Stanford Artificial Intelligence Laboratory
Memo AIM-329

July 1979

Computer Science Department
Report No. STAN-CS-79-747

USING PATTERNS AND PLANS TO SOLVE PROBLEMS AND CONTROL SEARCH

by

David Edward Wilkins

Research sponsored by

Advanced Research Projects Agency
and
National Science Foundation

COMPUTER SCIENCE DEPARTMENT
Stanford University

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 12 STAN-CS-79-747 AIM-329	2. GOVT ACCESSION NO. 9,T	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Using Patterns and Plans to Solve Problems and Control Search.		5. TYPE OF REPORT & PERIOD COVERED technical, July 1979
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-79-747 (AIM-329)
7. AUTHOR(s) David Edward/Wilkins 1		8. CONTRACT OR GRANT NUMBER(s) ARPA MDA903-76-C-0206 and NSF MCS 78-00524
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 14 3
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Avenue, Arlington, Virginia 22209 11		12. REPORT DATE Jul[illegible]79
		13. NO. OF PAGES 264
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) Mr. Philip Surra, Resident Representative Office of Naval Research, Durand 165 Stanford University		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)

The type of reasoning done by human chess masters has not been done by computer programs. The purpose of this research is to investigate the extent to which knowledge can replace and support search in selecting a chess move and to delineate the issues involved. This has been carried out by constructing a program, PARADISE (PAttern Recognition Applied to DIrecting SEarch), which finds the best move in tactically sharp middle game positions from the games of chess masters.

PARADISE plays chess by doing a large amount of static, knowledge-based

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20. ABSTRACT (Continued)

reasoning and constructing a small search tree (tens of hundreds of nodes) to confirm that a particular move is best, both characteristics of human chess masters. A "Production-Language" has been developed for expressing chess knowledge in the form of productions, and the knowledge base contains about 200 productions written in this language. The actions of the rules post concepts in the data base while the conditions match patterns in the chess position and data base. The patterns are complex to match (search may be involved. The knowledge base was built incrementally, relying on the system's ability to explaine its reasoning and the ease of writing and modifying productions. The productions are structured to provide "concepts" to reason with, methods of controlling pattern instantiation, and means of focusing the system's attention on relevant parts of the knowledge base. PARADISE knows why it believes its concepts and may reject them after more analysis.

PARADISE uses its knowledge base to control the search, and to do a static analysis of a new position which produces concepts and plans. Once a plan is formulated, it guides the tree search for several ply by focussing the analysis at each node on a small group of productions. Expensive static analyses are rarely done at new positions created during the search. Plans may be elaborated and expanded as the search proceeds. These plans (and the use made of them) are more sophisticated than those in previous AI systems. Through plans, PARADISE uses the knowledge applied during a previous analysis to control the search for many nodes.

By using the knowledge base to help control the search, PARADISE has develope an efficient best-first search which uses different strategies at the top leve. The value of each move is a range which is gradually narrowed by doing best-first searches until it can be shown that one move is best. By using a global view of the search tree, information gathered during the search, and information produced by static analyses, the program produces enough terminations to force convergence of the search. PARADISE does not place a depth limit on the search (or any other artificial effort limit). The program incorporates many cutoffs that are not useful in less knowledge-oriented programs (e.g., restrictions are placed on concatenation of plans, accurate forward prunes can be made on the basis of static analysis results, a causality facitity determines how a move may affect a line already searched using patterns generated during the previous search).

PARADISE has found combinations as deep as 19 ply and performs well when tested on 100 standard problems. The modifiability of the knowledge base is excellent. Developing a search strategy which converges and using a large knowledge base composed of patterns of this complexity to achieve expert performance on such a complex problem is an advance on the state of the art.

DD FORM 1 JAN 73 1473 (BACK)

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Stanford Artificial Intelligence Laboratory
Memo AIM-329

Computer Science Department
Report No. STAN-CS-79-747

July 1979

USING PATTERNS AND PLANS TO SOLVE PROBLEMS AND CONTROL SEARCH

by

David Edward Wilkins

ABSTRACT

The type of reasoning done by human chess masters has not been done by computer programs. The purpose of this research is to investigate the extent to which knowledge can replace and support search in selecting a chess move and to delineate the issues involved. This has been carried out by constructing a program, PARADISE (PAttern Recognition Applied to Directing SEarch), which finds the best move in tactically sharp middle game positions from the games of chess masters.

PARADISE plays chess by doing a large amount of static, knowledge-based reasoning and constructing a small search tree (tens or hundreds of nodes) to confirm that a particular move is best, both characteristics of human chess masters. A "Production-Language" has been developed for expressing chess knowledge in the form of productions, and the knowledge base contains about 200 productions written in this language. The actions of the rules post concepts in the data base while the conditions match patterns in the chess position and data base. The patterns are complex to match (search may be involved). The knowledge base was built incrementally, relying on the system's ability to explain its reasoning and the ease of writing and modifying productions. The productions are structured to provide "concepts" to reason with, methods of controlling pattern instantiation, and means of focusing the system's attention on relevant parts of the knowledge base. PARADISE knows why it believes its concepts and may reject them after more analysis.

PARADISE uses its knowledge base to control the search, and to do a static analysis of a new position which produces concepts and plans. Once a plan is formulated, it guides the tree search for several ply by focussing the analysis at each node on a small group of productions. Expensive static analyses are rarely done at new positions created during the search. Plans may be elaborated and expanded as the search proceeds. These plans (and the use made of them) are more sophisticated than those in previous AI systems. Through plans, PARADISE uses the knowledge applied during a previous analysis to control the search for many nodes.

By using the knowledge base to help control the search, PARADISE has developed an efficient best-first search which uses different strategies at the top level. The value of each move is a range which is gradually narrowed by doing best-first searches until it can be shown that one move is best. By using a global view of the search tree, information gathered during the search, and information produced by static analyses, the program produces enough terminations to force convergence of the search. PARADISE does not place a depth limit on the search (or any other artificial effort limit). The program incorporates many cutoffs that are not useful in less knowledge-oriented programs (e.g., restrictions are placed on concatenation of plans, accurate forward prunes can be made on the basis of static analysis results, a causality facility determines how a move may affect a line already searched using patterns generated during the previous search).

PARADISE has found combinations as deep as 19 ply and performs well when tested on 100 standard problems. The modifiability of the knowledge base is excellent. Developing a search strategy which converges and using a large knowledge base composed of patterns of this complexity to achieve expert performance on such a complex problem is an advance on the state of the art.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206 and National Science Foundation under Contract NSF MCS 78-00524. or any agency of the U. S. Government.

Acknowledgments

Sincere thanks to:

Hans Berliner, my adviser, for his patience, ideas, expertise, inspiration, and careful reading of this dissertation,

John McCarthy, my adviser, for being a reader, and for his confidence and support,

Bruce Buchanan, for reading this dissertation and making many helpful suggestions,

Paul Martin and Dick Gabriel, for their suggestions, hacking, and entertainment,

the Stanford Artificial Intelligence Laboratory and the people who make it work, for the computing aids and research environment that were essential to completing this research,

Pat ' ayes, for introducing me to artificial intelligence,

Juanita, for her love, and

my parents, for everything.

Table of Contents

CHAPTER I

Overview of the Problem

A) Patterns and Intelligence

The word "pattern" has 21 different meanings in the Oxford English Dictionary, yet none of them is as general as the meaning intended in the following discussion. A pattern is any collection of features that are in some way interrelated. This set of features is generally recognized in the environment by some entity, whether it be some form of organic life or a digital computer. A large part of human intelligence can be viewed as the process of matching stored patterns, as has been argued in such books as Pattern Matching as an Essential in Distal Knowing [Campbell66] and Behind the Mirror [Lorenz73].

Many domains of artificial intelligence (e.g., scene analysis, speech understanding) match objects they refer to as "patterns", and there is even a field referred to as "pattern recognition". These uses of patterns generally involve extraction of features from the input stream which may involve a parsing process using syntactic information. The features produced act as primitives for the higher level algorithms which do the actual semantic analysis in feature space. This does not appear to be the type of pattern matching which underlies human intelligence, although it may be a necessary part. One of the central concerns of this research is to use the semantic information of the patterns and the pattern matching process to actually do the analysis (at the feature level) and produce intelligent reasoning or behavior. On a complex problem, this may require formulating intermediate concepts during the matching process. Systems like MYCIN ([Shortliffe74]) can be viewed as expert systems which match patterns, but they do not develop such intermediate concepts in their reasoning (see chapter 2).

Humans often combine a searching process with pattern-based reasoning to solve problems. The searching process is used to verify the correctness of a proposed solution, to change a proposed solution, or to propose new solutions which better solve the problem. There are many examples. A scientist suggests a theory and then searches, through a series of experiments, to verify the theory. A chess master suggests a plan and then searches the consequences to see if it succeeds. A doctor suggests a diagnosis and then searches tests and symptoms to verify it.

The research described here selects a problem which requires search so that interesting questions about the interaction of search and knowledge can be investigated. Issues that are investigated include the problems of expressing and using pattern-based knowledge, the extent to which the pattern-based knowledge can be used to control the search and communicate discoveries in the search tree, and the requirements search and knowledge place on each other. These issues are investigated while solving an intellectually challenging problem. The processes of pattern matching and search seem important to human intelligence and may hold many insights into machine intelligence.

B) Chess as a Domain for AI research

This research uses the game of chess as its domain. This section and the next provide some background information on chess; readers familiar with computer chess and its literature may skip ahead to section D on page 11. Chess is an excellent example of a domain where humans seem to do pattern-based reasoning followed by search (see below). One of the central concerns of artificial intelligence is expressing knowledge and reasoning with it. Chess has been one of the most popular domains for AI research, yet brute force searching programs with very little chess knowledge play better chess than programs with more chess knowledge. There is still much to be learned about expressing and using the kind of knowledge involved in playing chess. CHESS 4.7 is probably the best chess program, and its authors make the following comments in [Slate77]:

But why is the program so devoid of chess knowledge? It is not because we are deliberately trading it for speed of execution. . . . It is not quite true to say that we don't know how to feed additional knowledge to the program. . . . Our problem is that the programming tools we are presently using are not adequate to the task. Too much work is needed to encode the sizable body of knowledge needed to significantly improve the quality of play.

Current computer chess programs do not reason about chess the way human chess masters do. Human masters, whose play is still much better than the best programs, appear to use a knowledge intensive approach to chess (see next section). They seem to have a huge number of stored "patterns", and analyzing a position involves matching these patterns to suggest plans for attack or defense. This analysis is verified and possibly corrected by a small search of the game tree. Such a search is often necessary, as the following positions show.

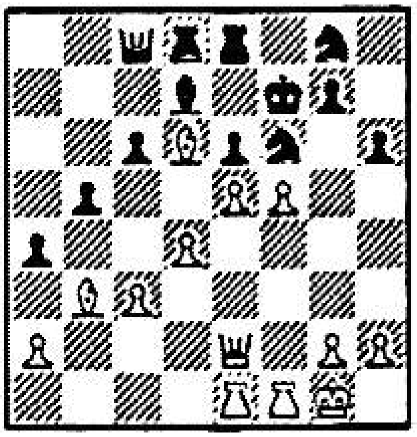


Figure 1.1
white to move

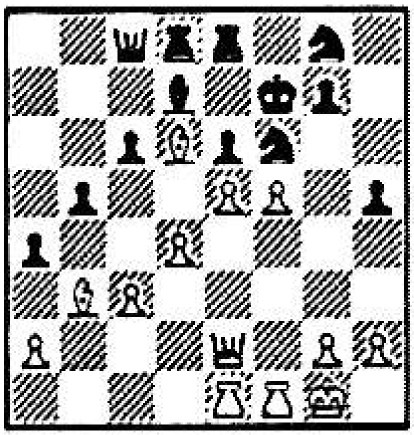


Figure 1.2
white to move

The position in figure 1.1 comes from an actual exhibition game played by a former world champion. He sacrificed his queen to begin a mating sequence which takes 19 ply if black chooses the line of longest resistance. The sequence is: 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. In the position of figure 1.2, black's pawn on his KR3 has been moved forward to KR4. In this very similar position, the queen sacrifice no longer leads to a won position (as noted in [Harris77]). Black can play K-R3 on his fourth move, and white can no longer deliver checks. It is unlikely that a chess player, human or computer, would be able to recognize the different effects of the queen sacrifice in these two positions without searching to verify the results of his static analysis.

As a domain, chess has many advantages for the type of knowledge engineering research being undertaken here. Some of these advantages are briefly summarized below:

1- **Chess is complex.** Although the game has been studied for hundreds of years, it is still intellectually challenging to humans and far from solved. The game tree is finite, but can be considered infinite in practice.

2- **Chess has a simple structure.** This complex problem comes from a set of very simple rules, so a chess researcher can spend his time studying the use of knowledge rather than representing the primitives of the domain.

3- **Much is known of how humans play.** Psychological research gives us insights into the way human masters play chess. This can be helpful in getting a program to do the type of knowledge-based reasoning a human does.

4- **Much is known of how computers play.** Chess has been a popular domain for AI research so the weaknesses of many techniques have already been analyzed. The problems have been well defined.

5- **A large body of knowledge already exists.** Hundreds of books have been written about chess through the years. The chess researcher already has a well thought out body of concepts about the game. He can spend his time getting a computer to reason with these concepts without first having to develop the concepts himself.

6- **Work in chess is easily sharable.** Most researchers in artificial intelligence know how to play chess, so they can easily understand and judge the program. When programs deal with more specialized fields (e.g., specialized branches of medicine or chemistry), most computer scientists must rely on the opinions of others to judge their performance.

7- **Work in chess is easily evaluated.** When a chess program is given a problem, it is usually easy to tell if it solves it correctly. More importantly, the performance of chess programs can be compared to the performance of many earlier chess programs, giving an accurate idea of where the power in each approach comes from.

8- **Chess can be readily decomposed into sub-games.** It would be a large task to encode all the knowledge of a human chess master, but chess can be easily sub-divided. A program can play the opening, the middle game, or many type of endgames without understanding all of chess.

C) Previous Work

This section briefly summarizes some of the work on which this research is based. Part 1 summarizes psychological studies of human chess masters, part 2 describes some computer chess programs, and part 3 describes the MYCIN system. Readers familiar with these subjects should skip ahead to section D on page 11.

1) Human performance studies

An excellent summary of psychological research on human chess players is given in [Charness77]. The interested reader is referred there for more details and for support of the general comments made below.

A major observable difference between human grandmasters and more average chess players appears to be in the initial perception of the position. Important features and moves stand out for a grandmaster as soon as he perceives a position. Experiments have shown this to be due to a chess-specific ability, and not to some unusual memory capacity. The current theory says humans store some small number of "chunks" in short term memory during perception. Grandmasters supposedly have a large number of stored chess "patterns" that have been developed over time. By matching one of these patterns, grandmasters can store as one chunk (pattern match) a large amount of information about the perceived position. This enables them to quickly store (and retrieve) much of the information about the position in only a few chunks (or patterns). Simon and Gilmartin estimate a human grandmaster has on the order of 10,000 to 100,000 stored patterns ([Simon73a]).

It appears the human masters do an analysis based on their stored patterns which suggests moves and plans, but they must search to verify their possibly erroneous analysis. Most of the current ideas on how human chess players search have come from protocols in which chess masters think out loud as they solve a problem. They search ahead, mentally playing moves for each side, to investigate the results of a particular move. The search has been called "progressive deepening" ([deGroot65]), because masters frequently search a line which has already been searched, several times to a deeper level. A second search of a line is frequently initiated after the master discovers some piece of information while searching another line. Masters use heuristic knowledge (supposedly from their "patterns") to restrict the search to relevant moves. The trees generated have on the order of 0 to 100 nodes. Each search of a line is usually terminated with some kind of evaluative statement. This evaluation also appears to come from the knowledge stored in patterns.

Humans appear to have one more essential ability. They can quickly and easily modify their pattern-based knowledge. This learning is evidenced by the fact that chess masters do not tend to repeat the same mistake.

To summarize, human masters appear to use pattern-based knowledge to analyze a position. They then conduct a search, controlled by pattern-based knowledge, to verify or correct the results of the analysis. (These claims are supported in [Charness77] and [deGroot65].) The research described here develops some of the tools and techniques needed for a computer to use this approach to play chess, and defines some of the issues involved.

2) Chess programs

Most of the better current chess programs (e.g., CHESS 4.7, KAISSA, CHAOS) rely heavily on searching and make only a minimal use of chess knowledge. Their only use of chess knowledge generally is in the calculation of legal moves, in the detection of simple features (i.e., features inexpensive to compute) used to order moves for searching, in determining which moves to allow during quiescence searching (i.e., a search beyond the depth limit done to quiesce the evaluation function), and in the evaluation function which gives an integer value for each chess position. Nowhere do these programs have the type of pattern-based knowledge characteristic of humans, and they cannot develop conceptual ideas about a position. Yet they play good chess. One of the most interesting things computer chess has demonstrated is that search covers ignorance very well.

Many things are problematic to a program which relies primarily on a minimax search (e.g., horizon effects, quiescence), and these are well documented in [Berliner74]. One problem is that all current search-oriented programs have a depth limit to keep the search tractable. They therefore cannot find combinations deeper than this depth (unless the quiescence search happens to find them). A search can be permitted to go to an arbitrary depth only if large amounts of knowledge are used to control the searching process, as is done in this research.

Of the many chess programs which have been written, a few are especially appropriate for comparisons. These programs are listed below with only a general description of their overall approach to chess. They are discussed in more detail in later chapters and in the references given.

TECH [Gillogly72]. This program was developed as a benchmark for what could be accomplished using search with very little knowledge. It uses a brute force search of all legal moves with material being the only evaluation function. A simple positional analysis orders moves for searching, and breaks ties between materially equal moves. Quiescence searching consists of trying sequences of captures. Tens to hundreds of thousands of nodes may be searched. Play is weak by human standards. TECH's performance is discussed in more detail in chapter 6.

TECH2 [Hayes76]. This program is an improved version of TECH. It searches more nodes because of more efficient coding and has a better quiescence search. Hundreds of thousands of nodes may be searched.

CHESS 4.4,4.6,4.7 [Slate77]. These are versions of the Northwestern program which has won numerous U.S. Computer Chess Championships, and is undoubtedly one of the strongest programs currently in existence. This program also relies primarily on a tree search which investigates all legal moves and generates tens of thousands of nodes. It relies heavily on a hash table which stores previously computed values so they need not be recalculated elsewhere in the search. The program spends some time ordering moves for searching, and this uses some (though little) chess knowledge. The evaluation function incorporates chess knowledge by checking a number of features in each position. The search uses iterative deepening, which means the search consists of a series of iterations. Each iteration is a full width search to a given depth with quiescence, and each subsequent iteration is a ply deeper than the previous one. The search has a number of heuristics which use the hash table and the previous search iteration to control tree growth and order moves for searching. (An accurate ordering for moves improves the efficiency of the alpha-beta algorithm, thus justifying doing a number of searches to different depths.) The quiescence search considers some checks in addition to captures. The performance of CHESS 4.4 is discussed in more detail in chapter 6.

KAISSA [Adelson-Velskiy75]. This program won the First World Computer Chess Championship in 1974. It also relies on a tree search which searches all legal moves to a certain depth while generating tens of thousands of nodes. There are some methods for controlling the search that have not been used in other programs. The most important of these is the "method of analogies" which makes use of information returned by an earlier search to avoid searching hopeless lines. This is discussed in more detail in chapter 4.

TYRO [Zobrist73]. Unlike the above programs, this one does not search all legal moves. It matches patterns in the position in order to suggest and order moves for the search. These patterns are also used by the evaluation function. The patterns are not very sophisticated; they cannot generate concepts, produce plans or strategies, or use information from a search. The program searches tens of thousands of nodes to discriminate between moves suggested by its analysis. It is discussed in more detail in chapters 2 and 6.

CAPS [Berliner74]. This program does a deeper analysis using more knowledge than any of the programs mentioned above. It determines the functions each piece is performing and looks for perturbing effects to suggest moves for searching. The program also employs knowledge to control the search and to analyze results from the search. The latter is done primarily by the "causality facility" which is discussed in detail in chapter 4. CAPS has more knowledge about tactics than about other aspects of chess. It plays tactical positions about as well as TECH and is weaker than TECH overall, but it accomplishes this by generating only a few hundred nodes in each search. The program can search 2 to 4 ply deeper than the programs which search all legal moves. It is discussed in more detail in chapters 2, 4, 5, and 6.

Pitrat's program [Pitrat77]. Unlike the above programs, this one does not play a full game of chess. It solves chess combinations when told how much material it should win, and is the most important example of the use of plans in chess. The program analyzes the original position using its knowledge of possible attacks to generate plans. (It only knows about certain types of attacks.) The plans are used to guide the search, and an analysis like the one in the original position is never done in the search. The program uses a very limited analysis to occasionally modify a plan during search, but only two modifications per plan are permitted. The range of problems which can be solved by this program is narrow, but it has no depth limit and can find combinations of 20 ply and deeper. Thousands of nodes are generated by the search when solving a hard problem. This program is discussed in more detail in chapters 2 and 6.

These programs use varying amounts of search and knowledge, and can be placed along a search-knowledge continuum. TECH would be very near the search end of the continuum; probably as near as possible for any program which plays reasonable chess. Programs like CHESS 4.5 would be close to TECH, though somewhat closer towards the knowledge end. Human grandmasters would be very close to the knowledge end of the continuum. Programs like CAPS and Pitrat's would be somewhere near the middle of the continuum. This research can be viewed as an attempt to create a program considerably closer to the knowledge end of the continuum than previous programs.

3) Programs using patterns and productions

The use of pattern-based knowledge lends itself readily to production rules. Whenever a pattern is matched, a corresponding action is taken which enables the system to act on the information obtained by matching the pattern. Simon and Chase suggest in [Chase73] that human masters exhibit chess skill by using their perceptual patterns as the conditions of production rules. Matching a pattern triggers the action part of its production rule.

While there is little hard evidence to support this, it is currently the most reasonable explanation of how humans use their patterns. In the research described here, each pattern becomes the condition of a production rule and is associated with some number of actions.

Productions have been studied in depth since Post introduced them in 1943 ([Post43]). There are many AI systems which use production rules, and they have varied goals and objectives. This research involves producing a program which is a performance-oriented expert in the domain of chess. Probably the best system using productions that can be compared to this research is MYCIN ([Shortliffe74],[Davis76]), another performance oriented domain-expert. This system is described very briefly below, stressing aspects which are useful for comparison.

The MYCIN system has many parts (some modules deal with acquiring new productions, for example), but the primary task is to give advice on the diagnosis of and therapy for infectious diseases. This involves using specific knowledge about the domain of infectious diseases and their therapies. The primary source of this knowledge in MYCIN is a knowledge base composed of about 400 production rules together with hundreds of definitions and simple facts. Each rule consists of a premise and an action. The premise contains Boolean combinations of clauses, each of which applies a predicate to an object in the system and tests its value. The action of a rule gives a conclusion that can be drawn if the premise is satisfied. Given a goal, MYCIN finds all rules whose actions bear on the goal. It then tries to evaluate the premises of these rules to see if any match. New goals may be set up to obtain information needed to match these premises. Thus the system attempts to chain itself through the rules backwards from the goal. This produces a depth first search which attempts to find a chain of reasoning that will satisfy the original goal.

Later chapters will compare the productions in this research to those in MYCIN, and will describe new techniques in reasoning with productions that must be developed to solve problems presented by chess but not addressed by MYCIN.

D) The Problem Domain of PARADISE

This research centers around development of PARADISE (PAttern Recognition Applied to Directing SEarch), a program written in MacLisp which finds the best move in tactically sharp middle game positions from the games of chess masters. The phrase "middle game" means the program's domain does not include castling, en passant captures, or pawn promotion. The phrase "tactically sharp" is meant to imply that success can be judged by the gain of a material rather than a positional advantage. The attacking side can generally win material with correct play, so the program uses different models for the offensive and defensive players during its search of the game tree. PARADISE is told which side is the offense and this never changes during analysis. The offensive player uses much more knowledge than the defensive player and finds the best move immediately in most cases. The defensive player tries obvious counter attacks and any defense that might possibly work, thus allowing a convincing proof of an offensive win. Important design decisions which have affected the development of PARADISE and its domain are mentioned below (without attempts to justify the decisions).

1 - Use knowledge to replace and support search.

A major thrust of this research is to investigate the extent to which knowledge can replace and support search in selecting a chess move and to delineate the issues involved. To reduce the amount of knowledge that must be encapsulated, the domain of the program has been limited to tactically sharp middle game positions. This is a large enough subfield of chess to retain all the advantages of the chess domain mentioned in section B of this chapter. Other subfields would also be good, especially certain endgames, but the complexity of the middle game requires extensive search. This provides a good testing ground for attempts to replace search with knowledge and to use knowledge to guide the search and communicate discoveries from one part of the tree to another.

2 - The program should achieve an expert level of performance.

To achieve expert performance, PARADISE encodes and uses patterns whose complexity is comparable to those which human masters seem to use. PARADISE exhibits expert performance on positions it has the knowledge to understand, and figure 1.1 shows that it can solve some problems that the best search based programs cannot.

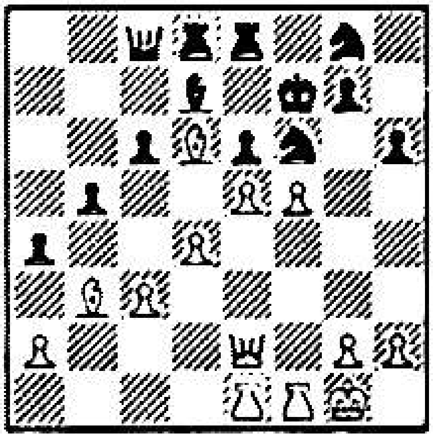


Figure 1.1
white to move

1	Q-R5ch	KxQ
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	

Probably the hardest problem PARADISE has solved is shown in figure 1.1. The winning combination was played by a former world champion in an actual game. White can mate by sacrificing his queen and playing nine additional moves (against black's line of longest resistance). Thus the search tree must go at least 19 ply deep to find this combination, with the move at ply 17 being neither a capture nor a check. This is considerably deeper than any other current chess program could probe in a reasonable amount of time, but PARADISE solves this problem after generating a search tree of 109 nodes, in 20 minutes of CPU time on a DEC KL-10. The program must use its knowledge expertly in order to correctly analyze the many variations of this mate in only 109 nodes. PARADISE was not groomed for this problem; it solved the problem after being developed on other problems.

3 - The knowledge base must be easy to modify.

A human master builds up his knowledge and ability incrementally. For a knowledge based program to achieve master level performance, it seems essential that the know'edge base be amenable to modifications and additions. This is one of the major design criteria of the program and PARADISE achieves this at the cost of inefficient execution.

4 - Proper use of knowledge is more important than speed of execution.

The goal is to build an expert knowledge base and to reason with it to discover plans and verify them with a small tree search. As long as this goal is reached, the amount of execution time used is not important. It is a design decision to sacrifice efficient execution in order to do things the "right" way. It is acceptable for PARADISE to spend 10 to 20 minutes of cpu time solving a problem, as long as its reasoning process performs expertly enough to justify the time. (It would not be acceptable to spend hours of cpu time on a problem.) When a desired level of performance is reached, the program could be speeded up considerably by switching to a more efficient representation at the cost of transparency and ease of modification.

6 - The performance of the program and its limits should be accurately measured.
In AI research it is important to get accurate insights into program performance and the origins of this performance, and to be able to communicate them to others. This is one reason chess is a good domain. To aid in evaluating performance, PARADISE has been tested on the first 100 positions in the book Win At Chess by Fred Reinfeld [Reinfeld58] which contains tactically sharp positions from master games. This selection of problems is representative of tactical problems of reasonable difficulty. Although intended to instruct humans, these problems have been used to test other chess programs (e.g., CAPS, TECH, and CHESS 4.4). Thus they provide a good domain for comparing the performance of PARADISE to that of other programs.

In developing the knowledge base for PARADISE, it must be shown that a hand-tailored piece of knowledge is not being written for each position. It is also desirable to get a handle on just how easy it is to modify the knowledge base. For these and other reasons, a developmental set of fifteen positions has been selected from among the first 100 in Win At Chess. The knowledge base of PARADISE has been developed by writing productions which solve these fifteen positions in a reasonable manner. Monitoring this development provides information on the ease of incremental additions and on how such additions affect program performance on problems it can already do. The 85 positions not in the developmental set were not considered during this development. The development process produced one version of the program, called PARADISE-0, which solves all fifteen problems in the developmental set reasonably.

PARADISE-0 was then tested on a test set of six positions picked at random from the remaining 85. The purpose of the test set is to measure the generality of the knowledge base by seeing how many problems PARADISE-0 could solve, and to measure the ease of modifiability by giving the program enough knowledge to solve the ones it could not do originally. The version of the program which solves both the developmental and test sets is called PARADISE. To give a better grasp of the generality of the program, PARADISE was then tested on all middle game positions in the first 100. The results of this development and testing are presented in detail in chapter 6. They show that PARADISE outperforms both TECH and CAPS on these 100 positions, while performing on a comparable level with CHESS 4.4. They also indicate that the knowledge base is easily modifiable and can be easily extended so that PARADISE can solve nearly all of these problems.

E) Overview of PARADISE

1) Introduction

PARADISE uses a knowledge base consisting of about 200 production rules to discover plans for the offense and to guide a small tree search which confirms that a particular plan is best. Every production has a pattern (i.e., a complex, interrelated set of features) as its condition (left-hand side). The patterns are built on each other, with more primitive patterns (e.g., a pattern which matches legal moves) being used as building blocks for more complex patterns (e.g., a pattern which matches a forking attack). Figure 1.3 shows an example of a production rule in PARADISE.

```
((DMP1)
(NEVER (EXISTS (SQ) (PATTERN MOBIL DMP1 SQ)))
(NEVER (EXISTS (P1) (PATTERN EXPRIS P1 DMP1)))
(ACTION ATTACK ((OTHER-COLOR DMP1) (LOCATION DMP1)) (THREAT (WIN DMP1)) (LIKELY 0)))
```

Figure 1.3
Production rule in PARADISE which recognizes trapped pieces

Figure 1.3 shows one of the simplest productions in PARADISE. It attempts to find an offensive plan which attacks a trapped defensive piece. The first three lines in figure 1.3

constitute the pattern or condition of the production rule while the fourth line constitutes the action. The first line specifies one variable that must be instantiated. Because of its name, the variable must be instantiated to a defensive piece that is not a pawn. The next two lines in the production actually instantiate the variable DMP1. They access two more primitive patterns that have already been matched, MOBIL and ENPRIS. The MOBIL pattern matches each legal move that can be made without losing the moving piece. If there are no squares which match MOBIL when DMP1 is the first argument, then DMP1 is trapped. Only in this case will there be any possible instantiations for DMP1 after the second line in figure 1.3 is matched. The third line prevents the pattern from matching if the trapped piece is already en prise. The idea is that in this case DMP1 should be captured outright rather than attacked. Thus this production matches only trapped pieces which are not en prise. The more primitive patterns such as MOBIL are discussed later in this section and in chapter 2. The language in which productions are written is described in detail in chapter 2.

Each production can be viewed as searching for all instances of its condition pattern in the position. For each instance found, the production may, as its action, post zero or more concepts in the data base for use by the system in its reasoning processes. A concept consists of a concept name, instantiations for a number of variables, and a list of reasons why the concept has been posted. In figure 1.3, the action of the production rule posts an ATTACK concept. This concept gives instantiations for two variables and tells the system that the opposite color of DMP1 would do well to attack the square which is DMP1's location. In addition one reason is given for suggesting this concept. This reason consists of attributes describing the threat of the concept and how likely it is to succeed. Concepts are discussed in more detail later in this section and in the next chapter.

The data base can be considered as a global blackboard where productions write concepts. These concepts interact through the execution of other productions to eventually suggest plans. Productions which know about attacking a square will eventually use the ATTACK concept posted by the production in figure 1.3 as a goal to possibly suggest other concepts or plans. How PARADISE organizes its productions to do this is described in chapter 2.

Plans are special instances of concepts. Instead of a number of variable instantiations, a plan has a tree structure which specifies relevent concepts for different possible positions that may develop as play proceeds from the current position. The concepts used in plans are the same as the concepts posted by productions. By using concepts specified in a plan it is following, PARADISE can quickly find the right move in a new position without repeating a whole analysis to discover the important concepts. Since plans are special instances of concepts, they also describe why they were suggested which is useful for determining which plan to search first. Plans are described in detail in the next chapter.

A search of the game tree (the major component of most chess programs) is used to show that one plan suggested by the pattern-based analysis is in fact the best. To offset the expense of the analysis, problems must be solved with small search trees. In fact, PARADISE can be viewed as shifting some of its search from the usual chess game tree to the problem of matching patterns. The search is "small" in the sense that the size of the search tree is on the same order of magnitude as a human master's search tree. (Tens of nodes, not thousands to hundreds of thousands as in many computer chess programs.)

PARADISE uses its pattern-oriented knowledge base in a variety of ways to significantly reduce the part of the game tree which needs to be searched. Search trees grow slowly enough that PARADISE is the first chess program which plays the middle game that does not place a depth limit or any other artificial effort limit on the search. This section gives a general overview of the structure of PARADISE showing the various ways in which the knowledge base is used. First the various uses of the knowledge base are enumerated, then flowcharts are presented which illustrate how these various system abilities interact in the search.

1 - Calculating primitives

PARADISE has functions which find primitive chess relationships in a position (e.g., which squares a piece attacks). PARADISE's knowledge base contains eighteen productions which use these functions to match low-level patterns (e.g., possible pins, possible discovered attacks, MOBIL, ENPRIS). These eighteen productions have no action parts associated with their conditions, so will hereafter be referred to as patterns. These patterns (along with the primitive functions) are then used by the rest of the productions as primitives to describe the chess position. The only function of these "primitive patterns" is to provide building blocks for the other productions to use. Both the primitive patterns and functions are described in detail in chapter 2.

Most of the time, these primitive patterns are matched whenever a new position is created. This is occasionally avoided when the plan being executed provides a concept which demands a move be played irregardless of features in the current position. The primitive patterns are relatively complex and matching them is expensive (about 4 seconds of cpu time on a KL-10 for an average position). They range from legal chess moves to calculating the safety of each piece as is done in MOBIL. The latter is a non-trivial calculation since it involves playing an exchange sequence of all pieces affecting the square in question (determining this sequence may require an analysis of the strengths of various pins). This calculation is described in detail in chapter 2. Although percentages vary from problem to problem, PARADISE on the average spends about half of its cpu time matching primitive patterns. In terms of time, this is therefore the most important use of the knowledge base.

2 - Static analysis

Given a new position, PARADISE does an expensive in-depth static analysis which uses, on the average, 12 seconds of cpu time on a KL-10 (although it may range from 1 to 40 seconds depending on the position). This analysis uses a large number of productions in the knowledge base to look for threats which may win material. It produces plans for the offense which should guide the search for many ply and should give information which can be used to stop the search at reasonable points. Much effort is invested to make sure that a plan has a reasonable chance of success before it is suggested. Many chess programs treat each newly created position in the same manner, performing the same static analysis on it. PARADISE cannot afford to do this because its pattern-based analysis is so expensive.

PARADISE avoids this behavior by using plans to guide the search for many ply. Through plans, PARADISE uses its knowledge to understand a new position on the basis of previously analyzed positions, so that a static analysis is only occasionally necessary on positions created during the search. The organization of the knowledge base and the representation of plans are described in chapter 2, and chapter 3 shows how a static analysis is done.

Since the static analysis usually discovers the important ideas in a position, the principal use of the search is to verify that a plan will work as expected. The search occasionally discovers an idea which is used to formulate the winning line, but discovery is usually done by the static analysis with the search usually acting as a verifier. Thus searching in PARADISE plays a different role than it does in the better chess playing programs (e.g., CHESS 4.7) where search is the primary discovery mechanism for generating the correct line of play.

3 - Producing plans

The knowledge base is used to produce plans during static analysis and at other times. Producing a plan is not as simple as suggesting a move, it involves generating a large amount of useful information. Plans must fulfill two important roles. The first responsibility of a plan is to guide the search so that static analyses can be avoided. This involves describing the principal lines of play which may ensue (by giving templates which match moves by the opponent), and providing relevant concepts for each one. The second responsibility involves producing an expectation or goal for the search. This is done by providing the reasons for suggestion of a plan (see chapter 2).

The search either proves that a plan succeeds or provides information for analyzing why the plan failed. One of the hardest problems the search faces is deciding what it means to "succeed", since there is almost always the possibility that a given line can be improved if more effort is expended looking at other alternatives. Through the reasons attached to each plan, the plans themselves provide much of the information the search uses to decide when to be satisfied that one plan will be better than any other. This information is also used by the search to decide which plans to search first, to determine when a plan is not working, and to decide if there is a plan somewhere in the tree which is more promising than the current one. How PARADISE determines "success" is described in chapter 4.

4 - Executing plans

The execution of a plan during the tree search requires using the knowledge base to analyze concepts specified in the plan. This often produces the right line of play without a static analysis. This use of the knowledge base could be viewed as a static analysis where the system's attention is focused on a particular concept or aspect of the position. Such a focused analysis can be done at a fraction of the cost of a full analysis which examines all aspects of the position.

5 - Generating defensive moves

PARADISE does static analyses and executes plans only for the offense. To prove that an offensive plan succeeds, the search must show a winning line for all reasonable defensive moves. Productions in the knowledge base are used by the search to generate all the reasonable defenses. These productions generally do not expend much effort to assure that a defensive move will work which means that suggestion of defensive moves is less expensive than the static analysis which suggests offensive plans. Defensive moves are suggested by looking for threats, by attempting to thwart an offensive plan, and by attempting to thwart lines the offensive has already used to win material. It is not the case that one subset of productions comprises the defensive model and another set the offensive model. The same productions look for threats for both offense and defense, and productions which thwart plans are also used by both offense and defense. (If PARADISE realizes a plan would work except for a threat by the defense, it employs productions to thwart the threat in order to develop an offensive plan.)

6 - Quiescence searches

When PARADISE finds that an executed plan has either succeeded or failed, it terminates the search and returns a range of values for the current position. One of the most important jobs of any chess program is to force a position to quiesce before evaluating it PARADISE uses a quiescence search to determine the value of a position when terminating although this search may return information which will cause the termination decision to be revoked. Productions in the knowledge base provide much of the knowledge used in PARADISE's quiescence search. Generating defensive moves for either side during the quiescence search involves using the same productions which generate defensive moves in the normal search, while generating offensive moves for either side involves using some of the productions used in a normal offensive static analysis as well as other productions not used in the static analysis. These latter productions are similar to productions used in the static analysis but they suggest only plans that are almost sure to work. Quiescence searches in PARADISE investigate not only captures but also forks, pins, multi-move mating sequences and other threats.

7 - Analysis of problem upon failure

When either the defense or offense has tried a move that failed, the search analyzes the information backed up with the refutation in an attempt to solve the problem (see chapter 4). The knowledge base is used in this analysis to suggest plans which thwart the opponent's refutation. In this way PARADISE constructs new plans using the information gained from searching an unsuccessful plan. The search also uses information from unsuccessful searches to avoid searching plans which have been suggested but cannot work. This avoids rediscovering the same refutation for each move at a node, something computer programs do frequently and human masters do very rarely. Using results of previous searches in these ways will be referred to as tree communication since information from various parts of the search tree is being communicated and used at other nodes in the tree. Tree communication is important in keeping the search tree small.

8 - Answering questions

The search often requests certain kinds of information about the current position. The knowledge base is used to provide such information. The most frequent requests ask if the position is quiescent and if there are any obviously winning moves in the position.

The above list describes eight general functions the knowledge base performs for PARADISE. The knowledge base, plans, concepts, and static analysis are described in chapters 2 and 3. This section will present the highest level modules in the search to show how all these things interact. The search has different models for the offense and defense as described in section D of this chapter. In order to "prove" a line really does win, an adequate line must be shown for every reasonable defense. The correct defense may be subtle so the search must be careful about skipping a defensive move. Thus the defensive model suggests any reasonable defense and spends very little time discriminating between defensive moves. It also tries obvious counterattacks. The offensive player, on the other hand, uses a large amount of knowledge, suggests many esoteric attacks, and invests much effort in making sure a plan is likely to work before suggesting it.

The flowchart in figure 1.4 shows how the search uses various sources for suggesting moves when the offense is on move. This routine is applied to a new position in the search after the defense has just made a move. The object is to quickly find the best move without doing a static analysis and possibly without matching the primitive patterns. The flowchart in figure 1.5 shows the principal subroutine used in figure 1.4. Many of the above mentioned uses of the knowledge base are brought to bear here. Whenever productions in the knowledge base are accessed, the initials "KB" appear in the flowcharts. The first time the knowledge base is accessed, the primitive patterns for the current position will be matched. The purpose of these flowcharts is to show how the various uses of the knowledge base interact. Chapters 2, 3, and 4 describe how all these mechanisms actually work.

OFFENSE SEARCH

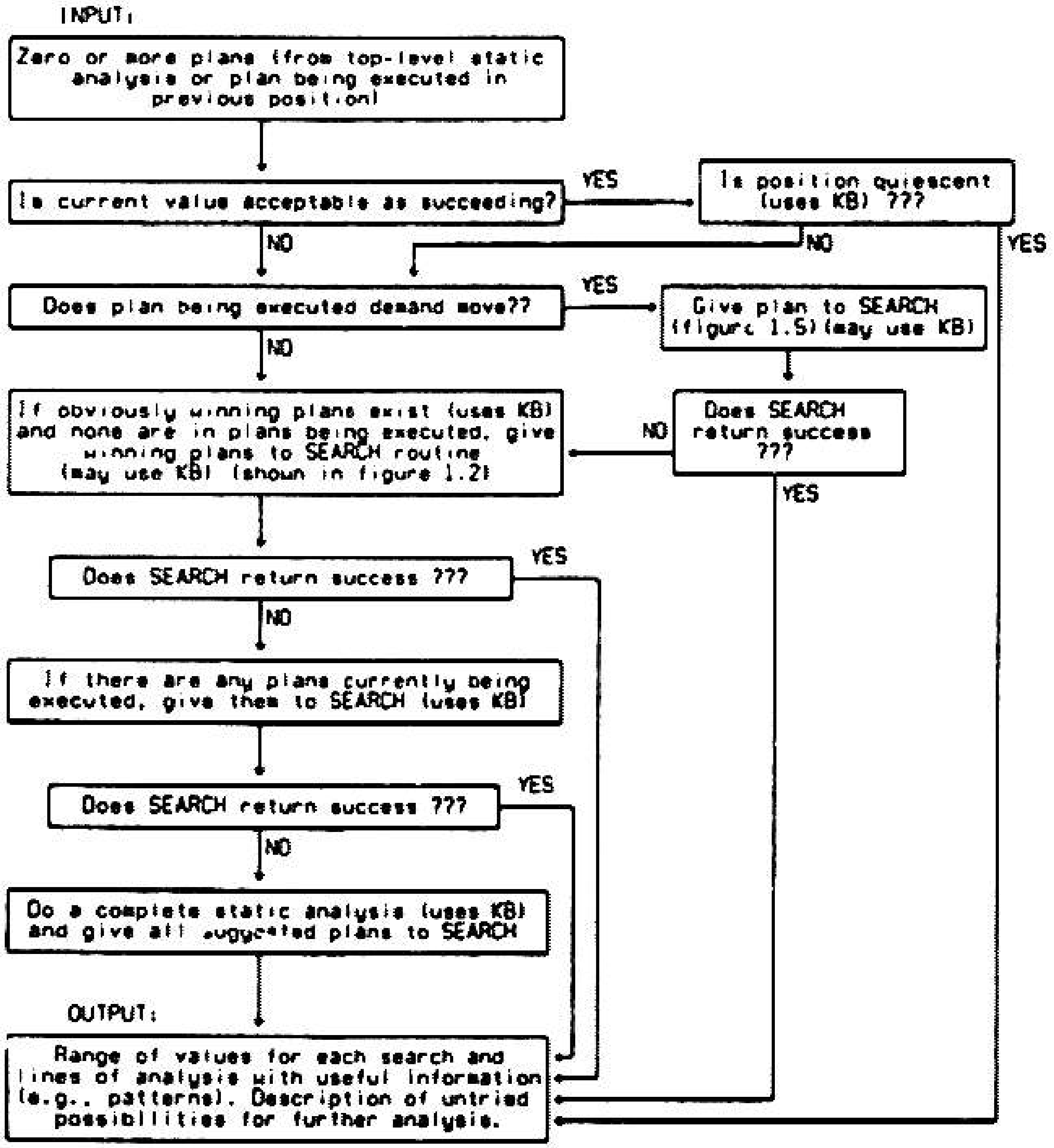

INPUT:
Zero or more plans (from top-level static analysis or plan being executed in previous position)
Is current value acceptable as succeeding?
YES
Is position quiescent (uses KB) ???
NO
YES
NO
Does plan being executed demand move??
YES
Give plan to SEARCH (figure 1.5) (may use KB)
NO
If obviously winning plans exist (uses KB) and none are in plans being executed, give winning plans to SEARCH routine (may use KB) (shown in figure 1.2)
NO
Does SEARCH return success ???
YES
Does SEARCH return success ???
YES
NO
If there are any plans currently being executed, give them to SEARCH (uses KB)
Does SEARCH return success ???
YES
NO
Do a complete static analysis (uses KB) and give all suggested plans to SEARCH
OUTPUT:
Range of values for each search and lines of analysis with useful information (e.g., patterns). Description of untried possibilities for further analysis.

Figure 1.4
Flowchart showing how search chooses moves for offense

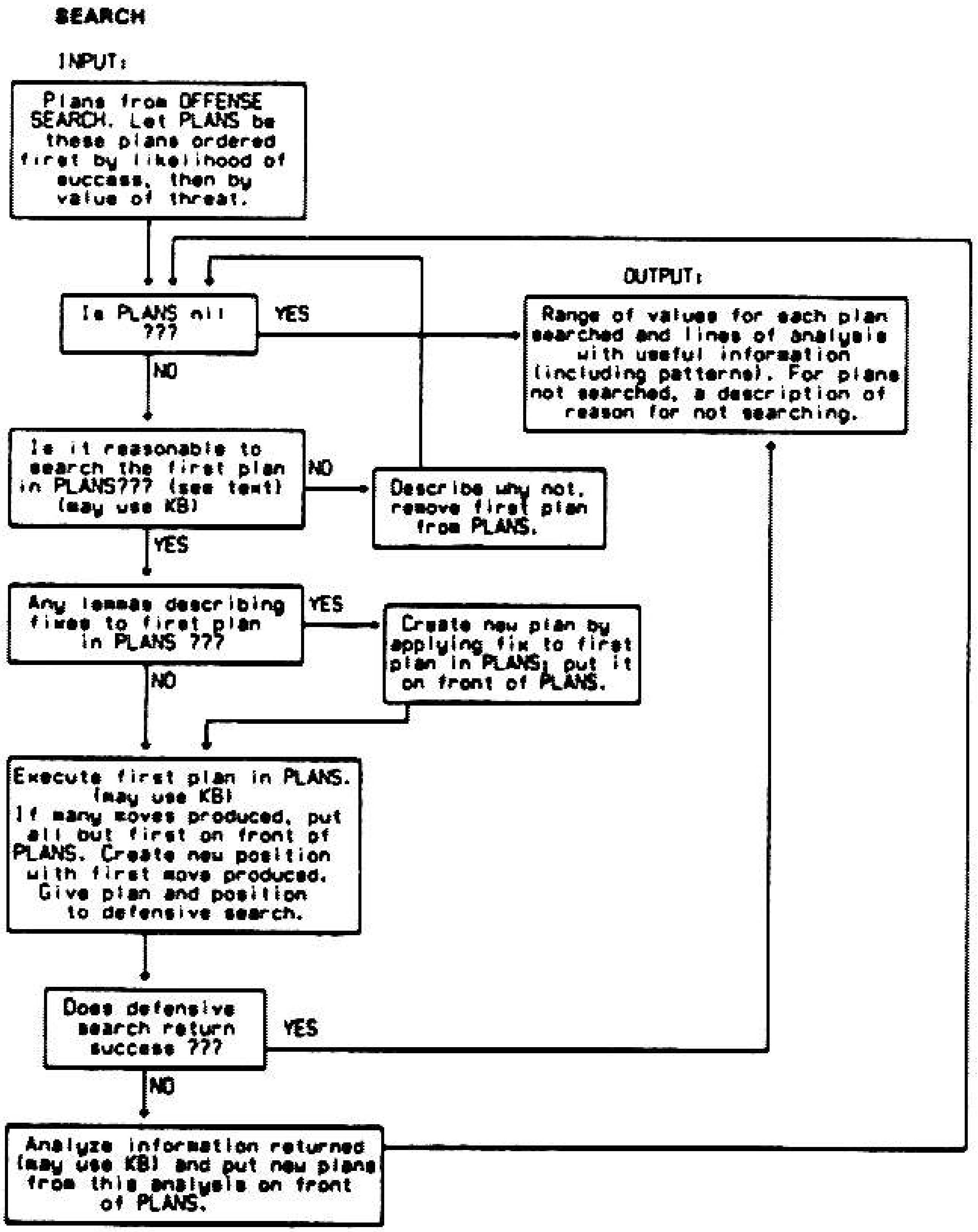

SEARCH
INPUT:
Plans from OFFENSE SEARCH. Let PLANS be these plans ordered first by likelihood of success, then by value of threat.
Is PLANS nil ???
YES
NO
OUTPUT:
Range of values for each plan searched and lines of analysis with useful information (including patterns). For plans not searched, a description of reason for not searching.
Is it reasonable to search the first plan in PLANS??? (see text) (may use KB)
NO
Describe why not, remove first plan from PLANS.
YES
Any lemmas describing fixes to first plan in PLANS ???
YES
Create new plan by applying fix to first plan in PLANS; put it on front of PLANS.
NO
Execute first plan in PLANS. (may use KB) If many moves produced, put all but first on front of PLANS. Create new position with first move produced. Give plan and position to defensive search.
Does defensive search return success ???
YES
NO
Analyze information returned (may use KB) and put new plans from this analysis on front of PLANS.

Figure 1.6
Flowchart of SEARCH routine

The offensive search portrayed in figure 1.4 distinguishes four separate categories of plans to be searched. First, it searches moves demanded by the plan being executed, then it looks for obviously winning plans and searches them, then it searches the plan being executed (which usually suggests moves rather than "demanding" them), and as a last resort it does a static analysis to find plans to search. As soon as a plan "succeeds" the process stops, but a description of all untried alternatives is included in the result. This facilitates returning to this position in an attempt to improve the successful result. The result also includes pattern information which other nodes will use in their analysis (tree communication).

Figure 1.5 depicts the SEARCH subroutine used in figure 1.4. The main point of this diagram is that each plan searched may expand into several moves which may all be searched. Regardless of which category the plan came from in figure 1.4, the SEARCH routine will execute the plan, attempt to apply previously discovered improvements which have been remembered, and analyze any failures in an attempt to fix the problem. All three of these processes may create new plans (and moves) from the original plan. Thus even though there are no obviously winning moves and the plan being executed misses the correct line, a static analysis can still be avoided by analyzing the failure of the given plan.

These flowcharts are too general to communicate how PARADISE actually works, but the phrase "Is it reasonable to search the first plan?" hides more detail than most. The following is a partial list of the criteria PARADISE checks to decide that a plan is not reasonable to search:

> -Could this plan have been used earlier and is it no more promising now than it was then?
>
> -Can it be determined by analyzing the refutation of a previous move that this plan cannot work?
>
> -Is the likelihood of this plan succeeding too small? This can be a local decision involving this plan alone, or a more global decision combining this plan with the plans which have already been attempted since the original position.

-Is the value threatened by this plan unacceptable?

-Is the value of this position or the value threatened by this plan outside the alpha-beta range of contention?

-Is there a more promising plan at some other node in the tree?

-Does this plan cause a repetition of position?

This section provides a general description of the way PARADISE solves chess problems. To summarize, an expensive static analysis produces plans which guide the search for many ply (thus avoiding repeated static analyses), and pattern-oriented information provides tree communication which prunes the search tree. Many problems must be solved in writing a program which fits the description given here, and the next two sections describe them. Section 2 describes problems encountered while using the knowledge base to do analysis, and section 3 describes ways to use knowledge to control the growth of the search tree.

After this, chapter 2 discusses in detail the organization of PARADISE's knowledge base, its productions, and its plans. The productions and plans are compared to those in other systems. An example problem is presented which shows how plans are used. Chapter 3 presents an example static analysis in some detail showing many of the productions used in the analysis. Chapter 4 describes PARADISE's tree search. All the mechanisms for controlling the search are described in detail, and comparisons are made with other systems. Chapter 5 presents traces of PARADISE solving three problems, showing how the mechanisms described in chapter 4 actually work. A reader wishing more examples may wish to skim chapter 5 early. Chapter 6 presents the results of testing that has been carried out to analyze PARADISE's performance. A comparison is made to the performance of other systems. Chapter 7 summarizes the contributions of this work, describes directions for future research, and dicusses in detail some of the issues this work has addressed.

2) Problems dealt with in static analysis

This section describes some of the major problems that must be addressed in building a knowledge based chess program like PARADISE. Since the problems of using a large knowledge base while doing expert reasoning are different from the problems of using knowledge to control the growth of a search tree, these two categories have been separated. The former is discussed in this section and the latter in the next section. The manner in which PARADISE deals with each of these problems in briefly described below. Detailed descriptions are presented in the following chapters.

1 - Complexity of pattern matching

Matching a pattern can be arbitrarily complex in PARADISE. It can involve running expensive procedures including searching (e.g., searching for pieces to fulfill certain functions). By contrast, many "expert" production systems in AI, for example MYCIN, have very straightforward ..ching processes which involve testing the values of variables and simple predicates. The matching process must be controlled fairly strictly in PARADISE. Productions must be written in such a way that searching can be minimized during the matching process. Some productions cannot be reasonably included in the knowledge base because of the cost of matching them. For example, suppose a production suggests the queen sacrifice of Q-R5ch in figure 1.1 but has enough knowledge to know that the same sacrifice would not work in figure 1.2. Such a production would almost certainly be too costly to match. There is always a tradeoff between search and knowledge which here becomes a tradeoff between search in the pattern matching and search in the game tree. The developer of the knowledge base must for each production decide if its utility outweighs the cost of matching it or if the matching cost is so great that the features in the production would be better discovered by searching the game tree.

2 - Large number of productions

The knowledge base in PARADISE contains about 200 productions. Since the search may visit hundreds of positions, it would be unreasonable to match all the productions for each position visited. The system must be able to quickly focus itself on the relevant productions in a given position. As described in section 1, PARADISE uses plans to avoid static analyses and do focussed analyses which access only a few productions in each position. In addition, the knowledge base is organized so that only relevant productions are attempted during a full static analysis. The details are provided in chapters 2 and 3.

3 - A solution must be created from concepts

To understand a chess position, a successful line of play must be found. Human masters understand concepts which cover many ply in order to find the best move. For example, a chess master might move his knight to decoy an enemy pawn which will then permit a smothered mate on the eighth rank by the rook. A knowledge based program should be able to come up with that idea, and therefore must have the ability to reason with concepts that are higher level than the legal moves in a chess position (e.g., the concept of a decoy or the concept of a piece being able to mate from the back rank). The following position from Berliner's thesis illustrates this point in a more concrete manner.

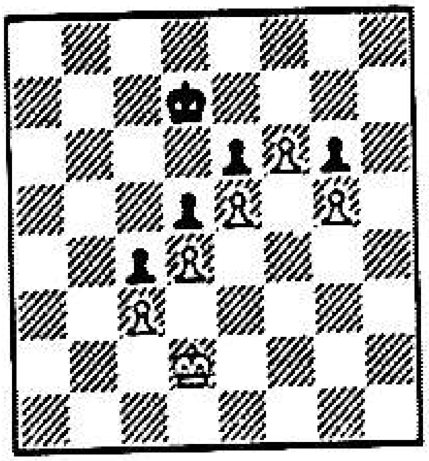


Figure 1.6
white to move

In the above position, most computer chess programs would search the game tree as deep as their effort limit allowed and decide to play K-K3 which centralizes the king. They cannot search deep enough to discover that white can win since white's advantage would take more than 20 ply to show up in most evaluation functions used by computer programs. The point is that the solver of the above problem uses conceptual reasoning to solve it, and the concepts must be higher level than the legal moves in a chess position. A human master would recognize the following features (among others) in the above position:

- The black king must stay near his KB1 to prevent white's king bishop pawn from queening.

- The pawns form a blockade which the kings can't penetrate.

- The kings can bypass this blockade only on the queen rook file.

- If white's king were nearby, it could attack black's pawn chain at its base (the king pawn).

For a knowledge based program to solve this problem, it must be able to express concepts at the level of conceptualization used above. For example, the system must be able to express the fact that the pawns blockade the kings from the queen knight file through the king rook file. The system must combine and reason with these concepts in unforseen ways. In the above example, the blockading concept would be used to develop the plan of moving the white king to the queen rook file in order to bypass the blockade, but if both black and white had pawns on their QN4 then the blockading concept would be used to decide that neither king can invade the other's territory. Expressing and using concepts is a major problem for a knowledge based chess program.

To clarify the problems involved in using concepts to reason about chess, a comparison to a well-known "expert" system such as MYCIN is helpful. The following phrase is the action part of a typical MYCIN production rule (from [Davis76]):

then there is suggestive evidence (.7) that the identity of the organism is bacteroides

Such an action effectively adds to the plausibility score for some possible answer. This presupposes that any one writer of all the rules knows every possible solution since such rules will never create a diagnosis that has not been mentioned in the action part of some rule.

Using actions similar to those used in MYCIN rules would be similar to a chess system where the action part of each rule was to add to the plausibility score of either 1) one or more of the legal moves in the position, or 2) some prespecified object which could lead (through other rules) to recommending a legal move. Such an approach is not satisfactory for the type of chess reasoning needed in the above problem. Concepts at a higher level than legal moves must be used in this reasoning, and objects cannot be prespecified (e.g., the extent of the blockade must be created dynamically). Thus, PARADISE must reason by linking concepts to create plans instead of filling in prespecified slots.

PARADISE must develop concepts and reason with them to create plans. Probabilistic reasoning may be useful in chess (e.g., to decide if one position is better than another) but PARADISE does not do probabilistic reasoning in its restricted domain. In most current expert production systems (such as MYCIN), the action parts of the rules are in some sense part of the solution or a pointer to something which can produce a solution. In chess the system must discover lines of play that were not implicit in the knowledge base. Thus the word "create" is used. MYCIN has a small solution set (120 organisms) and can mention each solution in its rules, while a chess program must discover an enormous number of lines using a set of productions which is much smaller in size (by an order of magnitude or more) than the number of plans it must recognize. This can be seen by looking at the actual production rules: MYCIN's actions mention the names of objects while PARADISE's actions (see next chapter) involve many variables whose instantiation is not determined until execution of the action.

When a production in PARADISE matches, it may have an action part which posts various concepts in the data base. PARADISE is able to express and use concepts by dividing its knowledge base into various knowledge sources. Each knowledge source provides the knowledge necessary to understand and reason about a certain type of concept. Knowledge sources and the processes in which they are involved are described in detail in the next chapter.

4 - Recovering from wrong deductions

After formulating a plan, a chess master uses a searching process to verify that his idea is the correct solution. This process many times involves changing the solution, discovering a new solution, or communicating newly discovered ideas as the basis for a re-analysis. When a chess master matches a pattern in may suggest a poor idea in this position, although the same pattern may suggest the correct idea in another position. He realizes this and has mechanisms for recovering from poor ideas. Similarly, a program for playing master chess from a knowledge base must be able to correct itself when it has wrong ideas. Search must almost certainly be used for making some corrections while other techniques will be needed for dealing with wrong ideas in static analysis.

PARADISE is wary of the concepts that have been posted by productions and does not treat them as facts that have been deduced. Each concept includes a description of why it was suggested. Using this information, other productions can decide later if the concept still looks right in light of what is known at that time. Chapter 2 describes how this is done. Most production systems do not address this problem. For example, when MYCIN matches a production, any assertions posted in the data base are treated as facts by the system (mistakes are corrected only when human expert notices them and changes the productions). PARADISE also uses search to find mistakes and provide information for correcting itself (this is described in chapter 4).

