. It is put on the list WMISS, that has moves not likely to succeed, since the terminal square is unsafe. EVALUATE then notices that the move intersects the defensive path of the Black queen bearing on its KN2. Now, TSQ(Black's KN2) indicates that this is a square where White is at present threatening to give check (with the queen). This results in the value of the move 1. R-B7 getting enough heuristic credit to establish it, based upon its static evaluation, as one of the potentially best moves in this position. Ultimately, mechanisms should be added which will be charged with proposing such function path intersecting moves; however, the present program is limited to merely detecting such occurrences in moves proposed by other agencies.

In unblocking, it is possible to move an own man that has a BLOCKING dysfunction, and thus make a previously blocked move possible. This would occur, for example, when a knight is occupying a square on which a queen would like to give check. This is noticed in EVALUATE by checking the origin square of a move in the vector TSQ. This will indicate the vacating potential of this square, both for other pieces that would like to get there and for discovered attacks. Thus the discovered attack is also an unblocking move, and the utility of such a move is noticed at the same time, when TSQ is examined. A simple example of an unblocking move is shown in Figure 3.7. White to play can unblock the square K5 where the bishop can give check by moving the knight that is there. This can be done most forcefully and effectively by 1. NxPch and as Black must recapture BxN, White follows up with 2. B-K5mate. The move 1. NxPch is orginally found by OCCUP, but put on the WMISS list because the checking square is adequately defended. EVALUATE later assigns a high value

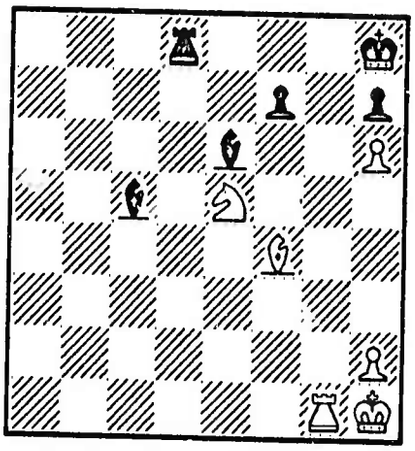

Figure 3.7

White to Play

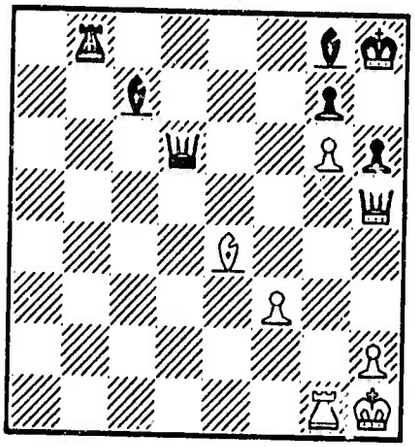

Figure 3.8

White to Play

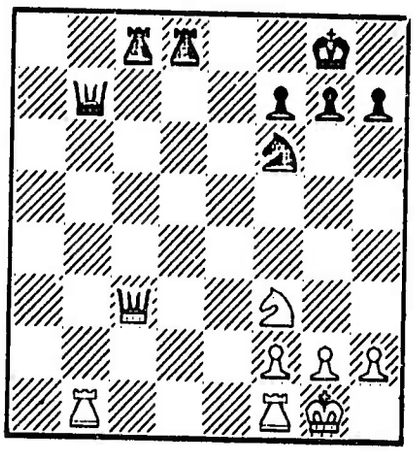

Figure 3.9

White to Play

to the move because it vacates a checking square while giving check and capturing a pawn. Thus our program finds the winning sequence easily.

It is also possible to take advantage of the fact that an enemy piece has a blocking function. This is done by evoking any other function that it may have (overloading). Thus, if it were to exercise the other function it would no longer be on the square on which the blocking function had to be performed. The rules for piece overloading apply perfectly well to this case. In fact such overloading is noticed as part of the piece overloading calculation described in Chapter II. A simple example of a sacrifice to achieve unblocking is shown in Figure 3.8. Here White can play 1. QxPch since the reply PxB unblocks the path of the White KNP thus allowing 2. P-N7mate. In our program the capture 1. QxPch is notices by OCCUP and put on the list WMISS because the capture is defended against. In EVALUATE the blocking function of the Black KNP is noticed (it blocks a check by the White KNP) and the move thus receives a high value. The mate is then found immediately.

Finally, moving a pinned piece is also an unblocking move, but one with undesirable side effects for the moving side. EVALUATE checks the PINS list associated with the moving piece's name to see if it was pinned. It then debits the evaluation of the move by the maximum valued piece, that moving the pinned piece will expose.

EVALUATE is aware of all the above cases of blocking and unblocking and credits and debits the value of any proposed move accordingly. In addition, there are cases involving opening of lines by moving a piece that presently limits the scope of one or more other pieces. This happens, for instance, if a knight moved in order to clear a path to a square where a queen could give check. This is not noted by the present program. There are also moves that produce side effects in the process of clearing a line. Such a move is 1. N-N6 in Figure 3.1 above where the line clearance creates a pin and at the same time takes advantage of it.

The present program handles these more complicated unblocking moves as follows: If there is any square, that has an object of interest (in the vector INT) to the piece that is being unmasked, anywhere on the line that is being unmasked, the proposed move receives heuristic credit for one-third of the value of this object of interest. This is far from an ideal solution. For instance, the analysis for Figure 3.1 should start by noticing that moving the knight clears the path to the Black rook on QR1 which is in the INT vector of the White rook. Then it should notice that the Black pawn on QR2 is bearing on QN6 and thus is responsible for its protection. Then when this pawn is also found to be on the newly opened line, the possibility of a pin arises, and this should be enough to fully justify trying the move. Besides the above, there are unblocking possibilities associated with every piece that could participate in a projected capture sequence on a square. This seems to be the full scope of blocking and unblocking possibilities.

e. Desperados

A desperado is a piece that tries to sell its life as dearly as possible. This can occur when it is trapped and bound to be lost. However, a more important case occurs when the piece is subject to capture, while an equivalent valued enemy piece is also subject to capture. Consider the position in Figure 3.9. If White were to play 1. RxQ, then Black could reply RxQ and the position would remain materially equal. Likewise, if White moved the queen away, Black could do the same without any change in the material balance. However, if White could use his first move to capture something, and thus sell his queen dearly, he could make a gain if Black's queen was still capturable after this. Thus in Figure 3.9, White can win by playing 1. QxN. Now if Black tries the counter-desperado QxR (QxN does not work because of QxQ), White wins with 2. QxRch followed by 3. RxQ. Thus it takes three desperado sacrifices in sequence to produce the correct solution. Our program finds both the desperado moves 1. QxN and 1. QxR attractive using the procedure explained below. It eventually finds 1. QxN to be correct by tree searching.

Detecting useful desperado moves is up to the board evaluation portion of EVALUATE. All captures are always proposed in the AGGRESSIVE goal state. The potential of a desperado capture can be found when the relative threat picture for both sides is known. Thus, it is up to the board evaluation computation to determine that after 1. QxN above both queens are en prise, but Black cannot remove his from attack while capturing the White queen (as could occur for instance after 1. QxR). In a program that always tries all captures, desperado sequences will always be found, as long as they terminate within the search horizon. However, many existing programs must waste a lot of time checking out foolish captures, if the other necessary conditions for the desperado do not exist to warrant the investigation.

f. Recognizing the Futility of Certain Moves

Figure 3.10 shows examples of moves, which the present program statically recognizes as worthless, that would however be searched by any program using the "all checks and captures" paradigm. White to play has four checks here and two captures and none of them are even slightly worthwhile. In the present program none of these moves gets enough of a REDEEMING VALUE to justify the amount of material that is expected to be lost in the transaction. Take for instance, the move QxRch. The rook is defended by the knight which has no other functions to perform. Thus there is no overload. When the queen is recaptured it will decoy the knight to a new position, but losing a queen for rook is too much of a cost to achieve this. The captured rook has only one defensive function to perform: guarding the check at Q4. However, since the capturing piece is also the one which would give the check against which the rook is guarding, this is detected as not being a guard destruction. Finally, moving the queen to Q8 does not block or unblock any important lines, and conditions are not ripe for a desperado (no enemy queen also en prise). Therefore this move receives a rating below the EXPCT for the position, and will not be searched unless some desperate set of circumstances are found to exist (they do not exist in the given position). The remaining checks and captures are evaluated as poor in a similar way.

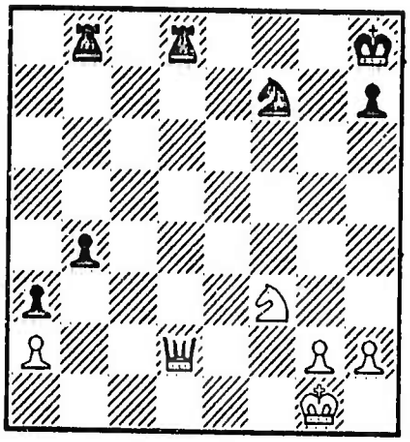

Figure 3.10

White to Play

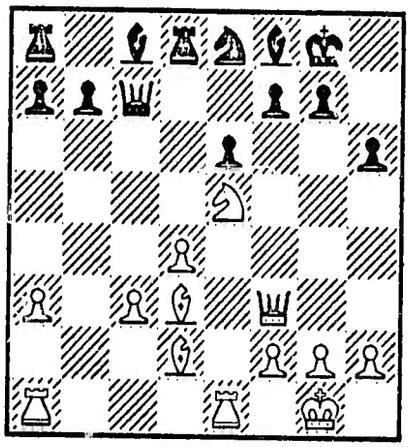

Figure 3.11

White to Play

Now consider the same position with an additional White rook on K1. In this new situation, the Black rook at Q1 has a new function: guarding the check by the White rook at K8. This would now provide some justification for the move QxRch, since it would destroy the guard against the check by the rook. That QxRch does not in fact ultimately work, is not important. With a few more changes in the position, it could be made to result in a mate. Hence, there is some potential in the move, with a White rook bearing on K8. There is NO potential in the move QxRch in the original position. The lack of a data structure which allows such functional comparisons could make it practically impossible to detect the difference in the two situations. This in turn makes for wasted tree searching effort.

3. Creating and Passing Descriptions

Another use of the representation is in the creating of descriptions which can be passed to some agency which could use the description to try to solve a problem. The usual format is that the description is accumulated during the search process as this moves from node to node. If the description is found to be relevant to a problem at a node, facilities in the program attempt to apply it. The only facility of this type in the present program is the CAUSALITY FACILITY, which is described together with examples in Section D below.

Two other possibilities for creating and using descriptions have been discovered while developing this program. However, neither of these is implemented as yet. The first is the use of themes to restrict the combinatorial explosion of tree searching. It would be possible, when evaluating a move, to keep track of why a move received a REDEEMING VALUE if it was a sacrificial move, or why it was recommended if not. This could be put into a description associated with the move, and would be passed forward as the tree search went deeper. The contents of all descriptions associated with the moves of a given side in the variation being presently examined, would constitute the theme which is being attempted. Then criteria would have to be set up by which all proposed moves would be judged. This would involve not only recognizing the potential of a move, as is done at present, but also whether the move should be tried here. This means considering whether the move was also possible earlier, and whether it appears to continue the "theme" to this point. There are several problems still associated with defining how a move can be acceptable to a theme.

The other use of descriptions that we envisage is the development of intermediate data from Refutation Descriptions. These descriptions would take the form of a lemma that is posited about things that could happen in a given position. Thus, if the capture of a rook allowed a mate in three, then certain key elements of the position relating to the capture and the mating sequence (as described in the Refutation Description) would be abstracted from the position. Then, if in a direct descendant of this position, these key elements still pertained, the view would be taken that the capture of the rook is still not feasible. How lemmas can be constructed and used is discussed in more detail in Chapter VI.

In the remainder of this chapter, we present examples from the program's performance that illustrate how the above described facilities operate in a practical environment.

C. EXAMPLES OF THE STATIC ANALYSIS PROCESS

In the first example, we present a complete view of all the higher level data generated as part of the static representation of a position. These data are generated from lower level data (location of pieces, bearing relations, etc.) by the routines OCCUP and FEATRS as explained in Chapter II. The moves generated from these data and then evaluated as part of the AGGRESSIVE state are also shown. We refer to the position in Figure 3.11.

Table III-1 shows the occupiabilities of each square. The table is in the form of a chessboard with squares corresponding to the squares of Figure 3.11. The top entry in each square pertains to the occupiability for White, the lower entry for Black. The number indicates the value of highest valued piece for that side that would be safe there. This is using the standard values of the pieces, e.g. pawn=1, etc., to king=31. When the letter associated with the number is S (safe), this indicates that if a piece of the side whose OCY this is, is presently bearing on this square, it would be safe there. If the letter is N (not safe), only pieces not at present bearing on this square would be safe there. For example, consider White's QB5 square. The values here are 2 N and 1 S respectively. This means that White pieces worth 2 points or less but not presently bearing on this square would be safe there. This excludes the pawn that is presently bearing on the square. The value for Black indicates that any piece worth 1 point or less and presently bearing (or not bearing) on the square would be safe there. This is clearly true; however, the pieces presently bearing on the square, each being worth more than 1 point would not be safe there.

TABLE III-1 - Occupiabi"ties of Squares

31 N 31 N	0 N 31 S	0 N 31 S	0 N 31 S	0 N 31 S	0 N 31 S	31 N 31 N	0 N 31 S
0 N 31 S	0 N 9 S	0 N 31 S	0 N 3 S	0 N 31 S	9 N 3 N	0 N 31 S	9 N 3 N
0 N 3 S	0 N 31 S	1 N 3 N	0 N 31 S	0 N 31 S	0 N 9 S	1 S 2 N	0 N 3 S
0 N 31 S	31 S 0 S	2 N 1 S	0 N 9 S	9 S 0 N	1 N 3 N	1 N 3 N	31 S 0 N
31 N 31 N	3 S 0 N	9 S 0 N	5 N 0 N	31 S 1 N	31 S 0 N	31 S 0 N	31 N 31 N
3 N 5 N	31 N 31 N	9 N 3 N	31 S 0 N	31 S 0 N	31 S 0 N	31 S 0 N	31 S 0 N
31 S 0 N	31 N 31 N	31 S 0 N	31 N 0 N	31 S 0 N	31 S 0 N	31 S 0 N	31 S 0 N
31 S 0 N	31 S 0 N	31 S 0 N	31 S 0 N	31 S 0 N	31 S 0 N	31 N 0 N	31 S 0 N

Table III-2 shows the physical blocking relations of pieces blocking each other's movement. It should be noted that there is no functional meaning associated with these relations.

TABLE III-2 - Blocking Relations

Blocking Piece	Blocks Path of	Ending on
Black Rook on Q1	Black Rook on QR1	KR1
Black Bishop on QB1	Black Rook on QR1	KR1
Black Bishop on QB1	Black Rook on Q1	QR1
Black Bishop on KB1	Black Rook on Q1	KR1
Black Knight on K1	Black Rook on Q1	KR1
Black Knight on K1	Black Rook on QR1	KR1
Black Pawn on K3	Black Bishop on QB1	KR6
Black Pawn on K3	White Rook on K1	K8
Black Pawn on QR2	Black Rook on QR1	QR8
Black Pawn on QR2	White Rook on QR1	QR8
Black Pawn on QN2	Black Queen on QB2	QR2
Black Pawn on QN2	Black Bishop on QB1	QR3
Black Pawn on QN2	White Queen on KB3	QR8
Black Pawn on KB2	Black Queen on QB2	KR2
Black Pawn on KB2	White Queen on KB3	KB8
Black Pawn on KN2	Black Queen on QB2	KR2
Black Pawn on KN2	Black Bishop on KB1	KR3
White King on KN1	White Rook on QR1	KR1
White King on KN1	White Rook on K1	KR1
White Rook on K1	White Rook on QR1	KR1
White Bishop on Q3	Black Rook on Q1	Q8
White Bishop on Q3	White Queen on KB3	QR3
White Knight on K5	Black Queen on QB2	KR7
White Knight on K5	White Rook on K1	K8
White Pawn on KB2	White Queen on KB3	KB1
White Pawn on KN2	White Quuen on KB3	KR1
White Pawn on QR3	Black Rook on QR1	QR8
White Pawn on QR3	White Rook on QR1	QR8
White Pawn on QB3	Black Queen on QB2	QB8
White Pawn on QB3	White Queen on KB3	QR3
White Pawn on QB3	White Bishop on Q2	QR5
White Pawn on Q4	Black Rook on Q1	Q8

TABLE III-3 - Piece Mobilities

Piece Name	Effective Mobility (Number of Squares)
White Pawn on Q4	0*
White Pawn on QB3	1
White Pawn on QR3	1
White Pawn on KR2	2
White Pawn on KN2	2
White Pawn on KB2	0*
White Knight on K5	2*
White Bishop on Q3	7
White Bishop on Q2	3
White Rook on K1	7
White Rook on QR1	4
White Quuen on KB3	9
White King on KN1	2*
Black Pawn on KN2	2
Black Pawn on KB2	2
Black Pawn on QN2	1**
Black Pawn on QR2	2
Black Pawn on KR3	0*
Black Pawn on K3	0*
Black Knight on K1	2*
Black Bishop on KB1	2*
Black Bishop on QB1	1*
Black Rook on K1	2
Black Rook on QR1	1*
Black Queen on QB2	5
Black King on KN1	1*

(*) Indicates a low mobility piece
(**) Indicates a pinned piece

TABLE III-4 - Assigned Functions

Piece	Function	Square	Reason
Black King on KN1	Defends	KR2	Attack on LMP (king)
Black King on KN1	Defends	KB2	Occupant
Black King on KN1	Defends	KB1	Pin Object
Black King on KN1	Overprotects	KB1	Occupant
Black Queen on QB2	Attacks	KR7	Pin Object
Black Queen on QB2	Attacks	QB3	Occupant
Black Queen on QB2	Attacks	QB3	Multiple Attack
Black Queen on QB2	Attacks	K5	Occupant
Black Queen on QB2	Overprotects	QN2	Occupant
Black Queen on QB2	Defends	KB2	Attack on LMP (king)
Black Queen on QB2	Defends	KB2	Occupant
Black Queen on QB2	Overprotects	Q1	Occupant
Black Rook on QR1	Overprotects	QR2	Occupant
Black Rook on Q1	Attacks	Q6	Pin Object
Black Rook on Q1	Attacks	Q5	Occupant
Black Rook on Q1	Overprotects	QB1	Occupant
Black Rook on Q1	Overprotects	K1	Occupant
Black Bishop on QB1	Defends	QN2	Occupant
Black Bishop on QB1	Defends	QN2	Attack on LMP (rook)
Black Bishop on KB1	Attacks	QN5	Multiple Attack
Black Bishop on KB1	Attacks	QR6	Occupant
Black Bishop on KB1	Overprotects	KR3	Occupant
Black Knight on K1	Overprotects	QB2	Occupant
Black Knight on K1	Overprotects	KN2	Occupant
Black Pawn on KB2	Overprotects	K3	Occupant
Black Pawn on KN2	Defends	KR3	Occupant
White King on KN1	Defends	KR2	Pin Object
White King on KN1	Overprotects	KR2	Occupant
White Queen on KB3	Attacks	KB8	Pin Object
White Queen on KB3	Attacks	QR8	Pin Object
White Queen on KB3	Overprotects	KB2	Occupant
White Queen on KB3	Overprotects	KN2	Occupant
White Queen on KB3	Attacks	QN2	Occupant
White Queen on KB3	Attacks	QN2	Attack on LMP (rook)
White Queen on KB3	Attacks	KB7	Occupant
White Queen on KB3	Attacks	KB7	Attack on LMP (king)
White Rook on QR1	Defends	QR3	Occupant
White Rook on K1	Overprotects	QR1	Occupant
White Rook on K1	Overprotects	K5	Occupant
White Bishop on Q2	Overprotects	K1	Occupant
White Bishop on Q2	Defends	QB3	Occupant
White Bishop on Q2	Defends	QB3	Multiple Attack
White Bishop on Q2	Attacks	KR6	Occupant
White Bishop on Q3	Guards	KR7	King Escape Square
White Bishop on Q3	Attacks	KR7	Attack on LMP (king)
White Knight on K5	Defends	Q3	Pin Object
White Knight on K5	Overprotects	Q3	Occupant

White Knight on K5	Attacks	KB7	Occupant
White Pawn on KN2	Overprotects	KB3	Occupant
White Pawn on QB3	Defends	QN4	Multiple Attack
White Pawn on QB3	Defends	Q4	Occupant
White Pawn on Q4	Defends	K5	Occupant

TABLE III-5 - Piece Safety

Piece		Status
White Bishop	Q2	Target of Interest to Black
White Pawn	QR3	" " " " "
White Pawn	QB3	" " " " "
White Pawn	Q4	" " " " "
Black Pawn	KB2	Target of Interest to White
Black Rook	QR1	" " " " "

In Table III-5 each of the pieces mentioned is barely defended. There are no en prise pieces in Figure 3.11, and all other pieces are overprotected.

The only piece on a pinned piece list is the Black pawn on QN2. MAT shows material to be even. POSIT indicates White has a very large space advantage. There are no moves on lists WTHRT or BTHRT meaning that there are no workable threats for either side. The list WMISS (White unworkable threats) contains the following moves and threat values (ranging from pawn=1 to king=31):

TABLE III-6 - List WMISS

Move	Threat Value
QxPch	31
B-R7ch	31
BxP	0
QxNP	5
NxP	0

The list BMISS (Black's unworkable threats) contains the following moves and threat values:

TABLE III-7 - List BMISS

Move	Threat Value
B-N5	3
BxP	0
QxP	1
RxP	0
QxN	0

The list WIDEA (White Ideas) is empty. The list BIDEA (Black Ideas) contains the idea of moving the Black pawn on QR2 in order to discover an attack on the White pawn on QR3.

After all the moves suggested for White above are evaluated in the AGGRESSIVE state the move stack looks as follows:

TABLE III-8 - Move Stack

Move	Evaluation (deviation from MAT value, 50 units = one pawn)
B-R7ch	1129
QxPch	1000
NxP	36
BxP	-109
QxNP	-353

From this information base, the program begins its search at the top node. The position is an advanced example of a decoy sacrifice. The program tries the proposed moves in the order given. Nothing works until 1. NxP is tried. This is found to win a pawn because if then QxN, the queen has been decoyed to a new position, and the further overload sacrifice 2. B-R7ch forces the king to abandon its protection of the queen.

Figure 3.12 shows an advanced example of guard destruction. It is Black to play and he can mate in two moves by 1.--QxNch, since 2. RxQ, is answered by R-N8mate. Here the program quickly recognizes the merit of QxNch since the knight is involved in defending the check at KN8 and the recapturing White rook also, while the two Black rooks which are not involved in the capture are known to also have checking functions at KN8. It is clear that most contemporary programs would also solve this problem, since 1. QxNch is a member of the set of checks and captures. The present program however, only looks at the move because of the destruction of the guarding (defensive) functions involved in capturing the knight and the overloading of the White rook, both of which guard the check. It is also interesting to note that the thing that is being guarded is a check and not a material object. This could lead to sacrifices for the sake of meaningless checks; however, at present the tree search is charged with determining the ultimate utility of a check.

An advanced example of piece overloading can be seen in Figure 3.13. Here Black to play wins a pawn plus additional material by 1.--BxP. The bishop cannot be captured by the rook because it is defending against the threat of R-K8ch followed by R-R8mate. And if the bishop is captured by the knight then QxRch wins more material. However, the latter continuation is forced as otherwise Q-N7mate will occur. This also prevents White from ever playing QxR. The program finds its way through all these complexities. However, since it does not recognize mate threats, it proposes the move BxP only as a tricky method of winning a pawn, rather than part of an involved way of creating a mating threat.

An example of compound themes involving unblocking and piece overloading can be seen in Figure 3.14. Here the White rook at N6 is occupying a square on which the White queen could give check. Therefore the goal of moving it is generated, and among the moves suggested is RxRPch. This move also overloads the Black pawn on KN2, which has been assigned the function of defending the pawn and the rook check at KR3, and also the pawn and the check by the knight at KB3. The program then does not have much difficulty in finding the sequence 1. RxRPch, PxR, (KxR, 2. Q-N6mate), 2. NxPch, K-R1, 3. Q-N8mate or 1.--K-N1, 2. NxPch, K-B2, 3. Q-N6mate.

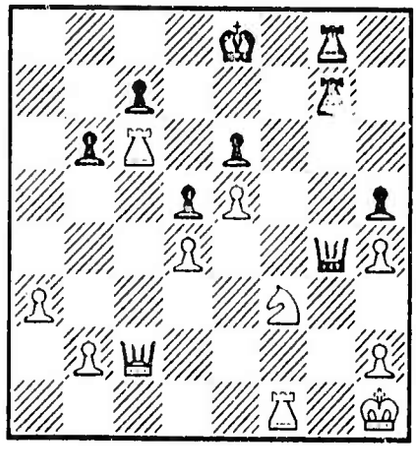

Figure 3.12

Black to Play

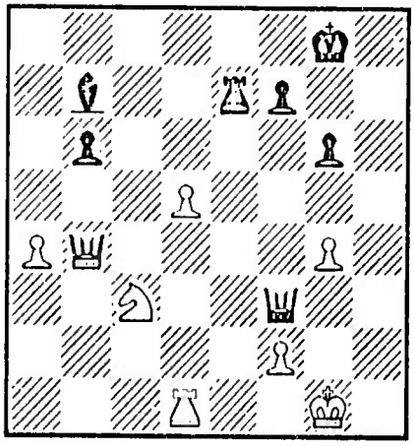

Figure 3.13

Black to Play

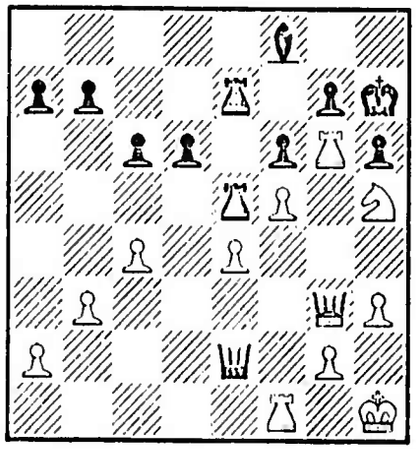

Figure 3.14

White to Play

An example of how the program notices the survival of a function in the face of an imminent capture of the function-performing piece can be seen in Figure 3.15. Here White has several moves which have attractive features. Of the three major possibilities -- RxR, N-B7ch, and Q-B8ch -- the latter appears the least attractive on cursory examination, as it appears to put the queen en prise. However, the program notices that after the capture, the White rook can recapture with survival of the checking function for which the queen move was originally selected, and therefore tries this move first. This results in finding a mate in two moves in the minimum number of nodes. Q-B8ch is preferred to N-B7ch since the latter case the check is seen to not have function survival value and thus has only the capability to win the exchange as a result of a capture sequence on KB7.

An example of the important concept of function retention is seen in Figure 3.16. This position occurs in one of the problems in "Win at Chess", just one move before the principal variation is discovered. Black to play can mate in one move by NPxNmate. There is, however, a similar move, QPxNch, which is not adequate since it allows the White king to escape. When this position is reached, the program notices the functional committment of the QP to guarding the White K4 square, and also finds that no other piece can pick up this committment should the QP relinquish it. Since controlling that square makes the difference between the king having none or one escape squares, the program assigns a considereably larger static value to NPxN, than to QPxN. As a result, the mate is found immediately, instead of possibly selecting the other move and finding that the king escapes, whereupon a later return to this position is required to find the right move.

In Figure 3.17, we see an example of how the program uses some representational information not having to do with functions. Here the Black queen is seen to have only two safe squares to which it could move. This is below what is considered minimum standard mobility for a queen and thus generates the goal of attacking it. Thereafter, the program does not take long to find the sequence 1. P-KN4, QxBP, 2. R-K2 winning the queen. We are not 100% satisfied with how the notion of low mobility is implemented in this program. For instance, in Figure 3.18 the Black queen can be won despite the fact that it has four safe squares which puts it above the minimum mobility requirement. Therefore the goal of attacking the queen is never articulated, and the sequence 1. R-R1, Q-N6, 2. B-B2, Q-N5, 3. R-R4, QxNP, 4. B-R7ch is never found. Conceivably it may be better for determining a queen's mobility to consider the number of avenues that have safe squares on them. In this case the queen has only two such avenues, which should be few enough to make it a likely target. Despite this shortcoming, it is clear that chasing the queen (single attack) only when it is a low mobility piece is very much to be preferred to examining all attacks on it at all times. The latter can result in much useless tree searching.

D. EXAMPLES OF THE USE OF THE CAUSALITY FACILITY

The data structure associated with the CAUSALITY FACILITY was presented in Chapter II. This section explains how it operates and gives examples.

