The University of Alberta

MACH: A Master Advisor for CHess

by

Michael William George

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1989

ISBN 0-315-52843-5

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Michael William George

TITLE OF THESIS: MACH: A Master Advisor for CHess

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1989

(Signed) [illegible]

Permanent Address:
60 Wentworth Street
Saint John, New Brunswick
Canada E2L 2R9

Dated 9 December 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **MACH: A Master Advisor for CHess** submitted by **Michael William George** in partial fulfillment of the requirements for the degree of **Master of Science.**

[illegible]
Supervisor

[illegible]

[illegible]

[illegible]

Date [illegible]

ABSTRACT

Since Shannon first described, in 1949, how a computer could play chess, much effort has been devoted to the development of a chess machine that could compete with the world's top human players. Over that time, many challenges have been made between man and machine, with man, to this point, still showing his superiority. Only recently have chess systems begun to narrow the gap of achieving grandmaster status.

For the most part, developers of chess systems have concentrated more on improving the speed of their systems and the number of moves ahead that can be searched, instead of analyzing why humans play so well. Psychological studies have shown that part of the strength of a grandmaster is directly related to his perception of a given chess position and his recognition of previous, similar ones. These refined abilities allow him to extract relevant patterns (or chunks) from the board and combine them into a similarity measure; this measure serves as a key for searching the player's vast mental database of previously stored patterns and positions to find similar ones. Matching similar positions triggers moves and strategies for the player to help guide his search for the best move.

By applying these same methods, we can improve the performance of a chess program. Since virtually every master and grandmaster game ever played is now readily available in machine readable format, we could instruct a chess program to search a database of games to find similar positions to those that it would encounter during normal play and subsequently extract the strategy, the moves, the blunders of the players involved and use them to play better chess.

This thesis develops an experimental testbed on which to determine such a system's feasibility. By providing chess programs with grandmaster advice, we believe, is one way of closing the gap between man and machine. As well, the ideas developed here have applicability to other domains.

Acknowledgements

To begin, I wish to thank my supervisor, Jonathan Schaeffer, for his advice and guidance, and allowing me to tap his formidable chess knowledge. His superior analytical skills have taught me a great deal, even though his choice of a hockey team is questionable. As well, a special thanks goes to Carol Smith and Bev Bayda who helped me format this document correctly.

Most of all, I need to thank my wife, Jan, whom I dragged 3000 miles so that I could get my M.Sc. Without her support, it never would have been possible. Last but not least is an acknowledgement to my future child whom I hope to teach as much as this graduate degree has taught me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. Motivation - Building a Better Chess Machine

It has been almost 40 years since Claude Shannon [Sha50] first described how a computer could play chess. Since then, a great deal of research has been conducted to improve on his methods. For the most part, current chess programs consist of routines to make and analyze moves along with small knowledge bases to enable them to play well. Unfortunately, despite the memory capacities (gigabytes) and lightning-fast processing speeds (100 MIPS or more with parallel processing, though this value is continually increasing) of modern-day computers, the top performing chess programs (e.g. Hitech [Ebe87], Cray Blitz [Hya86]) are only now beginning to challenge grandmasters. Why?

The fundamental reason for the slow progress of chess programs lies not in errors of programming but rather in the methods these programs utilize to choose the *best* move. Making and analyzing moves during a game involves tree searching and move generation [1], something which has been well studied. However, supplying a program with enough knowledge to play well is very difficult to do. Most chess knowledge lacks precision of expression; encoding details into rules and exceptions can be too restrictive. Indeed, knowledge surrounding grandmaster play and technique cannot be adequately expressed in words, let alone formalized.

We know that the "chess problem" (i.e. achieving a checkmate) is mathematically solvable but, unfortunately, is computationally intractable. No known algorithm exists which will yield the best move in a position, unless the complete exploration of every possible move is conducted. Those programs which apply these so-called *brute-force* tactics to choose the best move must battle the overwhelming size of the search tree. If we consider that there are, on average, between 40 to 45 complete moves per game (80 to 90 positions or *plies*) and from each of these 80 to 90 positions there exist approximately 35 legal moves, the

[1] For a more complete description, see [Bar80]

total number of positions that could potentially be explored is 35^{80}. Actually, this value is too large, since it does not take into account transpositions (i.e. the same positions arrived at through different sequences of moves). Even by reducing this number to 10^{40} and having a computer that can run at 1,000,000 MIPS (i.e. for simplicity, let this represent 10^{12} positions examined per second), it would still take over 10^{18} centuries to examine every complete path!