6 - Controlling the growth of the search tree

The ways PARADISE uses knowledge to control the growth of the search tree are described in the next section. Searching is mentioned here because many mechanisms for controlling tree growth require information which must be provided by the static analysis. For example, the static analysis must provide information which can be used to determine which plan to search first, and a description of what a plan should accomplish so the search can decide when to terminate. Chapter 2 describes how PARADISE does this.

3) Controlling the Search

One central idea in this research is that the patterns in the knowledge base can be used not only to analyze the initial state but also to guide the search and pass information. Some of the methods which PARADISE uses to control the search and communicate information are listed below. They have been divided into four categories on the basis of direction of information flow. The search tree should be thought of as having the initial position at its root and branching downward by making legal chess moves. Thus "down the tree" means going from one position to another which has been produced from the first by making a sequence of legal moves.

1 - Communicating knowledge down the tree (plans)

Most chess programs treat each position in the tree in the same manner, analyzing each one as a new problem. PARADISE avoids continual re-analysis by having a plan for what it is doing. Through plans, new positions can be understood on the basis of the analysis of a previous position. The plan language is more sophisticated than anything used before in computer chess. A plan will often provide a concept which accesses the correct group of productions at a new node, thus avoiding a complete analysis. Plans include descriptions of why they have been suggested. This information is used (at new positions down the tree) to quickly reject plans which are not working and to establish expectations which help decide when a plan has succeeded.

2 - Communicating knowledge up the tree

Patterns recognized deep in the tree can be communicated to increase understanding at nodes which are ancestors of these deeper nodes. Whenever PARADISE searches a move, the whole search tree is returned and each node in the tree may contain pattern matches from nodes lower in the tree. PARADISE uses such information to quickly get itself off a wrong track and onto the right one. To get on the right track, it uses productions in the knowledge base to analyze the information returned in order to discover problems and propose solutions. If the program finds the solution in this manner, it posts a lemma specifying narrow conditions under which this new solution should be used. This lemma is used at other nodes to avoid making the same mistake. PARADISE gets off a wrong track by using a causality facility similar to that in [Berliner74]. This facility rejects many bad moves by analyzing the trees of previous moves and their communicated patterns to determine effects of a proposed move. Berliner's search returned only bit masks for his causality facility to use (they represented pieces which moved, destination squares, squares moved over, names of targets, and squares in attacking lines). PARADISE has small trees so can afford to return the whole tree with associated patterns, thus providing much more information for the causality facility.

3 - Using local knowledge for controlling tree growth

The information in plans provides an understanding of what each move is threatening to do and how it intends to do it. This information is available at the current node in the tree (although it may have been passed down from a higher node), and can be used in many ways to prune the search tree and reorder moves at a node. Plans which threaten to achieve less than something already achieved are not searched. Plans which do not threaten to achieve the minimum expectation for success are not searched. Plans more likely to work in the current position are searched first. Plans which are extremely unlikely to succeed are not searched. Such forward pruning techniques have been tried in previous computer chess programs and have usually resulted in the program eliminating good moves and playing worse chess. In PARADISE these techniques prune the tree considerably without significantly impairing performance. This is due to the more accurate analysis of what a move threatens and how likely it is to succeed.

4 - Using global knowledge for controlling tree growth

Many times a poor line can only be pruned by viewing it in the global context of the whole tree. PARADISE has a number of search control mechanisms which require information from other nodes, some of which are described here. If a plan at the current node could have been employed earlier along the line leading to the current node and is no more plausible now, then PARADISE refuses to search it. In a position with many threats that do not affect each other, this keeps the system from attempting to string the threats together in every possible order.

Another example of using global knowledge is the restrictions placed on concatenating plans. The correct line in a chess position cannot always be seen with static analysis, so plans must be concatenated at times to build the correct line out of smaller plans. If PARADISE has decided to search a plan which is not likely to succeed, then it will not concatenate it with other plans which are not likely to succeed. PARADISE also uses global information to implement its best first search. Using information about unsearched plans elsewhere in the tree, the program will temporarily abandon lines when there are more promising ones elsewhere. Most of these techniques have not been used successfully in previous chess programs. PARADISE achieves a greater degree of success because it has the necessary knowledge to determine how likely a plan is to succeed and whether a plan is more plausible now than it was earlier.

CHAPTER II

The Representation of Pattern-oriented Knowledge and Plans

A) Reasoning with concepts: Knowledge Sources in PARADISE

1) Introduction

The idea of recognizing patterns in chess positions has been around for a long time (DeGroot did his work in the 1930's), yet no program makes extensive use of patterns. One of the major problems in using patterns is knowing what to do once you've recognized a pattern. As was demonstrated in chapter 1, it is not enough to just add some points to a "goodness score" for some move. A pattern must be able to trigger the formulation of a concept that the system can reason with. When a pattern match indicates a weak king side pawn structure, for instance, the system must understand what a weak king side pawn structure is and have some idea how to attack or defend it.

PARADISE combines patterns with actions that post concepts to produce production rules where the condition of each rule is the pattern and the action of each rule is the posting of concepts in the global data base. As the example in chapter 1 shows, a production rule in PARADISE is expressed as a list of variables to be instantiated and a list of expressions which are matched to instantiate these variables and to perform actions. Each expression in a production must match for the production to match. The last zero or more expressions may correspond to actions. The expressions before the "action" expressions constitute the pattern. Action expressions always match, so the question of whether a pattern matches or not is equivalent to asking if the production containing the pattern matches. These expressions are written in the Production-Language which is described in section B of this chapter.

Only by posting concepts can productions accomplish their purpose of producing a plan which describes a line of play. Before discussing productions in more detail, the structure and use of concepts will be discussed. A concept consists of a concept name, instantiations for a number of variables, and a list of reasons why the concept has been posted. Each concept is either a final product of the analysis (e.g., a plan or an answer to a question such as "is this position quiescent?"), or a goal that will be used by other productions. Most concepts posted by PARADISE are in the latter category.

Each concept that will be used as a goal has the name of a Knowledge Source as its concept name. For example, there is an ATTACK Knowledge Source in PARADISE to which the production in figure 1.3 refers to when posting its concept. The term "Knowledge Source" has been used many times in the AI literature, usually with a different definition in each case. In this thesis the term Knowledge Source is a proper name denoting a particular construct. Thus the meaning is not exactly the same as in previous uses of this term.

PARADISE organizes the productions in its knowledge base into Knowledge Sources (hereafter referred to as KSes). Each KS contains a number of productions which attempt to solve the same problem. The problems pursued by most KSes can readily be communicated to and understood by humans since they usually correspond to human concepts. This helps make the system's behavior transparent and explainable. Each KS is said to "know about" an abstract concept which is the problem the KS is trying to solve. The word "concept" is used since most such problems can readily be expressed as concepts which humans can understand. The word "abstract" is used to distinguish these more abstract correspondences to human concepts from the specific concepts posted by PARADISE in its data base. Most concepts posted by PARADISE are used as goals by KSes which know about abstract concepts. When the production in figure 1.3 matches, it posts a concept which corresponds to the ATTACK KS (i.e., the concept has ATTACK as its concept name). This concept is at some point used as a goal by the ATTACK KS which knows about the abstract concept of one side attacking a particular square.

In its simplest form, a KS is a group of productions which knows about some abstract concept, and a list of variables which any concept used as a goal must instantiate. The structure of a KS is described in more detail in part 3 of this section and the structure of a concept is described in part 4 of this section. There are currently twenty-eight KSes in PARADISE and they are described in detail in chapter 3. The next part of this section describes how these particular KSes were formulated during PARADISE's development.

2) Creating Knowledge Sources

There are at least two situations which indicate that PARADISE needs a new KS. One is when many productions are recognizing similar patterns, and a second is when a new abstract concept is needed to express a desirable plan. (Plans use the same concepts that are posted by productions.) This section gives an example of each situation.

When many productions are recognizing similar patterns in a position, some generalization of these patterns is a candidate for an abstract concept. This means a KS must be written which knows about the abstract concept. For example, PARADISE has a SAFE KS which knows about the abstract concept of making a square safe for a piece to land on. There are many reasons for wanting to make a square safe for a piece. The piece could fork two pieces from that square, or pin one piece to another, or mate the opposing king, or trap a piece, or any of a number of other things. There are also many ways to attempt to make a square safe for a piece. One can block the line of an opponent's piece which bears on the square or capture such a piece. One could move a friendly piece to provide additional support or make a threat somewhere else which can only be answered by an opponent's piece bearing on the square, thus decoying support from the square. Again, there are a number of other tactics.

If the first productions ever written recognized forks, there might be a production which recognizes forks from a square that can be made safe by blocking an opponent's protecting piece and another production which recognizes forks from a square which can be made safe by decoying an opponent's piece. Later, productions which recognize mates may be written which again recognize possibilities of blocking and decoying opponent's pieces to make the mating square safe. It is clear that each threat cannot be combined with each possible way of making a square safe since a huge number of productions would result where similar "safe-making" patterns are matched in many productions. When a production recognizes that a piece can make a threat from a square but cannot land safely on that square, it must post a concept saying that and then let productions which know about the abstract concept of making a square safe for a piece solve the problem (using the posted concept as a goal). Thus the knowledge is only encoded once and all recognized threats can use the same productions to make a square safe for a piece.

It should be noted that there must be a considerable amount of communication between the threat-recognizing pattern which expresses such a concept, and the productions which use the concept as a goal. Only the threat-recognizing production knows what the line of play will be once the critical square is made safe. It also knows what such a move threatens, and how likely such a move will be to succeed. This information is all needed later. For example, we would not want to decoy an enemy queen to make a square safe if the sole purpose of moving to the square was to attack this queen. Such knowledge must therefore be communicated in the concept itself when it is being posted. Part 4 of this section describes how concepts express this information.

New KSes may also be created when a new abstract concept is needed to express a plan. Suppose a forking pattern has matched and the system develops a plan to move a rook to a square where it can simultaneously attack an undefended knight and an undefended queen. This plan is then communicated down the tree so that after the opponent's reply, the system will know it should attempt to capture the queen or the knight. This avoids a complete reanalysis of the new position as most current chess programs would do. To communicate this plan the system must understand the abstract concept of safely capturing a piece. It is not enough to communicate that the rook should move to a particular square since the queen or knight may no longer be there. Likewise, it is not enough to communicate that the rook should capture one of these pieces if it can, since the queen may have moved to protect the knight. It is necessary to understand that the queen or knight should be captured without losing material. PARADISE has a SAFECAPTURE KS which understands this abstract concept, so such a plan would use a SAFECAPTURE concept. This abstract concept is closely related to the abstract concept of making a square safe for a piece. In fact, productions in the SAFECAPTURE KS will often post SAFE concepts, thus using the same abstract concept in analysis and in plan expression and execution.

3) Organization of Productions into Knowledge Sources

PARADISE achieves the ability to express and use concepts primarily by organizing its productions into KSes. In its simplest form, a KS is a group of productions which knows about some abstract concept, and a list of variables which any concept used as a goal must instantiate. The knowledge base currently has twenty-eight KSes, each containing productions that know about a particular abstract concept. If one production knows about more than one abstract concept, it may be in more than one KS. When a concept in the data base has a corresponding KS, PARADISE will eventually treat that concept as a subgoal and use the corresponding KS to solve the subgoal in an attempt to produce a plan. (This may be done by posting other concepts.) There may also be concepts which have no corresponding KS and are not treated as subgoals (e.g., plans may be considered as special instances of a concept, and there may be concepts which describe a position as non-quiescent in answer to a query). Such concepts are inspected by patterns attempting to match and by the searching routines. When a concept corresponds to a KS, it will sometimes be referred to as a goal or subgoal. Thus every goal is a concept but some concepts may never be used (or referred to) as goals.

A more detailed look at the structure of a KS is given in figure 2.0. The KS named KSNAME is shown. A concept corresponding to KSNAME must instantiate arg1 through argn. The condition is a pattern that must match before the KS is executed. The leftmost box in the diagram contains the productions of KSNAME, while the other boxes are KSes themselves, called sub-KSes. The productions in a KS are ordered. Each time a KS is executed with a concept as a goal, each production is tried in the given order. The KS is finished after one pass through the productions. Some productions have no action, but the fact that a pattern matches is inserted in the data base whether the pattern has an associated action or not. Other productions in the KS may then test if this pattern matched.

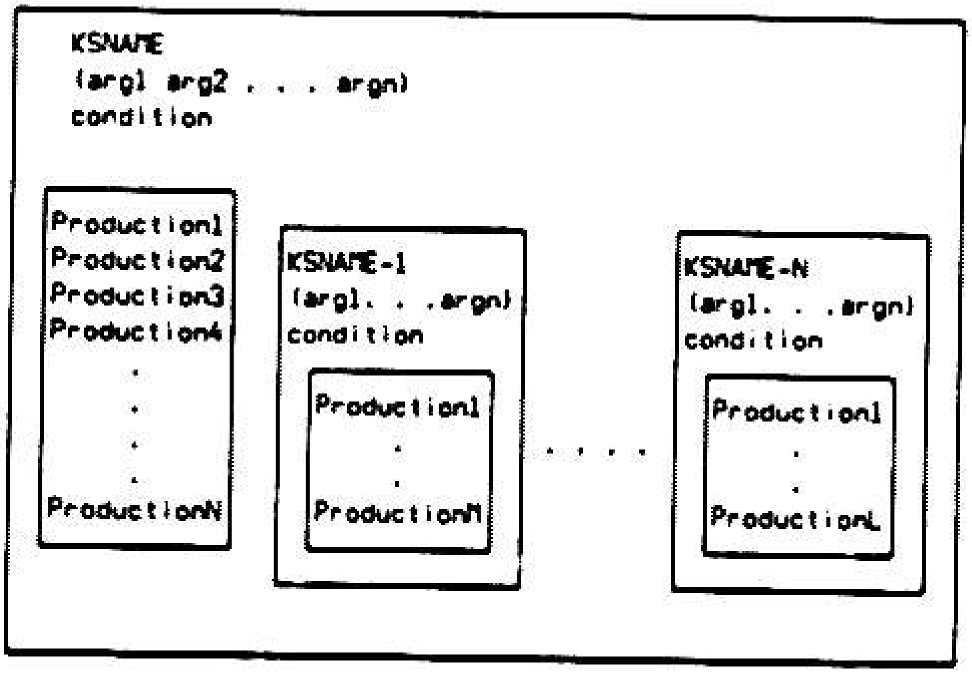


Figure 2.0
Structure of a Knowledge Source

The ordering in KSes is important only because some productions look for matches of other productions which must therefore be matched first. Thus an appropriate ordering can be determined syntactically from the rules. Productions may contain "return" expressions (described in section B) which cause termination of the execution of a KS with that production. This relies on the ordering but is only an efficiency consideration, since "returns" are used only when a production knows that no other production in the KS can match. (The same result would be produced in the "returns" were removed.)

When there are no sub-KSes (most KSes have none), all patterns that matched in a KS are removed from the data base after the KS has been executed for a particular concept. This keeps the size of the data base reasonable and prevents confusion between executions of different concepts. (Executing a concept means using a concept as a goal for its corresponding KS.) One exception is the PRIMITIVE KS, which contains the 18 patterns used as primitives in all other productions. This KS is executed only once, before any other KS, and its patterns are not deleted. Except for the PRIMITIVE KS, a KS's only effect on the system is the concepts it has posted in the data base (since the pattern matches are lost).

Sub-KSes provide a way for a complex KS to avoid the problems caused by having the matched patterns disappear after evaluation of a particular goal. When a KS has sub-KSes, the productions in the KS are matched for all concepts which correspond to the KS. All these matches are kept in the data base. Then each sub-KS is executed in turn, analyzing each concept which corresponds to KSNAME as well as each concept which corresponds to the sub-KS itself. These executions are normal, with pattern matches being deleted after analysis of each particular goal. (As the diagram implies, sub-KSes may not have sub-KSes themselves.) The pattern matches of the original KS, however, are not deleted until all sub-KSes are finished executing. Thus the productions in the sub-KSes have access to pattern matches for all concepts corresponding to KSNAME.

Sub-KSes can also be used for convenient shorthand notations of some KSes. For example, if one abstract concept involves the use of two other abstract concepts, the two can be written as sub-KSes of the original. Thus the original KS may have no productions of its own. Without sub-KSes the original KS would have a list of productions which was the concatenation of the two lists from the other KSes. The conditions on the sub-KSes mean that during execution all the productions in one of the sub-KSes may be skipped while the productions in the other sub-KS may still be executed. PARADISE has no other way to affect the control flow in this manner since the sub-KS structure has proved satisfactory.

Each KS expects a certain amount of information in a concept which it uses as a goal. At the very least, it expects an instantiation of its arguments. For example, the SAFE KS (for making a square safe for a piece) expects the particular square and piece to be instantiated. Among other things, the SAFE KS also expects some information on the plan to be followed once the piece gets to the given square. The representation of such information is discussed in the next section which describes the structure of concepts. (How productions in a KS accesses this information is described in section B of this chapter.)

4) Structure of a Concept in the Data Base

The context in which productions in PARADISE post concepts in the data base and how KSes then use the information in these concepts as subgoals to eventually produce a plan of action has been described. When a new KS is executed, the patterns which matched in previous KSes can no longer be accessed. (Frequently matched patterns are in the PRIMITIVE KS which can still be accessed.) Thus concepts in the data base must contain all the information that will be needed to execute other KSes and to guide the search. This can be a considerable amount of information, as illustrated below. As described in chapter 1, a concept consists of a concept name, instantiations for a number of variables, and a list of reasons why the concept has been posted. This list of reasons contains the information needed later.

To demonstrate the structure of a concept in PARADISE and the various requirements on accessing this structure, let us look at a particular concept the system would have while doing a static analysis of the position in figure 2.1.

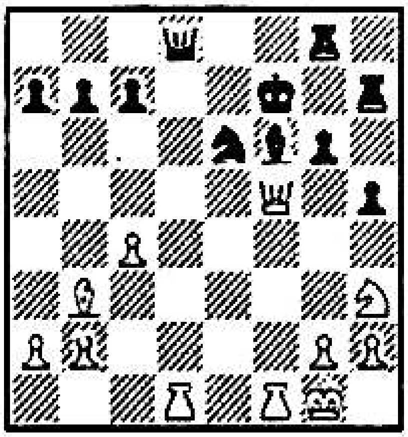


Figure 2.1
white to move

In this position PARADISE has a SAFE concept (among many others) for making the square Q7 safe for the white rook. (Currently the rook would be captured if it moved to Q7.) This concept has been suggested by two different productions. One recognizes the rook could skewer the black king to the rook on R7 and produces the following concept:

```
((SAFE WR Q7) (PLAN (SAFEMOVE WR Q7) (((BK NIL) (SAFECAPTURE WR BR))
                  ((ANYBUT BK) (SAFECAPTURE WR BR))))
        (THREAT (FORK WR BK BR)) (LIKELY 0))
```

SAFE is the concept name and WR and Q7 are instantiations for variables in the SAFE KS. PARADISE stores information about concepts on a list of attribute-value pairs (or property list), called an attribute list. In the example above, the PLAN attribute specifies the plan to be pursued after Q7 is made safe. (This is an expression in the plan-language which is described in section B of this chapter.) The THREAT attribute specifies what the rook threatens to gain on Q7. (This is an expression in the threat-language, described in section C of this chapter.) The LIKELY attribute describes how likely the concept is to succeed and is also described in detail later in this chapter.

Another production recognizes that the rook could fork black's king and queen from Q7 in such a way that any move by the king would enable the queen to be captured with check. The action of this production produces the following SAFE concept:

```
((SAFE WR Q7) (PLAN (CHECKMOVE WR Q7) (BK NIL) (SAFECAPTURE WR BQ))
        (THREAT (EXCH WR BQ)) (LIKELY 0))
```

This concept has the same concept name as the first one, the same variable instantiations, but a different reason (attribute list). PARADISE now faces the problem that in some sense the above two concepts are two distinct concepts but in another sense, there is just the one concept of making Q7 safe for the rook. (Frequently there will be many reasons for posting a concept.) One way to make Q7 safe for the rook would be B-R4 (protection by a friendly piece). In this case, the above two concepts can be treated as just one concept since this tactic will make Q7 safe no matter what the PLAN or THREAT is. In fact, most productions trying to make Q7 safe will not care what the THREAT is since this is only used when considering sacrifices and when comparing an opponent's threat to our own. Since many productions do not care about the values in the attribute lists and concepts frequently have many different reasons, it is imperative to treat all concepts with the same name and variable instantiations (though different attribute lists) as just one concept. Otherwise, many productions would needlessly be executed repeatedly on different versions of the same goal.

An obvious idea is to combine the values of the attributes and make one concept from the two above. For example, keep the most threatening THREAT value and combine the

PLAN values to form a plan with a branch so that either plan may be used. This would lose the information concerning which plan goes with which threat. But, as the following example illustrates, this information may be needed later (which means the productions may need to treat each of the original expressions as separate concepts). Let's consider making Q7 safe for the rook by playing QxB. Black can only blunt this threat by QxQ and then the rook can safely move to Q7 (enemy protection has been decoyed). QxB is a sacrifice and since white does not plan to recapture the black queen, he considers himself to be sacrificing queen for bishop. Such a sacrifice would only be made if moving the rook to Q7 threatens to win more than the value which was sacrificed. The threat to win the queen (which hopefully will be captured with check) meets this criteria but the threat to win the rook does not. However, if the sacrificial move decoys the queen away from Q7, the rook can no longer carry out its plan to win the queen after moving to Q7. The decoy should be rejected since it is decoying the very piece it wants to attack. (Note that the plan to win the rook would pass this criteria since the rook is not being decoyed.) This rejection requires knowing that the threat of winning the queen is only associated with the plan of capturing the queen or king. This decoy could not be rejected if the higher threat value of the queen-attacking plan had been associated with the plan value of the rook-attacking plan (as in a combination of attribute values).

The following observations may help clarify this example. Note that if we could decoy the queen by sacrificing a pawn, then the queen-attacking attribute list would still reject the move since the queen is being decoyed but the other attribute list would approve it since winning the rook is more threatening than the loss of the pawn. In fact the best move in this position is probably N-N5. This is a decoy which gives up a knight to make Q7 safe (after N-N5, NxN white's queen bears directly on Q7 and R-Q7 can be played safely). Both the queen-attack and the rook-attack threaten more than the loss of the knight, and neither plan is destroyed by decoying the black knight. (Note that it is important to check if the decoy destroys the plan since the system would otherwise suggest ridiculous moves like R-Q4 to decoy the black knight.)

For these reasons, all concepts in PARADISE with the same name and variable instantiations are combined into one concept, but the attribute list of this one concept is replaced by a list of attribute lists, one for each reason the concept was suggested. Thus the concept of making Q7 safe for the rook would be posted in the data base as follows:

```
((SAFE WR Q7)
  ((PLAN (CHECKMOVE WR Q7) (BK NIL) (SAFECAPTURE WR BQ))
        (THREAT (EXCH WR BQ)) (LIKELY 0))
  ((PLAN (SAFEMOVE WR Q7) (((BK NIL) (SAFECAPTURE WR BR)) ((ANYBUT BK) (SAFECAPTURE WR BR))))
        (THREAT (FORK WR BK BR)) (LIKELY 0)))
```

The productions can access this structure as one concept but can still access the necessary information about each reason. This information is, in fact, used in a number of ways. When a production accesses the values of attributes, it may not care which attribute list the values come from, or it may require that the values for each attribute come from the same list (as in the decoy example above), or it may want the "best" (in some sense) value for each attribute whether the different values come from the same list or not, or it may want values for one attribute to come from lists which have some particular value for another attribute. This accessing of the information in a concept is so complex that it can be looked on as pattern matching in itself. Thus the productions in a KS can be looked upon as matching patterns in the given chess position and matching patterns in different concepts.

Concepts and their accessing are quite complex. Intuitively, the productions in PARADISE are not making deductions, per se, but coming up with ideas that may or may not be right. Thus there are no "facts" that have certain deduced values that can be reasoned with. The complexity of later analysis requires that a production essentially put down "why" it thought of an idea. In this way, other productions can decide if the idea still looks right in light of what is known at that time. Most production systems avoid this complexity because the firing of a production is a deduction which adds a new piece of knowledge (or a new piece of evidence which supports a conclusion) to the system. This piece of knowledge or evidence is assumed right and the system is not prepared to later decide that it wasn't right to make that deduction.

6) Summary

A KS in PARADISE is a list of variables, a condition, a list of productions, and a list of sub-KSes, all of which know about a particular abstract concept. KSes communicate with each other and the rest of the system by posting concepts in the global data base. These concepts consist of a name, an instantiation of arguments, and a list of attribute lists. They are used as goals to produce a plan of action. KSes provide abstract concepts that are at a higher conceptual level than the productions themselves, and form the language that PARADISE thinks in. This section looks at abstract concepts as a higher level language.

Each pattern match in PARADISE provides new information about the current position since other patterns may check for these matches. For example, if a pattern which recognizes a trapped king position matches, other patterns can test for the existence of this pattern and in this way the system understands that the king is trapped. The action associated with a pattern may post concepts in the data base which KSes will use as goals. These concepts and the abstract concepts to which they correspond through KSes are at a higher conceptual level than the ideas produced by simply matching patterns. They can represent more vague and complex ideas, and can contain information which affects the execution of productions.

To understand chess positions using knowledge rather than a broad search, both pattern-level concepts and these higher level concepts are necessary in the reasoning process. As was shown earlier, a single pattern cannot recognize a whole plan of action in a complex chess position. PARADISE therefore uses KSes to provide abstract concepts. KSes are built out of productions and are in turn used as building blocks in the reasoning process which constructs a plan for a given position.

A major problem is the generality of these abstract concepts. How "high-level" should such a concept be? If they are too general then too many details are lost, and the system does not have the necessary facts to do some types of reasoning. For example, a decoy concept must be able to access the details of why it has been suggested so that decoys which destroy the original plan are not suggested. If the abstract concepts are not general enough, the system leans toward the extreme of needing a production for every possible chess position. The abstract concepts in PARADISE have been constructed to be as general as possible while still having access to all the detail necessary for expert-level performance. As shown in the previous section, this generality has made concepts in the data base so complex that productions must essentially match patterns in these concepts. This tradeoff of generality and specificity is discussed in more detail in chapter 7. (The actual KSes in PARADISE are described in chapter 3.)

In addition to providing abstract concepts, KSes help solve the problem of quickly accessing the relevant productions in a large knowledge base. Given the number of patterns a chess grandmaster seems to have, it is unreasonable to look through all of them each time. By communicating plans down the tree, PARADISE many times has particular goals in mind when it looks at a new position. It can then execute the relevant KS and immediately access the relevant productions.

The first chapter outlined some of the ways PARADISE attempts to communicate results from one part of the search tree to other parts. The abstract concepts provided by KSes are an integral part of this tree communication as they provide the language this communication uses. For example, plans are expressed using these concepts and they enable PARADISE to know what it wants to do before analyzing a new position in the search. The tree search will be discussed in detail in later chapters, the point being made here is that there are other uses of KSes than the ones in static analysis.

6) Similarities with other systems

The overall organization of the knowledge base has its roots in many previous AI systems. Numerous systems have used production systems to represent knowledge bases in a format amenable to modification. Most notably, the MYCIN system [Shortliffe74,Davis76] provides the ideas behind the representation of productions in PARADISE. However, PARADISE recognizes more complex patterns and its patterns access more complex data structures than is done in MYCIN (as the examples in this chapter, chapter 1, and chapter 3 show). One of the major differences between PARADISE and most other production systems (including MYCIN) is that the firing of a rule is not a deduction in PARADISE. As described in the last section, an action produces an idea which may later prove to be wrong and PARADISE must be able to deal with these wrong ideas.

Similar schemes for organizing productions into KSes have been used in other programs. In VIS [Moran73], more a "pure" productions system than a performance expert, productions are grouped into "procedures" for more efficient execution. In SU/X (and SU/P) [Feigenbaum77], a performance expert, productions are grouped into "KSes" which organize the productions according to the conceptual level of the deductions they do. PARADISE uses it KSes in new ways, however. They provide abstract concepts which form a language used for the reasoning in the static analysis of chess positions, for expressing plans, and for improved tree communication. This leads to complex communication between KSes which requires pattern matching in itself.

The productions in PARADISE communicate by posting concepts in the global data base which all productions may access. This idea comes directly from the HEARSAY speech understanding system [Erman76] where a global "blackboard" is used for communication between KSes. Unlike HEARSAY, PARADISE's blackboard is not structured (although the concepts written on the blackboard are).

AM ([Lenat76]) is a program which creates concepts and reasons with them, much as PARADISE does. AM separates concepts and tasks, while in PARADISE the concepts are also used as goals (or tasks). PARADISE's concepts combine the structure of both concepts and tasks in AM, using lists of attribute lists to represent the reason each concept has been suggested. In AM, it is assumed that tasks do not interact strongly and that concepts do not interact strongly. This can be done because AM's world is continuously growing and taking an action which is not the best at any one point is not especially costly. PARADISE, on the other hand, is continually constructing new worlds and retreating to the older ones when a poor move is made. It is very costly to make a mistake so the system must reason about how various concepts interact. To do this, the productions may need to access information from more than one concept in order to be matched. The next section describes the Production-Language which permits this.

B) Patterns and Productions in PARADISE

1) Introduction

Several issues influencing the design of productions in PARADISE will be briefly mentioned here, while the remainder of this section describes the Production-Language in which PARADISE's productions are written. The most important consideration in the writing of productions is the desire for expert performance. To accomplish this, it seems necessary to have a knowledge base which is easy to modify (this includes making changes and making additions). Otherwise incremental improvement will be very hard. To this end, productions in PARADISE are essentially independent of each other (the ordering in KSes can be determined syntactically) and are expressed in a straightforward manner, avoiding hidden side effects. Hopefully, a chess expert not familiar with the system could learn to understand and write productions without undue effort. (Chess experts have not been tested for ability to write rules in PARADISE, but several chess players have been able to understand previously written productions.)

Another issue faced in writing productions is the desire to communicate knowledge to the tree search. Computer chess programs can be placed on a continuum according to the bandwidth of the communication between static analysis procedures and the tree search. Most programs have a very small bandwidth. In fact, the fast searching programs do so little static analysis that there is little knowledge to communicate to the search. In PARADISE, the bandwidth is quite wide. This affects the design of the productions since the static analysis reasoning process is constantly concerned with building plans, attributes lists, and concepts which can communicate knowledge to the tree search. Many expressions in the productions are concerned with communication of knowledge to the search and not merely analyzing the best move in the current position.

Lastly, the issue of level of expression in patterns must be addressed. At a low level, a chess pattern could involve requirements placed on particular pieces and particular squares, as in Simon and Gilmartin's MAPP [Simon73] program. For example, MAPP pattern N155 states there must be a black pawn on the square C5 and a black pawn on the square D6. Such patterns will be referred to as "type 1" patterns. At a higher level, patterns could be expressed as an easily defined relationship between certain variables. The matching process must then search for instantiations for the variables which will make the relationship true. For example, a pattern which specifies that one piece must protect another might match the particular instance of the black pawns on C5 and D6. Such patterns will be referred to as "type 2" patterns. PARADISE's patterns are type 2, as are the patterns in Zobrist's program. The difference between type 1 and 2 patterns is the "grain size" (see [Davis76]) which the patterns use to access their domain.

At an even higher level, patterns could recurse upon themselves or instantiate an indefinite number of variables. Such patterns cannot be expressed as type 2 patterns and will be called "type 3" patterns. A type 3 pattern is not well defined here, rather it is meant to include any pattern which cannot be expressed as a type 1 or 2 pattern. A type 3 pattern may require a different grain size, or it may just require different types of processing within the same grain size. A chess example of a type 3 pattern would be the idea that a square is safe for a piece to land on. Matching safe squares would involve looking at all pieces bearing on a square (there may be none or a dozen), and ordering them for optimal deployment by both sides in order to determine who would win an exchange on the square. Recursion may be involved if one of the pieces is pinned since the safety of the pin object's square (i.e., the strength of the pin) would be a factor used to decide the order of deployment.

Type 1 patterns are clearly not suitable for the type of reasoning PARADISE must do. To write a pattern which describes a pin, an expert instructing PARADISE will describe the essential relationship between the pinner, the pinee, and the pin object. This results in a type 2 pattern which describes a relationship between three variables. Using type 1 patterns would involve writing a seperate pattern for every possible pin which is likely to occur. Besides being adequate to express such concepts, type 2 patterns can be written in a straightforward manner, making them easy to understand. Expressing type 3 patterns in a straightforward manner may be a problem (see chapter 7). Type 3 patterns have not been developed in PARADISE primarily because type 2 patterns have proved adequate for expressing the knowledge in PARADISE'S domain. Type 3 patterns would probably be needed to analyze chess end-game positions (see chapter 7), but PARADISE needs only one "pattern" that cannot be expressed in the type 2 Production-Language. This is the previously mentioned idea of a square being safe for a piece. PARADISE understands this by making it into a domain primitive function (although it does a very complex computation).

An example helps illustrate the kind of things PARADISE's type 2 patterns can and cannot do. A fork pattern in PARADISE will recognize one piece simultaneously attacking two opposing pieces by having the matching process instantiate three variables to the identities of the three pieces. While the patterns may use variables rather than speaking about particular objects, they may not have an indefinite number of variables. Thus the forking pattern cannot automatically match all pieces attacked by an offensive move, nor can it call itself recursively. PARADISE can generate all such pieces and can construct sets as the value of some variables, but these sets must be manipulated explicitly and relationships between individual members of these sets cannot be expressed.

2) Overview of the Production-Language

The environment in which productions in PARADISE operate has been described. The major features are KSes and their corresponding concepts which are posted in a global data base. Each production in PARADISE is in at least one KS. A KS is executed once for each concept corresponding to it in the data base. Thus each production has one specific concept as its goal whenever it is being executed (namely, the concept being used as a goal by the KS currently executing). In addition to this goal, each production has access to other concepts in the data base, patterns which have matched, and functions for domain primitives. Most productions are attempting to (eventually) suggest a plan of action with an accurate analysis of the possibilities of such a plan. (The structure of plans is discussed in section D of this chapter.) The only productions which do not try to suggest plans put concepts in the data base which the tree searching routines will access.

A production in PARADISE is expressed as a list of variables to be instantiated and a list of expressions, written in the Production-Language, which are matched to instantiate these variables and to perform actions. (See the production in figure 1.3.) Each expression must match for the production to match. All variable instantiations which cause a pattern to match will be found by the pattern matcher. The last zero or more expressions may correspond to actions and always match. The expressions before the "action" expressions constitute the pattern. Action expressions expect all the variables to be instantiated before they are matched.

Some variables in each production may be instantiated by the KS being executed using the particular goal which is being attempted. To illustrate this, consider figure 2.1 where white has the goal of making Q7 safe for the white rook. The production which may suggest QxB as a decoy would have at least five arguments: a piece and square for the intended move, a piece and square for the decoying move, and a defending piece for the piece to be decoyed. The piece and square for the intended move would be instantiated to the white rook and the square Q7 by the goal of the SAFE KS (before the decoy production is even attempted). The decoy production then attempts to match the other piece and square to some move which will decoy a defending piece (which also must be matched) from protecting Q7 (without destroying the plan to follow and without sacrificing too much material). If the match were successful, the plan would be to make the decoying move (in this example, moving the white queen to the square the black bishop is on) and if the defending piece (the black queen) captures the decoy, then reply by making the intended move (i.e., rook to Q7).

PARADISE restricts the possible instantiations of a variable by the name of the variable. There are v. .ables for each of the categories listed below. In this list the variable name (in parentheses) follows the category name. Integers may be appended to the end of the variable names to produce distinct variables names for different objects in one category.

SQUARE (SQ) matches any square on the board
COLOR (COL) matches either black or white
ATTRIBUTE (ATR) matches any value that an attribute might have in an attribute list
ANY PIECE (AP) matches any piece on the board
PIECE (P) matches any friendly piece on the board
SLIDING PIECE (LP) matches friendly pieces which move along lines (queens, bishops, rooks)
MAJOR PIECE (MP) matches any friendly piece except a pawn
SUBJECT PIECE (SP) matches any friendly piece except the king
PAWN (PN) matches any friendly pawn
KING (K) matches the friendly king
DEFENDING PIECE (DP) matches any defensive (opponent's) piece
DEFENDING (D) may be similarly appended to the other categories of friendly pieces.

The following parts of this section describe all the expressions in the Production-Language. Productions must access information about the current chess position. They do this by accessing primitive functions and patterns (see chapter 1) with the PATTERN and FUNCTION expressions in the Production-Language. Since pattern matches are deleted when a KS finishes executing, a production can only access patterns in the PRIMITIVE KS and patterns which have been matched earlier in the execution of the current KS. Part 3 of this section describes the PATTERN and FUNCTION expressions and then describ. : in detail each of the 12 primitive functions and the 18 patterns in the PRIMITIVE KS.

Productions must also access the concepts which have been posted in the data base. PARADISE is performance oriented so the Production-Language is very ad-hoc, and this is especially true of the expressions which access concepts. New expressions are added to the language as they are needed to express something, without attempting to be general. Part 4 of this section describes how the Production-Language provides access to concepts in the data base.

Part 5 of this section describes the connectives provided by the Production-Language for combining expressions into larger combinations. The connectives provide all types of Boolean combinations as well as ways of producing sets. Part 6 describes the action expressions provided by the Production-Language for posting concepts and plans in the data base. Part 7 summarizes the important aspects of the Production-Language. After the detailed description of the Production-Language, plans are described in section C of this chapter. Then in chapter 3, an example analysis is presented which includes actual productions written in this language.

3) Accessing Chess Primitives

PARADISE has both primitive functions and the patterns in the PRIMITIVE KS (which are expressed mostly in terms of the primitive functions) for accessing the chess primitives in the current position. Once a basic set of primitive functions has been developed, deciding whether a particular primitive should be a pattern or a function is mostly a store vs. recompute tradeoff. Functions are recomputed each time they are accessed while patterns are matched once, and each possible variable instantiation which matches the pattern is then stored in the data base. For example, the question of whether three given squares are in a line is a function since it is easier to compute this each time than to store all 3-square combinations which are in a line and then search the list for the three given squares. On the other hand, the question of whether a piece and square represent a legal move is a pattern since it is easier to search the list of legal moves than to recompute. Thus, all legal moves in a given position are stored as occurrences of the LEGALMOVE pattern (whose variables are a piece and a square to which the piece can move).

The FUNCTION and PATTERN expressions in the Production-Language are used to access functions and patterns. The FUNCTION expression in the Production-Language consists of the word FUNCTION, followed by the name of the function, followed by an Arg-Spec. An Arg-Spec is a list of specifications for each argument of the pattern in terms of the variables of the production containing this expression. A specification can be one of three things: an integer representing the internal name of an object, a variable of the production in which the Arg-Spec occurs (the variable's name restricts possible instantiations), or some function applied to such a variable (the function should yield the type of value expected by the pattern to be matched). Such functions can be created whenever needed. PARADISE currently has the following functions for transforming variable names:

LOCATION applied to a piece returns the square where that piece resides.
OCCUPANT applied to a square returns the piece which resides on that square.
COLOR applied to a piece returns the color of that piece.
OTHER-COLOR applied to a piece returns the color of the other side's pieces.
TYPE applied to a piece returns the type of that piece (king, knight, etc.)
VALUE applied to a piece returns the value of that piece. These values are:

PAWN	10
KNIGHT	32
BISHOP	33
ROOK	50
QUEEN	90
KING	320

To illustrate the use of a FUNCTION expression in a production, let us suppose the production has SQ1, SQ2, and SQ3 as variables. Then the expression (FUNCTION LINE SQ1 SQ2 SQ3) would match only if the three squares are in a line. The PATTERN expression is like the FUNCTION expression except that the word PATTERN replaces the word FUNCTION. If a production has P1 and DP1 as variables, then the expression (PATTERN LEGALMOVE P1 (LOCATION DP1)) would match only for legal captures.

Accessing patterns and executing functions may both instantiate variables in a production. Functions instantiate variables by looping through all possible argument values and calling the function. Instantiating more than one variable would result in nested loops, so functions are currently limited to only instantiating one variable. (There is a FUNCTIONE expression which instantiates more than one variable, but it only works for certain functions.) Primitives which need to instantiate more than one variable can usually be made into patterns. Accessing a pattern may instantiate all variables in the pattern.

Before describing other expressions in the Production-Language, the primitive functions and patterns used in PARADISE will be described. This should give something concrete to associate with the Production-Language. The primitive functions are described first and then the patterns in the PRIMITIVE KS are described. Readers not interested in chess details may skip ahead to part 4 of this section on page 66.

a) Functions

Each function used in PARADISE is described below. The first six are well-defined relations on the chess board. The function SAFEP however involves a complex computation that is described below. This is the one time PARADISE recognizes a type 3 pattern. EXCHANGE and OVERPROTECTED are two functions which provide additional access to the computations done in SAFEP. In most cases intuitive and inexact definitions are given to help the reader understand. Exact definitions are provided whenever confusion may result.

LINE SQ1 SQ2 SQ3
This matches if all 3 squares are in a line with SQ2 in the middle. This is independent of the current position and accesses the array CLR(SQ1,SQ2). Each entry in this array is a bit vector representing the squares which are on a line between SQ1 and SQ2. The array is also accessed by other functions which deal with lines.

CANATTACK P SQ1 SQ2
A piece is said to "bear directly" on a square if the opponent's king, when placed on the square, would be in check by the piece. CANATTACK matches if P would bear directly on SQ2 if he were placed on SQ1. This is read as "P can attack SQ2 from SQ1" and is used primarily to test what a piece can accomplish from a new square to which he may be moved. For example, in position 2.1 the white rook can attack the location of the black queen from Q7, but can not attack either black rook from Q7. The pin-like attack on the black king and rook can be recognized (using LINE above) by noticing that the white rook can attack the location of the black king from Q7, that the white rook can attack the location of this black rook from the location of the black king (this guarantees that only the king is between Q7 and the rook), and that Q7, the location of the black king, and the location of the black rook are in a line. CANATTACK accesses the array VUE(PIECE-TYPE,SQ). Each entry in this array is a bit vector representing the squares to which a piece of type PIECE-TYPE could move from SQ if the board was empty. (Pawns are not included.) This array is accessed by all functions which deal with moving pieces.

CANMOVE P SQ1 SQ2

This matches if P could move to SQ2 if he were placed on SQ1. This is similar to CANATTACK except that SQ2 must not be occupied by a friendly piece and if P is a pawn then SQ2 must be in front of SQ1 and unoccupied or diagonally adjacent and occupied by the opponent.

IN-VUE MP SQ1 SQ2

This matches if P could from from SQ1 to SQ2 on an empty board, and is a simple access to the VUE array. It is used to determine if MP could possibly affect SQ2 from SQ1 (since some intervening pieces may be removed). This is read as "SQ2 is in MP's vue from SQ1".

OCCUPIED-BY-COLOR SQ COL

This matches if SQ is occupied by a piece with the color of COL.

SAMEVALUE P1 P2

This matches if the two pieces have the same value, where same is defined as being within two points of each other. Thus knights and bishops have the same value.

SAFEP P SQ

This matches if the opponent could not capture P with gain when P is placed on SQ. Unlike the previous functions, this is not a well-defined, exact relationship. Many factors (described below) are taken into account but the complexity of the situation means that occasionally the answer may be wrong. Whenever a move is referred to as being "safe", it means that the function SAFEP matches for the piece being moved and its destination square.

SAFEP uses the OCCUP routine which calculates the occupiability of a particular square (SQ) for each side. The occupiability for a side is the largest material value which that side could place on SQ so that the opponent could not capture with gain on SQ. OCCUP calculates this by playing out a capturing sequence on SQ, deciding at each move which piece would be best for the side on move to capture with. (How the "best" piece is determined is described below.) With one exception, only moves to SQ are considered. Thus the occupiability does not reflect the fact that a piece may be giving up a critical defensive function elsewhere on the board to capture on SQ. The one exception is when the opponent has just moved a pinned piece to SQ. In this case the side on move may capture the pin object in which case the capturing sequence is ended.

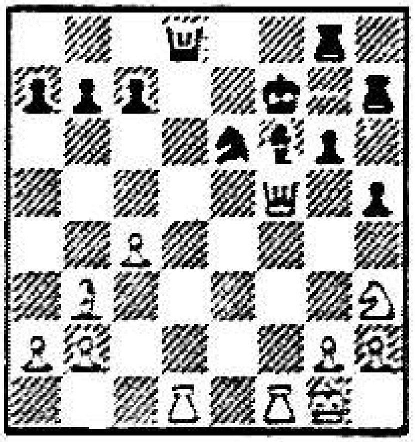
In this manner OCCUP builds up a tree which it evaluates with a minimax procedure to obtain the occupiability of the square. In this evaluation, the side on move can always accept the current value and not play a move (i.e., can quit the exchange whenever it wishes). This tree is linear except at nodes produced by deploying a pinned piece to SQ, where the side on move can choose to continue the exchange, or quit and capture the pin object. OCCUP must know the value of the pin (i.e., how much will be gained by capturing the pin object) to decide which of these two is better. How the value of a pin is found is described below.

To decide which piece is best to deploy for the side on move, OCCUP has a list of all pieces which bear directly on SQ. It wants to deploy the least valuable piece but this is complicated by the fact that some pieces may be pinned, some may pin opponent's pieces which could move to SQ, and some may have opponent's pieces bearing "through" them onto SQ. (A "through" occurs, for example, when two rooks bear along the same file onto SQ. The second rook becomes available for deployment to SQ after the first has been moved there.) There is a special case when a pinner is not given the responsibility of retaining the pin: that is when a friendly piece, not more valuable than the pinner, is lined up behind the pinner so that it will retain the pin if the pinner moves. (Complications arise if the "retaining" pinner is more valuable than the pinner. Not allowing this could be a source of error, though an unlikely and unimportant one.) A piece which has an opponent's piece bearing "through" it onto SQ is not deployed when the opponent has no pieces or only his king bearing directly on SQ (unless it is the only piece available). A pinner is only deployed if he is the least valuable piece available and is capturing a more valuable piece. A pinned piece is deployed when it is the least valuable piece available and the value of the pin is less than the value of any other available piece (except in special cases when other pieces are available and the opponent has no pieces or only one piece bearing on SQ).

To determine the value of the pin, OCCUP calls itself recursively on the square where the pin object resides. It is possible this recursion may circle back to the original square, in which case some arbitrary escape must be arranged. In this case OCCUP assumes the value of the circle-completing pin to be the value of the pin object. In practice this rarely occurs and no detected error has been caused by this convention. Another possible source of error is that recursive calls to OCCUP deploy pieces without regard to any deployment of those pieces in "surrounding" calls to OCCUP. Again, such errors have not surfaced in actual play.

Deployment of a piece in OCCUP is not trivial. When pinners are deployed, it must be noted that the pinned piece is now free. Likewise, deploying pinned pieces releases pinners. Whenever any piece is deployed, a check must be made for other pieces which may have been bearing "through" the deployed piece onto SQ. Calculating OCCUP for a square is expensive (in terms of cpu time) so the results are stored in an array to avoid recalculation.

Figure 2.2 gives the occupiability values for each square in position 2.1. The value for white is above the value for black in each square.



PAWN	10
KNIGHT	32
BISHOP	33
ROOK	50
QUEEN	90
KING	320

0 320	0 320	0 320	0 50	0 320	0 320	0 320	0 320
320 320	320 320	0 320	90 50	0 320	0 320	0 320	320 320
0 320	0 320	0 320	0 50	320 90	50 50	0 90	0 320
320 0	320 0	32 90	90 0	230 0	10 50	32 0	0 90
320 0	320 320	320 0	0 50	320 0	32 0	10 90	230 320
320 0	320 0	230 0	90 0	320 320	320 0	320 0	320 0
320 0	230 320	320 0	90 50	320 320	320 0	320 0	320 0
320 0	320 0	320 0	90 0	320 0	320 0	320 0	320 0

Figure 2.2
Occupiabilities of squares for position 2.1

The OCCUP routine described here is very similar to the one in Berliner's CAPS system (although Berliner's performed more functions than just calculating occupiabilities). The major difference is in the handling of pins, especially the recursion used to calculate the values of pins. The OCCUP described here seems to be a slight improvement on Berliner's since some pins are evaluated correctly in PARADISE while they are not in CAPS. As shown in Berliner's thesis, OCCUP is very sophisticated and seems to calculate occupiabilities as well as a chess master. It is important that the occupiabilities be accurate since pessimistic values would cause the system to discard good moves, and optimistic ones would cause it to try obviously losing moves. In practice OCCUP provides accurate values, even in complex positions. Errors in analysis have not been caused by wrong occupiability values. (This does not mean occupiabilities are never wrong, merely that the rare errors do not usually affect the analysis.) It should be noted that occupiabilities are not all-important. It is necessary to play moves to unoccupiable squares in some positions. Analysis must look at many things besides occupiabilities when considering a move.

Given the occupiability (OCC) of SQ for P's side, the function SAFEP must now determine if P would be safe on SQ. There are two cases: either P currently bears directly on SQ or it does not. If it does not then P is safe whenever OCC added to the value of any piece captured on SQ is as great as the value of P. The situation is more complicated when P bears directly on SQ. OCC plus any captured piece must still be as valuable as P, but in this case it still may not be safe since moving P to SQ removes P from bearing on SQ and OCCUP counted on P when it calculated OCC. By making the first move to SQ with P, we are in fact simulating a capture sequence on SQ that may have been used to calculate the other side's occupiability value. Thus for P to be safe, the opponent's occupiability value must zero unless we are capturing a piece in which case it must not be larger than the value of the captured piece. The situation is actually more complicated than this since P may not have been the optimal piece for his side to deploy first to SQ in the OCCUP calculation. A completely accurate answer would require redoing the OCCUP calculation and forcing a non-optimal deployment. However, this is expensive and does not seem to be justified in these circumstances. PARADISE must be able to cope with wrong answers anyway and the above approximation has been found to be adequate in practice if special-case approximations are made when P is pinned or a pinner.

As an example of the use of SAFEP, consider the location of the black bishop in position 2.1. The occupiability of this square is 90 for white and 50 for black. It is not safe to move the white queen to this square. Although white's occupiability is high enough, the queen bears directly on the square and black's occupiability is higher than the value of the bishop. Thus white would lose 50 (the opponent's occupiability) and gain 33 (the value of the captured bishop). It is safe to move the white knight here since it does not bear directly on the square and white's occupiability is high enough. Thus white may want to maneuver the knight to capture the bishop.

SAFENB P1 SQ
This is similar to SAFEP, the only difference being that the calculation of safety is made assuming that P1 does not bear directly on SQ (even if it does). This is useful for deciding if P1 can recapture on SQ should the opponent capture there.

EXCHANGE P1 SQ
This matches if the opponent can exchange P1 should P1 move to SQ. Only immediate (obvious) exchanges are considered. Exchanges which take many moves are ignored. This is read as "P1 can be exchanged on SQ". Normally, this is used when P1 can safely move to SQ to determine if P1 can remain on SQ or if he can be exchanged by the opponent. For example, in position 2.1 it is safe for white to play N-B4 but EXCHANGE matches for this move since the black knight can immediately capture on B4. Therefore if the reason for moving to B4 was so the knight could maintain some function on B4, it is probably futile. More specifically, EXCHANGE matches in the following cases: If SQ is not occupied then the opponent must have a piece which is not pinned, bears directly on SQ, and is not more valuable than P1. If SQ is occupied, then the occupant must have the same value as P1 and the opponent must have some piece which can safely recapture P1 on SQ. Knowing the opponent can safely recapture is not a trivial problem. If P1 can attack SQ from his current location then the opponent's occupiability for SQ must be as great as the value of the piece on SQ, otherwise some opponent piece that can attack SQ from his current location must be safe on SQ. Only the most obvious exchanges are detected since PARADISE may miss good moves if EXCHANGE matches when it shouldn't. Not having EXCHANGE match when it should means more work, but the correct solution should not be missed.

OVERPROTECTED COL SQ
This matches if the opponent has enough pieces bearing on SQ so that another supporting piece of COL will not make SQ safe for him. This is read as "SQ is overprotected against COL". It is used primarily to prevent suggestion of obviously losing moves attempting to make SQ safe. In position 2.1, the square K8 is overprotected against white. Thus no simple supporting move, such as B-R4, can make K8 safe for a white piece. (Tactics like removing the black queen may still work.) Once again, this function is an approximation. It actually matches if COL cannot safely land a pawn on SQ and the opponent has at least two pieces bearing directly on SQ. To accurately determine if a square is overprotected it is necessary to do an OCCUP-like computation where functions are assigned to pieces in the capturing sequence (this is done by Berliner in CAPS). This expense did not seem justified here since the function defined above is sufficient for discarding simple moves, and other productions can make SQ safe whether it is overprotected or not.

EQUAL   NOT-EQUAL
The meaning of these functions is obvious. They may be applied to any two arguments which are both pieces, both squares, both types of pieces, or both values of attributes.

b) Patterns

The following patterns constitute the PRIMITIVE KS of PARADISE. Before processing any goals in a particular position, PARADISE first calculates the instances of each pattern in the PRIMITIVE KS and inserts them in the data base. Unlike any other KS, patterns in the PRIMITIVE KS are matched for both sides (i.e., the offensive and defensive variables in each production are matched to both sides). For example, the LEGALMOVE pattern is matched for both sides, not just the side on move. These pattern matches are never deleted from the ata base while that position is in core memory. Five of the first six patterns below ar the same as the bearing relationships in Berliner's CAPS system. All these patterns car be expressed as productions (without actions) in PARADISE using the Production-Langu s to access the functions previously mentioned. Once again intuitive definitions are u nd more formal ones given where needed.

DIR P SQ
This matches when P bears directly on SQ. More explicitly, if the opposing king were on SQ then he would be in check by P. When this pattern matches it will be said that "P bears directly on SQ". This is equivalent to saying that P can attack SQ from his current location. By using his pattern instead of CANATTACK, both variables can be instantiated.

THRU LP AP SQ
This matches if LP bears "through" AP onto SQ. This means that P would bear directly on SQ if (and only if) AP (which now intervenes) were removed from the board. LP must, of course, be a slidin piece. This is read as "LP bears thru AP onto SQ." In position 2.1, the white queen bear thru the black bishop onto the black king's location, and the white rook on KB1 bears thru the other white rook onto QR1.

OTHRU LP P SQ
This is a special nstance of a THRU pattern where the intervening piece is the same color as LP and bears directly on SQ. This occurs when two friendly pieces are on a line which they can both move along. In position 2.1 the white rook on KB1 bears othru the white queen onto the location of the black bishop. OTHRU defines the ability of LP to exert control on S through a friendly piece which also exerts control on SQ.

ETHRU LP OP SQ
This is a special instance of a THRU pattern where the intervening piece is of a different color than LP and bears directly on SQ. This occurs when two pieces from opposing sides are on a line which they can both move along. It defines the ability of LP to exert control on SQ through an opponent's piece which also exerts control on SQ. There are no ETHRU patterns in position 2.1.

DSC LP P SQ
This is a special instance of a THRU pattern where the intervening piece is the same color as LP and does not bear directly on SQ. This occurs when two pieces of the same color are on a line which only LP can move along. It corresponds to the ability of LP to make a discovered attack on SQ if P is moved from the line. In position 2.1, the white bishop bears DSC through the white pawn on B4 onto the location of the black knight.

OBJ LP DP SQ
This is a special instance of a THRU pattern where the intervening piece is of a different color than LP and does not bear directly on SQ. This occurs when two pieces of opposing sides are on a line which only LP can move along. It corresponds to a pin ray along which LP could pin DP to a pin object. The pin object may be a piece on SQ, a piece farther down the line, or may not exist at all. In position 2.1, the white queen bears obj through the black knight onto Q7 and QB8.

LEGALMOVE P SQ
This matches when P can legally move to SQ in the current position. This can be defined using the functions given previously because castling, en-passant captures, and pawn promotion are not in PARADISE's domain of tactically sharp middle game positions (by definition).

PINRAY LP DP1 DP2
This matches when LP bears thru DP1 (an opponent's piece) onto (the location of) DP2 (another opponent's piece). In other words, this matches any ETHRU or OBJ pattern which has an opponent's piece on the square given by its third variable. There are two (and only two) pinrays in position 2.1. The white queen has a pinray thru the black bishop onto the black king and another thru the black KNP onto the black rook.

PIN LP DP1 DP2
This matches the instances of PINRAY in which DP2 will actually be threatened by LP should DP1 move off the pinray line. This occurs when LP can safely be put on DP2's location (uses SAFEP), LP can not be exchanged on DP2's location (EXCHANGE), and there is no piece friendly to LP and less valuable than DP2 which can legally capture DP2 in the current position (i.e., DP2 is already a dead man). In position 2.1, both pinrays happen to be pins.

ONLY-DIR P SQ
This matches when P is the only piece on his side that bears directly on SQ. This helps test if P is the only piece which can capture on SQ. There are numerous instances of this pattern in position 2.1, such as the black king being the only protection of the black knight's location.

ONLY-PROTECTION P SQ
This matches when P is the only piece on his side that bears directly on SQ and there is no piece which bears othru P onto SQ. This helps test if moving P will give up all control over SQ by his side (P may still bear directly on SQ from his new location). Most ONLY-DIRs are also ONLY-PROTECTIONs since most pieces do not have pieces bearing othru them.

DSC-THREAT P LP DP SQ

This matches when moving P to SQ will advantageously uncover a discovered attack by LP on DP's location. LP must bear DSC thru P onto DP's location. LP must be safe on DP's location and LP must not be able to be exchanged on DP's location. P must be able to legally move to SQ, and LP's location, SQ, and DP's location must not be in a line (or the attack won't be uncovered). This pattern also checks to see that DP cannot immediately and advantageously capture P after he moves to SQ since this would not be an advantageous uncovering. This is accomplished by checking, whenever DP bears directly on SQ, that P is not the only protection of SQ, that SQ is not overprotected against P, and that DP1 is not safe on SQ. This check is not exact since DP may still be able to capture advantageously on SQ, but once again the rare cases that are not matched will cause the system to err in the proper direction (i.e. by producing more DSC-THREATs than it should). It is said that moving P to SQ is "a DSC-THREAT move which uncovers P's attack on DP". In position 2.1 there is one DSC-THREAT: the white pawn on QB4 can move to QB5 and advantageously uncover the white bishop's attack on the black knight. The black knight can capture the pawn on QB5 but not advantageously.

MOBIL P SQ

This matches if P can legally move to SQ and cannot be captured with material gain on SQ. Whenever P captures a piece on SQ that is at least as valuable as himself, he cannot be recaptured with material gain since the opponent can do no better than recoup his loss. Moving P to SQ is said to be a mobil move. This is a very important concept since chess players (human and machine) generally make only mobil moves except in sacrificial situations. A program can save itself much effort by not trying legal moves which are not mobil. The accuracy of this pattern is important to system performance although many times non-mobil moves must be played. The use of SAFEP enables PARADISE to calculate mobil moves very accurately. Inaccuracies in this pattern have not been the cause of errors.

Most mobil moves are simply legal moves which are safe. However, otherwise unsafe moves that are made safe by uncovering obviously winning discovered attacks are also matched by this pattern. For example, if white can move a knight off a file in such a way that one of his rooks puts the black king in check, then the knight will not be captured with material gain unless he is captured by the black king. It is necessary to recognize such moves as mobil, otherwise every production which tried to match mobil moves would also have to check for such discovered attacks. For these unsafe moves to be matched, they must be DSC-THREAT moves which uncover some discovered attack, say P2 onto DP1. MOBIL then has a fairly complex pattern which assures two things: one, that P2 can gain more material by capturing DP1 than DP1's side can gain by capturing P1 on SQ (which is unsafe for P1), and two, that DP1's side cannot capture P2 unless this is less threatening than what P1 has threatened on SQ. The details of this are not important since PARADISE will try discovered attacks even if the uncovering moves are not mobil.

In position 2.1, the white rook on Q1 has mobil moves to the following squares: Q8, Q5, Q3, K1, QB1, QN1, and QR1.

MOBIL-STAY P SQ
This matches instances of MOBIL in which P cannot be exchanged on SQ. Intuitively, this means P can move to SQ without loss and can maintain himself there to perform some function (see EXCHANGE). Note that if the opponent must give up material to capture P on SQ, then P is mobil-stay there. In position 2.1, the white knight is mobil to B4 but not mobil-stay since the black knight can exchange him. All the mobil moves of the white rook on Q1 are also mobil-stay. (The move to Q8 is mobil-stay because black loses his queen but can only recapture a rook on Q8.)