1. Philosophy Behind the CAUSALITY FACILITY

During the tree search, each backing-up of a potential new principal variation and each move that produces an Alpha-Beta prune contributes data to the

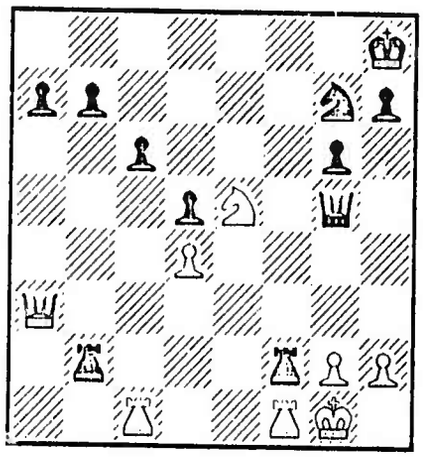

Figure 3.15

White to Play

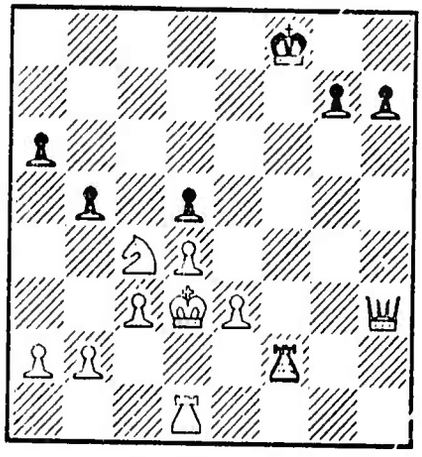

Figure 3.16

Black to Play

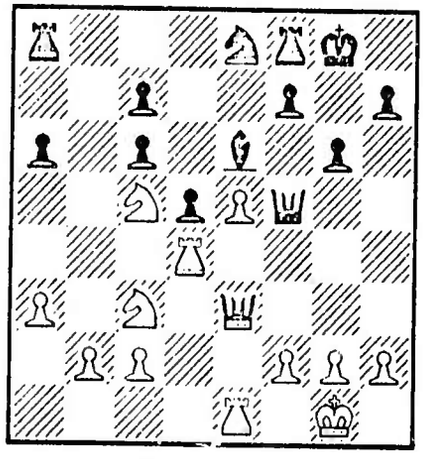

Figure 3.17

White to Play

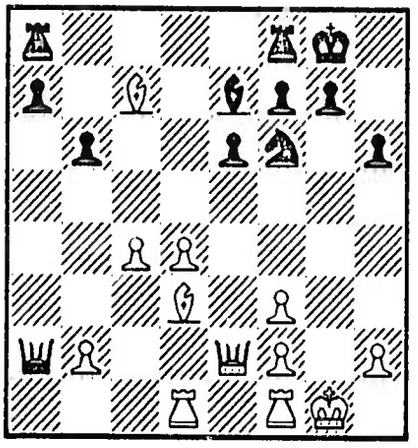

Figure 3.18

White to Play

Refutation Description associated with it. These data deal with changes in the representation of the position brought about when the move that is about to be backed up was executed. This Refutation Description is carried backward to any node that the tree search backs up to. There, if the results of the last move tried were not satisfactory, the CAUSALITY FACILITY consults the Refutation Description in order to decide what can be done about it.

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, to the best of our knowledge no one is using such a scheme, possibly because it is incomplete.

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 in Chapter II. 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 the OCY of critical squares, 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 later in this section. As a consequence, it is not necessary to make decisions about the goodness of certain moves for "defensive purposes" a priori. 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.

2. An Expository Example

We now turn to examples of how the CAUSALITY FACILITY operates. Figure 3.19 shows a position in which Black to play has a defensive task. White is threatening mate in two beginning with 1. Q-K8ch. When the program 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.

Now White 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

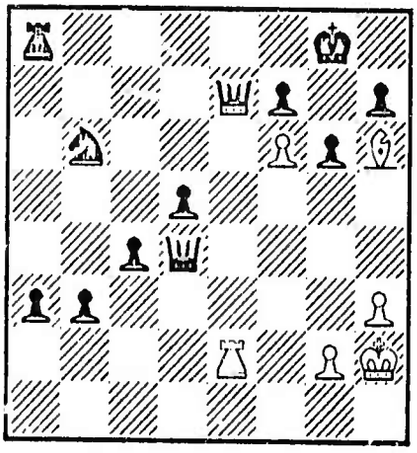

Figure 3.19

Black to Play

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 its status 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.

3. An Example from Actual Play

The above example was specifically constructed for expository purposes. In actual practice, the program uses the CAUSALITY FACILITY continuously to solve problems ranging from very simple ones (having a man en prise), to quite complex ones (a series of forcing moves terminating in a gain). The CAUSALITY FACILITY operates in all sub-trees of the search. Most instances of the application of causality only become apparent when one watches the program in a mode where it prints out all moves tried in the search and all causality decisions made. What tends to happen is that a problem is discovered, and if non-causality cannot be established, then nothing happens. Otherwise, the search begins on whatever counter-attacking moves are left plus the counter-causal moves that have been proposed. Usually the problem can be solved satisfactorily at the node. If not, then the search reverts to the node in the tree two ply above and the same thing goes on there. Thus, the solving procedure is applied recursively at any node that is a candidate, and continues until a solution is found. It is easiest to observe the process when it happens at the top node; however, one should bear in mind that the same thing happens for any sub-tree in the search.

The example in Figure 3.20 comes from a game played between TECH (White) and CAPS-II. It is Black to play. CAPS-II starts out by trying 1.--QPxP, 2. QxNP, QxN, 3. QxR. After investigating this variation for a while, it decides that Black does not get sufficient compensation for the material lost. It then backs up to where 2.--QxN was played and determines that there is no defence there. When the search returns to the point where 1.--QPxP was played the program decides that it is faced with a threat that was not caused by the last move. It therefore decides to examine the Refutation Description and generate the set of counter-causal moves. First the other aggressive move that was proposed, BPxP, is deleted as not having enough counter force to be seriously considered as impeding the opponent's known threat. The following counter-causal moves are then generated, given with duplicates together with the reason they were generated. Moving away a target or a piece that was captured: N-B3, N-K2, N-R3, P-KN4, P-KN3, P-KR3, P-KR4. Blocking a RPATH: P-KB3, N-KB3, Q-B3. Protecting an RSQS square: K-B1, Q-B3, Q-N4. Increasing the mobility of a low mobility piece that became a target but is not presently attacked: P-KR4, P-KR3. When evaluated, the moves are put in the following preference order: N-B3, N-K2, K-B1, N-R3, P-KR4, Q-N4, P-KR3, P-KB3, P-KN4, P-KN3, Q-B3. The program then tries the moves in the given order until it finds one that produces a value greater or equal to EXPCT, which is equal material.

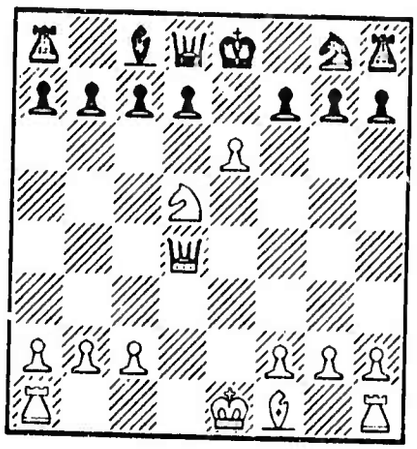

Figure 3.20

Black to Play

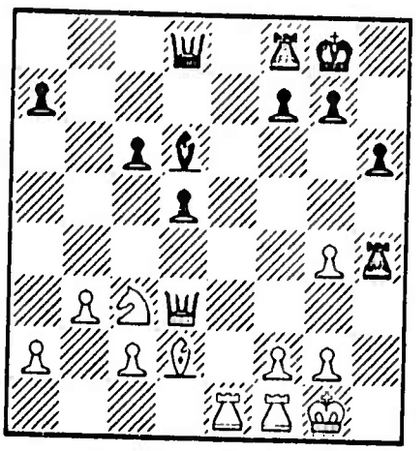

Figure 3.21

Black to Play

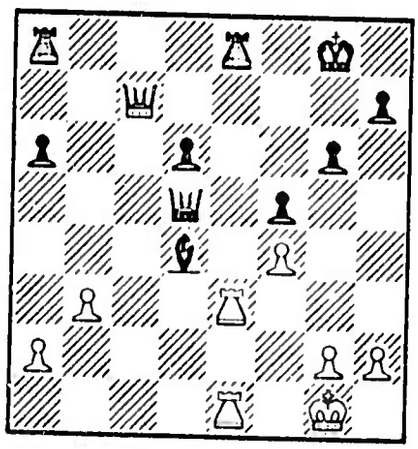

Figure 3.22

White to Play

1.--N-B3 is eventually found to be unsatisfactory because of 2. P-K7 winning the queen. 1.--N-K2 is also bad because of 2. QxNP. 1.--K-B1 is rejected because of 2. Q-B4 winning a pawn. Similarly, all moves down to P-B3 are rejected as losing material, most very quickly. 1.--P-B3 is found to be a satisfactory defensive move so the last three moves on the list are never tested. The program is still somewhat clumsy in homing in on the only correct solution. However, we feel it is to its credit that the correct defensive move was among those proposed immediately upon analyzing the Refutation Description. This is so, since the move P-B3 has really no other qualities to recommend it, and could easily be overlooked by a program that only investigates an arbitrary number of moves of some predetermined character.

4. Causality Reordering

An example of how the program uses causality in order to improve its attacking processes can be seen in Figure 3.21. 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 refuting pieces, and mentions the path of the queen in blocking the check. 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 counter-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.

5. Some Current Deficiencies

An example of how the CAUSALITY FACILITY is not as powerful as human notions of causality can be seen in Figure 3.22. Here it is White to play, and he is faced with a threat to his rook at K3. The program first tries 1. Q-B4 which is suggested by the AGGRESSIVE state. This move eventually fails because of the variation 1.--QxQ, 2. PxQ, BxRch, etc. When the search returns to the point where 1. Q-B4 was played, an examination of the Refutation Description shows that (since

the move 1.--QxQ was played by Black) White's QB4 is an RSQS square. Since 1. Q-B4 put a man onto an RSQS square, the program considers it possible that the consequences were caused by the last move. Thus it is not possible for the causal test to decide that the loss of the White rook at K3 can not be attributed to the move 1. Q-B4. For this reason the program does not understand that it is facing some consequences inherited from above in the tree. Later, the program reverts to the NOMINAL DEFENSE state and concentrates on finding defences for the rook at K3. However, a human player would probably not have had much trouble discerning in the Refutation Description, two problems. The first would have to do with the status of the White rook at K3; the other with the status of the Black queen at Q4. These problems are seen to be independent, and thus the Refutation Description can be seen as uncovering two problems; one of which was caused by the made move, and the other was not. This would then save time in getting to the crux of the matter, which is the defense of the rook on K3. However, this type of problem is not a serious inhibitor of the program's performance.

CHAPTER IV

BOUNDING THE TREE SEARCH

A. GENERAL CONSIDERATIONS

In Chapter I the need for a search that was controlled by the requirements and values found at its nodes was established. This is in contradistinction to a search that would be mainly controlled by a collection of preset parameters such as a maximum depth beyond which no probing (or only for special types of moves) is allowed. However, merely specifying the need for a flexible, no depth-limit search is not enough. An algorithm must be found that makes executing such a search a tractable problem. This chapter addresses this problem.

Two general techniques are employed. The first is the use of any workable method to try to establish a given node as a terminal node. In our program this involves defining several reference levels some of which are global and some of which are local to a node. These reference levels are computed dynamically as functions of the chess position they refer to, and serve to provide criteria for when a certain position no longer has the desired degree of goodness, or is far better than one could reasonably expect. Comparisons can be made between the reference levels at two nodes on the branch of the tree being currently investigated. Having both nominal and pessimistic evaluations of positions available also increases the types of comparisons that can be made. Individual moves are also subject a reference level test, which would thus dismiss poorly valued moves without further processing.

The second technique, which has never been used in a chess program before, is to partition the problem of move selection at a node into a set of hierarchical sub-problems. The reasoning behind this is similar to that presented in Chapter II for partitioning chess into the hierarchical sub-problems of tactics, position play, and long term strategy.

To explain the advantage of such a scheme consider the following. Let us assume that there are N legal moves at a node. Of this set, we are prepared to search K. If we are using a Greenblatt type of searching scheme [Greenblatt, et. al., (1967)], K will be determined by the current depth (e.g. a predetermined set of K's exist, and the magnitude of its elements decrease with depth; i.e. 9,9,6,6,3). In our scheme, K can be considered to be the number of moves that could in any conceivable way influence the tactics of the position at a given node. Since this is a non-numeric bound, we can never be sure how large K will be; it is a function of the position, expectation levels, etc. However, in our partitioning scheme, we assert that there are a maximum of P sub-problems to be solved at any node. Further, associated with each sub-problem is a goal state, and these goal states can be strictly ordered. This means that if an invoked goal state delivers a satisfactory answer, then it is assumed to be the correct answer. There is a certain risk in this, but careful partitioning of moves into goal states minimizes this.

The whole point of partitioning is the following. The K moves that we are prepared to search are partitioned in such a way that every move goes into the highest partition for which it qualifies. P, the number of possible sub-problems (partitions) can be as high as five. This means that we need only deal with somewhat more than K/5 moves at

a time. Moves that could have something to recommend them, but not in the current goal state, would never be included in the search if the present goal state produced a satisfactory answer. The logic of manipulating goal states provides that, when it becomes clear that a certain goal state will no longer satisfy the problem requirements, goal state transition rules are invoked to select the next appropriate goal state. These transition rules utilize certain information, relating to why the last state failed to produce a satisfactory answer, in deciding on the next state. When transiting to the new goal state, only those proposed moves that also qualify for the new goal state are retained. It should be evident that such mass pruning of alternatives is a very powerful technique. For instance, if it is found that a defensive problem such as a threatened mate existed, it would be possible to ignore a very high percentage of all attacking moves, that would otherwise be searched because they had a high static rating. In practice, we find that the ultimate move selected at a node comes from the first goal state tried nearly 50% of the time. Furthermore, the ordering of proposed moves within a goal state makes it likely that a satisfactory move is found before one-half of the moves in that state are explored. As the problem is explored in depth, its exponential growth rate is then correspondingly smaller.

B. PROPERTIES OF A GOOD SEARCH ALGORITHM

In devising a good search algorithm, one is mindful of avoiding, or at least minimizing, the exponential explosion that is the present stumbling block of all chess programs. However, an attempt to mitigate the effects of exponential growth must not leave important moves out of the analysis. Many programs, beginning with Bernstein [Bernstein, et. al., (1959)], have had search algorithms which limited the number of moves that could be searched at each depth. This resulted in being able to apply some control to the total amount of tree searching effort that the program engaged in. However, prespecifying the number of moves to be looked at has certain disadvantages. In a complex situation, it is possible that many moves have features to recommend them. However, the issue of over-riding importance could be one that is very difficult to detect (such as an impending mate). In such a case, a saving move such as moving the pawn in front of the king to create an escape square could be ommitted from the analysis since so many statically superior moves were found to be looked at (many reasonable attacking moves which failed to solve this problem). For this reason, the search sometimes produced results that were considerably below the expectation level that existed at the start of the tree search. To combat such problems, Greenblatt would then reinitialize the search and allow all moves at the top depth to be investigated. This was later somewhat generalized by the Northwestern group to allow investigation of all moves at odd depth levels (for the machine's moves) when an unpleasant situation presented itself. However, we felt that this was still not general enough, since if an opponent's good move (at an even depth) was not looked at, this could also cause a misappraisal of the position. Therefore we wanted a uniform treatment of every node, which would allow the investigation of interesting moves as long as there were some that could have an impact on the appraisal of that node. This meant being sensitive to the problem conditions associated with the node, so that one could be reasonably sure of the importance of any move to be investigated. It also meant being sensitive enough to abandon nodes quickly when an abnormally good result was found. As long as the search process can be justified as being sensitive to the problems at a node, the whole algorithm can be justified. More importantly, in order to improve such an algorithm, it is only necessary to do a better job of specifying sensitivity. And this in turn should result in search trees with still smaller branching factors.

In order to support the above objectives it is necessary to have very definite notions of what the expected value of a top node in a tree search is, and what an abnormal departure from that value is. Then methods must be available for changing the expected value of the top node as unexpected events occur.

In the past, programs such as the Greenblatt program have handled this problem by setting Alpha and Beta to plus and minus one pawn from the expected value of the top node. This has the advantage of restricting the initial search to "meaningful" branches of the tree. However, it has the disadvantage that sometimes a search returns without any move. Then all that is known is that something outside the normal range from the expected value can be forced on one of the opponents. No clue as to the magnitude of the sudden change nor any indication of what moves it is related to is available. Therefore there is no follow-up choice except to redo the tree search with greater latitude on where Alpha and Beta are set and on the number of moves which are investigated at each depth. However, when a principal variation and a value are returned by every search, it is possible to determine what caused the perturbation and have an initial estimate of what the real value of the top position is.

To control the search, it is useful to have five reference levels.

1) EXPCT defines the expected value of the top node.

2) ALPHA and BETA define absolutely only the best that each side has achieved so far at this node (and conversely the best that each side can hope to achieve). Alpha will be used to mean the value for the player on move at a given node, and Beta the value for the other player.

3) The question of what is a reasonable departure from EXPCT is handled by plus and minus MARG. The value of a node (position) will be said to DIFFER SIGNIFICANTLY from EXPCT if its value is assessed to be different from EXPCT by at least the value of the constant MARG. The current value of MARG is 68% of the value of a pawn.

Since Alpha and Beta are not artificially set at the start, every search will return not only a move but also a principal variation. These can then serve as a starting point for a new search, should one be required. The additional cost of this extra bookkeeping is trivial.

The tree control reference levels are shown in Figure 4.1. Alpha and Beta define the range of contention at a node. This means that no value outside these limits can ever be backed up to this node. EXPCT is the expected value of the top node. EXPCT plus and minus MARG define the range of aspiration. When a value outside this range is found, it is considered to be significantly different from the expected value. This causes immediate backing up, without trying to find a still more deviant value. Thus this value is not driven to quiescence as are the values within the range of aspiration. Therefore before backing up, one must be sure that the value being backed up is realistic, i.e. that it can be achieved if it becomes the new EXPCT.

A property of over-riding importance is that the whole search must strive for quiescence. This was shown in Chapter I. It means that for any facet of a position that is being explored (in this case tactics) no judgement should be rendered unless the value found has been quiesced as well as possible or it is outside the range of values

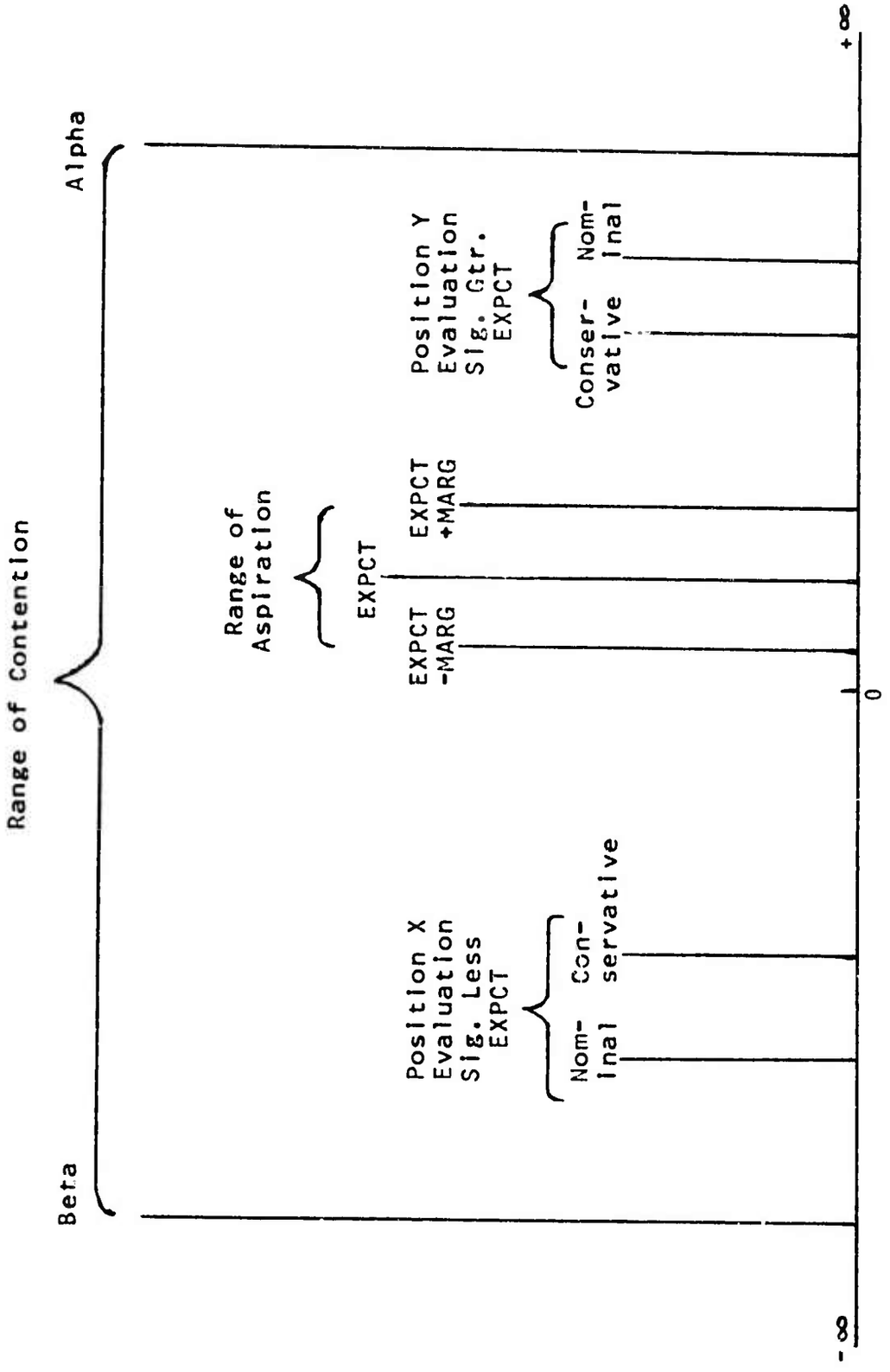

FIGURE 4.1 - Ranges of Contention and Aspiration, and Pessimism in Evaluation

that are reasonably derived from the original position. For tactical quiescence, a simple swap analysis is not sufficient, because certain attacking and defending moves can have a profound influence on the true value of the position. Thus it turns out that the whole search from the top of the tree must be a quiescence analysis. In fact it is correct to equate the notion of a tactical chess program with a tactical quiescence analyzer. This could mean that some branches could be terminated almost immediately while others may have to be pursued 20 or more ply in order to achieve a valid judgement of a terminal node (although in CAPS-II we are at present able to go only to a depth of 10 ply). This is in sharp distinction to the current generation of programs where the search proceeds to some prearranged maximum depth, and then an inadequate quiescence method (swap analysis) is invoked.

When qualifying moves for the tree search, we want to assure pertinence. This means that not only must the move have been proposed by a knowledgable and germaine agency, but the move must be considered to have a reasonable chance to improve on the status quo. Thus we want to generate a static evaluation of the move which will yield an optimistic outlook on its potential.

If at any time a value is found at any node in the tree which is better for the side to move than EXPCT plus MARG, then this is assumed to be an unusually satisfactory result and no attempt to improve upon it will be made at that time. The reasoning behind this is that this value is very likely to turn out to be unsubstantiable. In that case it will be dismissed more quickly if no further effort is spent in trying to find the local optimum. If, on the other hand, the value does survive back to the top of the tree, then resetting EXPCT to the new value forms a firm basis for conducting the next tree search. In that case the tree search is done over starting with the position defined by the end of the principal variation. When the tree search terminates with a value that is not significantly different from expectation, then it is assumed to be an optimal value. This process clearly converges eventually on a value which is optimum according to the above set of definitions.

C. THE ACTUAL TREE CONTROL ALGORITHM

1. Overview

The basic tree search used by the program is a depth first, mini-maxed search with Alpha-Beta pruning. This has been supplemented by algorithms which make risk decisions involving terminating the search at the point of testing. Some of these algorithms are new and others are relatively standard. All mechanisms (with the exception of the CAUSALITY FACILITY which was treated fully in Chapter III) will be explained as they are encountered. During the tree search the basic emphasis will be 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.

Our treatment of this subject will be to examine first the decisions made when arriving at a node for the first time. Then we will discuss the allowable set of manipulations of goal states. Next, we look at how to decide whether or not to search a proposed move. After that we investigate decisions encountered when leaving a node that has not been cut off. Finally, we examine the decisions made when returning to the top of the tree.

2. Decisions Made When Arriving at a Node for the First Time

When first arriving at a node, several tests are made to see whether it is even necessary to process this node at all.

a. Position Repetition

A check is made against all earlier positions in the variation currently being investigated to determine whether this position has occurred earlier with the same side on move. This would mean a repetition of position, and a value of "draw" would be assigned to the node and backup would be ordered. Similarly, if at least one position repetition has occurred in the game and nothing has happened since then to prevent further repetitions, then a check is made of the position at the current node to see if it is in the hash coded file of all previous positions in the game. Conditions that can prevent position repetions with previous situations in the game are: moving a pawn, making a capture, or castling (all irreversible actions).

b. Significantly Ahead of EXPCT

After the new node is statically evaluated, another check is performed. If the current material is significantly ahead of EXPCT for the side whose turn it is to move, then if the pessimistic evaluation of this position is still significantly ahead of EXPCT then the pessimistic value is assigned this position and backup is ordered. The pessimistic value is the lesser of the current material plus opponent's best threat, and the nominal value of the position.

One may question why the pessimistic value should be assigned. The reason 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 top 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 top 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 oscilation 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. The foregoing may be easier to visualize by referring to Figure 4.1.

c. Claim System

After the test for significantly ahead of EXPCT, the Alpha value (which represents the best that the side on move here has achieved thus far in this branch) at this node is examined to see if it can reasonably be re-evaluated to a value closer to the Beta value for this node. This is the CLAIM SYSTEM. The logic behind the CLAIM SYSTEM is as follows: Alpha and Beta define a range of contention that is being investigated in the current branch. Any value outside this range cannot be returned up the tree, since it would represent a logical mistake by one side or the other to get into a variation with such a value when a better variation can be forced in some other part of the tree. However, if the pessimistic evaluation of this position is better for the side on move than the local Alpha value, then some gain has clearly already been achieved. Thus it would be eminently illogical to allow the range of contention to remain at its original settings when it has been detected that one side has achieved an improvement over these settings (although this has not been confirmed by a search). Narrowing the range of contention in this manner allows more Alpha-Beta prunes to occur once the search begins to back up. The only risk in this procedure is that the pessimistic estimate on which the re-evaluation is based may be inaccurate. There is some risk to this, as there is too in making a forward prune. However, the opportunity to make additional prunes makes this risk worthwhile, and all evaluation functions are subject to such problems and could cause similar errors.

Consider a situation where the range of contention at a node is currently EXPCT plus and minus the value of a rook. The current position is materially equal to EXPCT and the only threat by the side not on move is directed against a pawn. Clearly, in this situation the side on move has already achieved a result which is better than the EXPCT minus the value of a rook that he was currently being given credit for. It would then be illogical to consider returning a value to this node which is lower that EXPCT minus the value of a pawn. If later in this variation, the side now on move were to unsoundly sacrifice a knight, this result would be recognized as outside the limits of reality, once the backing-up process began. This would result in an Alpha-Beta prune, meaning that the loss of the knight can not be forced. This is a correct action if the previous evaluation causing the CLAIM SYSTEM action was correct. This paradigm thus allows early prunes of branches containing faulty combinations.

There are of course serious problems in trying to determine statically what the magnitude of any threat really is. However, that is not a problem of tree searching theory, but rather a problem of improving evaluation functions. The expediency, of avoiding such dangerous decisions by doing more tree searching, is precisely why most of today's programs cannot explore enough of the search space that it is necessary to survey in order for significant improvements in playing strength to occur. Suffice it to say that the present program has enough of a grasp of threats (somewhat greater than that of contemporary programs) as delineated in Chapter III, so as not to make too many mistakes here. As evaluation improves, so will the effectiveness of this mechanism. In the meantime it both allows effective Alpha-Beta pruning of variations involving wild sacrifices, and establishes a minimum value for the current node; a value that the program is only allowed to improve on as the search progresses.

3. Manipulation of Goal States

Figure 4.2 shows the partitioning of the problem space. At the top, chess is divided into STRATEGY and TACTICS. TACTICS can be invoked at any node and always gets preference if it finds a solution that is better than EXPCT. Otherwise, STRATEGY can be invoked, but only at depth one in the present program. The purpose of STRATEGY is to at least supply a move if TACTICS fails to propose one. A higher aim is to play the most "strategical" move in situations where no clear tactic is required. In later versions, improvements in this module can make STRATEGY a method of proposing moves that meet long-range goals. The TRACING state can be invoked only at the start of a tree search and is responsible only for bringing the search to the end of the current principal variation. Any other state can follow it.

TACTICS is in turn partitioned into six goal states through which the program can pass. These are: PREVENTIVE DEFENCE (defending unexpected gains), AGGRESSIVE, NOMINAL DEFENCE (answering detected but unproven threats), DYNAMIC DEFENCE (answering a specific refutation detected during data backup), and KING IN CHECK. No claim is made that the partitioning in this program is optimum. It attempts to follow that employed by good human players, and it does appear to succeed in sub-dividing the legal move space into manageable sub-spaces.

Informally, the logical inter-relation between the goal states is as follows:

KING IN CHECK is invoked only when the king is in check.

In case the king is not in check, it is determined if the side on move is significantly ahead of EXPCT in material. If so, and if the opponent has any statically detected threats, the PREVENTIVE DEFENCE state is entered. Otherwise, the AGGRESSIVE state is invoked. The above logic determines the initial goal state at a node.

The NOMINAL DEFENCE state is for positions whose potential is considered worthwhile, but the opponent has a statically detected threat against material which the AGGRESIVE state was not able to handle.

The DYNAMIC DEFENCE state is the last court of appeal. It can be invoked from any move generating state in response to a Refutation Description backed up by the CAUSALITY FACILITY, if the problem was not caused by the last move.

a Goal State Transitions

The way the program progresses through the goal states, once having reached a given node, is shown in Figure 4.3. The goal state at each active node in the tree is remembered in the variable GN. 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.