In actual play, however, it has been determined that the average number of worthwhile moves in any position is slightly less than 2 (on average, 1.76 [DeG65]) and the average number of moves in a master level game is 42 [DeG65]. So that brings our total number of positions down to near 1.76^{84} - still a huge number. Given current trends in hardware technology, we can hazard a guess that it will take centuries to develop a machine that could examine that many positions in a reasonable amount of time. While it is probably true that in 5 to 10 years a chess program will be able to search a full 14 to 16 ply deep using the most sophisticated hardware available, unfortunately, there will always be cases where 16 ply is not deep enough to decide what the best move is - i.e. when at least one more move was needed to prevent a loss or secure a win. This is a problem inherent to programs which apply fixed depth searches (e.g. brute-force).

It is obvious that no chess player, regardless of his skill level, would ever come close to examining that many positions in a game; rather, a player must be highly selective in choosing which paths to explore due to the exponential growth of the search. It was once thought that grandmasters performed well because of their spectacular search ability, examining hundreds of positions at blinding speed while the poor beginner could look ahead but a few moves. Fortunately, this scenario had been discounted by the Dutch psychologist/computer scientist Adriaan De Groot some 20 years ago; in fact, he suggested that grandmasters may actually search *less* than beginners [DeG65].

So what is the solution? It is clear that using faster machines is not the answer; there are just too many paths to explore. There exists a well known hypothesis which suggests that doubling computing power will result in a performance improvement of close to 100 chess rating points[2]. However, a result by

[2] These points will be outlined later; for the time being, more is better.

Thompson [Tho82] suggests that this relationship does not hold once the program attains a master rating. In any event, human processing speeds are not even close to 100 MIPS and yet humans are able to play with remarkable precision.

So what are we missing? "Psychological studies have shown that chess masters have learned to 'see' thousands of meaningful configurations of pieces when they look at a chess position, which presumably helps them decide on the best move, but no one has yet designed a computer program that can identify these configurations" [Bar80]. This author believes these psychological results hold the key to building a better chess machine. By extracting knowledge from grandmaster results, we can tailor a chess program to examine only worthwhile strategies and search only useful branches. It is in this direction that this dissertation will proceed. We will attack the problems from a hardware-independent view, as most theoretical computer scientists do, and concentrate on applying the underlying principles involved in human problem solving.

1.2. Related Work

It will prove helpful to examine the work that has gone into game-playing programs over the last 30 years to identify their shortcomings, pitfalls but, more importantly, the researchers' intuition.

Ten years following the publication of Shannon, A.L. Samuel pioneered the study of learning as applied to the game of checkers through a series of experiments [Sam59, Sam67]. Like chess, checkers is a difficult game to play well, though it is less complicated (fewer types of pieces and fewer legal moves); as well, a complete search of its game tree is not practical, since there are somewhere near 10^{20} moves to be explored.

Using the standard minimax search approach, Samuel's program searched ahead only a few plies and applied a static evaluation function to determine which side was winning; based on this function, the program then chose the move that lead to the best position. As each position was encountered, a description of that position was "memorized" by the program along with the value from the evaluation function. When a previously stored position was encountered during subsequent play, its "positional-value" could be retrieved rather than recomputed. As more and more games were played, the program came across more and more familiar positions. Indirectly, the look-ahead power of the program became stronger without searching farther in the tree.

For more insight, refer to Figures 1.1 and 1.2. In the first figure, we see that the program has searched 3-ply from position *A* to position *D*, evaluating that the path through *B* and *C* is the best. The resulting value of 8 is passed up the line to position *A*. A description of this position along with the value of 8 is stored by the program for later use.

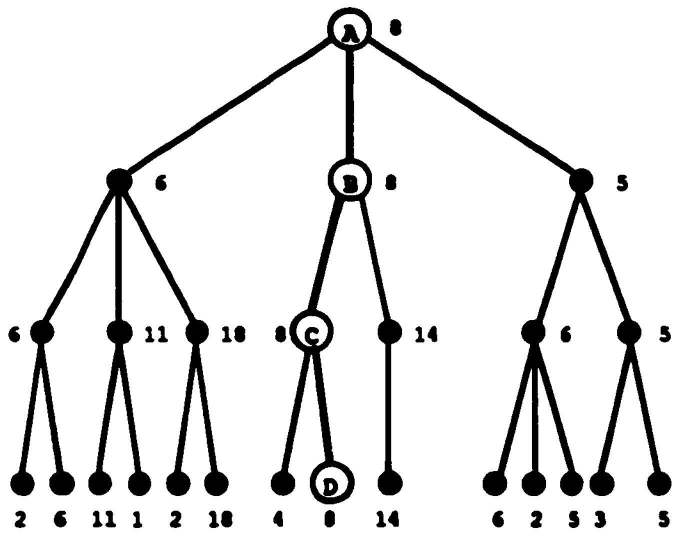


Figure 1.1 A typical 3-ply search conducted by Samuel's checkers program.