CHECKMOVE P SQ
This matches if P can legally move to SQ, and the opposing king will be in check after the move (either by a direct or discovered attack). Examples are R-Q7 and QxB in position 2.1.

MATE P SQ
This matches instances of CHECKMOVE where the opposing king has no legal move after P moves to SQ (except for possibly being able to capture P). Thus the only way for him to avoid mate is to capture P or, if P is a sliding piece, to interpose and block the checking line. This is used to help find quick mates, especially in quiescence analysis. The pattern is rather long since many things need to be checked. For every square that the opposing king can attack from his current location that is not occupied by a piece friendly to the king, it must be shown that either P can attack it from SQ, or that some piece which bears thru P's current location can attack it after P moves, or that some piece other than P bears directly on it.

PROTECTS P1 P2 SQ1 DP1
This matches if P1 is the only offensive piece which can stop DP1 from moving to SQ1 and winning P2 (this may involve, for example, capturing P2 on SQ1 or mating from SQ1). This pattern is used primarily to reject moves by P1 from consideration because of the defensive responsibilities such moves give up. Again only obvious situations are recognized since recognizing more subtle duties of P1 might reject moves that are good. This helps remove a number of obviously losing moves. There are three situations recognized by this pattern. First, if DP1 can legally capture P2, and P1 is the only-protection of P2's location then the pattern matches with SQ1 being P2's location. Second, if the MATE pattern matches for DP1 and SQ1, and P1 is the only-protection of SQ1, and there are no interpositions that might work (any interposition is deemed to work unless it moves a piece to a square that is overprotected against it) then this pattern matches with P2 being the king friendly to P1. Third, if DP1 can move to some square for which the MATE pattern matches for him, and no piece friendly to P1 bears directly on that square, and moving P1 to SQ1 is the only interposition that might work then the pattern matches with P2 again being the king friendly to P1. In position 2.1 the black king protects the black knight from capture by the white queen.

It should be noted that the idea of "protection" is too complex to be treated accurately by static analysis. The threatening piece may have subtle defensive duties elsewhere on the board which prevent him from making the threatened capture. This pattern helps weed out obviously losing moves, but PARADISE must be prepared to handle errors. Since errors will always be present, PARADISE does not spend more resources attempting to analyze defensive functions.

ENPRISE P1 DP1
This matches when P1 can legally capture DP1 with material gain in the current position. This is very simple to express since it is any mobil-stay move whose destination square is occupied. This pattern matches twice (once for each side) in position 2.1: the white queen is en prise by the black pawn and the black queen is en prise by the white rook.

Each time a KS is executed for a particular goal, the above patterns will be accessed by productions in that KS using the PATTERN expression of the Production-Language. The only other patterns which these productions can access are ones that were previously matched during the execution of the KS. Thus the above patterns are the only ones accessed by productions from every KS. The other expressions in the Production-Language which make it possible to express patterns like those above will now be described.

4) Accessing Goals and Concepts

The Production-Language uses the PATTERN and FUNCTION expressions described above to access patterns and functions. Productions must also access concepts that have been posted in the data base by other productions. By accessing attribute values in concepts, productions can determine how much a particular move threatens, how likely a particular plan is to succeed, and many other useful things. The expressions in the Production-Language which provide access to these values are described here. These expressions are ad-hoc and the descriptions below are fairly detailed so the reader may wish to skim this section.

The complexity necessary in accessing concepts in the data base has already been described in section A of this chapter. Each concept may have a number of attribute lists which give values for attributes. Some of the details of the expressions described here involve the values the final plans will expect for attributes. Plans will be described in detail in section C of this chapter, including a description of what attributes are expected and what the values of these attributes are expected to be. Suffice it here to say that the values of certain attributes are written as expressions in a Threat-Language which will also be defined in section C.

Whenever a production is being matched, some KS enclosing it is being executed on a goal. Thus there is one concept which is the goal currently being attempted. The word "goal" below refers to this particular concept (with all its attribute lists).

The expressions GREAT and ASGREAT provide the obvious arithmetical comparisons between integers. The expressions first convert their two arguments to integers which may involve accessing attribute values in the goal. This is normally used for comparing values of threats and pieces. The following are accepted as arguments by these two expressions: integers, names of attributes, variables of the enclosing production, names of functions followed by an Arg-Spec (as in FUNCTION or PATTERN expressions) for the function, and the function names PLUS and DIF which may be applied recursively to any of these arguments and provide arithmetical addition and subtraction. When an argument is the name of an attribute, the value of that attribute is retrieved from the goal and is evaluated as an expression in the Threat-Language to yield an integer. When an argument is a variable, the current instantiation of that variable is retrieved and evaluated as an expres-ion in the Threat-Language (the variable should be in the ATTRIBUTE category). When an argument is the name of a function, that function is applied to the given Arg-Spec and it should return an integer. The functions normally used here are ones defined by the Threat-Language. An example of a typical use of these expressions is in position 2.1 when the SAFE KS must decide if the threat of getting the white rook to Q7 is worth sacrificing queen for bishop. The GREAT expression is used and it evaluates the threat in the attribute list of the goal and compares it to a function whose evaluation yields the value lost by playing QxB. (This is explained in more detail in section C after the Threat-Language is introduced.)

The GOAL expression provides access to the values of attributes in the goal. Its argument is a list of attribute-value pairs. The language used to describe the values of attributes will not be described here except to say that these values may include variables from the enclosing production as part of their specification. GOAL expects all variables used within its argument to be fully instantiated. It matches if there is any attribute list in the goal whose value for an attribute matches the value for that attribute given in the argument to GOAL. If there is more than one attribute-value pair in the argument then all must match, but the match for each one may come from a different attribute list in the goal.

The GOAL1 expression is similar to GOAL except that it does not expect all variables to be instantiated and will instantiate them. To limit the number of instantiations (since there are in general a large number of attribute lists in a concept), only the "best" attribute list of the goal is considered. "Best" here is defined by the ordering function which orders the final plans for the tree search. This function is described in section C of this chapter. Basically this involves using the "likelihood" attribute to determine which attribute lists are likely to succeed, and then taking the most threatening (using the "threat" attribute) attribute list (of the likely ones) as the "best".

The CONCEPT expression is similar to GOAL but it accesses any concept in the data base instead of just the goal. Besides attribute-value pairs, it expects two other arguments: a concept name and an Arg-Spec for that concept. It matches as GOAL does except that the comparisons are not done against the goal but against any concept specified by the name and Arg-Spec. Like GOAL, CONCEPT will not instantiate variables in the attribute-value pairs of its argument. It will however instantiate variables in the Arg-Spec for the concept name. Thus variables in the enclosing production may be instantiated to any instance of the given concept for which the attribute-value pairs match.

The FOR-EACH-GOAL expression takes a list of expressions in the Production-Language as an argument and allows accessing, in turn, each individual attribute list in the goal. It does this by matching the expressions in its argument once for each attribute list in the goal. On each iteration a new goal is constructed which uses the variable instantiations and name of the original goal but only one of its attribute lists. This expression has no effect when the original goal has only one attribute list.

The REMOVE-ATR-LISTS expression allows accessing of particular attribute-lists according to values of their attributes. Two arguments are expected: a flag which should be T or NIL, and one attribute-value pair. This expression will remove from the goal any attribute lists which contain an attribute-value pair that matches the one given in the argument (or, depending on the flag, may remove any that do not match). Any variables used in the attribute-value pair should be instantiated since no instantiations are done by this expression. The expression matches if there is at least one attribute list left after the removal process, otherwise it fails. The removal of attribute lists from the goal only affects the remaining expressions in the enclosing production. When the next production in the KS is executed, the original goal will be intact.

The expression FIRSTMOVE is used only in productions that are in a KS which uses a plan as its goal (plans can be considered as special instances of concepts). The two arguments of FIRSTMOVE are both variables of the enclosing production, one for a piece and one for a square. They are instantiated (if necessary) to match the first move specified in the goal (a plan). This may involve converting part of the plan in the goal to a move (see section C of this chapter for a fuller description of plans).

One way knowledge is communicated through attribute lists is by having conditions as the values of certain attributes. Basically a condition is a list of expressions in the Production-Language (and so could be considered to be a "pattern") where some of the variables may have been replaced by the actual internal representations of objects to which the variables were bound in the production which created this attribute list. In this way, relationships between variables in the current production and variables in some previously executed production can be tested. Conditions are described in more detail in chapter 3. The CHECK-CONDITION expression is used to test such conditions. The one argument of this expression is the name of an attribute. CHECK-CONDITION assumes the value of that attribute name in the goal will be a condition and it uses the normal Production-Language interpreter to match this condition in the current context. No instantiation is done. Any attribute list whose condition does not match is removed from the goal (for the rest of the current production as in REMOVE-ATR-LIST). At least one attribute list in the goal must match its condition for CHECK-CONDITION to match (a non-existent condition implies a match). The relationship of the condition to the production in which the CHECK-CONDITION occurs is described in chapter 3.

The expression PIECESAFE expects two arguments: a variable representing a piece, and any argument that GREAT or ASGREAT would accept (i.e., one that can be converted into an integer representing a threat value). The piece must already be instantiated. PIECESAFE determines (using occupiabilities) what the opponent could gain (if anything) by capturing the given piece on its current location, and matches only if the value specified by the second argument is greater than this. This is used to avoid preparing an elaborate plan for some piece which is going to be captured immediately anyway. For example, in position 2.1 we would not want to play one of the white rooks to K1 with the idea of making it safe for the white queen to capture the black knight (since black can play PxQ with more gain than anything a white rook can threaten from K1). This expression does not access concepts in the data base but is included here since it does not fit anywhere else.

5) Connectives

Several logical connectives exist in the Production-Language for combining expressions. AND and OR have an obvious meaning. They take an indefinite number of expressions as arguments. They both instantiate variables of the enclosing production so that all instantiations which match the OR or the AND are found.

NOT and NEVER allow other expressions to be negated (e.g., to test that patterns or functions do not match). As arguments they take an indefinite number of expressions which are implicitly anded together. NEVER assumes that all concerned variables are instantiated and does a simple negation. NOT is more costly to execute but will instantiate variables. NOT loops through all possible variable values and collects the ones for which the expressions in the NOT did not match. Like FUNCTION, it is limited to instantiating only one variable to avoid nested loops. Also, if the expressions in NOT do not match for any instantiations, then NOT does not instantiate any variables but merely matches with variable values as is.

The EXISTS expression allows checking for the existence of some "pattern" (i.e., a list of expressions in the Production-Language) which may not involve the variables of the enclosing production. EXISTS expects two arguments: a list of variables distinct from the variables of the enclosing production and a list of expressions which are implicitly anded together. EXISTS matches if there is some way to instantiate its own variables so that its expressions will match. When the variables of the enclosing production are not instantiated, EXISTS finds all instantiations for which the EXISTS expression can find matches.

The UNIQUE expression is usually applied to a list of expressions (assumed AND) when some variables are not instantiated. It matches if there is one and only one instantiation which will match the expressions. It is used, for example, to check if there is only one piece protecting a certain square. This is a special instance of the ability to check the length of a list of possible instantiations, but this special instance has been sufficient to express the chess knowledge encoded in PARADISE.

The IMPLIES expression allows verification that a certain "pattern" will match for each instantiation which matches another "pattern". (A "pattern" is a list of expressions in the Production-Language.) This effectively generates a set (of possible instantiations) and then tests each member of the set. IMPLIES expects three arguments: a list of variables distinct from the variables of the enclosing production, a list of expressions corresponding to the premise ("if" part) of the implication, and a list of expressions corresponding to the conclusion ("then" part) of the implication. IMPLIES matches if all possible instantiations of the variables local to the IMPLIES which match the premise also match the conclusion. (It also matches if no instantiations match the premise.) All variables local to the IMPLIES must be instantiated by the premise. If variables in the enclosing production are uninstantiated then IMPLIES will find all instantiations which would match the IMPLIES. An example of the use of IMPLIES is in the MATE pattern described in the last section. Here the fact that the defending king can move to a particular square must imply that the square is covered by an offensive piece after the mating move.

6) Actions

The only part of the Production-Language that has not yet been described is the action expressions. If plans are considered as special instances of concepts, the only important action in PARADISE is the posting of concepts in the data base. However PARADISE has different actions for expressing plans than it does for expressing other concepts, and it also has actions which allow for different methods of specifying the attribute lists of the concept or plan to be posted. In all, there are five different action expressions in the Production-Language: two for posting ordinary concepts and three for posting plans. They differ in the method used for specifying attribute lists. The action expressions described here always match.

The ACTION expression is used to post a concept in the data base. It expects three arguments: the name of the concept to be posted, an Arg-Spec which gives variable instantiations for the concept, and a list of attribute-value pairs for constructing attribute lists. The new concept inherits any attribute-value pairs that the goal has unless a new value is specified for that attribute in the third argument to ACTION. The values of attributes may be many different things such as conditions, integers, plans, or expressions in the Threat-Language. The specification of attribute values may involve accessing attribute values in the goal (thus a new value may be constructed by building upon the old value for that attribute). To accomodate this, ACTION makes one attribute list for the new concept for each attribute list in the goal. The attribute-value pairs given as the third argument are evaluated once for each attribute list in the goal, each time forcing references to attribute values to come from that particular attribute list. Once the concept is constructed, it is posted in the data base. If there already exists a concept in the data base with the same name and variable instantiations, then the old and new concepts are combined into one by merging their attribute lists.

The ACTION-NEW expression does the same thing as ACTION except that no attribute values are inherited from the goal and only one attribute list is produced for the new concept. Thus the attribute-value pairs are only evaluated once and references to values of attributes in the goal can take two forms: the reference can be to the "best" attribute list in the goal (as in the GOAL1 expression), or it can cause a list to be made of the values for that attribute from all the attribute lists in the goal. The process of inserting the new concept in the data base is the same as in ACTION.

The PLAN expression posts plans in the data base. Plans are discussed in the next section. Here it is only necessary to know that a plan is a concept where the variable instantiations have been replaced by an expression in the Plan-Language. PLAN expects two arguments: an expression in the Plan-Language and a list of attribute-value pairs which are evaluated exactly as they are in ACTION to yield attribute lists. A new plan is constructed by evaluating the Plan-Language expression and combining it with the attribute lists. These new plans are inserted in the data base as concepts where INITIAL-PLAN is the concept name, the Plan-Language expression replaces the variable instantiations, and the attribute lists are as specified. The process of insertion checks if there are any INITIAL-PLAN concepts already there which begin with the same move in the Plan-Language expression. If so, the two are combined into one by merging their attribute lists and changing the Plan-Language expression so there is a branch after the first move which will allow execution of either plan at that point. It is important that the plans be combined in this manner so the tree search will have access to all pertinent continuations after searching along this plan for two ply.

The PLAN-NEW expression is like PLAN except that the attribute lists are evaluated as they are in ACTION-NEW.

An INITIAL-PLAN concept differs from a FINALPLAN concept in that no KS has analyzed these plans for what functions the moves in the plans give up in the current position (this will be discussed further in the next section). The important point here is that the INITIAL-PLAN KS will use these concepts (plans) as goals for execution. The productions in the INITIAL-PLAN KS may use the FINAL-PLAN expression which constructs new plans and posts them as FINALPLAN concepts. (Productions in other KSes may also use the FINAL-PLAN expression if they do not want the plan produced checked for relinquished functions.) FINAL-PLAN takes two arguments: an expression for calculating a plan and a list of attribute-value pairs. The attribute lists for the new plan are calculated as they were in ACTION (and PLAN). The expression for calculating a plan can be one of three things: the symbol "GOAL" which means the plan in the goal should be used as is, an expression in the Plan-Language which yields a completely new plan, or a "FIX" expression which specifies a new first move for the plan in the goal and will cause a modified version of the goal's plan to be created (the exact modification depends on the plan in the goal).

In addition to the above actions for posting concepts, there are a few miscellaneous actions. The RETURN expression takes no arguments and causes the KS being executed to stop after the current production is finished. No more productions will be tried on that particular goal although the same KS may try to execute other concepts. Numerous methods of controlling the execution flow can be imagined, but PARADISE has not found them necessary. RETURN is not necessary, rather it is used for efficiency. When a production knows that no other production in the KS can match, then it does a RETURN. For example, in PARADISE's MOVE KS (described in chapter 3) all productions which analyze safe moves come before productions which analyze unsafe moves. The last "safe" production therefore does a return so the system will not try to match all the other productions and fail. If the ordering of productions in a KS was a hindrance, all RETURN expressions could be removed and PARADISE would still produce the same result (although slightly more slowly).

The last expression to be described is the SET expression. This gives the ability to form sets (lists) as the value of variables in a production. The only category of variable which may have a set as its value is the ATTRIBUTE category. SET is used primarily to construct values for use in attribute lists. SET expects four arguments: a variable name, a list of variables distinct from the variables of the enclosing production, a list of expressions in the Production-Language, and a specification of an attribute value. SET finds all instantiations of the given variables for which the given Production-Language expressions match. It then evaluates the attribute value specification in terms of each instantiation and forms a list of these values as the value of the given variable. The attribute value specification should include a reference to at least one of the variables in the list given to SET.

7) Summary

All expressions in the Production-Language have been described. A production (or pattern) consists of a list of variables and a list of expressions in the Producion-Language which instantiate these variables. Accessing information about the current chess position uses the PATTERN and FUNCTION expressions which look up pattern matches in the data base and execute functions, respectively. Various logical connectives are provided for combining these expressions. A number of rather ad hoc expressions exist for accessing concepts and the values in their attribute lists. There are five expressions for posting concepts and plans in the data base, an expression for forming sets, and an expression for stopping execution of a KS after the current production.

There are two major considerations which influenced the design of the Production-Language. The first is the need to make the productions modular and easy to write and modify. Without a production language whose meaning is fairly transparent, it would be nearly impossible to build a large and expert knowledge base. This accounts for the straightforward implementation of most ideas and the lack of special features which could make the matching process more efficient at the cost of less transparency in the productions.

The second major consideration, which is antithetical to the first, is the fact that PARADISE is performance oriented. It is trying to find the best move in a chess position using a very small search tree. A large amount of effort is spent analyzing each position, so the system cannot afford to use the extra execution time which might be involved in using a general pattern language. For this reason the Production-Language is very ad hoc, each expression being added as it was needed to express some bit of knowledge in PARADISE. There has been no attempt to cover any domain or make a general language. Thus the Production-Language presented here should not be looked upon as a finished object but as a snapshot of the Language at a particular stage of its development.

It is easy to think of useful additions to the Production-Language. Such additions range from conditional expressions (which could easily be added to PARADISE) to type 3 patterns (which bring up issues that will be left for another thesis). Currently, such additions are just not necessary. PARADISE has been able to do a satisfactory job with the language presented here.

C) Plans in PARADISE

1) Introduction

The goal of the static analysis process in PARADISE is to produce plans. Plans seem to play an important role in the problem solving process of move selection for good human chess players. Alexander Kotov in his book *Think Like a Grandmaster* ([Kotov71]) writes:

> *. . . It is better to follow out a plan consistently even if it isn't the best one than to play without a plan at all. The worst thing is to wander about aimlessly. (p. 148)*

Most computer chess programs certainly do not follow Kotov's advice. They treat each position they analyze in a tree as an independent problem. Programs such as CHESS 4.7 can perform quite well without plans since they rely on search and not on chess knowledge. On the other hand, a knowledge based program must be able to formulate plans to justify its existence. The cost of processing the knowledge should be offset by a significantly smaller branching factor in the tree search. Plans help reduce the cost of processing knowledge by immediately focusing the program's attention on the critical part of a new position, thus avoiding a complete re-analysis of this new position. Having plans also reduces the branching factor in the search by giving direction to the search. Consider, for example, that most programs may try the same poor move at every alternate depth in the tree, always re-discovering the same refutation. PARADISE can detect when a plan has been tried earlier along a line of play and avoid searching it again if nothing has changed to make it more attractive. Most programs also suffer because they intersperse moves which are irrelevant to each other and work towards different goals. PARADISE avoids this by following a single idea (its plan) down the tree.

These are some of the reasons why plans are of central importance to a knowledge based chess program. This section describes the structure of plans in PARADISE. Plans can be considered as special instances of concepts. Each plan has a list of attribute lists like a concept does, a concept name which is either INITIAL-PLAN or FINALPLAN, and an expression in the Plan-Language. The Plan-Language expression takes the place of variable instantiations in a concept. To illustrate the various elements involved in plans, the following plan will be used as an example throughout this section. It is one of the two plans PARADISE most highly recommends in position 2.1.

```
(((WN N5)
   (((BN N4) (SAFEMOVE WR Q7)
       (((BK NIL) (SAFECAPTURE WR BR)) ((ANYBUT BK) (SAFECAPTURE WR BK))))
   ((BN N4) (CHECKMOVE WR Q7) (BK NIL) (SAFECAPTURE WR BQ) )))
 ((THREAT (PLUS (EXCHVAL WN N5) (FORK WR BK BR))) (LIKELY 0))
 ((THREAT (PLUS (EXCHVAL WN N5) (EXCH WR BQ))) (LIKELY 0)))
```

The first four lines are the Plan-Language expression for the plan. Part 2 of this section describes the Plan-Language. Line five is the first attribute list of the plan and line six is the second attribute list. Part 4 of this section describes what knowledge is necessary to evaluate a plan and how it is stored in the attribute lists. Part 5 sketches some of the issues involved in specifying plans and part 6 compares these plans to those of other systems. For the reader not interested in details, parts 5 and 6 will most readily communicate the issues involved in plans. In the above plan, the internal representation in PARADISE has not been printed unambiguously since the term "WR" does not specify which white rook. In the above example it should be obvious to which piece the names refer.

2) The Plan-Language

The Plan-Language expresses plans of action for the side about to move, called the offense. In general, the offense wants to have a reply ready for every defensive alternative. A plan can not therefore be a linear sequence of goals or moves but must contain conditional branches depending on the opponent's reply. When the offense is on move, a specific move or goal is provided by the plan. When the defense is on move, a list of alternative sub-plans for the offense may be given. Each alternative begins with a template for matching the move made by the defense. Only alternatives whose template matches the move just made by the defense are tried in the search. In the plan from position 2.1, the second through fourth lines give two alternative plans which may be executed after black's first move.

PARADISE has the following templates which adequately describe defensive moves in terms of their effects on the purpose of the plan being executed (P and SQ are variables which will be instantiated in an actual plan to particular pieces and squares, respectively):

(P SQ) matches when the defense has moved the piece P to the square SQ.
(NIL SQ) matches when the defense has moved any piece to SQ.
(P NIL) matches when the defense has moved P (to any square).
(ANYBUT P) matches when the defense has moved some piece other than P.
NIL matches any defensive move.

The plan for an offensive move can be one of two things: a particular move or a goal. A particular move is simply the name of a piece and a square, e.g. (WN N5) in the example plan. Such a plan causes PARADISE to immediately make that move with no analysis of the position other than checking that the move is legal. A goal is simply the name of a KS followed by an instantiation for each argument of the KS, e.g. (SAFEMOVE WR Q7) in the example plan. When executing a goal, PARADISE will construct a concept whose name is the name of the KS in the goal, whose variable instantiations come from the goal, and whose attribute lists come from this plan as a whole. This concept is posted in the data base and its corresponding KS executes it. Any plan produced by this execution will replace the goal in the original plan and this modified plan will be used. (This process expands and elaborates plans.) If executing the KS does not produce a plan then the original plan has failed.

The plan language expression in the example above can be read as follows:

> *Play N-N5. If black captures the knight with his knight, attempt to safely move the rook on Q1 to Q7. Then, if black moves his king anywhere try to safely capture the black rook on R7, and if black moves any piece other than his king, try to safely capture the king with the rook. A second alternative after black captures the knight is to attempt to safely move the rook on Q1 to Q7 with check. Then, if black moves his king anywhere try to safely capture the black queen with the rook.*

The first alternative after black's first move comes from the production which wants to pin the king to the rook (this plan also has a branch after black's second move). The last alternative comes from the production which wants to capture the queen with check. After playing N-N5 and NxN for black in the tree search, PARADISE will execute either the CHECKMOVE goal or the SAFEMOVE goal. Executing the SAFEMOVE goal executes the SAFEMOVE KS (described in chapter 3) which knows about safely moving a piece to a square. This KS will see that R-Q7 is safe and produce this as the continuation of the original plan, causing the system to play R-Q7 with no further analysis. If for some reason

R-Q7 was not safe then the SAFEMOVE KS will try to make Q7 safe for the rook (by posting a SAFE concept). If some plan is found, the original plan will be "expanded" by replacing the SAFEMOVE goal with the newly found plan. In general, the newly found plan will contain (SAFEMOVE WR Q7) though it may come after a sacrificial decoy or some other tactic. If posting the SAFE concept does not produce a plan and the alternative CHECKMOVE goal fails also, then the original plan has failed and a complete analysis of the position must be done. If black had answered N-N5 with a king move, it would not match the template (BN N4) and a complete analysis of the position would be undertaken without attempting to make Q7 safe for the rook.

When specifying plans in a production, any variable or Arg-Spec may be used for any of the particular pieces or squares, and a whole subplan may be retrieved from the attribute lists of the goal or from other concepts.

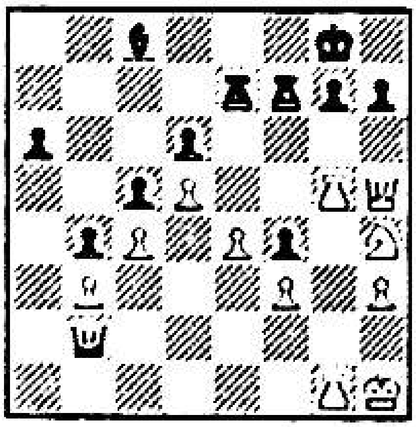
The most *important* feature of the Plan-Language is that offensive plans are expressed in terms of the KSes. Important advantages obtained by doing this are listed below:

> -The relevant productions are immediately and directly accessed when a new position is reached.
>
> -The system has one set of abstract concepts it understands, namely the KSes. It does not need to use a different language for plans and static analysis.
>
> -By use of conditionals, a plan can handle a number of possible situations which may arise.
>
> -The range of plans that can be expressed is not fixed by the program. The system has been designed to make the writing of productions (and thus forming KSes) reasonable to do. By forming new KSes, the range of expressible plans may be increased without any new coding (in the search routines or anywhere else) to understand the new plans. This is important since the usefulness of the knowledge base is limited by its ability to communicate what it knows (e.g., through plans).

One weakness in the way plans are represented is that alternative subplans are not associated with their own attribute lists. As each alternative subplan is produced, PARADISE uses its attribute list to sort this alternative with the others so that the most promising one will always be first. However, the attribute lists of this alternative are then grouped together at the end of the plan with the attribute lists of all the other alternatives. This is done so that plans will look like concepts and may be manipulated by the same routines. The alternative is to build a tree to represent a plan where attribute lists may occur many places in the tree. Using the concept-like representation instead of this tree, PARADISE loses the ability to reorder alternative subplans deeper in the tree by re-evaluating expressions in the original attribute lists. (It still reorders them on the basis of features in the current position.) The error that may result is a non-optimal ordering of alternatives on the offense's second or third move in a plan. In practice it is rare that the original order of alternatives can be changed merely by re-evaluating the original expressions in the attribute lists.

3) An example search from PARADISE

Figure 2.3 shows position 49 from [Reinfeld58] and the plan PARADISE suggests as best after a static analysis. The protocol presented below is a summary of the actual search generated by PARADISE in solving this problem. Most of the details are omitted, and the particular KSes mentioned are described in detail in chapter 3. This search shows how a plan guides the search so that static analyses are not necessary.



```
(PLY 1)
BESTPLAN
(((WQ R7) (BK R2) (CHECKMOVE WR R5) (BK NIL) (ATTACKP BK))
((NINT 1) (THREAT (PLUS (WIN BK) (EXCHVAL WQ R7))) (LIKELY 1)))
```

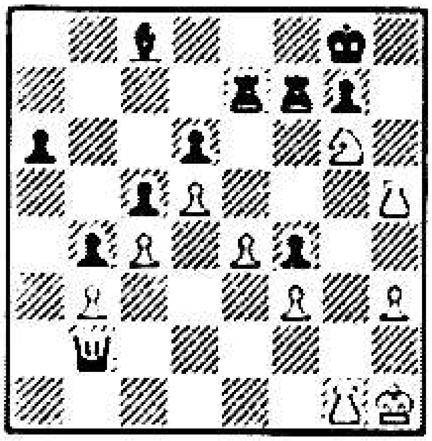
a plan produced by PARADISE

white to move

Figure 2.3

The search commences execution of this plan by playing QxPch and producing a new board position. The defense suggests both legal moves, KxQ and K-B1, but tries KxQ first. Since this move matches the (BK R2) template in the plan, PARADISE has ((CHECKMOVE

WR R5) (BK NIL) (ATTACKP BK)) as its best plan at ply 3 of the search. Execution of the CHECKMOVE KS produces ((WR R5) (BK NIL) (ATTACKP BK)) as a plan which causes R-R5ch to be played. Black plays his only legal move at ply 4, K-N1, which matches the (BK NIL) template in the plan. Thus PARADISE arrives at ply 5 with (ATTACKP BK) as its plan. Executing the ATTACKP KS posts many concepts, including MOVE and SAFE concepts, and ((WN N6) ((NIL (SAFEMOVE WR R8)) (NIL (SAFEMOVE WN K7)))) is produced as the best plan in this position. After playing N-N6, the position in figure 2.4 is reached.



(PLY 6)

INITIAL DEFENSIVE MOVES

((BB R6) (BB Q2) (BB N2) (BR B4) (BR B3) (BR B1))

Figure 2.4

Productions which know about defending against R-R8 suggest the 6 black moves in figure 2.4. Black plays BxP first in an effort to save the bishop from the ensuing skewer. At this point the plan branches. Since both branches begin with a null template, they both match any black move at ply 6. Thus PARADISE has two plans at ply 7: (SAFEMOVE WR R8) and (SAFEMOVE WN K7). Before executing a plan for the offense, PARADISE executes the QUIESCENCE KS which looks for obviously winning moves. Here R-R8 is suggested by the QUIESCENCE KS which causes the (SAFEMOVE WR [illegible] plan to be executed immediately. PARADISE plays R-R8, finds that black is mated, and returns to ply 6 knowing that black's BxP leads to mate. All this has been accomplished without a static analysis; the original plan has guided the search.

At ply 6, black uses KSes to refute the mating line. No new moves are found since all reasonable defenses have already been suggested. PARADISE has a causality facility (described in chapter 4) which determines the possible effects a move might have on a line of play. Using the tree generated for the mating line, the causality facility looks for effects a proposed move might have (such as blocking a square which a sliding piece moved over, vacating an escape square for the king, protecting a piece which was attacked, etc.). The causality facility recognizes that neither B-Q2 nor B-N2 can affect the mating line found for BxP so they are rejected without searching. Black plays R-B4 next, the causality facility having recognized that this move opens a flight square for the black king.

Again PARADISE has both (SAFEMOVE WR R8) and (SAFEMOVE WK K7) as plans at ply 7. Both SAFEMOVE goals would succeed, but R-R8 has a higher recommendation and is played first. Black plays his only legal move at ply 8, K-B2. The original plan has finally expired and PARADISE no longer has a plan at ply 9. Before trying a static analysis to find a plan, the program executes the QUIESCENCE KS in an attempt to find an obviously winning move. R-KB8 is suggested and PARADISE immediately plays this move without doing a static analysis. This is mate so PARADISE returns to ply 6 to look for other defenses. Both R-B3 and R-B1 are tried and both are quickly refuted by playing R-R8 from the SAFEMOVE goal of the original plan, K-B2, and R-KB8 from the QUIESCENCE KS.

The search then returns to ply 2 and tries K-B1 in answer to QxPch. The template in the original plan does not match K-B1 so there is no plan at ply 3, and PARADISE avoids playing R-R5 which is now not the best move. A static analysis is avoided however, because the QUIESCENCE KS quickly suggests Q-R8 and PARADISE returns from the search convinced that QxPch will mate.

PARADISE's plans are not always so accurate but this is a good example of how a plan guides the search. The above analysis uses about 130 seconds of cpu time on a DEC KL-10. It goes to a depth of 9 ply while creating only 21 nodes in the tree. Because of the guidance provided by the original plan, no static analysis was performed except on the original position. By comparison, the TECH2 program plays N-B5 at a depth setting of 4. At a depth setting of 6 it decides that QxPch is best, but it still does not realize that it can win because the horizon effect hides the mate from it. For the depth 6 analysis, TECH2 uses 210 seconds of cpu time on a KL-10 and creates 439,459 nodes by making legal moves (as well as making 544,768 illegal moves which are retracted).

4) Evaluating the worth of a plan: the Threat-Language

Given a number of plans to execute, the tree search must make decisions about which plan to search first, when to forsake one plan and try another, when to be satisfied with the results of a search, and other such things. To make such decisions, it must have information about what a plan expects to gain and why. Such information was available to the productions which matched to produce the plan since they were aware of the factors suggesting the plan. Since these productions are no longer around, they must communicate this knowledge (of which there is a significant amount) in the attribute lists of the plan. This must be done so as to provide many different types of access to this knowledge.

This problem is not serious in chess programs which rely on search. They have so little information about a move that accessing it is no problem. Static analysis in such programs usually assigns each move one integer which is a score representing the estimated value of the move. That integer is used primarily to order the moves at that node. Attempts to use this integer for more sophisticated purposes like forward pruning has usually resulted in worse performance due to the lack of information in the integer. Using this integer at descendant nodes in the tree has not even been attempted. PARADISE must, however, use its information about a plan at other nodes in the tree since it executes a plan without analyzing the newly created positions. By comparing one plan to another, this information is also used to eliminate plans from consideration.

To use the information about plans in an effective manner, PARADISE divides a plan's effects into four different categories which are kept separately (i.e., their values are not combined in any way). These four categories are: THREAT which describes what a plan threatens to actively win, SAVE which describes counterthreats of the opponent a plan actively prevents, LOSS which describes counterthreats of the opponent not defended against and functions the first piece to move in a plan will give up by abandoning its current location (thus providing new threats for the opponent), and LIKELY which describes the likelihood that a plan will succeed. Since the values for these categories must evaluate correctly at different nodes in the tree, they cannot be integers. Instead they (except for LIKELY) are expressions in the Threat-Language (described below).

These expressions are evaluated in the context of the current position and may return different values for different positions. The value of LIKELY is an integer. This integer represents the number of unforced moves the defense has before the offense can accomplish its plan. The most likely plans to succeed have a LIKELY of zero (every move is forcing). Both attribute lists in the example plan have a likelihood of zero.

These four categories have emerged during the development of PARADISE. Experience seems to indicate the system must have at least these four dimensions along which to evaluate plans in order to adequately guide the tree search. A few remarks may make them seem more reasonable. The LIKELY concept is very important to the system. In general, there are many tactics that rarely work but must be recognized to correctly handle the few positions where they do work. (Workability cannot always be determined beforehand; for example, whether or not a mating net works may depend on the location of a lowly pawn at the opposite end of the board.) To avoid wasting a lot of effort searching plans that probably won't work, PARADISE searches first the plans which are most likely to succeed even if they do not threaten as much. The value threatened must however be kept seperately and accurately since it is used to determine when the plan has succeeded and to make forward prunes. One also wants LOSS separate from THREAT. For example, if mate is threatened, it does not matter what is lost, but one would not want to threaten a pawn with a move that will lose a rook. The SAVE attribute is treated differently than the others since it (loosely speaking) measures the strength of a move which is merely trying to maintain the status quo. Thus it would not be used in calculating the value a plan should achieve for success, but it would influence the decision about which move should be tried first.

The values of SAVE, LOSS, and THREAT are expressions in the Threat-Language. The following are the basic expressions in the Threat-Language which can be evaluated to get integers:

Integers which evaluate to themselves.

(WIN P) which evaluates to the value of the particular piece P. This value is zero if P is not present on the board.

(EXCHVAL P SQ) which gives the value P can win or lose by starting an exchange on the square SQ. P may or may not be able to legally move to SQ. EXCHVAL will calculate what part of the value of P would be lost or what part of the value of the occupant of SQ would be won. This is done using the same calculation SAFEP used to determine if SQ was safe for P (uses occupiabilities with a special approximation for the case when P is not the optimal piece for his side to deploy to SQ). If P is not present on the board, EXCHVAL returns zero.

(EXCH P1 DP1) which returns EXCHVAL for P1 moving to DP1's location. Zero is returned if either piece is not present on the board. Once again, P1 may not be able to legally capture DP1.

(BOTH P1 DP1 P2 DP2) which calculates (EXCH P1 DP1) and (EXCH P2 DP2) and returns the the minimum of the two.

(FORK P1 DP1 DP2) which calculates (EXCH P1 DP1) and (EXCH P1 DP2) and returns the minimum of the two. This is just a shorthand for (BOTH P1 DP1 P1 DP2).

The Threat-Language has four connectives for combining threats. The connectives all take expressions in the Threat-Language as their arguments. Thus they can be applied to any of the above expressions or to any expressions formed by using these four connectives. The PLUS connective takes two threats as arguments and returns their sum. The LESS connective takes two arguments and returns their difference. The FORKTHREAT connective takes two arguments and returns the minimum of the two. Finally, any number of threats may be concatenated into a list where the list structure acts as an implicit connective. Such a list returns the maximum of all its arguments.

The example plan for position 2.1 had the following Threat-Language expression as the value of THREAT in the first attribute list:

```
(THREAT (PLUS (EXCHVAL WN N5) (FORK WR BK BR)))
```

This adds the threat of exchanging the knight on N5 to the threat to be made by the white rook on Q7, which is expressed using FORK since the rook will exchange itself for either the king or the rook. The threat of exchanging the knight on N5 is zero since white can safely answer NxN with QxN. Note that if the white rook was not on KB1 then this value would be negative. Also note that if there were a piece being captured on N5, this would automatically be accounted for (EXCHVAL would return the value of the piece) in the threat value, thus relieving productions of the necessity of checking for such threats explicitly. The THREAT attribute in the second attribute list of the original plan represents the threat of capturing the black queen after getting the white rook on Q7. PARADISE assumes that black will capture the knight on his first move. When it is less sure of black's reply, the THREAT is sometimes expressed using FORKTHREAT where the first argument would be the above threat and the second argument would be an expression for the threat made by the knight on N5.

The Threat-Language just described has a number of advantages, the following are the more important ones:

> -The sophisticated analysis of OCCUP used in EXCHVAL gives very accurate values. However, productions must be aware that perturbations in the current position change the assumptions that OCCUP used. Thus some situations require the use of WIN to keep the threat value optimistic.
>
> -The ability described above of EXCHVAL to "automatically" account for the capturing of pieces on squares being moved to, saves the productions a significant amount of effort.
>
> -The use of Threat-Language expressions instead of integers allows accurate re-evaluation of a threat at other positions in the search tree. For example, if the defense (on its first move) captures the piece which the offense wanted to move on his second move, then any EXCHVAL or EXCH in the threat which is applied to this piece will now evaluate to zero, reflecting the fact that the piece is no longer present to make those threats.

The chief weakness in the Threat-Language is the inability of EXCHVAL to give correct values for positions produced by moves which affect the square in question. For example, if we attack the queen by capturing the only piece which protected it, then we are threatening to win the queen outright and not exchange our piece for it. As mentioned earlier, productions must use the WIN expression to prevent these errors. The solution to this problem lies in redoing the OCCUP calculation with the line up of pieces altered. This is too expensive to be considered at present, although Berliner does this for selected squares in his CAPS program.

To illustrate the use of the four dimensions along which plans are evaluated, the function which orders plans for a normal tree search will be described. This same function is used to access the "best" attribute list in Production-Language expressions. The overriding consideration is the likelihood of success of a plan. Plans with better likelihoods come first regardless of the values of THREAT, SAVE, and LOSS (unless LOSS outweighs both THREAT and SAVE). If two plans have the same likelihood then the idea is to score them by adding the values of THREAT and SAVE together and subtracting the value of LOSS from that. However, if the opponent pursues his threat represented by LOSS instead of defending against the threat represented by THREAT, then THREAT may become more threatening. Thus the system must be able to calculate the value of the threat when no defending moves are made.

This is done by replacing every EXCHVAL and EXCH by the corresponding WIN during evaluation of the threat expression, and by having FORK, FORKTHREAT, and BOTH take the maximum of their arguments rather than the minimum when they occur at the top level of a threat expression. Such an evaluation ensures that an optimistic value will be produced. (It may be overly optimistic, but to prevent this the system must compare individual LOSS threats to individual THREAT threats and determine how they affect each other.) In comparing plans, PARADISE then makes two calculations for each one: first it computes the sum of the normal evaluations of THREAT and SAVE, and second it adds the normal evaluation of SAVE to the "no defense" evaluation of THREAT and subtracts from this the normal evaluation of LOSS. It then uses the smaller of these as the value of the plan since the defense has the choice of countering the threat in THREAT or playing the threat in LOSS. If the value computed in this manner still does not differentiate the two plans, various other considerations come into play.

This method of ordering plans is only one of many possible ways. In fact, the quiescence search in PARADISE uses a different ordering function. A different ordering function would also be appropriate when the system is in a defensive state rather than an offensive state. It should also be noted that the value obtained above for a plan need not be used by the search, which can create its own combination of THREAT, SAVE, and LOSS to determine expectations, forward prunes, and other such things.

6) How detailed should plans be?

Searching is expensive in PARADISE because of the sophisticated analysis done at many nodes, so it is very important to avoid spurious branches. To do this the system must quickly recognize when a plan is not working and abandon it before poor lines are investigated. On the other hand, it is also very expensive to completely analyze a position so the system wants to use its plan as long as possible. To achieve a balance, plans must adequately express the purpose they intend to achieve. Productions must carefully formulate their Plan-Language expressions at the right level of detail using appropriate goals and descriptions of defensive moves. When executing a plan, the system will retrieve a goal or move from the plan to apply to a new position. Care must be taken to ensure that this move or goal is reasonable in the new position. Plans may become unreasonable when they are too general or too specific. In either case, they have failed to express their purpose. The following example illustrates the issues involved.

Consider the first three ply of the plan for position 2.1, namely ((WN N5) (BN N4) (SAFEMOVE WR Q7)). Suppose this had been expressed as ((WN N5) NIL (WR Q7)). Playing R-Q7 after NxN is reasonable but if black moves his king instead, playing R-Q7 will lead the search down a blind alley. Thus, (WR Q7) is too specific and does not express white's purpose. This problem can be cured by a more specific description of black's reply. For example, the plan ((WN N5) (BN N4) (WR Q7)) will not be misleading when black does not play NxN since the template for black's move will not match. In actual fact, PARADISE does not know for sure that R-Q7 will be safe after N-N5 and NxN (although in this particular position it is). To avoid mistakenly playing R-Q7 when it is not safe, PARADISE uses a SAFEMOVE goal to more accurately express the purpose of the plan. (Goals such as SAFEMOVE can be used to verify that a certain state has been reached.) For example, the plan could be ((WN N5) NIL (SAFEMOVE WR Q7)). If black answers N-N5 with NxN then the SAFEMOVE goal is reasonable and quickly produces R-Q7 as the move to play after verifying its safety. If black answers with a king move then the SAFEMOVE goal is not what white wants to do, but little is lost since R-Q7 is not safe and the line will be rejected without searching (plans such as B-R4 for making Q7 safe are rejecting since they are not likely to succeed). However, if likely plans had existed for making Q7 safe, the search may still have been led astray. For this reason, PARADISE uses (BN N4) as the template in this plan. The purpose of N-N5 is to decoy the black knight to his KN4 and this template most accurately expresses this purpose.

Now let us consider the following more general plan: ((WN N5) NIL (SAFELY-ATTACK BK)). If black answers with NxN then the SAFELY-ATTACK KS should generate R-Q7 as a safe attack on the black king. If, instead, black moves his king then this KS should generate a check by the white queen or knight which would also be reasonable. Thus this plan produces reasonable moves for every black reply without ever causing a re-analysis of the new position. This would be a good plan if PARADISE knew (from its patterns) that it could get the black king in trouble after any reply. However, it only knows of the skewer of the black king to the rook, the threat of capturing the queen with check, and the fact that the white knight threatens the black king, rook, and knight from N5. It is accidental that the SAFELY-ATTACK goal works after black retreats his king. In the general case, such a goal would produce many worthless checks that would mislead the search. Thus this plan is too general to describe white's actual purpose.

It is very important to get the correct level of detail in a plan. The plan should handle as many replies as possible without causing a re-analysis, but it should avoid suggesting poor moves. The templates for describing defensive moves in the Plan-Language and the various KSes have been developed to allow PARADISE's plans to accurately express their purpose. The results have been quite satisfying: the productions in PARADISE now create plans which rarely suggest poor moves but which can still be used for as many ply as a human might use his original idea.

6) Comparison to other systems

a) Plans in robot problem solving

Plans are used in many domains of AI research. For various applications, comparisons between other systems and PARADISE are not worthwhile because of the difference in domain and/or the difference in the idea behind the word "plan". This section compares plans in robot problem solving (probably the most popular domain for systems using plans) to those in chess. The next section compares plans in PARADISE to those in Pitrat's system [Pitrat77], which uses plans to solve chess combinations.

ABSTRIPS [Sacerdoti74], NOAH [Sacerdoti75], and BUILD [Fahlman74] are examples of planning systems in robot problem solving environments. Plans in these systems are very different from those in chess so this section concentrates on a general description of these differences. The last paragraph in this section describes the ideas in PARADISE that may be useful in the robot planners.

Robot planning is frequently done in abstracted spaces which have been defined. These abstracted spaces omit details such as whether a door is open or not. There are a small number of such details, and the problem solver has some specific knowledge for each such detail which it can apply when required. For example, the problem solver has a procedure for opening a door should the door be closed when the robot wants to go through it. Such an approach is much more difficult in chess, at least with mankind's current understanding of the game. It is not clear how to define abstracted spaces. Small details are very important in chess and cannot be readily ignored, even for determining the first step of a plan. There is no set of "details" in chess which can be dealt with later using specific procedures. For example, a chess planner should not ignore the fact that a piece is pinned on the assumption that it can be unpinned when the plan calls for it to move. Thus plans in PARADISE are all at the same detailed level.

In the robot planning systems, the effects of an action are well defined. The program can easily and quickly determine the exact state of the world after it has moved a blue block. In chess, moves (actions) may subtly affect everything on the board. For example, a piece that was safe may no longer be (even though the move made has no direct effect on the piece), and the system may need to make an expensive calculation to determine this. Chess plans cannot make many assumptions about the state of the world in the future, but must describe the expected features of new states in the plan itself and then test for these features while executing the plan. Perhaps the monkey and bananas problem will help make this point. If the experimenter is sinister, he could connect the bananas to the box by pulleys and rope, invisible to the monkey, so that when the box is pulled under the bananas, the bananas are pulled up so they cannot be reached from atop the box. In a sense, the robot planners assume that no such crazy side effects will happen, while a chess plan must prepare for such things. This manifests itself in PARADISE in at least two ways. First, plans are not composed of simple actions but are goals which may require the knowledge in many productions to be interpreted. This postponement of evaluation permits checking of many complex features in the current world state. Second, PARADISE has been designed to make it easy to produce new concepts for expressing plans whenever they are needed to handle new "crazy side effects".

In most robot planning systems, tests for having achieved the goal or the preconditions of an action are trivial and give well defined answers. In chess, there may be only subtle differences between a position where a particular action is good and positions where the same action is wrong. It is also hard to know when a goal has been achieved, since there is always a chance of obtaining a larger advantage (except when the opponent has been checkmated). Thus in chess it is necessary for the plan to provide a considerable amount of information to help in the making of these decisions. This is done in PARADISE through the attribute lists in plans.

The number of things PARADISE can consider doing at any point in a plan is about an order of magnitude larger than the number of things most robot problem solvers usually contend with. There are an average of about 38 legal moves in a chess position (see [deGroot65]), so the planner has 38 reasonable choices on the very first step of a plan. (It is hard to eliminate moves as being unreasonable as is evidenced by the fact that programs like CHESS 4.7 and KAISSA use full width searches.) A robot usually has a much smaller number of possible actions which cannot be easily eliminated (for example, going to one of a few rooms or picking up one of a few objects). The problem becomes more acute when planning farther in the future than just the first action since the chess planner then has many more choices than the 38 legal moves. If the robot wants to open a door as its second action, the plan can simply say that, but in chess the inability to make assumptions about future states prevents this. If the plan is to trap the opponent's queen, then no particular move can be used to express this as the second part of a plan since the actual trapping move will depend on where the queen has moved in the interim. Instead the idea of trapping the queen must be expressed as part of the plan. The number of such ideas that will be useful is much larger than the number of legal moves that might be made. This leaves the chess planner with a large number of alternative choices and a planning problem that explodes at a very different rate than the robot planning problem.

Lastly, a chess system must have many ways to modify plans during execution. PARADISE modifies plans in the following ways: 1) a goal in a plan may be elaborated into a multistep subplan, 2) a whole plan may be inserted in the middle of another, 3) alternatives in a plan may be reordered, 4) parts of a plan may be deleted, and 5) plans may be concatenated if they are working towards the same objective. This represents a richer set of modifications than those used in most robot problem solving planners. Such systems may try to reorder actions and further specify parts of a plan, but most will not insert a plan in the middle of another.

To summarize, the robot planners have the flavor of establishing a sequence from a small number of well-understood operations until the correct order is found, while chess planners have more a flavor of "creating" the correct plan from the many possibilities. For this reason, a system which produces good chess plans needs a large amount of knowledge and non-trivial reasoning processes to produce plans.

Probably the most important idea PARADISE has for the robot problem solvers is that a plan may be viewed as a way to control what parts of a large knowledge base will be used to analyze each situation during plan execution. Plans in PARADISE can be viewed as telling the system what knowledge to use in analysis. In most robot planning systems plans are more accurately viewed as telling the system what action to take. By describing what knowledge to use in a new situation, plans in PARADISE can be viewed as defining a perspective for the system to use in new situations. Some amount of analysis from this new perspective is almost always done before the plan can continue executing. Thus PARADISE tries to delay execution in order to bring more knowledge to bear.

b) Plans in chess

Pitrat's program [Pitrat7. which solves chess combinations, is the most important example of the use of plans in chess. The language Pitrat uses for expressing plans has four statements with the move statement and modification statement being the most important. The move statement specifies that the piece on one particular square should move to another particular square. It may also be specified that the move must be a capture. The modification statement describes a modification to be made to a particular square. This can be one of four things: removal of a friend, removal of an enemy, moving a friend to the square, or getting an enemy to move to the square. The "friend" (or "enemy") may be any piece of the correct side or it may be specified what type of piece in should be (e.g., queen, bishop, etc.), but a particular piece cannot be specified unless it is the only piece of that color and type on the board.

Plans in PARADISE provide much more flexibility in expression than Pitrat's move statement. In PARADISE, types of moves other than captures can be specified (e.g., safe moves and safe capture moves). Also, particular squares do not need to be named in PARADISE since it can use a variety of goals to express the plan. The important square to move to may change depending on the opponent's move so it is not always possible to specify the square to move to in advance. For example, if white makes a move which traps and attacks black's queen then (after some desperado move by black) white would like to capture black's queen wherever it may be. PARADISE can express the plan of capturing the queen anywhere, while Pitrat's move statement cannot since the planner does not know which particular square the queen will be on.

Pitrat's modification statement is a goal which the system tries to accomplish. As the following example shows, these modifications are too simple to express the purposes behind their suggestion. Suppose white wants to get his queen from Q1 to Q8 in order to mate, but there is a white rook on Q4 blocking the way and the black queen on K7 protects Q8. What white wants to do is move his rook to clear the queen file and (hopefully at the same time) decoy the black queen from protecting Q8. A plan in Pitrat's language would specify that Q4 should be made vacant by removing a friendly man, that K7 should be made vacant by removing the enemy queen, and that the white queen should move to Q8. This does not express the purpose of any of the moves. It will work well for the winning combination but will also allow many wrong moves. For example, Q4 can be made vacant by moving the white rook along the queen file, and K7 can be made vacant by decoying the black queen to some square from which it still protects Q8. In the latter case, Pitrat's plan would still specify moving the white queen to Q8 where it would be lost.

PARADISE can avoid this since it has the ability to express its purposes. It can specify the goal of clearing the queen file and only rook moves along the fourth rank would be considered. It can specify that the black queen must be decoyed to get the white queen to Q8 and only decoys which remove the black queen's protection will be considered. Lastly, PARADISE would specify that the white queen's move to Q8 should be safe so this move will not be tried unless the preparations have accomplished their purpose. Instead of expressing the purpose behind the plan, Pitrat's modification statement expresses side effects that will happen if the purpose is accomplished. For example, Q4 becoming vacant is a side effect that happens when the rook clears the queen file for the queen. Unfortunately, it is also a side effect of many rook moves which do not clear the queen file, causing the system to waste effort searching poor lines.

The two remaining statements in Pitrat's plan language are the verification statement which simply tests if the desired state has been reached after attempting a modification (e.g., testing that Q4 is vacant), and the limited analysis statement which tells the system to do a limited analysis where it looks for simple combinations which attack one particular piece.

In summary, PARADISE has enough power of expression to express the purpose behind its plans. As the discussion in section 4 pointed out, this is necessary to quickly reject bad lines and keep the branching factor small. It is not enough to look for side effects that will be present after successful execution of the plan since bad moves may also leave those side effects.

Plans in PARADISE have two more major advantages over the plans in Pitrat's system. First, PARADISE has conditionals which allow specification of different plans for different replies by the opponent. The advantages of this are obvious: the system can immediately try the correct plan instead of searching an inappropriate plan and backtracking to correct itself. Second, PARADISE's plan language is modifiable. Simply by writing new productions, new goals and concepts can be created for expressing plans and the system will automatically understand these new plans. This property is necessary for any system that wishes to extend its domain or incrementally increase its expertise. Significant additions to Pitrat's plan language would seem to require a major programming effort.

Despite the shortcomings of the plans in Pitrat's program, they are much more sophisticated than any plans previously used in computer chess programs. It should be noted that Pitrat's program performs well in its domain. It processes nodes much faster than PARADISE and therefore can handle larger trees. The plans do an adequate job given these constraints. It may be the case that programs will obtain better performance by using Pitrat's approach of larger trees and less sophisticated plans, but the use of knowledge is only starting to be investigated by programs such as PARADISE and it is too early to draw conclusions. This issue is discussed in more depth in chapter 7.

CHAPTER III

An Actual Static Analysis in PARADISE

A) Introduction

There are currently twenty-eight KSes in PARADISE which contain about two hundred productions. Both these numbers grow as new rules are written to teach PARADISE new tactics. To show the overall organization of the knowledge base, each KS in PARADISE is described in the next section. Figure 3.0 attempts to show how strongly the KSes in PARADISE interact. This is not important to understanding PARADISE, so the reader not interested in these details may skip to the end of this section.

Every KS in PARADISE is in figure 3.0. An arrow from an "origin-KS" to a "destination-KS" means that some production in the origin-KS (when it matches) posts a concept corresponding to the destination-KS. Thus the reasoning processes in the origin-KS use the abstract concept provided by the destination-KS. Each KS depends on all the KSes to which it points to provide the abstract concepts it needs for its reasoning. (The former has expectations of the latter which the latter must meet.) Destination-KSes have no expectations about the KSes that point to them; they rely only on the information in concepts that are posted in the data base. The KS which corresponds to a concept has no idea which production or which KS posted the concept. All the information about why the concept was posted is contained in its attribute lists.

Figure 3.0 has been simplified somewhat to make it legible. The notes following it explain these simplifications.

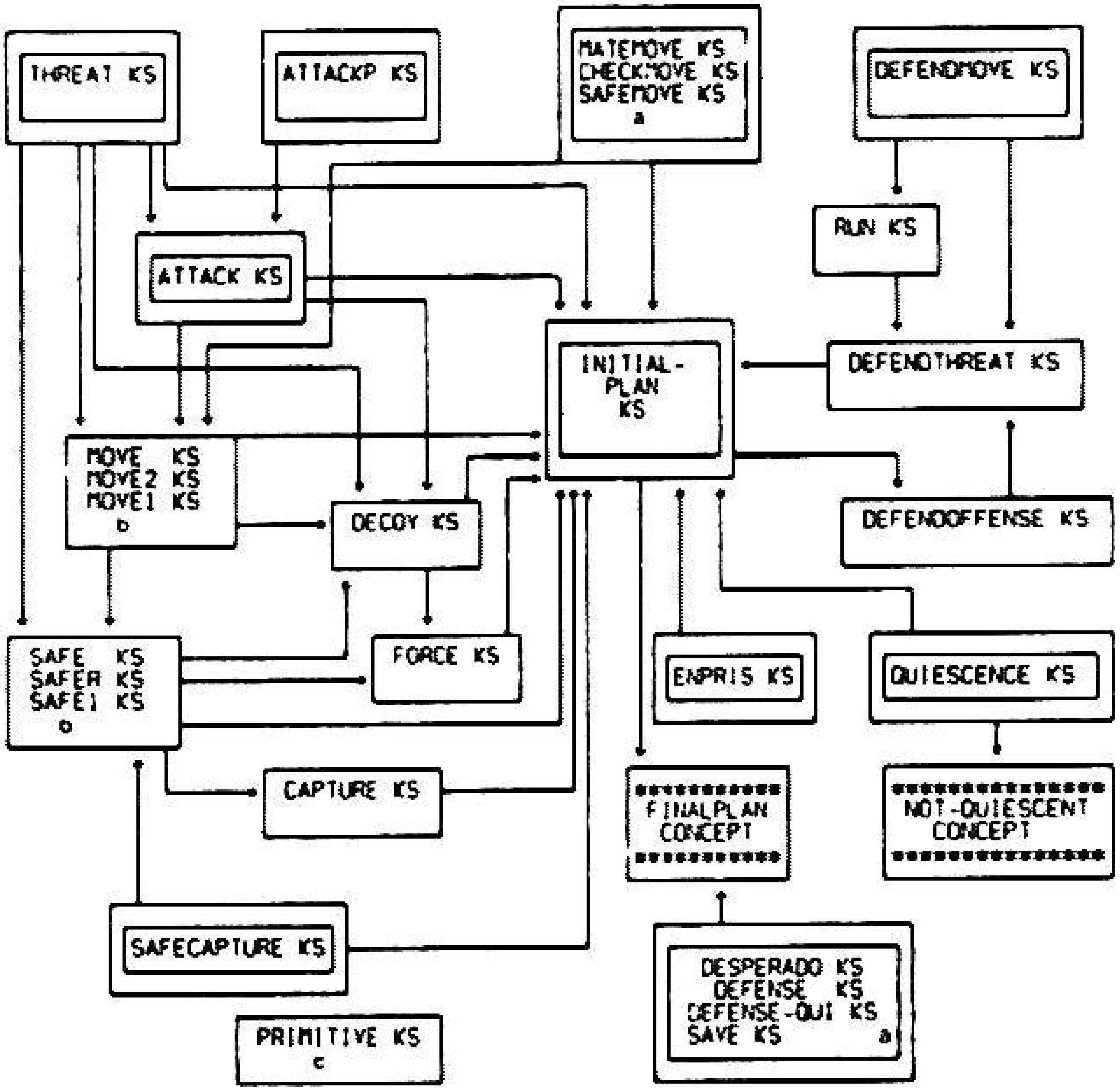


Figure 3.0
Diagram depicting which concepts are posted by each KS

Notes for figure 3.0:

a - The KSes in this box are together because they happen to have the same input and output arrows.

b - Three KSes are in this box. The first one has the other two as sub-KSes. Each KS may post instances of any KS below it in the box. Arrows into or out of the box may be referring to only one, two, or all three of the KSes. This grouping is necessary to make the diagram manageable.

c - The PRIMITIVE KS. All productions access this KS so every box with a KS in it relies on this KS. Interaction between other KSes and PRIMITIVE does not involve the posting of concepts. Rather, KSes interact with PRIMITIVE in the same way productions within one KS interact with each other.

An analysis in PARADISE consists of productions firing and posting concepts. Since the arrows in the diagram point from KSes whose productions post concepts corresponding to the KSes pointed to by the arrow, these arrows reflect the execution flow during an analysis by the program. Following the arrows through the diagram shows how PARADISE's attention might shift as it proceeds through an analysis. Boxes with no arrows leading into them can only have concepts corresponding to their KS posted by an external source. This external source is the tree search which posts concepts in order to execute plans, instigate static analyses, and request information. In the diagram, all KSes which have concepts corresponding to them posted by the tree search are surrounded by double boxes (e.g., the THREAT KS). Such KSes may also have their corresponding concepts posted by productions in other KSes, in which case there will be arrows leading into the box (e.g., the INITIAL-PLAN KS). Thus the double boxes can be viewed as "entry points" into the knowledge base, or starting points for the execution flow.