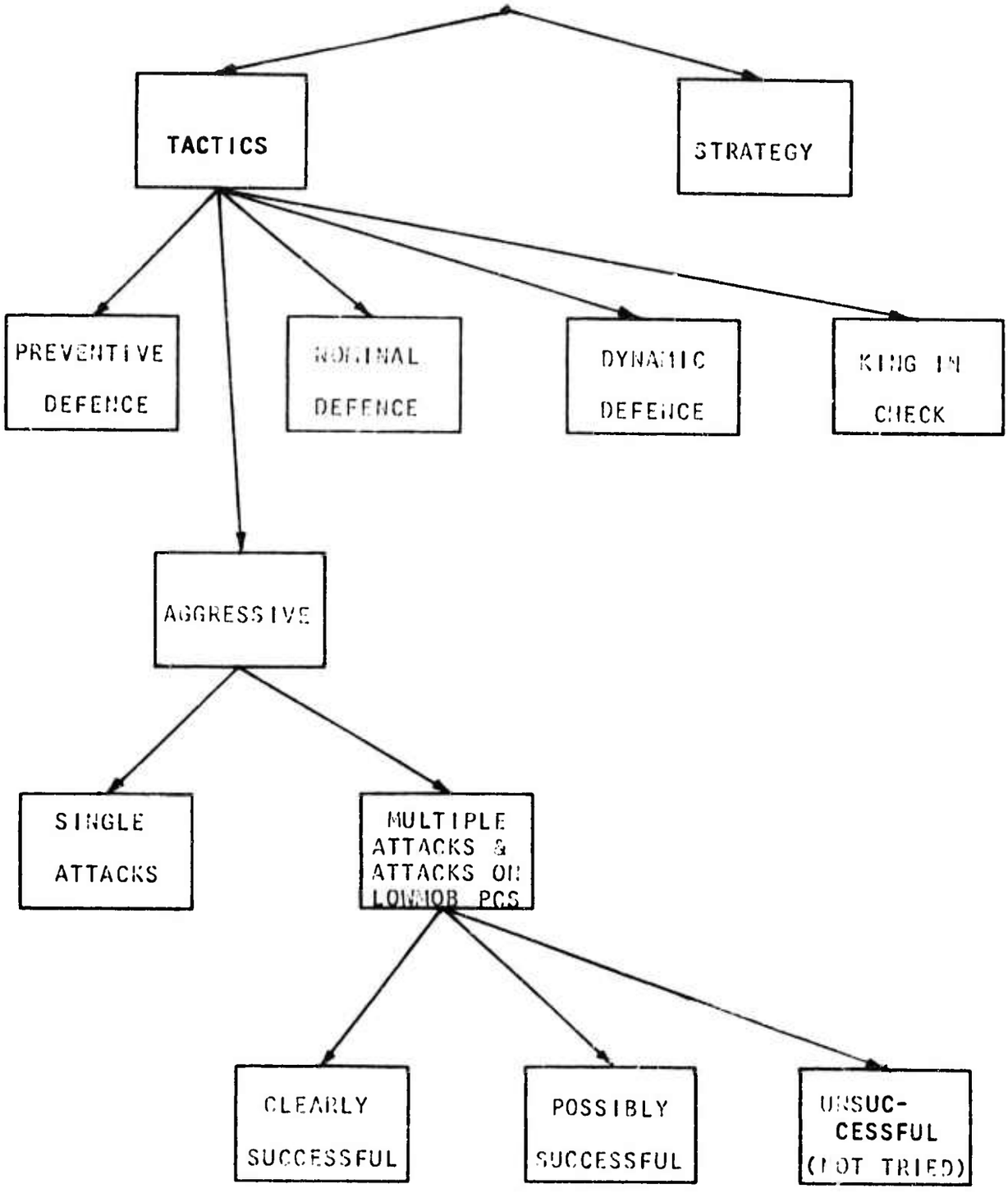

FIGURE 4.2 - Partitioning into Goal States

The "Return to Node" block at the top of Figure 4.3 shows the decision structure that pertains when returning to a node in most goal states. It is invoked as a sub-routine by the main transition diagram in the lower half of Figure 4.3. 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 Chapter III.

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 will result in their being tried earlier, but does not change their value.

Following now the flow chart in the lower part of Figure 4.3, 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 node is exited permanently. 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 the same. As long as there is no such occurrence, 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

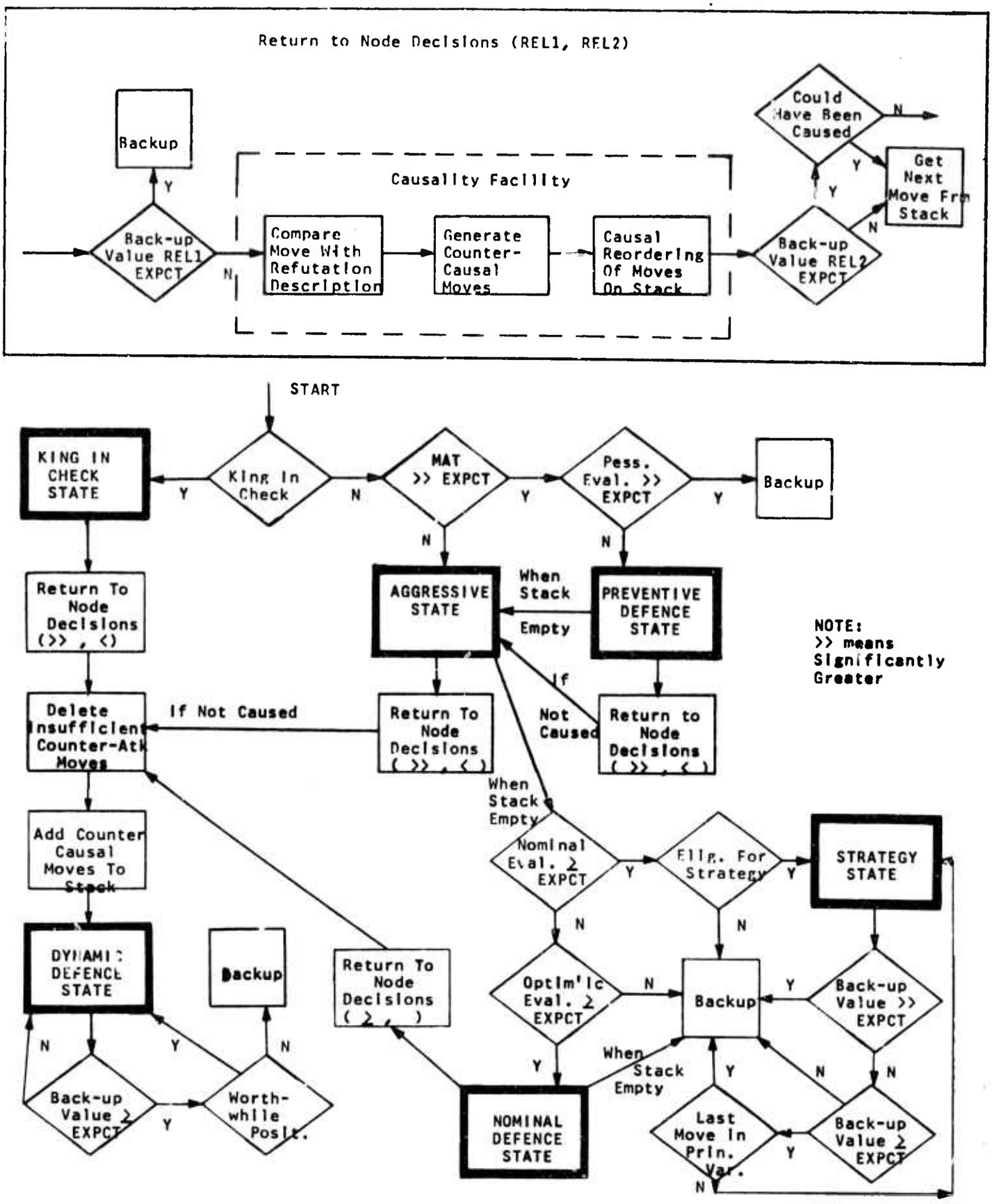

FIGURE 4.3 - Goal State Transition Diagram

reasons at the moment why the side on move should not try to make a successful attacking move. As shown in Figure 4.2, the AGGRESSIVE state is really a set of move generating states. The details of each move generator are explained in Chapter II. The states are arranged so as to generate moves in order of forcefulness. Upon evaluation a likelihood of success measure exists. 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 exit this node permanently. 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 are described in Chapter II. They produce 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 top 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.

b. Inter-relationship of Goal States at Different Nodes

During the development of the program, it became apparent that there exist relations which limit the type of goal state that can be a successor to a particular goal state earlier in the branch being currently investigated. One example of such a relationship is the notion of "loss of initiative". Intuitively, this could be thought of as a situation where a move which was proposed by the AGGRESSIVE state at a particular node leads to a defensive state at a later node without anything else worthwhile being accomplished in the interim. This should then constitute a reason for node termination. We tried two different implementations of this particular idea, but neither one worked satisfactorily. However, we feel that there is only a question of the correct formulation involved here; the basic idea appears very workable, in fact, necessary for tree searching economy.

There also exists a relation between STRATEGY and TACTICS. This is that TACTICS must be available to discriminate between feasible strategical moves and non-feasible ones. The reverse does not apply. The only place that we know of, that strategy serves in evaluating tactical ideas, is at the end of a tactical variation, in evaluating its strategical impact. But this can usually be done by static analysis without resorting to tree searching. The gist of this relation is captured very well in this program.

4. Decisions Made When Examining a Proposed Move

Every proposed move is statically evaluated before being put on the move stack. The details of this evaluation were discussed in detail in Chapter II. When it comes time to try a move at a node, the top move on the move stack is selected for examination. The move is guaranteed to be relevant to the current goal state as otherwise it would have been deleted during goal state change. The value of the move is then tested to see if it is greater than the Alpha value at the node. If so, the move is tried and the search continues. Otherwise, the move is deleted and the next move on the stack is examined. This mechanism is known as forward pruning [Slagle, (1971)] in the literature. In this program the number of moves that are

forward pruned during a tree search regularly approximates the total number of nodes in the final tree. This makes it the most powerful overt device used in the tree search. It is difficult to compare its power with the covert action of Alpha-Beta, which prunes not single alternatives but all remaining alternatives at a node. Thus the power of a single Alpha-Beta prune could be several times that of a forward prune, dependent largely on the expected branching factor of the program.

Another device for controlling the pertinence of moves was tried during this research. However, it failed to perform the desired services and was ultimately abandonned. This device is variously called the search-and-scan strategy [Newell and Simon, (1972)] or dynamic ordering [Slagle, (1971)]. It has previously been implemented in the problem solving MATER program [Baylor and Simon, (1966)] and in the chess playing program COKO-III [Kozdrowicki and Cooper, (1973)]. This strategy has also been subjected to theoretical investigations by Kozdrowicki [Kozdrowicki, (1968)]. In this program it was implemented as follows: Before a move is selected for further searching at depth N, the value of the most powerful move in the current stack is found. Since moves are evaluated optimistically, this presumably represents the maximum effect achievable at this node. Now a search is conducted backward up the current variation, stopping at depths that are an even numbered distance away from the depth N. At each such depth, M, the static value of the last move tried is compared with the maximum effect at depth N. The algorithm now looks for situations where these two values are farther apart than a constant called the RELUCTANCE. In such a case, an attempt is made to determine whether there exists an as yet untried move at depth M, of somewhat greater power (called the THRESHOLD), than the static evaluation of the current alternative at depth M. If so, the move being currently tried at depth M is given a new static evaluation equal to its maximum effect at depth N and is put into a new place on the move stack according to this value. It is also marked as having been previously searched. The most highly evaluated alternative on the move stack at depth M is now chosen for searching, with the search reverting to depth M. The purpose of the device is thus to arrange that the search not be side-tracked in a sub-branch which appears less promising than one somewhere else.

This algorithm was found to do very well in situations where some clear achievement was possible, either a gain or preventing a loss. However, in the run-of-the-mill situation it produced a tremendous amount of useless retraction of moves, and multiplied the usual search effort by a factor of ten or more. It is possible that the RELUCTANCE, which was set equal to one pawn, was set too low. It is also possible that the THRESHOLD, which was set equal to one pawn, was set too low. Another possibility is that the loss of causal information associated with the retraction of the search to depth M was responsible for a general lack of understanding of what was happening. However, the most likely cause of this failure was that reverting to an earlier level in the search prevents the usual backing up during which the Alpha-Beta algorithm has an opportunity to terminate nodes which are superceded. Since Alpha-Beta plays a rather important role in trimming the search tree (see Table IV-1) the loss of this capability probably accounts for the results found. Quite possibly, some additional tests are required to detect board positions in which nothing unusual appears achievable and in such cases scan search should not be used. We did notice that the low RELUCTANCE and THRESHOLD values made little difference, whether a mate was to be found or merely a small amount of material was at stake. Where the algorithm failed was in situations where no change in material could be effected.

Another device was invented during this project, but has not yet been implemented. We call it thematic testing. The idea is to prevent moves that have little to do with one another from being concatenated in a search, even though they may individually appear to be worthwhile. If several interesting but unrelated moves exist, the cost of searching them in every possible order is great. The savings that can be realized by only looking at "meaningful" sequences can therefore be significant. A possible implementation of this idea is the following: During evaluation, moves are partitioned into two classes: those that clearly "work" such as favorable captures, and those that are not "sure to work" but have redeeming features. Each move that falls into the latter class is marked to indicate the sacrificial themes which constitute the redeeming features. These themes as explained in Chapter III are decoying, overloading, and unblocking, etc. Associated with each theme is a set of squares which are pertinent to the execution of the theme. Whenever a move has passed the forward prune test and is the candidate for sprouting, the following operations are performed: If the move is not a redeemed move (it clearly works) then it is passed. If some theme marks have been set earlier in this variation for the side on move, then if this move is a redeemed move, the following tests are done: If this move was not also on the move stack at some earlier depth in the current variation (it was not an interesting move before), then it is passed. If it did appear, but is now given a higher static evaluation than it was earlier, it is also passed. Otherwise, at least one of the theme marks on the proposed move must coincide with one of the set theme marks from earlier in the variation. If this condition is not satisfied, then the move is deemed to be irrelevant to the continuation of any enterprising action that has gone before, and is not searched. In all cases, themes and squares associated with redeeming moves are carried forward for use of future nodes.

This algorithm tries to define "continuation" of a sacrificial theme. In this it asserts that several decoy moves may be strung together, but not a decoy followed by an overload sacrifice. This is generally correct, but may at times impose a stricture that would cause missing a good continuation. Also it is clear that all "clearly workable" moves should not be searched in any arbitrary order. An example of this is what could happen in a king and queen versus king endgame. Here there would be near infinite sequences of workable checks by the queen. However, nothing would be accomplished unless the losing king was already considerably restricted. This type of problem can be solved by recognizing certain properties of the parent position. To recognize these properties would involve detection and analysis of configurations of chunks [Chase and Simon, (1973a)], and resultant classification of positions. Many necessary types of chunks would deal with relationships between pieces, that are different from bearing relations (i.e. king safety). Further sophistication in the development of themes would thus be dependent on having better recognition facilities.

5. Decisions Made When Leaving a Node That Has Not Been Cut Off

When leaving a node (BACKUP in Figure 4.3), after having exhausted all proposed moves in all invoked states, one final check is made. If all proposed moves had to be examined, then there is a great likelihood that no satisfactory move was found and that the value of this node is below EXPCT. On exit from a node a check is therefore made to see if the value of the node is significantly below EXPCT. If it is, then if the current value of material is better than this, but is still

significantly below EXPCT, this value is assigned the node. This is done to get the most conservative estimate of the value of a position for backing-up purposes. The set of reasons for doing this was presented above at the beginning of section C.

6. Decisions Made When Returning to the Top of the Tree

When the tree search is exited at the top node, there are still some decisions to be made. These decisions have to do with whether the result reported by the search is considered satisfactory or whether it is deemed necessary to further refine the results of the search. Refinement may be necessary, since the search is terminated at any node at which a result that is significantly different from EXPCT is encountered. Such a termination only guarantees a result significantly different from EXPCT, but does not guarantee that its magnitude could not be greater. Thus any result, that is backed up to the very top, may not be optimum.

Therefore when a result that is significantly different from EXPCT is returned, it is necessary to redo the search to see if a value that is an even greater distance from EXPCT can be found. Due to the mechanics of the search, we can be sure that any value that is returned to the top of the tree is conservative. Therefore, if a more optimum value exists, it must be still more different from EXPCT than the returned value and in the same algebraic direction. Before starting out on redoing the search, the value of EXPCT is reset to the returned value of the last search. Alpha and Beta at the top of the tree are reset to embrace the range from the new EXPCT to infinity in the correct algebraic direction. The principal variation associated with the search is also saved, and the new search begins at the tail end of this variation. Since taking conservative estimates at all terminated nodes assures that any value returned by the next tree search must lie between the new ALPHA and BETA, we can be sure that this procedure cannot cycle and must ultimately converge on a value. In practice the greatest number of such progressive deepenings of the search that we have noticed is four. We use the term progressive deepening for this procedure, as this seems to correspond well to the phenomenon with this name that psychologists have observed in human chess players [De Groot (1965)]. However the deepening takes place only with respect to the value of the position, not with respect to an understanding of it.

This method of organizing the search has been found to be clearly superior to the usual alternative of finding the best move at every node. Finding an optimum at every node even after a value significantly different from EXPCT has been found, means much of this work inevitably is wasted, as this whole branch has a great likelihood of being cut off. Since there are many false leads that the program is continually following, values significantly different from EXPCT occur frequently. However, not many of them survive to the top of the tree. Thus a single value significantly different from EXPCT assumes the character of an anomaly, and results in the other side using his resources to refute it. This will actually occur an overwhelming percentage of the time. However, in the case where the anomaly actually survives, the cost is doing the search over again. This cost is clearly a linear function of the time for one tree search, and is thus much cheaper than the cost of following unlikely consequences to a conclusion, which is an exponential function.

Whenever a value is returned to the top of the tree which is within the bounds of EXPCT plus or minus MARG, it is known to be optimum with respect to the capability of the program. In that case, EXPCT is set to the value returned by the search and the principal variation is saved before making the move. If the opponent's next move corresponds to the one expected by the principal variation, the next search begins at the tail of the retained principal variation.

D. DISCUSSION AND RESULTS

The work on an effective method of heuristic tree searching is probably the most important general contribution of this thesis. One need only consider the facts presented in Chapter I, to see how central to the whole business of getting a computer to play good chess is the problem of keeping the growth of the tree under control. As reference points for the results here obtained, the following should be kept in mind: The rate of exponential growth of a search tree is proportional to the average number of moves searched at a node. The average chess position contains about 35 legal moves [Slater, (1950)]. The average middle game position contains slightly more moves than this, and most of the positions tested with this program were of this type. Consider a position with 35 legal moves, where all moves are to be searched and the order of searching is random. If in this situation every potential terminal node were to have a different value, then the expected branching factor under an Alpha-Beta search is approximately 15 (using formulas in [Fuller, et. al., (1973]). A program such as TECH [Gillogly, (1972)], which searches all legal moves but has many branch endings with equal terminal value, and also uses some ordering in the searching of moves, has a branching factor of about 8 to 10. The branching factor for programs such as the Northwestern program or the Greenblatt program which search only a select sub-set of all legal moves is about 5 to 6. Human beings are known to search trees of arbitrary depth and produce no more than 200 nodes in the process [Newell and Simon, (1973)]. Since human searching thus converges instead of diverging, its branching factor must be less than 1.0, although some maximum effort rules almost certainly exist also.

The data reported in this chapter was collected during runs of CAPS-II on two basic tasks: sequences of problems from "Win at Chess" [Reinfeld, (1958)] (the basic calibrating task for the program), and game fragments. The book "Win at Chess" is a collection of 300 chess combinations from Master practice. It is considered a basic work on chess tactics and is frequently referenced as such. It can be classified as a moderately advanced instrument for teaching chess tactics to human beings. We will specify the nature of the task wherever any data are presented. The following data were collected on sequences of problems from "Win at Chess". Figure 4.4 shows the characteristic distribution of number of successors to a node, across all nodes in the tree. No data was of course included for nodes at maximum depth, which are not eligible for sprouting. The distribution has the excellent feature that by far the most probable number of successors is one. However the expected number of successors across the distribution is 2.0. This is considerably better than those of other programs, but still a good way from what would be necessary for natural convergence of the search. The problem is that the weighted effect of the number of zero-successors (which is the next most prevalent category by far) is still not enough to outweigh the right hand side of the curve which although it harbors infrequent occurences, still has enough weight to prevent convergence.

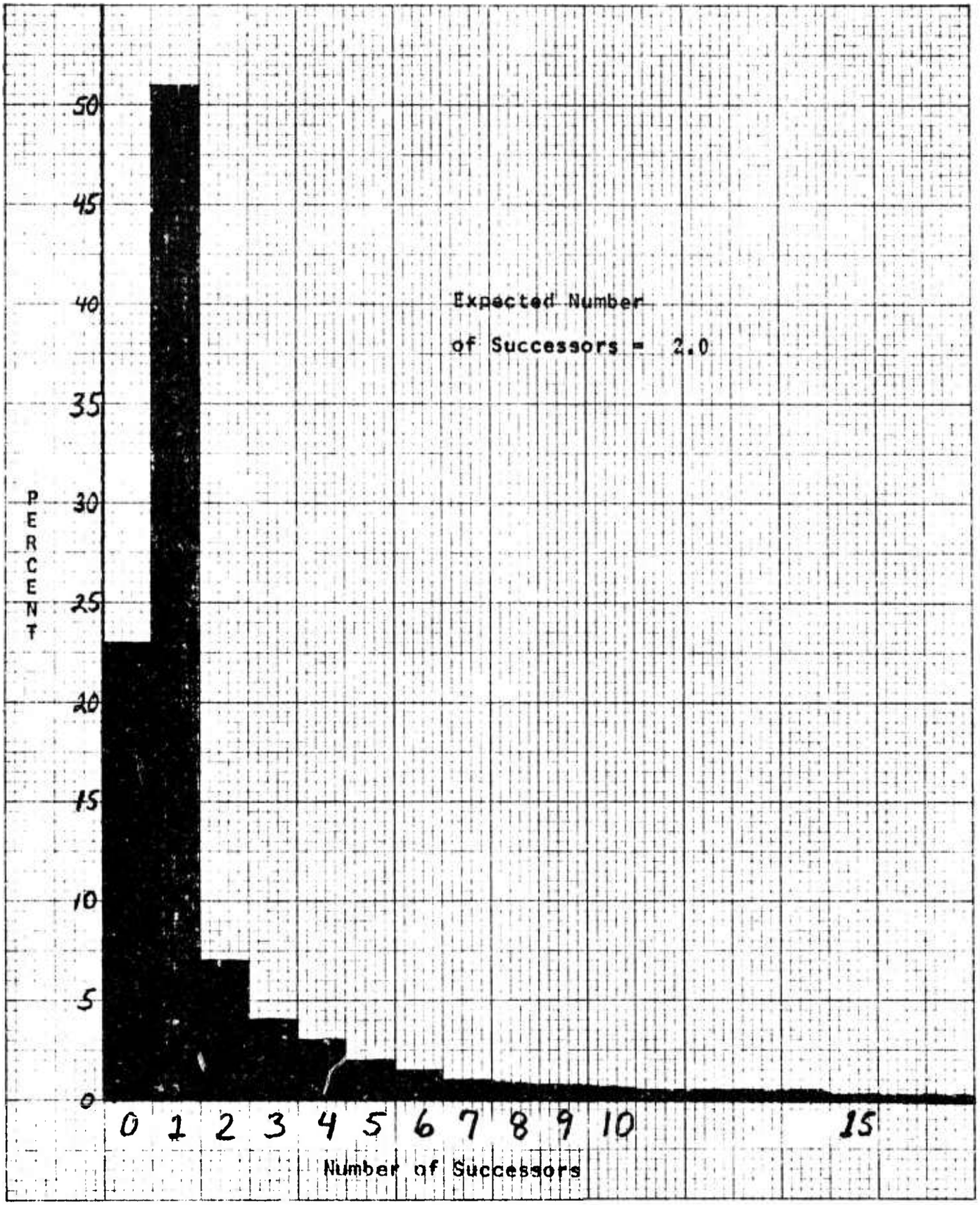

FIGURE 4.4 - Successor Distribution

Before analyzing the mechanisms in the program that produce the spectrum shown, another important phenomenon having to do with the structure of trees generated by this program will be examined. It turns out that the expected number of successors -- 2.0, found above does not predict actual tree size correctly. One would normally do this by assuming that the number of bottom nodes is equal to the branching factor raised to the "maximum depth" power. Here the formula predicts 1024 bottom nodes for our depth 10 searches. Since the average number of bottom nodes was between 50 and 60, this prediction is too high by a factor of 20. Further data collected shows that the assumption that there is an equal likelihood of a node sprouting, regardless of where it is in the tree, is incorrect. In fact, there is a strong dependence between nodes at successive depths. Thus a negative correlation exists between the expected value of the number of successors at a node and the number of nodes its parent had (see also section E below). The expected number of successor-successors that a node has can be seen in Figure 4.5 to be 2.3. If the branching factor for two ply of search is 2.3, then it must be 1.5 for a single ply. This turns out to predict actual tree sizes on this task very well. This low growth rate, compared to the branching factor of 5 to 6 for more conventional programs, makes it possible for this program to go quite a bit deeper in its search than these other programs.

Data taken on game fragments had all the above characteristics, except that the branching factors were larger. Thus the expected number of successors at a node was 2.5, and the expected number of successor-successors was 4.8. This comes out to a branching factor of about 2.2. Apparently in positions which have a sharp tactical character, such as the ones in "Win at Chess", the program does less searching before coming to a conclusion (right or wrong) than it does in positions which are less sharp and represent the more usual state of affairs. In all the data we collected, the highest branching factor ever encountered was in a game fragment; it was about 3.0.

Let us now consider the spectrum of likely number of successors of non-bottom nodes as shown in Figure 4.4. That over 50% of all these nodes have but one successor indicates that most problems the program deals with are well defined in terms of its ability to handle them. This does not mean that all problems the program investigates are worthwhile ones, in terms of some high human criterion, but merely that the problems it proposes to investigate are usually well formed enough so that a solution can be reached based on the first alternative tried.

The grossest exceptions to this occur at the right-hand tail of the curve. Here the program clearly does not understand what the problem is and thrashes about exhausting suggested alternatives until it either finds an acceptable one or exhausts the supply of suggested moves. The state in which this most often occurs is the DYNAMIC DEFENCE state. Here there are frequently many moves suggested to counter a particular problem. While the set of suggested moves is almost always adequate to the need, the methods of assigning heuristic values to moves proposed in response to these descriptions is still not very good. As a result, the program sometimes has to go through a large set of moves before finding an adequate one. However, it is felt that this problem is not of a permanent nature and only requires the application of some effort to improve the assignment of heuristic values, together with the possibility of further monitoring of the progress of the defence which could lead to the application of additional causal information.

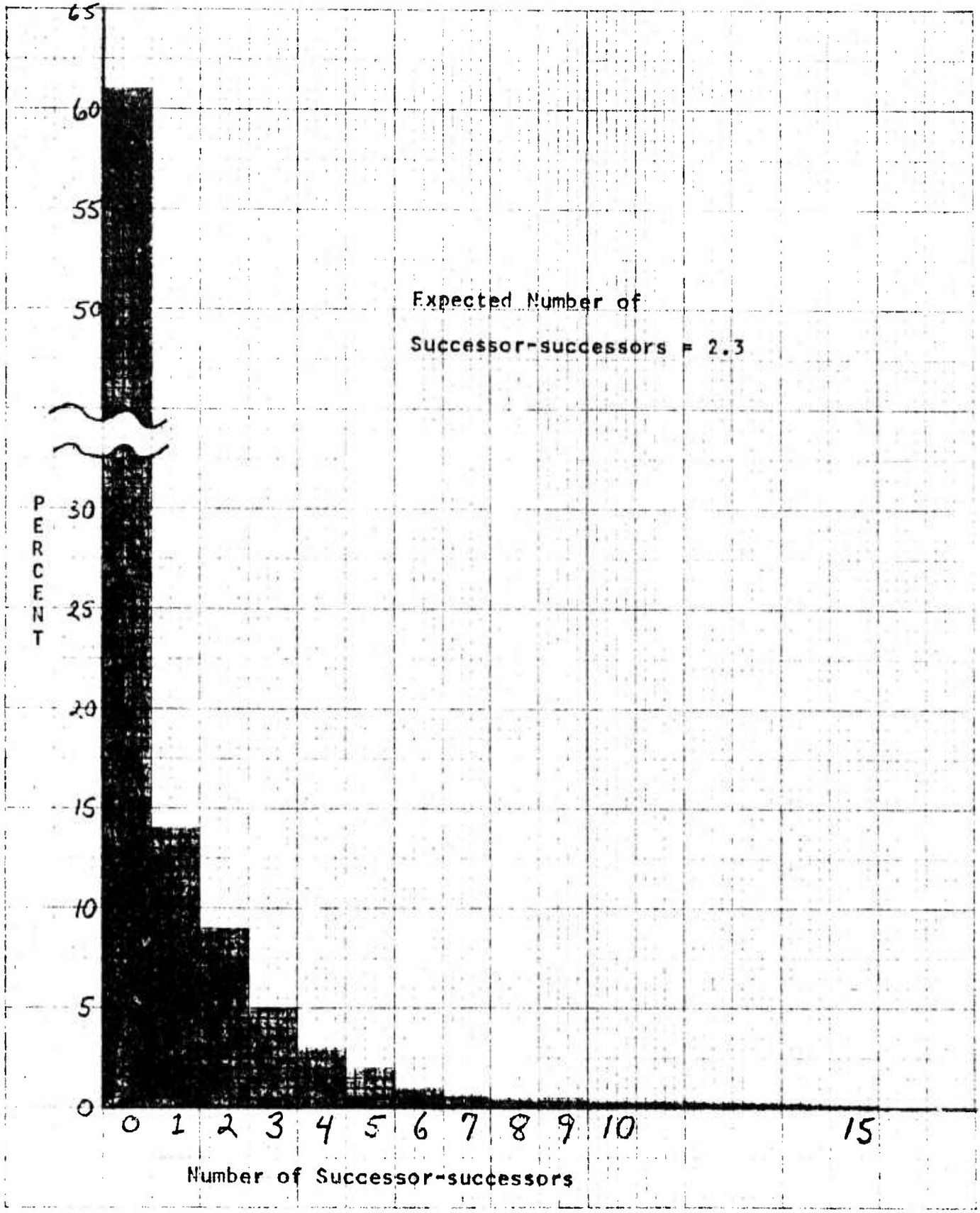

FIGURE 4.5 - Successor-successor Distribution

The real reason for the success of the tree searching paradigm lies in the ability to stop the search at any arbitrary node, based on the semantics of the node in relation to the problem environment. These results are presented in Table IV-1. The data were collected over 37 randomly selected problems from "Win at Chess" and 24 consecutive positions in a game fragment played by CAPS-II. The figures given are as a percentage of the total number of nodes in the trees.