If we now look at Figure 1.2, we can see where this memorization of position *A* becomes important. One of the terminal nodes after a 3-ply search from *E* is *A*. The program already knows its evaluation value based on the previous search to *D*. In fact, this value is more accurate than the static value of *A*, since it involved a search. In effect, the program now has information from a 6-ply search through *A* without using additional search time. One can see that as more familiar positions are encountered, the deeper the effective search will become.

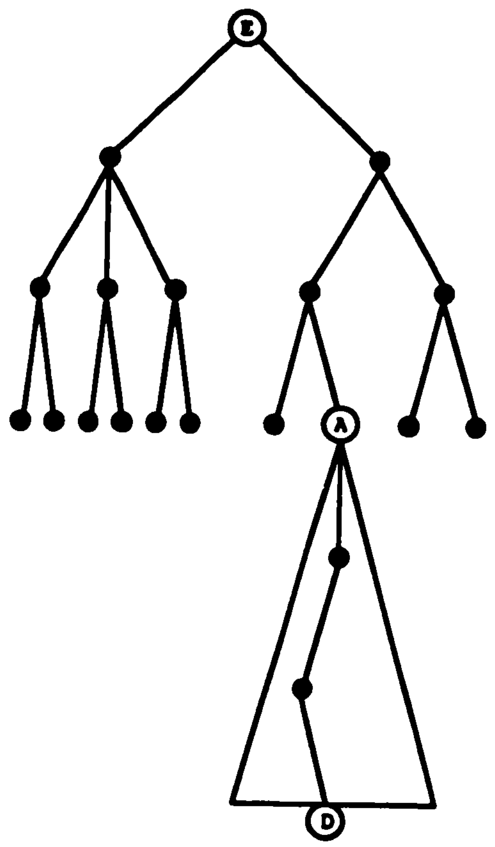


Figure 1.2 The effect of saving a score.
When a saved position is encountered after a normal search,
the effective search depth increases.

Samuel found, however, that even though the program fared well in the beginning- and end-games, its poor play throughout the middlegame resulted in its inability to achieve expert status, since he was restricted to using a small IBM-704 and writing all his positions to tape (in his later experiments, the program did achieve a master level, though).

Using such limited and primitive resources, Samuel placed great importance on efficiency - i.e. getting programs to use as few resources as possible. It can be inferred from Samuel's results that performance (in some sense) of any system can be hampered by lack of efficiency. In a discussion surrounding this topic, Lenat, Hayes-Roth and Klahr [Len79] outlined the abilities that an AI program must possess to be efficient, or as they described to be "cognitively economical." These abilities included,

- *Intelligent learning* - being able to sense change in the environment and adapt and/or modify itself to reflect the change.
- *Intelligent redundancy* - storing the results of frequently requested searches so they do not have to be recomputed, referred to as *caching*.
- *Intelligent focus of attention* - using predictions to filter expected results and use "surprises" to help the system model the environment more consistently.

In Samuel's follow-up study [Sam67], he attempted to improve his program by improving the evaluation function. Of course, a more detailed evaluation function meant greater computational time which translated into longer search time; consequently, Samuel employed methods (alpha-beta pruning, for example) to reduce the number of nodes examined. However, an even greater problem was being exposed through his research - the relative importance of knowledge.

In 1987, van der Meulen addressed, at least partially, this problem for the game of chess [Van87]. Instead of using a single evaluation function for all chess positions, van der Meulen applied a *set* of specialized evaluation functions, one for each type of chess position. By using a set of 500 chess positions and knowing the correct move in each, he constructed groups of similar positions using *statistical discrimination* and assigned a correct evaluation function (i.e. one that would produce the correct move) to each group by solving a system of inequalities. As a position was encountered during actual play, it would be mapped into one of these groups and the corresponding evaluation function of that group would be applied to that position, hopefully yielding the correct move (since it did so using the test positions of the group). By employing different functions to different sets of positions, thereby shifting the importance of certain parameters, the problem of weight assessment in evaluation functions was more suitably addressed. Unfortunately, his methods for determining similar positions were quite unnatural and certainly non-trivial.