The final goal of an analysis is to produce concepts which do not correspond to KSes. In PARADISE there are currently two such concepts, the FINALPLAN and NOT-QUIESCENT concepts. In the diagram the boxes for these concepts have no arrows leading out of them. These boxes are highlighted by rows of asterisks since they represent the final goal of an analysis. Thus an analysis (or execution flow through the knowledge base) can be seen as starting from a double box and following the arrows (possibly through loops) until a box with asterisks is reached.

A brief description of each KS in PARADISE follows. (These detailed descriptions may not be useful on first reading -- it might be appropriate to read a few descriptions and skim the rest.) A description of a KS includes the following elements:

-variables for which the KS expects instantiations

-values the KS expects in the attribute lists of its goals

-KSes for which productions in the KS post corresponding concepts

-what the KS tries to accomplish and how it is used (i.e., the KS's abstract concept)

-what kinds of conditions are tested (conditions give productions in earlier KSes a way to constrain instantiations of variables in later KSes and are explained below)

B) Actual KSes of PARADISE

PRIMITIVE

This KS is treated differently than all the others. It has no variables to instantiate and is matched for both sides. The pattern matches are never deleted while the current position is in core memory. The sole purpose of the PRIMITIVE KS is to provide matches patterns for other KSes to access. The eighteen productions (patterns) currently in PRIMITIVE are described in detail in chapter 2.

THREAT COL

This is the largest KS in PARADISE, consisting of about forty productions, many of which are quite complex. (The number grows considerably as the size of the domain understood by the program grows.) The THREAT KS finds all the threats for the side represented by the variable COL (color). It does not expect any attributes in its goal and does not test any conditions. This KS is applied to the initial problem to determine the threats for the side on move. By using plans, we hope to use the THREAT KS only infrequently in the tree search. Productions in this KS post the following concepts: ATTACK, MOVE, MOVE1, SAFE, SAFER, DECOY, and INITIAL-PLAN. The following types of threats are recognized: trapped pieces, pieces that can be trapped, discovered attacks of many sorts, forks of many sorts, captures, pins, mating nets of many sorts (especially ones where a piece is sacrificed to decoy the king), ability to force a capture with check, overburdened pieces, and some threats that cannot be easily labelled.

ATTACK COL SQ3

This KS finds moves which will cause pieces of side COL to attack the square SQ3. A typical use of this KS would be to generate attacks on an opponent's piece that is trapped (SQ3 would be its location). Besides such uses in static analysis, this KS is also used to express plans in the Plan-language. ATTACK expects a THREAT attribute in the goal which describes the threat that will be made if SQ3 is successfully attacked. No other attributes are referenced except for COND (described in the next paragraph). This KS has about ten productions which analyze moves that directly attack SQ3, moves that can attack SQ3 in two moves, moves which will uncover a discovered attack in the most advantageous manner, and moves to decoy enemy pieces that are preventing attacks. INITIAL-PLAN, MOVE, and DECOY concepts are posted.

The COND attribute is a condition. ATTACK checks any COND in its goal before it suggests moving the piece P1 to the square SQ in order to attack SQ3. The value of COND will be a Production-Language expression which uses the variables P1 and SQ (other variables in the condition will already have been instantiated by the production which posted the ATTACK goal). ATTACK will not suggest the move by P1 unless the pattern in COND matches. An example use of COND would be in the THREAT KS production which posts the goal of attacking a trapped piece. This COND would state that P1 should not produce any new escape squares for the trapped piece by moving to SQ. By using conditions, the productions in one KS can assure that functions they had in mind for particular pieces will not be sabotaged by the actions of other KSes.

ATTACKP P

This KS has only one production which simply posts an ATTACK with COL being the opposite color of P and SQ3 being P's location. It therefore expects whatever ATTACK would expect for attribute lists. This is used to express plans in the Plan-Language. The ATTACK KS cannot be used directly since P's location may have changed during the search. Using this KS provides for a dynamic computation of P's location just before execution of the ATTACK KS.

MOVE P1 SQ3

This KS attempts to get the piece P1 onto the square SQ3 in such a way that the opponent cannot capture him without loss (i.e., he cannot be exchanged). Once a production notices that a piece would be advantageously placed on a certain square (e.g., placing a rook on the eighth rank when the opposing king is confined to that rank), it posts a MOVE goal and lets MOVE decide if it's possible to actually get P1 onto SQ3 safely. MOVE expects the attributes LIKELY and THREAT to express the likelihood and threat of the plan to be executed after P1 is placed on SQ3. Threats made in getting P1 to SQ3 will be calculated in MOVE. The PLAN attribute should contain the plan to be executed after P1 gets to SQ3. MOVE actually does its analysis in the KSes MOVE1 and MOVE2 which are sub-KSes of MOVE and are described below. The MOVE KS itself has only one production which takes no action and simply matches all sequences in which P1 takes two moves to get to SQ3 (for the case when P1 cannot get to SQ3 in one move). Three move sequences are not considered since they give the opponent too much time to save himself. Note that only moves by P1 are counted when determining the length of a "sequence". Moves by other pieces (for example, moves which clear lines) may be part of a plan to get P1 to SQ3 in one move.

MOVE2 P1 SQ3

This KS analyzes MOVE and MOVE2 goals to find solutions involving two moves by P1, and has all the same expectations as MOVE. It attempts to move P1 to SQ2 and from there to SQ3. It tests the attribute COND for a condition on SQ2 in order to prevent this move from sabotaging the rest of the plan. For example, a fork production which posts a MOVE goal (so that P1 can fork two pieces from SQ3) may have a COND stating that the move of P1 to SQ2 should not attack one of the pieces we are attempting to fork. MOVE2 has a small number of productions since it simply posts MOVE goals for P1 moving to SQ2 (while updating LIKELY, THREAT, and PLAN to reflect the idea of moving from SQ2 to SQ3). Since P1 can legally move to SQ2, these MOVE goals will eventually be analyzed by the MOVE1 KS. If SQ3 is not safe for P1 then MOVE2 puts a PRECONDITION attribute on the new SQ2 goal which tells MOVE1 that it must make SQ3 safe for P1 before it can post the plan beginning with the move of P1 to SQ2. MOVE2 also posts goals to clear SQ2 if it is occupied by a piece friendly to P1.

MOVE1 P1 SQ3

This KS analyzes MOVE and MOVE1 goals to find solutions in which P1 can move from his current location to SQ3 (possibly after the path has been cleared or SQ3 has been made safe). It has the same expectations for PLAN, LIKELY, and THREAT as MOVE and also expects the PRECONDITION attribute from MOVE2. MOVE1 has about a dozen productions which consider moving P1 to SQ3 directly, moving friendly pieces to uncover P1's line to

SQ3, and decoying enemy pieces to unblock P1's line to SQ3. MOVE1 also must consider the problems of making SQ3 safe for P1 and satisfying any PRECONDITIONs. These problems usually result in the posting of SAFE goals although in some situations MOVE1 can solve safety problems with the plans it creates. For example, an uncovering move might either capture or pin an enemy man who was protecting SQ3, thus making it safe. MOVE1 checks the SAFE-COND attribute as a condition in those productions which attempt to get P1 safely on SQ3 by decoying or capturing some enemy piece. The enemy piece is represented by the variable DP1, and SAFECOND will put conditions on DP1 to ensure he is not the object of P1's attack from SQ3 (otherwise the purpose of getting P1 to SQ3 is negated by the act of getting him there). MOVE1 posts the following concepts: INITIAL-PLAN, SAFE, SAFER, DECOY, and any concept mentioned in a PRECONDITION.

SAFE P1 SQ
This KS attempts to find ways to make the square SQ safe for the piece P1 to land on. It expects PLAN, LIKELY, and THREAT attributes which represent the plan after any moves which make SQ safe for P1. Therefore the plan in the PLAN attribute will usually contain the goal (SAFEMOVE P1 SQ) somewhere. SAFE must add any plans and threats produced by moves which make SQ safe. SAFE has no productions of its own. It uses the two sub-KSes, SAFE1 and SAFER, to solve its goals. Both these KSes, like MOVE1, check any condition in SAFE-COND before capturing or decoying an enemy piece (DP1) to make SQ safe.

SAFE1 P1 SQ
This KS analyzes SAFE and SAFE1 goals to make SQ safe for P1, but only uses tactics that will not work when the enemy has a piece which can exchange P1 on SQ. Tactics that work in exchange situations are in the SAFER KS. For example, if the opponent has a pawn bearing on SQ and P1 is a queen, then no move which provides an additional support for SQ can possibly make it safe for P1. It is necessary to capture, decoy, or pin the pawn. Thus SAFER goals are posted when the opponent can exchange and SAFE goals are posted when he cannot (SAFE goals execute both the SAFE1 and SAFER KSes). SAFE1's expectations were described under SAFE. There are about half a dozen productions in SAFE1 which recognize the following: situations where nothing will work (e.g., SQ being overprotected against P1 when P1 can legally move to SQ), supporting moves, moves which uncover a discovered support, moves to decoy enemy pieces that will unblock supports, and moves which support SQ by attacking a sufficiently valuable piece in such a way that the attack will support SQ if the piece moves. This KS posts FORCE, INITIAL-PLAN and DECOY concepts.

SAFER P1 SQ
This KS analyzes SAFE and SAFER goals to make SQ safe for P1, using tactics that may work even if the opponent can exchange P1 on SQ. SAFER's expectations were described under SAFE. There are about fifteen productions in SAFER which recognize the following: situations where nothing will work, moves which decoy an enemy piece bearing on SQ, moves which capture enemy pieces bearing on SQ, moves which capture enemy pieces bearing ETHRU or OTHRU on SQ, moves which block the lines of enemy pieces bearing on SQ, moves which sacrifice P1 on SQ when some other piece also has a SAFE or SAFER goal for SQ, moves which start an exchange on SQ in the hope of leaving P1 on

SQ after the exchange, and attacking moves which drive away enemy pieces that protect SQ. Of course, the productions are not as simple as these descriptions. They must check many things to assure these tactics will work and that they will not adversely affect the plan to be executed after P1 gets to SQ. This KS posts INITIAL-PLAN, FORCE, DECOY, and CAPTURE concepts.

DECOY P1 SQ DP1
This KS tries to decoy the enemy piece DP1 in order to make it more plausible for P1 to move to SQ. DECOY does not know why DP1 must be moved: for example, he may be blocking P1's line to SQ or protecting SQ. DECOY does know that the decoying process should not disturb the moving of P1 to SQ. For example, P1 should not be sacrificed to decoy DP1 and DP1 should not be decoyed to a square that blocks P1's line to SQ. This KS expects LIKELY, THREAT, and PLAN attributes which represent the plan to be executed after any moves made by the decoying process. This means that the plans in the PLAN attribute will usually contain the goal (SAFEMOVE P1 SQ). DECOY has four productions which find a move, P2 to SQ1, which threatens something where only DP1 can capture safely on SQ1. P2 may be sacrificed when THREAT warrants it. DECOY posts FORCE and INITIAL-PLAN concepts and checks the condition DECOY-COND which may constrain the instantiation of P2 and SQ1.

FORCE P1 SQ P2
This KS determines if moving P1 to SQ is forcing. It assumes the move is desirable to make it if is forcing, and that it prepares a move by P2. FORCE expects LIKELY, THREAT, and PLAN attributes to represent the plan to be used after the move of P1 to SQ. It also expects a DECOY attribute which tells if the purpose of the move is to decoy a piece (and if so, what piece). FORCE checks if P2 is en prise and if it is, requires the forcing threat to be as large as the en prise threat to P2. There are about eight productions which look for captures and attacks that are threatening. They post INITIAL-PLAN concepts.

SAFEMOVE P1 SQ
This KS is used primarily to express plans in the Plan-Language. It has two productions: one which produces the move P1 to SQ as an INITIAL-PLAN if the move is MOBIL-STAY in the current position, and another which posts a MOVE goal for P1 to SQ when the move is not MOBIL-STAY. The MOVE goal may use other KSes such as SAFE or DECOY to make P1 to SQ work. The routines which execute plans in the search will use plans produced by posting a SAFEMOVE goal to elaborate the original plan being executed.

CHECKMOVE P1 SQ
This KS is like SAFEMOVE except that it also requires the move of P1 to SQ to put the opposing king in check.

MATEMOVE P1 SQ
This KS is like CHECKMOVE except that it also requires that the move of P1 to SQ leaves the opposing king with no legal moves (there may be legal interpositions).

SAFECAPTURE P1 DP1
This KS is also used primarily to express plans. It is just like SAFEMOVE except that P1

moves to DP1's location, which is computed dynamically as the productions are being matched. The productions must check for the possibility that DP1 is no longer on the board.

CAPTURE P1 DP1
This KS has only one production. It tries to capture DP1 with a MOBIL move by some piece other than P1. It produces an INITIAL-PLAN concept and expects PLAN, LIKELY, and THREAT attributes for the plan to be executed after the capturing move.

ENPRI COL
This KS is used by routines in the tree search to quickly produce a reasonable move for the defense. It has two productions that have no expectations. They suggest en prise captures and mates for the side COL, using the ENPRISE and MATE patterns in PRIMITIVE. INITIAL-PLAN concepts aru produced.

QUIESCENCE COL
This KS is used by the quiescence search to produce move suggestions. It finds all threats that seem to definitely work for the side COL. QUIESCENCE has no expectations and produces INITIAL-PLAN and NOT-QUIESCENT concepts. It has about fifteen productions (which include some from THREAT and ENPRI) that recognize the following: en prise captures, one move mates, attractive forks, attractive discovered attacks, attractive pins, and certain kinds of mating nets.

DEFENDMOVE P1 SQ
This KS is primarily used to produce defensive moves. As described in chapter one, PARADISE has very different models for the defense and offense. The offense does an in-depth analysis to find the best move, while the defense simply tries any move which has a chance to thwart the offense. DEFENDMOVE suggests moves (for DP1) which prevent or lessen the effect of the opponent moving P1 to SQ. DEFENDMOVE expects a value for the SAVE attribute, which represents the value P1 threatens to win by moving to SQ. It posts RUN and DEFENDTHREAT concepts. DEFENDMOVE has about twenty productions which recognize the following: moves which block P1's line to SQ, moves which decoy P1, moves which capture pieces bearing on SQ or pieces bearing ETHRU or OTHRU onto SQ, moves of any piece on SQ, moves which attack pieces affecting SQ or pieces guarding escape squares of the piece on SQ (thus forcing these pieces to flee to less favorable locations), moves which block the lines of pieces supporting SQ, moves which add support to SQ, moves which discover attacks on SQ, moves of pieces which are threatened with being trapped, forked, or pinned if P1 lands on SQ, moves which block P1's intended attacking line after he is on SQ, moves which open new escape squares for pieces trapped by P1 moving to SQ, and desperado moves. DEFENDMOVE considers sacrifices when the SAVE value warrants it.

DEFEND-OFFENSE P1 SQ
This is similar to DEFENDMOVE. The difference is that only "strong" or obviously necessary moves are made to prevent P1 from moving to SQ. This is used by the offense to prevent threats the defense may have. DEFEND-OFFENSE produces fewer suggestions than DEFENDMOVE and is justified by the assumptions made about PARADISE's domain. Since the offense is trying to find a combination that will win material, he is unlikely to

start by playing a move designed only to make an escape square for a piece the defense might trap. Such weak defenses are therefore not included in DEFEND-OFFENSE. This is not a large assumption since a knowledge based program playing a complete game of chess should be able to determine when it is in a defensive or offensive goal state (see for example [Berliner74]). DEFEND-OFFENSE has the same expectations as DEFENDMOVE, produces DEFENDTHREAT concepts, and has about ten productions. All of these productions except one are in DEFENDMOVE also. This one is a production that moves a piece in danger without producing every non-losing move (as the RUN KS does on goals produced by DEFENDMOVE).

RUN DP1 P1 SQ
This KS suggests moves for DP1 on the assumption that it is in danger on its current location because of the enemy's threat to move P1 to SQ. All non-losing moves that get DP1 on a square where P1 can not attack it from SQ are suggested as DEFENDTHREAT goals. RUN has four productions which rate these moves as to whether they capture enemy pieces and how much mobility they provide.

DEFENDTHREAT DP1 SQ
This KS suggests INITIAL-PLAN concepts for moving DP1 to SQ after first analyzing what threats are made by this move. It is assumed that the goal has a SAVE attribute representing the value this move will save. The only value considered as an actual threat is a capture on SQ. DEFENDTHREAT has about six productions which add a few bonus points to the THREAT and SAVE values to help order these moves which are designed to prevent a threat and not to win material. The following situations get bonuses: a check, a protection of a friendly piece that is in trouble, and an attack on an enemy piece who is threatened by the attack. This KS also checks any condition in the COND attribute which may constrain the instantiations allowed for DP1.

SAVE DP1 SQ
This KS suggests INITIAL-PLAN concepts for desperado captures which may prevent DP1 from mating the opposing king from SQ. This KS contains only one production and is used by routines in the tree search when a mate has not been avoided and any sacrifice which stops the mate is acceptable.

DESPERADO COL
This KS suggests INITIAL-PLAN concepts for any non-losing captures. This KS contains only one production and is used by routines in the tree search when a failure has not been avoided. Hopefully, exchanges brought about by moves suggested here may prevent the losing line.

DEFENSE DP1 SQ
Unlike the previously mentioned KSes, this one takes a plan as its goal. Moving DP1 to SQ is the first move in the plan specified by the goal. This KS has the same purpose as DEFENDTHREAT: it adds new threat values to the plans. It is only called by the search when the search wishes to use more execution time to gain greater accuracy in evaluation of defensive threats. There are three productions in DEFENSE which post FINALPLAN concepts.

DEFENSE-QUIDP1 SQ

This is like DEFENSE except that a few more productions are included. This KS is only used during quiescence analysis since the expense of executing these productions is only justified then.

INITIAL-PLAN P1 SQ

This KS also takes a plan as its goal. Its purpose is to analyze what the plan will lose and put this in the LOSS attribute of the plan. It may also suggest fixes to the plan to counteract a loss. Since optimistic values are desired, only very definite losses are recognized. Moving P1 to SQ is the first move in the plan specified by the goal. The following are losses recognized by this KS: an en prise piece, P1 being pinned, P1 unprotecting a piece the opponent can now capture, P1 unprotecting a square which makes a mating move safe, and P1 unprotecting a square where it could interpose to thwart a mating move. To keep things optimistic, the patterns check that no other piece can stop the threat P1 is allowing, and that P1 can not affect the allowed threat from SQ.

This KS has about a dozen productions. After recognizing the above losses, PLAN concepts are posted for plans whose THREAT or SAVE attribute outweighs the calculated LOSS, and for plans which begin with a check. PLAN concepts represent final solutions to the static analysis problem. Other productions suggest FINALPLAN concepts that represent "fixes" of the goal plan. For example, if there is a capture which thwarts the allowed threat and does not affect the goal plan, then this capture is put on the front of the goal plan to produce a "fix" for the plan. If the LOSS value outweighs both the SAVE and THREAT values and no fixes are found, then this KS will post a DEFEND-OFFENSE concept for the allowed threat. In this way, more complicated fixes that are less likely to work may be found.

C) Example of a Static Analysis

In this section, a trace of the analysis done by PARADISE on position 2.1 is presented. Some of the actual productions which match to produce the N-N5 plan are presented with brief pointers to other things noticed by the system. The trace presented was actually printed out by the program. The program prints its internal representation for pieces as names like "WR" for a white rook. Thus the rooks in position 2.1 are not disambiguated in this trace. The program prints squares in algebraic notation. It is necessary to switch from the more familiar English notation because it is not clear whether squares in patterns should be labelled from black or white's point of view. Algebraic notation is used in the remainder of this document, and figure 2.1 is reproduced below with the rows and columns labelled for this notation.

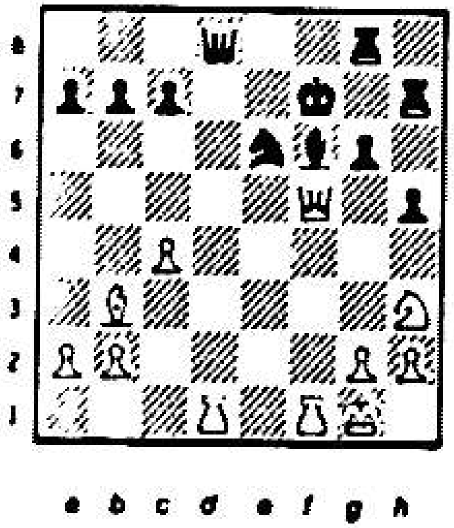


Figure 2.1
white to move

First, PARADISE executes the PRIMITIVE KS which leaves a large number of pattern matches in the data base. A THREAT concept is then posted for white and all the productions in the THREAT KS are executed. The PIN-ATTACK production recognizes that the white rook can skewer the black king to the black rook from d7. Figure 3.1 shows this production in English as a "pretty-print" routine (similar to those used in MYCIN) might print the internal representation. The internal representation is also given. All function and pattern names used are defined in chapter 2.

PIN-ATTACK LP1 SQ DMP1 DMP2

IF:
LP1 can attack the location of DMP2 from the location of DMP1, and
the location of DMP1 is safe for LP1, and
the location of DMP2 is safe for LP1, and
LP1 cannot be exchanged on DMP1's location, and
LP1 cannot be exchanged on DMP2's location, and
SQ, the location of DMP1, and the location of DMP2 are in a line, and
LP1 can attack the location of DMP1 from SQ, and
either SQ is in LP1's vue from his current location or
  LP1 can legally move to a square from where SQ is in his vue, and
it must not be true that LP1 can legally capture DMP1 and LP1's location,
  DMP1's location, and DMP2's location are in a line, and
DMP1 has a mobil move to some square,
THEN:
post the concept (MOVE1) of moving LP1 to SQ with attributes as follows: The THREAT is to fork DMP1 and DMP2 with LP1. The LIKELY is 0. The SAFECOND condition says that DMP1 and DMP2 should not be moved in an attempt to make SQ safe for LP1. The PLAN is to safely move LP1 to SQ and then safely capture DMP2 if DMP1 moves anywhere, and safely capture DMP1 if any piece other than DMP1 moves.

```
PIN-ATTACK
((LP1 SQ DMP1 DMP2)
(FUNCTIONE CANATTACK LP1 (LOCATION DMP1) (LOCATION DMP2))
(FUNCTION SAFEP LP1 (LOCATION DMP1))
(FUNCTION SAFEP LP1 (LOCATION DMP2))
(NEVER (FUNCTION EXCHANGE LP1 (LOCATION DMP1)))
(NEVER (FUNCTION EXCHANGE LP1 (LOCATION DMP2)))
(FUNCTION LINE SQ (LOCATION DMP1) (LOCATION DMP2))
(FUNCTION CANATTACK LP1 SQ (LOCATION DMP1))
(OR      (FUNCTION IN-VUE P1 (LOCATION P1) SQ)
         (EXISTS (SQ1) (PATTERN LEGALMOVE P1 SQ1) (FUNCTION IN-VUE P1 SQ1 SQ)))
(NEVER (PATTERN LEGALMOVE LP1 (LOCATION DMP1))
         (FUNCTION LINE (LOCATION LP1) (LOCATION DMP1) (LOCATION DMP2)))
(EXISTS (SQ2) (PATTERN MOBIL DMP1 SQ2))
(ACTIONNEW MOVE1 (LP1 SQ) (THREAT (FORK LP1 DMP1 DMP2)) (LIKELY 0)
 (SAFECOND (FUNCTION NEQ DP1 DMP1) (FUNCTION NEQ DP1 DMP2))
 (PLAN  (SAFEMOVE LP1 SQ)
   (((DMP1 NIL) (SAFECAPTURE LP1 DMP2)) ((ANYBUT DMP1) (SAFECAPTURE LP1 DMP1))))))
```

Figure 2.3
PIN-ATTACK production in pretty-print and internal form (THREAT KS)

In PIN-ATTACK, the reference to CANATTACK suggests possible pieces for pinning while the references to SAFEP and EXCHANGE check that both pieces are actually threatened by the attacking piece. The reference to LINE makes sure that the pieces are in a line and instantiates SQ to squares from which the pin can be made. The reference to

CANATTACK assures a clear path from SQ to DMP1. The references to IN-VUE check that there is some reasonable hope of getting LP1 to SQ. The next expression checks that the pin pattern is not already present. Requiring DMP1 to have a MOBIL move avoids duplicating work being done by the production suggesting attacks on trapped pieces.

Executing the action of PIN-ATTACK inserts the following MOVE1 concept in the data base:

```
((WR D7)
    (SAFECOND (FUNCTION NEQ DP1 BK)(FUNCTION NEQ DP1 BR))
    (LIKELY 0) (THREAT (FORK WR BK BR))
    (PLAN   (SAFEMOVE WR D7)
       (((BK NIL) (SAFECAPTURE WR BR)) ((ANYBUT BK) (SAFECAPTURE WR BK))))
    (REASON (PIN-ATTACK WR D7 BK BR)) )
```

This concept gives the MOVE1 KS the information it needs to accurately evaluate the possibility of getting the white rook safely on d7. After executing the THREAT KS, there are many concepts besides this one in the data base. In fact, the PIN-ATTACK production itself also matches for moving the rook to c7 or e7 as well as for moving the queen to any of these squares. The variable instantiations of each concept in the data base are given below. The attribute lists are omitted so it is not apparent where these ideas originate.

```
(THESE ARE GOALS )

(OCCURRENCES-OF ATTACK)
((WHITE F6))

(OCCURRENCES-OF MOVE)
((WN G5) (WB G8) (WB G6) (WB F7) (WR E7) (WR F6) (WR F8) (WR E7) (WR F6) (WR E8)
 (WQ G8) (WQ G6) (WQ F8) (WQ G7) (WQ E7) (WQ F6))

(OCCURRENCES-OF MOVE1)
((WB E6) (WB C4) (WB D5) (WB G8) (WR C7) (WR D7) (WR E7) (WR F8) (WR A6) (WR B6)
 (WR C6) (WR D6) (WR E7) (WR F8) (WQ C7) (WQ D7) (WQ E7))

(OCCURRENCES-OF DECOY)
((WQ E6 BK))

(OCCURRENCES-OF INITIAL-PLAN)
(((WQ E6) (((BK E6) (SAFEMOVE WR E1) (BK NIL) (ATTACKP BK))
           ((BK E6) (SAFEMOVE WR E1) (BK NIL) (ATTACKP BK))
           ((BK E6) (SAFEMOVE WN G5) (BK NIL) (ATTACKP BK))
           ((BK E6) (SAFEMOVE WP C5) (BK NIL) (ATTACKP BK))))
((WR D8)))
```

The goal of attacking f6 is suggested because the black bishop on f6 cannot move. The MOVE and MOVE1 goals stem from various possible pins and forks which involve all black's major pieces, and a possible mating net which may result from some queen moves. The goal of decoying the black king is suggested so that the black knight on e6 can be captured. (More MOVE, MOVE1, and DECOY goals will be suggested later.) Lastly, two plans have already been suggested. The first is not likely to succeed and suggests sacrificing the white queen for the black knight in hopes of getting a mating net with the rooks. (Its LIKELY is 1 since PARADISE knows that the second check won't be mate and it can't see a forcing move for white on his third move.) Four alternatives are given after the first move, the first two of which involve moving both white rooks to e1. The second plan is likely to succeed and suggests capturing the black queen with the white rook.

PARADISE now executes the ATTACK KS on the ATTACK goal. This KS produces many more MOVE and MOVE1 goals for moves which attack f6. In all, there are 26 MOVE goals and 24 MOVE1 goals. The MOVE KS is now executed. Since it has sub-KSes, the productions in the MOVE KS are executed for all MOVE goals, and all pattern matches produced by such executions are kept in the data base. Then all MOVE and MOVE2 goals are processed by the MOVE2 KS which looks for two move attacks. (This is a normal execution with pattern matches being deleted after each particular goal is finished.) There is a MOVE goal for getting the white rook on d1 to f7 since this would attack f6. Execution of the MOVE2 KS on this particular goal results in creation of a MOVE1 goal for moving the white rook to d7 (as the first of two moves to f7). Now the MOVE1 KS is executed for all MOVE1 and MOVE goals. A close look at the MOVE1 goal for (WR D7) shows there are now three attribute lists: the one from the PIN-ATTACK production, the one just mentioned (this two-move attack has a LIKELY of 1 while the other two attribute lists have LIKELYs of 0), and one from a production in the THREAT KS which recognized that after the rook is on d7, the black king must move so that the black queen can be captured with check. The M1UNSAFE production, shown in figure 3.2, is in the MOVE1 KS and matches the (WR D7) goal.

M1UNSAFE P1 SQ3

IF:

P1 can legally move to SQ3, and

P1 is not mobil to SQ3, and

P1 cannot be exchanged on SQ3, and

all attribute lists with preconditions are removed,

THEN:

post the concept (SAFE) of making SQ3 safe for P1 with attributes changed as follows: PLAN becomes NIL followed by the plan in the goal, and COND becomes NIL. Do not execute any more productions in this KS.

```
M1UNSAFE
((P1 SQ3)
(PATTERN LEGALMOVE P1 SQ3)
(NEVER (PATTERN MOBIL P1 SQ3))
(NEVER (FUNCTION EXCHANGE P1 SQ3))
(REMOVE-ATR-LISTS T ((PRECONDITION NIL)))
(ACTION SAFE (P1 SQ3) (PLAN NIL (GOAL1 PLAN)) (COND))
(RETURN))
```

Figure 3.2
M1UNSAFE production in pretty-print and internal form (MOVE1 KS)

This production is fairly straightforward. P1 and SQ3 are instantiated by the goal being executed, in this case to WR and D7. Since the move is not MOBIL, it must be made safe. If P1 could be exchanged, then SAFER would be used instead of SAFE. Goals with preconditions are handled by other productions. All the attributes of the goal are inherited by the new concept unless explicitly changed (e.g., SAFECOND, LIKELY, and THREAT are inherited). M1UNSAFE produces the following SAFE concept in the data base (all three attribute lists are shown):

```
((WR D7)
 ((SAFECOND (FUNCTION NEQ DP1 BK)(FUNCTION NEQ DP1 BR))
    (LIKELY 0) (THREAT (FORK WR BK BR))
    (PLAN NIL (SAFEMOVE WR D7) (( (BK NIL) (SAFECAPTURE WR BR) )
            ((ANYBUT BK) (SAFECAPTURE WR BK) )))
    (REASON (M1UNSAFE WR D7) (PIN-ATTACK WR D7 BK BR)) )
 ((SAFECOND (FUNCTION NEQ DP1 BQ))
    (LIKELY 0) (THREAT (EXCH WR BQ))
    (PLAN NIL (CHECKMOVE WR D7)  (BK NIL) (SAFECAPTURE WR BQ) )
    (REASON (M1UNSAFE WR D7)(CHECK-FORCE WR BQ WR D7 BK)) )
 ((DECOYCOND (NEVER (FUNCTION CANATTACK DP1 SQ1 F7)))
    (LIKELY 1) (THREAT (PLUS (EXCHVAL WR F6) (EXCHVAL WR F7)))
    (PLAN NIL (SAFEMOVE WR D7) NIL (SAFEMOVE WR F7) )
    (REASON (M1UNSAFE WR D7)(M2SAFE WR D7 F7)(DIRECT2 WR F7 F6)(TRAPPED BB)) ))
```

The three attribute lists should be self-explanatory since their origins are known. The REASON attribute gives a trace of the productions which fired to produce a given concept and provides explanatory capabilities. In PARADISE it is always apparent exactly why a plan was suggested; there is no need to trace through obscure code to discover what the system was thinking about. There are other goals in the data base after the MOVE KS has finished executing. Their variable instantiations are given below:

```
(THESE ARE GOALS )

(OCCURRENCES-OF SAFE)
((WR D7) (WQ G6) (WQ F6))

(OCCURRENCES-OF SAFER)
((WP G4) (WN G5) (WN F4) (WR D6))

(OCCURRENCES-OF DECOY)
(( (WQ E6 BK))

(OCCURRENCES-OF INITIAL-PLAN)
(( (WR E1) NIL (SAFEMOVE WR E5) )
 ( (WR E1) NIL (SAFEMOVE WR E6) )
 ( (WN F2) NIL (SAFEMOVE WN E4) ))
```

The MOVE goals produced three SAFE goals and four SAFER goals. The DECOY goal produced by the THREAT KS is still around. In addition there are three new plans. (The two plans produced by the THREAT KS are omitted above but they are still in the data base.) These three plans all have a LIKELY of 1 and are trying to attack the black bishop on f6. PARADISE now executes the SAFE KS which consists of two sub-KSes. First the SAFE1 KS is executed on the SAFE goals and then the SAFER KS is executed on both the SAFER and SAFE goals. A number of productions in both SAFE1 and SAFER match the (WR D7) goal. Among them is the DECOY-BLOCKER production from the SAFE1 KS which is shown in figure 3.3. It recognizes when an enemy piece can be decoyed to uncover a protection of the goal square.

In the SAFE KS, P1 and SQ are instantiated by the goal (to WR and D7 in this case). LP1 and DP1 are instantiated by the THRU pattern in DECOY-BLOCKER (to WQ and BN in this case). The variables SQ1 and P2 in the DECOYCOND attribute are the variables instantiated by the DECOY KS to the decoying move.

DECOY-BLOCKER P1 SQ DP1 LP1

IF:

P1 can legally move to SQ, and
LP1 bears thru DP1 onto SQ, and
any condition in SAFECOND matches (on DP1),

THEN:

post the concept (DECOY) of decoying DP1 to make the move P1 to SQ possible. Attributes are changed as follows: DECOYCOND specifies that the decoy should not be made along the line from LP1 to SQ and that LP1 should not make the decoying move.

```
DECOY-BLOCKER
((P1 SQ DP1 LP1)
 (PATTERN LEGALMOVE P1 SQ)
(PATTERN THRU LP1 DP1 SQ)
(CHECKCOND SAFECOND)
(ACTION DECOY (P1 SQ DP1)
 (DECOYCOND (NEVER (FUNCTION LINE (LOCATION LP1) SQ1 SQ))
        (NEVER (FUNCTION EQUAL P2 LP1)))))
```

Figure 3.3
DECOY-BLOCKER production in pretty-print and internal form (SAFE1 KS)

The DECOY goal posted in the data base by this production looks much like the SAFE goal presented earlier except that each attribute list now has a DECOYCOND attribute and the REASON attribute has been updated. The goals in the data base after execution of the SAFE KS are given below:

```
(THESE ARE GOALS )

(OCCURRENCES-OF DECOY)
((WP G4 BP) (WN G5 BN) (WN F4 BN) (WR D7 BR) (WR D7 BQ) (WQ E6 BK) (WQ F6 BQ))

(OCCURRENCES-OF FORCE)
((WB A4 WR) (WQ B5 WR))

(OCCURRENCES-OF CAPTURE)
((BP WP) (BN WN) (BQ WR) (BQ WQ) (BK WQ))

(OCCURRENCES-OF INITIAL-PLAN)
(((WR E1) NIL (SAFEMOVE WN G5) (((ANYBUT BK) (SAFECAPTURE WN BK) )
    ((ANYBUT BR) (SAFECAPTURE WN BR) )))

 ((WR E1) NIL (SAFEMOVE WN G5) (((ANYBUT BK) (SAFECAPTURE WN BK) )
    ((ANYBUT BR) (SAFECAPTURE WN BR) ))))
```

There are seven DECOY goals. Another production in SAFE also matches the (WR D7) goal and suggests decoying the black queen in order to make this move safe. The goal of decoying the queen is only suggested for two of the three attribute lists since SAFECOND in the attribute list concerned with capturing the queen with check prevents removal of the queen. Two FORCE goals are generated by productions in SAFE, since (WB A4) and (WQ B5) may help the white rook get to d7 (especially if they are forcing moves). Five CAPTURE goals are suggested, and two new plans are suggested (again, ones described earlier have been omitted above). Both these plans have a LIKELY of 0, and realize that moving either white rook to e1 may drive the black knight from his post, enabling the white knight to safely stay on g5 and fork black's king and rook. PARADISE now executes the DECOY KS on the above goals. The goal of decoying the black knight to help (WR D7) matches the MOVE-DECOY production in the DECOY KS.

In the DECOY KS, P1, SQ, and DP1 are instantiated by the goal (to WR, D7 and BN in this case) where DP1 must be decoyed to help make it possible for P1 to move to SQ. MOVE-DECOY attempts to instantiate P2 and SQ1 to a move that would decoy DP1. MOVE-DECOY checks that DP1 will likely capture a piece making a threat from SQ1, that DP1 will not still bear directly on SQ after being decoyed to SQ1, that any sacrifice of P2 appears worthwhile, that DP1 is not pinned to a piece so valuable that DP1 won't move, and that P2 does not bear directly on SQ since a protection of SQ would be lost by the decoy (this is allowed however when P1 bears thru DP1 onto SQ, and is also allowed in other productions).

The action of MOVE-DECOY posts a FORCE goal since moving P2 to SQ1 will only be effective if it threatens something (i.e., is forcing). MOVE-DECOY matches the (WR D7 BN) decoy goal in two different ways, suggesting both (WN G5 WR) and (WN F4 WR) as force goals. The following concept is posted in the data base as a FORCE concept for the (WN G5 WR) match:

```
((WN G5 WR)
 ((DECOYCOND (NEVER (FUNCTION LINE (LOCATION WQ) SQ1 D7))
             (NEVER (FUNCTION EQUAL P2 WQ)))
  (SAFECOND (FUNCTION NEQ DP1 BK) (FUNCTION NEQ DP1 WR))
  (LIKELY 0) (THREAT (FORK WR BK BR))
  (PLAN (BN G5) (SAFEMOVE WR D7) (( (BK NIL) (SAFECAPTURE WR BR) )
             ((ANYBUT BK) (SAFECAPTURE WR BK) )))
  (DECOY BN)   (REASON (MOVE-DECOY WR D7 WN G5 BN) (DECOY-BLOCKER WR D7 BN WQ)
               (M1UNSAFE WR D7)(PIN-ATTACK WR D7 BK BR)) )
 ((DECOYCOND (NEVER (FUNCTION LINE (LOCATION WQ) SQ1 D7))
             (NEVER (FUNCTION EQUAL P2 WQ)))
  (SAFECOND (FUNCTION NEQ DP1 BQ)) (LIKELY 0) (THREAT (EXCH WR BQ))
  (PLAN (BN G5) (CHECKMOVE WR D7)  (BK NIL) (SAFECAPTURE WR BQ) )
  (DECOY BN) (REASON (MOVE-DECOY WR D7 WN G5 BN)
          (DECOY-BLOCKER WR D7 BN WQ)(M1UNSAFE WR D7)(CHECK-FORCE WR BQ WR D7 BK)) )
 ((DECOYCOND (NEVER (FUNCTION LINE (LOCATION WQ) SQ1 D7))
             (NEVER (FUNCTION EQUAL P2 WQ)))
  (LIKELY 1) (THREAT (PLUS (EXCHVAL WR F6) (EXCHVAL WR F7)))
  (PLAN (BN G5) (SAFEMOVE WR D7) NIL (SAFEMOVE WR F7) )
  (DECOY BN) (REASON (MOVE-DECOY WR D7 WN G5 BN)(DECOY-BLOCKER WR D7 BN WQ)
          (M1UNSAFE WR D7)(M2SAFE WR D7 F7)(DIRECT2 WR F7 F6)(TRAPPED BB)) ))
```

The DECOYCOND and SAFECOND attributes in the above goal are no longer needed but the other attributes are used by other KSes. The variable instantiations of all the goals in the data base after execution of the DECOY KS are given below. INITIAL-PLAN concepts are omitted since no new ones have been posted.

```
(THESE ARE GOALS)

(OCCURRENCES-OF FORCE)
((WP C5 WN) (WN G5 WR) (WN F4 WR) (WB A4 WR) (WR D7 WQ) (WR D5 WQ) (WR D3 WQ)
 (WR D2 WQ) (WQ B5 WR))

(OCCURRENCES-OF CAPTURE)
((BP WP) (BN WN) (BQ WR) (BQ WQ) (BK WQ))
```

The FORCE KS is now executed on the above FORCE goals. It looks for any attack that may possibly be forcing. In the FORCE KS, P1 and SQ are instantiated to the move being proposed as (hopefully) forcing, in this case WN and G5. A production in the FORCE KS matches the (WN G5 WR) goal since N-g5 attacks the black king, knight, and rook. The action of this production is:

```
(PLAN ((P1 SQ) (GOAL PLAN)) (THREAT (PLUS (EXCHVAL P1 SQ) (GOAL1 THREAT))))
```

This action forms an INITIAL-PLAN concept in which the plans from all the attribute lists with the best LIKELY value are combined to form conditional branches. Thus the posted INITIAL-PLAN concept has a Plan-Language expression that starts with (WN G5) and then branches to include the Plan-Language expressions which are values of the PLAN attribute in the two attribute lists of the (WN G5 WR) goal which have a LIKELY of 0. The threat of exchanging the WN on G5 is added to the THREAT attribute of each attribute list in the goal. In all, five new INITIAL-PLAN concepts are posted by the execution of the FORCE KS on the above goals. They are listed below.

```
(OCCURRENCES-OF INITIAL-PLAN)
((((WQ B5) NIL (SAFEMOVE WR D7)
          (((BK NIL) (SAFECAPTURE WR BR) )
           ((ANYBUT BK) (SAFECAPTURE WR BK) )
           ((BK NIL) (SAFECAPTURE WR BQ) )))
 ((WB A4) NIL (SAFEMOVE WR D7)
          (((BK NIL) (SAFECAPTURE WR BR) )
           ((ANYBUT BK) (SAFECAPTURE WR BK) )
           ((BK NIL) (SAFECAPTURE WR BQ) )))
 ((WR D7) (BQ D7) (SAFEMOVE WQ F6) )
 ((WN F4)
   (((BN F4) (SAFEMOVE WR D7)
       (((BK NIL) (SAFECAPTURE WR BR)) ((ANYBUT BK) (SAFECAPTURE WR BK))))
   ((BN F4) (CHECKMOVE WR D7) (BK NIL) (SAFECAPTURE WR BQ) )))
 ((WN G5)
   (((BN G5) (SAFEMOVE WR D7)
       (((BK NIL) (SAFECAPTURE WR BR)) ((ANYBUT BK) (SAFECAPTURE WR BK))))
   ((BN G5) (CHECKMOVE WR D7) (BK NIL) (SAFECAPTURE WR BQ) ))))
```

The first two plans have LIKELYs of 1 and recognize that (WQ B5) and (WB A4) will make d7 safe for the rook. The remaining three plans all have a LIKELY of 0. The first of these three suggests (WR D7) to decoy the black queen so that the white queen can take the black bishop. The last two are the ones we've traced. The CAPTURE KS now executes the capture goals. None of them succeed so no new concepts are added to the data base.

The INITIAL-PLAN KS is now executed, using all the INITIAL-PLAN concepts mentioned so far as goals. INITIAL-PLAN concepts that begin with a check are matched by a production which posts them as FINALPLAN concepts with no LOSS attribute. For the non-checking plans, this KS realizes that they will lose the en prise queen for a pawn. The (WR D8) and (WN F4) plans are rejected since they do not have a THREAT or SAVE attribute which justifies losing a queen. All the plans with a LIKELY of 1 have such a THREAT and are posted as FINALPLAN concepts (with a LOSS attribute describing the loss of the queen). The plans beginning with the move of a white rook to E1 have attribute lists with LIKELYs of both 0 and 1. In these plans, the LIKELY 0 attribute lists are removed (because their threats are too low), and the LIKELY 1 attribute lists are retained with a LOSS attribute added. Thus there are only two FINALPLAN concepts which have a LIKELY of 0: the (WR D7) plan and the (WN G5) plan. The final plan for (WN G5) is shown below with all its attribute lists.

```
( ((WN G5)
    ((((BN G5) (SAFEMOVE WR D7)
        ((((BK NIL) (SAFECAPTURE WR BR)) ((ANYBUT BK) (SAFECAPTURE WR BK))))
    ((BN G5) (CHECKMOVE WR D7) (BK NIL) (SAFECAPTURE WR BQ) ))))
 ((THREAT (PLUS (EXCHVAL WN G5) (FORK WR BK BR))) (LIKELY 0)
    (REASON (PLANOK WN G5)(FCHECK WN G5)(MOVE-DECOY WR D7 WN G5 BK)
         (DECOY-BLOCKER WR D7 BN WQ)(M1UNSAFE WR D7)(PIN-ATTACK WR D7 BK BR)) )
 ((THREAT (PLUS (EXCHVAL WN G5) (EXCH WR BQ))) (LIKELY 0)
    (REASON (PLANOK WN G5)(FCHECK WN G5)(MOVE-DECOY WR D7 WN G5 BK)
         (DECOY-BLOCKER WR D7 BN WQ,(M1UNSAFE WR D7)(CHECK-FORCE WR BQ WR D7 BK)) )
 ((THREAT (PLUS (EXCHVAL WN G5) (PLUS (EXCHVAL WR F7) (EXCHVAL WR F6))))
    (LIKELY 1) (REASON (PLANOK WN G5)(FCHECK WN G5)(MOVE-DECOY WR D7 WN G5 BK)
   (DECOY-BLOCKER WR D7 BN WQ)(M1UNSAFE WR D7)(M2SAFE WR D7 F7)(DIRECT2 WR F7 F6)(TRAPPED BB))))
```

The rejection of the (WR D8) plan causes a production to post a DEFEND-OFFENSE goal to prevent the black pawn from capturing the queen. Executing the DEFEND-OFFENSE KS produces six DEFENDTHREAT goals which produce six INITIAL-PLAN goals. Executing the INITIAL-PLAN KS again results in five more FINALPLAN concepts being posted. All five have a LIKELY of 1 and involve moving the white queen to a safe square and then attempting to take the black queen with the white rook. The static analysis is now complete. Twelve FINALPLAN concepts have been posted but only two of them have a LIKELY of 0. Thus the search will first try to show that the (WR D7) and (WN G5) plans will win material.

This analysis takes almost 25 seconds of cpu time on the DEC KL-10. (The production interpreter is inefficient.) Thus a static analysis is costly and PARADISE tries to avoid doing such an analysis by using its plans to guide the search. As chapter 6 shows, PARADISE has been very successful in this endeavor.

The following list mentions some of the more important features of the static analysis process in PARADISE:

-The static analysis is done entirely by the production rules which can easily be modified to increase or make more precise the knowledge available to the system. These same rules are also used to interpret plans and communicate knowledge in the tree search.

-Each concept (and plan) produced has a REASON attribute listing each production that matched in the process of suggesting the concept. This provides a good explanation of exactly why PARADISE believes something. This is helpful for quick debugging and easy modification.

-The attribute lists of each concept contain enough information so that the system essentially knows "why" it is doing something. Thus wrong ideas can be discarded readily and combinations of ideas that are antithetical to each other can be avoided.

-The plans produced by the static analysis are quite sophisticated. The example plan shown above will guide the search for 5 ply, but contains enough information to enable quick abandonment of the plan if it is not working. The information in the attribute lists allows for comparing of plans and fairly sophisticated cutoffs in the tree search (see chapter 4).

-The analysis is not easily led off the track in the sense that most plans suggested make some sense. Few ridiculous plans are suggested.

CHAPTER IV

Controlling the Tree Search

A) What makes a good tree search?

Whenever a tree being searched is so large that the resources available cannot search the entire tree, the searcher must provide for terminating the search before available resources are exhausted. It is useful to classify search algorithms as knowledge-controlled or parameter-controlled based on how they terminate. A parameter-controlled search is controlled by a set of preset parameters which cause termination when some feature of the search exceeds some limit specified in a parameter. To be parameter-controlled, at least one of the parameters must concern something unrelated to the relationship between the current node and some goal, such as the size of the tree or amount of effort expended. All the terminations in such a search need not be caused by the parameters; the distinguishing feature is that without the parameters causing some terminations the search would not converge and terminate in a reasonable period of tir Nearly all computer chess programs employ parameter-controlled searches with the most important parameter for termination being a depth limit on the search.

A knowledge-controlled search is controlled by features of the problem and information discovered during the search. When the search is terminated at a node, it is because of some relationship between this node and the goals of the search. Studies indicate that human chess grandmasters use a knowledge-controlled search, deciding to terminate the search when a definite evaluation of a line has been determined, a particular problem solved, or a discovery made. In this way a grandmaster searches very few nodes which are not relevant to proving that one move is best in the original position. Maximum effort limits certainly exist for humans, but a grandmaster will rarely abandon investigation of a critical line in a tactically sharp position because of inadequate mental abilities. It should be noted that the distinction between knowledge-controlled and parameter-controlled is not formally defined. There can be fuzziness about whether a given tree-pruning device is making a valid knowledge-based cutoff or is using a little knowledge and a lot of concern about amount of effort to make a cutoff not really supported by what is known.

It has been argued elsewhere (e.g., in [Berliner74]) that the searching algorithm of a knowledge based chess program should be knowledge-controlled. A parameter-controlled search with a depth limit misses combinations deeper than its depth limit. Knowledge-controlled searches have not been practical for computer programs because it is hard to get them to converge. Convergence seems to require that many nodes be recognized as terminal nodes, and that only a small subset of all legal moves be searched at most nodes.

When only subsets of the legal moves at a node are searched, a program can err in its analysis any time the best move is not searched at a node. Often the best move in a position is subtle and will not be noticed by a computationally inexpensive move selector. Many chess programs have searched only subsets of the legal moves. Bernstein's [Bernstein59] was the first, while Greenblatt's was the first to play with class C strength. The Northwestern program also did not search all the legal moves in its earlier versions. Some of these programs have tried to recover from missing the best move; for example, by re-initializing a broader search when the initial search returns an unsatisfactory result. However, all these programs fall far short of expert performance and one of the contributing factors is that some good moves are never searched. CHESS 4.7, the current version of the Northwestern program, searches every legal move and seems to be playing better chess than any program before it.

Human grandmasters search only a few moves at each node but they rarely miss the best move. Programs often miss the best move because they do not have enough knowledge to use in selecting the moves to search. However, any commitment to spending a significant amount of resources processing knowledge at each node must result in a large reduction in the size of the search tree to keep the problem tractable. Berliner's program employs more knowledge than any previous program and can search deeper since fewer moves are searched at each node. However, the amount of knowledge encoded is very small compared to that of a human master. The program uses a parameter-controlled search and misses the best move on occasion. It is difficult to add more knowledge to the program.

The approach used in PARADISE to the above problems is to use a large amount of knowledge in the analysis in order to avoid skipping the best move. The idea is to add the missing knowledge when the best move is skipped, until the program reaches a given level of performance. The knowledge base is designed for easy modification so that gaps in the knowledge can easily be filled. To offset the expense of using knowledge, PARADISE uses a small knowledge-controlled tree search.

To achieve convergence in the search, every possible effort must be made to label nodes as terminal without introducing errors. Many possible terminating criteria require knowledge on the level of understanding used by human masters. A grandmaster looks at the features of a particular position and uses his evaluation in the global context of the tree, knowing what problem he is trying to solve, and then decides (using this information) if the position merits more search or not. A program must use similar techniques to obtain a knowledge-controlled search. A program needs to avoid searching positions when the result will not affect the overall appraisal of the situation. To do this, it must recognize positions that are too poor for the side on move (he's made a mistake), too good for the side on move (the opponent's made a mistake), and positions irrelevant to the current problem being solved. A program should also terminate whenever further results from the current line will not affect the top level choice of move, when a particular piece of information has been discovered, or when searching the current line is not the best way to obtain a result.

PARADISE uses techniques like those mentioned above to achieve a knowledge-controlled search within its domain. There is no depth limit or other artificial cutoff. PARADISE's knowledge-controlled search relies on three things: 1) the large knowledge base which can be brought to bear for analysis, 2) the program's ability to communicate information from one part of the search tree to another, and 3) the assumptions made about the domain (in particular, that the position is tactically sharp and the offense has prospects to win material). The original designation of offense and defense is never changed by the program. The search is wide and deep enough to find many combinations (e.g., the 19 ply combination given in chapter 1). The remainder of this chapter describes the search in more detail, showing the techniques used to achieve a knowledge-controlled search.

(N.B. During this discussion, the program is considered to be growing its search tree downwards from the original position which is the root node at the top of the tree.)

B) Overview of PARADISE's search

1) Introduction

Most search-oriented programs search each move in the original position in order to determine its "true" value, and then select the move with the best value. (A "true" value would be win, lose, or draw, but during this discussion a "true" value is considered to be the evaluation a human grandmaster might produce after a few minutes of analyzing a given move.) PARADISE's search employs a different paradigm. The search's mission is to prove that one move in the original position is better than any other. This can often be done without knowing the "true" value of each move. For example, suppose that in some position one move will lead to a complicated mate while no other move leads to an advantage. The program does not need to prove that the winning move actually leads to mate (the "true" value). Once it has been shown that none of the other moves forces an advantage, it is only necessary to show that the mating move leads to some advantage (which may be much easier that showing the mate).

When PARADISE uses its search to investigate a plan, it invests some amount of effort to determine a range within which the value of the plan lies. (The idea of having a program use ranges in this way was first proposed in [Berliner79].) To show that one move is best in the original position, it is only necessary to show that the lower bound of the range of one plan is better than the upper bound of the ranges of all other plans. PARADISE continues to initiate searches which narrow the ranges of plans being searched (possibly until a range converges to a single value). Eventually the ranges are narrowed enough to decide that one move is best. A plan may need to be searched more than once during this process. The search could be described as "progressively deepening", a term DeGroot used to describe the chess analysis of human grandmasters ([DeGroot65]).

The program analyzes the situation at the top level after each narrowing of a range to develop a strategy for proving that one plan is best. For example, one strategy might prove that all the moves except the best so far cannot win, while another strategy might prove that the best move so far wins convincingly. The different strategies used are described in section G of this chapter. By continually reassessing the situation, PARADISE avoids investing a lot of effort in an endeavor which may not be helpful in showing that one move is best. This is important in order to obtain a knowledge-controlled search. This formal searching paradigm was first developed in Berliner's B* search (see [Berliner79]), and will be made more concrete as the details of the system are presented in the remainder of this chapter.

Since the calculation of primitives is expensive, PARADISE stores each position and all the information calculated about it on a disk file. This makes the cost of searching a line again reasonable. Searching a plan returns information in the form of a tree. These trees correspond to the trees traversed during the search, with a certain amount of information at each node, including the following: the plans searched, the actual moves which each plan caused to be searched, the range of values obtained by searching each plan and each move, the reason for termination of each search, a small number of patterns that were matched at this node, the name of the disk file which has more information about this node, and any plans which have not yet been investigated. These trees are kept in core which is possible since PARADISE grows small trees. They provide information which can be used for many purposes including deciding where to search next, analyzing why a plan failed, and reconstructing a line which is to be searched again.

Each search initiated from the top level is best-first. PARADISE searches depth first down a line (as shown in the example search in chapter 2) but keeps track of unsearched alternatives and their expectations as it goes down. If at some point one of the unsearched alternatives seems better than the current plan, the current line of search is terminated and the search backs up to try the more promising alternative. Deciding which unsearched plan is more promising is not based strictly on the expectation of each plan. There is a cost involved in abandoning a line and then later coming back to it so PARADISE is reluctant to abandon the current line. This reluctance (described in section G of this chapter) diminishes as the current line's depth in the tree becomes larger than the depth of the alternative. Terminating the current line inserts information in the tree which is used whenever the line is searched again (for example, if the alternative fails and the system decides the original line was best after all). Re-searching lines results in progressive deepening of the tree. The details of this are described later in this chapter.

With the best-first search, the program always searches from what it thinks is the most promising non-terminal node in the tree. This type of search is necessary for PARADISE's search to be knowledge-controlled. Without abandoning lines for better alternatives, PARADISE would in some cases use unreasonable amounts of resources trying to make a poor line work. The success of the best first search depends on the ability to make reasonable decisions about which alternative is more promising. The tactical sharpness of PARADISE's domain make such decisions easier than they would be in positional situations.

The search is continually striving to arrive at a quiescent position before evaluating. Given the importance of quiescence, the search can be divided into four major components: the offensive search, the defensive search, the quiescence search, and the top-level search strategy. (The program is initially told which side is the offense and this is not changed during the search.) The offensive search makes decisions which control the searching process whenever the offense is on move. It attempts to terminate whenever possible and when it cannot, it decides which offensive plan to search next. The offensive search is described in section C of this chapter. The defensive search controls the searching process when the defense is on move. It is concerned with finding the move which will best thwart the offensive plan, and is described in section D of this chapter. The quiescence search tries to determine the value of the current position by playing the best move or two for each side until a quiescent position is reached. It is described in section E of this chapter. The top-level search strategy decides where to search next in order to prove that one plan is best. This involves deciding which nodes are most promising, and efficiently restarting a search that has be suspended. This aspect of PARADISE is described in section G of this chapter.

It should be noted that PARADISE's search is not strictly a best-first search, but more like Berliner's B* search (see section H of this chapter). A classical best-first search does not use ranges as values and does not select different strategies at the top level. One other difference is PARADISE's use of plans to guide the search. Once a plan is selected, it is used until it is exhausted or until the program determines that it is not working. Decisions about which node is most promising are effectively made only when PARADISE changes plans, and not at every node as in a classical best-first search.

2) Measures used in PARADISE's search

To obtain a knowledge-controlled search, the program must make every possible effort to terminate branches of the search. Branches which aren't succeeding should be terminated immediately and branches which are succeeding should be terminated as soon as enough information has been obtained to help show that the initial move is best at the top node. PARADISE can only guess at how much is "enough". It does this by defining a value called the threshold, which is initially set to a value of about a pawn and two-thirds. The search is terminated whenever the threshold is achieved (i.e., the offense has won at least a pawn and two-thirds). The program realizes that a larger advantage may be achieved, and may later restart the search from the point of termination after setting the threshold higher.

Many termination decisions involve deciding when a position is too good or too bad for the side on move. To do this the program must have an expectation of the value of a position and be able to change that expectation on the basis of newly discovered information. A major problem is deciding when to be satisfied with a result. Closely related to this is the problem of obtaining satisfaction with as little effort as possible. The way PARADISE expresses the values of positions and its expectations, described below, helps solve both these problems.

It can be very expensive to find the "true" value of a position since all contingencies must be investigated to do so. Since all that is needed is enough information to show that one move is better than another, PARADISE does not worry about obtaining the value of a position. Rather in determines a range of values that the position lies in and narrows the range (by investing more resources) as much as necessary to show that one move is best. All ranges have a bottom and a top where the bottom is the best value the defense can hope to achieve. The offense has proved that the bottom of the range can be achieved. The top of the range is the best the offense can hope to achieve by investing more effort. This usually does not involve a proof, but is merely an optimistic evaluation of potential threats.

The bottom of each range is "hard" in the sense that it reflects a result from the search. The top of each range is often "soft" since it represents a very optimistic appraisal that may have no basis as yet in the search results. A proof is involved in the bottom of the range since the defense always invests total effort until a satisfactory result has been achieved. This is based on the assumption about PARADISE's domain that one side can win material. The defense is always satisfied with equality or better.

Comparing values of positions and determining success is more complex when the values are ranges rather than integers. For example the alpha-beta algorithm becomes more complex when the alpha and beta values are ranges. (PARADISE's algorithm is described in section C of this chapter.) Figure 4.1 shows nine measures used in PARADISE's tree search when it is analyzing a particular node. (Nine is not as large as it seems, since each range produces two measures instead of one.) In the diagram, the offense is trying to achieve a positive score while the defense is trying to achieve a negative score.