TABLE IV-1 - Performance of Tree Control Devices

Effect	Problem Positions	Game Positions
Forward Prunes of Proposed Moves	101%	98%
Nodes at Maxdepth	23%	25%
Zero-Successor Nodes not at Maxdepth	16%	16%
Nodes Terminated by Alpha-Beta	37%	33%
Found Result Sig. Gtr. EXPCT	2%	1%
Found a Satisfactory Defence	1%	1%
Found no Defensive Moves	0.3%	0.4%
Position not Worth Defending	6%	3%

Let us try to interpret these results. The total number of forward prunes of proposed moves is approximately equal to the total number of moves searched. This points out several things. For one, the device is indispensable, since otherwise the total amount of work that the program would have to do would increase tremendously. This is due to the fact that not only would twice as many nodes have to be examined, but in all likelihood most of these would have successors which in turn would have sucessors until the reason for the static rejection of the move is discovered dynamically. Secondly, the reason that the forward prune device gets much work to do is that many inadequate moves are proposed. This is an inevitable consequence of the proposing mechanisms, which attempt to understand the underlying structure of the position and propose any move which affects this structure in a desired way. Clearly, many of the moves proposed will affect the structure, but cause some patently obvious other problem in the process. These moves could be weeded out right after evaluation takes place; however, we prefer to keep them on the move stack for examination in other eventualities.

The above data show that about 24% of all nodes in the tree are bottom nodes. This compares with 16% of zero-successor nodes at higher levels in the tree; these being due to some of the above specified mechanisms. Thus 40% of all zero-successor nodes in the tree are self-terminating (not a function of maximum depth). If one adjusts this self-terminating 16% of all nodes to consider as a base only non-bottom nodes, then 21% of all nodes eligible for sprouting are found not to sprout. This corresponds well with the 23% of Figure 4.4; that data coming from a different run of the program. The detailed reasons for the occurrence of zero-successor nodes are analyzed below.

From Table IV-1, one can see the important role of the Alpha-Beta pruning algorithm. The number of such prunes is equal to about 35% of the total of all nodes searched. The Alpha-Beta pruning algorithm is a well studied device [Slagle and Dixon, (1969)]. It is known that it cannot generate logical errors in the search, and that its maximum efficiency is proportional to the square root of the number of bottom nodes that would be generated under minimax. In this program its efficiency is hard to

estimate since the branches that are being cut off are of varying lengths. This is due to the fact that the cut-offs occur at varying depths and many backed-up values do not originate at the maximum depth (due to non-bottom zero-successor nodes). However, Alpha-Beta clearly contributes significantly. This is shown by the data, and also by the fact that the Search-and-Scan algorithm (which essentially disabled Alpha-Beta) produced so much larger trees in many cases.

The percentage of zero-successor nodes in a tree search is about 21% of all non-bottom nodes. There are four tests that can terminate a non-bottom node without a successor. These are: 1) Alpha and Beta equal or crossed, 2) Ahead of EXPCT in material and opponent has no compensating counter-threat, 3) No acceptable moves suggested by any state, and 4) Position repetition noted.

TABLE IV-2 - Causes of Zero-Successor Nodes

Effect	Problem Positions	Game Positions
Alpha and Beta equal or crossed	60%	85%
Significantly Ahead of EXPCT	3%	1%
No Satisfactory Moves Proposed	36%	13%
Position Repetition	1%	1%

Table IV-2 shows the relative frequency of occurence of each of these cases. The most obvious thing that stands out from these data, is the overwhelming role played by the Alpha and Beta equal pruning rule. The fact that it accounts for about three-fourth of the zero-successor nodes must in large measure account for the slow growth rate of trees in this paradigm. The Alpha and Beta equal rule is in turn made possible by the Claim System. The only way, that Alpha and Beta could be (become) equal while descending in the tree, is for an agency to be able to narrow the range between them during this time. Thus the Claim System examines the Alpha at a new node, in an attempt to limit it to the maximum effect obtainable by the opponent in the position.

The next most powerful effect, which accounts for almost all the rest of the action, is the fact that in about one-fifth of all positions that are terminated without search, no satisfactory move was proposed. This effect appears to be due to a combination of mechanisms. First of all there is close control on the goal states which are allowed to propose moves. Second, the evaluation of moves (while far from excellent) is adequate enough so that moves that are proposed to solve some problem but fail to solve a more global requirement, are evaluated correspondingly. Thirdly, the forward prune device, which refuses to search any move that does not evaluate to be equal or better than the current Alpha value, causes rejection of moves so that when no adequate move has been proposed, none will be searched.

It is interesting to note that the two devices that are most commonly written about and implemented in the cause of node termination -- significantly ahead of EXPCT, and position repetition -- account for only about 2% of the action between them. Of these, the significantly ahead of EXPCT test is usually implemented by setting Alpha and Beta at the top of the tree to EXPCT plus and minus MARG and then finding Alpha-Beta prunes which show that a position ouside these limits has been found. The fact that two orders of magnitude more prunes are possible in our program shows that sensitivity to other parameters of a position can have a big pay-off. It should be

pointed out that since malerial balance is the only term in the terminal evaluation function, the likelihood of Alpha and Beta becoming equal is considerably greater than if the terminal evaluation also included small values for other terms such as mobility, etc. However, far from being a criticism of the present approach, this just points out the importance of not mixing tactical and positional values in the search, and rather treating them in hierarchical fashion.

Since the data were collected across a large number of positions involving about 50,000 nodes in total, we feel some significance can be attached to the difference in performance of the first three tests on the two lasks. It is quite reasonable to assume that the average game position was more bland in character than the tactically sharp positions of "Win at Chess". The likelihood of not being able to propose any move which met a node's requirements appears to be higher in sharp positions, and this is borne out by the data. Such posilions are also more likely to produce a node in which one side is significantly ahead of EXPCT in material, and since the board state is turbulent, less likely to produce a situation in which Alpha and Beta can be squeezed together by the Claim System. This appears to be a reasonable interpretation of the data in Table IV-2.

Once searching from a node has begun it is possible to return to that node with new information that makes it unneccessary to search further at that node. One such item of information is a backed-up value leading to an Alpha-Beta prune. However, there are other mechanisms in the program which can also terminate a node, but depend on the value of EXPCT and on the current goal state for their power. The following mechanisms can terminate a node when returning to it: 1) Alpha-Beta pruning, 2) A result significantly better than EXPCT has been found, 3) A result greater than or equal to EXPCT was found while pursuing a defensive task, 4) The current defensive task is hopeless (all suggested moves have been rejected), and 5) The current position is not worth defending (after having rejected all AGGRESSIVE moves). Table IV-3 shows the relative frequency of occurence of each of these as a percent of all non-bottom nodes with one or more successors.

TABLE IV-3 - Reasons for Termination of Nodes with Successors

Effect	Problem Positions	Game Positions
Alpha-Beta Cut-off	61.6%	56.2%
Found Result Sig. Gtr. EXPCT	3.1%	1.8%
Found a Satisfactory Defence	2.0%	1.1%
Found no Defensive Moves	0.5%	0.6%
Position not Worth Defending	10.0%	5.1%

Again the effect of these devices is somewhat greater in the sharp problem positions, than in the game positions. This is part accounts for the larger branching factors obtained in the latter type of position.

Alpha-Beta plays a dominant role in this set of mechanisms. However, it is impossible to tell from the data collected how many of the Alpha-Beta prunes would have been realized without the operation of the Claim System. In those cases where Alpha was reduced without producing an "Alpha and Beta Equal or Crossed" condition, the reduction must have helped in producing later Alpha-Beta prunes. It is important to

understand exactly what the meaning of an Alpha-Beta prune is. When such a prune occurs, the last move tried at this node was good enough to produce a refutation of an earlier move by the opponent. Thus it is not necessary to look at any more moves at this node. This phenomenon occurred in the vast majority of nodes which had sprouts, indicating that the program is spending most of its time refuting unsound moves. However, our statistics do not show what goal state produced the move that resulted in the prune, nor how many prior moves had been tried. But Figure 4.4 is a good estimate of the latter quantity.

The mechanisms "Found Result Significantly Greater EXPCT" and "Found a Satisfactory Defence" also are indicative of a degree of success at this node. However, in these cases a result is produced which could become part of a principal variation or else result in an Alpha-Beta prune at a node an even number of ply higher in the tree. In contrast, "Found No Defensive Moves" and "Position not Worth Defending" are admissions of inadequacy which will probably produce an Alpha-Beta prune at some higher node an odd number of ply away.

The "Found Result Significantly Greater than EXPCT" test shows that a small percentage of such results appear. Even so, it is clear that continuing the search in such cases is more wasteful than redoing the search if such a result survives to the top of the tree. Likewise, the notion of being on defence and having found a satisfactory one contributes a small amount to overall node termination.

The "Current Position Not Worth Defending" test is the complement of the "Position Significantly Ahead of EXPCT" test in Table IV-2. In such a case, the side on move is behind in material but appears to have some threats which could rectify the situation. After these threats have been examined, and found to be inadequate, the situation is re-examined in terms of material on the board and capture threats still pending (even though the latter were not immediately executable). If this evaluation still is not enough to reach the EXPCT level, the search is abandoned. The number of such prunes turns out to be many times greater than the complementary test; a result that appears to indicate that our methods of detecting threats give the benefit of the doubt to the party not on move. Then when these threats are found not to be dangerous, the other test prunes the node. This shows another advantage of the goal state approach, since it would otherwise be impossible to pause for re-examination after having examined all AGGRESSIVE moves.

Our statistics do not indicate how the remaining nodes that had successors were terminated. Logically this must have occurred either when the AGGRESSIVE state produced a satisfactory move, or in the KING IN CHECK state, or by exhaustion of all suggested moves.

E. A THEORETICAL ANALYSIS OF TREE GROWTH

There has been quite a bit of work done on trying to understand how search trees grow. This has been possible since CAPS-II has branching factors in the range of 1.5 to 3.0, making it possible to examine trees over a reasonable range of maximum depths without causing computationally intractable tasks. In order to understand tree growth, it is necessary to consider the anatomy of a node at sprouting time. If a position is tactically "alive" then it is possible for the move generator in charge (which in practice means the AGGRESSIVE or one of the DEFENSIVE ones) to generate many moves which

it feels can materially affect the value of that node. Whenever a move is actually tendered for searching, it must pass the forward prune test, which does not allow any move to be tested whose optimistic static value does not equal or exceed the Alpha value for that node. Further, if a result is ever backed up to this node, which is significantly better than EXPCT, then the search at this node is immediately abandoned.

Now, let us consider the optimum value (VAL) of the position at any node in the tree as being the minimaxed result of a complete search of that node's sub-tree. Assuming the position is tactically alive, then the number of moves that will be searched is a function of how many moves are statically evaluated as better or equal to Alpha, and the algebraic relation of Alpha, VAL and EXPCT to one another. If one allows the relations: equal (=), somewhat greater (>), significantly greater (>>), somewhat less (<), and significantly less (<<), then 125 cases can be distinguished. Let us associate ">" with being less than MARG greater, and ">>" with being MARG or more greater. Then if A > B, and B > C, then either A > C or A >> C. There are then 31 valid relationships among Alpha, EXPCT, and VAL. Many of these cases fall into classes where one relationship is of over-riding importance. Let us examine these classes.

If Alpha >> EXPCT, then by the rules of our search no more sprouts will be looked at, and the search backs up. This takes care of six cases leaving 25.

If EXPCT > or >> VAL, then we are trying to solve an impossible problem at this node since we can never find a move that will come up to EXPCT. Hopefully such cases are rapidly disposed of in the present program by causality. If that fails then many moves could be searched in vain. This takes care of 11 more cases, leaving 14.

If Alpha > or >> VAL, the program is again wasting its time at this node since it has already achieved a better value than it can hope to find here. The forward prune test must be counted on here to dismiss many proposed moves as being inferior to what has already been achieved. However, if Alpha is only slighty larger than VAL, or if the situation is full of unresolved tactical complexity, then the forward prune could very well not be sensitive enough to eliminate a significant portion of the work at such a node. This class encompasses nine cases, but only three new ones, leaving 11.

If VAL >> EXPCT then we can find a solution at this point which will allow backing up as soon as such a significantly greater value is found. Experience has shown that the program is usually able to find such highly successful moves quickly, thus this class is of little concern. There are five cases in this class, leaving six.

The remaining six cases are:

1) Alpha = VAL = EXPECT. Here no improvement is possible and the forward prune can not be of any help since any proposed "good" move must be at least a little better than Alpha (due to optimism). Therefore much searching could take place here while the program is in the AGGRESSIVE state, trying every possibility in the hope of finding a move that makes a difference. We conjecture that this case is a major cause of higher branching factors.

2) Alpha < EXPCT < VAL >> Alpha. Here improvement of the existing Alpha is possible, however, since no value >> EXPCT is achievable, the program may likely have to search through all proposed alternatives. There are two exceptions to this: in a

DEFENSIVE state the tree search is exited when a value ≥ EXPCT is found, and, if the best value at this node is found quickly, the new Alpha may produce some forward prunes of the remaining moves.

3) VAL = EXPCT >> Alpha. This case is very similar to the one above, except no forward prune help can be expected.

4) Alpha = EXPCT < VAL. If this occurs in a DEFENSIVE state, the program will back-track. Otherwise, if the small improvement in the position is found quickly then hopefully forward prune will cut down the remaining work.

5) Alpha = VAL > EXPCT. Here no improvement is possible. In a DEFENSIVE state back-track occurs, but in other states no help from forward prune can be expected.

6) VAL = EXPCT > Alpha. Here some improvement is possible and termination can occur quickly if in a DEFENSIVE state. Otherwise, much searching will occur until all alternatives are exhausted.

On examination of the above cases, it becomes clear that many states exist where the program will engage in what appears to the informed observer (for he knows the value of VAL while the program does not) as generating and testing. This occurs when the program is in the AGGRESSIVE state and no VAL >>EXPCT is achievable. While we can think of no sure way of preventing this behavior, it is conceivable that limiting the number of AGGRESSIVE tries at a node when EXPCT has already been achieved is a good possibility. This could be reasonably safe if it were only applied to the analysis moves of the program's side, and if (say) the top four ply of search were unaffected by this rule.

In actual play two types of situations have been identified; those in which the program can find what is to it a clearly best move, and those situations in which it can not. By clearly best, we mean that the backed-up value assigned to such a move be at least MARG more than any other value that can be backed up. In situations where there is no clearly best tactical move and the position is tactically "alive", the program may go into an extended search, as the chance of a damaging situation from one of the above classes occuring is very high. If the situation is not tactically alive, then the search will usually terminate relatively quickly, with some tactically acceptable move which was either proposed by the AGGRESSIVE state or the STRATEGY state becoming the move made. In situations where a clear-cut solution exists, the program will usually find it in less than five minutes of CPU time; occasionally taking up to 10 minutes in cases where it makes many false starts. It has also been noticed that in positions that have a sharp character (as for instance those in "Win at Chess"), or in positions where the program thinks it has found a superior move, the branching factor across the whole tree is approximately 1.5. This would be very tolerable, if it were valid for all positions. However, in positions that are tactically alive with many possibilities but no clear-cut solution, the branching factor tends to go up to 3.0, which while considerably better than the 5.0 to 6.0 found in most of today's programs, still tends to produce rather gigantic trees for 10-ply searches. For instance, with a constant branching factor of 3.0, a 10-ply tree would contain almost 30,000 non-terminal nodes, as against 112 non-terminal nodes for a branching factor of 1.5.

Below, three examples of tree growth are examined. The experimental technique used was to run CAPS-II at varying depths on the same problem and note the width of the resulting trees. By the width at depth "n", we mean the total number of nodes that were sprouted in the whole tree at depth "n" during the search. We will try to show that there is a pronounced difference between the growth patterns of trees, depending strictly upon the notion of a clearly best move. To do this, we examine the same position under search conditions which allow finding a clearly best move and under conditions which do not.

The first example derives from the position in Figure 4.6, and is one in which CAPS-II is unable to find a clearly best move no matter how deep the search, up to 15 ply (for an explanation of this see Figure 5.12). These results are shown in Figure 4.7, which gives the results for depth 5, 7, 9, 11, 13, and 15 searches. The graph shows a general exponential growth of the tree with increasing depth. Actually the width of the tree at a given depth tends to decline very slightly as the depth of search goes up (we call this the "Flattening Effect"). However, in any individual tree one can at best hope to see a slight slowing of the exponential growth. This could be due to such factors as 1) greater opportunity for cut-offs at greater depth because of the way the Claim System works to press Alpha and Beta toward each other, or 2) depletion of material making it more difficult to come up with meaningful tactical issues. However, both of these effects seem to be of a very low order. Although they probably exist, they are very much subordinated to the effects that cause individual positions to be more or less tactically alive.

The following points should be noted with respect to this example and the next: Whenever the curve moves in a sideways or slightly downward direction it means that the side on move is coming to grips with whatever the problem may happen to be (or conversely it may be posing problems) at the rate of approximately one node for every parent node. On the other hand, when there is a considerable vertical rise it means that the side on move here is either having to search a bit to find the right response in order to raise Alpha up to EXPCT, or it is busy generating one offensive try after another without much success. This frequently results in a pattern of a side-wise step followed by an exponential rise (the step phenomenon). This indicates that one side (the exponential rise side) is either busy trying to pose issues which the other side has little trouble turning aside, or it is having trouble reacting to a threat. In either case, the exponential rise is an indicator that this side is having problems coming to grips with the position.

The second example derives from the position in Figure 4.8, and the tree growth graph is shown in Figure 4.9. This graph shows growth curves for depth 7, 9, 11, 13, and 15 searches. To understand what happens here, it is important to know the solution to the problem which is: 1. R-R8, after which White threatens to queen the pawn, winning, while Black cannot play 1.--RxP because of 2. R-R7ch winning the rook. So Black's only alternative is to keep checking the White king on the rank, which for most computer type analyses, will result in postponing any real consequences over the search horizon. However, White wins the position quite easily by merely approaching the Black rook with his king until an effective check can no longer be given. However, this requires 13 ply of searching as follows: 1. R-R8, R-R8ch, 2. K-B2, R-R7ch, 3. K-K3, R-R6ch, 4. K-Q4, R-R5ch, 5. K-B5, R-R4ch, 6. K-N6, R-R8, 7. P-R8=Q. Therefore the program must search to depth 13 in order to find a proper solution to this problem. In that light, it can be seen that the curves for depths 7 through 11 show a general

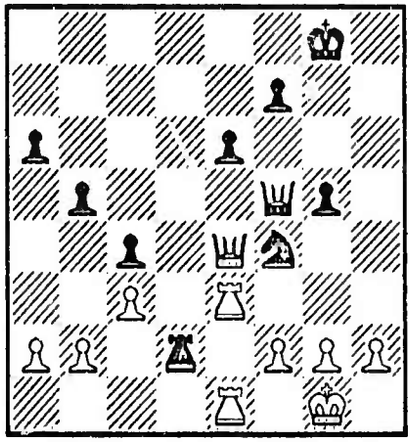

Figure 4.6

Black to Play

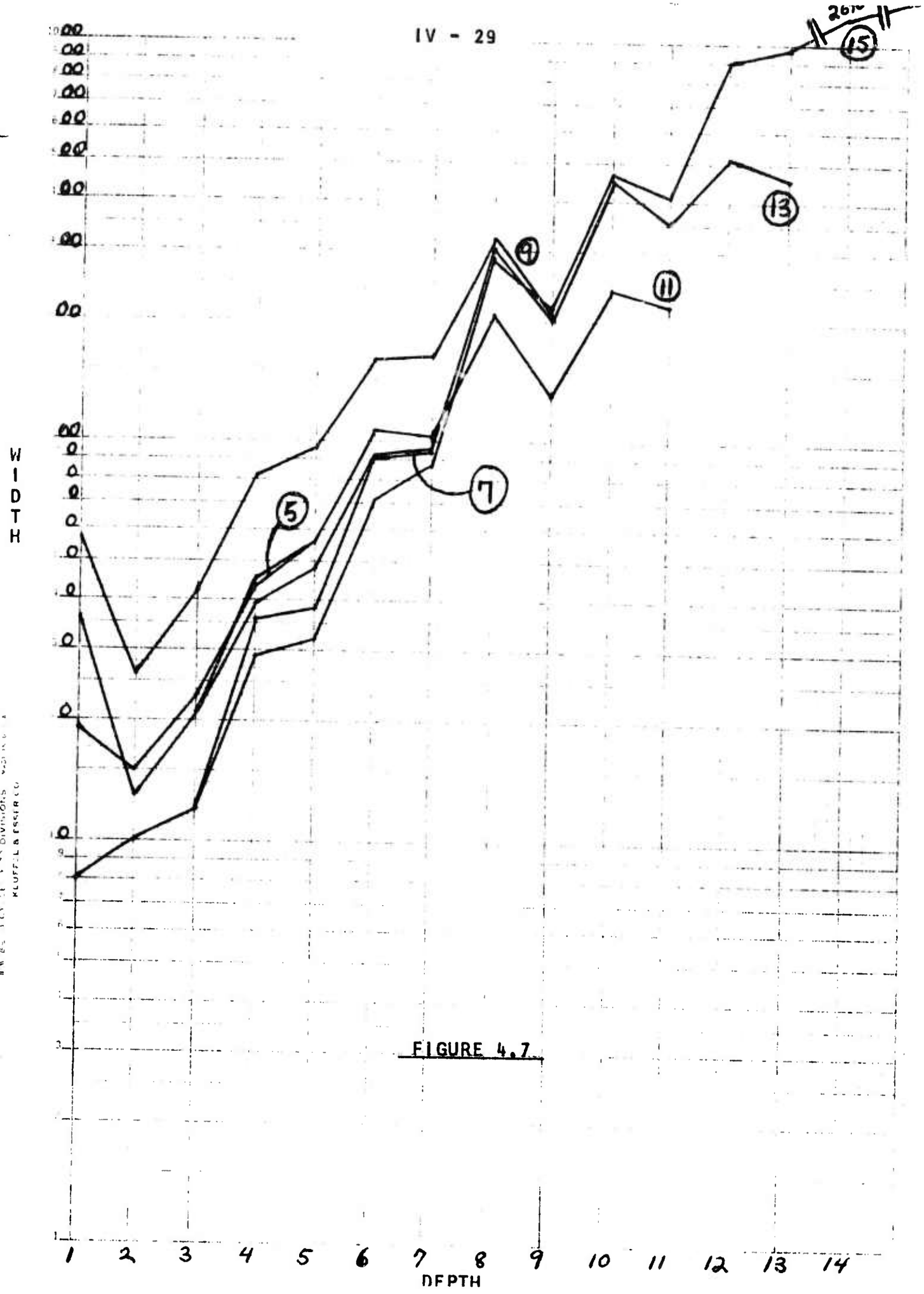
WIDTH
DEPTH
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
SEMI LOGARITHMIC
46 5490

FIGURE 4.7

exponential increase as the program looks in vain for a solution and fans out in the process of doing so. It is interesting to note the step phenomenon at the end of each of these curves, but not prior to that. The flat part of the curve is always coming on the last even ply, which would make it Black's move, and the rise is coming on White's reply. The explanation for this, is that Black is trying the capture RxP on this ply and now White plays R-R7ch. We are now at maximum depth and the procedure for a situation where the king is in check is to use the evaluation of the best of all legal replies. Here, any king move results in a situation where both rooks are attacking each other. This is again in an area of incompetence of the current terminal evaluation function, and thus this position is considered as nearly even. Because of this, it appears that the move RxP is successful (but only when made this close to maximum depth) and White continues to look for alternatives to R-R7ch. It investigates R-B8ch as the only meaningful alternative, but finds that Black's answer KxR is too strong. This drama is played out again and again at the horizon of each search, and accounts for the step phenomenon tails which show that White is having a problem which Black has no trouble coping with at this point.

The first break in this pattern is for the curve labelled 13W. In this curve the program thought it found a superior move (the right answer). However, it only managed to get to a situation in which it got a favorable (but incorrect) verdict from the terminal evaluation function. The principal variation for that search was: 1. R-R8, R-R8ch, 2. K-R2, R-R7ch, 3. K-R3, R-R6ch, 4. K-R4, R-R5ch, 5. K-R5, R-R4ch, 6. K-R6, R-R3ch, 7. K-R7. In this position the terminal evaluation function gives more credit for the threat to promote the pawn than for Black's threat to capture it. Therefore this position is considered as constituting progress for White. We are aware of this problem in the terminal evaluation, and it will be fixed in an early future version of the program.

The reason the king marches up the edge of the board is that there is a heuristic in the program which rewards decentralizing the king. This is very useful in middle games, and since almost all tests of the program were in middle game situations, no attempt was made to provide additional sensitivity to which phase of the game the current situation is in. This can be easily done by something as simple as counting pieces to decide that the end-game heuristics should now be used. That would call for king centrality instead. With that in mind, we put a temporary patch in CAPS-II which rewarded centralizing the king, and ran the problem again. This time the curve labelled 13R was generated, and the principal variation was: 1. R-R8, RxP, 2. R-R7ch, K-N1, 3. RxR, indicating that the program had been unable to find anything better for Black than losing his rook. A look at the tree print-out confirmed that it had indeed found the correct solution. It is interesting to note that this curve is considerably lower (comprising about one-half the number of nodes) than the curve 13W. This indicates that the more appropriate guidance provided by the "centralize" heuristic not only got to the right answer, but considerably shortened the solution process.

A final curve with the "centralize" heuristic is the one labelled 15R, which is run at depth 15 just to determine the additional amount of work the program does when given more depth freedom. The thing that is very noticable about Figure 4.9 is the definite curve flattening, once the program has reached the proper solution depth. In Chapter VI, we will speculate as to how this problem would be solved using lemmas.

The final example in Figure 4.10 shows curves for depth 9, 11, 13, 15, 17, and 19 searches of a position (Figure 4.11) that CAPS-II did not understand, but in which it

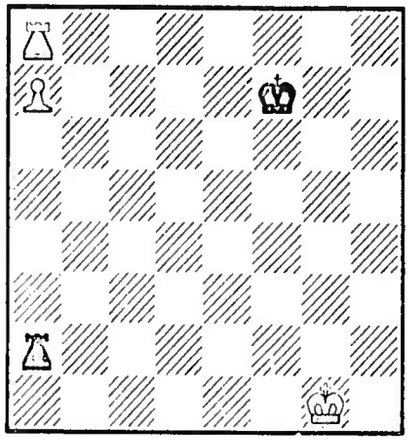

Figure 4.8

White to Play

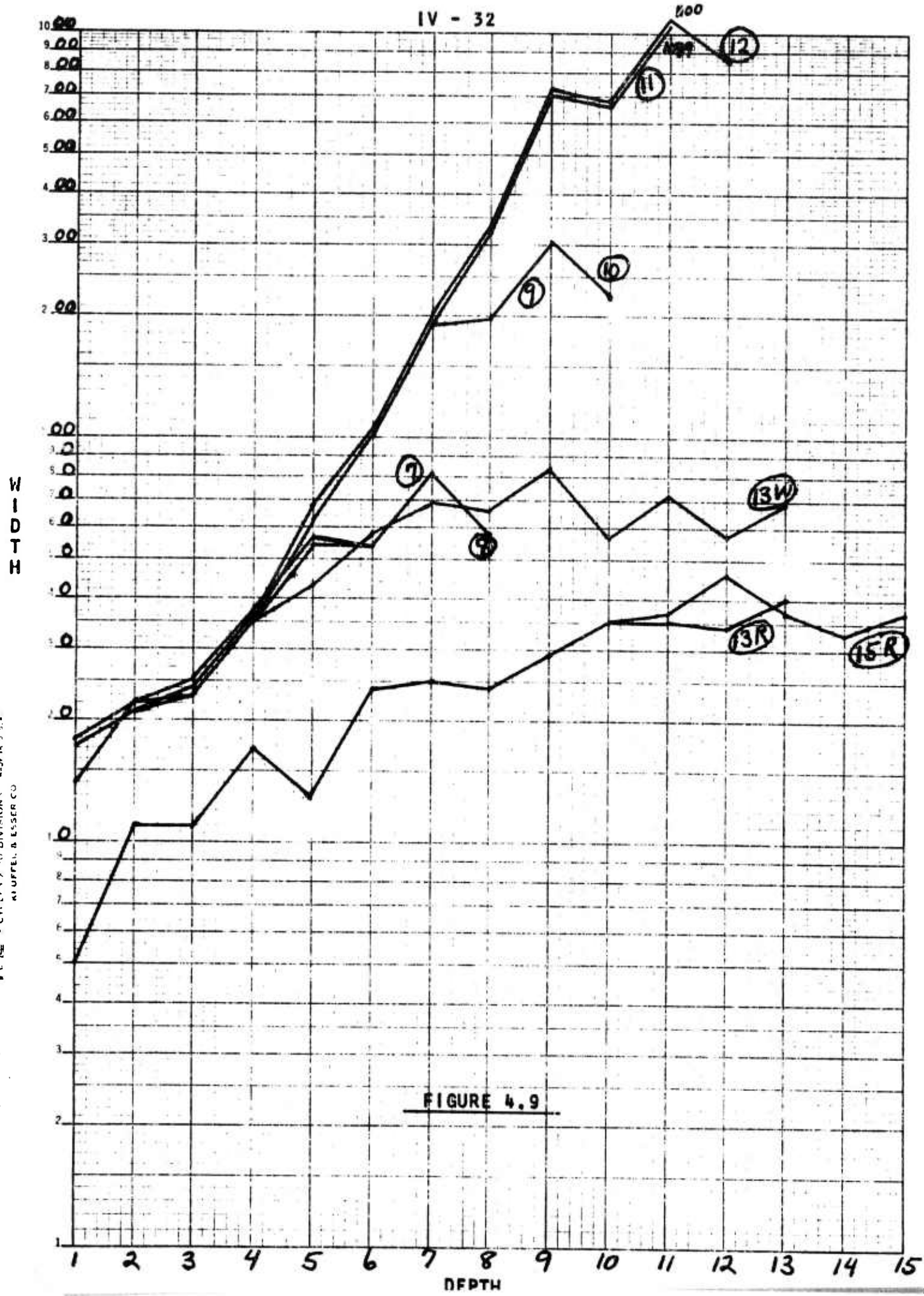

FIGURE 4.9

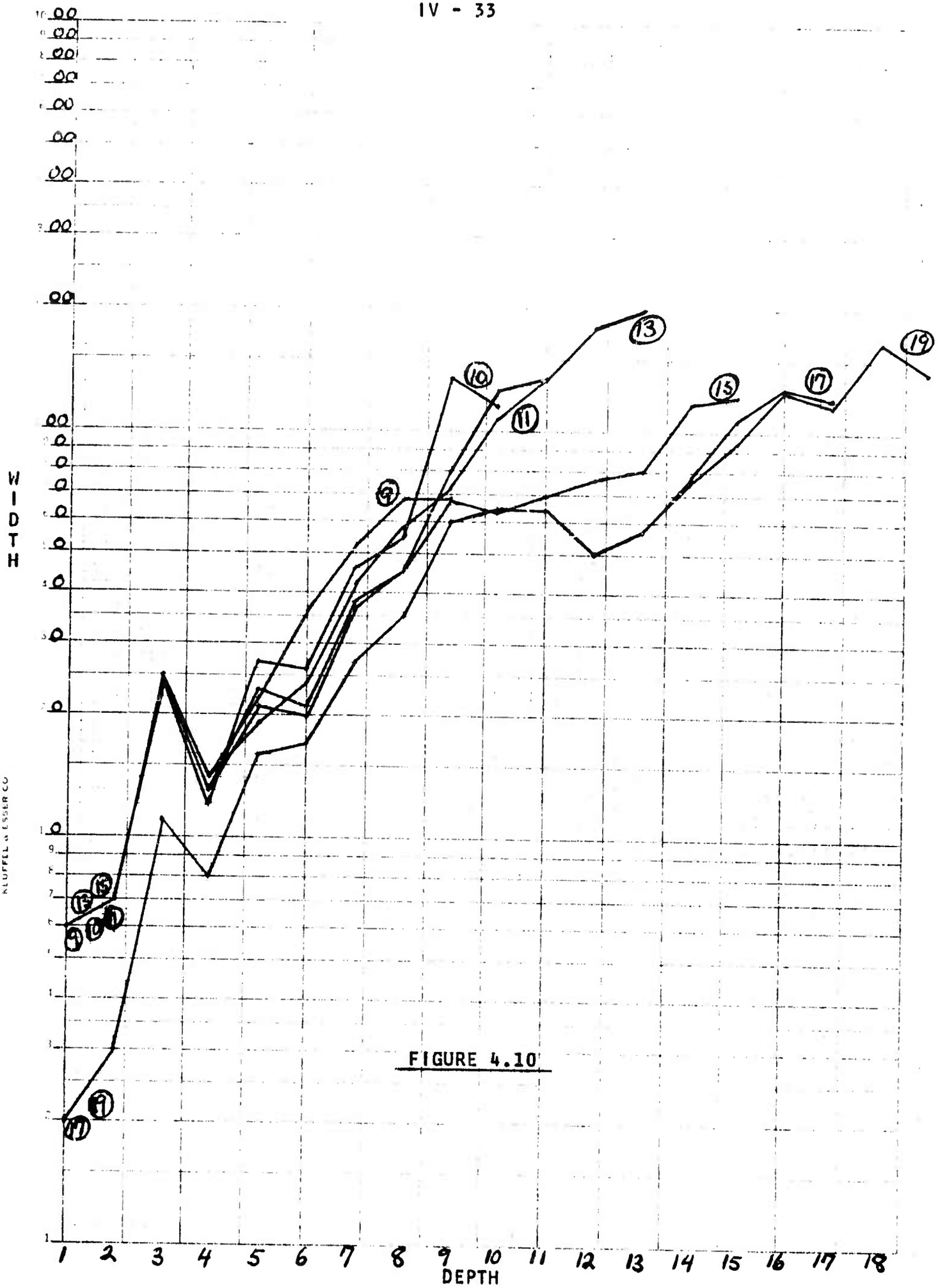
WIDTH
DEPTH
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
SEMI LOGARITHMIC 46 5490
KEUFFEL & ESSER CO