Recently, building on the methods of Samuel, David Slate worked with transposition tables in chess as a means of learning through experience [Sla87]. His program, called *Mouse*, actually learned to avoid repeating mistakes, albeit only tactical ones. However, the experience used was only that of the program and useful only for identical positions; we will see later why positions are not required to be identical to be useful.

The idea of similarity and identifying similar positions has been a prevalent theme in chess research for the last 20 years. However, implementing correct procedures has been a problem because the notion of similarity requires a great deal of chess knowledge.

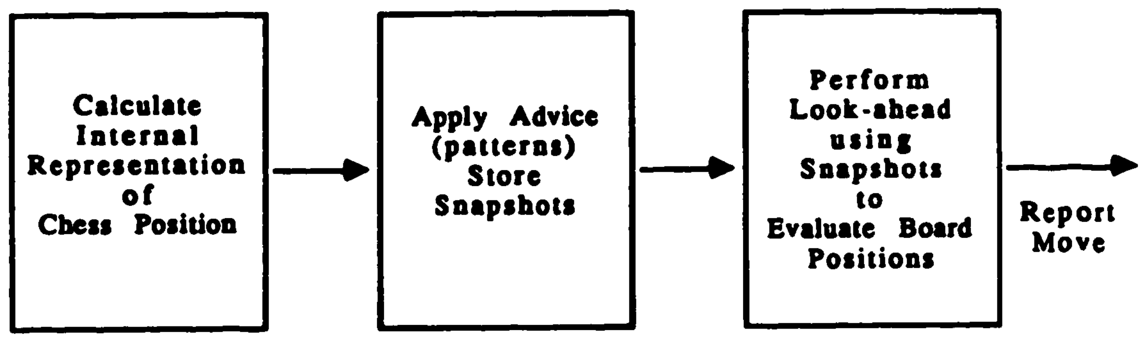

Calculate Internal Representation of Chess Position
Apply Advice (patterns) Store Snapshots
Perform Look-ahead using Snapshots to Evaluate Board Positions
Report Move

Figure 1.3 Three basic steps performed by the Zobrist et. al. program.

A big step towards this goal came in 1973 at the University of Southern California [Zob73]. Unlike most chess programs of the day which relied on brute-force and mathematical algorithms to choose the best move, the USC system searched a given board position for all instances of certain patterns and recorded them internally into a "snapshot." These patterns (or chunks) were coded in a simple but effective chess language, related mostly to attack and defence properties of the position. Figure 1.3, taken from [Zob73], outlines the three basic steps employed the program. As the authors pointed out, "for the first time the computer [was] no longer completely dependent on expert programmers for the acquisition on chess knowledge. We [saw] in our approach the fascinating possibility that the game's greatest players, such as Fischer, could record their chess technique for posterity." A significant result from this approach was that minor changes in chess knowledge simply involved changing the description of the patterns; using conventional programming techniques, on the other hand, would involve many hours of programming and debugging. Given the hypothesis that human masters use pattern recognition skills to recall similar situations that aid in their move choices [ChS73a, Sim73a, Sim73b], the USC method certainly models human problem solving more closely and provides a more natural approach.

Further to this notion of pattern knowledge and recognition, Bratko and Michie developed a package known as *Advice Language 1* (AL1) [Bra80] which was used to transfer knowledge surrounding chess endgames to a chess program. They constructed Advice Tables which contained advice lists (tactics to use against the opponent), sets of goals (things for which one should strive) and a number of move constraints to improve efficiency. Interestingly, their method required the pieces of advice to be ordered such that the most ambitious goals came first (e.g. to mate); by doing so, it guaranteed that the program would play towards the most desirable goal (to win).

It should be evident that pattern recognition in chess is required for players to be successful; why should it not also be required for chess programs? To accomplish this, however, will require a program that learns and uses experience, not necessarily its own. We will explore these notions next.

1.3. Learning, Experience & Chess

AI research on learning has been going on for years. There are many views on the subject, but Simon's perspective is probably the most general; he describes learning as "any process by which a system improves its performance" [Coh82]. This improvement can be realized by applying new methods and/or knowledge to the problem or by improving current methods by making them faster, more accurate or more consistent. Figure 1.4 illustrates more clearly this simple model of learning. The environment supplies information to the learning element to help it make improvements to the knowledge base of the system; this new knowledge, in turn, enhances the performance of the system. During execution of a task, information is fed back to the learning mechanism to further develop the knowledge base.