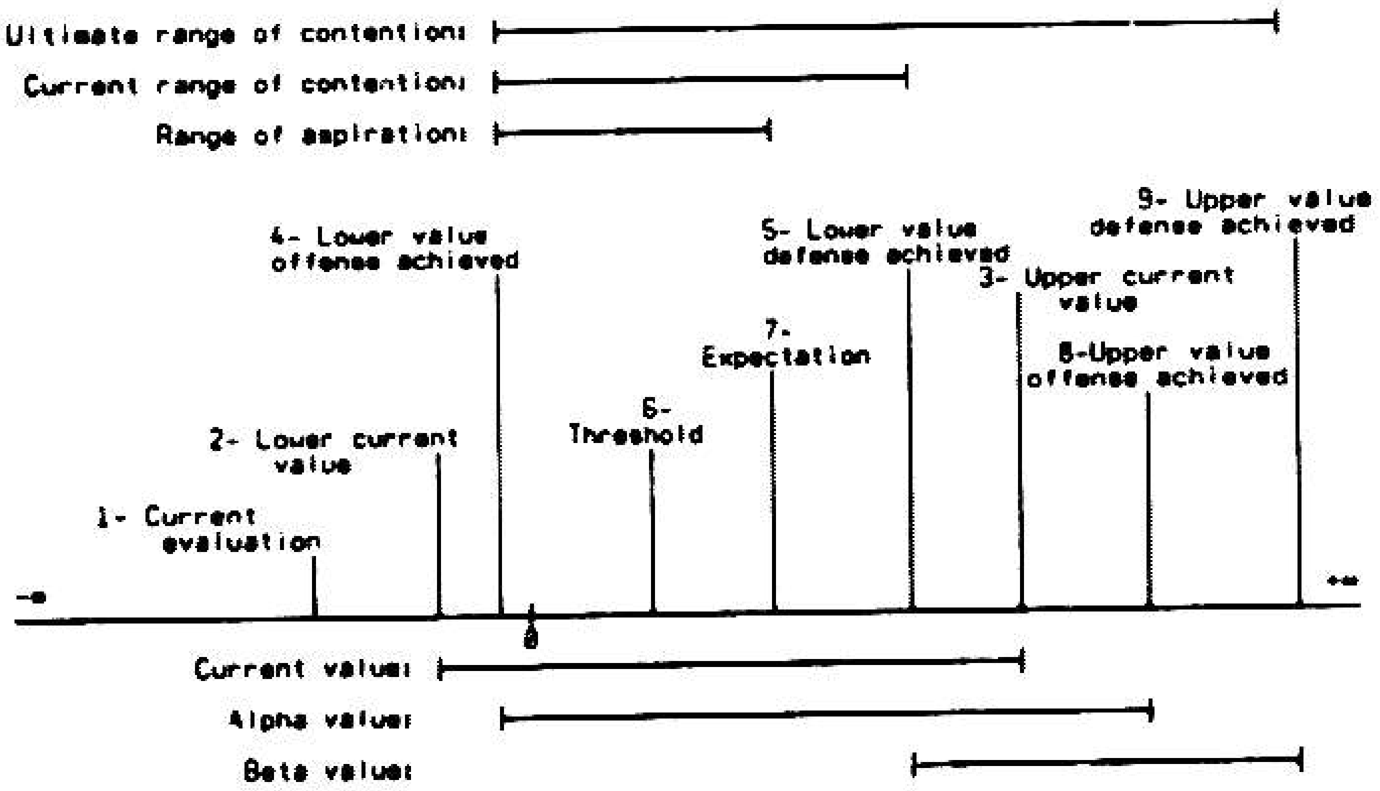

Ultimate range of contention:
Current range of contention:
Range of aspiration:
9- Upper value defense achieved
4- Lower value offense achieved
5- Lower value defense achieved
3- Upper current value
7- Expectation
8-Upper value offense achieved
6- Threshold
2- Lower current value
1- Current evaluation
-∞
+∞
0
Current value:
Alpha value:
Beta value:

Figure 4.1
Measures used in the search with ranges of contention and aspiration

Of these 9 measures, 6 are the two end points of 3 different ranges. Measures 4 and 8 define the range which represents the alpha value of the alpha-beta algorithm, while measures 5 and 9 define the range for the beta value. The third range is the current value of the position which is defined by measures 2 and 3. This range is not used by the search to control tree growth, but rather is the "answer" returned by the search. It is included in the diagram to show how the eventual result usually relates to the other measures. The other 3 measures are not parts of ranges. Measure 1 is simply the result of applying the evaluation function to the current board position. This value is used frequently in calculating how a particular threat relates to the other measures in the diagram. Measure 7 is the expectation of the current plan, and measure 6 is the threshold which the search uses to describe how much information it wishes to gain from the current search.

The relative ordering shown above is the one found at the majority of nodes, but many of the measures can have somewhat different relative orderings. These measures are defined in more detail in the following paragraphs (descriptions of the constraints on the relative ordering are included). The upper bounds of the three ranges (measures 3, 8, and 9) are frequently the same value in practice, and need not be carefully distinguished by the reader. Each measure listed is numbered and described below.

1- Current evaluation. This is merely the result of applying the evaluation function to the current position without checking for quiescence. It may have any relationship to the other values in the diagram since the true value of the position (after quiescence) may be infinitely different in either direction. When the offense is about to search a plan, the search requires that the current evaluation (1) added to the threat of the plan must be larger than the lower value achieved by the offense (4).

2- Lower current value. The purpose of the search is to define this value. It is the value the offense has shown it can obtain in this position. This value is not known until termination of the search at this node since it can only be calculated by searching until a quiescent state is reached. The offense is trying to improve this value to meet his expectation (7) while the defense hopes to keep this value below the lower value achieved by the offense (4).

3- Upper current value. The search is also trying to define this value. It is the best the offense could hope to obtain if its most threatening unsearched plan would completely succeed. The program does not calculate this value until termination of the search at this node. This value is never greater than the upper value the defense achieved (9) and never less than the expectation (7) or the lower current value (2).

4- Lower value the offense achieved. This is the bottom of some range which the offense has achieved on another branch of the variation leading to this node. This is part of the alpha value used by the alpha-beta algorithm. The offense has proved that he can achieve at least this much on this branch and is not interested in any node whose value will be less than this. This is the lower bound of both the current and ultimate range of contention. The search will never investigate nodes whose value is outside either range of contention. If a node is outside the ultimate range of contention it can be terminated forever, while a node which is only outside the current range of contention can be terminated with provision being made for possibly returning to search again. If this value is not defined, it is assumed to be minus infinity.

5- Lower value the defense achieved. This is the bottom of some range which the defense has achieved on another branch of the variation leading to this node. It is less than or equal to the upper value the defense achieved (9) which is the corresponding top of the range. The offense can do no better than this value unless he returns to the branch from which this value came and invests more effort to improve the result (possibly up to the upper value the defense achieved). This is used as part of the beta value for the alpha-beta algorithm. Any cutoffs made using this value are not final since the defense may not be able to achieve this value if the offense invests more effort. Any alpha-beta cutoffs made with this value insert information in the tree which can be used to undo the cutoff later. This value defines the upper bound of the current range of contention (though not of the ultimate range of contention). The expectation (7) may be greater or less than this value. The search is not interested in this node unless the lower value achieved by the offense (4) is less than this value. If this value is not defined, it is assumed to be plus infinity.

6- Threshold. This is the threshold defined by the best-first search stategy. The offense will be satisfied for now if it can achieve this much. The threshold is greater than the lower value the offense achieved (4). Once the threshold is attained, plans which are not likely to succeed are dropped from consideration. Even if the threshold is achieved, the search strategy may later return to this node to search it again with a higher threshold. At the start of a new problem, the threshold is initially set to the value of winning a rook for bishop exchange (i.e., slightly less than 2 pawns).

7- Expectation. This is the value the offense expects to obtain by searching the current plan. It is calculated by adding the threat of the current plan to the current evaluation (1). It may be larger or smaller than the threshold (6), but will never be out of the ultimate range of contention (4-9). This is the upper bound of the offense's range of aspiration for the current plan. If a plan is not searched, this expectation may be used to determine the upper current value (3). If the search ever achieves this expectation, it assumes no better result can be obtained for this plan.

8- Upper value the offense achieved. This is the top of some range which the offense has achieved on another branch of the variation leading to this node. It is never less than the lower value the offense achieved (4), and may be used to determine the current upper value (3). Like the lower value the offense achieved (4), this value is used as the alpha value in the alpha-beta algorithm. When this value is used, the cutoff is final. If this value is not defined, it is assumed to be plus infinity.

9- Upper value the defense achieved. This is the top of some range which the defense has achieved on another branch of the variation leading to this node. None of the other eight values is ever larger than this one. This value defines the upper bound of the ultimate range of contention. Like the lower value the defense achieved (5), this value is used as the beta value in the alpha-beta algorithm. When this value is used, the cutoff is final. If this value is not defined, it is assumed to be plus infinity.

C) Offensive search

1) Overview

An overview of the offensive search is presented in the flowcharts in figures 1.3 and 1.4 of chapter 1. This section describes in detail the processes referred to in these flowcharts. Let us assume the search is looking at a new position with the offense to move. It first tries to terminate based on the value of the position. If the current evaluation is as good for the offense as either the threshold or the expectation (which was calculated 2 ply earlier), then the quiescence search (described in section E of this chapter) is applied to see if this value holds. If so the search terminates without further investigation of this node, and the lower current value becomes the value returned by the quiescence search.

Whenever PARADISE terminates, it must put information in the tree to be returned for use should this line be searched again in an effort to improve the value for this node. For each point where the search may be re-initialized, a RE-SEARCH pointer is inserted in the tree. If the above termination happened because the expectation was met, PARADISE assumes that the value will never be improved upon. In this case the upper current value is the same as the lower current value and no RE-SEARCH pointers are inserted in the tree. Things are different if the expectation has not been met and termination is resulting from the threshold being achieved. In this case, the current upper value becomes the expectation and one RE-SEARCH pointer which states that a static analysis should be performed in this position (to obtain plans for improving the value) is inserted in the tree.

If the termination attempt has failed, the search tries to find the best plan for the offense. It would be simple to do a static analysis and use the plans suggested, but this is too expensive. To reduce the effort, the search uses the following four sources of plans (see figure 1.3): 1) plans demanded by the plan currently being executed, 2) plans which are obviously winning, 3) plans suggested (but not demanded) by the plan currently being executed, and 4) plans produced by a static analysis. Plans are suggested for searching by using these sources one by one in the given order until the search returns a successful result.

The current plan makes a demand when it specifies a particular move rather than a goal. (Most plans use goals like SAFEMOVE rather than specify a particular move.) This is the simplest and fastest way to get a plan for execution. The specified move is searched after checking that it is legal. With the exception of LEGALMOVE, the primitives are not calculated for the current position. These plans are tried first because they are inexpensive to obtain and because the current plan usually does not specify a particular move unless it is sure that it is the best move. Particular moves are typically specified in sacrificial plans where the branch which matches when the defense captures the sacrifice will specify a particular move in reply.

In general, the plan being executed has branches only for the best defensive moves. In most positions the defense has a large number of poor moves which it tries after its best ones have failed. Since the continuation in the current plan was meant to answer the better defensive moves, it will many times not be the best plan against a poor defensive move. (For example, if the defense moves his queen en prise as a last ditch effort, the offense should capture the queen rather than continue with its plan which wins a rook.) For this reason, the search uses obviously winning plans as its second source of plans before trying the plan currently being executed. Obviously winning plans are found by posting a QUIESCENCE concept in the data base and executing it. The QUIESCENCE KS eventually produces plans for any obvious attacks that work. This is essentially a small and inexpensive static analysis with limited knowledge. Most of the expense of this calculation stems from calculating the primitives for the current position, but these need to be calculated before executing the current plan in any case. This same calculation is used to generate moves in the quiescence search.

Many times one of the obviously winning plans is specified in the plan currently being executed (PARADISE will match a particular move to its corresponding goal such as SAFEMOVE, SAFECAPTURE, etc.). When PARADISE finds this to be the case, it skips the search of obviously winning plans and goes directly to searching the current plan. When searching an obviously winning move, PARADISE will not do a static analysis any time during the search. If the plan is obviously winning, the win should be discovered by executing the plan and either doing a quiescence search or trying more obviously winning plans. This keeps the search from growing a large tree while searching an obviously winning move, but in some cases the win may not be found. For this reason, PARADISE keeps track of which plans were searched as obviously winning plans and may search them again (if everything else fails) to obtain a better value (allowing static analyses the second time). This way of searching obviously winning plans helps PARADISE get correct results with less effort. This is largely due to the fact that PARADISE's analysis of obviously winning plans is so accurate that they rarely fail. When they do fail, a different win is often found, so searching the failed plan again is still unnecessary.

The search uses the current plan as its third source of plans. The current plan may provide more than one goal, and each goal may produce a number of elaborated plans after execution. The search must decide which plan is the best one to search first and it must also detect plans which are not working and abandon them before more effort is wasted. As mentioned in chapter 2, the plans have been constructed to express their purpose accurately. This is the principal method for abandoning plans which aren't working. The plans themselves simply stop suggesting things. Non-working plans are also abandoned by the cutoffs described later in this section and by evaluating their threat language expression in the context of the current position to see if they are still threatening. This re-evaluation of threats also helps select the best elaborated plan from among those suggested by execution of a goal in the current plan. In general, the search will execute goals in the current plan in the order they are given except that workable goals are tried before non-workable goals. A goal is workable if the move it is trying to suggest is both legal and safe. (The system knows which move is trying to be suggested by goals such as SAFEMOVE and SAFECAPTURE.)

The fourth source of plans used by the search is static analysis. This is used only as a last resort since it is computationally expensive. It cannot be avoided in many cases because it is necessary to find threats which the opponent has permitted by his last few moves. Such threats cannot always be foreseen at the top level. A static analysis is accomplished by posting a THREAT concept in the data base except when the offense is in check. In this case a DEFENDMOVE concept is posted for each defensive move which captures the offensive king. The static analysis process is described in chapter 3, and chapter 2 briefly describes the ordering function used to order the plans produced by the static analysis. The best-first search strategy will sometimes terminate the search before a static analysis is done. This is described in section G of this chapter.

Once a plan is selected from one of the four sources mentioned above, it is sent to the SEARCH routine which is flowcharted in figure 1.4 and described in the next section.

2) The SEARCH routine

The SEARCH routine controls the actual searching of all the plans produced by the current source of plans (described in the last section). It attempts to use all available information to ensure that these plans are reasonable to search. It may use available information to improve a plan before search. If a plan does not succeed, an analysis of this failure is undertaken in an attempt to discover the problem. Each action taken by the SEARCH routine is described below, roughly in the order in which they are taken.

REPETITION

If the current position has occurred earlier along this line of play, the search is terminated immediately and both the lower and upper current values are set to 0. PARADISE assumes the defense will be satisfied to claim a draw by repetition.

SET EXPECTATION

The SEARCH routine orders the plans from the current source according to their expectation. In the following descriptions, the best plan according to this ordering is the current plan and the current expectation is the expectation of this best plan. An expectation is calculated from an attribute list by the algorithm described in section C-4 of chapter 2. Briefly, it takes the minimum (since the defense wants a negative score) of the following two calculations: 1) a normal evaluation of SAVE added to a normal evaluation of THREAT, and 2) a normal evaluation of SAVE added to an optimistic evaluation of THREAT with a normal evaluation of LOSS subtracted from this. The idea is that the defense has the choice of defending against the offense's threat (case 1) or using counter threats (case 2) in which case the offensive threat becomes more powerful. These evaluations occur in the context of the current position even though the Threat-Language expressions being evaluated may have been formed while analyzing a previous position. Given the calculated expectation of each attribute list of a plan, the system chooses the best (maximum) one which comes from a "likely-to-succeed" attribute list as the expectation of the plan. This means that if some attribute lists have a LIKELY of 0, then ones with a LIKELY of 1 are not considered in the calculation. (If the expectation is too low, only the LIKELY 0 attribute lists are terminated by the search so the current plan may be searched again with its LIKELY 1 attribute lists being best.) The current expectation is never permitted to be higher than the upper value achieved by the defense.

SAME PLAN AT HIGHER LEVEL

Most chess programs invest effort searching a number of moves in every possible order. Many times a move which looks good in the original position fails and is then tried 2 ply deeper in the tree (after an alternative move in the original position) since it still looks good. Frequently, the alternative move will have no affect on the original move and so it is doomed to fail at ply 3 just as it did at ply 1. Nevertheless, most programs will search it at ply 3 (and ply 5 if they get that deep). Human masters know why a move has failed and do not search it again until something makes the move look more promising.

Many times PARADISE will not search such a move again at ply 3. It does this by keeping track of all the plans suggested as it goes down the tree. If the current plan is the same as one suggested earlier and looks no more promising now, then it is terminated without searching. (The SEARCH routine starts over and calculates a new expectation for the remaining plans.) A plan looks "no more promising now" if all its attribute lists are the same as the other plan's. If all attribute lists are the same, then the two plans have been suggested for exactly the same reasons and have the same THREAT, LIKELY, SAVE, and LOSS attributes. In such cases it is fairly certain the plan will not succeed in the current position if it fails earlier along the same line. This certainty rests on the assumption that these are tactically sharp plans. In fact, when PARADISE suggests a plan that is more a positional suggestion than a sharp tactic, it inserts an attribute which excludes the plan from this cutoff. Another reason this cutoff does not introduce unacceptable error is that PARADISE's analysis is sophisticated enough to know why a plan has been suggested. This analysis includes a detailed description of a plan's expectation, and any improvement in the situation will generally change this expectation. Termination by this cutoff does not affect the current value or insert RE-SEARCH pointers in the tree.

PLAN NOT LIKELY TO SUCCEED

If the LIKELY attribute of the current plan is greater than 1 (i.e., the opponent has 2 or more non-forced moves), then this plan is terminated without searching. (This is actually enforced in the static analysis since PARADISE's static analysis no longer produces plans with a LIKELY greater than 1.) PARADISE will not concatenate 2 plans with LIKELYs of 1 along the same line. This would have the same effect as trying a plan which has a likely of 2 or larger. Once again the validity of this cutoff rests on the tactical sharpness of the domain and on PARADISE's accurate calculation of LIKELY values. In positional situations, the concept of a forcing move would not be an appropriate measure of likelihood of success. There are certainly tactical combinations that have more than 1 non-forced defensive move. It is hard to say how important such combinations are to good chess, but this restriction has not hampered PARADISE in any of the 100 test positions. It should be noted that the concept of a forcing move is fairly general in PARADISE. It includes not only attacks on pieces which must move to safely, but also threats to get a piece to an important square. Thus the "not-likely" restriction is not as restrictive as it may seem.

PARADISE also refuses to search plans with a LIKELY of 1 when it has been shown that the offense can achieve the initial threshold value (i.e., the offense will be at least rook for knight ahead). This cutoff does not affect the current value or insert RE-SEARCH pointers in the tree. This is part of the best-first search strategy and is discussed in more detail in section G of this chapter.

INSERTION OF RE-SEARCH POINTERS

Whenever the search is terminated, RE-SEARCH pointers may be inserted in the tree to be returned. This action is described here so it can be referred to throughout this section. To insert RE-SEARCH pointers, PARADISE creates one pointer in the tree for each plan produced by the current source of plans which has not yet been eliminated from consideration. This pointer is marked as a RE-SEARCH pointer and contains the plan to be searched with all its attribute lists. Whenever such a pointer is inserted in the tree, the expectation of its plan is calculated. This calculation uses the best value from any attribute list, whatever its LIKELY value. If this expectation is higher than the upper current value, then the upper current value is increased to reflect the possibility of achieving this expectation. If a static analysis has not yet been done in this position (i.e., we are searching plans from one of the first three sources), then a RE-SEARCH pointer is also inserted whose plan is the word "ANALYZE". This tells the system it should do a static analysis in this position to obtain plans for improving the value. Inserting an ANALYZE pointer requires the upper current value to be at least as high as the expectation inherited from 2 ply earlier in the tree. This process accurately documents all unexplored possibilities in the tree and changes the range of current values to reflect these possibilities.

One action often performed in conjunction with inserting RE-SEARCH pointers is the removal of the most likely attribute lists. This action is taken when the system knows that no plan with the current best LIKELY value can succeed. It consists of removing any attribute lists with the same LIKELY as the most LIKELY attribute list of the current plan. This is done to all plans under consideration. If a LIKELY 0 plan is current, only LIKELY 1 attribute lists are kept. If all attribute lists of a plan are removed by this process, the plan disappears (i.e., is no longer under consideration and will not produce a RE-SEARCH pointer). Thus if a LIKELY 1 plan is the best under consideration, removing the most likely attribute lists would result in elimination of all plans from consideration. An ensuing insertion of RE-SEARCH pointers may still produce an ANALYZE pointer however.

FORWARD PRUNE

If the current expectation is less than the lower value achieved by the offense, the system removes the most likely attribute lists. Since the best plan has an expectation lower than a value already achieved by the offense, there is no reason to search it. Only attribute lists with higher LIKELY values could have a better expectation so all attribute lists except these are eliminated. The SEARCH routine starts again with the plans remaining under consideration, but these plans are less likely to succeed and will often be cutoff by the "better plan at higher level" cutoff described below. This kind of forward prune introduces errors but PARADISE avoids mistakes in its domain because of its accurate analysis of expectations which tend to be optimistic. Programs with less knowledge often introduce unacceptable error with such a cutoff.

ALPHA-BETA

If the value achieved by the defense is not larger than the lower value achieved by the offense, all plans are removed from consideration. In this case the search has already achieved a value as good as it can hope for. If this condition is true for the upper value achieved by the defense (which is rare), then this node is outside the ultimate range of contention and no RE-SEARCH pointers are in: rted. If this condition is only true for the lower value achieved by the defense (as is usually the case), then this node is outside the current range of contention and RE-SEARCH pointers are inserted for all plans.

NOT AGGRESSIVE

PARADISE assumes the offense is trying to win material. If the current expectation is less than the value of the original position (at the root node), then the current plan is not aggressive and the most likely attribute lists are removed. The SEARCH routine starts over again with the remaining plans. This is another type of forward prune and the discussion of error given there applies here.

BETTER PLAN AT HIGHER LEVEL

This cutoff helps implement the best-first search strategy by terminating when there is a more promising node elsewhere in the tree. As the search goes down a branch, it carries along a list of untried alternatives. This list specifies, for each previous ply in which the offense is on move, the expectation of the best untried alternative. The current expectation is compared to these to see if this is the best node to search. This comparison is not based strictly on the values of the expectations since PARADISE is reluctant to abandon the current line because of the cost involved if it becomes necessary to search this line again. Section G of this chapter describes how this reluctance is implemented and the issues involved.

The following three conditions will prompt the search to decide not to search from this node in favor of a better plan elsewhere: 1) the current plan is LIKELY 1 and there is a LIKELY 0 alternative, 2) the LIKELYs of the alternatives and the current plan are the same and the expectation of an alternative is enough greater than the expectation of the current plan to overcome the system's reluctance (described in section G) to switch lines, and 3) the lower value achieved by the defense is less than the expectation of an alternative (again modulo the system's reluctance). Whenever one of these conditions is true, PARADISE inserts RE-SEARCH pointers for all plans under consideration and terminates the search.

There are two important restrictions to this cutoff as described. First, this cutoff is not applied when the source of the plans under consideration is the plan currently being executed. To apply this cutoff the current plan must be an obviously winning plan or must have been produced by a static analysis. This permits a plan to finish once it starts. If the "better plan at higher level" cutoff were applied to plans in the middle of their execution, the system would often not follow up on a sacrifice because another plan looks better after the defense takes the sacrifice. Second, if the offense is in check, the expectation of the current plan is assumed to be the expectation calculated 2 ply earlier in the tree. This is done because expectations may not be accurately calculated when the offense is in check. An alternative to this would be to eliminate replies to check from this cutoff, but sometimes the defense captures a piece with his checking move leaving the current position so poor that the offense will want to abandon it for a better plan elsewhere.

CAUSALITY

PARADISE has a causality facility which determines the possible effects a move might have on a line of play. Using a tree generated by the search, the causality facility looks for effects a proposed move might have (such as blocking a square which a sliding piece moved over, vacating an escape square for the king, protecting a piece which was attacked, etc.). It is described in detail in section F of this chapter, and is used here to reject plans which are doomed to failure. An example of this would be when the defense has a 3 move forced mate unless the offense plays the one correct defensive move. Most chess programs will not immediately recognize a 3 move mate and defend against it, so they try many attacking moves for the offense and for each one they discover that the offense gets mated. The tree which shows the mate, which may be quite large, is regenerated each time. By using its causality facility, PARADISE avoids much of this behavior. After the mate is discovered the first time, PARADISE uses the causality facility to compare any proposed move to the tree which shows the mate. If causality determines that the mating line will not be affected by the proposed move, then the move can be eliminated without searching.

If the offense has already searched a plan from this node which has failed, the causality facility is used to check that the current plan has some hope of avoiding this winning defensive line. This process involves checking if the first move made by the offense in the line permitted the defense to win, and, if it didn't, checking that the current plan can somehow affect the winning defensive line. If the causality facility determines that the current plan cannot prevent some winning defensive line, then the current plan is removed from consideration and the SEARCH routine starts again with the remaining plans.

The causality facility can only be applied to certain trees which have been produced by searching plans at this node. Causality is not applied to the following trees: 1) trees in which the offense has come out even or ahead but is not satisfied since the threshold hasn't been achieved (i.e., the defense must actually win), 2) trees produced by the quiescence search (since errors are more likely in the narrow quiescence search), 3) trees produced by the search of an obviously winning plan (since the prohibition on static analyses in these trees may introduce errors), 4) trees in which a relevant line was terminated by a "better plan at higher level" cutoff (since the offense may do better with more effort), and 5) trees in which a relevant line was terminated by repetition of position.

LEMMAS

PARADISE has an extremely limited facility for posting lemmas about a position. A "lemma" is some postulation the system has made about the situation which is used in reasoning processes at nodes other than the one which posted the lemma. The current node has access to lemmas posted by other nodes at the same depth in the tree which descended from the same offensive move 2 ply earlier (i.e., nodes created by following the current line and allowing the last defensive move to be different). A lemma in PARADISE can have only one form. It states that if the intention is to play a particular move, MOVE1, then MOVE2 should be played instead but only if MOVE2 captures the particular piece P1. If P1 is not instantiated, then the condition at the end always matches. When such a lemma has been posted (which rarely happens), its effect is much the same as that of the "killer" heuristic. Namely, the move that worked best at some other node at this level is tried first at the current node. Some effort is made to ensure that the "killer" move has some of the same features now as it did when it was the "killer". This is done by checking the condition on P1 and by ignoring the lemma if the defense moved P1 on his last move.

These lemmas are posted only in certain situations. If a plan has been searched and has failed, and an analysis of the refutation (described below) suggests a move which has not otherwise been suggested by the analysis, and this move succeeds, then (and only then) PARADISE posts a lemma of the above form. This rarely occurs since PARADISE usually suggests the correct plan for the offense without analyzing refutations. This facility is an unimportant part of PARADISE but is significant because it lays the framework for future research. One promising future development which could be pursued is a language for expressing stronger lemmas. Mechanisms like those described here could use these lemmas to communicate knowledge from one node in the tree to another.

PLAN ELABORATION

If the first action in the current plan is a goal (rather than a particular move), the SEARCH routine posts a instance of this goal in the data base by using the instantiations provided by the goal and the attribute lists of the current plan as a whole. This goal instance in the data base causes the corresponding KS to be executed. This may cause other KSes to be executed but eventually results in plans being suggested. This can be viewed as a small static analysis that is focused on one particular aspect of the position. If no plans are produced by this analysis, the current plan is eliminated from consideration and SEARCH starts over with the remaining plans. If one plan is suggested (which is often the case if the plan is working), the current plan is elaborated by replacing its first action with the plan suggested by this analysis and SEARCH continues. If more than one plan is suggested, a number of elaborated plans are placed under consideration and SEARCH starts over again so that these plans will be ordered during the calculation of expectations.

SEARCH AND TEST SUCCESS

After plan elaboration, the current plan begins with a particular move. This move is now made, creating a new board position. The defensive search (described in the next section) is called on this new position. It returns a range of values for the position and the tree generated to determine these values. This tree becomes part of the tree returned by the SEARCH routine. If the lower value returned by the defensive search is equal to or greater than either the threshold or the current expectation, the search is terminated at this node. The lower current value becomes the lower value returned by the defense, and the upper current value becomes the upper value returned by the defense although this may be raised by the insertion of RE-SEARCH pointers in the tree. If the expectation has been achieved, the system removes the most likely attribute lists before inserting RE-SEARCH pointers for the remaining plans. In this case, no likely plan can expect to do better than the result obtained. If only the threshold has been achieved, RE-SEARCH pointers are inserted for all plans under consideration except the current one.

ANALYSIS OF FAILURE

If the search of the current plan showed no gain for either side or an offensive gain not large enough to meet either the threshold or expectation, then the current plan is removed from consideration and SEARCH starts over with the remaining plans. If the search of the current plan shows the defense winning material, an analysis of the failure is undertaken. A DEFENDMOVE goal is posted for the best defensive reply to the first move of the current plan. DEFENDMOVE goals are also posted for every checking move made by the defense on relevant branches of the tree produced by searching the current plan. Patterns describing checks are included in the tree so these checking moves can be determined without restoring positions which occurred in the tree. Executing these DEFENDMOVE goals produces plans which attempt to prevent or lessen the effects of the moves specified in the goals, i.e., counter-causal plans. If there is a move which can prevent the defense's winning line, this counter-causal plan generation will almost always suggest it. Almost any plan which might help is suggested and there are usually a large number of them.

In the defensive search (described in section D), the defense tries to prevent offensive successes in exactly the above manner. The defense tries all plans which are suggested. The offense, however, is more interested in finding the correct offensive move than the correct defensive move. Therefore it only tries the suggested counter-causal plans which begin with captures. Good defenses which don't start with captures will be missed, but the offense cannot afford to search all proposed defenses. More defensive knowledge could improve the system's performance since good counter-causal moves (including non-captures) could be searched if many poor ones were not suggested. This deficiency is due to PARADISE's philosophy of generating defensive moves quickly (using little knowledge in the process). Searching only counter-causal captures has hurt PARADISE's performance only once in the first 100 test positions (see chapter 6). The correct offensive plan is usually waiting to be searched or will be generated by the next source of plans.

If a counter-causal capture is suggested, it is searched immediately. If it succeeds, the search is terminated in the same manner described above, and a lemma is posted for the lemma facility described above. If there is no success after trying counter-causal moves, then the SEARCH routine starts over and investigates the plans still under consideration.

3) What controls the search?

The underlying reason for the success of PARADISE's searching paradigm is the system's ability to view each node in the context of the more global problem environment. The search may be terminated at any node if the node's relationship to the problem environment warrants it. The various mechanisms used by the offensive search were described in the last section, but little was said of the importance of each. This section briefly describes the importance of each mechanism to the overall performance of the search.

Any reasonable tree searching paradigm in a chess program will check for repetition of position and use something equivalent to the alpha-beta algorithm. These are essential. What is referred to as "alpha-beta" in PARADISE is different than any algorithm used before since the alpha and beta values are ranges. PARADISE's algorithm compares the current value to these ranges in the proper way and, if an alpha-beta cutoff is justified, interacts with the best-first search strategy by inserting the appropriate RE-SEARCH pointers and updating values to show the untested potential. These extensions of the alpha-beta algorithm, which produces many cutoffs, are essential to the performance of the search. The repetition of position check rarely produces a cutoff, but prevents an infinite loop when it does since PARADISE has no depth limit.

Three additional mechanisms in the SEARCH routine are absolutely necessary for the search to be knowledge-controlled. The first is the "better plan at higher level" cutoff. This is the implementation of the best-first strategy and is the primary mechanism for giving the search a global view of the problem environment. Another necessary mechanism is the use of the threshold value to stop the search. Using this threshold implies that the values must be ranges and that lines must be searched again since the true value of a node is not found. The search could not be knowledge-controlled if it attempted to find the true value of every node it generated. The "not likely to succeed" cutoff is also necessary. Without it, PARADISE would not be able to control the concatenation of poor plans which could make arbitrarily deep searches. All three of these mechanisms produce frequent cutoffs.

The forward prune cutoff is important though not as critically as those mentioned above. It is not critical because the search would usually be terminated anyway by either the "better plan at higher level" cutoff or because the threshold has been achieved. It is important because it does not insert RE-SEARCH pointers for the pruned plans. This may save much effort should this node need to be searched again for a better value. This cutoff occurs fairly frequently.

Two mechanisms in the SEARCH routine are not critical for the offensive search but are absolutely necessary for the defensive search (described in the next section). These mechanisms analyze findings returned by searches and help the defense overcome its lack of knowledge. The first is the causality facility which rarely makes a cutoff in the offensive search since the defense rarely has a line which thwarts many offensive attacks (at least in PARADISE's limited domain). The second is the process of generating counter-causal moves by analyzing trees (this rarely finds the correct offensive move).

Three other mechanisms are not critical for the offensive search and do not occur in the defensive search. The "same plan at higher level" cutoff is helpful when it produces a cutoff but this is rare. A plan is rarely the same after a few moves have been made and PARADISE's analysis realizes this. The lemma facility is very restricted and rarely used. Lastly, the "not aggressive" cutoff is not important, primarily because the analysis rarely produces plans which don't threaten anything.

D) Defensive search

1) Introduction

The philosophy of the defensive search is to use limited computational resources by suggesting all reasonable defenses without using a large amount of knowledge to screen the better ones from the poorer ones. A major reason for adopting this philosophy is that the proofs generated by the search to show an offensive plan wins are much more convincing if the winning line is shown for all reasonable defenses. Another reason is that the correct defense can be subtle and might be missed if the defensive search is more selective. The defensive search puts some effort into finding the best move to try first, but after that it is content to do nothing and wait for unfavorable results to be returned from searches. It then analyzes these results in order to find the correct defense. This involves the same counter-causal analysis used by the SEARCH routine.

By waiting for something bad to happen and then responding, the defensive search always suggests moves that are somewhat relevant to the problem at hand. This is a considerable improvement over programs which do not do a causal analysis of the results of a search since they must search many irrelevant moves. PARADISE still tries many defenses that a human master would not consider, but this cannot be helped until the decision is made to apply larger amounts of knowledge to the defensive analysis. The analysis of the causality facility prevents many of these poor defenses from being searched.

2) The actual defensive search

The defensive search starts by trying to find the best defensive move. If there is only one legal move, this is played immediately without computing most of the primitives. In all other cases, a QUIESCENCE concept is posted in the data base and executed as a goal. This will suggest any attacking threats the defense might have. These are ordered in the same way the offensive static analysis orders plans, except when the defense is ahead in which case an algorithm which favors captures is used. This is done to avoid checks so the offense will quickly terminate the search instead of replying to checks. This algorithm also favors checks over moves which neither capture nor check. All plans suggested are placed under consideration unless they do not threaten to attain the lower value the defense achieved.

Unless the current evaluation is worse for the defense than the lower value the defense achieved, the system next tries to find good defending moves to consider along with the attacking ones. If the defense is in check, this is done by posting a DEFENDMOVE goal for the move which captures the king. Otherwise, the defense selects the alternative in the offensive plan which answers the most general defensive move (the details are not important). The defense converts the first goal in this alternative to a move. Unless an already suggested attacking move negates the effects of this proposed offensive move (this determination is described in section E of this chapter), a DEFENDMOVE goal is posted for the proposed offensive move. These goals are then executed and the suggested plans are placed under consideration (along with the already suggested attacking plans). The system keeps track of which DEFENDMOVE goals have been executed so they will not be repeated during the counter-causal analysis (below).

If the defense has pieces en prise, the one involving the greatest loss is also chosen for defending against. If the effects of the offense's most threatening capture are not negated by the defense's best plan under consideration, a DEFENDMOVE goal is posted for this capture. In this case, the plans suggested are ordered and only the best one is placed under consideration. Because of this, PARADISE will allow this DEFENDMOVE goal to be executed again.

If, at the end of this whole process, no plans have been placed under consideration, PARADISE posts a DEFENDMOVE goal for some safe check the offense has (if one exists). Any plans suggested are placed under consideration. If still no plans are under consideration, desperado moves for en prise pieces and mobil captures which check are suggested. All plans placed under consideration are now ordered with the same function used to order plans in the offensive static analysis. This function takes into account both the THREAT and SAVE attributes (as well as LOSS and LIKELY), so that both attacking and defending moves are ordered reasonably for searching.

All the above analysis attempts to find the best move to play first. The search is now ready to be initiated, and PARADISE will return to this point each time it searches a plan in order to repeat the following steps. If the current value is so poor that the defense's expectation is worse (for the defense) than the lower value the defense achieved, then the first move under consideration is rejected. This cannot happen if the current evaluation is as good for the defense as the lower value the defense achieved (which is usually the case). This cutoff is not completely straightforward since the defense has not invested much effort analyzing threats and this must be counteracted (since its expectation may be low). The LOSS attribute of the first plan under consideration is not considered (making it look more threatening), and the threats of all plans under consideration are added together. If the defense is in check, attacks which would be legal if the defense were not in check are also considered.

The best plan under consideration is now searched. This involves creating a new board position and calling the offensive search which returns a tree. If no plans are under consideration and no search has yet been initiated from this node, the offensive search is called without changing the board position, letting the offense make two moves in a row. This is done to find the offense's best line for use in counter-causal analysis. This is called a null move analysis and is described in more detail in the next section.

Three conditions allow the defensive search to terminate without investigating all moves under consideration. The search is terminated if the lower value returned by the last search is as good as or better than the value of the original position, or if the lower value shows that the defense is ahead, or if the lower value shows the defense has done no worse that the lower value achieved by the offense. Termination involves putting all the required information in the tree (e.g., ordering all searches attempted, inserting certain patterns matched in this position) and backing up.

If the search is not terminated, the upper and lower values achieved by the defense are updated with the results of the last search. A counter-causal analysis (described in section C of this chapter) is then done on the results of the last search. Any DEFENDMOVE goals generated are ignored if they have already been executed. The plans produced by the counter-causal analysis become the first plans under consideration and the defensive search returns to the point from which it initiates searches.

By this process of continually thwarting each offensive line, the defense will eventually try its best defensive move. One complication arises when each move tried by the defense has permitted the offensive line which refuted it. In other words, each move the defense has tried has been a poor move which gives the offense a new attack. In this case, the offense may be able to refute each move without playing the principal variation it would use against the best defense, so no counter-causal analysis will suggest the best defense. The defensive search checks for this situation. If no plans are under consideration and each move that has been searched permitted its own refutation (the causality facility determines this), then the defensive search does a null move analysis (if one has not already been done). By allowing the offense to make two moves in a row, the principal variation will be discovered and the counter-causal analysis will find the best defense. An example of this is shown in the next section.

3) Example of a null move analysis

On certain occasions, the defensive search permits the offense to play two consecutive moves so that it can analyze the result. The process is referred to as a null move analysis, and has not been used effectively in previous chess programs. PARADISE does not let a null move analysis affect a quiescence analysis. If the offense decides to terminate and do a quiescence search rather than play a second consecutive move, the quiescence search recognizes the null move and plays first for the defense in the quiescence analysis.

PARADISE uses the null move analysis in two different situations. The defensive search invests a substantial amount of effort trying to find the best defensive move at a new node (as described in the last section). If this analysis fails to suggest any moves, then the defensive search does a null move analysis rather than try some randomly chosen move. The idea behind this is that the null move analysis will return the main variation so that a counter-causal analysis will quickly find the best defense, while a randomly chosen move will likely start an unimportant variation that should probably not need to be searched. The second situation in which a null move analysis is done is described in the previous section. Briefly, if no moves are currently under consideration and the moves already searched permitted their own refutations, then a null move analysis is done. Again this is an attempt to find the main variation when poor defensive moves have so far uncovered only unimportant variations. An example of this situation is given below.

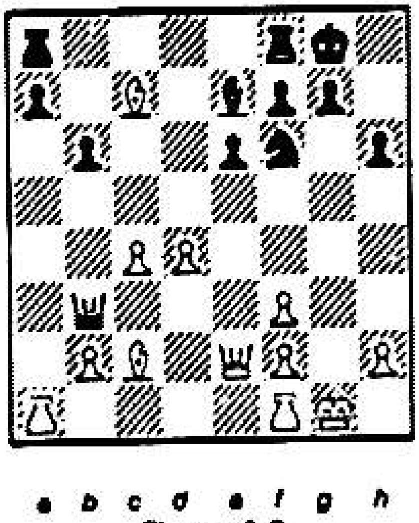


Figure 4.2
black to move

The position in figure 4.2 derives from position 80 in [Reinfeld58] and is one of the positions PARADISE has been tested on. White has just moved his bishop from d3 to c2 attacking the black queen. The defensive search must find the best defense for black and should at least try moving the black queen to b2 and to b4. When PARADISE's defensive search initially analyzes this position, only one plan is suggested. Q-b2 is suggested because it captures the white pawn and also moves the black queen to a "safe" square. The offensive plan has expired so no plans are suggested to thwart it, and only the plan which the system considers best (namely, Q-b2) is suggested to save the queen since PARADISE is not sure this is a major problem.

Q-b2 is searched and quickly loses to the discovered attack, WB-h7ch. The counter-causal analysis tries to stop B-h7 but all suggested moves are rejected because they do not save the black queen or instigate an attack on white. The analysis realizes the white queen captured the black queen on b2 in the previous search, but this produces no plans because in the position in figure 4.2 it does not look possible to get the white queen to b2. It would be an error to quit without trying any other defensive moves because Q-b2 actually permits the discovered attack by the white bishop.

PARADISE recognizes this situation. The causality facility determines that Q-b2 permitted white's attack, so the defensive search does a null move analysis. In this analysis the white bishop takes the black queen, and the counter-causal analysis of this line suggests many moves, including Q-b4. In this way PARADISE finds all reasonable defenses in this position.

This example is somewhat deceiving. PARADISE does not rely on a null move analysis to save en prise pieces. The program has been tested on 92 problems, and it did fewer than 15 null move analyses in these 92 search trees. The null move analysis in the above example occurs because three special conditions are present. One, the offense does not have a plan. Frequently the offensive plan will specify capturing an en prise piece if it doesn't move, and this results in all defenses being suggested. Two, no attractive defensive moves exist so none are tried which get answered in such a way that the en prise condition of the black queen appears important. Three, the defensive search thinks Q-b2 is the best move to save the queen. If it thought Q-b4 was best then both moves would be tried since Q-b2 is also suggested because of its attacking possibilities. The system usually does a good job of picking the best defense.

E) Quiescence search

The quiescence search attempts to determine the true value of a position, although a range may be returned. In any position, either side may have threats which could alter the apparent balance of material. PARADISE attempts to take this into account by playing out a sequence of moves to investigate the best threats. Both sides are treated exactly the same; the quiescence search makes no distinction between the offense and the defense as the regular search does. The program uses its knowledge to statically determine which threats are best, and then plays only the best threat or best defense for each side during the quiescence analysis. At nodes where a defending move is played, a second move is sometimes tried when the first fails. Thus the tree produced by the quiescence search is mostly linear with occasional nodes that have a second branch.

It is possible for PARADISE to recover when the quiescence search does not find an existing winning attack. During the quiescence search, the knowledge-based analysis may post concepts which describe the position as non-quiescent in which case the system recognizes that the result may be significantly improved for one side if a full search were employed. Similarly, whenever the quiescence search returns a range, the program may search the node again using a full search to find more attacks.

Most chess programs try only captures and checks in their quiescence searches, but PARADISE notices a wide range of attacks, many not beginning with a capture or a check. The quiescence search begins by using the QUIESCENCE KS to find threats for both sides. This KS notices captures, many kinds of forks, pins, skewers, discovered attacks, and many kinds of mating attacks. Under certain conditions, it will even suggest a sacrifice to set up a fork. Checks simplify the algorithm somewhat. If the side on move is in check the system does not look for any threats, but posts a DEFENDMOVE goal to reply to the check. If the best threat of the side on move is a check, then PARADISE plays this check without looking for the threats of the other side. Selecting the "best" plan involves using the THREAT, LIKELY, LOSS, and SAVE attributes in an algorithm almost exactly like the one described in chapter 2. The only difference in the algorithm used in quiescence searching is that captures are favored slightly over non-captures.

Having found the threats for both sides, the quiescence search looks for a plan by the side on move which is better that any plan of the side not on move. The system must check for cases where the plan by the side on move will cancel the effect of a plan of the side not on move. A plan is considered better than a plan of the side not on move if any of the following are true (hereafter the side not of move will be referred to as the "opponent"): 1) the first move of the olan checks, 2) the plan is more threatening than the opponent's plan (this calculation involves the LOSS, LIKELY, SAVE, and THREAT attributes), 3) the first move of the plan captures the first piece to move in the opponent's plan, 4) the first move of the plan either adds a protection to the destination square of the first move of the opponent's plan or captures a piece which protects this destination square, and the destination square is not overprotected against the side on move (OVERPROTECTED is a primitive function), and the plan is more threatening than the opponent's plan when the opponent's THREAT attribute is changed to only give credit for the exchange value of the first move, 5) the same situation as 4) occurs except the destination square is overprotected and the first move of the plan both adds a new protection to that square and captures a piece which protected it, and 6) the first move of the plan moves a piece which the opponent's first move was to capture, and the plan is more threatening than the opponent's plan when the opponent's THREAT attribute has the expression for the initial capture removed.

Situations 4) and 5) attempt to recognize the case where the side on move may be able to prevent the opponent from carrying out any threat which extends beyond his first move. For this reason the opponent's THREAT must be changed to reflect only his first move. In this way the system notices that an opponent's knight move to an empty square for the purpose of forking two pieces will have its effect cancelled, while an opponent's knight move which captures a queen will not have its effect cancelled. Situation 6) attempts to recognize the case where the effect of the opponent's first move may be cancelled, but longer range effects may not be cancelled. Again the THREAT attribute of the opponent's plan must be changed to reflect the fact that only the first part of the threat has been cancelled. In this way PARADISE notices that an opponent's knight move which captures a queen will have its effect cancelled (if the queen moves), while a knight move which threatens to capture a pawn and mate will not have the mating threat cancelled just because the mating move no longer captures a pawn.

If no offensive plans have been suggested for either side, then the search is terminated and the current value is returned as both the top and bottom of the range. If the side on move has a better plan than any plan of the side not on move, then this plan is selected as an offensive move. If no such plan exists and no special conditions exist (described below), then a DEFENDMOVE goal is posted to defend against the best plan of the side not on move. The best plan produced by analyzing the DEFENDMOVE goal is selected as a defensive move. If no plans are suggested, a DESPERADO goal is posted and the best plan suggested during its analysis is selected as a defensive move. If still no plans are suggested, then the side on move skips his move, letting the other side make two moves in a row. This differs from a null move analysis in that a counter-causal analysis is never done on the result and a move is never tried for the side on move. This situation occurs when the other side has a one move mate and no defensive move in the current position can prevent the mate.

One special condition recognizes the case when the side not on move could capture a piece with gain, but has no real threat because the en prise piece can easily retreat. The program recognizes this as a quiescent position. A second special condition recognizes the case when the side on move has a threat which the side not on move cannot answer, but again the side not on move has a capture which is not really a threat but must be dealt with because the threatened gain outweighs the threat of the side on move. It would be a mistake in this case not to save the en prise piece and follow with an attack. How PARADISE recognizes and deals with these situations is described in more detail in the next paragraph.

The first special condition exists when the side not on move has only one threatening plan, and it does not threaten mate and does not threaten to capture a trapped piece. In this case PARADISE does not post a DEFENDMOVE goal; instead it terminates as if no plans were suggested, assuming that the side on move will be able to escape from this one threat. The second special condition exists when the opponent (the side not on move) has one or more threatening plans and the side on move has a threatening plan (which is not as threatening as the opponent's best plan). The program looks for situations where the first move of the opponent's best plan releases a pin which permits an attack, where the first piece to move in the opponent's best plan can be captured, and where the opponent's best plan threatens mate and an opponent's piece which is the only protection of one of the king's flight squares can be captured. In all cases PARADISE checks that the defensive move made to stop the opponent's plan will not prevent the best plan for the side on move from being executed. If these conditions are met, the side on move selects a defensive move (e.g., capturing the protection of a flight square) for an offensive purpose (i.e., to use the best plan of the side on move). The move is treated as an offensive move in the discussion below. This second condition takes priority over the first, i.e., if a defensive move can be played as an offense, then the program will not terminate because of the assumption that it can escape from the opponent's threat.

Before playing the selected move, PARADISE applies the alpha-beta algorithm to possibly terminate the search. If an alpha-beta cutoff occurs, the range returned may not have the same bottom and top values. Unlike the regular search, the quiescence search does not terminate when the threshold has been achieved. For example, the regular search would terminate as soon as it has won the opponent's queen, while the quiescence search is as greedy as alpha-beta will let it be and will keep attacking, trying to win more than the queen. This is justified because the quiescence search is inexpensive compared to regular searching. Any time the program quits with less than it could have won, there is a chance it will have to return to the current node and search again for a better value. Since the quiescence search gets extra information for little cost, it seems worth the gamble to try to win as much as possible. Every time a re-search is prevented it should balance many instances where too much information is gathered. The program avoids searching for long mating sequences by strongly favoring captures when it has selected an offensive move and the side on move is well ahead.

The regular search will sometimes call the quiescence search after it has already searched some moves from the current node. Before playing the selected move, the quiescence search checks if the move has already been searched. If it has, the quiescence search is terminated without changing any values. Once all the above conditions have been checked, the quiescence search plays the selected move and calls itself recursively. The side not on move becomes the side on move and the above analysis is repeated (no distinction is made between the offense and the defense).

The value returned by this search is generally backed up as the value of the current position except in two situations. In both situations the value returned by the search is worse for the side on move than the current evaluation of the position. If the selected move is offensive and the side not on move has no threats whatsoever, then the current evaluation is returned as the value of the current position. (The side on move tried an attack which did not work so the results are rejected.) If the selected move is defensive and the selected move unprotected a square which the side not on move landed on during the resulting search, then the second best defense suggested is also searched and the best result chosen. Examples of quiescence searches by PARADISE appear in chapter 5 as parts of example problem solutions.

To summarize, the quiescence search in PARADISE is an ad hoc combination of heuristics and knowledge which performs with expertise over the domain of 100 positions on which the program was tested (see chapter 6). Any quiescence search makes errors, but PARADISE's errors have tended to be inaccuracies rather than gross misevaluations. This is due to the fact that the knowledge does an excellent job of finding attacks and defenses and analyzing their relative worths. The success of the quiescence search depends on the tactical sharpness of the domain. Only material advantages must be noticed; the program does not notice small positional advantages.

One reason for the success of PARADISE's quiescence search is that it is invoked for semantic reasons, instead of being invoked at every position at a certain depth in the tree (as is done in programs with a depth limit). In PARADISE, quiescence searches are usually done when no plans are being suggested or when the offense has succeeding in attaining the threshold and wants to verify the result. In the first case, the position is usually quiescent and the quiescence search recognizes this. In the second case, a winning combination has already been found for the offense. The quiescence search will find any obvious counter-attacks, and there are usually no obscure counter-attacks which win for the defense in the positions on which PARADISE has been tested.

PARADISE's quiescence search appears to correctly analyze a much wider range of tactical positions than do the quiescence searches of previous programs. This is because PARADISE recognizes a wider range of threats which include forks and sacrifices, while previous programs generally play only captures and checks (and not even all of them at times). PARADISE's ability relies on the accuracy of its knowledge and analysis. The quiescence search tries only one move at most nodes and two moves on occasion. Most other programs generate much larger trees during quiescence searching, trying all captures and many checks.

F) Causality Facility

1) Introduction

Human chess masters discover things during their searching process which they then use for analysis. For example, suppose a master mentally searches a move in a position and discovers that the move loses because of a two move mate that he had not previously noticed. Having discovered this two move mate, he will not mentally search other moves unless he knows they will avoid this mate in two. PARADISE uses its causality facility to do this type of reasoning. The causality facility enables the program to use information discovered during a search in its analysis. The causality facility is based on ideas developed in [Berliner74], and a comparison of the two approaches is given in section H of this chapter.

The causality facility in PARADISE reasons about a move in the current position and a line that was generated by searching a move in the current position. For each line searched, the search returns the tree traversed during the search with a certain amount of information at each node. This information includes the following: the plans searched, the actual moves which each plan caused to be searched, the range of values obtained by searching each plan and each move, the reason for termination of each search, and a small number of patterns that were matched at this node. The causality facility can determine if a given move in the current position either permits or influences (affects) a line described by a tree.

The primary use of the causality facility is to reject a proposed move without searching. In this case, it reasons about a proposed move in the current position and an unsatisfactory line that was generated by searching another move in the current position. If the causality facility determines that the first move tried in the unsatisfactory line did not permit the refutation described in the tree (e.g., by unprotecting a critical square), and that the proposed move could not influence the refutation in the tree in any way, then the proposed move is rejected without searching. The defensive search also uses the causality facility to determine if moves tried have permitted their refutations, in which case a null move analysis is initiated.

PARADISE would not be able to achieve a knowledge-controlled search without using the causality facility to reject moves without searching. Whenever many bad moves in a position lose for the same reason, it is the causality facility which keeps PARADISE from generating the same (possibly large and expensive) tree to refute each move. This section will first give some example positions which show the type of reasoning expected of the causality facility. These examples also show how difficult the problem is. After the examples, the details of PARADISE's causality facility are given. In section H of this chapter, the causality facility is compared to the causality facility in CAPS and to the "method of analogies" in KAISSA.

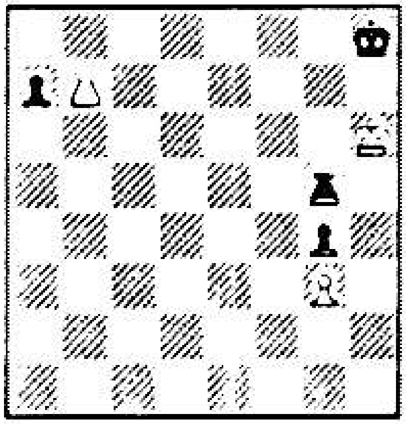


Figure 4.3
black to move

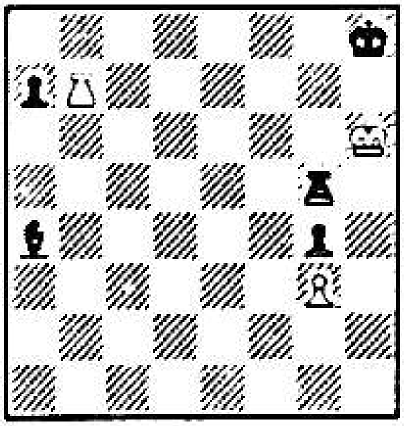


Figure 4.4
black to move

The position in figure 4.3 derives from position 6 in [Reinfeld58] after white plays his rook from b6 to b7. (This saves the rook from capture by the black pawn.) Suppose now that PARADISE does not notice black's threat of a back rank mate, and attempts to save black's rook by playing it to c5. White replies R-b8 which PARADISE recognizes as equivalent to mate. After backing up to the position in figure 4.3, PARADISE has additional suggestions to move the black rook to d5, e5, and f5. It is important that PARADISE recognize the threatened back rank mate in the refutation of R-c5 so that it does not repeat its mistake. If the mate had taken hundreds of nodes of searching to discover, it would be expensive to try these three other rook moves. PARADISE uses its causality facility to do this type of reasoning, and to reject R-d5, R-e5, and R-f5 without searching (after searching R-c5) in figure 4.3. The details are described later in this section.

Figure 4.3 shows the type of reasoning the causality facility is intended to do, but figure 4.4 shows how difficult the problem can be. In figure 4.4, R-c5 fails just as it did in figure 4.3, but now R-e5 should not be rejected, although R-d5 and R-f5 should be. The only influence R-e5 has on the back rank mate is to attack one of the squares over which the "final" check is given. If the system were to say that R-e5 influences the mating line because it attacks a square that an attack is given over, it would search many moves that should be rejected (e.g., R-f5 and R-d5 in this position). On the other hand, a general analysis of which possible interpositions may work is extremely hard. Each side may have a number of pieces bearing on the interposition square and some of them may be pinned. This example shows the subtle ways in which a move may influence an already played line.

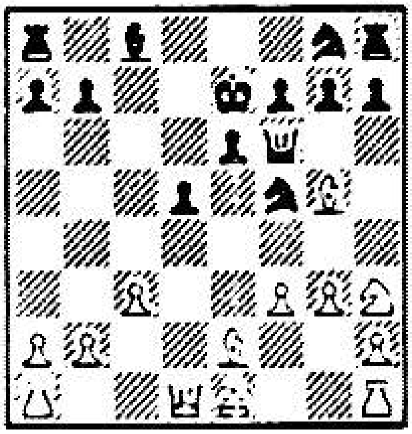


Figure 4.5
black to move

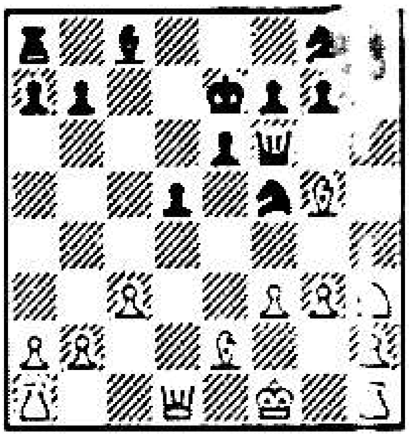


Figure 4.6
black to move

The above two positions show that possible counterattacks by the opponent may subtly affect the influence a move might have on a line. In figure 4.5, there is no way black can save his queen. Suppose the program first plays B-d7 for black (for lack of anything better). It quickly finds that white obtains a distinct advantage after the white bishop captures the queen and black recaptures. It is fairly obvious to human masters that P-b6, N-h6, and other such moves cannot save the black queen. The causality facility should prevent PARADISE from searching such moves since they do not affect the losing line produced for B-d7.

Now consider black's move K-d7. It seems obvious that this cannot affect white's winning of the queen, and so should not be searched. In figure 4.5 this is true, but in figure 4.6, K-d7 should be searched because it allows a black counter attack that may save the position. The only difference in the positions is that the white king is moved over one square in figure 4.6, but now if white answers K-d7 by capturing the black queen, black can reply N-e3 winning white's queen. The fact that K-d7 helps prepare a counterattack influences white's winning line in figure 4.6. Such subtle influences are hard to recognize.

There is a tradeoff between recognizing general influences (i.e., viewing any type of interaction as an influence), and recognizing only more specific influences. By recognizing general influences, it is possible to make almost no mistakes at the cost of searching many moves which the program thinks influence a line (although a human could see that some of them have no influence). By recognizing more specific influences, it is possible to reject more moves without searching at the cost of possibly introducing more errors. PARADISE handles the problem in figure 4.4 by recognizing the influence of an attack on a square over which an opponent's attack is given only when the side on move already has a piece bearing on the square, and the opponent's attack is a check. This is fairly specific and may introduce errors.

PARADISE handles the problem in figure 4.6 more generally. Neither PARADISE, CAPS, or KAISSA would be able to recognize the difference in influence of the move K-d7 between figures 4.5 and 4.6. It appears that the causality facility in CAPS would not see K-d7 as avoiding the loss of the queen, so it would not search that move in either position (thus making an error in figure 4.6). It also appears KAISSA would not notice that K-d7 can improve the situation in figure 4.6. PARADISE thinks K-d7 can influence the loss of the queen since the queen is no longer captured with check. It searches this move in both positions, thus avoiding the error made by the other programs, but at the cost of searching it and similar moves in positions like figure 4.5. In this case PARADISE leans towards being more general and eliminating errors.

These examples show what the causality facility is intended to do and also show how hard the problem is. PARADISE, CAPS, and KAISSA have facilities for doing this type of reasoning, but they all make tradeoffs between effectiveness and correctness when deciding the specificity of the influences to be noticed. These programs are compared in section H of this chapter. PARADISE tries to notice many kinds of influences so that it will not make errors. The causality facility has been effective and errorless over PARADISE's test domain, but there are certainly influences it would miss. (The trees produced by PARADISE are so small that all causality decisions can be checked.)

2) PARADISE's implementation

The causality facility reasons about a move and a tree produced by searching an unsatisfactory line. As described in section C of this chapter, not all lines can be used (e.g., causality cannot be applied to lines terminated by the "better plan at higher level" cutoff). To reject a proposed move without searching, two determinations must be made by causality. First, it must decide that the first move in the unsatisfactory line did not permit the ensuing consequences. This is referred to as a PERMIT determination in PARADISE. Second, the causality facility must decide that the proposed move could have no influence on the unsatisfactory line. This is referred to as an AFFECT determination. The PERMIT determination is also used by PARADISE in the defensive search to instigate a null move analysis when all defensive moves have permitted their refutations.

This part of section F describes PARADISE's algorithm for using the PERMIT and AFFECT determinations to make a causality decision. The particular PERMIT and AFFECT influences checked for are not described until part 4 of this section. Part 3 of this section presents examples of the causality facility in action which should clarify the algorithm described here.