FIGURE 4.10

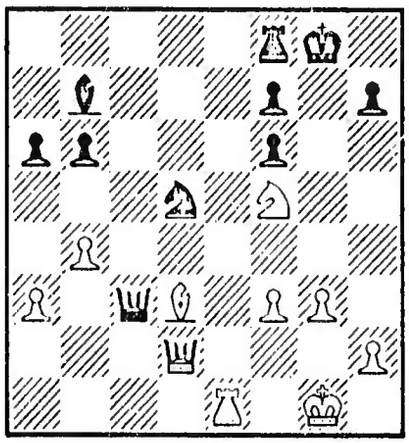

Figure 4.11

White to Play

found a (to itself) satisfactory solution. Although the problem can be solved at well below 9-ply depth, the program's perceptual mechanisms were not good enough to discover the key move. However, it finds a solution to what it conceives of as the main problem. This same solution comes out again and again as the depth of search is increased. Since the position is tactically alive, the program acts as if the critical depth for solving this problem is well below the nine ply that is represented in the first curve. Here we see a perfect illustration of the flattening effect. The curve at each increasing depth has a smaller slope than any before it. This is a real phenomenon, and is probably due to the fact that once a "solution" is found, it is possible to get many cut-offs and thus slow the growth of the tree.

One final observation is useful in trying to understand tree growth. That is that most positions appear to have their own step phenomenon characteristic. If this is true, then in each position the play for one side is relatively easy and for the other relatively difficult. The step phenomenon can be seen in the three graphs we have just presented. For instance in Figure 4.7, nodes generated in moving from even to odd plies move sideways, and those from odd to even move up. In Figure 4.9, odd to even move sideways, and even to odd move up. And in Figure 4.10, there is a very pronounced effect of odd to even moving sideways, and even to odd moving up. This step phenomenon has been reported by other researchers and is almost certainly related to the workings of the Alpha-Beta pruning algorithm. For instance, if one can correlate one side establishing a good Alpha value near the top of the tree, with this being an "easy" position for that side, then one can see how this would make it easy to turn aside voluminous but meaningless gestures by the other side.

CHAPTER V

PERFORMANCE OF THE PROGRAM

In this chapter, we examine the tests that have been performed using the program. These tests range from very simple ones on the effectiveness of the data structure, to full games played by the program. After each presentation of results of the tests, we present those conclusions which seem warranted, and pose those open questions that were raised and appear unanswerable at present.

A. A TEST ON THE REPRESENTATION

Since all of the program's ability to understand structure in chess is dependent upon the primitives it computes in every position, a great deal of effort was of necessity expended to see that such primitives yielded correct results as far as could be determined. This was certainly achieved for simple things such as the different bearing relationships. Also, things such as the correct noticing of pins, correct assignment of functions, correct noticing of the various types of attacks, the causality facility, etc. were established to be very reliable. However, the occupiability (OCY) computation was not always correct. This is of concern since many judgements CAPS-II makes depend on this value, and when errors exist these may cause lost opportunities or increased tree searching activity in order to recover. In view of this, a short test sequence was run on the first 20 moves of Game I of the recent Fischer-Spassky chess match. The test sequence involved 40 positions, with 64 squares per position and two values (White and Black) per square. This comes to 5120 observations. Over this set, CAPS-II computed 3 occupiabilities wrong, e.g. an error rate of approximately six-hundredth of one percent. The author in checking the output of CAPS-II made approximately 6 to 8 times that many mistakes in the checking process, that is, there were on the order of 20 situations where the author thought the computed result was wrong at first glance, but it turned out to be correct. This appears to indicate that the program's perceptual accuracy in this dimension is already superior to that of a master, and probably the present degree of refinement of the computation is too high. Nevertheless, all the errors turned out to be due to situations where a pinned piece's influence on a square was wrongly calculated. We now know how to fix this problem and expect to do so during a future program revision.

B. TESTS ON CHESS POSITIONS

Most of the development and testing of the program was on the set of problems contained in the bock "Win at Chess" [Reinfeld, (1958)]. This was because this volume is rich in examples of functions being performed by pieces, and how disturbing those functions can lead to tactical combinations. The program was originally constructed with a view toward solving these problems, and many of the features in the program arose from considerations of what it would take to solve specific problems in the set. When an initial version of the program that seemed to perform somewhat as desired was first brought up, it was tested on the first four problems in "Win at Chess" plus three other problems which the author had selected as important. The program was debugged and small improvements were made on it until it reached a level of performance which appeared reasonable, considering the program's basic problem solving facilities. At this time, the program could only solve one out of the first four problems in "Win at Chess" correctly, and had not been tested on any of the others. This version of the program

was then tested on the first 20 problems and managed to get seven of these correct. This series of tests revealed further bugs which were corrected. The resulting program was named CAPS-I (Chess as Problem Solving).

1. CAPS-I Tests

CAPS-I was then tested on the first 100 problems in "Win at Chess". This book is organized into chapters of 20 problems each, with the intention that each succeeding chapter be more difficult than the last. However, there has been no standardization of this difficulty factor through experimentation. Thus, though there is a general difficulty trend, there seems to be quite a bit of variability from chapter to chapter. The tests of CAPS-I were run under the following conditions. If it failed to solve the problem by the time the number of nodes in the tree reached 500, the solving process was terminated and it was counted as wrong. In scoring these and all other results, it was usually quite easy to tell whether the answer was correct or not. However, in some cases the program got the first move correct, but the supporting analysis as evidenced by the principal variation and the tree print out were not convincing. In such cases one-half credit was given. Table V-1 shows the results of these tests.

TABLE V-1

Chapter	I	II	III	IV	V	All
Number right	10	11	12	10.5	5	48.5
Avg. nodes (right)	49.3	80.1	115.2	136.7	158.2	103.6
Avg. nodes (wrong)	65.0	37.0	60.0	64.0	53.5	56.1
Avg. nodes (wrong+unfin)	195.5	152.8	280.0	112.4	202.3	189.3

As can be seen, CAPS-I got 48.5 problems correct out of 100. In 15 of the problems, it had to be stopped at 500 nodes without having found a solution. This means that in 85% of the problems CAPS-I was able to deliver an answer within 500 nodes (about two minutes of CPU time). It is interesting to note that the average number of nodes required to solve the problems goes up as a function of increasing chapter number (difficulty). However, that is the only thing that correlates well with difficulty as it exists in the book. Other interesting data items are that the amount of effort on problems that were finished incorrectly is rather constant and usually less than the effort on problems solved correctly. Considering both wrong and unfinished in the same category by assigning 500 nodes for all unfinished problems seems to only obscure any effects that are present.

CAPS-I appeared to have a reasonable perceptual grasp of what the task was in a given position. What we mean by perception, here and later in this chapter, is the ability of the program to statically diagnose generally what the problem is and what moves are likely to play a part in the solution. The above is borne out by the data; the program usually terminating quickly. In cases where its static analysis routines could decode the position and find the key move at the top, the program did very well when little other analysis was required. In cases where the decoding was unsuccessful or where further complicated analysis was required, the program would either terminate in standard time with an incorrect answer, or go into a generate and test mode in more complex positions until it eventually ran out of

alternatives. In analyzing the output trees, one could notice several instances of logical failure in the tree search, which prevented CAPS-I from recognizing valid defences. This resulted in CAPS-I doing much better in problems where straight forward albeit perceptually hidden solutions were possible, than in situations where a certain amount of dynamic analysis was necessary. CAPS-I also played ordinary games of chess very poorly.

2. CAPS-II's General Performance

As a result of these tests, a few additions and improvements to the data structure were made. However, the most effort in the development of CAPS-II went into tree searching. It was found that there were logical inconsistencies in the way goal state changes were implemented, and this sometimes allowed CAPS-I to ignore problems by simply not going to the correct goal state, or leaving the correct one prematurely. The result of correcting this logic was a noticeable increase in tree size. However, this was absolutely necessary as such logic errors clearly could not be tolerated. Besides this, all the non-standard tree searching devices discussed in Chapter IV were developed for CAPS-II. As a result, this version of the program showed great improvement in application of its searching effort. Also the accuracy of its performance went up to the point where it was clearly in the class with the average program that competed in the annual ACM tournaments. The program was then retested, this time on the first 200 problems in "Win at Chess". This time it was decided to use a time criterion, instead of counting the number of nodes. Five minutes was set as a maximum time. The reason for using time was so that the results could be compared to the performance of other programs and human players. A tabulation of the performance of CAPS-II in the same format as Table V-1 above can be seen in Table V-2. For purposes of this table, the number of nodes was established using time in seconds multiplied by 4.5, which is approximately the average number of nodes processed per second.

TABLE V-2

Chapter	I	II	III	IV	V	All
Number right	12	14	12.5	10.5	7	56
Avg. nodes (right)	167.8	226.4	206.1	453.6	285.3	260.6
Avg. nodes (wrong)	227.2	269.1	346.5	371.7	297.0	291.6
Avg. nodes (wrong+unfin)	508.0	449.1	490.0	806.4	436.0	537.3

Comparing these results with Table V-1, we note that CAPS-II did 16% better on the problem set than did CAPS-I. This was a uniform achievement, CAPS-II not scoring worse on any chapter than CAPS-I. There were, however, five instances that were solved by the earlier version and not by CAPS-II. At first glance one might ascribe the better performance of CAPS-II to the fact that it had five minutes of time available, while CAPS-I had only about two minutes. To refute this idea, the 15 problems on which CAPS-I failed to finish were run for five minutes. Of these, CAPS-I was able to finish only one with a correct result, thus indicating that the discrepancy in performance was due to other factors.

TABLE V-3

Chapter	VI	VII	VIII	IX	X	All
Number right	7	8	5	12.5	6	38.5
Avg. nodes (right)	101.7	429.3	144.0	512.1	412.6	356.9
Avg. nodes (wrong)	238.5	231.3	458.6	319.5	526.0	375.6
Avg. nodes (wrong+unfin)	837.0	697.5	636.8	466.6	879.3	725.4

CAPS-II's performance on the second hundred problems in "Win at Chess" is summarized in Table V-3. Across the set of 200 problems, CAPS-II achieved 47.25 % correct. Attesting to the fact that the first one hundred are simpler than the second one hundred are the comparative percentages of correct solutions, 56% and 38.5% respectively. One can also detect a general increase in time taken by the program on all problems, whether solved correctly or not, as a function of chapter number. However, there is quite a bit of variability here.

In examining the trees generated by CAPS-II, it was possible to discern a much more workmanship-like approach to problem solving. In these trees issues arose and were dealt with in a satisfactory manner until an appropriate conclusion was reached. This was in clear distinction to the problem solving of CAPS-I which was highly perception dependent, and unable to follow issues very well to their conclusion. This is also manifested by the fact that CAPS-I could make practically no headway on the more difficult problems in later chapters of "Win at Chess", whereas CAPS-II did many of these very well and would have obtained a higher score but for several malperformances of the terminal evaluation function, and four cases of problems being solved in times of between five and ten minutes. We consider the logic of tree searching in CAPS-II to be very close to absolutely correct. However, an apparent penalty of this greater consistency in searching, has been a tendency for the program to search too much in situations in which it cannot find a solution that is clearly superior to any other. The answer to this would appear to be algorithms which still further restrict the moves which are allowed to be searched. Also, an effective method is needed for hopping around in the tree when results indicate that branches of greater promise than the one currently being looked at, have as yet not been investigated. Upgrading the uses of the representation, as suggested in examples later in this chapter would very likely raise the performance level of CAPS-II on these problems by 20 to 30%.

A graph showing comparative performances of a Class "A" player, TECH, and CAPS-II is shown in Figure 5.1. Here the task was the first 200 problems in "Win at Chess", and all solutions had to be achieved in five minutes (of CPU time in the case of the programs) or less. The Class "A" player achieves a clear superiority over both programs. In fact, his performance on any chapter is only exceeded once by either program, and that is by CAPS-II in chapter IX. This chapter seemed to be exceptionally well suited to the talents of CAPS-II as it scored much higher than would be expected considering the difficulty of the chapter. This chapter seemed to be largely made up of problems which required a clever first move, together with precise calculation of deep principal variations. More detail on this phenomenon is given later in this chapter. Over the whole problem set, CAPS-II did a few percentage points better than TECH. However, there were many reversals of form from chapter to chapter.

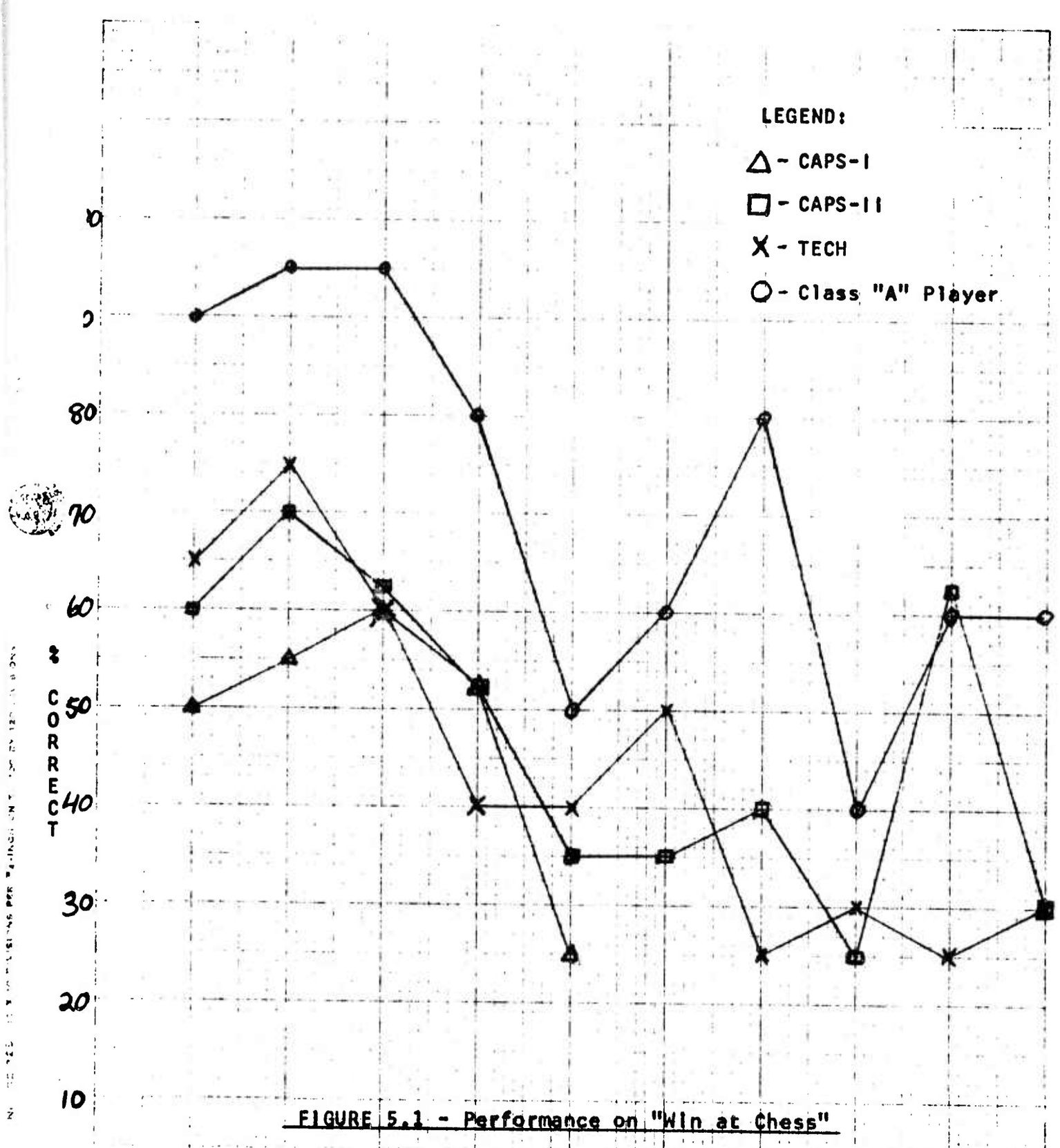

FIGURE 5.1 - Performance on "Win at Chess"

A more illuminating view of the relative performances of CAPS-II and TECH can be seen in Table V-4.

TABLE V-4

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.

This table shows the performance of TECH versus CAPS-II on individual problems as a function of the depth of the principal variation. 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.

The interesting thing about Table V-4 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 percentage of all problems, at every depth. The percentage increases slighly 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 thesis.

TABLE V-5

Depth	Total	% CAPS-I	% CAPS-II
1-2	12	62	62
3-4	48	56	71
5-6	21	45	48
7-8	8	38	38
9-10	5	40	40
>10	6	0	0

When one uses the depth criterion to examine the performance of CAPS-I versus CAPS-II on the first 100 problems, the results in Table V-5 are obtained. These show rather clearly that the main performance increment came on problems of depth three and four. This tends to bear out our earlier analysis that many of the problems were heavily dependent on perception. This would apply to the problems with short principal variations, e.g. one and two ply. The longer principal variations are associated mainly with problems with rather straigh-forward winning sequences not involving any intricate defences or many sub-variations. this only leaves the problems of medium length, many of which required detailed analysis of offensive and defensive ideas in order to bring in a correct answer.

TABLE V-6

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 V-6. 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 V-7

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 V-7 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.

Since the book "Win at Chess" was made up exclusively of examples extracted from Master play, it was felt that CAPS-II should also be tested on more mundane examples of the type that computer programs are likely to run into in their everyday existence. For this purpose a set of such tactical problems was extracted from the repertoire of the Northwestern University program (NWP) [Atkin, et.al., (1965)] as it performed in winning the first three ACM Computer Championships. This comprised a set of nine games which can be found in the open literature. We examined these games, and extracted from them 43 positions which had singular tactical solutions. This included all the most difficult positions in these games and most of those with medium difficulty, plus a very few that were rather simple (e.g. requiring only a capture that could not be rebuffed). The positions were selected in such a way as to attempt to get as many defensive problems as possible into the set. However, this resulted in getting only seven defensive problems. The difficulty was that the solution had to be singular, and while there were many defensive situations, there were not many with a clear-cut defence that was superior to any other. The problems were also chosen in such a way as to ignore whether the NWP was the one on move or not. However, 35 out of the 43 problems turned out to be with the NWP on move. We attribute this to the fact that this program was forcing the play in most instances, and since most of the problems are aggressive problems, this division resulted. Since these problems arose in an environment where moves were not individually timed, but rather groups of moves were timed, we decided to allow CAPS-II as much time as it needed to terminate on each problem. The search was again conducted to depth nine. This resulted in CAPS-II exceeding five minutes of CPU time on two of the problems. However, its average time over the whole set was only 87 seconds per problem, which is well below the allowed average of 180 seconds.

TABLE V-8

Depth	Cases	% Right/CAPS-II	% Right/ NW	% Right/NW Oppnts
1	6	100	100	---
2	10	70	90	---
3	6	67	---	67
4	11	59	73	---
5	1	0	---	0
6	6	42	67	---
7	0	---	---	---
8	1	100	100	---
9	2	50	0	0
TOTAL	43	65	75	50

The results of these tests can be seen in Table V-8. The first thing to notice is that this problem set is considerably easier than the "Win at Chess" set as evidenced by the fact that CAPS-II scored 65% right versus only 47.25% right in the "Win at Chess" set. Other than that, the program does not do nearly as well on the problem set as the NWP did in actual play. This we attribute to the greater completeness and debuggedness of the NWP. There were, however, five instances in which CAPS-II found the correct solution to a problem that the NWP did not find. These problems were in situations involving pins (which apparently the NWP does not handle too well) and where a deep sequence of moves was required to diagnose the correct move at the top. When one compares the performance of CAPS-II against the performance of NWP's opponents, the situations is quite different. Although the sample contains only eight cases, it seems rather clear that CAPS-II is better able to do incisive tactical things than the average NWP opponent. Again this does not signify a great deal since some of the NWP opponents were very weak. Also there is as yet next to no positional or strategical knowledge in CAPS-II. However, it does show, that even in its present state, it can in restricted situations outperform programs with a more complete approach.

An interesting test which helps to reveal some of the program's perceptual ability was performed on a sequence of 14 positions from the book "Rate Your Own Chess" [Bloss, (1972)]. These positions are all mates in two or three moves, with one exception which was a mate in four. The book was written with the idea of finding a device that discriminates playing strength in a simple way. The method used was to correlate the amount of time taken to solve a problem with the playing strength of the solver. The 14 positions tested were the ones which produced the best correlations. CAPS-II was given these positions without any special instructions which might cause it to consider only sub-classes of the moves it would normally consider. One interesting result of this test was that CAPS-II solved 5 out of the 14 positions correctly in 5 seconds or less. This apears 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 strenth, CAPS-II achieved the highest possible rating, a grandmaster rating, by solving the problem in 5 seconds. (It should be noted that the subjects across which the test set was standardized did

not include any players above expert strength, so this is an extrapolated result by the author of the book). 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).

For the five problems it solved in five seconds or less, the program achieved a master rating. For the set of 14 problems, CAPS-II solved 11 correctly and achieved a class A rating. However, these ratings are inflated. The book's author recommends giving a rating of about 1400 (bottom of class C) for all problems that are not solved correctly. However, this clearly causes statistical distortions when the player being rated is below the class C level. For instance, TECH which is a 1250 rated program, achieved a rating near 1600 on the set of problems. However, we are here not trying to establish a rating for the program, but rather to point to some indicators of its perceptual potential.

3. Analysis of Selected Problems

Probably the most outstanding thing CAPS-II has ever done is to solve the problem in Figure 5.2. 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. Only the motivation for the initial move left any room for something short of enthusiasm. 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.

Because of the depth of this combination, we feel quite safe in saying that no program in the world today could duplicate this performance in any standard time frame. There are two side points worth mentioning: First, 1. QxPch, K-B1 leads nowhere. Second, 1. NxP was not chosen because the KBP is blocking a check and is thus overloaded. This function is not yet in the program, and it was thus fortunate that the knight move was suggested because it clears a square from which an attack (by the KBP) can be carried out on a low mobility piece (the Black queen). Third, since the program only searches ahead nine ply, the final position in the principal variation was processed as follows: since the king was in check, all legal moves were generated. The best that Black was found to be able to do was to give back a knight thus leaving White two pawns ahead.

In Figure 5.3 we see another interesting aspect of the program, one which was replicated quite a few times during the series of tests. It is White to play and CAPS-II very rapidly finds: 1. QxPch, KxQ, 2. N-N6ch, K-N1, 3. R-R8ch, K-B2, 4. R-B8ch, QxR, 5. P-Q6mate. Then in retracing the search it finds that by moving the king to B1 in response to 1. QxPch, Black can put up great resistance. In fact it does not find the win of more than the original pawn, but concludes that 1. QxPch is

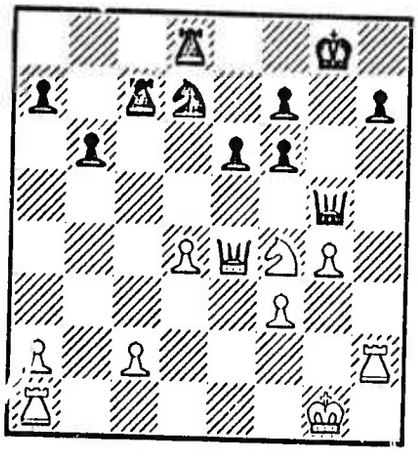

Figure 5.2

White to Play

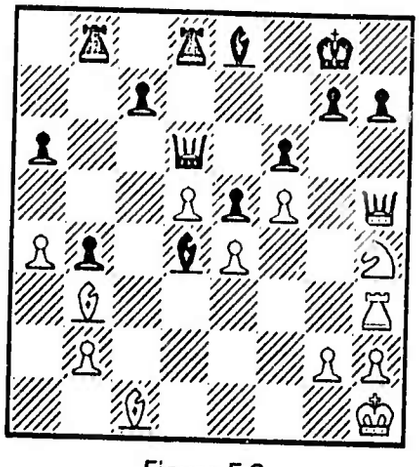

Figure 5.3

White to Play

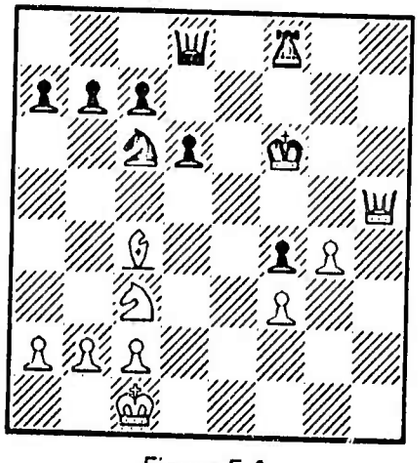

Figure 5.4

White to Play

clearly the best move anyway. It is typical of human analysts, that the whole idea of refusing the queen sacrifice is not even mentioned in the solution to the problem.

An interesting example of CAPS-II's ability can be seen in Figure 5.4. The author noticed this position in "Attack and Defence in Modern Chess Tactics" [Pachman, (1973)], a chess textbook. Here it was presented as showing an interesting maneuver in which a knight was relocated with gain of time. Pachmann's solution runs 1. Q-R6ch, K-K4, 2. Q-K6ch, K-Q5, 3. Q-Q5ch, K-K6, 4. Q-Q3ch, K-B7, 5. Q-B1, K-N6, 6. N-K2ch, K-R5, 7. Q-R1ch K-N4, 8. Q-R5ch, K-B3, 9. Q-R6ch, K-K4, 10. Q-K6mate.

The solution had several sub-variations and seemed somewhat contrived to this author, who thought a quicker mate should be possible. After a few minutes thought, I discovered a mate in five as follows: 1. Q-R6ch, K-K4, 2. Q-K6ch, K-Q5, 3. Q-K4ch, K-B4, 4. Q-Q5ch, any, 5. Q-N5mate. This seemed like a reasonable task to give CAPS-II, so I did. To my surprise the program found a mate in four by 1. Q-R6ch, K-K4, 2.Q-K6ch, K-Q5, 3. Q-Q5ch, K-K6, 4. N-Q1mate. It took CAPS-II 50 seconds of CPU time to find this. To-date it is the only instance of this program outdoing its author. It is interesting to note that a new version of the program now being worked on, which does hopping around in the tree, found the same mate in only 9 seconds of CPU time.

In order to shed some light on the details of the trees that CAPS-II generates, we present the next two examples. The first is seen in Figure 5.5. The associated tree is shown in Figure 5.6. This tree is quite representative of the medium size trees generated by CAPS-II.