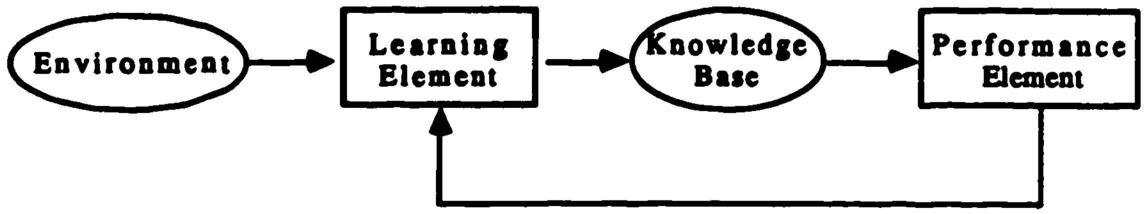


Figure 1.4 A general model of learning.

For the most part, four types of learning situations exist:

- learning by memorizing (rote learning)
- learning by taking advice
- learning by example
- learning by analogy

The first type of learning is characteristic of Samuel's checker player. By memorizing positions, his program improved its performance by recalling these positions when similar ones were encountered during play. Rote learning simply involves storing new knowledge so that when it is needed again, retrieval (vs. computation) is all that is necessary. Learning by taking advice is illustrated in Bratko and Michie's AL1 package. The system interpreted high-level pieces of advice, relating it to what it already knew, and proceeded to fulfill the supplied goals. Learning by example is similar to rote learning but involves generalization; like induction, the methods must determine the relationships involved in the examples and generalize them to be applicable to the problem(s) at hand. Learning by analogy is similar to learning by example in the sense of generalization; in this case, though, analogies must be recognized and used to improve the performance of the system. These last two types have received little attention in the literature and are probably the most difficult to specify.

Tied to the concept of learning is that of experience. Past situations must be recalled to guide and improve. Certainly, chess machines must both "experience and learn" to become more effective.

By examining human learning and experience, one can hopefully open the doors to creating a more powerful learning machine. For example, in the Data Processing (DP) world these days, phrases like "...with 3 to 5 years experience using CICS in a DOS/VSE environment..." is an often used phrase required of Systems Analysts' positions. The operative word here is *experience* - that intangible skill that is so vital in the DP world, and in most areas, in fact. Both educational and work experience provide people with the necessary skills to solve problems that are identical or *similar* to previous situations. The more experience a person possesses, in general, the more easily and effectively he/she can deal with problem situations.

To master a skill, such as playing chess, is no different. To become a grandmaster (GM), one must play thousands of games not only to gain "experience" but also to learn different openings, attacks and counterattacks, see different strategies and combinations of moves and to be exposed to endgame play. From this intense exposure, a GM remembers important fragments of games (sometimes even the entire game score) to allow him to relate his future games to similar past encounters. This recall is a fundamental component of a GM's experience, as we shall see; it influences many of his decisions and fine-tunes his expertise.

Unfortunately, experience, the proverbial *best teacher*, is lacking in today's computer chess systems. From incomplete knowledge, these systems usually do not learn from their mistakes; they could make the same wrong move in similar situations, game after game, although Slate's Mouse system was able to overcome this drawback in specialized situations. From playing thousands of games, GMs, on the other hand, accumulate a wealth of knowledge and rarely make the same mistake twice. What if chess programs could, somehow, tap into this experience of GMs? Remember, recording the chess technique of the world's greatest players was something Zobrist et al. conceived of doing. A chess system that could improve by observing the moves and strategy of the GMs could not help but play like them; certainly, this is what most advanced players do to improve their performance. Learning from others' experience (by example or analogy, in a sense) may provide the key in making these programs behave more human-like, a characteristic foreign to many expert systems.

1.4. The Idea

Currently, it is possible to have virtually every master and grandmaster chess game ever played in machine readable format (e.g. ChessBase [Edi87]). If a chess program could search this vast database of games and find identical or similar positions to those that it would encounter during normal play, i: could extract the strategy, the moves, the blunders of the players involved and use them to play better chess (if a GM played a certain way, there is a high probably that it was the "best" way to play).

The uses of this information would have far reaching implications. For example, the size of a chess program's "opening book" could be expanded; no longer would it have to be "taught" an opening or be coded with the strategy behind the opening. Instead, the program could follow a GM's strategic openings. In addition, with well-designed similarity functions, a program could match similar positions and follow the GM's plan throughout the middlegame. As well, endgame performance (the nemesis of most chess programs) could rise dramatically. Since GMs sometimes follow "standard" endgame strategies much too difficult to code into chess knowledge or heuristics, a program following the example of a GM's endgame play could increase its likelihood of a victory.