When doing either a PERMIT or AFFECT determination for a given tree and move, PARADISE traverses the tree and checks each node it visits for PERMIT or AFFECT influences. During this traversal, every branch must be followed from a node where the side currently on move is on move. Since the side not on move has achieved a satisfactory result in this tree, every move by the side on move had to be refuted. If the given move permits or affects even one of these, then it may permit or affect the whole tree. At nodes in the tree where the side not currently on move is on move, only the best branch is traversed. The side not on move may have tried many moves which did not refute the original move before finding the refuting move, so only the best branch (which is the refuting move) needs to be traversed. Such a traversal returns a list of influences describing why the move permits or affects the line. This list can be used to compare how two different moves permit the same line.

During the PERMIT determination, PARADISE traverses the tree (using the tree traversal algorithm described above) looking for any of seven permitting influences the given move may have in the tree. (These seven influences are described in part 4.) If any influences are found during the tree traversal, then the move permits the unsatisfactory tree. To reject a proposed move from consideration on the basis of an unsatisfactory tree, the causality facility first does a PERMIT determination using this tree with the first move of the unsatisfactory tree being the move that is checked for permitting influences. If any permitting influences are found, this tree normally cannot be used to reject another proposed move since its first move permitted the refutation. However, PARADISE does not give up this easily. It next checks if the proposed move permits the unsatisfactory tree. If it permits the tree for all the same reasons that the first move in the tree permitted the tree, then the tree can still be used to reject the proposed move since the proposed move would also permit this tree. (The examples in part 3 show how this actually works.) This type of test is not done by the causality facility in [Berliner74].

If the PERMIT determination finds that the unsatisfactory tree did not permit its own refutation, the an AFFECT determination is done to find any influences the proposed move might have on the unsatisfactory line. This involves another tree traversal in which the causality facility looks for possible counterattacks and any of ten affecting influences which are described in part 4. If any influences are found during the tree traversal or counterattack analysis, then the proposed move affects the unsatisfactory tree. If the proposed move does not affect the tree, then it is *rejected* without searching. If it does, then PARADISE checks if the first move in the unsatisfactory tree also affects the tree. If so, the proposed move is *rejected* unless it affects the tree for some reason that the original move in the tree did not. Again, this type of test is not done by the causality facility in [Berliner74].

The following examples should help clarify this description of PARADISE's causality facility.

3) Examples

How PARADISE's causality facility solves the problems presented earlier in this section will now be described. In figure 4.3, the unsatisfactory line consists of a tree with only two nodes: one node for BR-c5 and one for WR-b8. PARADISE first checks if BR-c5 permits this line and finds one reason why it does. While looking at the node for WR-b8, the causality facility notices that BR-c5 unprotects a square over which the white rook delivers a check, and that black has another piece bearing on that square. Thus PARADISE realizes the black rook could have interposed on g8 if it had not moved. This line cannot be used to reject another move unless the proposed move also unprotects g8 (thus permitting this line for all the same reasons). PARADISE next does a PERMIT determination for BR-e5 and finds that this move also unprotects g8, thus permitting the line for all the same reasons R-c5 does. An AFFECT determination is then done for R-e5 and this line, but no reasons are found so R-e5 is rejected without searching since it does not affect the line (and permits it for the same reason R-c5 does). BR-d5 and BR-f5 are rejected in the same manner.

In figure 4.4, the same unsatisfactory line is again generated for R-c5. All PERMIT determinations produce the same results as in figure 4.3. However, the AFFECT determination for BR-e5 now notices one reason why the move affects the line: it attacks e8, a possible interposition square for the check from b8, that is also attacked by another black piece. An AFFECT determination is now done for BR-c5 to see if this move affects the line for all the same reasons. R-c5 is not found to affect the line so BR-e5 is not rejected. PARADISE finds that BR-e5 and BR-f5 do not affect the unsatisfactory line so they are rejected without searching.

The following discussion applies to both figure 4.5 and 4.6 since PARADISE would not treat them differently. An unsatisfactory line has been produced by playing BB-d7. The tree contains a node for white's capture of black's queen followed by some nodes for possible black recaptures on f6. A PERMIT determination is done for BB-d7 and this line. No reasons are found so the line can be used to reject proposed moves (no PERMIT determination need be done on this line for proposed moves). Moves such as BP-b6, BP-b7, etc. are found to have no effect on the line (by the AFFECT determination) and are rejected without searching. BK-d7 is found to affect the line for one reason: the black king has been moved from a square where it was placed en prise at some node in the line. BB-d7 is found not to affect the line so BK-d7 is not rejected. Thus PARADISE will try BK-e7 in both figures 4.5 and 4.6.

4) PERMIT and AFFECT influences

When doing a PERMIT determination on a move and a tree, PARADISE traverses the tree and returns a list describing each reason the given move permits the tree. (The tree traversal algorithm is described in part 2 of this section.) A reason consists of an influence name and the name of a specific piece or square involved in the influence. At each node of the tree traversal, PARADISE looks for any PERMIT influence that the given move has. The seven PERMIT influences checked for at each node are described below (without all the detail actually checked for by the program). The bold face name is the name of the influence in each case and SQ is a variable that is instantiated to a particular square whenever a certain influence is found.

1- (VACATE SQ) The move vacates a square (SQ) which the opponent moves over at this node. (This uses the move stored at this node in the tree and checks for unblocking a line used by an opponent's rook, queen, or bishop.)

2- (VACATE SQ) The move vacates a square (SQ) over which the opponent delivers a check at this node. (This also uses the move stored at this node.)

3- (UNPROTECT SQ) The move unprotects a square (SQ) (i.e., moves so that it no longer bears directly on the square in question) which the opponent lands on at this node. (This also uses the move stored at this node.)

4- (UNPROTECT SQ) The move unprotects a square (SQ) that at this node in the tree is the location of a friendly piece which is en prise. (This accesses en prise patterns that have been stored in the tree.)

5- (SACRIFICE SQ) The piece moved by the move is en prise at this node on the square (SQ) which is the destination square of the move. (This also uses en prise patterns in the tree.)

6- (NO-ESCAPE SQ) The friendly king is in check at this node or is threatened by a discovered attack, and the destination square (SQ) of the given move is a square adjacent to the king. (This uses both en prise and discovered attack patterns that have been stored in the tree.)

7- (UNPROTECT SQ) The move unprotects a square (SQ) over which the opponent delivers a check at this node and another friendly piece currently bears on this square. (This also uses en prise patterns in the tree.)

This set of permitting functions is adequate for the positions PARADISE has been tested on, but certainly more would be needed to handle all middle game positions without errors. By returning a list of reasons, PARADISE can compare how two different moves permit a line. For example, the list of reasons for one move permitting a line might be ((UNPROTECT D5) (VACATE C4) (NO-ESCAPE G8)), and the list for a second move permitting the same line might be ((UNPROTECT D5) (VACATE C4) (NO-ESCAPE G8) (UNPROTECT D7)). PARADISE could then determine that the second move permits the line for all the same reasons the first move does (as well as one more reason).

An AFFECT determination traverses the tree in the same manner as a PERMIT determination, and again returns a list of reasons. During an AFFECT determination PARADISE checks eight influences the given move might have at each node visited during the traversal. In addition, it checks two influences in the original position only, in an attempt to recognize counterattacks. These ten influences, the first seven of which correspond to the PERMIT influences, are described below (without all the detail actually checked for by the program). The ninth and tenth influences listed below are the two checked only in the original position. (P and SQ are variables that are instantiated to a particular piece or square whenever a certain influence is found.)

1- (OCCUPY SQ) The move occupies a square (SQ) which the opponent moves over at this node.

2- (OCCUPY SQ) The move occupies a square (SQ) over which the opponent delivers a check at this node.

3- (PROTECT SQ) The move protects a square (SQ) (i.e., moves so that it now bears directly on the square in question) which the opponent lands on at this node.

4- (PROTECT SQ) The move protects a square (SQ) that at this node in the tree is the location of a friendly piece which is en prise.

5- (MOVES P) The piece (P) moved by the move is en prise at this node on the square (SQ) which is the origin square of the given move.

6- (ESCAPE SQ) The friendly king is in check at this node or is threatened by a discovered attack, and the origin square (SQ) of the given move is a square adjacent to the king.

7- (PROTECT SQ) The move protects a square (SQ) over which the opponent delivers a check at this node and another friendly piece currently bears on this square.

8- (CAPTURES P) The move captures a piece (P) which is moved at this node.

9- (ATTACKS P) The move threatens an opponent's piece (P) in the current position which is not en prise already and which is not the first piece to move in the unsatisfactory tree. The move is recognized as affecting the unsatisfactory tree because of the counterattacking possibilities. This is checked only in the current position (without traversing the tree).

10- (DSCATTACKS P) The move sets up a discovered attack in which the attacker, when uncovered, can safely capture the attacked piece (P). Like ATTACKS, this influence is checked only in the original position.

Like the PERMIT influences, this set of AFFECT influences is adequate for the positions PARADISE has been tested on, but not adequate to handle all middle game positions without errors. For example, the effects of moves which take two moves to manifest their defensive possibilities are not noticed. To be more specific, suppose the only way to thwart an opponent's attack is to move our knight from the first rank to the third rank, and then after the opponent further develops his attack, to move the knight from the third rank to the fifth rank from where the knight can protect some square critical to the opponent's attack. The causality facility would not notice how the original knight move would affect the attack since the knight attacks no critical square from his post on the third rank. It is not clear at present how to formulate a causality facility which would never make errors and would still be effective for rejecting moves without searching. It is not even clear whether it is possible to do so. ([Berliner77] discusses these issues.)

G) PARADISE's search strategy

1) Overview

PARADISE's search strategy has already been briefly described. By determining ranges for the values of plans, the system attempts to discover only the information needed to show that one move is best in the original position. Section C of this chapter describes how ranges are determined and how they are compared (e.g., in the alpha-beta algorithm). Searches are initiated repeatedly from the original position until one move is shown to be the best. During each search, PARADISE attempts to narrow the range of values only by the amount specified by the threshold. After each such search, the program again analyzes the situation at the top level to determine the best strategy for proving that one move is best. Using the selected strategy, a plan is chosen for searching and the threshold is reset. A given plan at the top level may be searched many times; eventually its range of values will converge to one number.

During each search, the program attempts to search the most promising plan for the offense. Lines of search are abandoned when another alternative looks more promising than the current one. How PARADISE keeps track of alternatives and abandons lines is described in section C of this chapter, and the details involved in comparing one alternative to another are described in part 2) of this section.

Whenever the program abandons a line without finding its "true" value (e.g., when it stops searching because there is a better alternative) it may be necessary to search the line again (e.g., if the better alternative does not work out). One way PARADISE controls the search is by abandoning lines whenever its threshold (described below) is achieved. Initially the threshold is set so that the offense must win at least rook for knight in the current position and also be at least rook for knight ahead overall (i.e., if the defense is ahead, the offense may need to win more than rook for knight). The search returns when the threshold is achieved rather than investing more effort to improve the result. If the results returned do not provide enough information to show that one move is best, lines may be searched again with a higher threshold value. Part 3) of this section describes how a search is restarted once it has been abandoned. Part 4) of this section describes how PARADISE decides which abandoned search to restart.

PARADISE uses the initial threshold for one other purpose. Once the initial threshold is obtained, the search will consider only plans which have a LIKELY of 0. This is implemented in the "plan not likely to succeed" cutoff in the SEARCH routine and is important for controlling tree growth. This is done because it is not reasonable to invest large amounts of effort trying ideas which don't seem likely to succeed when a winning line has already been found. PARADISE calculates LIKELYs accurately and LIKELY 1 plans rarely succeed, especially when a LIKELY 0 plan has cracked the defensive position already. This cutoff could produce errors, though not serious ones since the error would be the selection of a winning move which was not the "best" winning move. No such errors have been made in any position PARADISE has analyzed.

The best-first search with ranges and top-level proof strategies is an important part of PARADISE's ability to achieve a knowledge-controlled search.

2) Determining the most promising node

Decisions to abandon a line for a more promising line are made by the "better plan at higher level" cutoff in the SEARCH routine (described earlier), and by the offensive search just before it decides to use a static analysis as a source of plans for the SEARCH routine (described below). As the search goes down a branch, it carries along a list of untried alternatives. This list specifies the expectation of the best untried alternative for each previous ply in which the offense is on move. The offensive search will abandon the current line before doing a static analysis in two cases. The reasoning behind the first case is that when we are backing up to try a better plan, the backing-up process should not be interfered with by trying a static analysis. In the second case, the offense has an alternative that is better than the top of what is now the ultimate range of contention, so something the defense has already achieved will keep the current node from ever reaching the expectation of the alternative. More formally, these two cases are defined as follows: 1) a search of a plan produced by the plan currently being executed (i.e., not a plan produced as an obviously winning move) was terminated because there was a better plan at a higher level, and this higher level is above the current level, and 2) the upper value the defense achieved is worse for the offense than the expectation of one of the untried alternatives. In both cases, the termination inserts an ANALYZE pointer in the tree so that the static analysis may be carried out when the line is searched again.

To see if the current node is the best one to search, PARADISE makes comparisons to the alternative expectations that have been passed down the tree. In the second case above, the upper value achieved by the defense is compared and in the "better plan at higher level" cutoff, the current expectation is compared. Neither comparison is based strictly on the value being compared and the alternative expectations since PARADISE is reluctant to abandon the current line. This reluctance has three sources, listed here from least important to most important: 1) the bookkeeping costs of re-searching a line, 2) the fact that abandoning a line provides no results for establishing ranges of contention and other useful information, and 3) the fact that PARADISE is unlikely to make a good choice when it chooses a node to search again.

This reluctance is exhibited by adding a reluctance value to the value being compared each time it is compared to one of the alternative expectations. The reluctance value varies with each alternative since it is dependent on the difference in depth in the tree of the alternative and the current node. The current line is abandoned if an alternative expectation is greater than the value being compared plus the reluctance value. If the current node is much deeper than an alternative the reluctance value may become negative indicating that PARADISE is becoming reluctant to keep searching.

If the reluctance is small, the system tends to be indecisive, frequently abandoning a line only to return and search it later. If the reluctance is large, the system tends to be bull-headed, refusing to abandon a line despite better alternatives. The latter error can be more serious since it results in a possibly exponential tree search while too small a reluctance has only the drawbacks enumerated above. Little experimentation has been done on the effects of particular reluctance functions, but small changes in the reluctance function appear to have large effects on system behavior. The reluctance function used in PARADISE was developed by trial and error and works well in the given domain. This function is more reluctant to abandon the current line when the current plan threatens mate than when it threatens something less. This is because mating attacks usually grow small trees (so sticking with them isn't as costly) and a mate is a very definite result which is useful for limiting the growth of the tree elsewhere. (Again this relies on the system's accurate analysis of mate threats.) PARADISE's reluctance function is shown in figure 4.7. A reluctance of 20 means that the current plan will not be abandoned unless the alternative expectation is more than 2 pawns better than the current expectation.

Ply difference	Reluctance for small threat	Reluctance for mate threat
2	20	20
4	10	10
6	-2	10
8	-10	0
10	-36	0
12,14	-100	0
16,18	-100	-12
20,22,24	-100	-36
26 and up	-100	-100

Values pawn 10, knight 32, queen 90

Figure 4.7
PARADISE's reluctance function

3) How a search is restarted

Searching a line again is not overly expensive because the information calculated about each position is stored on the disk for later retrieval. In addition, PARADISE's trees are so small that all trees generated are kept in core. Such a tree contains a moderate amount of information about each node including the following: the range of values obtained by searching each plan and each move, the name of the disk file which has more information about this node, and any plans which have not yet been investigated (i.e., RE-SEARCH and ANALYZE pointers).

To search a line again, PARADISE traverses its in-core tree until it arrives at the node of its choice (described in part 4 of this section). Once at the chosen node, PARADISE retrieves the information about this node from the disk and starts executing one of the plans specified in a RE-SEARCH pointer (or does a static analysis for an ANALYZE pointer). None of the positions traversed over need to be restored from the disk. When a position is traversed over in which the defense is on move, a process is set up which will invoke the normal defensive search should the search back up to that node with an improved value for the offense. In other cases, new discoveries can be noted in the in-core tree and these intermediate positions need never be restored from the disk. There is some overhead involved in recalculating the values of various measures as the tree is being traversed, but the cost of restarting a suspended search is less than the cost of processing one node.

4) Deciding where to search again

For two reasons, the strategy for choosing a node to re-search at the top level has not been well developed. One reason is that PARADISE has usually solved its test positions without needing to search lines again. The other reason is lack of sufficient resources. Problems which require lines to be searched again are usually long and require large amounts of cpu time. Thus it is very expensive to test a small change on even one position. To analyze a strategy, it is necessary to test it on a wide range of positions and this is prohibitively expensive on the heavily loaded computing facilities used to develop PARADISE. As harder problems are undertaken, better strategies may need to be developed. The current algorithm and the issues involved are described below. Examples of PARADISE using the strategies described below appear in the next chapter.

Ideally, the program would like to search from the node which will most quickly lead to a proof that one move is best at the top level. A classical best-first search will search from the node which has the most promise for achieving a good result. However, this is not always the same as the node which will most quickly lead to a proof that one move is best. This can be illustrated in two different situations. Suppose the system has narrowed the choice to two moves: the first one has already achieved something while the second has not as yet been shown to achieve anything. Promising unsearched plans remain in the trees that have been produced for both top level moves. In some cases the best move may be found most quickly by searching the first move and showing that its success outweighs anything the second plan might hope to achieve. In other cases, the best strategy may be to search the second plan and show that none of its unsearched plans work, thus showing the first move is best because of the success it has already achieved.

Another situation where it may not be best to search the most promising node occurs in a tree where the defense has many good moves at one node. Since the offense must refute every defensive move at this node, it may be best to first search the defensive move for which the offensive reply is weakest. In this way the program may quickly discover that this line will not work.

There is no theory of heuristic search which describes useful strategies or the issues involved in picking strategies. PARADISE has developed a few strategies and defined a few criteria for choosing strategies and these are described below. They are far from adequate as a theory of heuristic search, but they represent a first step towards understanding the problem and they work for the problems on which PARADISE has been tested.

After some searching has been done, PARADISE has a tree that has one branch at the top level for each plan that has been searched. Each of these top level branches has a range representing its value; the range may be narrowed down to a single value. No range has a bottom value which is better than the top value of all other ranges, so it is not clear which plan is the best. The program must pick some point to initiate another search in an attempt to further narrow ranges and find the best plan. Scattered throughout each of these top-level subtrees are RE-SEARCH pointers (which specify a particular plan to be searched), and ANALYZE pointers (which specify that a static analysis should be undertaken to produce plans for searching). Searching plans from these pointers may improve the value of the subtree. Each of these pointers contains information about the threat of its unsearched plan and about the likelihood of its success. In a classical best-first search, these threats would be used to determine the most promising plan (perhaps taking into account that the defense can choose its best line at nodes where it is on move) and this plan would be searched next. As explained earlier, this is not always the best strategy. The strategy used by PARADISE to pick a RE-SEARCH or ANALYZE pointer for searching is described below.

PARADISE considers five criteria in determining which pointer to search next. These criteria are briefly summarized below, with higher priority criteria first in the list. How PARADISE actually uses these criteria in the decision process is described after the summary. The entire process described below is repeated each time the search returns to the top level, so the program is continually reevaluating its strategy.

1- **Overall view.** The program likes to have an overall view of the possible solutions. It will not extensively investigate one top-level plan until it knows something about all top-level plans.

2- **Promise of success.** This is the most important criterion in classical best-first searches. PARADISE prefers RE-SEARCH pointers which are likely to succeed.

3- **Relationship of previous searches.** PARADISE's decision is influenced by the relative values already discovered. The program looks for such cases as all plans (at the top level) failing, many plans succeeding, and only one plan succeeding while the rest fail.

4- **Previous success.** If a particular plan has already met with some success, PARADISE is much more willing to invest more effort in it than in a plan which has not yet met with any success.

6- Depth of RE-SEARCH pointer. To avoid constructing long elaborations when a short solution is present, the program prefers RE-SEARCH pointers which are close to the top level. Long elaborations can involve contorted attempts to make a poor plan work and should be avoided if possible.

How PARADISE uses these five criteria to choose a place to begin searching is described below. The program gradually narrows the number of plans and RE-SEARCH pointers it is considering as possible candidates for search initiation. The steps described below are done, in the order specified, to eventually narrow the possibilities under consideration to one.

To obtain an overall view of the situation, PARADISE will not search a plan a second time until all plans at the top level have been searched at least once. If any plan has achieved the original threshold then any plan with a LIKELY of 1 is forever removed from consideration for searching and any RE-SEARCH pointer whose plan has a LIKELY of 1 is removed from the tree. (The justification for this is given earlier in this section.) Once all plans have been searched once, the program next looks at individual RE-SEARCH and ANALYZE pointers within the trees that the search has produced. Only the most promising RE-SEARCH pointers are kept under consideration. When the program traverses a tree to find these pointers, it only follows what is currently the best branch at any node where the defense is on move. As described in the introduction to this section, there are situations where the best node to search may depend on the various options the opponent has. The method used here ignores this, is selected for simplicity, and could be improved with more research.

PARADISE considers each RE-SEARCH pointer to have one of the following three levels of promise: 1) a LIKELY 0 plan threatening the threshold, 2) a LIKELY 0 plan which does not threaten the threshold, and 3) a LIKELY 1 plan. ANALYZE pointers are considered to be at level 1, the most promising level. Any pointers at the same level are considered to be equally promising. For example, when the threshold is still at its original value, a LIKELY

0 plan threatening to win a knight is considered just as promising as a LIKELY 0 plan threatening to mate. PARADISE eliminates from consideration as the next search initiation point any pointers at a less promising level than the most promising pointer under consideration. LIKELY 0 plans that do not threaten the threshold (level 2) are ignored whenever the original threshold has been achieved, but are helpful in positions where the program can only win a pawn.

It should be noted that PARADISE has much more information available than what it actually uses. It knows the THREAT, LIKELY, LOSS and SAVE of each plan in a RE-SEARCH pointer as well as every reason for its suggestion. An accurate estimate of promise of success could be obtained from these values, and such an estimate is the principal determinant in choosing a search point in a classical best-first search. PARADISE chooses to ignore this information for two reasons: 1) simplicity, and 2) the three other criteria yet to be described are considered more important for finding a search point which will quickly prove that one move is best at the top level. No claim is made that this is right or best; more research needs to be done in this area. What the work on PARADISE does indicate is that many criteria are useful in finding a search point. In addition to the promise of a node, things like previous success, the depth of a node in the tree, and the relationship of previous searches should be considered.

PARADISE next looks at the relationship of previous searches (whether or not they are still under consideration as search initiation points) to decide on a strategy. One of the following three relationships holds: 1) one plan has achieved the original threshold but none of the others have, 2) more than one plan has achieved the original threshold, and 3) no plans have achieved the threshold. PARADISE's strategy for each of these 3 cases will be described in the next few paragraphs. Any plan whose range of values is narrower than two-thirds the value of a pawn is considered as having its value completely determined. Any plan whose value is completely determined is removed from consideration as a possible search initiation point. Special circumstances arise in cases 1 and 2 when all plans which have achieved the threshold are no longer under consideration as search initiation points. In these special circumstances, the program uses the strategy for case 3 on all plans still under consideration.

PARADISE's strategy for the above three cases is presented here. The effect of a "strategy" here is to eliminate some plans from consideration as candidates for search. The possibilities still under consideration after application of these "strategies" will then be further reduced (eventually to one RE-SEARCH or ANALYZE pointer) when the program considers the fourth and fifth criteria involved in choosing a search point. (If only one plan remains under consideration at this point, the program skips down to the fifth criteria.)

In case 1 only one plan has achieved the threshold. In this situation PARADISE eliminates the one plan which has succeeded from consideration as a possible search point. The idea is to narrow the ranges of all other plans and show that their tops are worse than the bottom of the range of the plan that has succeeded. This is called the DISPROVE-REST strategy (as in [Berliner79]) since the system is trying to prove that the rest of the moves don't work. The plan which has succeeded has achieved the threshold and therefore has achieved a significant material gain. Because of the way PARADISE measures ranges and uses the threshold, the DISPROVE-REST strategy is usually best in this case. If a smaller, less significant advantage had been achieved, this would not be the case.

Most plans will not achieve a significant material gain so it should be reasonable to show that they do not work. This is so because the threshold has been achieved, so only plans with a LIKELY of 0 need to be refuted. The alternative is to show that the plan which has already succeeded will be able to win so convincingly that even the best untried threats of other plans could not do better. Since untried plans frequently contain mating threats, this would frequently involve showing that the successful plan mates. This may not be a reasonable task. Once a search has been initiated and has returned, the situation is reevaluated. Thus if one of the plans PARADISE is trying to disprove succeeds, this will be quickly recognized as case 2 and a different strategy will be employed. Note that if only a small advantage has been achieved (the threshold has not been achieved), then the best strategy may be to show that the best plan can achieve a greater success. This falls under case 3 and is discussed later.

In case 2, more than one plan has achieved the threshold. In this situation, PARADISE removes all plans which have not achieved the threshold from consideration as possible search points, and resets the threshold to be the best bottom limit that has been achieved plus the value of one pawn. This is similar to the PROVE-BEST strategy in [Berliner79]. Since the program has at least two plans which have succeeded, it must eventually show that one is better than the others. It does this before showing that other plans do not work since proving that one successful plan is more successful than other successful plans may produce such a convincing success (e.g., a mate) that other plans will never have to be disproved.

In case 3, the threshold has not been achieved by any plan or has been achieved by a plan no longer under consideration as a search initiation point. Usually, PARADISE makes no change to plans under consideration in this case. In one special case it does. This occurs when there are exactly two plans under consideration, the threshold has not been achieved, and one of the plans has achieved some positive gooc while the other has not. In this situation, PARADISE again employs a DISPROVE-REST strategy by removing the plan that has achieved some good from consideration as a possible search point. Although DISPROVE-REST is usually considered too dangerous to use when the threshold has not been achieved (because of a possible search explosion since plans with LIKELYs of both 0 and 1 must be refuted), it is tried in the special case when only one plan has to be disproved. This special circumstance also requires that another plan has achieved some positive good so that a best-move proof will be accomplished if the other plan is disproved. Chapter 5 contains an example where PARADISE uses the DISPROVE-REST strategy in this situation to obtain a best-move proof.

Once the selection of a strategy has limited the number of plans under consideration, PARADISE considers the criterion of previous success to further narrow the possibilities. Only plans which have achieved the best result so far are kept under consideration. More specifically, of the plans currently under consideration, the one whose range has the best bottom limit is kept under consideration. In addition, any plan under consideration whose bottom value is within half a pawn of this "best" plan is also kept under consideration. All other plans are removed from consideration. This restriction severely limits the number of plans under consideration: in most cases only one plan remains under consideration, occasionally there are two, and only rarely are there three or more. In this way PARADISE places great importance on previous success. The program would rather search a plan which has won two pawns and threatens to win a knight than one which has not won anything but threatens to mate. It is not clear if this is a good strategy. It has worked well in PARADISE's domain since initial searches frequently find some advantage when a winning combination exists.

If more than one RE-SEARCH or ANALYZE pointer still remains under consideration, PARADISE chooses the shallowest one (i.e., the one closest to the top level) as the point to initiate a new search. If there is still a tie, PARADISE just picks the plan and pointer which happen to be first in its tree. This criterion of shallowness plays an important part in the system since PARADISE ignores much of the information about the promise of a RE-SEARCH pointer in order to use the shallowness criterion instead. Before initiating a search, PARADISE resets the threshold to be the best bottom limit that any plan has achieved (whether under consideration for searching next or not) plus the value of one pawn. If this value is less than the original threshold, then the original threshold is used.

To summarize, PARADISE uses five criteria to choose the point to initiate the next search. PARADISE has a few different strategies and places great importance on the ideas of previous success and shallowness. Information on the promise of an untried plan is largely ignored (except for the likelihood of success), and no analysis of the opponent's alternatives is done. These are only two areas in which improvements could be made. More research needs to be done to determine good strategies and the criteria involved in choosing them, since harder problems will probably require more sophisticated decisions about where to initiate searches. PARADISE's search is compared to other searches in the next section.

H) Comparison to other systems

1) Introduction

PARADISE's overall search paradigm is to represent values by ranges and then narrow the ranges only enough to show that one move is best. This is done by selecting different strategies at the top level and repeatedly doing best-first searches to narrow a particular range. No previous chess program or problem-solving program has used such a paradigm. Berliner's B* algorithm ([Berliner79]), a domain independent algorithm, was the first to develop these ideas, but the B* algorithm has never been applied to an intellectually challenging problem. This section briefly mentions parts of PARADISE's search which are new and compares its best-first search to best-first searches in other chess programs. The next section compares the causality facility to similar mechanisms in other programs, and the final section compares PARADISE and B* in some detail.

PARADISE is the first chess program to use ranges as values and to develop methods for handling ranges in the alpha-beta algorithm and other parts of the search. PARADISE makes effective use of some tree pruning techniques that have been tried before with less success, e.g., the "forward prune" cutoff and the "same plan at higher level" cutoff. PARADISE will frequently use its forward prune to eliminate all moves except the one or two best on the basis of its static evaluation. Many programs make effective use of similar forward pruning techniques, but never to eliminate that many moves. The "same plan at higher level" cutoff refuses to search a plan that could have been tried earlier unless it looks more promising now. This keeps the program from trying its moves in every possible order down a line. Other programs have had little success in attacking this problem.

PARADISE has an efficient means of restarting a search after it has been suspended and backtracking has occurred. No other program which does large amounts of analysis and data storage at each node does this. PARADISE's quiescence analysis covers a much wider range of threats than the quiescence searches of other chess programs. PARADISE uses a null move analysis to occasionally get on the right line without wasting effort on spurious lines. Other programs (e.g., KAISSA) occasionally let one side make two moves in a row, but this technique has never before improved performance and reduced the size of the search as it has in PARADISE.

Best-first searches have been used in previous chess programs (e.g., COKO which is described in [Kozdrowicki73]). These programs do not use ranges or top-level strategy selection as PARADISE does. By using the results of its static analysis and its reluctance function, PARADISE makes much better decisions about which node is more promising than previous programs have. PARADISE also differs from most best-first searches in that it only looks for more promising alternatives when the offense is changing plans. As long as PARADISE is executing a plan that is working, it does not look for better alternatives.

2) Comparison of causality facilities

Two programs have facilities similar to PARADISE's causality facility. KAISSA's method of analogies ([Adelson-Velskiy75]) and CAPS's causality facility ([Berliner74]) are both based on the same idea. The idea is to use information about moves, lines, attacks, squares, pieces, etc. to determine the influence a move might have on a previously analyzed line. If a move will have no influence on an unsatisfactory line, then it can be rejected without searching. Human chess players appear to use causality type reasoning: if a move fails then it is sometimes obvious that other moves will fail for the same reason.

The three programs have different implementations of this idea. The causality facility in PARADISE reasons about a proposed move in the current position and an unsatisfactory line that was generated by searching another move in the current position. If the causality facility determines that the first move tried in the unsatisfactory line did not permit the line (e.g., by unprotecting a critical square), and that the proposed move could not influence the line in any way, then the proposed move is rejected without searching. The causality facility in CAPS reasons about an unsatisfactory line. Like PARADISE, it determines if the first move of the line permitted the line. If not, CAPS tries only moves generated by the causality facility as counter-causal moves. This is similar to PARADISE's counter causal analysis and produces only moves which influence the unsatisfactory line.

Thus the effect is the same as in PARADISE where moves generated by any mechanism (counter causal analysis or otherwise) are checked for their counter-causal nature by the causality facility. PARADISE has a major advantage over CAPS in that it can compare the influence two different moves have on the same line since it returns a list of reasons describing an influence.

The method of analogies is considerably different than these two causality facilities. It reasons about two positions, the current one and one that has already been searched and found to be unsatisfactory no matter which move is played. For the method of analogies to apply, all admissible (see [Adelson-Velskiy75] or interpret loosely as "legal") moves in the current position must also be admissible in the unsatisfactory position. The method of analogies then determines if any of the differences in the current position can affect any of the unsatisfactory lines. (Information about the lines is retained as a number of sets as described below.) If not, then the new position can be assumed unsatisfactory without searching any moves in it.

CAPS and PARADISE have similar approaches to causality while KAISSA's is somewhat different. KAISSA's approach cannot reason about individual moves, but can only determine that a whole position is unsatisfactory. The method of analogies is only applicable when all lines from one position failed and when there is another position whose admissible moves are all present in the unsatisfactory position. It seems that such a situation would not occur frequently, and KAISSA's authors do not assess, even subjectively, how the method of analogies has performed in their program. Reasoning about individual moves is useful, for example, in positions where there is only one good move and many bad ones that lose to the same complicated attack. Once CAPS and PARADISE have tried one bad move, their causality facilities should eliminate all the other bad moves without searching them. KAISSA would have to try all moves, and the method of analogies would only help if it could be applied to positions that are produced after playing a bad move and which have all their admissible moves present in a position that occurred in the search of a previously tried bad move.

Despite the different approach used by the method of analogies, all three implementations will here be referred to as causality facilities (for ease of reference). The major difference between these three causality facilities is the wealth of information returned from a previously analyzed line in PARADISE. In both KAISSA and CAPS, all that is returned from an analyzed line is a number of sets. These sets contain squares that were moved to, squares that were moved over, pieces that moved, squares that were newly attacked by moves that were made, pieces that were attacked, and so forth. These sets do not contain timing information (e.g., what order moves occurred in, whether new attacks existed simultaneously), and do not show which move caused which effects (e.g., for a square in the set of "squares moved to", it is not possible to determine which piece moved to that square). PARADISE does return this kind of information. After searching a line, PARADISE returns a tree which contains each node generated during the search. Each node describes which piece moved, its origin and destination squares, and a number of attacking patterns (including discovered attack patterns) which matched at that node. With such a tree, PARADISE can determine which attacks were happening simultaneously, which moves were first, which pieces moved to which squares, and which moves caused which attacks. Returning this much information would probably not be practical in KAISSA since it produces much larger trees than PARADISE.

As described in section F of this chapter, it is very difficult to notice all the subtle ways a move might influence a line. Any of these three causality facilities can make a mistake by not recognizing a subtle influence, which may cause the program to reject a good move without searching it. ([Adelson-Velskiy75] references a "proof" that the method of analogies never makes a mistake. However, the definition given in the proof for "influence" is not adequate and does not appear to recognize such subtle influences as those described in section F. For example, all moves in figure 4.5 fail and it appears that the method of analogies would be applicable to figure 4.6, yet would not detect the influence.) There is a tradeoff involved. By recognizing general influences (i.e., viewing any type of interaction as an influence), it is possible to make almost no mistakes at the cost of searching many moves which the program thinks influence a line (although a human could see that some of them have no influence). By recognizing more specific influences, it is possible to reject more moves without searching at the cost of possibly introducing more errors.

It is difficult to compare performance of the three implementations. There are a small number of examples of CAPS's causality facility in [Berliner74], and PARADISE appears to make more causality cutoffs over this small set than CAPS does (see section A of chapter 5). PARADISE has made some major improvements over the causality facility in CAPS. PARADISE returns a list of reasons describing an influence which makes it possible to compare two different influences. By doing this, PARADISE can reject moves that would otherwise have to be searched. PARADISE separates the AFFECT and PERMIT determinations and can use them for other purposes than rejecting moves. For example, the defensive search uses PERMIT determinations to determine when to initiate null move analyses.

No example or subjective evaluation of the method of analogies in KAISSA has been published. The algorithms of the three programs are not similar enough to check for subsumption since each program uses different tradeoffs in generality when checking for a particular influence. One thing is clear: PARADISE has enough information available to both make fewer mistakes and make more cutoffs than either CAPS or KAISSA. It would be a difficult project at present to show that either of these has actually been accomplished.

3) Comparison to B*

Like PARADISE, B* narrows ranges with best-first searches until it shows that one move is best at the top level. It has both a DISPROVE-REST and PROVE-BEST strategy which are similar to strategies in PARADISE. While sharing these concepts, the two searching algorithms have different implementations. (For example, as described below, a "range" represents something different in PARADISE than it does in B*.) B* refers to the top of a range as the optimistic value and the bottom of a range as the pessimistic value.

Perhaps the largest difference between PARADISE and B* is that B* has never been applied to an actual problem. Before B* could be useful in a game like chess, it would be necessary to define methods for determining optimistic and pessimistic values, methods for choosing between search strategies, methods for comparing values so that algorithms like alpha-beta could be used, methods for efficiently restarting a search, and other things. B* assumes reasonable solutions for the above problems exist in its domain. However, given a domain, it is not obvious that such reasonable solutions exist. For example, it is not clear that it is possible to define optimistic and pessimistic values for chess positions in such a way that B* would behave nicely in a chess program. PARADISE actually defines all these details to produce a working chess program. In doing this, PARADISE has developed many concepts similar to concepts in B*, but with important differences which make the program work effectively in its domain. These differences are described below.

Ranges in PARADISE are defined (by their use) differently than they are in B*. PARADISE partially avoids the difficult problem of putting optimistic and pessimistic values on chess positions. In PARADISE, the bottom of a range is a value which has already been proven and the top is the best the offense may be able to achieve with more effort. There is no concept of "optimistic value" when the defense is on move, since the defensive search always searches until a satisfactory result is found or an unsatisfactory one proven. The concept of "pessimistic value" would be inexact in PARADISE; the bottoms of ranges are more accurately referred to as "the best value achieved so far". Most algorithms in PARADISE treat the bottom of a range as the true value of a position, but they are aware that the value may (or may not) be improved with more searching effort. PARADISE relies on the distinction between offense and defense in its use of ranges; correct interpretation of a range depends on whether the offense or defense is on move. B* does not have an offense and defense in its hypothetical domain, and its ranges are interpreted the same at every node.

One major difference between B* and PARADISE is the method used for backing up. B* backs up whenever it achieves a value better (or worse) than any alternative. It backs up to the point where minimax no longer sees the new value as the main variation. PARADISE, on the other hand, uses its threshold for backing up at offensive nodes. A changed value will be backed up as soon as it has achieved the threshold, and not until then. The defensive search does not back up until a satisfactory result is found or an unsatisfactory one proven.

B* has both a DISPROVE-REST and PROVE-BEST strategy which are similar to strategies in PARADISE. As described in section G of this chapter, PARADISE actually recognizes many situations and special cases before choosing a search point. Many slightly different strategies could be differentiated in PARADISE as opposed to these two in B*.

Once a search has been initiated, both B* and PARADISE use a best-first search. B* does this by comparing the optimistic value of the current node to the optimistic value of the best alternative (a value which is carried down by the search). PARADISE does not define a "best" alternative. Instead, the search carries down a whole list of alternatives. Besides an optimistic value, each alternative describes its likelihood of success and its depth in the tree. This information allows for a richer comparison in deciding which node is best (e.g., the reluctance function varies with depth in the tree). One other difference is that PARADISE does not consider other alternatives when it is executing a plan that is working. The "best-first" aspect only comes into play when a new plan is being considered.

I) Summary and Conclusions

Any knowledge-controlled search can easily become uncontrolled when the program misperceives the situation or is unaware of a better alternative. PARADISE is the first program to achieve a knowledge-controlled search in the middle game of chess. The search is controlled by using a best-first search which helps the program view each node in a global context. The program is always aware of other possibilities, and misperceptions are usually discovered before much effort is expended. PARADISE can be overwhelmed by positions where there are many possibilities, but in a test of 100 problems this happened only once (see chapter 6). Inserting a poorly written production in the knowledge base which often suggests poor plans can also cause PARADISE's search to become uncontrolled.

The success of PARADISE's search algorithm is due in part to the tactical sharpness of the domain. Compared to "positional" chess positions, tactically sharp positions make it easier to describe what has been accomplished by a particular line and to decide which alternatives are more promising. It is not clear how much PARADISE's search depends on this since it has not been tried in other domains. Berliner experimented with a best-first search and reported in [Berliner74] that it did "very well" in tactically sharp positions but that it greatly multiplied the effort expended on ordinary positions. However, PARADISE should avoid some of this multiplied effort because it retains enough information to restart an abandoned search for less than the cost of processing one node. Further investigation is needed to determine the usefulness of this search strategy in other domains.

The following list summarizes the major features of PARADISE's search:

-The search employs different strategies to show, with a minimum of effort, that one move is best at the top level.

-A best-first search strategy is used which gives the program a global view of the tree and helps avoid lengthy analyses of poor lines.

-Using information obtained from static analysis, plans, and the reluctance function, the program makes reasonable decisions about which node in the tree is the most promising to search.

-The search can efficiently search lines a second time because all pertinent information is kept in core in a tree which points to disk files containing more specific information about each position.

-The program specifies values of lines and positions by ranges. This allows the system to gain some amount of information about a line without investing all the effort required to determine its exact value. Algorithms which compare two values (e.g., alpha-beta) handle the possibilities which arise when comparing two ranges.

-PARADISE uses a threshold to define its current level of aspiration. The search is temporarily satisfied whenever the threshold is achieved and changes the threshold in the process of using a particular strategy to prove that one move is best.

-Each search backs up with the whole tree that has been generated. This tree includes (among other things) each move made, reasons for termination, patterns which describe situations, and unexplored possibilities. This information communicates discoveries that have been made during the search.

-The causality facility uses the tree returned by the search to reject proposed plans which cannot affect the problems discovered in a previous search.

-The counter causal analysis uses the tree returned by the search to propose new plans which may solve problems encountered in a previous search.

-A static analysis in PARADISE generates much information in the attribute lists of suggested plans. This information helps the search terminate branches and recognize when a plan is not working.

-By using the information in attribute lists, PARADISE avoids searching every possible ordering of available threats. During the search, a plan which could have been applied earlier is not applied in the current situation unless it now looks more promising than it did earlier.

-The quiescence search accurately handles a much broader range of positions than the quiescence search of any previous program. By employing specific pattern-based knowledge, it considers many different types of attacks which make a position non-quiescent.

-The defense will use a null move analysis to better understand a position on occasion. This involves letting the offense make two moves in a row and then analyzing the result to better understand the situation.

CHAPTER V

PARADISE in Action

A) A typical medium-sized search

This chapter presents PARADISE's solutions to three problems in order to show the interaction of the various elements of the search and the kind of trees produced by the program. Section A presents the actual protocol produced by PARADISE during a typical medium sized search in which the program finds the best line immediately. Section B outlines one of PARADISE's largest searches in which the DISPROVE-REST strategy is used to find the best move indirectly. Section C presents another actual protocol where PARADISE does not find the winning line immediately and must repeatedly initiate searches from the top level until the winning line is found.

These three problems are all among the first 100 in [Reinfeld58]. The problem in section A is in the developmental set, but the problems in sections B and C are not in either the developmental set or the test set. Thus PARADISE has not been especially groomed in any way for these latter two problems.

The first example, shown in figure 5.1, is problem 82 in [Reinfeld58]. This problem was chosen so that comparisons can be made with the search tree produced by the CAPS program which is given in [Berliner74]. Such a comparison is interesting since CAPS has more knowledge about tactics than any other program that plays a full game of chess.

All cpu times mentioned are on a DEC KL-10. PARADISE is written in MacLisp. About 75% of the code, including all the frequently used routines, are compiled.

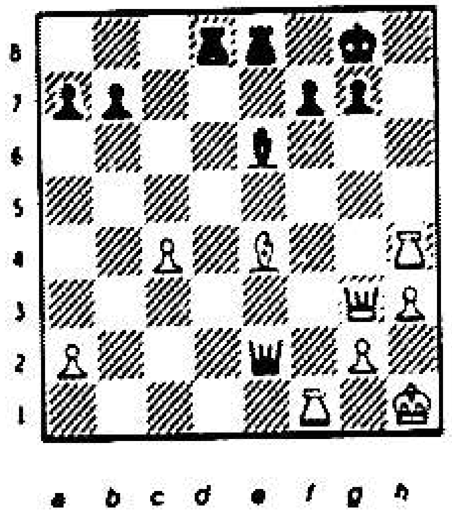


Figure 5.1
white to move

PARADISE's initial static analysis on the above position uses about 20 seconds of cpu time, and produces 7 plans. The one beginning with (WB H7) has a LIKELY of 0 and is searched first. The other six plans have a LIKELY of 1 and begin with the following moves: (WQ A3) (WR H7) (WR H8) (WQ F2) (WQ F3) (WQ E1). The protocol which follows was printed by PARADISE as it analyzed this problem. Explanatory comments have been added in italics. This is a typical small to medium sized tree produced by PARADISE.

```
(TOPLEVEL (VALUE  0) (THRESHOLD  16))
PLY 1 (EXPECT  320)
BESTPLAN: ( (WB H7)  (BK F8)  (WQ A3) )
NEWBOARD:  (E4 WB H7)
```

The value of the initial position is 0 and the threshold is 16 (White is trying to achieve a positive score.) The best plan specifies moving the WB to H7 and if black replies by moving his king to F8, then WQ-A3 is played. The reasoning behind the suggestion of this plan is fairly specific. PARADISE knows that after B-H7 black must move his king either to H8 where white has a discovered check, or to F8 where white can deliver a second check to which black's king cannot move in reply. The expectation of this plan (after "EXPECT") is 320 (90 is the value of a queen, so 320 threatens mate). Wherever the word "NEWBOARD" occurs, PARADISE constructs a new board position by playing a legal move.

```
(PLY: 2) DEFMOVES: ( (BK F8) (BK H8))
NEWBOARD:  (G8 BK F8)
```

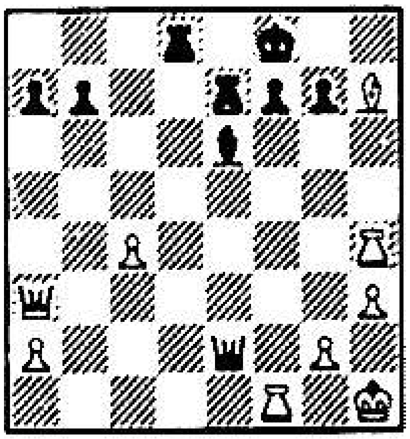
The word "DEFMOVES" labels the list of moves under consideration by the defensive search. (Each move is the first part of a plan under consideration.)

```
TRY CURRENT PLAN
PLY 3 (VALUE  0)  BESTPLAN: ( (WQ A3) )
NEWBOARD:  (G3 WQ A3)
```

The phrase "TRY CURRENT PLAN" means that the program is obtaining the next move by executing a plan that was inherited from an earlier analysis. In this example the current plan specifies an actual move rather than a goal, so the move is made immediately without calculating the primitives in this position. The current evaluation of this position is 0 (given after the word "VALUE").

```
(PLY: 4) DEFMOVES: ( (BR E7))
NEWBOARD:  (E8 BR E7)
```

PARADISE knows (BR D6) will not help, CAPS searches it after refuting (BR E7).



```
NULL PLAN  (VALUE  0)  (STATIC-ANALYSIS 20.4 SECONDS)  (4 PLANS)
PLY 5  BESTPLAN: (((WB D3) ((NIL (CHECKMOVE WR H8)) (NIL (SAFECAPTURE WB BQ))
(NIL (SAFEMOVE WR H8) (((BK NIL) (SAFECAPTURE WR BR)) ((ANYBUT BK) (SAFECAPTURE WR BK)))))))
(NEWEXPECT . 90)
NEWBOARD:  (H7 WB D3)
```

The current plan is finished and there are no obviously winning moves so the system does a static analysis which takes 20.4 seconds of cpu time and produces 4 plans. The best one begins with (WB D3) and continues with either a rook check on H8 threatening mate, the capture of the black queen by the bishop, or the rook move to H8 followed by a skewer of the black king to the black rook. Once again the static analysis accurately recommends the winning plan. This is an improvement over the performance of CAPS which finally suggested B-D3 as a defensive move since it protected the white pawn and white rook which are both en prise. The expectation is now 90, having been recalculated from the new current plan which only expects to win the black queen.

```
(PLY: 6) DEFMOVES: ((BR D3) (BR D3) (BQ D3) (BQ F1) (BP G5) (BQ D3) (BK G8))
NEWBOARD:  (D8 BR D3)

TRY OBVIOUSLY WINNING MOVE
PLY 7 (VALUE  -33)  BESTPLAN: ( (WR H8) )
(NEWEXPECT  607)
NEWBOARD:  (H4 WR H8)
(EXIT OFFENSE (VALUE 1300  1300))
```

The system finds R-H8 as an obviously winning move. It knows the move will mate (although it plays it to make sure), so it doesn't check the current plan. If the system hadn't been sure of the mate, it would have executed the current plan after noticing that R-H8 duplicates the (CHECKMOVE WR H8) goal in the current plan.

```
(PLY 6) REFUTE: ( (WR H8))
(PLY: 6) DEFMOVES: ( (BQ D3) (BQ F1) (BP G5) (BQ D3) (BK G8))
CAUSALITY:  (BQ D3) LINE:  (D8 BR D3) NO
NEWBOARD:  (E2 BQ F1)
```

PARADISE backs up to ply 6. A counter-causal analysis tries to refute the move R-H8 by white but no new moves are suggested. The causality facility compares the proposed Q-D3 move to the tree produced for the R-D3 move and determines that Q-D3 cannot help. Q-D3 is therefore rejected without searching and Q-F1 is played (the causality facility approves it). The causality facility makes use of considerable information returned by the searching process. CAPS, which also has a causality facility, searches both R-D3 and Q-D3 in this position. In fact, more than half the moves rejected by causality in the remainder of this protocol are searched by CAPS.

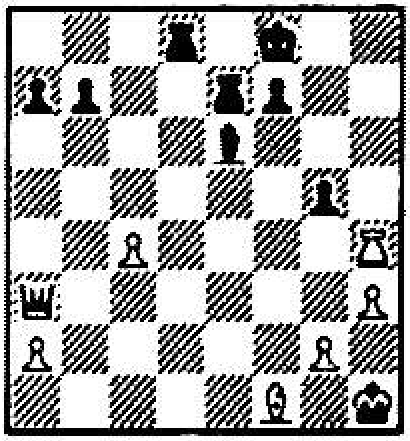
```
OBVIOUSLY WINNING MOVE DUPLICATED
TRY CURRENT PLAN
PLY 7 (VALUE  -50)  BESTPLAN: ( (WB F1) )
NEWBOARD:  (D3 WB F1)
```

B-F1 is suggested as a winning move, but it duplicates the SAFECAPTURE goal in the current plan (and is not a sure mate) so the current plan is tried.

```
(PLY: 8) DEFMOVES: ( (BR D1))
NEWBOARD:  (D8 BR D1)

(VALUE  49)  TRY QUIESCENCE SEARCH
(PLY: 9) NEWBOARD:  (H4 WR H8)
(QUIESCENCE VALUE (1300  1300))
```

The current plan is finished and the threshold has been achieved so the offense tries a quiescence search to see if the value holds up. A value of 1300 (which is mate) is returned so the search backs up with success for white.

```
(PLY 8) REFUTE: ( (WR H8))
(PLY: 8) DEFMOVES: ( (BP G5) (BR D2) (BR D7) (BP G6) (BP F5) (BP F6) (BK E8) (BK G8))
NEWBOARD:  (G7 BP G5)
```



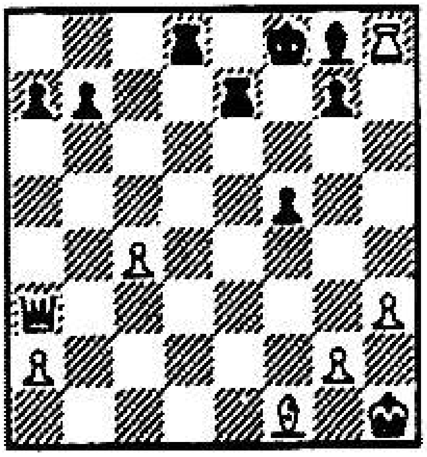
```
(VALUE   49)  TRY QUIESCENCE SEARCH
(PLY: 9) NEWBOARD:  (H4 WR H8)
DEFENDING MOVE SELECTED
(PLY: 10) NEWBOARD:  (F8 BK G7)
(PLY: 11) NEWBOARD:  (H8 WR D8)
DEFENDING MOVE SELECTED
(PLY: 12) NEWBOARD:  (E7 BR D7)
(PLY: 13) NEWBOARD:  (D8 WR D7)
(PLY: 14) NEWBOARD:  (E6 BB D7)
(PLY: 15) NEWBOARD:  (A3 WQ A7)
DEFENDING MOVE SELECTED
(PLY: 16) NEWBOARD:  (D7 BB C6)
(QUIESCENCE VALUE (109   109))
(EXIT OFFENSE (VALUE 109   109))
```

A quiescence search to a depth of 16 shows that P-G5 fails for black. The quiescence search knows that the threshold has been achieved and that the white rook on H4 is not in danger, yet it plays aggressive moves (at plys 9 through 15) where the regular search would be satisfied with the result already achieved. This seems to generate nodes unnecessarily. PARADISE does this because it is a cheap way to get better results. The quiescence search is very i:.expensive compared to the regular search. The system thinks it has enough information, yet it may be necessary to search this line again if the "true" value is not found. Since PARADISE sees an inexpensive way (quiescence searching) to improve the result, it risks generating a few unnecessary (though inexpensive) nodes in order to avoid a possible re-search later.

```
(PLY 8) REFUTE: ( (WR H8))
(PLY: 8) DEFMOVES: ( (BR D2) (BR D7) (BP G6) (BP F5) (BP F6) (BK E8) (BK G8))
CAUSALITY:  (BR D2) LINE:  (D8 BR D1) NO
CAUSALITY:  (BR D7) LINE:  (D8 BR D1) NO
CAUSALITY:  (BP G6) LINE:  (G7 BP G5) NO
NEWBOARD:  (F7 BP F5)
```

Black tries other moves at ply 8. Causality rejects R-D2 and R-D7 on the basis of the tree generated for R-D1, and it rejects P-G6 using the P-G5 tree.

```
(VALUE . 49) TRY QUIESCENCE SEARCH
(PLY: 9) NEWBOARD: (H4 WR H8)
DEFENDING MOVE SELECTED
(PLY: 10) NEWBOARD: (E8 BK G8)
```



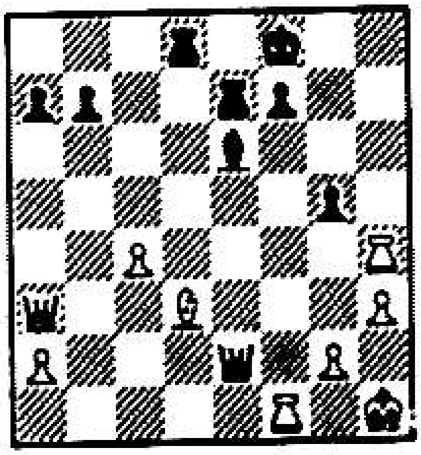
```
(PLY: 11) NEWBOARD: (A3 WQ A7)
(QUIESCENCE VALUE (59 . 59))
(EXIT OFFENSE (VALUE 59 . 90))
```

Again the quiescence search confirms the offensive success. This time the value of 59 does not meet the expectation so the offensive search returns a range of 59 to 90 since the expectation of 90 may have been achieved if a static analysis had been done. An ANALYZE pointer is put in the tree.

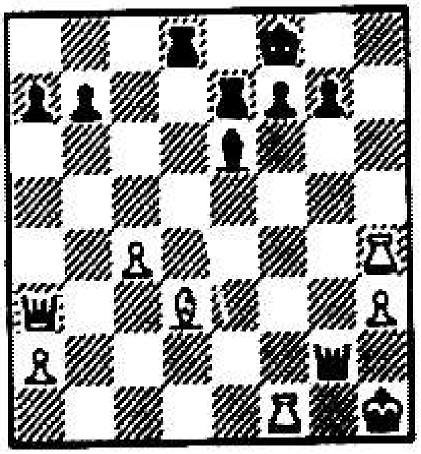
```
(PLY 8) REFUTE: ( (WR H8))
(PLY: 8) DEFMOVES: ( (BP F6) (BK E8) (BK G8))
CAUSALITY: (BP F6) LINE: (F7 BP F6) NO
NEWBOARD: (F8 BK E8)

(VALUE . 49) TRY QUIESCENCE SEARCH
(PLY: 9) NEWBOARD: (A3 WQ A7)
(QUIESCENCE VALUE (59 . 59))
(EXIT OFFENSE (VALUE 59 . 90))

(PLY 8) REFUTE: ( (WQ A7))
(PLY: 8) DEFMOVES: ( (BK G8) (BR A8) (BP A6) (BP A5) (BP B6))
CAUSALITY: (BK G8) LINE: (F8 BK E8) NO
CAUSALITY: (BR A8) LINE: (D8 BR D1) NO
CAUSALITY: (BP A6) LINE: (D8 BR D1) NO
CAUSALITY: (BP A5) LINE: (D8 BR D1) NO
CAUSALITY: (BP B6) LINE: (F8 BK E8) NO
DEFMOVES: ()

PLY 7  BESTPLAN: ((CHECKMOVE WR H8) )
 TERMINATE: PLAN SUCCEEDED
(EXIT OFFENSE (VALUE 59 . 90))
```

The offensive search terminates since the threshold has been achieved. This inserts RE-SEARCH pointers in the tree for plans still under consideration and an ANALYZE pointer since there has been no static analysis at this node. The range of 59 to 90 is returned.

```
(PLY 6) REFUTE: ( (WR H8) (WR H8) (WR H8) (WB F1))
(PLY: 6) DEFMOVES: ( (BP G5) (BK G8))
NEWBOARD:  (G7 BP G5)
```



```
OBVIOUSLY WINNING MOVE DUPLICATED
TRY CURRENT PLAN
PLY 7  (VALUE . 0)  BESTPLAN: ( (WB E2) )
NEWBOARD:  (D3 WB E2)

(PLY: 8) DEFMOVES: ( (BP H4))
NEWBOARD:  (G5 BP H4)

(VALUE . 49)  TRY QUIESCENCE SEARCH
(PLY: 9) NEWBOARD:  (A3 WQ A7)
(QUIESCENCE VALUE (59 . 59))
(EXIT OFFENSE (VALUE 59 . 90))

(PLY: 8) DEFMOVES: ()
PLY 7  BESTPLAN: ((CHECKMOVE WR H8) )
 TERMINATE: ALPHA BETA

(PLY 6) REFUTE: ( (WB E2))
(PLY: 6) DEFMOVES: ( (BQ E3) (BQ D2) (BQ E5) (BQ G2) (BQ A2) (BK G8))
CAUSALITY:  (BQ E3) LINE:  (D8 BR D3) NO
CAUSALITY:  (BQ D2) LINE:  (D8 BR D3) NO
CAUSALITY:  (BQ E5) LINE:  (D8 BR D3) NO
NEWBOARD:  (E2 BQ G2)
```



```
TRY OBVIOUSLY WINNING MOVE
PLY 7  (VALUE . -10)  BESTPLAN: ( (WK G2) )
NEWBOARD:  (H1 WK G2)
(PLY: 8) DEFMOVES: ()
```

The defensive search now does a null move analysis. It has no idea what to do so it lets the offense make two moves in a row. However, the offense calls the quiescence search which knows it is the defense's move. It decides the defense can escape from the threat of R-H8 so it calls the position quiescent.