In order to interpret these trees, the following should be noted. The number preceeding a move is the depth at which this move is being considered, while the letter pair following the move is the goal state for that node at the time the move is being tried. For instance "0. B-R7ch AG" means the move B-R7ch is being considered at depth 0 while the program is in goal state AG. Goal state encodings are as follows:

AG - AGGRESSIVE
DC - DYNAMIC DEFENCE
KC - KING IN CHECK
KD - KING IN CHECK (DYNAMIC DEFENCE)
ND - NOMINAL DEFENCE
PD - PREVENTIVE DEFENCE
SR - STRATEGY

CAPS-II starts out with 0. B-R7ch and first looks at the response 1. K-R1 which is given precedence because it decentralizes the king the most. Because of the way aggressive moves are presently being generated, the move 2. B-Q3ch is not specifically elicited since it does not cause a double attack or an attack on a low mobility piece. The fact that it is a discovered check is known in another move generator, but the two ideas making a double threat are not connected at the top level in the present version. Therefore CAPS-II generates the various discovered checks that are possible and tries them one at a time in generate and test fashion. This occupies lines 001 through 048 before it finally stumbles onto the correct

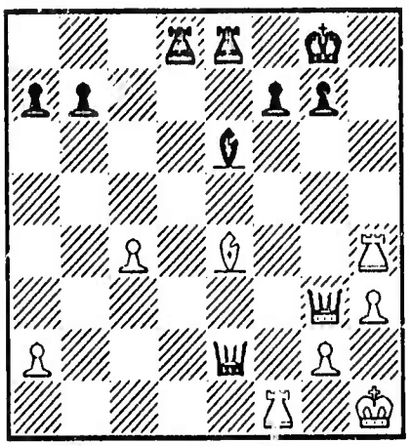

Figure 5.5

White to Play

Figure 5.6

0. B-R7ch	AG	1. K-R1	KC	2. B-N6ch	AG	3. K-N1	KC	4. B-R7ch	AG				001
								4. BxPch	AG	5. BxB	KC		002
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG								003
				8. P-KR4	AG								004
6. QxPch	AG	7. KxQ	KC	8. RxPch	AG								005
6. RxB	AG	7. KxR	AG	8. R-B4ch	AG								006
				8. Q-B3ch	AG								007
				8. Q-B4ch	AG								008
				8. Q-B7ch	AG								009
				8. QxPch	AG								010
				8. Q-N6ch	AG								011
				8. Q-B2ch	AG								012
								4. R-R8ch	AG	5. KxR	KC		013
6. Q-R4ch	AG	7. K-N1	KC	8. B-R7ch	AG								014
				8. BxPch	AG								015
				8. Q-R7ch	AG								016
				8. Q-R8ch	AG								017
								4. RxP	AG	5. BxR	AG		018
6. BxBch	AG	7. KxB	KC	8. Q-B4ch	AG								019
				8. Q-B3ch	AG								020
				8. Q-B7ch	AG								021
				8. QxPch	AG								022
				8. Q-N6ch	AG								023
				8. Q-B2ch	AG								024
6. B-R7ch	AG	7. K-R1	KC	8. B-N6ch	AG								025
				8. QxPch	AG								026
				8. B-B5ch	AG								027
				8. B-N1ch	AG								028
				8. B-K4ch	AG								029
				8. B-Q3ch	AG								030
				8. B-B2ch	AG								031
				8. B-N8ch	AG								032
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG								033
				8. P-KR4	AG								034
				2. B-K4ch	AG	3. K-N1	KC						035
				2. B-B5ch	AG	3. K-N1	KC	4. B-R7ch	AG				036
								4. R-R8ch	AG	5. KxR	KC		037
6. Q-R4ch	AG	7. K-N1	KC	8. Q-R7ch	AG								038
				8. B-R7ch	AG								039
6. P-KR4	AG												040
								4. QxPch	AG	5. KxQ	KC		041
6. R-R7ch	AG	7. K-N1	KC	8. R-R8ch	AG								042
				8. R-N7ch	AG								043
				8. RxP	AG								044
				8. R-R4	AG								045
				8. R-R6	AG								046
				8. R-R5	AG								047
6. R-N4ch	AG	7. K-R1	KC	8. R-R4ch	AG								048

				4. Q-N4 AG	5. QxRch AG	049
		2. B-Q3ch AG	3. Q-R4 KC	4. RxQch AG	5. K-N1 KC	050
			3. K-N1 KC	4. BxQ AG	5. BxP AG	051
	1. K-B1 KC	2. Q-R3ch AG	3. R-K2 KC	4. QxRch AG	5. KxQ KC	052
				4. RxPch AG	5. KxR KC	053
6. R-B4ch AG	7. K-K1 KC	8. Q-R4ch AG				054
		8. B-N6ch DD				055
		8. QxRch DD				056
		8. R-R8ch DD				057
6. QxRch DD	7. KxR KC					058
6. B-N6ch DD	7. KxB KC	8. Q-N3ch AG				059
6. R-K4 DD	7. QxBP PD	8. Q-B3ch AG				060
		8. QxRch DD				061
		8. B-N6ch DD				062
		8. Q-Q3 DD				063
6. Q-R4 DD						064
				4. B-Q3 ND	5. RxB AG	065
6. R-R8mate						066
					5. QxB AG	067
6. R-R8mate						068
					5. Q-K5 DD	069
6. R-R8mate						070
					5. QxPch DD	071
6. KxQ KC	7. RxB AG	8. QxR PD				072
					5. QxRch DD	073
6. BxQ KC	7. R-Q8 AG					074
					5. Q-K4 DD	075
6. R-R8mate						076
			3. R-Q3 KD	4. QxRch AG	5. R-K2 KC	077
6. B-Q3 PD	7. QxPch AG	8. KxQ KC				078
	7. Q-K5 AG	8. BxQ PD				079
	7. QxRch AG	8. BxQ KC				080
						081
RAISE EXPCT 4146 (White expects to win at least a pawn)						082
						083
0. R-R8ch AG	1. KxR KC	2. Q-R4ch AG	3. K-N1 KC	4. Q-R7ch AG	5. K-B1 KC	084
6. RxPch AG	7. BxR KC	8. Q-R8ch AG				085
6. QxPch AG	7. KxR KC					086
6. Q-N8ch AG	7. KxQ KC					087
6. Q-R8ch AG	7. K-K2 KC	8. Q-R4ch AG				088
				4. B-R7ch AG	5. K-R1 KC	089
6. B-N6ch AG	7. K-N1 KC	8. B-R7ch AG				090
		8. BxPch AG				091
		8. Q-R7ch AG				092
		8. Q-R8ch AG				093
6. B-K4ch AG						094
6. B-B5ch AG	7. K-N1 KC	8. Q-R7ch AG				095
		8. B-R7ch AG				096

6. B-N1ch	AG	7. K-N1	KC	8. B-R7ch	AG								097
				8. Q-R7ch	AG								098
6. B-Q3ch	AG	7. Q-R4	KC	8. RxQch	AG								099
		7. K-N1	KC	8. BxQ	AG								100
											5. K-B1	KC	101
				2. P-KR4	AG								102
0. QxPch	AG	1. KxQ	KC	2. RxPch	AG	3. BxR	KC	4. R-R7ch	AG	5. K-N1	KC		103
6. R-R8ch	AG	7. KxR	KC										104
6. R-R4	AG												105
6. R-N7ch	AG	7. KxR	KC										106
6. RxB	AG												107
6. R-R6	AG												108
6. R-R5	AG												109
								4. R-N4ch	AG	5. K-R1	KC		110
6. R-R4ch	AG	7. K-N1	KC	8. B-R7ch	AG								111
				8. R-N4ch	AG								112
				2. R-R7ch	AG	3. K-N1	KC	4. R-R8ch	AG	5. KxR	KC		113
								4. R-N7ch	AG	5. KxR	KC		114
								4. R/7xP	AG				115
								4. R-R4	AG				116
								4. R-R6	AG				117
								4. R-R5	AG				118
				2. R-N4ch	AG	3. BxR	KC						119
0. B-R7ch	AG	1. K-R1	KC	2. B-N6ch	AG	3. K-N1	KC	4. B-R7ch	AG				120
								4. BxPch	AG	5. BxB	KC		121
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG								122
				8. P-KR4	AG								123
6. QxPch	AG	7. KxQ	KC	8. RxPch	AG								124
				8. R-R7ch	AG								125
				8. R-N4ch	AG								126
6. RxB	AG	7. KxR	AG	8. R-B4ch	AG								127
				8. Q-B3ch	AG								128
				8. Q-B4ch	AG								129
				8. Q-B7ch	AG								130
				8. QxPch	AG								131
				8. Q-N6ch	AG								132
				8. Q-B2ch	AG								133
								4. R-R8ch	AG	5. KxR	KC		134
6. Q-R4ch	AG	7. K-N1	KC	8. B-R7ch	AG								135
				8. BxPch	AG								136
				8. Q-R7ch	AG								137
				8. Q-R8ch	AG								138
6. P-KR4	AG												139
								4. RxP	AG	5. BxR	AG		140
6. BxBch	AG	7. KxB	KC	8. R-B4ch	AG								141
				8. Q-B3ch	AG								142
				8. Q-B4ch	AG								143
				8. Q-B7ch	AG								144

				8. QxPch	AG								145
				8. Q-N6ch	AG								146
				8. Q-B2ch	AG								147
6. B-R7ch	AG	7. K-R1	KC	8. B-N6ch	AG								148
				8. B-B5ch	AG								149
				8. B-N1ch	AG								150
				8. B-K4ch	AG								151
				8. B-Q3ch	AG								152
				8. B-B2ch	AG								153
				8. B-N8ch	AG								154
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG								155
				8. P-KR4	AG								156
				2. B-N8ch	AG	3. KxB	KC	4. R-R8ch	AG	5. KxR	KC		157
6. Q-R4ch	AG	7. K-N1	KC	8. Q-R8ch	AG								158
6. P-KR4	AG												159
								4. QxPch	AG	5. KxQ	KC		160
6. RxPch	AG	7. KxR	KC	8. R-R7ch	AG								161
				8. R-B4ch	AG								162
				2. B-K4ch	AG	3. K-N1	KC						163
				2. B-B5ch	AG	3. K-N1	KC	4. B-R7ch	AG				164
								4. R-R8ch	AG	5. KxR	KC		165
6. Q-R4ch	AG	7. K-N1	KC	8. Q-R7ch	AG								166
				8. B-R7ch	AG								167
6. P-KR4	AG												168
								4. QxPch	AG	5. KxQ	KC		169
6. R-R7ch	AG	7. K-N1	KC	8. R-R8ch	AG								170
				8. R-N7ch	AG								171
				8. RxP	AG								172
				8. R-R4	AG								173
				8. R-R6	AG								174
				8. R-R5	AG								175
6. R-N4ch	AG	7. K-R1	KC	8. R-R4ch	AG								176
								4. Q-N4	AG	5. BxB	PD		177
6. QxPch	AG	7. KxQ	KC	8. RxB	AG								178
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG								179
				8. P-KR4	AG								180
6. QxQ	AG	7. RxQ	AG	8. R-R8ch	AG								181
				2. B-N1ch	AG	3. K-N1	KC	4. B-R7ch	AG				182
								4. R-R8ch	AG	5. KxR	KC		183
6. Q-R4ch	AG	7. K-N1	KC	8. B-R7ch	AG								184
				8. Q-R7ch	AG								185
6. P-KR4	AG												186
								4. QxPch	AG	5. KxQ	KC		187
6. RxPch	AG	7. KxR	KC	8. R-R7ch	AG								188
				8. R-B4ch	AG								189
6. R-R7ch	AG	7. K-N1	KC	8. R-R8ch	AG								190

					8. R-N7ch	AG						191
					8. R/7xP	AG						192
					8. R-R4	AG						193
					8. R-R6	AG						194
					8. R-R5	AG						195
6. R-N4ch	AG	7. BxR	KC									196
									4. RxP	AG	5. BxR	AG 197
6. B-R7ch	AG	7. K-R1	KC	8. B-N6ch	AG							198
				8. B-B5ch	AG							199
				8. B-N1ch	AG							200
				8. B-K4ch	AG							201
				8. B-Q3ch	AG							202
				8. B-B2ch	AG							203
				8. B-N8ch	AG							204
6. QxPch	AG	7. KxQ	KC	8. R-R7ch	AG							205
				8. R-N4ch	AG							206
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG							207
				8. P-KR4	AG							208
				2. B-Q3ch	AG	3. Q-R4	KC	4. RxQch	AG	5. K-N1	KC	209
						3. K-N1	KC	4. BxQ	AG	5. BxP	AG	210
		1. K-B1	KC	2. Q-R3ch	AG	3. R-K2	KC	4. QxRch	AG	5. KxQ	KC	211
0. RxP	AG	1. BxR	AG	2. B-R7ch	AG	3. K-R1	KC	4. B-N6ch	AG	5. K-N1	KC	212
6. BxBch	AG	7. KxB	KC	8. R-B4ch	AG							213
				8. Q-B3ch	AG							214
				8. Q-B4ch	AG							215
				8. Q-B7ch	AG							216
				8. QxPch	AG							217
				8. Q-N6ch	AG							218
				8. Q-B2ch	AG							219
6. B-R7ch	AG											
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG							220
				8. P-KR4	AG							221
												222
6. B-R7ch	AG								4. B-B5ch	AG	5. K-N1	KC 223
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG							224
				8. P-KR4	AG							225
												226
6. QxPch	AG	7. KxQ	KC	8. R-R7ch	AG							227
				8. R-N4ch	AG							228
6. B-R7ch	AG								4. B-B5ch	AG	5. K-N1	KC 229
6. R-R8ch	AG	7. KxR	KC	8. Q-R4ch	AG							230
				8. P-KR4	AG							231
												232
6. QxPch	AG	7. KxQ	KC	8. R-R7ch	AG							233
				8. R-N4ch	AG							234
									4. B-K4ch	AG	5. K-N1	KC 235
6. RxQch	AG	7. BxR	KC	8. QxPch	AG				4. B-Q3ch	AG	5. Q-R4	KC 236
				8. Q-R4	AG							237
				8. B-N6	AG							238
												239

			7. K-N1	KC	8. B-N6	PD								240
6. BxQ	AG	7. RxB	AG	8. P-QR4	PD							5. K-N1	KC	241
														242
6. BxQ	AG	7. RxB	AG	8. P-QR4	PD							5. B-R4	KD	243
														244
							3. K-B1	KC	4. Q-R3ch	AG	5. R-K2	KC		245
6. R-R8ch	AG	7. KxR	KC	2. QxPch	AG	3. KxQ	KC	4. R-R7ch	AG	5. K-N1	KC			246
														247
6. R-R4	AG													
6. R-N7ch	AG	7. KxR	KC											248
6. RxB	AG													249
6. R-R6	AG													250
6. R-R5	AG													251
														252
6. R-R4ch	AG	7. K-N1	KC	8. B-R7ch	AG				4. R-N4ch	AG	5. K-R1	KC		253
				8. R-N4ch	AG									254
														255
				2. R-R8ch	AG	3. KxR	KC	4. Q-R4ch	AG	5. K-N1	KC			256
6. B-R7ch	AG	7. K-R1	KC	8. B-N6ch	AG									257
				8. B-B5ch	AG									258
				8. B-N1ch	AG									259
				8. B-K4ch	AG									260
				8. B-Q3ch	AG									261
				8. B-B2ch	AG									262
				8. B-N8ch	AG									263
6. Q-R7ch	AG	7. K-B1	KC	8. Q-R8ch	AG									264
0. R-B6	SG								4. P-KR4	AG				265
0. R-R6	SG													266
0. R-R7	SG													267
0. R-B5	SG													268
0. Q-Q6	SG													269
0. R-R5	SG													270
0. B-N6	SG													271
0. Q-K5	SG													272
														273

31. B-R7ch (Move made)

Principal Variation = 31. B-R7ch, K-B1, 32. Q-R3ch, R-K2, 33. B-Q3, QxRch, 34. BxQ, R-Q8
Number of Nodes = 489
Time taken = 115 secs.

2. B-Q3ch. Thereafter it quickly decides that this wins for White and in line 051 backtracks to depth 1 and tries the other king move at that point. The move 2. Q-R3ch is tried first here, since it is the only safe check and allows the king no escape squares. This move for some strange reason is a difficult one for humans (possibly because it involves moving a piece away from the scene of action to deliver a "long" check; see Chapter VI). Then on 3.--R-K2 CAPS-II tries 4. QxRch and 4. RxPch with no useful results in a portion of the tree lasting until line 063.

Up to now the White side has always been under control of the AGGRESSIVE state. Having exhausted the suggestions of the AGGRESSIVE move generators, and still finding the position satisfactory, the program now goes into the NOMINAL DEFENCE state in which it notices that it has a rook and the QBP en prise. Many moves proposed to solve this defensive problem. Among them is the move 4. B-Q3, which is proposed on both accounts. Through a fortuitous accident Q3 is a square from which the White queen can fork the Black queen and the Black rook at Q1. This results in a TSQ value being assigned this square. Further, when a different piece considers moving to this square, even though it may be unsafe, the piece is regarded as safe (on the assumption that if it were captured the recapture would result in the threat for which the TSQ value was assigned in the first place. Thus the move 4. B-Q3 is regarded as moving to a safe square while centralizing the bishop and accomplishing the defence of the rook. This is most fortunate. The correct reasoning should be that it clears a path to a mating square (R8) for the rook, and simultaneously attacks the queen, but this is outside the present capability of the program. However, once the move is tried, it quickly becomes apparent that Black must lose his queen since it is attacked and R-R8mate is also threatened. By line 080, CAPS-II has established that it must win at least a pawn (a rather conservative view) and announces this together with an upward change in its level of aspiration.

It now begins to try the other recommended, but as yet untried, moves at the top level to see if there is something better. It takes from line 084 to line 102 to dismiss the sacrifice 0. R-R8ch, which humans also consider worth looking into. Then it takes from line 103 to line 120 to dismiss the relatively senseless sacrifice 0. QxPch. After that it goes back to 0. B-R7ch which, due to the reordering rules has now filtered back to the top of the untried moves stack. It then spends from line 120 to line 211 generating a remarkably similar sub-tree to the one that was originally generated for 0. B-R7ch. The variations stemming from 1. K-R1 are more elaborately treated. However, the variations stemming from 1. K-B1, which is now the principal variation, are quickly terminated and it is not necessary to go through the discovery of the key move 4. B-Q3 again, since it is stored in the principal variation. This whole episode brings out an interesting point which could lead to improvements in the control structure. Here 0. B-R7ch is being searched while another recommended move, 0. RxP has as yet not been examined. This should probably not be done while 0. B-R7ch is still the principal variation. Then if all aggressive suggestions are exhausted and the result is higher that the original expectation, all searching can terminate. On the other hand, if another variation were to supercede the present principal variation, then it would be all right to re-examine it. In this case, such a rule would result in a saving of about 35% of the effort on this position. CAPS-II then spends from line 212 to line 265 investigating another senseless sacrifice, 0. RxP. After that, control reverts to STRATEGY, which proposes eight moves, none of which are appealing enough to be searched by

CAPS-II. When STRATEGY suggests the move which is now the principal variation, the whole search comes to a halt. Thus CAPS-II has a bit of good luck here. If it were not for the en prise position of the White rook at KB1, the problem would not be solved. As pointed out above, the move 4. B-Q3 should be recognized as a double attack, after which the solution would come off smoothly.

An example of a small tree can be seen in Figure 5.7. This tree derives from the position in Figure 5.8, which is at the crisis point of a famous opening trap. Here it is very much to CAPS-II's credit that it finds and dismisses the obvious line in a mere nine nodes (line 002 and 003). In returning from this variation, it decides, based on causality and alternative considerations, that the results of this line cannot be upset at any point after depth 0. This means that the program recognized that the consequences of 0. PxB (e.g. 1. QxQch, 2. KxQ, 3. BxPch, and 5. BxR) could not have been sensibly avoided below the depth 0 level in the tree. This is in itself a tremendous feat, rivalling the economy with which a human could make such a decision. The CAUSALITY FACILITY gets exclusive credit for this work, which avoids what could have been a search of considerable magnitude, if all the "sensible" alternatves at depths 4, 6, and 8 were to be explored, in the style of most of today's programs. The program then fixes on the correct idea, 0. QxN, and again discovers the principal variation on the first try. Thus by line 005, it has found the essence of the whole solution as usually presented in textbooks. However, in line 007 it begins a whole new and interesting tack in the analysis. The move 1. P-B4 is suggested by the AGGRESSIVE state as a multi-purpose pawn move. It turns out also to create a flight square for the king. In protecting the bishop at N5, and thus renewing the attack on the Black queen, the mate at Q8 is again indirectly threatened (this is not statically known to the program). Line 007 through 014 are now spent finding and verifying a defence to the Q-Q8mate threat. Once 2. Q-Q3 is found to hold onto Black's gains, CAPS-II tries three other suggestions from DYNAMIC DEFENCE in lines 015 to 017, before coming to the conclusion that the level of aspiration should be raised. After that it quickly dismisses Black's only remaining alternative, and recognizing that there is no move at the top which can improve on the present principal variation, it outputs its move. The value of the position, as frequently happens when there is one clearly superior move, is underestimated. However, that does not hurt anything, as the important thing is that it not exceed what can be achieved. On a later move, the level of aspiration would again be raised.

4. Some Current Deficiencies

The next few examples deal with things that the program does not handle properly at present. In Figure 5.9 White to play wins quickly with 1. R-K8, QxR, 2. Q-B6 and the threat of Q-R8mate can not be met. This turns out to be very simple for even moderately advanced human players, since they all recognize the basic pattern of the White queen and bishop on the long diagonal in conjunction with the particular weakened Black king position. However, our program doesn't have an inkling of the solution. There are several reasons for this, the primary one being that it does not yet generate moves that threaten mate. Let us assume that a facility for generating moves that threaten mate did exist in CAPS-II. Then the program would be able to find 2. Q-B6 and follow through the subsequent mate. However, it would still not be able to play the preliminary 1. R-K8. We dwell on this example as prototypical of a whole series of problems. The approach to solving

Figure 5.7

```
0. PxN     KC  1. QxQch  AG  2. KxQ     KC  3. BxPch  AG  4. B-K2    AG  5. BxR     AG   002
6. B-N5ch  AG  7. K-Q1   KC  8. PxP     AG                                               003

0. QxN     KC  1. BxQ    AG  2. B-N5ch  AG  3. Q-Q2    KC  4. BxQch   AG  5. KxB     KC   004
6. PxB     AG  7. PxP    AG                                                               005

               1. Q-Q8ch  AG  2. KxQ     KC  3. R-Q1ch  AG  4. N-Q2    KC               006

               1. P-B4    AG  2. QxP     PD  3. Q-Q8mate                                 007

                              2. Q-B4    PD  3. Q-Q8mate                                 008

                              2. Q-Q3    PD  3. QxQ     AG  4. BxQ     AG  5. PxKP    AG   009
6. BxP     PD  7. P-QR4   AG                                                            010

                                             3. PxKP    AG  4. QxP     PD  5. Q-Q8mate     011

                                                            4. Q-Q2    PD  5. PxP     AG   012
                                                                           5. P-QR4   DD   013
                                                                           5. P-QR3   DD   014

               1. P-QR3   DD  2. QxB     PD                                              015

               1. P-QR4   DD  2. QxB     PD  3. RPxP    AG  4. RxR     PD               016

               1. P-B5    DC  2. QxB     PD                                              017
                                                                                         018
RAISE EXPCT  4086   (Black stands better)                                                019
                                                                                         020
0. K-K2    KC  1. N-Q5ch  PD                                                             021

9.-----  QxN      (Move made)

Principal Variation = 9.---- QxN, 10. P-B4, Q-Q3, 11. PxKP, Q-Q2, 12. PxP
Number of nodes = 49
Time taken = 12 SECS.
```

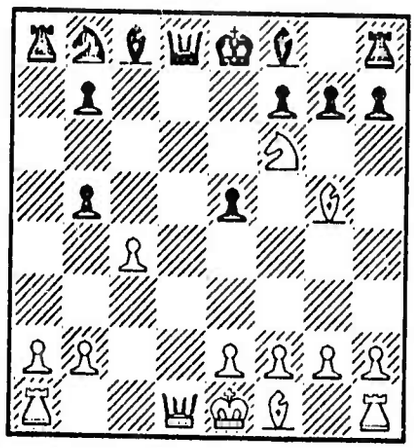

Figure 5.8

Black to Play

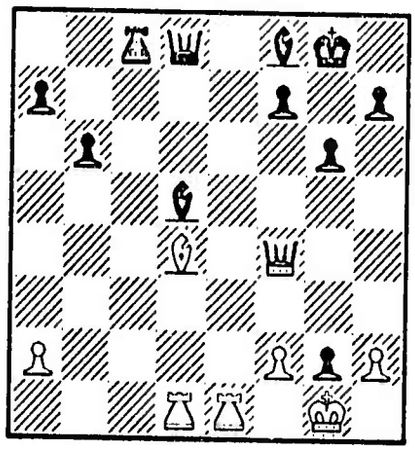

Figure 5.9

White to Play

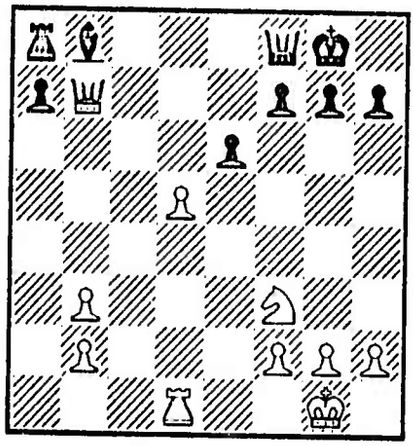

Figure 5.10

White to Play

such problems, involving preparation, is extremely important. If it is not done economically, the program can be bogged down forever in searching patterns or doing means-end analysis for many unimplementable ends. Here two basic solutions appear possible. The first involves trying the move 1.Q-B6 and finding that it leads to nothing after 1.--QxQ. Then it is necessary to establish that this threat would be worthwhile if the Black queen were not in a position to play QxQ. This would generate a lemma binding the Black queen to a defensive function on the square KB6. This in turn would give rise to the goal of attacking a piece with a defensive function, and so 1. R-K8 would ultimately be generated. The second approach involves noticing the basic pattern of queen and bishop on the long diagonal which is attainable in the original position. Then noticing that the Black queen prevents its attainment, gives rise to the goal of distracting the Black queen. Both approaches require some very good indicator of which threats or patterns are worth trying to get to work when they do not work immediately. For instance, the immediate 1.Q-K5 also threatens mate or assumes a desirable pattern. This is then met with P-B3 after which 2. QxP wins a pawn. Now white could be satisfied with this or try to get rid of the blocking KBP or look for some other method of getting the queen on the diagonal. It is not an easy problem.

The problem in Figure 5.10 is very illuminating for understanding a certain kind of myopia that today's programs have. It will presumably be cured when a lemma facility exists, but the present program is still susceptable to this problem. In this position 1. QxR does not work because of 1.--BxPch winning the White queen, as the program quickly finds out. The actual solution is 1. P-Q6 (blocking the bishop's diagonal), BxP, 2. RxB winning. However, the program next tries 1. PxP and now gives the Black response 1.--PxP a low rating because it leaves the rook en prise. This is very strange to human eyes, since we have already established that the rook is not really en prise as long as BxPch is possible. Clearly, what is needed here again is a lemma facility which makes some statements about the safety of the Black rook and under what conditions these are true. As a result of the above internal misunderstandings, CAPS-II finally decides it can win a pawn by 1. PxP, and completely misses the lemma upsetting idea of 1. P-Q6.

A simpler problem, which also came up in Game II note H, below, is seen in Figure 5.11. Here Black to play can win material by the simple 1.--R-B1 attacking the White queen and x-raying the White rook. The reason CAPS-II fails to find this move is that the White rook is considered as being overprotected by the queen and thus 1.--R-B1 is only a single attack on a piece which is not low in mobility. The solution to this problem consists of placing all pieces defended or overprotected by a given piece, on the interest vector of any attacking piece when it is attacking the piece that has these defensive functions. In the above position, this would result in the rook at QB1 being considered undefended when a move that attacks its defender is being examined. Then 1.--R-B1 should be found rather easily.

Another interesting failure, due to a function that is as yet not implemented. can be seen in Figure 5.12. Here it is Black to play and the winning idea is 1.--N-R6ch with the following two variations:

2. PxN, QxPch, 3. K-R1, QxPmate and
2. RxN, QxQ!, 3. RxQ, R-Q8ch and mate next move.

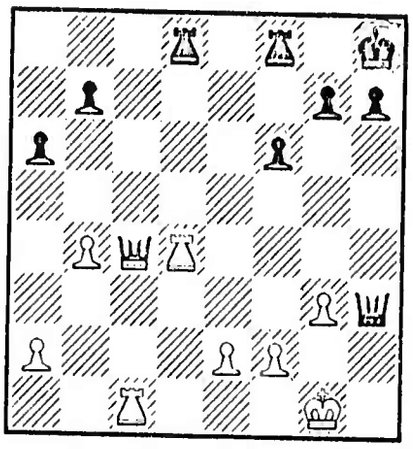

Figure 5.11

Black to Play

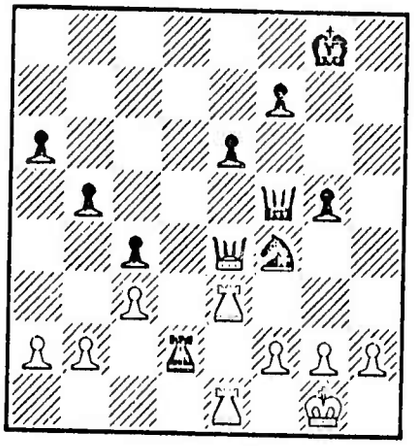

Figure 5.12

Black to Play

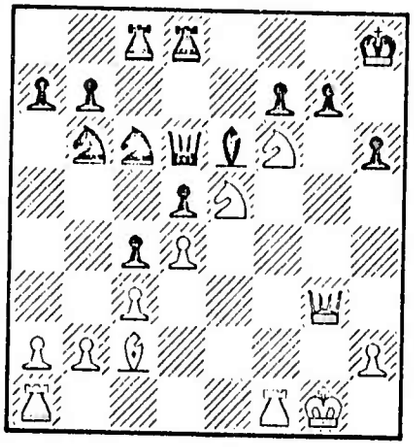

Figure 5.13

White to Play

The program finds the first variation very well, but in the second it misses the critical move 2.--QxQ because it sees the queen is defended and it does not consider decoying the White rook to K4 after having sacrificed a knight to be a sensible idea. The important point (that is not implemented in CAPS-II) is that the White rook also blocks a check at Q8 by the Black rook. This is the function of the White rook that becomes critically overloaded, but which is not noticed. It is worth noting that CAPS-II is smart enough not to just try the capture 2. QxQ on the grounds that it doesn't lose anything.