More importantly, this idea could be extended to other areas of expertise such as medicine or law. For medical diagnosis, past cases of millions of patients could be stored in a database and indexed by a similarity measure (e.g. the major symptoms of each illness). A physician could then determine the symptoms of a given patient, search this database for similar cases and consult the prescribed diagnosis.

In the field of law, analyzing past cases is a common occurrence. Lawyers frequently search for precedents to help design their cases. A case database could be constructed to aid the lawyer in this search. For example, each case could be indexed by its type (e.g. a murder), how the trial was conducted (e.g. by judge and jury), and the damages involved (e.g. $1,000,000). Within minutes, the lawyer would have, at his fingertips, a list of similar cases and their outcomes.

Indeed, for these domains, finding a similarity measure is much easier than for chess. In fact, a few keywords may be all that is needed to distinguish dissimilar cases.

1.5. The Thesis

In theory, this sounds like a hopeful solution. Notwithstanding, there lies many issues to be addressed before any practical implementation can arise. Therefore, it is not the intent of this project to efficiently design, code and implement a complete experience teacher for chess. Rather, we wish to devise an experimental testbed on which to determine such a system's feasibility. In other words, our task is to see whether a chess program's performance could be improved with the addition of this experience and we want to determine this result as quickly as possible. It will be evident that our implementation will be (in spots) grossly inefficient; however, the speed of the experimental system will certainly not influence its correctness. To build a wonderfully quick and efficient system would be pointless if it could not improve the play of the chess program.

Two critical issues must be addressed to implement such a system: **similarity** and **search**.

Where one draws the line to determine whether two positions are similar is certainly a non-trivial problem. We must be able to weed out useless positions since our database of games and positions will number in the millions. As mentioned previously and described more thoroughly later, psychology theory suggests that human players recognize certain patterns on the chess board and the combinations of these patterns guide the players' search for quality moves and enables them to recall "similar" positions. This so-called "chunking" hypothesis will serve as a basis for our similarity tests.

In regards to search, once a similarity value for a position has been calculated, it is necessary to access only worthy candidate positions in our database swiftly since time is so crucial in a real-time application. Our effort expended on developing effective similarity measures would be futile if simil [illegible] ons could not be retrieved within a few seconds.

By no means are those the only concerns to be addressed; indeed, additional problems will arise which must be solved at some point to produce a workable system. For example, how do we develop a database of useful patterns, large enough to produce an effective similarity measure? As well, once the patterns in a position have been found, how do we determine those that are most important to the ensuing strategy of the game from which the position was extracted? Moreover, what happens if we encounter a position that contains no recognizable patterns? Suggestions to combat these barriers will make up a large part of our discussion.

Assuming our experimental system succeeds, though, many significant implications would arise. First of all, our knowledge base would be easily extendible since not only could information from middlegame and endgame books be included but also 6000 new and useful GM games, on average, could be added each year from various sources. Secondly, "human-like" play (a potentially important feature) would be a possibility, in hopes of negating some of the inferior "machine-like" tendencies. But more importantly, a program's performance could improve without additional programming effort; new games mean new positions which effectively increases the chess knowledge at the program's disposal. The more available positions results in more matched positions (similar or identical) for a chess program which, theoretically, should make it play stronger games.

1.6. Outline

Our discussion will be divided into 5 chapters. Chapter 2 will introduce the theory associated with chunking and describe some experiments in chess cognition. With the basics outlined, chapter 3 will provide an in-depth description of our testbed system (referred to as MACH - Master Advisor for CHess) detailing the modules that are (or should be) integrated as well as discussing some options for an efficient search routine; these modules will be designed to work independent of any chess program. Our language for describing chess patterns will then be supplied in Chapter 4. The following chapter will detail the results of the experiments, showing where MACH did and did not help the chess program (for our tests, we will interface with the chess system *Phoenix* [Sch86]). Finally, the sixth chapter will suggest ways for improvement and outline a series of tasks to be completed that will produce an effective and useful experience teacher for chess.

Chapter 2

Skill & Perception in Chess

2.1. Human Game Playing

Humans are very good at learning board games. After given a brief introduction to the rules of the game including the goals involved, humans usually play reasonably well. In fact, the moves made by a beginner are almost always legal, most often poor but usually never random [Eis73]. From the beginning, even inexperienced players generate primitive strategies (e.g. "capture pieces"). Games, by definition, are goal directed which guide players in move selection and strategy development.