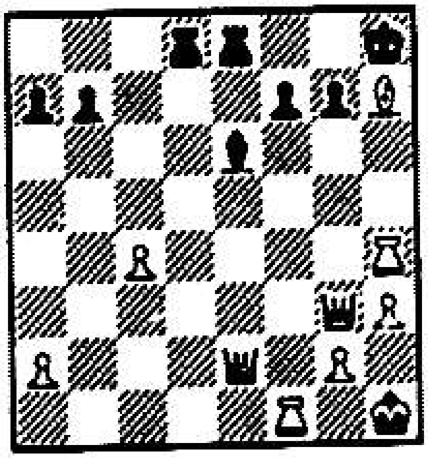
```
(VALUE . 89)  TRY QUIESCENCE SEARCH
assumed escape from (WR H8)
(QUIESCENCE VALUE (89 . 89))
(EXIT OFFENSE (VALUE 89 . 89))

(PLY: 8) DEFMOVES: ()
(PLY 6) REFUTE: ( (WK G2))
(PLY: 6) DEFMOVES: ( (BQ A2) (BK G8))
CAUSALITY:  (BQ A2) LINE:  (D6 BR D3) NO
CAUSALITY:  (BK G8) LINE:  (G7 BP G5) NO
(PLY: 6) DEFMOVES: ()

PLY 5   BESTPLAN: ( (WQ A5) NIL (CHECKMOVE WQ D8) )
no unlikelys when threshold achieved: quit
(EXIT OFFENSE (VALUE 59 . 89))
```

The search backs up to ply 5 and tries the next plan suggested by the static analysis. This plan has a LIKELY of 1 and the threshold has already been achieved, so it is rejected without searching. No RE-SEARCH pointers are inserted in the tree.

```
(PLY 4) REFUTE: ( (WR H8) (WB D3))
(PLY: 4) DEFMOVES: ()
PLY 3  LAST PLAN
(EXIT OFFENSE (VALUE 59 . 320))
```

The search backs up to ply 3. There are no more plans to execute here, but a static analysis has not yet been done so an ANALYZE pointer is inserted in the tree and the upper current value is changed to 320 to reflect the possibility of achieving the original expectation.

```
(PLY 2) REFUTE: ( (WQ A3))
(PLY: 2) DEFMOVES: ( (BK H8))
NEWBOARD:  (G8 BK H8)
```



```
NULL PLAN (VALUE 0) (STATIC-ANALYSIS 14.2 SECONDS) (2 PLANS)
PLY 3  BESTPLAN: ( (WB D3) )
(NEWEXPECT . 320)
NEWBOARD: (H7 WB D3)
```

The defense tries K-H8 at ply 2 and the original plan does not match this move so a static analysis is undertaken. Again PARADISE finds the right idea immediately. To compare, CAPS did not have the necessary knowledge and tried 3 other moves first, generating 80 nodes to disprove them, before stumbling onto B-D3.

```
(PLY: 4) DEFMOVES: ( (BK G8))
NEWBOARD: (H8 BK G8)

(VALUE 0) TRY OBVIOUSLY WINNING MOVE
PLY 5  BESTPLAN: ( (WB E2) )
(NEWEXPECT  90)
NEWBOARD: (D3 WB E2)

(PLY: 6) DEFMOVES: ()
(VALUE  99) THY QUIESCENCE SEARCH
(QUIESCENCE VALUE (99  99))
(EXIT OFFENSE (VALUE 99  99))

(PLY: 6) DEFMOVES: ()
PLY 5   LAST PLAN
(EXIT OFFENSE (VALUE 99  320))
(PLY 4) REFUTE: ( (WB E2))
(PLY: 4) DEFMOVES: ()

PLY 3   BESTPLAN: (( (WB E4) )
(NEWEXPECT  10)
 TERMINATE: FORWARD PRUNE
(EXIT OFFENSE (VALUE 99  320))
```

The search backs up to ply 3 and the offense tries B-E4 which has been suggested by the static analysis. It's expectation is 10 and B-D3 has already achieved 99 so a forward prune occurs. The search terminates without inserting RE-SEARCH pointers for LIKELY 0 plans. They are inserted for LIKELY 1 plans which would increase the upper current value except that it has already been increased to 320 at ply 3 since no static analysis was done there.

```
(PLY 2) REFUTE: ( (WB D3))
(PLY: 2) DEFMOVES: ()

PLY 1  BESTPLAN: ( (WQ A3) NIL (CHECKMOVE WB H7) )
no unlikelys when threshold achieved: quit
```

The search returns to the top level to try the next plan but it is terminated because it has a LIKELY of 1 and the search is over. CAPS invests much effort at this point searching other offensive moves.

```
BEST MOVE: (E4 WB H7)   VALUE: (99 . 320)
PRINCIPAL VARIATION:
1 (E4 WB H7)      (G8 BK F8)
2 (G3 WQ A3)      (E8 BR E7)
3 (H7 WB D3)      (E2 BQ F1)
4 (D3 WB F1)      (F7 BP F5)
5 (H4 WR H8)      (E6 BB G8)
6 (A3 WQ A7)
```

```
TOTAL TIME: 297 SECONDS
NODES CREATED: 36   (22 REGULAR, 14 QUIESCENCE)
STATIC ANALYSES: 2

WHERE TIME WAS SPENT:
49%   calculating primitives
33%   quiescence searching (overlaps primitive calculations for 14 nodes)
11%   static analysis
9%    defense determining initial move
5%    defensive counter-causal analysis
3%    disk IO
2%    offensive plan elaboration
0%    causality facility, evaluation function, creating board positions
```

Many things should be noticed in this example. The plans do an excellent job of guiding the search: only two static analyses are done in the entire search. The plans are so accurately specified by the analysis that they never once lead the search off the correct line. White's play in the search is error-free. The causality facility makes good use of information returned from the search. Many black moves which would otherwise have been searched are eliminated in this manner. This same information is used by the counter-causal analysis to generate all reasonable defenses for black. The range values accurately express the potential of each node so that the system does not have to waste effort determining the exact values. Tnu best first search strategy plays a minor role in this example. The threshold is used to make many cutoffs, but only one search had to be initiated from the top level. The example in the next section shows more of the search strategy.

On this problem, CAPS generated a tree of 489 nodes in 115 seconds to obtain a principal variation of B-R7ch, K-B1, Q-R3ch, R-K2, B-Q3, QxRch, BxQ, R-Q8. This is slightly inaccurate since there is a mate in one for white at the end of this variation which CAPS's quiescence analysis did not recognize, but it is clear that CAPS understands the problem and its solution. PARADISE generates only 36 nodes to CAPS's 489 for the following 3 reasons (primarily):

-PARADISE has more knowledge available during static analysis and can accurately analyze a position and produce good plans. CAPS generates many nodes by not playing the correct offensive move first on some occasions. This knowledge is also the reason PARADISE uses more than twice as much cpu time to produce its 36 nodes.

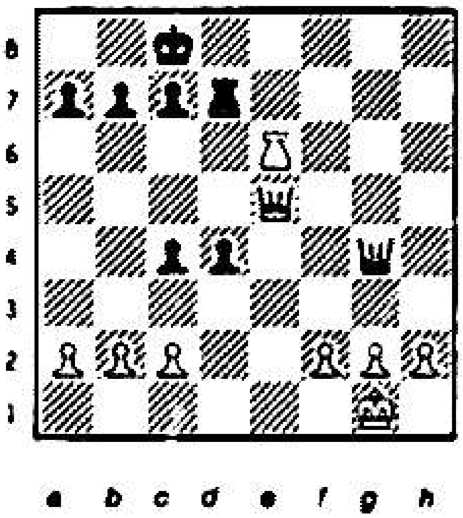
-PARADISE returns more useful information from its search and can therefore use its causality facility to eliminate moves that CAPS searches.

-PARADISE has a best-first search strategy while CAPS is committed to finding the true value (within alpha-beta) of each node it searches. This enables PARADISE to terminate as soon as some information is discovered without having to look for better alternatives.

This comparison shows the advances PARADISE has made in the use of knowledge. It is not meant to belittle CAPS which pioneered some of the techniques basic to this approach. In fact, CAPS is the only program with which a comparison is appropriate. For example, on this problem CHESS 4.4 (running on a CDC-6400) produces a tree with 30,246 nodes in 95 seconds of cpu time without finding the solution (which is too deep for it). Looking at the details of such a tree would not be helpful.

B) An example of the DISPROVE-REST strategy

PARADISE solves the problem in figure 5.2 using its DISPROVE-REST strategy. This is the first instance of a chess program using such an indirect strategy to solve a problem. This problem is number 69 in [Reinfeld58] and the winning line (not found by the program) is shown in the figure. The tree is the largest yet produced by PARADISE, so only the top level decisions made by the search strategy are given below.



1. Pf2f3	Qg4h4
2. Re6e8ch	Rd7d8
3. Qe5e6ch	Kc8b8
4. Qe6d7	Qh4e1ch
5. Re8e1	Rd8d7
6. Re1e8ch	Rd7d8
7. Re8d8 mate	

Figure 5.2
white to move

PARADISE's static analysis suggests 9 plans to try in this position. The first move of each plan (in the order tried) is: (WR E8) (WQ H8) (WR H6) (WR F6) (WQ F6) (WQ C5) (WR G6) (WP F3) (WP H3). The value of the initial position is 10 and the threshold is initially 16. PARADISE first searchs (WR E8) and returns (10 . 380) as its value. The "better plan at higher level" cutoff has terminated the search at two different points so the value has not been completely determined. There are RE-SEARCH pointers for LIKELY 1 plans at depths of 3 and 7 in the tree and a RE-SEARCH pointer for a LIKELY 0 plan at a depth of 3 in the tree (at the same node as the LIKELY 1 RE-SEARCH).

PARADISE now searches (WQ H8) at the top level, finds that white cannot win anything, and returns a value of (10 . 10). (WR H6) is tried and a value of (-1300 . -1300) is returned since white gets mated. By analyzing the tree returned from this search, the causality facility eliminates (WR F6) (WQ F6) and (WR G6) from consideration. (WQ C5) is searched but also returns (-1300 . -1300). Up to this point, PARADISE has generated less than 20 nodes and invested little effort.

(WP F3) is now searched and a value of (20 . 100) is returned. This search produces about 90 nodes since many different defenses are tried and the offense misses the best move a few times. There are two RE-SEARCH pointers in this tree on branches where the offense is not already ahead by a rook or more. One is an ANALYZE pointer at ply 3 after black replies at ply 2 with (BR G7). The other is an ANALYZE pointer at ply 5 on the winning line shown in figure 5.2. When the node with this pointer is searched again, the system will do a static analysis and find the winning line shown above. The quiescence search of this node finds that white can win one of the black pawns, so a lower value of 20 is returned reflecting the win of a pawn by white. PARADISE then generates about 60 nodes searching (WP H3) and a value of (10 . 10) is returned.

PARADISE is at the top level again and has searched each plan once. Only two remain under consideration, (WR E8) with a value of (10 . 380) and (WP F3) with a value of (20 .100). Since the tree for the rook move has a LIKELY 0 RE-SEARCH pointer and the plan beginning with (WP F3) is LIKELY 1, the node with the LIKELY 0 pointer is chosen for searching. PARADISE traverses the tree to this node and initiates a search of the plan in the RE-SEARCH pointer. This search fails (after generating 5 nodes), so PARADISE returns again to the top level. The value of (WR E8) is still (10 . 380), but there are no LIKELY 0 RE-SEARCH pointers in its tree.

PARADISE now selects the DISPROVE-REST strategy since both plans are LIKELY 1, and one of them has already achieved some positive good. The system is trying to show that (WP F3) is best by investigating all the RE-SEARCH pointers in the tree of (WR E8), and hopefully showing that this plan cannot win more than a pawn. If there had been more than these two plans under consideration, the system would attempt to improve the value for (WP F3) (since it has already achieved something) and would find the winning line in figure 5.2. Selecting DISPROVE-REST causes another search of (WR E8) to be initiated from the top level. Four plans are searched at ply 3 (each has a RE-SEARCH pointer at the same node), but they all return ranges whose upper value is 10 or less (about 15 nodes are generated). The search returns again to the top level. The value of (WR E8) is still (10 . 380) since RE-SEARCH pointers remain at ply 7 in its tree.

Once again PARADISE selects the DISPROVE-REST strategy. (Nothing important has changed; if the previous search of plans at ply 3 had achieved something positive, a different strategy might be selected at this point.) Yet another search of (WR E8) is initiated from the top level. This time the system traverses down to the RE-SEARCH pointers at ply 7 and searches both plans specified there. These plans fail (after generating 15 nodes) and a value of (10 . 10) is returned to the top level since there are no more possibilities to investigate.

PARADISE now concludes that (WP F3) is the best move since it wins at least a pawn and possibly much more, while no other plan can win more than a pawn. The program reaches this conclusion without finding the principal variation shown in figure 5.2. This reasoning is valid as well as being novel for a computer chess program. It should be noted why PARADISE finds 10 as the value of (WR E8) and (WP H3) when these values could be 20. PARADISE does see opportunities to win a pawn in the searches of both (WR E8) and (WP H3). However, it knows that the pawn-winning move can not achieve more than the win of a pawn so the search is terminated by an alpha-beta cutoff before the pawn-winning move is made (since (WP F3) has already won a pawn).

This example shows how the system continually backs up to the top level and reassesses the situation before initiating more searching. This is important because it avoids wasting effort by searching poor variations at great length.

C) A best-first search which restarts searches

This last example is a search where PARADISE makes use of its best-first search strategy to control the growth of the tree, and then re-searches lines until the best move is determined. The problem is shown in figure 5.3 (number 96 in [Reinfeld58]). The winning combination is too deep to be found by most computer chess programs. CAPS can search deep enough, but does not find the combination because it lacks the necessary knowledge.

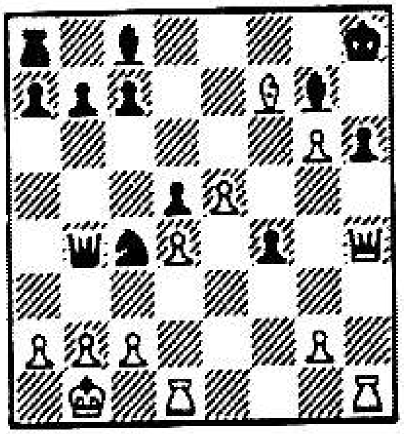


Figure 5.3
white to move

PARADISE's initial static analysis suggests 4 plans in this position. They all have a LIKELY of 0 and begin with the following moves: (WQ D8) (WQ H6) (WB D5) (WQ F4). The actual protocol printed by the program is presented below with comments in italics.

```
(TOPLEVEL (VALUE  -5) (THRESHOLD  18))
PLY 1 (EXPECT  320)
BESTPLAN: ( (WQ D8) )
NEWBOARD:  (H4 WQ D8)
```

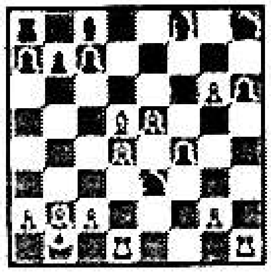
The value of the initial position is -5 and the threshold is 18. (White is trying to achieve a positive score.) The expectation shows a mate. This move will actually lead to mate if followed by RXH6

```
(PLY: 2) DEFMOVES: ( (BQ F8) (BB F8))
NEWBOARD:  (B4 BQ F8)
```

```
NULL PLAN (VALUE -5) (STATIC-ANALYSIS 14.6 SECONDS) (4 PLANS)
PLY 3 BESTPLAN: (((WR H6) (BB H6) (SAFEMOVE WQ F6) (BB NIL) (SAFEMOVE WR H1))
(NEWEXPECT 185)
 TERMINATE: BETTER PLAN AT HIGHER LEVEL
```

The current plan has expired so a static analysis is done. The best plan suggested begins with a rook sacrifice which threatens mate, but the expectation is only 185 because of the sacrifice. PARADISE therefore terminates the search because one of the plans at the top level looks more promising. RE-SEARCH pointers are inserted in the tree for each plan produced by this static analysis. The system will later return to this node and search the plan shown above. Before backing up to the top level, a quiescence search is done so that an accurate lower value can be returned for this node.

```
QUIESCENCE ANALYSIS
(PLY: 3) NEWBOARD: (D8 WQ F8)
DEFENDING MOVE SELECTED
(PLY: 4) NEWBOARD: (G7 BB F8)
(PLY: 5) NEWBOARD: (F7 WB D5)
(PLY: 6) NEWBOARD: (C4 BN E3)
```



```
DEFENDING MOVE SELECTED
(PLY: 7) NEWBOARD: (D5 WB F3)
(PLY: 8) NEWBOARD: (E3 BN D1)
(PLY: 9) NEWBOARD: (F3 WB D1)
(PLY: 5) value taken here
(EXIT OFFENSE (VALUE -5 185))
```

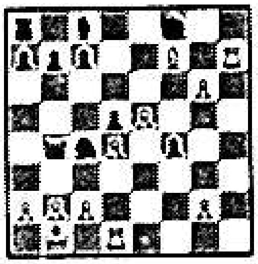
The quiescence search exchanges queens and then tries to grab a pawn at ply 5. This fails but the system knows that the bishop move was made purely for offensive reasons (black had no threats), so it uses the value at ply 5. Thus a range of (-5 185) is returned to the top level which reflects the fact that nothing has been won but a value of 185 is still threatened.

```
PLY 1 BESTPLAN: (((WQ H6) ((NIL H6) (SAFEMOVE WR H6))(NIL (SAFEMOVE WR H6))))
(NEWEXPECT 268)
NEWBOARD: (H4 WQ H6)
```

A new plan is tried at the top level. It will not lead to a quick mate, but it would if there were a white pawn on D6 and black did not threaten mate on the move. Thus this is a promising plan and cannot be dismissed using static analysis without search (even by good human players).

```
(PLY: 2) DEFMOVES: ( (BB H6))
NEWBOARD: (G7 BB H6)
```

```
TRY CURRENT PLAN
PLY 3  (VALUE  -94)  BESTPLAN: ( (SAFEMOVE WR H6) )
NEWBOARD:  (H1 WR H6)

(PLY: 4) DEFMOVES: ( (BK G7))
NEWBOARD:  (H8 BK G7)

NULL PLAN  (VALUE  -61)  (STATIC-ANALYSIS 7.1 SECONDS) (2 PLANS)
PLY 5  BESTPLAN: ( (WR H7) (BK NIL) (MATEMOVE WR H8) )
(NEWEXPECT   268)
NEWBOARD:  (H6 WR H7)

(PLY: 6) DEFMOVES: ( (BK F8))
NEWBOARD:  (G7 BK F8)
```



```
TRY CURRENT PLAN
PLY 7  (VALUE  -61)  BESTPLAN: ( (MATEMOVE WR H8) )
UNEXECUTABLE PLAN INHERITED
(STATIC-ANALYSIS 9.4 SECONDS) (2 PLANS)
BESTPLAN: ( (WB D5) NIL (CHECKMOVE WR F7) )   (NEWEXPECT  -42)
 TERMINATE: FORWARD PRUNE
BESTPLAN: ( (WB D5) NIL (CHECKMOVE WR F7) )   (NEWEXPECT  268)
 TERMINATE: BETTER PLAN AT HIGHER LEVEL
QUIESCENCE ANALYSIS   TERMINATE: ALPHA BETA
(QUIESCENCE VALUE (-61   -61))   (EXIT OFFENSE (VALUE -61   268))
```

The current plan fails since (WR H8) is not a mating move, so a static analysis is done. The best LIKELY 0 plan has an expectation of -42 which is worse than the already achieved value of -5, so all LIKELY 0 attribute lists are removed by the forward prune cutoff. The best LIKELY 1 plan happens to be the same as the best LIKELY 0 plan, but its expectation is now 268. Since it is LIKELY 1, PARADISE decides to terminate since there are more promising plans elsewhere. RE-SEARCH pointers are inserted in the tree for both LIKELY 1 plans. A quiescence search tries to find an accurate lower value for this position, but the position is much worse than the -5 already achieved so an alpha-beta cutoff occurs. The offense backs up with a range of (-61 268) for this node.

```
PLY 5  BESTPLAN: ( (WR H8) (BK H8) (CHECKMOVE WR H1) (BK NIL) (ATTACKP BK) )
(NEWEXPECT   208)
 TERMINATE: BETTER PLAN AT HIGHER LEVEL
QUIESCENCE ANALYSIS   TERMINATE: ALPHA BETA
(QUIESCENCE VALUE (-61   -61))   (EXIT OFFENSE (VALUE -61   268))

(PLY 3) BACK UP FOR BETTER PLAN AT HIGHER LEVEL. RE-SEARCH WILL ANALYZE HERE.
(EXIT OFFENSE (VALUE -61   268))
```

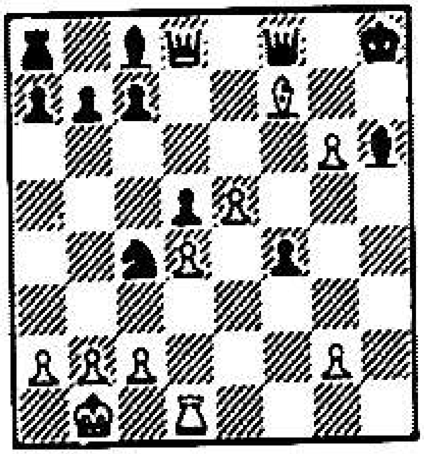
The search refuses to interrupt the backing up process at ply 3 in order to do a static analysis. Instead an ANALYZE pointer is inserted in the tree at this node and the search returns to the top level.

```
PLY 1  BESTPLAN: ( (WB D5) )
(NEWEXPECT  5)
NEWBOARD:  (F7 WB D5)

(PLY: 2) DEFMOVES: ( (BQ B2))
NEWBOARD:  (B4 BQ B2)
(EXIT OFFENSE (VALUE -1300  -1300))

(PLY 1) REFUTE: ( (BQ B2))
PLY 1  BESTPLAN: ( (WQ F4) )
(NEWEXPECT  5)
CAUSALITY:  (WQ F4) LINE:  (F7 WB D5) NO

LAST PLAN
TOTAL TIME: 148 SECONDS
NODES CREATED. 17   (10 REGULAR, 7 QUIESCENCE)
STATIC ANALYSES: 3
```

The third plan at the top level attempts to win a pawn, but white is quickly mated. The refutation is analyzed, but no new plans are suggested since none of them begin with captures (see chapter 4). The causality facility eliminates the fourth plan. Each plan has now been searched once, so the program prints some statistics.

PARADISE now picks a node with a LIKELY 0 RESEARCH pointer to search again. The plan beginning with (WQ D8) is preferred since it has achieved the best lower value. There are RESEARCH pointers at ply 3 from the first static analysis done in this search since a "better plan at higher level" cutoff occurred.

```
2 PLANS FOR RESEARCH:  ((WQ D8) (-5  185))  ((WQ H8) (-61 . 268))
(RESEARCH AT PLY 3 ON  (WQ D8)) Try for success
(RESEARCH (H4 WQ D8) PLY 1) (RESEARCH (B4 BQ F8) PLY 2)

PLY 3  BESTPLAN: (((WR H6) (BB H6)) (SAFEMOVE WQ F6) (BB NIL) (SAFEMOVE WR H1 ))
(NEWEXPECT  185)
NEWBOARD:  (H1 WR H6)

(PLY: 4) DEFMOVES: ( (BB H6))
NEWBOARD:  (G7 BB H6)
```



```
TRY CURRENT PLAN
PLY 5  (VALUE  -45)  BESTPLAN: ( (SAFEMOVE WQ F6) (BB NIL) (SAFEMOVE WR H1) )
NEWBOARD:  (D8 WQ F6)

(PLY: 6) DEFMOVES: ( (BB G7) (BQ G7))
NEWBOARD:  (H6 BB G7)
```

```
OBVIOUSLY WINNING MOVE DUPLICATED
TRY CURRENT PLAN
PLY 7  (VALUE  -45)  BESTPLAN: ( (SAFEMOVE WR H1) )
NEWBOARD:  (D1 WR H1)
```

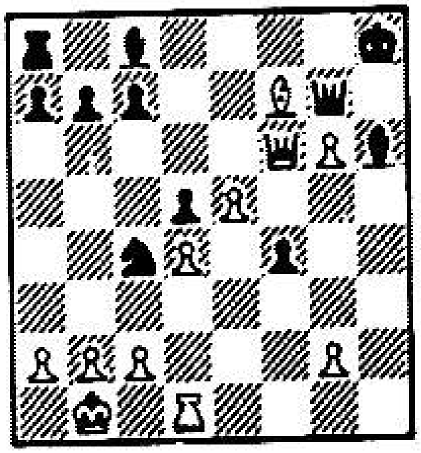
The search is still being guided by the plan in the RESEARCH pointer. This plan was produced by the first static analysis of this protocol (at ply 3 of the original search of (WQ D8)).

```
(PLY: 8) DEFMOVES: ( (BB H3))
NEWBOARD:  (C8 BB H3)
```

This is played since it is the only legal move and PARADISE does not bother to calculate the primitives for this position. If any analysis whatsoever was done, the search would be terminated without playing this move.

```
TRY OBVIOUSLY WINNING MOVE
PLY 9  (VALUE  -45)
NEWBOARD:  (H1 WR H3)
(EXIT OFFENSE (VALUE 1300  1300))

(PLY: 8) DEFMOVES: ()
(PLY 6) REFUTE: ( (WR H3) (WR H1))
(PLY: 6) DEFMOVES: ( (BQ G7))
NEWBOARD:  (F8 BQ G7)
```

Having found the mate, PARADISE backs up and tries the queen interposition at ply 6 for black.



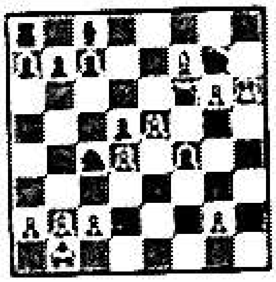
```
NULL PLAN  (VALUE  -45)  (STATIC-ANALYSIS 17.9 SECONDS) 8 PLANS)
PLY 7  BESTPLAN: ( (WQ D8) )
(NEWEXPECT  185)
NEWBOARD:  (F6 WQ D8)

(PLY: 8) DEFMOVES: ( (BQ F8) (BQ G8))
NEWBOARD:  (G7 BQ F8)
 TERMINATE: REPETITION

PLY 7  BESTPLAN: ( (WR H1) NIL (CHECKMOVE WR H6) )
(NEWEXPECT  185)
NEWBOARD:  (D1 WR H1)
```

The first plan tried at ply 7 having failed, the search backs up to ply 7 (the position in the last diagram) and tries the second plan suggested by the static analysis.

```
(PLY: 8) DEFMOVES: ( (BQ F6))
NEWBOARD:  (G7 BQ F6)

TRY CURRENT PLAN
PLY 9  (VALUE  -144)  BESTPLAN: ( (CHECKMOVE WR H6) .
NEWBOARD:  (H1 WR H6)
```

The current plan leads the system astray. PxQ is not seen as an obviously winning move since it only wins the queen while white is more than a queen behind.

```
(PLY: 10) DEFMOVES: ( (BK G7))
NEWBOARD:  (H8 BK G7)
```

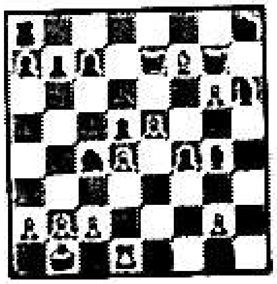


Without its queen, the offense decides that another plan at ply 7 is better than the current line so a quiescence analysis is done to obtain a lower value for the range to be returned.

```
(PLY 11)  (VALUE  -111)
 BETA IMPLIES BETTER PLAN AT HIGHER LEVEL. RE-SEARCH WILL ANALYZE HERE.
QUIESCENCE ANALYSIS
(PLY: 11) NEWBOARD:  (E5 WP F6)
DEFENDING MOVE SELECTED
(PLY: 12) NEWBOARD:  (G7 BK H6)
(PLY: 13) NEWBOARD:  (F7 WB D5)
assumed escape from (WB C4)
(EXIT OFFENSE (VALUE -52  185))

(PLY 9) BACK UP FOR BETTER PLAN AT HIGHER LEVEL. RE-SEARCH WILL ANALYZE HERE.
(EXIT OFFENSE (VALUE -52  185))

PLY 7  BESTPLAN: ( (WQ E7) (NIL (CHECKMOVE WQ E8))NIL (CHECKMOVE WQ F8))) )
(NEWEXPECT  185)
NEWBOARD:  (F6 WQ E7)
```

A third plan which threatens mate is tried at ply 7. This is the plan the system considered to be better at ply 11 in the previous search.

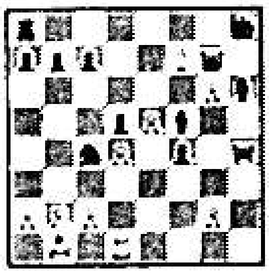
```
(PLY: 8) DEFMOVES: ( (BB G4) (BB F5) (BQ F8) (BQ F7))
NEWBOARD:  (C8 BB G4)
```



```
TRY CURRENT PLAN
PLY 9  (VALUE   -45)  BESTPLAN: ( (CHECKMOVE WQ E8) )
UNEXECUTABLE PLAN     BESTPLAN: ( (CHECKMOVE WQ F8) )
UNEXECUTABLE PLAN INHERITED
 BETA IMPLIES BETTER PLAN AT HIGHER LEVEL. RE-SEARCH WILL ANALYZE HERE.
QUIESCENCE ANALYSIS
assumed escape from (BB D1)
(EXIT OFFENSE (VALUE -45   185))

PLY 7  BESTPLAN: ( (WQ H4) NIL (CHECKMOVE WQ H6) )
(NEWEXPECT   185)
NEWBOARD:  (F6 WQ H4)
```

A fourth plan which threatens mate is tried at ply 7

```
(PLY: 8) DEFMOVES: ( (BB F5))
NEWBOARD:  (C8 BB F5)
```



```
TRY CURRENT PLAN
PLY 9  (VALUE   -45)  BESTPLAN: ( (CHECKMOVE WQ H6) )
UNEXECUTABLE PLAN INHERITED
(STATIC-ANALYSIS 16.3 SECONDS) (1 PLAN)
BESTPLAN: ( (WP G4) NIL (SAFEMOVE WP G5) )
(NEWEXPECT   -22)
 TERMINATE: FORWARD PRUNE
QUIESCENCE ANALYSIS
(PLY: 9) NEWBOARD:  (F7 WB D5)
 TERMINATE: ALPHA BETA
(EXIT OFFENSE (VALUE -35   -35))
```

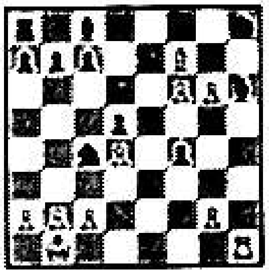
PARADISE again returns to ply 7 (the position in the last large diagram), having searched each plan there that threateneed mate. The first three were terminated because another plan at ply 7 looked more promising, but the fourth plan failed with a value of 35. The system recognizes that the first 3 plans were terminated for this plan that failed, so it decides to search them again immediately (after a fifth plan is rejected by the forward prune cutoff)

```
(PLY 7) REFUTE: ( (BB F5))
PLY 7  BESTPLAN: ( (WB D5) ((NIL (CHECKMOVE WQ F8) )(NIL (SAFECAPTURE WB BN) )))
(NEWEXPECT   -14)
 TERMINATE: FORWARD PRUNE
Search original move since alternative failed
(RESEARCH (F6 WQ E7) PLY 7)   (RESEARCH (C8 BB G4) PLY 8)
(STATIC-ANALYSIS 18.0 SECONDS) (0 PLANS)
(RETURN RE SEARCH (C8 BB G4) PLY 8  OLD-VALUE (-45 . 185) NEW-VALUE (-45 . -45))
(RETURN RE SEARCH (F6 WQ E7) PLY 7 NEW-VALUE (-45 . -45))
```

(WQ E7) is selected first for searching again but it fails when no continuations are suggested. This shows the sensitivity of PARADISE's productions to the position. Rd1h1 does not work in this position but the position in which it does work is very similar. PARADISE suggests this move there but not here. (WR H1) *is now tried again.*

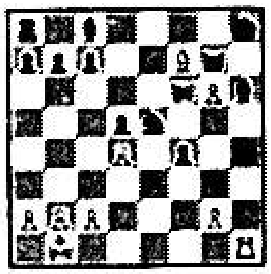
```
(RESEARCH (D1 WR H1) PLY 7)   (RESEARCH (G7 BQ F6) PLY 8)
PLY 9  (STATIC-ANALYSIS 6.7 SECONDS) (2 PLANS)
 TERMINATE: ALREADY SEARCHED (WR H6)
BESTPLAN: ( (WP F6) NIL (MATEMOVE WR H6) )
(NEWEXPECT   185)
NEWBOARD:  (E5 WP F6)

(PLY: 10) DEFMOVES: ()
```

A static analysis suggests RxB and PxQ at ply 9, but RxB is already in the tree since it was searched on the first search from this node. Therefore it is rejected and PxQ is tried. The defense cannot find a reasonable move and tries a null move analysis.



```
OBVIOUSLY WINNING MOVE DUPLICATED
TRY CURRENT PLAN
PLY 11  (VALUE  -45)  BESTPLAN: ( (MATEMOVE WR H6) )
NEWBOARD:  (H1 WR H6)
(EXIT OFFENSE (VALUE 1300 . 1300))

(PLY 10) REFUTE: ( (WR H6))
(PLY: 10) DEFMOVES: ()
(RETURN RESEARCH (G7 BQ F6) PLY 8 OLD-VALUE (-52 . 185) NEW-VALUE (1300 . 1300))

(PLY 8) REFUTE: ( (WR H6) (WP F6))
(PLY: 8) DEFMOVES: ( (BN E5))
NEWBOARD:  (C4 BN E5)
```

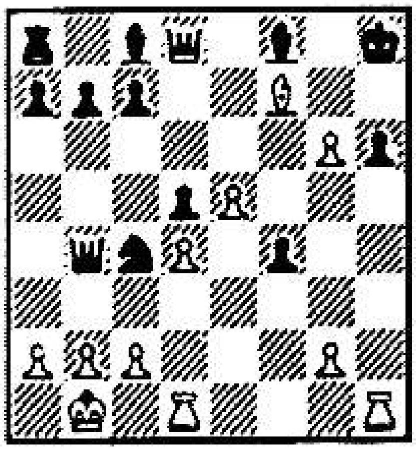
The re-search has returned an improved offensive value to ply 8, so the system tries to find better defenses for black.



```
PLY 9  (VALUE  -55)
(STATIC-ANALYSIS 17.7 SECONDS) (7 PLANS)
BESTPLAN: (( (WR H6) ((NIL (SAFECAPTURE WP BN)) XNIL (KILL WQ BN) )))
(NEWEXPECT   185)
NEWBOARD:  (H1 WR H6)
(EXIT OFFENSE (VALUE 1300 . 1300))

(PLY 8) REFUTE: ( (WR H6))
(PLY: 8) DEFMOVES: ()
```

```
(RETURN RESEARCH (D1 WR H1) PLY 7 NEW-VALUE (1300 . 1300))
PLY 7  BESTPLAN: ( (WB E6) (( (NIL E6)  (WQ E8) NIL (CHECKMOVE WQ E8) )
  (NIL (SAFECAPTURE WB B8) )( (NIL E6) (SAFEMOVE WQ E8) NIL (CHECKMOVE WQ E8) )))
(NEWEXPECT  185)
 TERMINATE: PLAN SUCCEEDED
(EXIT OFFENSE (VALUE 1300 . 1300))

(PLY 6) REFUTE: ( (WR H6) (WR H1))
(PLY: 6) DEFMOVES: ()
(PLY: 4) DEFMOVES: ()
PLY 3  BESTPLAN: ( (WQ F8) NIL (SAFECAPTURE WB BP) )
(NEWEXPECT  5)
 TERMINATE: FORWARD PRUNE

(RETURN RESEARCH (B4 WQ F8) PLY 2 OLD-VALUE (-5 . 185) NEW-VALUE (1300 . 1300))
(PLY 2) REFUTE: ( (WQ F6) (WR H6))
(PLY: 2) DEFMOVES: ( (BB F8))
NEWBOARD:  (G7 BB F8)
```

The defense looks for better moves at ply 2



```
TRY OBVIOUSLY WINNING MOVE
PLY 3   (VALUE  -5)  BESTPLAN: ( (WR H6) ((NIL (SAFECAPTURE WQ BP) )
   (NIL (SAFECAPTURE WQ BP))  (NIL (SAFECAPTURE WB BP))) )
(NEWEXPECT  15)
NEWBOARD:  (H1 WR H6)

(PLY: 4) DEFMOVES: ( (BK G7))
NEWBOARD:  (H8 BK G7)

TRY QUIESCENCE SEARCH
(PLY: 5) NEWBOARD:  (H6 WR H7)
(EXIT OFFENSE (VALUE 1300 . 1300))
```

There cannot be a static analysis after an obviously winning move so a quiescence analysis is tried immediately since the current plan is not executable.

```
(PLY: 4) DEFMOVES: ()
(PLY 2) REFUTE: ( (WR H7) (WR H6))
(PLY: 2) DEFMOVES: ()
(RETURN RESEARCH (H4 WQ D8) PLY 1 NEW-VALUE (1300 . 1300))
```

```
BEST MOVE: (H4 WQ D8)   VALUE (1300 . 1300)
PRINCIPAL VARIATION:
1 (H4 WQ D8)      (B4 BQ F8)
2 (H1 WR H6)      (G7 BB H6)
3 (D8 WQ F6)      (F8 BQ G7)
4 (D1 WR H1)      (G7 BQ F6)
5 (E5 WP F6)      ANY
6 (H1 WR H6)

TOTAL TIME: 450 SECONDS
NODES CREATED: 47   (35 REGULAR, 12 QUIESCENCE)
STATIC ANALYSES: 8

WHERE TIME WAS SPENT:
41%    calculating primitives
23%    static analysis
18%    quiescence searching (overlaps primitive calculations for 12 nodes)
7%     defense determining initial move
6%     disk IO
3%     offensive plan elaboration
1%     defensive counter-causal analysis
0%     causality facility, evaluation function, creating board positions
```

This example shows how the best-first search strategy controls the growth of the tree. The game tree grows exponentially and the search can get completely (and forever) lost by just a slight misperception of what is going on or by not being aware of better alternatives. In the above problem, PARADISE uses its best-first strategy to terminate the search of a plan any time something has gone slightly wrong making another alternative look more promising. This is very important for obtaining a knowledge-controlled search.

It is reasonable to terminate searches as quickly as PARADISE does because the mechanism for re-searching lines is efficient. Practically the only investment of effort in switching lines is doing disk input and output of positions and their patterns. Because of its re-searching, this problem used double the disk IO resources of the problem shown in section A of this chapter (6% to 3% of the total cpu time), but little other effort was involved. No analysis is repeated.

Many things pointed out in the example in section A are also illustrated in this example: the way each cutoff in the search handles ranges properly, the sensitivity of PARADISE's analysis, and the ability of the plans to accurately guide the search and avoid static analyses. This combination is too deep to be found by other chess programs; CAPS, TECH, TECH2, and CHESS 4.4 are not able to find the mate.

CHAPTER VI

Analyzing the Program's Performance

A) Analyzing performance on the test positions

1) Introduction

The techniques used by PARADISE to represent and reason with knowledge have been described. To evaluate these techniques, it is necessary to analyze the program's performance along many dimensions in an attempt to answer the following questions: Does PARADISE solve problems and analyze situations with great expertise? How general is the program's expertise (i.e., how large is its domain)? How easy is it to add or modify knowledge? How does added knowledge affect performance on positions other than the ones for which the knowledge was added? How is system performance affected when large amounts of knowledge are added? What kind of errors does the system make? Where does the program invest its resources (i.e., what kind of structure is imposed by this approach)?

This section examines the results of tests that were performed in an attempt to answer these questions. Section B summarizes the results, and section C compares PARADISE to other chess programs with respect to these questions.

The tests that have been performed involve the first 100 positions in [Reinfeld58] which contains tactically sharp positions from master games. This selection of problems is representative of tactical problems of reasonable difficulty. Although intended to instruct humans, these problems have been used to test other chess programs (e.g., CAPS, TECH, and CHESS 4.4). Thus they provide a good domain for comparing the performance of PARADISE to that of other programs.

As described in chapter 1, a developmental set of fifteen positions was selected from among the first 100. The knowledge base of PARADISE has been developed by writing productions which do these fifteen positions in a reasonable manner. The 85 positions not in the developmental set were not considered during this development. This development process produced one version of the program, called PARADISE-0. PARADISE-0 was then tested on a test set of six positions picked at random from the remaining 85. If the program could not solve a problem in the test set, the necessary knowledge and/or search control mechanisms were added so the problem could be solved. The version of the program which solves both the developmental and test sets is called PARADISE. PARADISE was then tested on all middle game positions in the first 100. The results of this development and testing are presented in the remainder of this section.

2) Developmental and test sets

The developmental set consists of positions 1, 9, 13, 27, 32, 42, 49, 51, 52, 55, 68, 76, 78, 82, and 90 which were selected for the variety of concepts required to solve them. The test set consists of positions 11, 46, 50, 67, 84, and 93 which were chosen at random. PARADISE-0's performance on the test set is described in the next paragraph.

PARADISE-0 solved problems 11 and 50 without change. Problems 46 and 84 required small modifications to the knowledge base. Problem 46 required a more accurate threat language expression for assessing the threat of a discovered attack so that the full potential of such an attack would be realized. (PARADISE's analysis is more accurate on this problem than is Reinfeld's: he claims white can win the exchange while PARADISE's analysis shows that white can win only a pawn). Problem 84 required the knowledge base to realize that a mobil mating move might have to be made safe since the opponent might be able to exchange the mating piece. Problem 67 required a rather large addition to the knowledge base in the form of one production which looks for ways to use a decoy to set up a discovered attack. To solve problem 93 reasonably, it was necessary to add 6 productions to the knowledge base (in 5 different KSes) and to program more knowledge into both the causality facility and the quiescence search. Thus problem 93 inspired the only program changes between PARADISE-0 and PARADISE.

It is hard to draw conclusions from such a small test set, but some things are obvious. PARADISE-0 already had most of the necessary knowledge to solve four of the six problems. Thus the program has some degree of generality and does not tend toward the extreme of having one specific production for each position it solves. Problems 46, 84, and 67 were solved after modifying the knowledge base without making any other changes. The modifications for 46 and 84 took less than 5 minutes each to discover and implement, while the production added for problem 67 took less than 25 minutes to design and implement. This indicates that the knowledge base is easily modifiable.

3) PARADISE's performance on the first 100 problems

To better test the system's generality, PARADISE was then tested on all of the first 100 problems. (The 21 problems in the developmental and test sets had not all been solved by the same version of the program.) Eight problems (2, 18, 33, 41, 43, 86, 87, and 100) were eliminated from consideration since they are not in PARADISE's domain. These eight problems require queening pawns and/or knowledge about the ability of passed pawns to queen. End-game problems that can be solved using middle-game tactical ideas are considered to be in PARADISE's domain.

PARADISE's performance on a problem is classified into one of three categories: I) problem solved as is (possibly with a minor bug fix), II) problem solvable with a small addition to the knowledge base, or III) problem not solvable without a change to the program or a major change to the knowledge base. A problem is not considered solved because the correct move is selected. A solved problem means the correct move is selected for the correct reason (i.e., all lines were analyzed adequately), and that all reasonable defenses have been tried and refuted. Category II helps measure the modifiability of the knowledge base. It is meant to include solutions which require no changes whatsoever to the program and only a small addition to the knowledge base. Small means that it takes less than 20 minutes total of human time to identify the problem and write or modify one production which will enable PARADISE, with no other changes, to solve the problem in a reasonable way (i.e., no ad hoc solutions).

Since the developmental and test sets had not all been solved by one version of the program, these were tried first. Adding new knowledge during program development would (hopefully) not adversely affect performance on earlier problems, but this had to be confirmed. For example, productions added to the knowledge base to solve the last 5 problems in the developmental set might produce so many suggestions in the first 5 problems that the search would become untractable in those problems. The testing proved that the new knowledge had no adverse effects. All 21 problems in the developmental and test sets fell into category I. Thus the same version of the program solves every problem in the two sets. In every case, the analysis is either the same as or sharper than that produced by developmental versions of the program. This result provides strong evidence that the knowledge base is easily modifiable. If productions are written carefully and intelligently (a skill the author developed while writing productions for the developmental set), they do not appear to adversely affect the system's performance on positions unrelated to the new productions. This is an essential quality for a modifiable knowledge base.

Reinfeld divides the first 100 problems into five groups of 20, and claims an increase in difficulty with an increase in group number. Figure 6.1 shows the program's performance for each of the five groups. In this table, the row for PARADISE gives credit only for the problems in category I. PARADISE would become PARADISE-2 by leaving in the productions written to solve problems in category II. The row for PARADISE-2 in the table gives the totals for both categories I and II.

	I	II	III	IV	V	All
in domain	18	19	18	20	17	92
PARADISE solved	14 (78%)	13(68%)	17 (94%)	14(70%)	11(65%)	69 (75%)
PARADISE-2 solved	18(100%)	18(95%)	18(100%)	19(95%)	16(94%)	89 (97%)
Average nodes (P2 solved)	19.8	28.7	35.0	58.4	48.4	38.1
Average cpu time (P2 solved)	145.8	267.1	292.6	532.4	425.5	332.5
Average number static analyses	1.7	3.2	3.1	5.7	5.1	3.7

Performance of PARADISE on first 100 problems

Figure 6.1

In figure 6.1 and throughout this chapter, cpu times are given in seconds on a DEC KL-10 processor. PARADISE solves 75% of the first 92 problems on its first attempt (i.e., without any added knowledge). Of these 69 solved problems, 9 required minor bug fixes such as misspellings, using the LEGALMOVE pattern when DIR was obviously meant, and bugs which manifested themselves when black was on offense (white was on offense in all developmental and test positions except one). PARADISE does well on Group III because 7 of those 20 problems were in the developmental and test sets. Similar tactics are involved in many of the Group III problems so PARADISE already had most of the necessary knowledge.

PARADISE-2 solves 89 of the 92 problems. (The three failures are discussed in the next part of this section.) This indicates that these problems do not push the limits of the expressibility of the production language nor the ability of the tree search to discover and communicate information. For the 20 problems solved by PARADISE-2 but not by PARADISE, only 13 productions were written. In two problems the added knowledge involved modifying a production already present in the knowledge base. Five problems were solved by using one of the 13 productions that had already been written for another problem. This result speaks well for the extendability of the knowledge base. To avoid the extreme of having a production for every position, each production added to the knowledge base should significantly increase the set of positions the program can solve. With a small sample of 20 problems, the fact that 13 productions bring 18 of the 20 problems into the program's domain is encouraging.

These results indicate that the generality of PARADISE is reasonable and that the modifiability of the knowledge base is excellent. The generality of the productions being written appears adequate. Although the distinction between a new piece of knowledge and the tuning of an old piece of knowledge is fuzzy, it is useful to note that these 20 positions were added to PARADISE's domain by writing 13 new pieces of knowledge and tuning 2 already existing pieces of knowledge.

The last 3 rows in figure 6.1 substantiate many of the claims made about the program. The number of nodes generated (row 4 in figure 6.1) is small. When this research was undertaken, a tree size of *32 + 2*d*d (where d is the depth)*, was established as a goal. Depth is here defined as the depth of the deepest non-capture in any branch of the principal variation. (See part 5 of this section for a discussion of this definition.) The number 32 was chosen arbitrarily to give a fixed number of nodes (without regard to depth) for the program to do quiescence analyses since captures at the end of a variation are not counted in the determination of depth. It also makes a reasonable goal for problems with depths less than 4. This goal was achieved for the most part: of the 89 solutions produced by PARADISE-2, 87% of the trees met this goal (counting both regular and quiescence nodes).

The program's tree size seems to be of the same order of magnitude as that of a human grandmaster (see [DeGroot65]). Since the program adequately analyzes all relevant lines when it solves a problem, this tree size indicates a high degree of expertise on problems the program has the knowledge to understand (given enough cpu time).

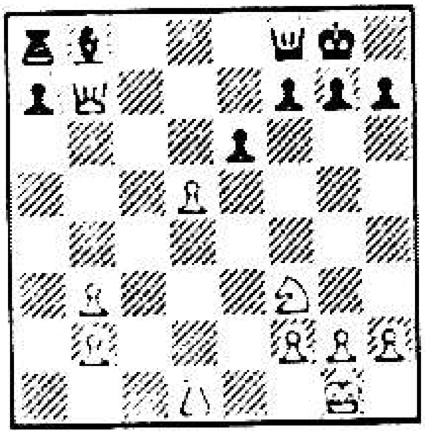
The small number of static analyses per search (row 6 in figure 6.1) show that the plans do an excellent job of guiding the search. The statistics on cpu time show that, in this program, using knowledge is expensive. The program could be speeded up considerably and the issues involved are discussed in chapter 7.

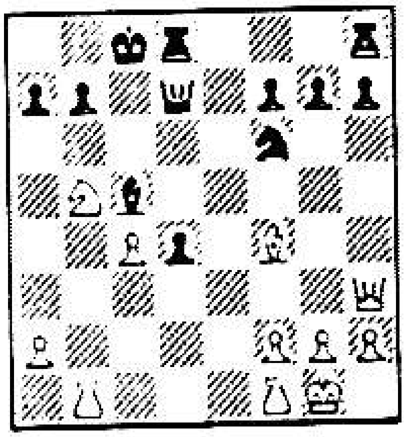
4) How PARADISE goes wrong

PARADISE was not able to correctly analyze 23 of the 92 problems and 3 of these could not be solved after small additions to the knowledge base. A failure by PARADISE can be classified in one of the three following categories: 1) The best plan is never suggested. PARADISE tries every plan that has been suggested and then quits without having achieved an acceptable result since it has run out of ideas. This can be cured by filling a hole in the program's knowledge so that the best plan will be suggested. 2) The search becomes unbounded. If a position is rich in possible attacks and defenses, with most combinations running many ply before a quiescent position is reached, then PARADISE may use an unreasonable amount of time searching. During testing, PARADISE was limited to 45 minutes of cpu time per problem. (This limit was exceeded only once: see below.)

This may be cured by having more specific knowledge to provide sharper analysis, by having better tree control mechanisms, and/or by having better quiescence determiners. 3) A mistake is made in the analysis. For example, PARADISE may not solve a problem (even though recommending the best move) because the causality facility has eliminated a good defensive move which should have been searched, or because the quiescence analysis has made a large error (small errors must be tolerated).

Of the 20 problems solved by PARADISE-2 but not by PARADISE, 19 fell into category 1 initially. One fell into category 3 because of a mistaken quiescence analysis, but a production added to the THREAT and QUIESCENCE KSes fixed the problem. There were no failures caused by the search becoming unbounded. The 3 problems which PARADISE-2 could not solve are shown in figure 6.2. They are examined in detail below.


problem 31
white moves


problem 71
white moves

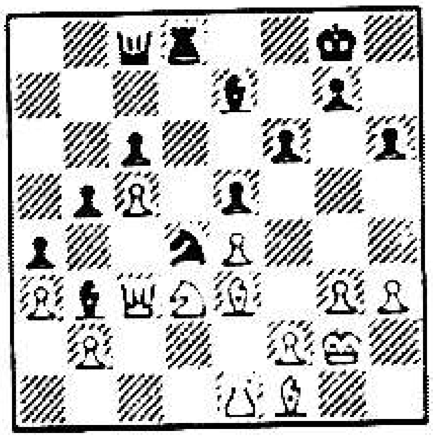

problem 91
black moves

Figure 6.2
Problems not solved by PARADISE-2

In problem 31, white plays P-Q6 and wins the black bishop which must move to save the black rook. If QxR is played immediately, black replies BxPch and wins white's queen. (White still has a won position after this, but it is an involved pawn ending which goes well beyond the depth to which any current program can reasonably search.) PARADISE-2 could easily solve this problem by including a production which tries to block a checking move by a piece blocking a support of an en prise piece. Such a production could be expressed in only a few lines in the production-language. However, this is unacceptable because it is an ad hoc solution. The "correct" way to find P-Q6 is by a counter-causal analysis of the refutation of QxR. This move is suggested by PARADISE's counter-causal analysis but is not searched because offensive counter-causal moves are tried only if they are captures (see chapter 4). Changing this restriction would involve a program change, so PARADISE-2 cannot solve this problem. This problem would be classified as a category 3 failure.

Problem 71 is a category 1 failure. The initial static analysis does not suggest NxRPch which wins for white by enabling the white queen to move to QR3 and then up the queen rook file. The static analysis has the goal of moving the white queen to QR3 and then up the queen rook file, so a fairly simple production in the MOVE KS suggests NxRP at the top level. With this new production PARADISE-2 at first got the correct answer, but a bug in one of the productions in the DEFENDMOVE KS prevents some good defensive moves from being tried at ply 4. When this defensive production is fixed, the search becomes unbounded (i.e., uses more than 45 minutes of cpu time) because of the many defensive and offensive possibilities. This is the only example of a category 2 failure during testing of the program (with the exception of a bug in the repetition of position check). There were many category 2 failures during development of the program when poorly written productions suggested too many plans.

Position 91 is also a category 1 failure. Black can win a pawn and a much superior position by playing B-K3. Black threatens mate with his queen and knight and white must play BxN to avoid it, allowing PxB, 2. Q-any PxPch winning a pawn. Whenever PARADISE suggests mating sequences, it first expends a large amount of effort doing a sensitive analysis of the upcoming possibilities. Problem 91 must be understood on a similar level of specificity, or the advantages of this knowledge would be lost and the search might become unbounded (see chapter 7). However, such a specific production would require more than 30 minutes of human effort to design and implement so PARADISE-2 was not given credit for this problem.

6) Comparing performance to that of other chess players

Additional insights are gained into the ability and structure of PARADISE by comparing its performance to that of other chess players, both computer and human. This comparison consists of analyzing performance on the 92 positions of the first 100 which are in PARADISE's domain. Figure 6.3 compares the percentage of these problems that were solved by PARADISE, PARADISE-2, CHESS 4.4 (running of a CDC-6400), TECH, CAPS, and a human player with a class "A" rating. In this comparison, PARADISE and PARADISE-2 were limited to 45 minutes of cpu time while the others were limited to 5 minutes (cpu time for the programs and real time for the human). This is due to the availability of statistics, but is not as unfair as it seems. As figure 6.1 shows, PARADISE-2 uses an average of only 5.5 minutes of cpu time on the 89 problems it solves. Also, PARADISE could probably be speeded up by a factor of 2 or more with more efficient production matching. (The system spends more than 91% of its time matching productions.)

	PARADISE	PARADISE-2	CAPS	TECH	CHESS 4.4	CLASS A HUMAN
Group I	78%	100%	67%	78%	94%	89%
Group II	68%	95%	74%	84%	95%	95%
Group III	94%	100%	61%	61%	78%	94%
Group IV	70%	95%	50%	40%	70%	80%
Group V	65%	94%	41%	47%	76%	53%
All 92	75%	97%	55%	61%	83%	83%

Percentage of problems solved by various chess players
Figure 6.3

The CAPS and TECH programs seem distinctly weaker than the others over this test set. They are outperformed in every group except on group II by PARADISE. CHESS 4.4 and the human seem equal, with PARADISE somewhat weaker. PARADISE finds groups I and II harder than 4.4 and the human do, but it outperforms 4.4 on group III and the human on group V. This indicates that the generality of PARADISE over this domain is comparable to that of other good chess playing programs. PARADISE solves many problems besides those in the test and developmental sets, providing strong evidence that there is not one specially tuned piece of knowledge for each problem solved.

PARADISE-2 appears stronger than the other players over this domain. PARADISE can easily be extended to outperform both CHESS 4.4 and this particular class "A" human. This indicates that the goal of easy modifiability has been attained. It appears reasonable to develop the program's expertise incrementally. No immediate limits to such a development can be recognized in this domain of tactically sharp positions (see the discussion in the next section). Increasing the expertise of any of the other programs in figure 6.3 this dramatically would require a major development effort or a significantly faster computer.

Judging from the resource investment of PARADISE and the performance of other chess players, groups IV and V are noticeably more difficult than the first three groups. It is interesting that both PARADISE and PARADISE-2 do roughly as well on these two groups as they do on groups I and II. This is not true for the other programs or for the human. Since TECH and CHESS 4.4 (both of which rely primarily on search) are weak on the latter groups, it appears that the problems in the latter groups are harder because their combinations are deeper. These programs generally solve problems in which no branch of the principal variation is deeper than their depth limit (unless their quiescence search goes deeper on the branch in question). They fail on problems which exceed their depth limit. This is brought out by figure 6.4.

DEPTH	NUMBER	PARADISE	PARADISE-2	CAPS	TECH	CHESS 4.4	CLASS A HUMAN
1-2	11	73%	100%	55%	100%	100%	100%
3	18	89%	100%	78%	89%	100%	94%
4	29	66%	100%	69%	93%	100%	79%
5	9	78%	89%	33%	22%	89%	78%
6	12	75%	100%	58%	0%	58%	67%
7	4	100%	100%	25%	0%	50%	100%
8-9	5	80%	100%	60%	0%	20%	100%
10-12	4	50%	50%	0%	0%	0%	25%

Percentage of first 92 problems solved as a function of depth
Figure 6.4

Figure 6.4 shows the performance of the players in figure 6.3 on these 92 problems as a function of depth. Depth is here defined as the depth of the deepest non-capture in any branch of the principal variation. (This is the same definition used in [Berliner74].) This definition of depth fits well with programs like TECH and CHESS 4.4 since their quiescence analyses will play captures past their depth limit. Capturing sequences at the end of a variation will often be self-evident to a good human player also.

Figure 6.4 shows that TECH and CHESS 4.4 usually miss problems only when they cannot look deep enough. TECH solves almost all problems of depth 4 or less, only 2 problems of depth 5, and none deeper than 5. CHESS 4.4 solves almost all problems of depth 5 or less, but its performance deteriorates with increasing depth after that. The program's quiescence analysis is sophisticated enough to solve some deeper problems. The data available was not always sufficient to determine if CHESS 4.4 solved problems for the correct reason, so the program's performance may be weaker than figure 6.4 indicates.

It is fairly easy to predict which problems TECH and CHESS 4.4 can solve, given some knowledge of their quiescence analysis. This is not true of the other players in figure 6.4, all of whom use a more knowledge-based approach. PARADISE relies almost completely on its knowledge and fails whenever an important piece of knowledge is missing. To the extent the program has knowledge about both short and long range tactics, it would be as likely to fail on shallow problems as on deep ones. Figure 6.4 shows that PARADISE is more successful on the problems of depth 7 to 12 than it is on the problems of depth 1 to 4. This is partially due to the fact that a higher percentage of deep problems are in the developmental and test sets. Nevertheless, this data indicates that PARADISE's knowledge of tactics as long range as those used in groups IV and V seems to be nearly as complete as its knowledge of shorter tactics, which explains PARADISE's uniformity of performance on groups I, II, IV, and V.

CAPS appears to be somewhat weaker on deep problems than on shallow ones, but not as markedly as the search-oriented programs. The problems of depth 10 to 12 are beyond the depth limit of CAPS. The human's performance is not markedly weaker on the deeper problems. His poor performance on group V therefore stems from shallow problems as well as deep ones and is probably due to Reinfeld's ability to guess the knowledge which might be missing in a human player. PARADISE probably performs well on group V because it did not learn chess the way a human does. It was taught by the developmental set (selected from all groups), and would not necessarily be missing the same knowledge as a human (even though a human developed the knowledge base).

The eight problems not in PARADISE's domain are not considered in the above statistics. It is interesting to look at the performance of the other programs and the human on these eight positions (since they all play end-games). CAPS, TECH, and CHESS 4.4 can solve only 1 of these 8 problems (the same one for each program). Therefore, relative to these problems, these programs do not have a significantly larger domain than PARADISE. The human player gets 6 of these 8 problems correct, showing that he is a more general problem solver.

Figure 6.5 compares the average tree size (in number of nodes) for PARADISE, CAPS and CHESS 4.4. PARADISE attempts to use knowledge whenever possible in order to produce trees of the same order of magnitude as those produced by human masters. Figure 6.5 shows that this has been accomplished for the most part. CAPS uses a lot of knowledge but invests only about one-fifth of a second per node calculating. CAPS relies on the search to correct mistakes made by inadequate knowledge and this results in trees one order of magnitude larger than those produced by PARADISE. CHESS 4.4 has little chess knowledge and relies almost entirely on search to discriminate between moves. It generates trees that are three orders of magnitude larger than those generated by PARADISE. The two knowledge-oriented programs (especially PARADISE) grow larger trees for the deeper combinations (groups IV and V), just as most humans would. CHESS 4.4's tree size seems unrelated to the depth of the combination. CHESS 4.4 is of course the best chess player of these 3 programs.

	PARADISE-2	CAPS	CHESS 4.4
Group I	15.6	167.8	24,307.3
Group II	28.7	226.4	34,726.0
Group III	35.0	206.1	24,200.1
Group IV	56.4	453.6	31,538.1
Group V	48.4	285.3	25,917.5
All 100	38.1	266.6	28,496.8

Comparison of average tree size on first 100 problems (for solved problems)
Figure 6.5

6) Where PARADISE invests resources

Figure 6.6 shows where PARADISE invests its computational resources. All numbers are averages over the 89 problems solved by PARADISE-2. A majority of the cpu time is spent calculating the patterns in the PRIMITIVE KS. The majority of this time is spent calculating the occupiabilities of squares. Although one static analysis uses more than 12 seconds of cpu time (this does not include primitive calculation), the system spends only a small fraction of its time doing static analyses. This is because plans guide the search so accurately that a static analysis is rarely needed.