In Figure 5.13, we see another interesting mating pattern, which of course is not known to CAPS-II. With White to play there is a mate in two with 1. Q-N6!, (threatening Q-R7mate), PxQ, 2. NxPmate. The failure here is again due to the inability of CAPS-II to notice mating threats. The move 1. Q-N6 even when tried is only considered to be useful for attacking the KBP. Actually CAPS-II realizes that the queen may be safe on N6 since this is a square where a White knight can give check, and this fact is known to the data structure. However, it does not have access to the idea of a mate threat. Instead of the book solution, the program finds an interesting but inferior solution, which we present because it shows a certain degree of ingenuity. CAPS-II likes 1. N-K8 (threatening queen and pawn, whereupon RxN loses the queen by the discovered attack NxPch). If instead 1.--Q-B1, 2. N-N6ch wins even more material, so the program succeeded in finding a clear win, but not the right one.

C. FULL GAMES PLAYED

We next present two games that were played between CAPS-II and TECH playing at depth 3. These games are probably more illustrative of the strengths and weaknesses of CAPS-II in its present state, than any other evidence presented. It was originally hoped to match our program with TECH playing at tournament setting where it takes about three minutes per move. However, it became clear after these games that there is still quite a bit of work to be done before this would be a match in which our program had a chance to win. As it was, there was a great time mismatch in the contest; TECH taking about 20 seconds of CPU time per move as against about 6 minutes per move for CAPS-II. However, this and other existing gaps seem fairly sure to be narrowed in the near future. The comments on the games are intended to be insightful for the structure of the programs rather than to arbitrate over what move is the best in each situation.

GAME I

WHITE- CAPS-II BLACK- TECH (Depth 3)

1.	P-K4	P-K4	16.	P-QR3 (C)	P-KB4
2.	N-KB3	N-QB3	17.	Q-Q3	RxP
3.	P-Q4	PxP	18.	Q-B4ch	K-R1
4.	NxP	N-B3	19.	P-QB3 (D)	P-B5 (E)
5.	NxN (A)	NPxN	20.	PxP	P-B3
6.	N-B3	P-Q4	21.	BxP?? (F)	RxB
7.	PxP	PxP	22.	Q-B1	QxP (G)
8.	B-N5ch	B-Q2	23.	QxP	R/7xBP!! (H)
9.	BxBch	QxB	24.	Q-B8ch	R-B1
10.	Q-K2ch	B-K2	25.	QxRch	RxQch
11.	O-O	O-O	26.	K-R1	RxRch
12.	B-B4	P-Q5	27.	RxR	BxP
13.	N-K4	NxN	28.	R-B5 ?	Q-Q8ch
14.	QxN	QR-N1	29.	R-B1	QxRmate
15.	B-K5	R-N5 (B)			

(A) The combination of this move and the next is weaker than the immediate 5.N-QB3. However, CAPS-II plays the first move searched that is equal tactically to the best result it can achieve.

(B) This illustrates a frailty of TECH; it makes the positionally preferred move of all those that are equal to the tactically best result. Here positional means most centralizing and mobilizing. Since RxP does not win anything, R-N5 is preferred.

(C) Apparently with a view toward 16.--RxP, 17. BxP and both the rook and the QRP are attacked, or if 16.--R-QR5, then 17. P-QB3 since now the rook is undefended. However, in the first line the QRP can never be taken with gain.

(D) Again the first acceptable tactical move is chosen. Strangely enough Black cannot play PxP as QxP/3 recovers the pawn with interest, however the move is positionally inane.

(E) 19.-- P-Q6 is much superior.

(F) A very instructive blunder. White thinks he will win a pawn in this way since after 21.--RxB, 22. Q-B1 attacks both rooks.

(G) At this point in the analysis performed by CAPS-II on the previous move, several remedies were suggested for Black. R/5xBP and R/7xBP were searched and found to be tactically inadequate. Several moves were suggested to remove or defend each of the rooks, but none of the these except the two above mentioned received a high enough static score to pass the forward pruning device. The instructive thing is that QxP was suggested by the defensive move generator both to defend the rook at QN7 and the rook at KB5. In CAPS-II when a move is multiply suggested it retains the highest static rating. However, the important point of a move having two useful functions, each of which it should receive credit for, was lost. This has since been partially remedied.

(H) A tremendously strong move after which White must lose additional large amounts of material, e.g. 24. Q-K8ch, R-B1 and there is no good defense to the threat of RxRmate.

GAME II

WHITE - TECH (Depth 3) BLACK - CAPS-II

1.	P-K4	P-K4	17.	BxP	Q-R4ch
2.	N-KB3	N-QB3	18.	P-QB3?? (D)	R-K1
3.	P-Q4	PxP	19.	B-K4	B-B3 (E)
4.	NxP	Q-R5 (A)	20.	P-QN4	Q-K4 (F)
5.	N-QB3	B-B4	21.	P-B3	P-B4
6.	B-K3	NxN	22.	0-0	PxB
7.	BxN	BxB	23.	PxPch	N-B3
8.	QxB	Q-KB3	24.	QxP	QxBP?? (G)
9.	P-K5	Q-K2	25.	QR-B1 (H)	QxP
10.	N-Q5	Q-Q1	26.	RxB	K-N1 (I)
11.	P-K6	P-KB3 (B)	27.	R-B7 !	K-B1?? (J)
12.	B-QN5??	P-QB3	28.	RxP?	Q-K2?? (K)
13.	PxPch	BxP	29.	RxQ	RxR
14.	Q-K3ch	K-B1	30.	RxNch	K-N1
15.	B-B4	PxN	31.	QxR	P-R4
16.	BxP	B-R5 (C)	32.	R-B8mate	

(A) In moves four through seven, one can again see the lack of positional influence in Black's play.

(B) This fine defensive move has been previously discussed in Chapter III. Now it is TECH's turn to show its weaknesses. 12. B-N5 is also played when TECH searches to depth five, since it is the positionally preferred move which loses no material in the analysis: 12.--P-B3, 14. PxPch, BxP, 15. Q-K4ch and the quiescence analysis shows that Black can start no sequence of CAPTURES at this point that yields any gain in material. The fact that pieces remain en prise does not register with TECH, but CAPS-II is not fooled by instance of the Horizon Effect.

(C) Very imaginative, but B-KN5 preventing 0-0-0 would be better. Now White would be well advised to castle Q-side instead of taking the pawn. Then there would be threats of Q-R3ch, BxN, and BxP and only 17. 0-0-0, N-K2!, would hold on, 18. BxP being met by Q-B2! threatening the bishop and mate at QB7 and thus only losing a pawn. Both programs are unaware of this.

(D) White has many problems and cannot see to the end of it all. If 18. Q-Q2, R-K1ch, 19. K-Q1, R-Q1 wins. If 18. Q-B3, QxQch, 19. PxQ, R-N1, followed by BxP regains the pawn since 20. B-K4, R-K1, 21. P-B3, P-B4 loses the bishop. If 18. K-B1, Q-N4ch, 19. K-N1, QxB, 20. Q-R3ch, N-K2, 21. QxB, QxNP recovers the pawn. If 18. K-Q1, BxPch, 19. KxB, Q-B2ch, 20. Any, QxB again reestablishes the status quo. It is not clear if CAPS-II worked all this out, since it was running in a mode with no tree print out. However, considering its general performance level and the fact that it took quite some time on its 16th move, I consider it highly likely that it worked out something similar to this since all the variations are rather forcing even though they are deep. One thing is clear, this is the first time in annotating computer chess games, that I felt such a deep

analysis was an attempt to explain what was actually being calculated. Now, TECH, being unable to see far enough, thinks it can hold onto the pawn, but instead gets into a pin which loses another piece.

(E) This seems rather strange since 19.--P-B4 wins the bishop more cleanly. However, there is intrinsically nothing wrong with the move since 20. BxB, RxQch, 21. PxR, Q-N3 is worse for White than losing the bishop.

(F) Here CAPS-II begins to go positionally astray. This move is made as being the most centraliazing of several that are equal materially. It is worth noting that it relies on the variation 21. BxB, QxPch winning everything, which might however be somewhat obscure for human eyes. However, the move is a positional error, since Q-R3 preventing K-side castling and covering the Black QRP, and also Q-R6 are considerably stronger.

(G) With his last move White got a little more out of the position than Black need have allowed. However, here the obvious QxKP threatening mate was more than sufficent to win. Instead CAPS-II thinks it can win two pawns.

(H) In performing its analysis the program did not consider this move, since chasing the queen is considered pointless. CAPS-II does not consider this a double attack, since the bishop is "over-protected" by the queen. This problem is known to us since it also occurred in several of the "Win at Chess" positions (see the discussion of Figure 5.11 above). Here it provides a glaring example of how important superfine specification of all detail is if one wants to have a program that economically and yet effectively operates in its environment. What we mean is that clearly generating all attacks on a queen is a waste of time unless it is a low mobility piece. However, when a queen is attackable a special routine is required which now places all pieces that the queen is currently charged with defending or over-protecting on the potential target list. Then the effect of this move would be noticed by the aggressive move generators.

(I) Now Black notices that White is threatening mate in three moves by 27. R/6xNch, PxR, 28. RxPch, K-R1, 29. Q-B7mate. TECH would have had to be running at depth 6 before it could have decided on the move 27. RxNch, so it clearly had no such intentions. However, the deeper searching program is looking for truth, since that will ultimately pay off best, and is not concerned with its opponent's frailties. Therefore the text move, which meets the threat. However, in this position which is difficult by master standards, neither program really understands what is going on. Objectively speaking, after the text, White has a winning position. The only defence, leading to a draw, was 26.--R-K2!, 27. Q-R8ch, R-K1, 28. Q-R7. White cannot play 27. R-B8ch because K-B2! wins. Nor can Black play 26.--Q-K2 because of 27. R-B7, Q-K3, 28. P-K5, QxKP, and White mates in three beginning with 29. R-B7ch.

(J) This blunder is due to a level of aspiration problem which is very difficult to trace since CAPS-II took 45 minutes of CPU time on this move. The correct move is 27.--Q-B1, after which White can still win by 28. Q-Q4 threatening both Q-B4ch and P-K5. Then 28.--RxP does not work because of 29. QxR, NxQ, 30. RxQch, KxR, 31. R-B8ch. On a rerun of this position, when the program was given the expectation of maintaining the status quo materially, it did play 27.--Q-B1. Therefore we conjecture that CAPS-II was expecting to win the KP, and now when it finds out this is no longer possible, it failed to adjust its level of aspiration downward properly and thus makes a

blunder allowing mate in three by 28. R-B7ch, K-N1, 29. RxPch, K-B1, 30. Q-B7mate. However, TECH fails to see this since it is a depth 6 effect, and in turn makes a blunder.

(K) A horrible move just when the game could be saved by 28.--R-K2!, after which White could no longer win. It is very instructive, in terms of understanding the intricacy of program structure, to see how this happened. CAPS-II looked at 28.--R-K2 but in the variation 29. RxR, QxR, 30. QxQch, the move 30.--KxQ is evaluated only in terms of its offensive potential due to the way CAPS-II is organized. (This has since been remedied). This meant that 30.--KxQ was known to recover the queen, but there was no knowledge that it would also protect the en prise knight. The static score for this move was therefore so low (since it was expected that a knight would be lost in any case) that the move was not searched on the first pass, and CAPS-II came to the conclusion that about three pawns worth must be lost for Black. The level of aspiration was then reset accordingly and the Alpha and Beta values at the top were set to what was thought to be an optimistic loss of three pawns for Black and plus infinity for White. Now with the lower level of aspiration, the move 30.--KxQ was searched and the whole variation was found to produce a very good result for Black. However, when this was backed up to the top, it was cut off by Alpha-Beta as being outside the limits of what Black could achieve. So the utility of 28.--R-K2 could not become known due to the initial error by the static evaluation function and the consequences which followed it. One might suggest that this would not have happened if Alpha and Beta had been reset to plus and minus infinity on the second search after the level of aspiration had been reset. However, this approach has been tried previously in this program, and has at times resulted in the program getting into an infinite loop as follows: When the second search in done, the real value of the move 28.--R-K2 would be found. However, this is so much better than the revised expectation that another search is performed with a higher level of expectation. However, now the static value of 30.--KxQ is not high enough to be searched, so a low value is returned to the top of the tree, and the expectation is lowered again, etc. To prevent this, the current paradigm is necessary. However, this move shows how closely each agency must work with every other if the present economy is to be achieved, and how frequently a mistake by one agency cannot be covered up by a combination of actions of the others. This point is pursued further in chapter VI. After the move in text, the game becomes hopeless.

CHAPTER VI

REFLECTIONS, CONCLUSIONS AND RECOMMENDATIONS

A. DOES BEING A GOOD CHESS PLAYER HELP DOING CHESS RESEARCH

The author is a former Correspondence Chess World Champion, and has been among the top ranked over-the-board players in the U. S. for over 20 years. As such, I am frequently asked how much my being a top chess player has helped in doing this research. The following statements summarize my attitude toward this matter:

1. There is nothing in this thesis that is inherently dependent on having more than a rudimentary knowledge of chess.

2. Being a good player makes it easier to discriminate inadequacies in the behavior of a program, not only in the actual selection of a move but also in the methods that support this. Some of these inadequacies may appear quite normal to a weaker player. Being able to discriminate problems is the first step toward finding a solution. Thus being a good player has helped to identify certain problems that other models of chess have, and that early versions of this work had. It has also helped in discriminating useful facets that other disciplines (psychology in this case) have had to offer.

3. This is the totality of effects that I have noted. It seems clear, however, that at some later stage in the development of chess programs, higher level knowledge must be available to such a program. This knowledge must either be input from a good human player or, if a machine is capable of assimilating such knowledge from its environment, then probably a good human player should supervise this activity.

B. ON MODELS OF PHENOMENA

In science there is a spectrum along which models of phenomena can be arranged. At one end of the spectrum one finds such universal models as relativistic mechanics which are considered to explain completely the phenonmena with which they concern themselves. At the other end of the spectrum one finds the statistical type of laws such as those in agriculture relating to crop yields. It is no accident that sciences that deal in universal laws are considered to be the firmer sciences, since these laws allow much more quantitative prediction.

It appears that not much has been done in computer chess toward moving in a direction of universals. It is considered adequate to merely specify moves which have a good statistical chance of being appropriate, and then let the tree search weed out the good from the bad. However, as we have noted earlier in this document, the tree search is overburdened. It is therefore incumbent on the designer of a chess program to do all he can to relieve the tree search of unnecessary work by using his data well. This involves not taking the statistical posture if it can be avoided.

Consider the following example. A knight move exists which forks two major pieces. However, the square on which the fork could take place is guarded by a pawn. The statistical method considers the value of the major pieces that would be attacked as positive features, and the fact that the knight could be captured as a negative factor.

Summing all these would probably yield an evaluation which indicates the move is worth trying. However, there is one overwhelming causative feature here which such an approach ignores. That is, that if the knight were captured all its threats would also disappear. Therefore the primary question that must be asked is whether the pawn that will be doing the capturing is rendering any other important services for the opponent. Only if this is true, is if even worth considering the impact of the move, since the capture of the knight could then produce other consequences. Clearly, using such AND/OR sequential logic produces a much superior utilization of the information at hand. This is the approach that we have tried to follow in this work. We consider the AND/OR description of events on the chess board to be an approach with no limits on its accuracy, given only that we are willing to continue the process of adding tests as long as this proves useful. Our experience with this approach indicates that it is many times more powerful in perceiving worth, than a statistical approach which would consider two to three times as much information.

C. ON BRINGING UP THE PROGRAM

The author has developed one previous chess program [Berliner, (1970)] which belonged in the class of programs spawned by the initial efforts of Greenblatt. Since a program of that type has a fairly well defined structure, the process of bringing up and improving such a program involves implementing well defined control structures and then adding ever more analysis and evaluation procedures to the program to improve its acumen. The work on tree searching may include some consideration of what types of move go into the quiescence analysis, and how to adjust the width of search for different depths of the tree under various time constraints. Other than that, all the effort is in getting the right moves into the search and then improving the terminal evaluation function in order to detect worthwhile advantages. In the current program, however, each of these tasks had new dimensions.

During the development of the program, the author was confronted with three generic problems which kept reappearing. This was true even though each time the problem was dealt with in such a manner as to make it go away at that point in the development process. These problems were:

The Move Proposal Problem - Making sure that the correct move in a given position is in fact generated by some move generator.

The Move Appreciation Problem - Making sure that once a move had been generated, its potential is properly evaluated so that it will be searched at a reasonable point in the total analysis.

The Move Selection Problem - Making sure that the tree search does not grow too large, while at the same time making sure that no incorrect decisions within the tree cause sub-trees with meaningful nodes to be cut off.

The move proposal problem involves having special move generators for each purpose. Thus if a move generator is not complete or a purpose not defined, important legal moves may not be generated. This is quite different than just assigning high values to "good" moves in a pass-in-review type move generator. Firstly, the set of purposes is vast, and some are inevitably left out. Secondly, a move generator which is complete for a particular purpose may still be very blind to what else any move

suggested by it may accomplish or destroy. This raises problems for the evaluation function which must accept the word of a move generator that this is a worthwhile move, while at the same time trying to evaluate the move in a more global context.

The move appreciation method in this program attempts to be more of an AND/OR discrimination tree, rather than a sum of factors type of selection procedure. It became clear early in the development of the program, that the factors mentioned in Chapter III do not sum very well. Therefore, the evaluation was developed to respond to factors that appeared important in the environment. Frequently, when several important factors pointed in opposite directions, it was sufficient to note that one pointed in a favorable direction. Thus this was an OR condition; something that would be difficult to construct in a sum of factors scheme. However, such decisions also allowed worthless moves to come under scrutiny. Thus, there was usually an additional nested AND level which again posed restrictions on the move qualification process. This procedure of adding more conditions and attempting to do away with statistical appraisals of factors continued throughout the development of the program.

Finally, move selection, which in effect is determined by the structure of the search tree, was a continuing problem. The different goal states evolved from needs that the program evinced, since the absence of such a goal state would frequently send the program off on a wild goose chase. Also the transition rules between goal states provided many opportunities for making design errors. It was our experience that apparently air-tight definitions of transition conditions frequently had a loop-hole which was sooner or later found by the program and used to miscarry the analysis. Needless to say, any tree search which could perpetuate such logic errors could not be held responsible for the validity of its output. As mentioned in Chapter V, almost the whole effort in going from CAPS-I to CAPS-II was in straightening out this aspect. There were also several tree pruning rules which failed to work out and later had to be removed; however, the goal state management problem was paramount in this area until it became solid.

D. WHERE DOES THE POWER COME FROM

The present program does not yet play as well as even our previous program. This is due mainly to its lack of consistency. However, this does not disturb us. The mandate for a program that can play really well is that it be able to search deeply when required and discriminate effectively at all times. The present program is well on its way to doing this. It already has deep searching capability, and we expect this to increase in the near future. It does not do much positional and strategical evaluation. However, since it looks at a very limited number of nodes, the multiplication factor for additional computing done at each node is low. Therefore, additional discrimination facilities can probably be put in at low computing cost when tactics has reached the appropriate level of competence. We consider the present program extendable to first getting rid of the remaining tactics problems and then addressing the positional and strategic aspects of the game. We expect to be able to do this without significantly altering the time it takes for the program to make a move.

Thus, our program's power is manifest not in its ability to beat its opponents, but rather in its ability to circumnavigate a large search space. This derives partly from the representation, and partly from the management of the tree search. We consider first the influence of the representation in providing structure which slows exponential growth.

1. The Representation

We can consider power to be related to the depth of projection of a particular device. Thus a device which projects a future state of the board two ply from now is more powerful than one that projects a state one ply from now. It is also obvious that power is not linearly related to the depth of projection since we are dealing with an exponential growth process. Table VI-1 shows the depth projecting capabilities of the various bearing relationships.

TABLE VI-1 - Depth Projection of Bearing Relationships

Relation	Depth
DIR	1
OBJ	2
ETHRU	2 (minimum)
DSC	3
OTHRU	3 (minimum)

To see how we arrive at these figures consider the relation DSC(PC, SQ). This relation states that if a certain intervening piece were to move away, PC would then be able to move to SQ. Clearly, that requires a three ply sequence: The intervening piece moves, the opponent moves, and then PC moves. Thus if SQ were known to contain some interesting target, knowing the relation DSC would allow projection of part of a board state that could not occur for at least three ply. For some of the relations such as ETHRU and OTHRU which are transitive, the depth projection could be even higher depending upon the number of intervening pieces that exist between PC and SQ. Clearly the role of knowing that there is a target of interest on SQ should not be minimized. Otherwise, just knowing that a certain piece could reach a certain square, would not give much useful knowledge in the attempt to reduce the rate of exponential growth.

While bearing relations give information about what is possible, functions * give information about what is essential, given a particular world view of what is going on on the board. Thus while moving an intervening piece away to create a discovered attack could lead to some gain, it is not clear that it will unless there is a worthwhile target which can in fact be captured with impunity. However, moving away a piece that has a defensive function to fulfill will much more frequently cause a loss. Table VI-2 shows the projection potential of the various functions. The minimums can be worked out from the definition of the function; the maximums are those that could reasonably be encountered in a game situation.

(*) The first known use of functional analysis in chess is in a Carnegie Institute of Technology term paper entitled "Chess Program" by Duane R. Packard and Thomas P. Cunningham. In this paper which was inspired by Dr. Allen Newell the authors apply a functional analytic method to several chess positions. The method was used to simulate the playing of one game.

TABLE VI-2 - Depth Projection of Functions

Function	Minimum Depth	Practical Maximum
Attacking (a material object)	1	5
Attacking (a threat square)	3	9
Defending (a material object)	2	6
Defending (a threat square)	4	10
Blocking (opponents activity)	2	8
Blocking (own activity)	3	9
Guarding Escape Square	2	6

The above maximums occur when there are several functions associated with a single objective on a square. When the program is investigating function conflict (overloading), we are projecting a minimum of three-ply; e.g. capture on square A, recapture on square A and capture on square B. If the squares on which the overloading takes place have many functions associated with them the projection could easily be 8 or more ply. It is true that most of today's programs have data structures than will allow them to predict what is going to happen on an OCCUPIED square most of the time. However, the present structure also deals with:

1) Squares that are blocking important activity that is to take place on another square.

2) Squares to which certain pieces could potentially move.

3) Conflicts that arise when a piece needs to perform functions on more than one square.

Thus the above data structure serves to point to tactical goals which could be reasonably be expected to be achieved many moves away. This is due to the fact that functions are assigned alternately for each side so as either to create a balance or an imbalance. This fact is then noted. It is then possible for the program to judge what would happen if one unit of strength were added or taken away. Thus if one views tactics as the domain of what can be accomplished by forcing moves, then the data structure is well equipped to tell what kinds of things can be influenced and how to influence them. Since this influence could take many ply of searching to unravel, the data structure can point to features which are out of the perceptual domain of any program we know of. Thus it is possible to generate moves which could:

1) Begin a chain of captures of indefinite length leading to a gain in material (capture of an en prise piece).

2) Require an immediate response to prevent such a sequence from becoming possible (single threat moves).

3) Make possible a future gain by invoking more than one of the above type of threats with a single move (multiple threat moves).

4) Capture an apparently well defended piece because one of its defenders has another important duty to perform.

5) Occupy an appparently well defended square because one of its defenders has another important duty to perform.

6) Have a good likelihood of causing a gain by attacking a low mobility piece that will have difficulty in removing itself from danger.

All the above presume tht everything on the board except what concerns the square(s) in question is inconsequential. This is a simplifying assumption that frequently proves to be wrong. However, it does provide a basis for doing business, and the tree search still arbitrates over the ultimate worth of any move. Thus as long as the tree search is not asked to do too much work -- leading to unacceptable levels of exponential growth -- this method is satisfactory. It can always be improved by the simple expedient of finding a few more of the moves to be worthless. This is done by adding some more AND/OR conditions to the static analysis. Each new condition is added as a test at a tip node of the present discrimination tree. If there are N tip nodes, adding a new test to the tree adds 1/N to the length of the average path in the tree. Thus this procedure adds computing cost in inverse proportion to the number of branches already existing in the discrimination tree. This cost goes down as a function of the size of the discrimination tree; a most commendable property, especially when compared to the exponential cost of tree searching.

2. The Tree Search

It is more difficult to explain why the tree search is powerful. There does not appear to be much question that it is more powerful than the standard approach. The branching factors are 1/3 to 1/2 that of standard programs. This can not be explained away by saying that this program does not play as well as the best standard programs. The fact is that these branching factors have been more or less maintained throughout the development of the program. The largest increase in the size of the branching factor came during the time when several errors in the transition rules between goal states were found. On the other hand, the addition of the PREVENTIVE DEFENCE goal state resulted in a noticeable decrease in tree size. As the program gets better, the branching factors do not change significantly. One could associate getting better with searching more moves and/or being better able to evaluate terminal positions. The latter does not change the branching factor. However, we have found that as more moves tend to be searched, it is possible to either partition the problem further, thus creating more goal states, or to improve the specification that decides which moves are really pertinent, thus eliminating some moves from the search. This also improves the ordering of proposed moves, and tends to maintain the branching factor about where it was.

The tree search appears to derive most of its power from the partitioning idea, rather than from the set of stopping rules. Of the new stopping rules, the Claim System seems to be the most powerful. Also, the ability to stop and retrace the search (search-and-scan) because of a neglected alternative at a previous node appears important, when it can be implemented correctly.

The partitioning of move selection at a node, however, seems to be mainly responsible for the fact that 50% of all nodes have only one successor. It is the ability to find quickly the correct goal state for a node, and then have only a limited

number of moves be associated with the goal state that provides the selectivity of the tree searching method. As mentioned elsewhere, much improvement in this is still possible. Most of the goal states can be further broken down. The AGGRESSIVE state is a good example. With an adequate amount of analysis of the position at a node, and with theme information to further restrict things, it should be possible to limit the moves that are generated to those attacking a specific piece or dealing only with a small set of squares. This type of further subdivision would probably help the DYNAMIC DEFENCE state most. For deep consequence descriptions, as many as 20 or so moves may be generated. Some further selectivity in the descriptive process or in the goal state must be found here.

E. HOW CAN LEMMAS BE IMPLEMENTED

We have at several places in this document referred to lemmas which could be used to save information from one position to the next, both within a tree and from tree to tree. We will now make these notions more concrete. Lemmas are to be used to save information about a possible move, from a position in which this information is first discovered. Let us look at the four possible categories into which a move can fall with respect to its static evaluation and actual goodness.

1) A move is statically evaluated as good and is good

2) A move is statically evaluated as good and is bad

3) A move is statically evaluated as bad and is good

4) A move is statically evaluated as bad and is bad.

Moves in categories 3 and 4 are never tried by the program. This, incidentally, brings out the importance of evaluating all moves optimistically, since we would like to have the occurence of moves in category 3 be very, very low. When a move is in category 1, there is also no problem since the move will be accepted as good by the program and its view is thus consistent. The moves in category 2 pose the problem that lemmas address. They appear to be good or have some redeeming features, but are not. Furthermore, they end up being tested again and again in slightly different environments, since positions in a search tree are highly auto-correlated. In most instances, the same result comes out --- they are rejected. This is because, although the position is slighly changed, there are still some basic factors in the position that prevent the possibly good move from being good. Lemmas address the retaining of this essential constancy of the position with respect to this move.

Moves in category 2 can be further classified into those appearing to be strictly good and those that are worth trying because they have some redeeming features. In the first class are the moves 1.--QxR and PxR in Figure 1.9 (page I-9). It is relatively easy to construct lemmas for this type of move. When it is tried and refuted, the backing up process will bring back the Refutation Description contained in RPCS, RSQS, RPATH, RTGTS,TGTSQS, and TPATH. To this must now be added the present location of all pieces that participated in the refutation. The lemma also receives a value, which is the amount that would be lost if the move were tried. Then a condition for this move not to be rejected on account of the lemma in future positions is:

1) That one or more of the named pieces no longer is in its old position.

2) That one or more of the named pieces now has a new defensive function, worth at least the value of the lemma, that it has been assigned.

3) That one or more of the squares named as path squares in the Refutation Description is now occupied when it was not before

4) That one or more of the threat path squares has a new piece that can now occupy it and could not have in the position in which the lemma was discovered

5) That one or more of the squares that was an RSQS square is now controlled by an own man when it was not before.

Unless one of these conditions is fulfilled, there should be no essential change in the position which would make the formerly not-good move, now good. Unfortunately this is not a complete mechanism, since it is possible to counter such a refutation indirectly. This means that at the end of such a variation, which formerly was a decisive refutation, we could now have a series of moves which make the refuter sorry he did his "refutation". This will usually take the form of having a man in a position to attack one of the RSQS square, when this man was not in such a position before. However, it could also come about by utilizing a line through a point that was vacated by either side's men during the refutation. However, an implementation which left out some of these fine points would undoubtedly do more good that harm, since refutations of refutations would tend to be rather deep variations that only very good opponents could calculate. Also, the indirect defence mechanisms discussed in Chapter III, Section D, could possibly be used here.

The problem of creating lemmas about a move that was considered worthwhile because it had redeeming features, but did not work out, is similar. Here again the Refutation Description plays a role. What has happened is that the redeeming features were not "redeeming" enough, so that the Refutation Description tells where the redeeming process failed. Therefore a necessary condition for a move with redeeming properties to be tried at some later stage is that one of the following be true:

1) Any of five conditions mentioned above is true.

2) The suggested move has some new redeeming features that it did not have before.

3) An RSQS square is now attackable by one or more of own men which could not perform this service before

While the above is still rather sketchy, there is enough information on how lemmas could be implemented and enough mechanisms already in place to try this soon. It is interesting to note that a much simpified form of this idea has been tried in a bridge analysis program [Berlekamp, 1963)]. It should be noted that it is important to express the lemma in a language which is neither too detailed nor too fuzzy. The former case would result in continuous re-examination as changes in single conditions, which by themselves do not upset the validity of the lemma, have to be looked into. The latter case could result in being unable to define or detect a critical change in the truth value of the lemma. It should be apparent that lemmas can concern themselves with other

things than the material issues presented in the example below. For instance, lemmas could be posited about the conditions under which control of an open file is retained.