Quality of human play can be dramatically improved through teaching and experience; people can be taught certain moves, powerful strategic concepts and to recognize key patterns of pieces. In the chess world, part of the strength of the master or grandmaster is his ability to perceive the current situation (i.e. determine the important features and characteristics) and recognize that it is similar to one from another game (or games) he has previously played. For the most part, this ability allows him to extract relevant patterns from the board, which might include a king in check, an open file or a passed pawn. The combination of the important patterns in a given board position serves as a key for searching the master's vast mental database of previously stored patterns and positions (50,000 or more [Sim73b]) to find similar ones (i.e. those that resemble the current position, at least as far as the master is concerned). Associated with each of these positions is the so-called "best" move or strategy - something the player has executed many times before and knows to be superior. Unlike today's computers, the human brain has a very complex associative memory and mental "programs" which allow the player to encode a position into chunks and reconstruct similar positions within a few seconds - indeed, a vital component for the master player.

Hence, an advanced player can be influenced to make a particular move based on his experience of being in a situation similar to the one at hand and remembering the best move or strategy to perform. As a result, only a limited amount of look-ahead may be needed to evaluate the best move (7 plies seems to be the maximum search depth most grandmasters usually require [Cha77]). In some instances, though, he just *knows* what move to make (consider the Blitz Tournaments). Since beginners remember only a few thousand patterns, they, more often than not, are faced with unfamiliar positions and must resort to more search. In fact, De Groot has shown that most of their search time is spent examining worthless moves, unlike the grandmaster who seems to know the best paths to explore [DeG65].

Of course, this superior ability does not mean that the GM possesses some sort of photographic memory or is significantly more intelligent than the novice. Obviously, a certain aptitude is required (spatial abilities, for example) but more importantly, it is the hours of practice and dedication that is needed to build up a large knowledge base of patterns, along with their meanings and a search method tailored to pursuing useful moves - no experts develop overnight, by any means. Generally, to emulate the successes of a Fischer or a Kasparov one must have the motivation, the time and the desire to play and study chess for many, many years; some estimate a minimum of a decade is needed [Cha77].

To continue, though, let us look deeper into how these hypotheses were developed.

2.2. Development of a Theory of Human Game Playing

As previously stated, De Groot, in the mid 1960s, explored the reasons why master-level players performed so much better than their novice counterparts. Some eight years later, Chase and Simon continued to examine the performance of chess masters [ChS73b]. Their studies involved a number of experiments including a memory test, similar to that conducted by De Groot.

For a period of 2 to 10 seconds, masters and beginners alike were shown meaningful chess positions (i.e. those taken from actual games) containing 20 to 25 pieces and asked to reconstruct what they had just seen. Masters' reconstructive abilities were 93% accurate, experts placed 72% of the correct pieces on the correct locations, class A players scored only 50% while beginners remembered a mere 33% of the pieces.[3]

[3] Players are ranked based on a number of factors, including the number of games won and lost and with whom they were played. In the USCF (United States Chess Federation) system, the scale is: Class E (beginners) < 1200, Class D: 1200-1399, Class C: 1400-1599, Class B: 1600-1799, Class A: 1800-1999, Expert: 2000-2199, Master: 2200+ and GMs generally have ratings over 2500 [Cha77].

These same players were later shown random positions (i.e. where pieces were haphazardly placed) for the same length of time and asked to reconstruct them. Surprisingly, the beginners performed just as well (or poor) as the masters; all players could recall only 3 or 4 pieces (about 20%). These results conclusively showed that the superior players were subject to the same short-term memory constraints as the weaker players. But when these advanced players were faced with subject material that was meaningful, they truly outperformed the less experienced, indicating that some deeper perceptual processes must have been at work. The ability to recognize familiar patterns on the chess board, Chase and Simon found, was a significant difference between good players and outstanding players.