The system spends more than a quarter of its time quiescence searching, but this includes the time spent calculating primitives for nodes generated in the quiescence search. Only a fraction of the program's time (excluding primitive calculation) is invested in picking defensive moves and the time spent elaborating offensive plans is minimal. A significant amount of time is spent doing disk IO because of the implementation of the best-first search strategy. The causality facility, the evaluation function, and the routine which creates new board positions do not use a significant amount of computation. The latter two are major users of cpu time in search-oriented chess programs.

Nearly all the cpu time is spent executing productions in the knowledge base. This includes primitive calculation, static analyses, plan elaboration, knowledge base accesses made in determining defensive moves, quiescence searches, and counter-causal analysis. The time spent interpreting productions completely dwarfs any other investment of resources (e.g., employing the best-first strategy, generating the tree, using the causality facility, etc.). Since the process of matching productions is inefficient, this hints that a large gain in speed can be obtained. The only computationally significant use of resources outside of matching productions is disk IO.

	Mean	Highest	Lowest	Standard Deviation
NODES:				
CPU time in seconds for whole problem:	332 5	1958.	19.	396 6
Nodes created during search:	38.06	215.	3.	41 9
Regular nodes created during search:	25.12	147.	2	27.07
Quiescence nodes created during search:	12.53	103.	0	19 64
CPU time per node in seconds:	8.53	14 4	3 37	2.17
PRIMITIVES, KNOWLEDGE:				
CPU time per node calculating primitives	4 46	6 34	1 15	1 16
Percentage of CPU time calculating primitives:	52 65	74 3	34.07	8 47
CPU time spent executing knowledge base:	304 4	1760 6	17 05	359 2
Percentage of CPU time executing knowledge base:	91 6	97.7	81.48	2 52
STATIC ANALYSIS:				
Number of static analyses:	3.73	35	0	5 96
Percentage of CPU time spent doing static analysis	11 61	33.7	0 0	10 04
CPU time per static analysis in seconds:	12 22	26.5	2.18	5 14
Percentage of regular nodes requiring static analysis:	12.61	50.0	0.0	11.27
Percentage of all nodes requiring static analysis:	8.95	33.3	0 0	8 07
QUIESCENCE:				
Number of quiescence searches:	6 98	46	0.	8 64
Percentage of CPU time quiescence searching:	25.56	86.0	0 0	18 53
CPU time per quiescence search in seconds	16 05	157.4	0.71	19.95
Percentage of regular nodes requiring quiescence search:	25.55	52.6	0 0	12 00
CPU TIME, MISCELLANEOUS:				
Percentage of CPU time counter-causal analysis:	.29	25.0	0.0	4 30
Percentage of CPU time spent on defensive initial analysis:	9 16	31 1	0 0	5 75
Percentage of CPU time spent elaborating offensive plans:	2 22	11.1	0.0	1 93
Percentage of CPU time doing disk IO:	5.61	11.4	0.46	2.24
Percentage of CPU time in causality facility.	.29	1.8	0.0	.40
Percentage of CPU time in evaluation function:	.09	1.1	0.0	0.15
Percentage of CPU time creating new positions:	.0	0 0	0.0	0 0
Percentage of CPU time garbage collecting:	43.0	56.5	36.5	3.29

Resource investment in PARADISE (88 solved problems)

Figure 6.8

B) Conclusions

Does PARADISE solve problems and analyze situations with great expertise?

When the system is not missing necessary knowledge for a particular problem, the answer is yes. Over the sample of 89 problems, PARADISE is able to accurately analyze all lines while generating an average of only 38 nodes per problem. Human chess grandmasters generate search trees of approximately this size ([DeGroot65]). The system's best first search strategy quickly recovers from mistakes before many nodes are generated. The plans generated by the system do an excellent job of guiding the search. This is evidenced by the fact that a static analysis is done at less than 9% of the nodes that are generated. Since the analysis is well focussed by the plans, PARADISE can add a large number of productions without a large increase in processing time per node. Mechanisms for communicating discoveries from one part of the search tree to another play a significant role in the system's ability to produce small trees.

How general is the program's expertise?

This question cannot be answered definitely from the available data. The performance data indicates that PARADISE can solve many problems in addition to the ones which were used to develop the knowledge base. Thus the knowledge base has some degree of generality. There is not one specific piece of knowledge for each problem. The comparison with other programs indicates that PARADISE is as general as other good chess programs on the type of problems which appear in [Reinfeld58].

The program cannot solve problems unless they involve the type of tactics it understands. It knows nothing about other aspects of chess, e.g., openings or end-games. While [Reinfeld58] seems fairly representative of middle game tactics, it is possible that many tactical situations in actual play could be understood by programs like CHESS 4.4 but not by PARADISE. On the other hand, PARADISE understands problems which are too deep for any search-oriented program.

Is the knowledge base easy to modify?

Definitely yes. With little human effort PARADISE was able to understand 20 of the 23 positions on which it initially failed. The answers to the following questions further substantiate this claim.

How does added knowledge affect performance on positions other than the ones for which the knowledge was added?

The system's modifiability would be questionable if new knowledge hurt system performance on other problems. Fortunately, this is not the case. This is substantiated by the fact that all 21 positions in the developmental and test sets were solved by PARADISE after the knowledge base had been developed incrementally by solving each of the 21 positions with a different version of the program. This is also fairly intuitive to the person writing the productions. The knowledge in PARADISE is specific enough that there is usually little question that a new well-written production will improve performance. If the new production matches in other positions, it will probably improve performance, not weaken it. It is possible to write a production which will make poor suggestions in other problems, but experience has shown that a resulting correction of the new production eliminates the problem.

The limiting factor in this approach is the need for specificity. For example, a production cannot suggest a move just because it is a check. Such a production would weaken performance on many problems. On the other hand, the winning move may have only subtle features suggesting it, other than the fact that it checks. In PARADISE it is necessary to write a specific production recognizing these subtle features since the search control mechanisms are not powerful enough to terminate if every check is searched. In some problems this required specificity may be prohibitive. For example, PARADISE could not solve problem 91 for just this reason. (Problem 91 could be solved by investing enough human effort to write the specific production.) This issue is discussed further in chapter 7.

How does added knowledge affect resources required?

No ultimate limit can be placed on the cpu time required to execute a production since a poorly written production may take arbitrarily long to execute. In PARADISE, only the productions in the THREAT KS take a significant amount of time to execute. They may take as long as a second or two (in the worst observed case), while executing most other productions requires only a few milliseconds (many of their variables are already instantiated by the goal). Thus adding most productions will not significantly increase the processing time at each node. Productions in the THREAT KS are executed only during a static analysis (this is true for many other productions also), so the cost of executing them is incurred infrequently. (PARADISE does a static analysis on only 9% of the nodes produced.) A well-written production has the most restrictive parts of the pattern first, so it quickly fails when it does not match. Thus a given production requires significant resources to execute only when it actually matches. A typical production only matches in a small minority of positions. Thus even a large production in the THREAT KS significantly increases execution time only on problems where it matches, and then only by increasing the cost of an infrequent static analysis by a second or two at most.

More productions also require more space. About 25,000 of PARADISE's 200,000 words (on a DEC KL-10) are occupied by the knowledge base. Adding new productions can increase this, but not enough to harm performance when large amounts of core memory are available.

What are the limits of knowledge base extendability?

No such limits were found during testing of the first 100 positions in [Reinfeld58], and no immediate limits are apparent. The time required to match productions does not appear to be a limiting factor. The need for specificity of knowledge has already been mentioned as a limiting factor. Probably the most immediate limiting factor is the ability (or lack of it) to communicate information in the attribute lists of concepts. The production language does not provide general ways for accessing the information in the data base. The ultimate solution is to use a more general knowledge representation such as KRL ([Winograd77]). The present ad hoc solution is employed because of efficiency considerations. These issues are discussed further in chapter 7.

In what ways does the system fail?

The search may become unbounded when there are many offensive and defensive possibilities in a long combination. However, testing has shown that this almost never happens as long as the productions are specific enough. Another possible error is a mistake in analysis. This may involve missing the right move in a quiescence search, underestimating the threat of a plan, terminating a plan because of a poor causality decision, or any of a number of other things. Testing has shown that this type of error is also rare in PARADISE. As the difficulty of positions increases, the likelihood of these errors may increase, but many of them can be avoided by correcting poor productions in the knowledge base. By far the most prevalent reason for failure in PARADISE is missing knowledge. The system simply gives up on a problem without finding a suitable move. Hopefully the gaps in the knowledge base can be easily filled by writing productions.

Where does the program invest its resources?

Almost exclusively in the processing of knowledge. PARADISE spends 91.6% of its cpu time executing productions in the knowledge base and 5.6% doing disk input and output. The other 3% is used on a number of computationally insignificant tasks including applying the evaluation function, generating new board positions, and constructing the tree. This is radically different from the structure of search-oriented programs which spend most of their time doing the activities mentioned above as computationally insignificant for PARADISE. This means that any speed up in the matching of productions, particularly the productions in the PRIMITIVE KS, will produce a nearly equivalent speed up in the program as a whole.

C) Comparison to other systems

The contributions of PARADISE can be brought out by comparing it to other programs. The program's performance is compared to that of other programs earlier in this chapter. This section compares the structure and features of PARADISE to those of other programs.

PARADISE does many things which no other program does successfully. These will be briefly mentioned here so that the following comparisons can concentrate on areas which PARADISE has in common with the given program. PARADISE is the first program to achieve a knowledge-controlled search in the middle game. The chess programs listed below all have depth limits with the exception of Pitrat's program which allows a static analysis only in the original position, permits only two modifications per plan, and must be told the value of the combination it is to find, thus artificially limiting the effort expended. PARADISE is the first chess program to develop concepts during its reasoning in such a way that the concepts can be used for other purposes (e.g., tree communication) and the reasoning can be explained to the user. PARADISE uses a best-first search which repeatedly initiates searches from the top level until ranges of values have been narrowed enough to show that one move is best. PARADISE's success with this search may be largely due to the fact its domain consists of tactically sharp positions. All the programs listed below use depth-first searches.

The differences between PARADISE and search-oriented programs (e.g., KAISSA, CHESS 4.7, TECH) are described earlier in this chapter and in chapter 7. In chapter 4, PARADISE's causality facility is compared to the one in CAPS and to the method of analogies in KAISSA. In chapter 2, PARADISE's plans are compared to those in Pitrat's program. The comparisons below stress mechanisms within each program that are similar to mechanisms in PARADISE. Comparisons are made with CAPS, Pitrat's program, and TYRO, since they represent the most serious and well-known attempts to use chess knowledge in a program.

CAPS [Berliner74]

CAPS pioneered many of the techniques used in PARADISE. The two programs are compared throughout this dissertation (see especially chapter 4 and section A of chapter 5), so only a summary of these comparisons is presented here. Static analysis in CAPS is based on primitives that are the same as PARADISE's six simplest primitive patterns. Compared to PARADISE's analysis, CAPS's analysis does not investigate ideas as deeply, does not determine as much information about a plan (threats, etc.), and does not invest as much effort making sure an idea is worthwhile. Thus CAPS's analysis is much quicker but more shallow. PARADISE finds some deep threats that CAPS does not. CAPS does not home in on the best move as quickly as PARADISE (as is evidenced by its trees having an order of magnitude more nodes), but it does outperform search-based programs in this respect (on the positions it solves).

The biggest difference between the two programs is the use of plans in PARADISE. CAPS does not produce plans, it simply suggests moves. PARADISE produces more information with its suggestions (each plan has at least THREAT, LIKELY, SAVE, and LOSS attributes) which enable the search to prune many branches. PARADISE uses plans to guide the search and rarely does a static analysis, while CAPS has no corresponding mechanism and analyzes each position it creates.

Another big difference between the programs is PARADISE's best-first search which uses ranges of values and different strategies to prove that a move is best. CAPS has a depth-first search and searches down blind alleys which PARADISE either avoids or quickly terminates. PARADISE saves effort since it does not always have to find the "true" value of a move. As the example in chapter 5 shows, PARADISE can show that a move is best without finding the principal variation.

CAPS has a causality facility similar to the one in PARADISE, but in PARADISE the searching process returns more useful information to the causality facility. In CAPS only bit maps are returned from the search which represent pieces moved, squares moved over and to, pieces attacked, and squares which attacks move over. In PARADISE the whole tree which was generated is returned, where each node in the tree describes the move made and gives some patterns that matched at that node. These patterns describe threats of different kinds, including both direct and discovered attacks. PARADISE also has the ability to compare the influence two different moves have on a line. A more detailed comparison is given in chapter 4.

The combination of all the improvements mentioned here enables PARADISE to achieve a knowledge-controlled search while CAPS has a depth limit of 9. CAPS uses less cpu time and the next chapter discusses some of the tradeoffs which occur when selecting between an elegant but slow approach and a faster but less elegant one. One last but important advantage PARADISE has over CAPS is the ability to easily add new knowledge. Adding new ideas to PARADISE is relatively easy but would be a considerable programming project in CAPS.

Pitrat's program [Pitrat77]

This program, hereafter referred to as PITRAT, is the most important example of the use of plans in chess. Its plans are compared in detail to those of PARADISE in section D of chapter 2. This comparison shows that plans in PARADISE are more accurate at expressing their purpose and guide the search more accurately by making it easier to recognize when a plan is not working.

Like PARADISE, PITRAT deals only with tactically sharp positions, but it recognizes only certain kinds of attacks. PITRAT's analysis finds four types of attacks: captures of a piece (though not one having only a few moves), double attacks of certain kinds, attacks on the opposing king under very restricted conditions (loosely, only attacks beginning with safe checks and attacks where the opposing king has no safe moves), and pawn promotions. PARADISE's analysis recognizes all these types of attacks except pawn promotion, and in addition recognizes many more. For example, PARADISE recognizes more double attacks, captures of men with few moves, and many kinds of attacks on the king that don't begin with check and don't require the king to be trapped. PITRAT would not handle some of the positions in PARADISE's developmental set since they deal with this kind of king attack. It seems that PARADISE handles a broader range of positions although the relative size of the domains is hard to estimate.

PITRAT must be told how much material it should win. This further simplifies its problem by pre-specifying a well-defined goal. This eliminates one of the hardest problems chess programs face since the program does not need to worry about when to be satisfied with a result. When this material expectation is high, the growth of the search tree is strictly controlled since all moves which threaten less than this expectation need not be searched. PARADISE invests considerable effort calculating its own expectation and cannot eliminate any good moves until it has shown a better result can be obtained with another move.

Besides needing to know the expectation, PITRAT has other artificial limits which keep the search tractable without a depth limit. The program allows only two modifications per plan. It does a full static analysis only in the original position (thus limiting the combinations that can be found). A limited analysis on one piece is allowed for correcting a plan at other nodes in the tree (but only twice per plan). Since PITRAT's trees are quite large (thousands of nodes), its static analysis is too expensive (in terms of cpu time) to do often. This also controls tree growth because using the static analysis in the search would produce many more branches (which might require a depth limit to keep the search tractable). PARADISE does a static analysis whenever it decides one is needed, and does not limit the number of modifications that can be made to a plan.

Unfortunately, PITRAT was not tested on the problems in [Reinfeld58] but on a selection of problems it could solve taken from other books. The examples in [Pitrat77] are among the hardest problems that PITRAT can solve. On many of them PARADISE would require large amounts of cpu time and some new knowledge to solve the problem. To facilitate comparison, PARADISE has been run on two of the examples presented in [Pitrat77] which it can solve without new knowledge. Both programs found the correct combination in both examples and statistics on tree size and resource expenditure bring out the different approaches used. The first example is shown in figure 6.7.

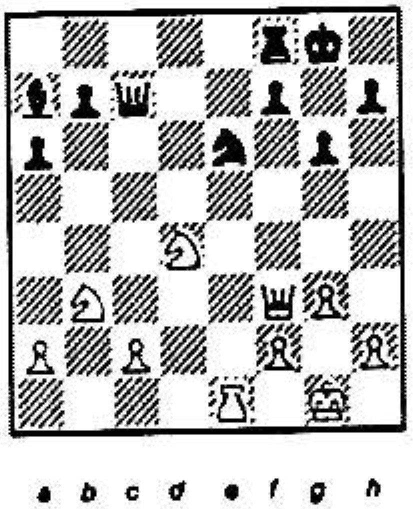


Figure 6.7
white to move

In the above position, white wins the black knight by capturing it with the rook. If bl
recaptures on e6 with his pawn, then white's queen captures the black rook on f8 and after black's king recaptures, white forks the black king and queen by moving his knight to e6. PITRAT's initial analysis notices that if the black king were on f8, then the white knight could fork the black king and queen from e6. This produces the plan of getting the black king onto the square f8 and then moving the white knight to e6. During execution of this plan, the program decides to get the white queen to f8 to lure the black king there and this produces the goal of removing the black pawn on f7 which leads to the discovery of capturing on e6 with the rook. To convince itself that R-e6 wins against all replies, PITRAT uses 71 seconds of cpu time (running FORTRAN on an IBM 370-168) and generates 12,766 plans and 4,800 nodes. The proof tree (the minimal tree which proves the winning combination) contains 790 nodes. The program tries 17 different moves in the original position.

PARADISE's initial analysis of this position notices that white's queen could mate if it were safely placed on f8, and that the white knight can quickly be brought to bear on f8. It therefore suggests the plan of moving the rook to e6 to decoy the black pawn on f7 in hopes of getting a mate with queen and knight. After the rook is recaptured by the black pawn in the tree search, PARADISE realizes the mating plan will not work and does not waste effort investigating it. Instead it does another static analysis which suggests sacrificing the queen to set up the fork. This plan quickly wins. To convince itself that R-e6 wins, PARADISE uses 219 seconds on cpu time (running MAC-LISP on a DEC KL-10) and generates 5 plans and 28 nodes. The program tries 2 different moves in the original position.

PARADISE's tree is much smaller (28 nodes vs. 4,800 nodes) because the program applies more knowledge to the problem. Instances of this are described below. PARADISE searches only a few defensive moves for black while PITRAT tries many. For example, on black's reply to R-e6, PITRAT tries many moves while PARADISE tries only P-e6 and B-d4 since it believes no other black move can possibly regain the knight that has already been captured. This explains why the tree generated by PARADISE is smaller than PITRAT's proof tree.

Although PARADISE's initial plan does not see the eventual fork, it does recognize the possibility of a combination by noticing the attacking lines of the white knight, rook, and queen. The threat of the queen on f8 and the desire to decoy the black pawn on f7 are both recognized. PITRAT's initial plan does notice the eventual fork, but actually employs less knowledge in its recognition. Certainly the white knight can fork the black king and queen if the king is on f8. But a black pawn bears on the white knight's forking square and the program does not know that this problem can be solved. The program also does not know that there is a reasonable chance of forcing the black king to f8. In general, there will be numerous instances of a forking move (safe or unsafe) if an opponent's piece could be forced to a different square. Thus it is not surprising that PITRAT suggests a plethora of plans: 12,756 during the search and 17 in the original position.

PARADISE has enough knowledge to quickly recognize that it's original plan is not working and so avoids wasted effort. As shown in chapter 2, PITRAT cannot recognize failing plans as quickly. PARADISE finds the combination immediately because it quickly rejects the original plan and reanalyzes the situation. Since PITRAT never repeats its initial analysis, it is almost forced to recognize the forking idea in the original position which in turn means that many poor plans will necessarily be suggested. PITRAT cannot quickly recognize that these poor plans are not working and so generates many nodes searching them in depth (sometimes until the two modification per plan limit is exceeded).

The other example on which both programs were run is the 19-ply mate shown in figure 1.1 of chapter 1. It should be noted that PITRAT actually solves an easier problem than PARADISE since it is looking for a mate that is has been told exists, while PARADISE looks for any combination without knowing if one exists or not. PITRAT solves this problem in 69 seconds of cpu time while generating 6,439 nodes and 17,619 plans. The proof tree for the mate contains 117 nodes, and 5 different moves are investigated in the original position. PARADISE solves this problem in 20 minutes of cpu time while generating 109 nodes. The program does 24 static analyses which suggest 122 plans. Four different moves are investigated in the original position. This example reinforces the conclusions drawn from the first example while highlighting the differences in cpu usage. There is obviously a tradeoff here between elegance and resource investment, and chapter 7 discusses this in more detail.

PARADISE can find some combinations which PITRAT cannot because it knows of more kinds of attacks and will do static analyses during the search. On the other hand, PITRAT correctly analyzes positions that PARADISE does not. These positions are usually deep combinations, many times involving more than one sacrifice. The reason for this is that PARADISE spends a lot of effort making sure that plans it suggests have a reasonable chance of working, while PITRAT does not. For deep combinations, PARADISE may not have a production which finds anything that appears to work while PITRAT will suggest a number of ideas, one of which may happen to work. Extending PARADISE to handle these positions would involve writing productions to recognize the possibilities in these combinations. At the level of specificity currently used in PARADISE's knowledge base, adding new productions does not appear to increase the size of the search tree. There are certainly opaque combinations that could not be suggested by reasonable productions at this level of specificity. (Less specific productions may require changes in PARADISE's search to better control tree growth.) These "opaque" combinations represent the domain where PITRAT's approach may produce better results. This limit to PARADISE has not been fully explored, but does not appear to be overly severe since the first 100 positions in [Reinfeld58] do not approach it.

Unlike PARADISE, PITRAT would be hard, if not impossible, to extend. Extending PITRAT to cover the types of king attacks that PARADISE recognizes would involve recognizing more attacks in the original position. PITRAT's search is already quite large and it may be necessary to use the type of knowledge PARADISE does on the king attacks to keep the search tractable. Allowing a full analysis in the tree would probably make PITRAT's search untractable without a depth limit of some sort.

TYRO [Zobrist73]

This program relies heavily on search since it looks at an average of 20,000 nodes for each move it makes ([Levy76]), but it has a "pattern language" for feeding knowledge to the program. The patterns that are matched (in [Zobrist73]) are at the level of PARADISE's simpler primitive functions and patterns. They notice attacking lines which result in possible forks, pins, and defenses, but no higher level concepts are recognized. The actions by which these patterns can affect the system (in [Zobrist73]) are limited. They can suggest a move for searching with a score for ordering it with other suggestions, and they can change a term or coefficient in the evaluation function. Such actions provide little information for controlling tree growth. There are no plans, no concepts to communicate discoveries in the tree, and little information for pruning branches in the tree.

TYRO analyzes its patterns extremely fast, so it can still search many nodes. However, it gets little information from these patterns. CHESS 4.7 for example, probably obtains a better ordering for moves to search simply by doing iterative deepening in the search. A pattern language is also unnecessary as a programming aid for changing terms and coefficients in the evaluation function. For these reasons, other search-based programs (e.g., KAISSA, CHESS 4.7, TECH2) do not use the type of pattern matching used in TYRO.

CHAPTER VII

Conclusions and Issues

A) Contributions

This section summarizes the contributions that have been made by this research. The next section of this chapter discusses weaknesses in PARADISE and directions for future work. The last section describes some of the issues involved in developing a program like PARADISE, and what light this research has shed on those issues.

PARADISE performs expertly on the problems it solves by using knowledge that appears to be easily modifiable. The knowledge base is used by the static analysis to reason with concepts and produce sophisticated plans. The knowledge is also used in the search to produce a knowledge-controlled search. The program currently uses large amounts of cpu time to do its analysis, but it could be speeded up considerably. Furthermore, the cost of PARADISE's evaluation in an initial position can actually be spread over a number of moves in an actual game. If the opponent makes a reply which was considered in the analysis of the previous move, PARADISE does not need to repeat its analysis of the current line. There is no depth limit so searching this line after getting 2 ply further along it will not change the analysis. PARADISE need only look for other possibilities. If more information is needed about the line actually being played, the program can re-start the search from the point it was previously terminated since all pertinent information is retained. No analysis previously done on the line being played need ever be redone.

Because of the resources it uses and its limited domain, PARADISE is far from being a complete, practical chess player. The important contribution of PARADISE is the development of new ways of representing and using chess knowledge during analysis of a chess position. Even if computers master the game of chess without using large amounts of knowledge, the ability to use knowledge and do this type of reasoning about actions is still very desirable. PARADISE is a first step in this direction and the following list summarizes the significant contributions of this research.

-PARADISE has developed a workable set of chess primitives and rules that can support a fairly sophisticated analysis.

-The static analysis reasons with concepts it posts to create plans. This is done entirely by the knowledge base of production rules.

-The knowledge base can be easily extended or modified to increase or make more precise the knowledge available to the system. New knowledge does not appear to adversely affect performance.

-The attribute lists of each concept (and plan) produced by the analysis specify enough information so that the system essentially knows "why" it is doing something. This provides a good explanation of exactly why PARADISE believes something which is helpful for debugging and easy modification. It also allows the system to resolve contradictions and avoid combining antithetical ideas.

-The system understands one set of concepts which are provided by the KSes. The same concepts are used for doing static reasoning, expressing plans, and analyzing information returned from a search (e.g., in a counter causal analysis).

-With its non-linear plans, the program can immediately and directly access the relevant productions when a new position is reached during the search. In this way, the system communicates information from one analysis to many other nodes in the tree.

-The plans express their purpose accurately so the system can quickly determine when a plan is failing.

-The range of plans that can be expressed is not fixed by the program. By forming new KSes, the range of expressible plans may be increased without any new coding to understand the new plans.

-The static analysis provides information for the tree search by giving each plan at least a LOSS, SAVE, LIKELY, and THREAT attribute. This information helps the search terminate branches, order plans, and recognize when a plan is not working.

-PARADISE has a Threat-Language for specifying attributes. Threat-Language expressions are accurate and allow accurate re-evaluation of an attribute at other positions in the search tree.

-The search employs different strategies to show, with a minimum of effort, that one move is best at the top level.

-A best-first search strategy is used which gives the program a global view of the tree and helps produce a knowledge-controlled search.

-Using information obtained from static analysis, plans, and the reluctance function, the program makes reasonable decisions about which node in the tree is the most promising to search.

-The search can efficiently search lines again because pertinent information is kept in core memory.

-PARADISE specifies values of lines and positions by ranges. This allows the system to gain some amount of information about a line without investing all the effort required to determine its exact value. Algorithms which compare two values (e.g., alpha-beta) handle the possibilities which arise when comparing two ranges.

-PARADISE uses a threshold to define its current level of aspiration. The search is temporarily satisfied whenever the threshold is achieved and changes the threshold in the process of using a particular strategy to prove that one move is best.

-Each search backs up with the whole tree that has been generated. This provides information which the causality facility and the counter causal analysis use to analyze problems discovered during a previous search.

-PARADISE avoids searching every possible ordering of available threats. A plan which could have been applied earlier is not applied in the current situation unless it now looks more promising than it did earlier.

-The quiescence search employs specific pattern-based knowledge to consider many different types of attacks and accurately handle a broader range of positions.

-The defense occasionally uses a null move analysis to better understand a position.

B) Future Work

PARADISE has many weaknesses and shortcomings which point to a number of directions for future research. This section briefly describes six areas where PARADISE could be improved or extended if the hard problems involved are studied.

More general communication between concepts and productions.
The KSes and concepts in PARADISE, and the way productions interact with them, are basically ad hoc. These constructs and communication facilities are adequate for doing the type of reasoning PARADISE does. They were not formalized into a more general, domain-independent knowledge representation scheme because of efficiency considerations in the final running program. (It also would have been considerably more work.) Probably the most glaring weakness caused by this in PARADISE is the inability of productions to access all the information in posted concepts. The production-language provides certain ways for accessing the information in concepts. These ways have been adequate for the positions tested so far, but they do not permit all types of potentially desirable access.

The most desirable solution would be to develop a general knowledge representation scheme that is not tailored to PARADISE's domain, but instead provides general communication facilities between concepts, productions, and KSes. Such a representation scheme already exists (at least conceptually) in KRL ([Winograd77]). It appears that much would be learned about KRL and about PARADISE's approach to problem solving by coding PARADISE's knowledge in KRL. PARADISE's KSes, concepts, and productions would all be units in KRL and the knowledge in concepts could be accessed using KRL's communication facilities.

Better understanding of searching lines again.
As described in chapter 4, the strategies for choosing which line to search again have not been well developed. As harder problems are attempted, this shortcoming may hinder performance. Work on PARADISE has defined a number of important criteria to be

considered in choosing a point to initiate a search, but more criteria need to be considered. PARADISE could make better use of the criteria it already considers. Development of better searching strategies could quickly lead to improved program performance on harder problems. These issues are discussed in more detail in chapter 4.

Better tree communication (lemmas).

PARADISE communicates information from one node in the tree to other nodes better than any other chess program. It uses plans to communicate information from a previous static analysis, and it uses information in the tree returned from a search when it employs the causality facility or the counter causal analysis. There is still room for much improvement, especially in the direction of communicating discoveries from an already searched line.

PARADISE uses the tree returned by a search only along the principal variation which led to that tree. It may also be possible to communicate a discovery to some other line that is not part of the principal variation leading to the discovery. The method of analogies in KAISSA does this. PARADISE makes one inadequate attempt in this direction with its lemma facility. A more powerful facility for postulating lemmas from search results in such a way that the lemmas can be used at many points in the tree would be of great help in controlling the search. The problem is that subtle (almost imperceptible) changes can vitally affect the situation a lemma attempts to describe. Better ways of describing lemmas or of deciding when to ignore their advice are needed before they can be useful.

Learning and acquisition of knowledge.

PARADISE has, to a large extent, achieved its design goal of an easily modifiable knowledge base. This may provide a good domain for implementing machine learning. The problem of automatically writing new rules from scratch in PARADISE would be extremely difficult at present. PARADISE's rules access such a diverse set of concepts in such an ad hoc way that a great deal of knowledge about the knowledge base would have to be encoded in any program which was attempting to write rules. The syntax and semantics of PARADISE's production-language are not easily modeled. The domain of chess also rules out some learning techniques. For example, the approach used in Meta-DENDRAL ([Buchanan78]) does not appear feasible in chess. The detailed testing required in Meta-DENDRAL's "plan-generate-test" loop does not seem possible in chess. Chess has no "data points" similar to those used as input in Meta-DENDRAL. One might conceive of a chess position together with the best move in that position as being a data point, but the difference in grain size between such a data point and the patterns checked for by rules in PARADISE is so large as to render Meta-DENDRAL's techniques unworkable. (A chess program must know why a move is best.)

Although the automatic writing of production rules in PARADISE appears to be a difficult, though interesting, problem, the program's structure is well-suited to other types of learning and knowledge acquisition. Developing a program to modify existing rules or aid a human in creating or modifying rules seems possible. TEIRESIAS ([Davis76]) provides knowledge acquisition capabilities for the MYCIN program. Most of the techniques developed in TEIRESIAS can be applied to PARADISE. The explanation system in TEIRESIAS could be applied to PARADISE in a straightforward manner since PARADISE knows the chain of rules which produced each of its concepts, much as MYCIN does. The explanations could be better in PARADISE since most KSes correspond to human concepts. Thus an explanation system in PARADISE could answer "in order to make Q7 safe for the white rook" when asked a question about a production in the SAFE KS, rather than saying "because RULE 047 was used" as TEIRESIAS does. PARADISE's structure would also permit application of the knowledge acquisition techniques developed in TEIRESIAS. This would require some amount of work since PARADISE's use of KSes to structure the knowledge base is not present in MYCIN. In addition to knowledge about rules, concepts, and plans, a knowledge acquisition program in PARADISE would have to be given knowledge about KSes and the ability to deal with them before it could be effective.

Combining PARADISE with other programs.

Since PARADISE operates in a restricted domain, the question of combining it with other programs arises. Many schemes can be imagined. The simplest would be to run PARADISE in parallel with a conventional chess program. The conventional program would play the chess game and only make use of PARADISE's ability when PARADISE found a combination beyond the program's depth limit within the set time limit for that move.

PARADISE might also play a part in some combination of programs which can collectively play chess. Because of PARADISE's assumption about tactical sharpness and offense/defense, its abilities would be used when the side on move is in an aggressive or attacking state. If the position is recognized as being strategical or defensive, the knowledge in PARADISE should not be brought to bear since it would only waste resources. If PARADISE's knowledge base is much improved so that it knows about the vast majority of tactics, and its efficiency of execution and control of the search tree is improved so that it nearly always returns an answer in a reasonable amount of time, then PARADISE could be trusted with the tactical analysis. In this situation, PARADISE could be run in parallel with a program which was designed to know about positional advantages. The latter program would constrain its search by not investigating tactical sequences and would spend all its time analyzing positional aspects of the position. In this case, any winning tactical combination found by PARADISE would be played immediately, but the other program would provide a good strategical move whenever PARADISE fails to find a combination. There are other reasonable ways to combine PARADISE with other programs. Attempts to do so may shed light on subdividing a problem and getting cooperating programs to work well together.

Extending PARADISE's domain.

Extending PARADISE's restricted domain is another worthwhile project. Three directions immediately come to mind: further extending the program's tactical ability to cover most tactics, extending the program to handle strategical situations as well as tactics, and extending PARADISE to play end-games. There do not appear to be any serious problems in further extending PARADISE's tactical ability. The current knowledge base has been built incrementally and no limits have yet been reached. It appears that large numbers of reasonable production rules could be added without significantly impairing performance. (A static analysis was done at only 9% of the nodes in the test domain so new productions are not used that often.) It may be necessary to develop better searching strategies or better search control mechanisms for harder problems.

The problem of getting PARADISE to do strategy as well as tactics is much harder, if it is even reasonable. It would certainly be helpful to recognize the type of patterns PARADISE can express in evaluating a strategical situation. However, the advantages obtained during strategical maneuvering are so slight that many of PARADISE's search control mechanisms would be very dangerous. Many cutoffs which decide that a move is not worth searching or that another move is more promising rely on the tactical sharpness of the domain to make good decisions. In strategical situations it may be necessary to search most moves so as not to miss a good strategical move.

Chess end-games seem well-suited to PARADISE's approach. Patterns would be useful, even necessary, in analyzing end-game positions (see figure 1.6 in chapter 1), although the production language may have to be improved to recognize some end-game patterns (see next section). A strictly controlled search which could investigate the long sequences that often occur in end-games would also be helpful. End-game knowledge is concerned about different things than the tactical knowledge PARADISE currently has, so the program would probably need different primitives to encode this knowledge. (It would also, of course, have to be told about the queening of pawns, stalemate, etc.) It might also be necessary to develop better tree communication facilities (e.g., a good lemma facility), to adequately control deep searches in the end-game. Michie and his group have developed an "advice language" for expressing end-game knowledge ([Michie78]). This has only been used on very simple end-games and PARADISE's ability to reason with dynamically created concepts will probably be needed for more general end-game play.

C) Issues

1) Level of patterns

Chapter 2 defines (though not precisely) type 1, type 2, and type 3 patterns. Type 1 patterns speak of particular pieces and particular squares as in MAPP ([Simon73]). Type 2 patterns are expressed as a relationship between a certain prespecified set of variables. The matching process must then search for instantiations for the variables which will make the relationship true. Type 2 patterns are used in PARADISE and in Zobrist's program. A type 3 pattern is one that cannot be expressed as a type 2 pattern. For example, a type 3 pattern may recurse upon itself or instantiate an indefinite number of variables.

Work on PARADISE indicates that for the most part, type 2 patterns are adequate for expressing the knowledge involved in middle game tactics. As chapter 2 points out, type 1 patterns are obviously not suitable and the only knowledge corresponding to a type 3 pattern in PARADISE is the primitive function SAFEP which is written directly in LISP and not in PARADISE's production language. Useful type 3 features not available in PARADISE's production language and why these features are not needed in PARADISE are discussed below.

Patterns in PARADISE can test conditions over sets which have an indefinite number of possible members by use of the "IMPLIES" expression. By using IMPLIES, a pattern can, for example, find a good continuation for every possible king move the opponent may use to reply to a checking attack, no matter how many such king moves there are. However, the instantiations of each good continuation disappear as the next is matched. When the IMPLIES expression matches, indicating that the checking attack is worthwhile, the individual continuations are no longer around to use in the final plan. A more complex way of specifying type 3 patterns could allow an indefinite number of such continuations to be kept and used later in the final plan.

When accessing a set generated by an IMPLIES expression, it is not possible to test the relationship of one member of the set to another. This might be necessary if a pattern is to recognize a pawn blockade of indefinite length since pawns in one blockade must have adjacency relationships with each other. Such a blockade pattern would be needed to correctly analyze the end-game position shown in figure 1.5 in chapter 1. A method for specifying type 3 patterns could allow sets to be defined on the basis of relationships between members of the set.

PARADISE has not suffered from the inability of its patterns to handle the above situations. This is mostly due to the choice of domain. PARADISE does not need to recognize continuous pawn blockades in its domain, and in general it is rare in PARADISE's domain that sets with indefinite number of members need to be recognized as a unit. In the end-game this would not be true since type 3 patterns may be needed to recognize pawn blockades and other things. The problem of not having instantiations from each member of a set produced by evaluating an IMPLIES expression has also not hurt PARADISE. In the above example of a checking attack, the continuation in the final plan will specify that the opponent's king should be attacked. This summarizes all the particular attacks found during the IMPLIES and guides the search so that a static analysis need not be done. It does waste some effort by not using the analysis done by the IMPLIES expression, but it is often a better plan since the IMPLIES only looks for attacking possibilities and does not always find a detailed attacking plan. (If the production did look for detailed plans for every continuation it would be too specific, a consideration discussed in the next section of this chapter.)

Two of the primary features which distinguish a "pattern" in a "pattern language" from a program in some programming language are modifiability and transparency. The production language in PARADISE has been designed to be easy to modify and extend, and hopefully to be understandable by a chess expert with little knowledge about computer programming. Transparency and modifiability are major problems in specifying type 3 patterns. To compute SAFEP (PARADISE's one need for a type 3 pattern), it is necessary to take into account all pieces which affect the square in question. If some of these pieces are pinned, the analysis must determine the strengths of these pins which may require a recursion of SAFEP to see if a pin object is actually threatened by the pinner. This becomes a complex LISP program and it would be a difficult problem to define a type

3 pattern language that could express this program in a transparent and modifiable manner.

2) Generality vs. specificity

Production rules which encode chess knowledge can be written at many different levels of specificity. At the most specific extreme is the hypothetical situation of having a huge data base of all possible chess positions, each of which is paired with the best move in that position. Viewing each of these pairs as a production rule yields a very specific pattern (matching the exact chess position) and a very specific action (telling which move to play). Very general chess knowledge can be seen in many competitive chess programs. For example, Zobrist's program might recognize the fairly general pattern that a move will cause a pin or will check. The associated action, again general, will be to add points to the plausibility score for that move. The problem with such general knowledge is that it does not capture the essence of a combination. For example, the specifics needed to distinguish a good check from a bad check are not recognized so many unproductive moves will be searched. For this reason, programs using general knowledge need large search trees to generate a correct analysis.

Knowledge-based programs like PARADISE need to express knowledge at a level of generality between these two extremes. The knowledge must be specific enough to capture at least part of the essence of a combination. Specific ideas about a position should interact to produce a gain in understanding and eventually suggest good moves for good reasons. However, the extreme of one production per possible position must be avoided. Any group of productions must be able to correctly analyze a whole class of possible positions. PARADISE seems to have a reasonable level of specificity in its productions. They are specific enough to capture the essence of most combinations, yet appear to solve a whole class of positions. (As described in chapter 6, PARADISE solves 69 problems with productions developed for 21 of them, and solves 20 more by adding only 13 productions.)

This tradeoff of generality and specificity is an ever present concern in PARADISE. It must first be considered when developing the primitive functions and patterns, and the production language which accesses them. Once the primitives and production language have been specified (at some level of generality), the grain size of the production rules is essentially fixed. But the generality-specificity tradeoff is much more than an issue of grain size. The most important generality considerations in PARADISE occur during the writing of actual rules (after grain size has already been set by development of primitives and the production-language). This is illustrated below (in figures 7.1 and 7.2). Generality considerations are also involved in other parts of PARADISE (e.g., in developing the threat-language and plan-language which can express plans and their threats at varying levels of generality).

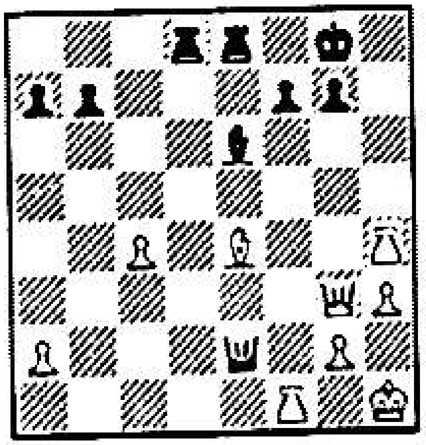


Figure 7.1
white to move

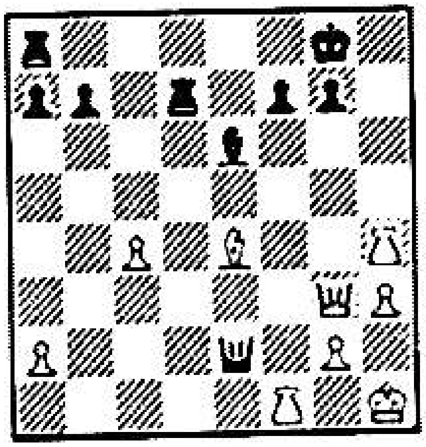


Figure 7.2
white to move

Figure 7.1 is problem 82 in [Reinfeld58]. It is the same as figure 5.1 and PARADISE's solution to this problem is presented in chapter 5. The plan suggested by PARADISE's static analysis which leads to the solution is (WB H7) (BK F8) (WQ A3). The productions which lead to the suggestion of this plan notice that the black king has only two moves after the check by white's bishop. Suggesting the check on this basis would be too general so the productions look for favorable continuations after both black king moves. They notice that if the black king flees to h8, it will be subject to a discovered attack, and that if it flees to f8, the white queen can check it in such a way that it has no legal moves. On this basis, the above plan is suggested, and it specifies the actual queen check which was found to answer the king's move to f8. (No continuation is given for the case of the king moving to h8 since the productions did not analyze particular moves for taking advantage of the discovered attack.)

The productions which suggest this plan are specific. They have good reason for suggesting this plan even though they do not notice all the facets of the eventual winning combination (e.g., the preparation of a back rank mate by the white rook on h8). To obtain a search as small as that in PARADISE, the productions would be too general if they suggested the bishop check just because it leaves the black king only two moves. Such productions might however be just right for a system that uses more search and less knowledge than PARADISE. If anything, the productions in PARADISE which suggest the above plan may be too specific. Figure 7.2 brings out the issues involved.

Figure 7.2 is a slightly altered version of figure 7.1 where essentially the same combination still wins for white. In figure 7.2 however, the above plan will not be suggested by the productions in PARADISE because the black king would now have legal moves after the white queen checks from a3. Perhaps this indicates these productions are too specific, but the same productions have found a number of good plans during the testing of PARADISE and seem to be performing satisfactorily. The winning combination in both the above positions consists of checking with both bishop and queen in order to set up a situation where the white bishop can then move to attack the black queen and set up a back rank mate by the white rook at the same time. Productions could be written to recognize this whole combination statically, but such productions would be too specific with PARADISE's current grain size. They would need to look 3 or 4 moves ahead, would be very expensive to match, and would help in very few positions. PARADISE should solve figure 7.2 by suggesting the bishop check in order to drive the black king away from h8 so the white rook can get there. This is specific, but not too specific. This plan is also suggested in figure 7.1 but is not as likely to succeed as the plan with the queen check described above.

PARADISE is commited to using specific knowledge to obtain a knowledge-controlled search. This means the system will have large amounts of knowledge. PARADISE uses KSes to efficiently access its knowledge, and uses plans to avoid doing a knowledge-based analysis in many positions. The person writing the productions is responsible for writing reasonable productions that are neither too general nor too specific. Such a person must work within the confines of the grain size established by PARADISE's primitives and production language (see next paragraph). PARADISE sometimes uses general productions when more specific ones could easily be expressed. More general productions are especially desirable when it appears easy to check if the suggested plan is working. For example, if a major piece is sacrificed unsoundly, it will usually be easy to determine this in the search since one side will be a piece down.

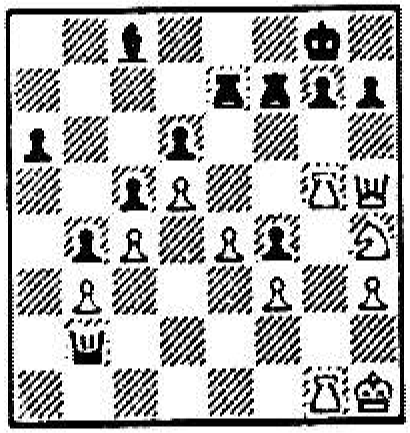
Sometimes the production language prevents a production from being as specific as it might like to be. For example, if the production language made it easier to check the effects of a piece after it had made 2 or 3 moves, productions would be able to be more specific when looking for deep combinations like the one in figure 7.1. One reason productions in PARADISE may be kept from being much more specific is the system's restricted ability to communicate through concepts that are posted in the data base. This is discussed in more detail in section B of this chapter.

The search in PARADISE expects a certain balance of search and knowledge. This means the person writing the rules in PARADISE must select a reasonable level of specificity since the methods used to control the search depend on it. The next section discusses this in more detail.

3) Search vs. knowledge

PARADISE has a considerable amount of chess knowledge, yet there are many holes in this knowledge, even in the limited domain of middle game tactics. Until a perfect knowledge base is foreseeable, a searching process will be necessary in programs to correct mistakes in the analysis and to fill in gaps in the knowledge. But a search may always become intractable in certain situations (unless there is some artificial effort limit). Thus no combination of search and knowledge will be a complete problem solver unless it has the ability to learn from previous mistakes. The issue of learning is not addressed here, but it is clear that various combinations of search and knowledge are possible in a chess program. The following discussion describes the trade-offs involved.

The issue discussed here is the trade-off between using large amounts of specific knowledge which strictly control the search, and using smaller amounts of more general knowledge combined with search mechanisms which halt the search quickly. To better define this issue, consider the following position from chapter 2.



white to move
figure 2.3

White has a five move mate in this position beginning with QxP. Chapter 2 describes PARADISE's solution of this problem which begins with a sophisticated plan. The initial static analysis involves many specific productions which check that black's king can be effectively attacked after it captures the white queen. The search generates only 21 nodes and a second static analysis is never done. This approach to the problem uses specific knowledge. Another approach to this problem is to search any check as a mating possibility and then have mechanisms in the search to quickly terminate the search if the check is not working out. This has the advantage of making the analysis much cheaper, and never missing a check which works. This latter approach uses loosely structured knowledge (a term used to differentiate it from specific knowledge).

Since PARADISE can express knowledge on many levels of specificity, the knowledge base can be written with either specific or loosely structured productions. If this were done, it might be possible to experiment with different balances of search and knowledge. The problem with this is that all chess programs, including PARADISE, have a search which expects a certain level of specificity in the analysis. PARADISE does not have mechanisms to quickly terminate the search when loosely structured knowledge is used. In the above example, PARADISE would need some way to determine that a check "does not work out". Similarly, a search-oriented program could not use an analysis based on specific knowledge because its search does not have the mechanisms to use the analysis in controlling the search (which must be done to offset the extra time spent analyzing).

There are two theoretical problems with using specific knowledge. First, there will always be holes in the knowledge base which may cause an incorrect analysis. Second, even after a hole in the knowledge is discovered which involves an obscure combination, it may not be reasonable to notice the obscure combination statically at the necessary level of specificity. An "obscure" combination is one which begins with a move that initially does not look good for any straightforward reasons. Thus any simple production for suggesting the move might not be specific enough and would suggest poor plans in many situations. The first problem may be solved by having a program learn or by having a human add knowledge. The second problem may be solved by some combination of more powerful pattern recognizers and better methods for controlling searches of plans suggested by less specific knowledge.

The major theoretical problem with using loosely structured knowledge is controlling the search. The above example shows the difficulties in this approach. Suppose some production suggests most checks (including QxP) for searching and specifies that these checks threaten mate. The search must determine when one of these checks is not working out. Aft r a few moves it may be obvious to a human or to a program with specific knowledg that the check is not working out, but in the program with less specific knowled , the same production that suggested checks originally will still (in general) be sugg ng checks which threaten mate. Various methods for terminating such a search could b ed. The program might do something dependent on depth, or it might stop believing it oosely structured productions at some point, or it might refuse to concatenate two plans from loosely structured productions. These suggestions all introduce errors similar to those introduced by a depth limit (e.g., if the program allows four of these checking plans to be concatenated, it will miss combinations with five checks). Other terminating methods seem to involve more specific knowledge about the checks being suggested, which brings in the problems of using specific knowledge.

PARADISE uses pecific knowledge and tends towards the knowledge end of the search-knowledge continuum. Search-oriented programs like CHESS 4.7 and TECH2 fall near the search end of the continuum, while CAPS and Pitrat's program fall somewhere in between. The m of search and knowledge a program uses helps determine the type of problems it can solve (see below). The search-oriented programs solve many of the same problems as PARADISE, but with less elegance and less expense. The remainder of this section describes some of the tradeoffs between these different approaches.

First, it should be pointed out that PARADISE's use of cpu time is inflated compared to that of the other programs. The other programs have all been tuned to run efficiently (TECH2 and CHESS 4.7 are written in assembly language) while PARADISE is still a research vehicle (written in MacLisp). PARADISE spends more than 91 percent of its cpu time in the inefficient production-language interpreter, and this could probably be speeded up by a factor of two by "compiling" the productions. This would make PARADISE's cpu usage comparable to that of the other programs, at least over the first 100 positions in [Reinfeld58]. In addition, PARADISE's data structures could be considerably more efficient and the cost of PARADISE's analysis could be spread over many moves in an actual game of chess (see section A of this chapter).

As described above, both the search-based and knowledge-based approaches have theoretical limits which imply that each approach can solve some problems the other approach cannot. Unlike PARADISE, programs with loosely structured knowledge have artificial effort limits. PARADISE can solve combinations which are deeper than the depth limits of such programs (or which exceed the artificial effort limits in Pitrat's program). As figure 6.4 shows, TECH and CHESS 4.4 rarely make a mistake on problems within their depth limit and rarely solve problems beyond their depth limit. PARADISE performs as well on the deep problems as on the shallow ones, and has a distinct advantage because of this on these 100 problems. (These problems are not extraordinarily deep; a human grandmaster would solve nearly all of them.)

On the other hand, the search-oriented programs can solve "obscure" combinations which PARADISE's productions could not recognize at a suitable level of specificity. It is hard to judge how restrictive this is, since improved pattern languages may be able to express even "obscure" combinations adequately. Except for one problem, PARADISE did not encounter this limitation on the 100 test problems. Currently, the drawbacks of knowlege based programs are practical, not theoretical.

In practice, PARADISE may not be able to solve many problems, even easy ones, because of holes in its knowledge. This is a serious drawback since programs with full-width searches almost never miss combinations within their depth limit. The strength of this effect depends on the completeness of the knowledge base. This effect can work both ways with programs like CAPS and Pitrat's since they also do not search all legal moves and may have holes in their knowledge. One might expect PARADISE to miss combinations more often than these two programs since its more specific knowledge suggests fewer moves, but this has not been the case in the comparison of CAPS and PARADISE on the test positions. PARADISE recognizes mating attacks that both CAPS and Pitrat's program miss. PARADISE's knowledge base appears complete enough to outperform CAPS on the first 100 positions in [Reinfeld58], even on positions within CAPS' depth limit. PARADISE-2's knowledge base is complete enough to outperform CHESS 4.4 on these problems.

The primary justification for the cpu time used by PARADISE and the biggest advantage of the approach used in PARADISE is the extendability of the knowledge base. This means the effect of holes in the knowledge can easily be reduced in PARADISE. The play of the other programs mentioned in this dissertation cannot be noticeably improved with easily made program or knowledge modifications. This would involve programming a more sophisticated analysis, or controlling the search better so that it could search deeper. (Play can, of course, be improved by switching to a significantly faster computer in most cases.) It would be hard to incrementally improve performance of a program by increasing its depth limit since each increment of one ply multiplies the effort required by the branching factor (usually about 5 or 6 for programs with full width searches that have a reasonable move ordering for alpha-beta). The possibility of incremental improvement through improved analysis is discussed below.

Programming a more sophisticated static analysis in any of the programs mentioned in this dissertation (except PARADISE) would probably not be easy for the programmer. Even if it were, these programs are commited (by tree size) to limit the amount of processing per node. CAPS spends more time per node than the others, but still processes five nodes per second of cpu time. CHESS 4.7 processes around 3600 nodes per second on a Cyber 176 ([Robinson79]). If CHESS 4.7 spent even an additional ten milliseconds per node in analysis, the size of the tree it could search in a given amount of time would be greatly reduced. Easy incremental improvement of CHESS 4.7 (and similar search-based programs) by making analysis progressively more sophisticated does not seem possible. This means the theoretical limitations imposed by the depth limit cannot easily be overcome.

The time constraints on CAPS and Pitrat's program would permit slight increases in the time spent on analysis. Such slight increases must involve loosely structured knowledge, so any knowledge which recognizes a new tactic would suggest more moves to be searched in many positions. This increases the size of the search tree exponentially. These program are already running efficiently and would either have to develop new search control mechanisms, or use significantly more cpu time to cope with this increase in branching factor.

For PARADISE's knowledge base to achieve the completeness needed to rival the best search-oriented programs (in middle game tactics), many productions may yet have to be added. There is no evidence that suggests large numbers of productions will harm PARADISE's performance (although problems may arise). New productions do not seem to significantly increase the branching factor or the time required to do an analysis.

As discussed earlier in this chapter, an increase in branching factor is avoided when added productions are at the right level of specificity. The limiting factor in PARADISE's incremental improvement appears to be the ability to recognize more complex tactics at the correct level of specificity.

New productions do not significantly increase the analysis effort largely because plans guide PARADISE's search so well that a static analysis is done at less than nine percent of the nodes generated. Thus most new productions will rarely be executed. Even when executed, productions usually do not match and the system can often determine this with little effort. Even when matched, well-written productions take only a few milliseconds to a few hundred milliseconds to match. Compared to the 12 seconds of cpu time PARADISE spends on a typical analysis, this is not significant.

Weighing the above tradeoffs in order to judge which approach is best depends on the current technology. With developments such as faster machines, parrallelism, or better hashing techniques, programs which rely on search can search deeper and perform better. The development of better pattern recognizers might increase the power of PARADISE's production language allowing PARADISE to perform better. There is little doubt that with current machines and our current understanding of how to use knowledge, good search-oriented programs solve a larger class of problems with less expense than do the best knowledge-based programs. However, techniques for using knowledge are only beginning to be developed and understood. This research shows ways to effectively use a large knowledge base which can be easily extended. As progress is made in representing and using knowledge, a knowledge-based program may eventually be able to approach (and even surpass) the performance of human chess masters.

[Adelson-Velskiy75]
Adelson-Velskiy G M, Arlazarov V L, Donskoy M V, "Some methods of controlling the tree search in chess programs", *Artificial Intelligence 6*, pp. 361-371, 1975.

[Berliner73]
Berliner H J, "Some necessary conditions for a master chess program", Proceedings of the 3rd International Joint Conference on Artificial Intelligence, pp. 77-85, 1973.

[Berliner74]
Berliner H J, "Chess as problem solving: The development of a tactics analyzer", Unpublished doctoral thesis, Carnegie-Mellon University, 1974.

[Berliner77]
Berliner H J, "On the use of domain-dependent descriptions in tree searching, in: *Perspectives on Computer Science*, Jones, A.K. (Ed), Academic Press, 1977.

[Berliner79]
Berliner H J, "The B* tree search algorithm: a best-first proof procedure", *Artificial Intelligence 12*, pp. ?, 1979.

[Bernstein59]
Bernstein A, et. al., "A chess playing program for the IBM 704", *Proceedings of the Western Joint Computer Conference, AIEE* , pp. 157-159, March 1959.

[Buchanan78]
Buchanan B G, Mitchell T M, "Model-directed learning of production rules", in *Pattern-Directed Inference Systems*, Waterman and Hayes-Roth (Ed), New York: Academic Press, 1978.

[Campbell66]
Campbell D T, *Pattern Matching as an Essential in Distal Knowing*. New York: Holt, Rinehart and Winston, 1966.

[Charness77]
Charness N, "Human Chess Skill", Chapter 2, in *Chess Skill in Man and Machine*, Frey, P. (Ed), Springer-Verlag, 1977.

[Chase73]
Chase W G, Simon H A, "The mind's eye in chess", *Visual Information Processing*, Chase, W G (Ed), New York: Academic Press, 1973.

[deGroot65]
de Groot A D, *Thought and Choice in Chess*. The Hague: Mouton, 1965.

[Davis76]
Davis R, "Applications of meta level knowledge to the construction, maintenance and use of large knowledge bases", AIM-283, Computer Science Department, Stanford University, 1976.

[Fahlman74]
Fahlman S E, "A planning system for robot construction tasks", *Artificial Intelligence 5*, pp. 1-49, 1974.

[Frey77]
Frey P W, "An introduction to computer chess", Chapter 3, in *Chess Skill in Man and Machine*, Frey, P. (Ed), Springer-Verlag, 1977.

[Gillogly72]
Gillogly J, "The technology chess program", *Artificial Intelligence 3*, pp. 145-163, 1972.

[Harris77]
Harris L R, "The heurictic seach: an alternative to the alpha-beta minimax procedure", Chapter 7, in *Chess Skill in Man and Machine*, Frey, P. (Ed), Springer-Verlag, 1977.

[Hayes76]
Hayes J E, Levy D N L, *The World Computer Chess Championship*. Edinburgh: University press, 1976.

[Kotov71]
Kotov A, *Think Like a Grandmaster*. Dallas: Chess Digest, 1971 (translated by B. Cafferty).

[Kozdrowicki73]
Kozdrowicki E W, Cooper D W, "COKO III: the Cooper-Koz chess program", *Communications of the ACM*, pp. 411-427, July 1973.

[Levy76]
Levy D, *1975 U.S. Computer Chess Championship*. Woodland Hills: Science Press, 1976.

[Lorenz73]
Lorenz K, *Behind the Mirror*. New York: Harcourt Brace Jovanovich, 1977 (translated by R. Taylor).

[Michie78]
Michie D, Bratko I, "Advice table representations of chess end-game knowledge", Proceedings of the Third AISB/GI Conference, D. Sleeman (ed), pp. 194-200, 1978.

[Pitrat77]
Pitrat J, "A chess combination program which uses plans", *Artificial Intelligence 8*, pp. 275-321, 1977.

[Post43]
Post E, "Formal reductions of the general combinatorial problem, *American Journal of Mathematics*, vol. 65, pp. 197-268, 1943.

[Reinfeld58]
Reinfeld F, *Win At Chess*, Dover Books, 1958.

[Robinson79]
Robinson A L, "Tournament competition fuels computer chess", *Science*, vol. 204, pp. 1396-1398, 1979.

[Sacerdoti74]
Sacerdoti E D, "Planning in a hierarchy of abstraction spaces", *Artificial Intelligence 5*, pp. 115-135, 1974.

[Sacerdoti75]
Sacerdoti E D, "The nonlinear nature of plans", Stanford Research Institute Technical Note 101, January 1975.

[Shortliffe74]
Shortliffe E H, "MYCIN - a rule-based computer program for advising physicians regarding antimicrobial therapy", AIM 251, Computer Science Department, Stanford University, 1974.

[Simon73a]
Simon H A, Gilmartin K, "A simulation of memory for chess positions", *Cognitive Psychology*, vol. 5, pp. 29-46 1973.

[Simon73b]
Simon H A, "Lessons from perception for chess-playing programs (and vice versa)", Carnegie-Mellon University, Computer Science Research Reviews 1972-73, pp. 35-40.

[Slate77]
Slate D, Atkin L, "CHESS 4.5- The Northwestern University chess program", Chapter 4, in *Chess Skill in Man and Machine*, Frey, P. (Ed), Springer-Verlag, 1977.

[Winograd77]
Winograd T, Bobrow D G, "An overview of KRL, a knowledge representation language", *Cognitive Science 1*, pp. 3-46, 1977.

[Zobrist73]
Zobrist A L, Carlson F R, "An advice-taking chess computer", *Scientific American*, vol. 228, pp.92-105, 1973.