To illustrate an actual lemma in a rather simplified situation, consider the position in Figure 4.8 (page IV-31). Here after White plays 1. R-R8 Black tries RxP and finds out that it loses a rook after 2. R-R7ch. Thereafter, Black checks the White king along the rank and after every check tries again to play RxP; each time finding that this loses to R-R7ch. The first time this sequence was discovered a lemma could have been constructed. This lemma is of the type that applies to a move that appears to be good and is in actuallity bad. Here RPCS is the White rook located on KR8. If this rook were to move it would invalidate the lemma. RSQS is the set KR7 and QR7. If either of these squares were to get an additional Black man controlling them or a White man occupying them, it would invalidate the lemma. RPATH is the set of squares KN7, KB7, K7, Q7, QB7, QN7. If any piece were to newly occupy such a square it would invalidate the lemma. RTGTS are the Black king on KB2 and the Black rook which becomes a target on QR2 when it takes the pawn there. If the king were to move this would invalidate the lemma; the rook plays no role, since it comes under attack on a different square from the one it is on now. TGTSQS is the set KB7 and QR7 and as long as no new defensive piece controls them the lemma remains valid. TPATH are the set of squares KN7, KB7, K7, Q7, QB7, QN7. As long as no new piece occupies these, or a defensive piece moves into a position to do so, the lemma remains valid. It is easy to check that the lemma will remain valid as long as the Black rook only moves up and down the QR-file. Since all the Black pieces are involved in the lemma, no additional piece can come to their help (a condition which might make an indirect defence possible). Thus, unless the White king manages to move to a newly precarious position (making a new counter-attack possible) the lemma remains valid. We have not yet determined a method of dealing with the latter situation.

F. SOME SPECULATIONS

The following protocol was collected with myself as subject and data collecter. It was written down immediately after first looking at the position in Figure 5.4 (page V-11). At that time I was struck by the curious inability to see the key move 2. Q-R3ch, which has been confirmed in other good players. However, there are other interesting features in this protocol too. It should be compared to Figure 5.5 (program's tree search) to get an appreciation of the differences in time allocation to problems that exist between a Master and the present program. As an instance of this. consider the amount of effort that it takes the Master to find the move 2. B-Q3ch (after 1. B-R7ch, K-R1) as against the program's effort.

Master Player's Protocol for Position in Figure 5.4

Well lets see here; if something is going to happen here it must be	01
to the Black king. It's either going to be a combination with R-R8ch	02
and the queen mating or something that starts with B-R7ch. Lets see.	03
If R-R8ch then KxR, Q-R4ch, K-N1 and the king is going to get away	04
over K2 unless I can get the (White) rook into it. Well how about	05
B-R7ch. Clearly, K-R1 is no good since it loses the queen; but what	06
about K-B1? Not much point in RxPch or B-Q3 even if it worked. Lets	07
see, B-Q3, RxB, R-R8ch, K-K2, Q-N5ch and K-Q2 and what has White got?	08
Nothing. But (1.) R-R8ch is no good either. Well lets see after	09

B-R7ch, K-B1 at least we haven't sacrificed anything, and if we get the rook to safety then Black's king is still uncomfortable. Q-Q6ch is pointless and there are no other checks --- wait, Q-R3ch, now Black must play R-K2 and the escape square is blocked. So B-Q3 then wins; defends the rook and threatens mate --- and the queen too. So that is the sequence 1. B-R7ch, K-B1, 2. Q-R3ch, R-K2, 3. B-Q3 wins at least the queen. 10 11 12 13 14 15 16

The time for the above protocol was between 60 and 90 seconds. It is interesting to speculate what must have gone on in the Master's head in order for the above protocol to come out at the English language level. Using the model of chess that we have been describing, here is what appears to be a necessary set of operations:

In line 01 the Black king is singled out as the most likely goal for meaningful moves. This means that a static analysis of the board has already discovered that the king is subject to one inevitably sound check (B-R7) and one check with decoy value (R-R8ch). This together with a presence of other White pieces in the vicinity of the Black king and a lack of defending pieces should be enough to invoke an "endangered king" chunk which would then require generating aggressive moves against the Black king. By lines 02 to 03, the two checks appear to have nearly co-equal status in the scheme of things (a condition which is probably noted and could be used for later jumping back to such a neglected alternative when the one currently being searched is not working out according to expectations).

At line 04, the actual search begins. It is hard to tell whether "and the king is going to get away over K2" is a fact recognized statically, or whether the continuation of play (after 1. R-R8ch, KxR, 2. Q-R4ch, K-N1) 3. Q-R7ch, K-B1, 4. Q-R8ch, K-K2 was actually played out but not verbalized. My intuition says that the latter is true. Then, when the king arrives at K2, the fact that it has several escape squares going further toward the queen-side is enough to convince the Master that the king has essentially made it to safety (a chunk of some kind associated with king safety is probably invoked here).

At this point (line 05), some back-tracking occurs, apparently with the idea of finding a way to produce a meaningful role for the White rook on KB1. We know of no exact equivalent for this structure, and it would be done in our projected program by merely trying those rook moves which had merit when they were proposed. However, here the Master has apparently already discovered that moves of the rook, such as RxPch are of no value since the Black bishop on K3 has no other function to perform except guard this pawn. This must have been discovered at the same time that the initial structuring of the board took place and is now being saved in lemma form. This indicates that the sequence 1.RxPch, BxR was possibly tried at the start and a lemma formed upon its rejection. However, the notion of reconfiguring the board so that the lemma is no longer valid appears to be a tool of the Master for which we as yet have no implementation ideas.

When the Master fails to find something that can achieve this transformation, he then recalls the alternative at the top level in the tree and appears to hop directly back to try it (line 06). Again he starts out by classifying the alternatives for Black, apparently with the notion of retaining one for jumping back to, as he did at the previous ply. However, here he appears to select the worse alternative first and then immediately finds a reason for permanently rejecting it (line 06). Thus he has

succeeded in forcing a slimming of the tree under consideration. But after starting to consider the other, better alternative, he finds himself stuck. Apparently the moves that he originally considered to be part of the "aggressive set" do not comprise an adequate continuation now.

The lemma about RxPch is examined again and a new thought intrudes: moving the bishop to Q3 with the idea of clearing the line for a check at R8 and also attacking the Black queen and defending the White rook (line 07). (Notice incidentally, that the Master has no problem in seeing statically that a bishop at Q3 would defend the rook at KB1 through the Black queen at K7). The key move 2. Q-R3 check is not noticed, possibly because it was not in the original set of moves that were associated with the attack, or possibly because it involves moving a piece away from an area in which the attack is taking place in order to deliver a long check. In the latter case a definite concentration of perception appears to be taking place, having to do with sectors of the board. This may be due to a limitation of human short term memory; a problem that a machine apparently need not have. Having only generated the two alternatives, 2. RxPch and 2. B-Q3, the Master again dismisses the former (probably by re-examining the lemma). Apparently he has grave doubts about the latter (possibly because the bishop is moving en prise, and he remembers that in a previous variation the king escaped over K2), but tries it anyway to see if some new knowledge can be discovered. He does a little tree searching (line 08) following out the only line of promising checks until he again comes to a divergence point (after 2. B-Q3, RxB, 3. R-R8ch, K-K2, 4. Q-N5ch, K-Q2 (the latter apparently selected statically as better for king escape purposes than P-B3)). Here he is again faced with a reformulation of the task, and decides that effectively nothing has been accomplished. So he again returns to the top to look at the previously rejected alternative which, however, has not been completely dismissed (line 09).

Now there appears to be a critical comparison made about the sum of knowledge of the two candidate moves at the top (line 09 through 11). The critical distinction appears to be that one line sacrifices material, but since neither line appears to produce anything exceptional, it is probably better to concentrate on the one that at least does not invest any material. Thus there has been a shift of the expectation level, and the Master would now be satisfied to merely keep the Black king uncomfortable while securing the position of his own rook at KB1. When he returns to the critical position after 1. B-Q3ch, K-B1, he therefore starts looking for new moves to investigate (line 12). Before concentrating on the defence of the rook at KB1, he checks over the aggressive possibilities once more. He now notices a check (Q-Q6ch) which had been suppressed before (probably because it was seen to be well guarded against by the Black rook at Q1). Somehow, noticing this diagonal check brings out the possibility of the queen finding another square on this diagonal. Now everything falls into place. The fact that the reply R-K2 (to 2. Q-R3ch) is forced is probably found by extending the search, although this is not verbalized. However, the very thought of K2 becoming blocked appears to evoke something important from a previous analysis, e.g. the fact that the king always escaped across K2. Here there seems to be a lemma relating to the Black king's safety and the free square K2. Once a condition for nullifying this lemma is discovered, everything else works very quickly. The notion of the mate on the back rank evokes the idea of moving the bishop. The bishop is also needed to defend the rook at KB1 as was discovered in a previous variation which failed. The fact that this move also attacks the queen appears to be very secondary (line 14). The rest is merely a verification and summing up procedure.

It is very interesting to note the different kinds of mechanisms invoked here. In the perceptual area (recognizing the merit of moves statically) the program comes off fairly well. It found 2. Q-R3ch immediately, but took a long time to discover the value of 2. B-Q3ch in the variation in which the king went to R1. However, the Master clearly has tree control mechanisms which far outdo the program. He remembers critical nodes to return to when he gets stuck. He remembers facts in manipulable form and then does logical operations on them (i.e. the fact that the Black king always escapes across K2 and that blocking this square may make the king a worthwhile target again). In these, he is able to maintain a locus of control much closer to the problem in the position than what the program is able to achieve. It seems reasonable to suppose that these facilities are not descriptive of Chess Masters alone, but rather of humans as a species. However, in the hands of the Master, the effect is more pronounced. Thus some additional control methods probably still need to be developed to obtain small search trees. There is one area where the program appeared to have an advantage. This is in the perception of Q-R3ch. Probably the initial analysis of the position told the Master to keep his perceptual scanning mechanisms concentrated on the right-hand side of the board. There is no doubt some efficiency involved in this, as the recognizing of features all over the board would no doubt take longer than if only a certain sector had to be dealt with. However, a factor of two appears to be a doubtful saving, although it could possibly matter to an efficiency oriented mechanism. Instead, I wonder if there is not some need in the human to restrict his attention to a small part of the board in order to avoid losing things from short-term memory. This is a problem that computers do not appear likely to have, and thus a certain form of myopia could possibly never plague a program.

The above analysis is very revealing about how search-and-scan is probably implemented in humans. The Master appears to remember nodes at which several competing alternatives existed. Then when the chosen alternative fails to work out according to expectation, he returns to the lowest node in the current variation which had such a choice point. So far this is very similar to our implementation. However, a noticable difference now appears. Instead of arbitrarily chosing the next alternative in line, the Master now sums up the knowledge about what he knows about the various alternatives, and based upon any of several tie-breaking criteria decides to try one (possibly the same one from which he just retreated) or continue back up the tree. The critical difference appears to be that there is some knowledge (possibly lemma like) that has been backed up and is used for comparing the relative merit of alternatives along several dimensions. However, this knowledge must be gained through a validation procedure which at a minimum must characterize what the problem (that caused the move to fall short of expectation) is, and whether no other move existed on the way back up, which overcame this problem. This sounds very much like the standard back-tracking procedure for collecting refutation descriptions. The only thing that is missing is some higher level characterization of what the problem is. Based upon the above considerations, we wonder whether search-and-scan is not a surface phenomenon which is observed, but relies on a different paradigm (depth-first search) for its effect. In that case, knowing that a viable alternative exists at a previous node, becomes merely a signal for back-tracking.

G. SUMMARY AND FUTURE

1. Contributions of this Thesis

The most significant things in this thesis are:

1) The CAUSALITY FACILITY and the method of collecting and using the Refutation Description (Our thoughts were first turned to the subject of collecting descriptions by some statements of Minsky [Minsky, (1968)]).

2) The notion of problem partitioning and Goal States.

3) The analysis of the Horizon Effect, both Positive and Negative, and its import on program design.

4) The demonstration that quiescence must dominate control of the search and the progress made toward achieving this.

5) The demonstration in this program that there are languages (with elements such as bearing relations, paths, and functions) which can serve to describe chess positions in such a way that the description can be used to predicate meaningful action elsewhere in the tree.

6) The Claim System.

7) The implementation of a scheme of functional analysis.

8) The implementation of a tree searching scheme involving very sensitive level of aspiration methods and the ability to progressively deepen a potential solution.

9) The ability to separate offensive from defensive issues in the tree search, and generate defensive moves in response to dynamic requirements.

10) The reintroduction of a basic organization in which moves are generated for positive reasons.

11) The notion of themes (unimplemented).

12) The notion of lemmas (unimplemented).

2. Implications of this Work

We feel that the above aspects of this work have been treated adequately in the pertinent sections of earlier chapters. We would, however, like to call special attention to the following implications of the above.

The understanding of the Horizon Effect is particularly important to progress in Computer Chess. When one builds a program, it is not difficult to understand certain knowledge limitations that the program may have with respect to the domain of discourse. Similarly, if a verification procedure is incorporated in the program as is for chess the tree search, then it is possible to understand that certain things may

not be verifiable. As an instance of the latter, one can consider variations that only produce a definitive result at a depth beyond the maximum depth of the program. However, the Horizon Effect produces results much more insidious than this. It can cause a program to ignore certain truths that it has already discovered. Thus the Horizon Effect is a source of almost unpredictable error -- for every new item of knowledge in the evaluation, a new error can result. This type of error, since it is uncontrollable whenever a maximum depth exists, produces an upper bound on the effectiveness of any such program. Since all programs (including human ones) must have a maximum depth, we can only insist that such maximum depth be so far away from the top node in the tree, that in essentially all cases the effects being investigated have died out by the time the maximum depth is reached. That is the importance of achieving quiescence, and without it a program will never play Master chess (nor probably Class A either).

It is not clear whether the self-converging search that is needed for complete quiescence analysis is possible with the mechanisms developed and discussed in this thesis. However, on the basis of results already achieved with this program, it appears that the additional tree slimming that will be made possible by themes and lemmas will come close. This would create a vehicle which will allow a very large degree of custom handling of each node. Part of this custom handling would consist of detecting chunks of positional information by consulting a discrimination net of such information. This would then allow better classification of positions which in turn would allow better goal state discrimination. It appears as if additional facilities to supplement the tree search are also needed. An instance of this is the ability to analyze null-move sequences. Such a facility could then be used for 1) Determining if a statically detected threat can really be carried out, 2) Postulating indirect defences (were it carried out, if I then took two moves in a row, what could I do), and 3) Analyzing the strategic feasibility of a plan by synthesizing the plan elements (moves) without initially considering opponent's replies.

Further, even the units that are already performing well will probably have to be improved in order to achieve the type of efficiency that humans already possess. The example in Figure 3.22 (page III-30), which showed a basic deficiency in the CAUSALITY FACILITY, is indicative that a still higher level language may be required to separate a sequence of issues contained in a Refutation Description. This may consist of gathering frequently occurring configurations of the current descriptive elements (functions, paths, squares, etc.) and giving names to these.

There appears to be little doubt that such improvements in level of abstraction is one of the powerful tools for improving programs. In this way they will be able to come to grips with problems in less time than it would take for a program that has a less abstract representation and which thus must pay the exponential cost of tree searching to make up for this. In this program, the functions, and the ability to generate and exploit different types of functions, make up the highest level of abstraction. This is quite good by present standards and is very helpful in guiding the program in being more selective. However, it is possible to consider a pattern of functions around an important square on the board as a chunk of chess information. Similarly, all the functions being performed by a specific piece could also be considered a chunk describing that piece's role in the scheme of things. These chunks are essentially algorithmic in nature, and appear to involve nothing more than collecting the presently generated functions around some focus. It may

then be possible to give names to such collections, depending upon its members. Besides this type of computed chunk, there are also chunks which seem to depend very little on the notion of what moves are possible. These chunks identify long-range goods and bads, and are essentially spatial in nature. They include such notions as what a safe king position is, and that certain pawn configurations are superior to others. Such higher level abstractions can be identified by consulting a discrimination net of known abstractions, to see if the one at hand is known. The implications of such a match then become actions for the program to try, and information that could be used in evaluation.

On the basis of our experience with level of abstraction in this program, we conjecture that continuing progress in chess will be dependent on the invention of ever higher level languages in which chess concepts can be expressed. Each such language level would then have an asymptote, defined by the power of the language, beyond which it would not be possible to improve the strength of the program, given that only a certain amount of time was available to compute a move. Also, as concepts are agglomerated into ever higher level concepts, we expect that they would get to be more fuzzy and would require a more complex control structure than used at present in order to produce the same level of reliability as can be obtained with less fuzzy concepts. We feel that this increased conceptualization is evident in the history of chess, and should make it possible ultimately to equal and exceed the performance of the best human players.

The current method of controlling level of aspiration and expectation in the tree appears to work very well. Further, it appears to be very much like what the author feels he does when playing chess. The deepening of the search also follows the pattern of our own play, except that in the deepening process the program has gained knowledge only of the new expectation level. Instead humans also have refined their understanding of the position, and this is reflected in their not trying certain alternatives over again; not only because they do not meet the aspiration levels, but because something was noticed about the move during the last analysis, which changes the understanding of its candidacy. This feature of human search will hopefully be adequately supported by lemmas. During the tree search it is almost inevitable that some lemmas would be created at the top level of the tree, and these would then be used to guide the deepening process, and in doing the tree search after the opponent's next move.

At present it is common to treat chess programming as a problem of imparting knowledge to a program. Yet we have pointed out that present models of chess have tremendous deficiencies in tree searching, representation of knowledge, and to some degree in organization of effort. Until a harmonious solution to these problems is found, it is not likely that infusion of more knowledge into a program will be of any help. However, when a solution to the above exists, the problem of producing a Chess World Champion may very well reduce to devising a method for such a program to acquire knowledge from experience.

3. Criteria for Progress

Finally, I would like to propose three simple tests for deciding when a program is ready to compete with good human players:

1) When such a program, recognizing that it can win some material, deliberately postpones such a maneuver, in the realization that it can be achieved later, possibly to greater advantage. This involves having a lemma that recognizes the material winning sequence and realizes that it still applies at the horizon. Thus the program will not be fooled into converting its advantage prematurely (Positive Horizon Effect) when a reason exists for delaying this gain.

2) That such a program be able to win positions such as the one in Figure 1.7 (page I-9). This involves having a long range planning facility.

3) That such a program, when leaving its opening book, be able to continue the ideas behind the moves it previously made. This means that the moves that have been made so far must be associated with themes, which can then be used for guiding future play. It also means that certain lemmas may have to be posited (i.e. this pawn was sacrificed in order to get a lead in development, and is not considered to be a great asset to the opponent; e.g. don't be in a hurry to win it back).

AGGRESSIVE - A goal state that generates all moves that result in attacks on low mobility pieces, double attacks, discovered attacks, captures, and moves that vacate important squares.

ALPHA - A value, associated with a node in the search tree, which represents the best result that the side not-on-move at this node can hope to achieve.

ALPHA-BETA - A tree pruning algorithm which eliminates branches in a depth-first mini-max search which can logically not yield an optimum solution since they have already been superceded elsewhere in the tree.

ASPIRATION (level of) - A control scheme, involving the variables EXPCT, ALPHA and BETA, and the constant MARG, that determines when values found in the tree search make it logical to back up.

BEH - The relation of a vertically bearing piece bearing on a pawn from behind, thus being able to control the square(s) in front of it as it advances.

BETA - A value, associated with a node in the search tree, which represents the best result that the side on move at this node can hope to achieve.

BRD - A bit-vector which indicates which squares are occupied.

CAUSALITY FACILITY - A module of the program which can determine whether a set of consequences can be definitely dissociated from the last move tried at a node. The module also generates a set of counter-causal moves.

CLAIM SYSTEM - A method of reducing the alpha value at a node to the maximum expectation of the opponent of the side to move at this node.

CLR(SQ1,SQ2) - Defines the set of all squares on a straight line between squares SQ1 and SQ2.

COUNTER-CAUSAL MOVES - A bag of moves which are generated by the CAUSALITY FACILITY in order to counter-act a Refutation Description.

DECOY - A motive in a sacrifice of a man, intending to lure a more valuable man to the capture square.

DIR(PC,SQ) - A bearing relation such that 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.

DISTRACT VALUE (of a move) - The value of a move in getting an opposing man to quit another attack in the act of capturing the moving man.

DSC(PC,SQ) - A bearing relation such that 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.

DYNAMIC DEFENCE - A goal state which results in the producing of moves to counteract the causal description of some undesirable consequences that have been backed-up to this node.

DYNAMIC EVALUATION - Evaluation of a position or move using a tree search.

EN PRISE - A chess term indicating that a man is presently (partly) endangered.

ETHRU(PC,SQ) - A bearing relation such that 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.

EVALUATE - The sub-routine that statically evaluates proposed moves.

EXPCT - The expected value of the position represented by the top node of the search tree.

FEATRS - A sub-routine of the program which computes the values of a set of features of the chess board.

FUNCTION(PC,SQ,DUTY) - A binding of a piece, PC, to an important role, DUTY, on square, SQ. Roles can consist of Attacking, Defending, Blocking, Overprotecting, Supporting, Pinning, and Escape Square Guarding.

GOAL STATE - A state that the tree search at a node can be in, which governs the types of moves selected for searching.

HORIZON EFFECT - A problem that exists in tree searches to fixed depths which causes errors in terminal evaluation since not all terms in the evaluation function are driven to quiescence.

INT - A vector that contains for every piece type the locations of all opposing pieces that would become en prise if they were attacked by a piece of this type.

INTERPOSE - A sub-routine that takes a pair of arguments which define a straight line and then generates all the pseudo-legal moves for the side on move to squares between the two arguments which define the end points.

KING IN CHECK - A goal state that produces all legal moves when the king is in check.

LOW MOBILITY PIECE (LMP) - A piece which has sub-standard mobility and is therefore judged to be a good target.

MARG - A constant that defines the range of aspiration around EXPCT. If a value which differs from EXPCT by more than MARG is found, it is said to differ significantly from expectation.

MOVEAWAY - a sub-routine of one argument that generates all the pseudo-legal move of the piece occupying the argument square.

MOVTOCON - A sub-routine which takes as a single argument and generates all the pseudo-legal moves of the side on move by pieces which at the momoment do not bear DIR on the argument square, and which result in such a piece now bearing either DIR on the argument square or OTHRU or ETHRU through a piece which has a function on the argument square.

NOMINAL DEFENCE - A goal state which results in producing moves directed solely at defending against threats that are statically detected but not necessarily proven to be effective.

NOMINAL VAUE (of a position or node) - That value resulting from an evaluation of the position which is the best estimate of its value.

OBJ(PC,SQ) - A bearing relation such that 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.

OCCUPY - A sub-routine of one argument that generates all the pseudo-legal moves of the side on move that result in occupying the square named in the argument.

OCCUP - A sub-routine that computes Occupiabilities under differing conditions.

OCCUPIABILITY - A quantity that defines the degree of safety of a square for pieces of a given side.

OCY(COLOR, SQ) - The value of Occupiability for square SQ and side COLOR.

OTHRU(PC,SQ) - A bearing relation such that 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.

OVERLOAD (DEFENSIVE) - A motive in making a sacrificial move, intending that a defender against the move made wil be forced to relinquish another defensive role.

PESSIMISTIC VALUE (of a position or node) - That value of the position that considers all threats of the side not-on-move to be executable while considering none of the threats of the side on-move.

PIECE - A chess piece, synonomous with chess man.

PIN - A piece is said to be pinned, if moving it would allow the capture of a more valuable piece of its own side, when such a capture would not otherwise be possible.

PIN OBJECT - The piece that would be captured if a pinned man were to move.

PREVENTIVE DEFENCE - A goal state which is invoked when the side on move is materially ahead of expectation, and results in generating moves intended to preserve the material advantage.

PSEUDO-LEGAL MOVE - A move that would be legal if leaving or moving a king into check were legal, and if castling while in check were legal.

REDEEMING VALUE (of a move) - A function of defensive overload, decoy value, and distract value, that is assigned to a move that on initial evaluation does not appear to be good.

REFUTATION DESCRIPTION - A description of a set of tactical consequences in a sub-tree which is accumulated and backed up for use by the CAUSALITY FACILITY.

RPATH - Part of the Refutation Description: the set of squares across which the refuting pieces moved.

RPCS - Part of the Refutation Description: the set of pieces which participated (moved) in the refutation.

RSQS - Part of the Refutation Description: the set of squares to which the refuting pieces moved.

RTGTS - Part of the Refutation Description: the set of pieces which became newly attacked during the refutation.

SETUP - The sub-routine that calculates the basic bearing relations once a new position has been reached in the analysis.

STATIC EVALUATION - Evaluation of a position or move using analysis techniques not involving tree searching.

STRATEGY - A goal state which results in moves being proposed one at a time from a "strategy" source (TECH's positional move generator) and being tried until a satisfactory one is found.

TGTSQS - Part of the Refutation Description: the set of squares on which pieces, that became newly attacked during the refutation, resided.

TPATH - Part of the Refutation Description: the set of squares across which passed threats on newly attacked pieces.

TYPE - The type of a chess piece, consisting of one of the set pawn, knight, bishop, rook, queen, or king.

VUE(SQ,TP) - The set of all squares that would be legally accessible to a piece of type TP, located on square SQ on an otherwise empty board.

Atkin, L.R., Gorlen, K., and Slate, D. (1971), "Chess 3.0 - An Experiment in Heuristic Programming", (Northwestern University), *Unpublished*, 1971.

Baylor, G.W., and Simon, H.A. (1966), "A Chess Mating Combinations Program," *Proceedings of AFIPS 1966, SJCC*, Vol. 28, 1966, pp. 431-447.

Berlekamp, F. (1963), "Program for Double Dummy Bridge Problems", *Journal of the ACM*, Vol. 10, 1963, pp. 357-364.

Berliner, H.J. (1970), "Experiences Gained in Constructing and Testing a Chess Program", *Proceedings of the IEEE Sysmposium on Systems Science and Cybernetics*, October 1970.

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.

Bernstein, A., et. al. (1959), "A Chess Playing Program for the IBM 704", *Proceedings of the Western Joint Computer Conference, AIEE*, March 1959, pp. 157-159.

Bloss, F. D. (1972), *Rate Your Own Chess*, Van Nostrand-Reinhold Co., 1972.

Botwinnik, M.M. (1970), *Computers, Chess and Long-range Planning*, Springer Verlag, 1970.

Baylor, G.W., and Simon, H. A. (1966), "A Chess Mating Combinations Program", *Proceedings Spring Joint Computer Conference*, 1966, pp. 431-447.

Chase, W. G. and Simon, H. A. (1973a), "Perception in Chess", *Cognitive Psychology*, Vol. 4, No. 1, January 1973.

Chase, W. G. and Simon, H. A. (1973b), "The Minds Eye in Chess", Chapter 8, pp. 215-281, in *8th Annual Psychology Symposium Volume: Visual Information Processing*, Chase, W. G. (Ed), Academic Press, 1973.

De Groot, A.D. (1965), *Thought and Choice in Chess*, Mouton and Co., 1965.

Fuller, S.H., Gaschnig, J. G., and Gillogly, J.J. (1973), "Analysis of the Alpha-Beta Pruning Algorithm", Computer Science Dept., Carnegie-Mellon Univ., July 1973.

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

Greenblatt, R.D., et. al. (1967), "The Greenblatt Chess Program", *Proceedings of the 1967 Fall Joint Computer Conference*, pp. 801-810.

Kister, J., Ulam, S., Walden, W. Wells, M. (1957) "Experimentas in Chess", *Journal of the ACM*, Vol. 4, No. 2, April 1957, pp. 174-177.

Kotok, A. (1962), "A Chess Playing Program for the IBM 7090", *Bachelors Thesis*, MIT 1962.

Kozdrowicki, E. W. (1968), "An Adaptive Tree Pruning System: A Language for Programming Realistic Tree Searches, *Proceedings ACM National Conference*, 1968, pp. 725-735.

Kozdrowicki, E. W., and Cooper, D. W. (1973), "COKO III - The Cooper-Kozdrowicki Chess Program", *Journal of the ACM*, July 1973, pp. 411-427.

McCarthy, J., et. al. (1965), *Lisp 1.5 Programmer's Manual*, MIT Press, 1965.

Minsky, M. (1968), "Introduction", in *Semantic Information Processing*, M. Minsky (Ed.), MIT Press, 1968.

Newell, A. and Simon, H.A. (1972), *Human Problem Solving*, Prentice-Hall, 1972

Newell, A., et. al. (1963), "Chess Playing Programs and the Problem of Complexity", in *Computers and Thought*, E.A. Feigenbaum and J. Feldman (Eds.), McGraw-Hill, 1963.

Newell, A. (1955), "The Chess Machine: An Example or Dealing with a Complex Task by Adaptation", *Proceedings Western Joint Computer Conference*, pp. 101-108, 1955.

Nilsson, N. J. (1971), *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

Pachman, L. (1973), *Attack and Defence in Modern Chess Tactics*, David McKay Co., Inc., 1973.

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

Shannon, C.E. (1950), "Programming a Computer to Play Chess", *Philosophy Magazine*, Ser. 7, Vol. 41, No. 314, March 1950, pp. 256-275.

Slagle, J. R. (1971), *Artificial Intelligence: The Heuristic Programming Approach*, McGraw-Hill, 1971.

Slagle, J. R. and Dixon, J. K. (1969), "Experiments with some Programs that Search Game Trees", *Journal of the ACM*, Vol. 16, No. 2, April 1969, pp. 189-207.

Strachey, C.S. (1952), "Logical or Non-Mathematical Programs", *Proceedings ACM National Meeting*, 1952, pp. 46-49.

Simon, H. A., and Gilmartin, K. (1974), "A Simulation of Memory for Chess Positions", *Cognitive Psychology*, Vol. 5, pp. 29-46, 1974.

Slater, E. (1950), "Statistics for the Chess Computer and the Factor of Mobility", *Symposium on Information Theory*, pp. 150-152, London, Ministry of Supply, 1950.

Wulf, W.A., Russell, D.B., and Habermann, A.N., (1971), "BLISS: A Language for Systems Programming", *Communications of the Association for Computing Machinery*, Vol. 14, No. 12, December 1971.