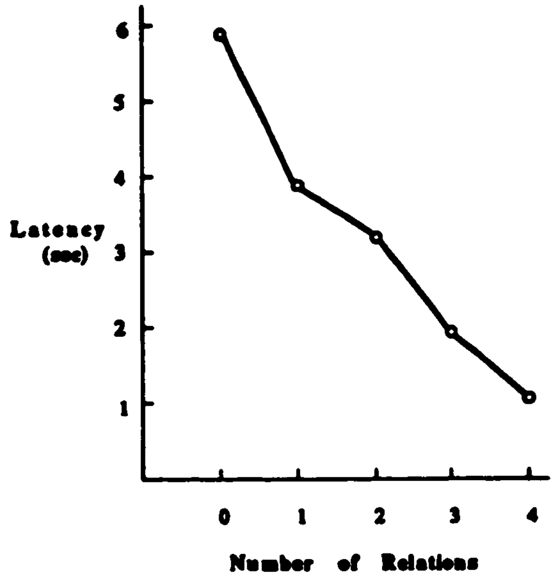


Figure 2.1 Average time required to place a chess piece as a function of the number of chess relations.

To better understand these perceptual processes, Chase and Simon devised what they called a perception task. Two chess boards were placed on a table, one containing the control position and the other empty. The chess players were then asked to reconstruct the control position on the empty board as quickly and as accurately as possible (they could look back to the position as often as they wished). Their head movements were used to segment the patterns (or chunks) of the position, assuming, of course, that after each head movement one pattern was recognized (i.e. one chunk was encoded in the player's memory). It was found that the players placed within-chunk pieces significantly quicker than between-chunk pieces. Pieces within a single chunk were more closely related than those from different chunks; in particular, they were related in terms of attack and defense properties of the pieces, as well as common color, type and proximity factors. In fact, they were able to determine that if the player paused for more than 2 seconds, a new chunk was being assimilated. Figure 2.1, taken from [ChS73a], clearly illustrates that as the number of relations among a group of pieces (or centred on one piece) increases, the time required for the placement of those pieces decreases.

2.3. Eye Movements

It would seem that the first few seconds (after any move) are very [illegible], as it is in this time frame that the master assesses the current situation to determine his best recourse, [illegible] the board to find the important or so-called *salient* pieces that typify a chunk. Two Soviet psychologists, Poznyanskaya and Tikhomirov, in 1969, determined that "human intellectual activity is ... clearly reflected in eye behaviour, which reflects the grasping of various types of interrelationships between elements of situations and establishes the properties of the specific elements as a result of this interrelationship." [Poz69].

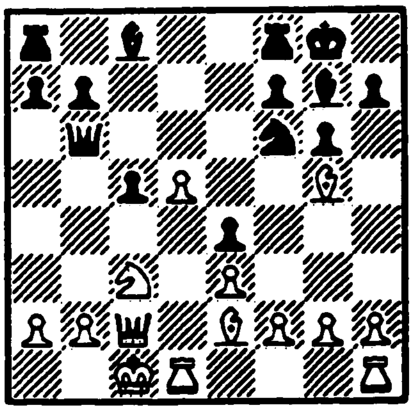


Figure 2.2 Middle game position used by Tikhomirov and Poznyanskaya.

They placed a chess expert before a chess position (see Figure 2.2) with instructions to seek the best move while they observed his eye movements during the first 5 seconds of the experiment. Since the human eye moves in saccadic motions, they were able to record 20 different fixations that rested on squares occupied by pieces that any chess player would deem important to that position. Some fixations landed on empty squares or the edges or corners of the board while the majority of the fixations moved between pieces that were in one or more chess relations (recall previous discussion) with each other. Interestingly, the subject seemed to "know" where to look next. This suggests that while the player is focusing on the current square (the human eye can clearly focus on an area of only 1° in radius [Not71]), he must also be examining the periphery of that square. From visual scanning experiments based on known scanning rates of the human eye, it seems that some parallel processing must be occurring - i.e. the actions of searching the periphery for the next square and preparing to move the eye there must overlap in time [Eli71].

The study of these eye movements is relevant to the study of chess skill not only because it is an objective test ("...eye behaviour depends on the problems facing the subject, reflects the nature of the problem, and most importantly has a problem-solving function" [Poz69]) but also because it serves as strong evidence in favor of the hypothesis that players first become aware of the structural patterns of the position (perception) before they begin to look for a move.

2.4. Simulations of Memory

To further explain human eye movements and support the claims of Poznyanskaya and Tikhomirov, Simon and Barenfeld [Sim69] developed a program (called PERCEIVER) to duplicate the eye movements of the human player; i.e. find a square on the board, acquire information about the peripheral squares and, based upon the simple relations of attack and defense, move to another square on the periphery.

PERCEIVER was able to show the same pre-occupation with important squares as did the human expert (see Figures 2.3 & 2.4). Hence, from using this simple program, it is clear that the chess master first examines the board for chess significant information (picks out the patterns) and then applies the best strategy based upon this information.

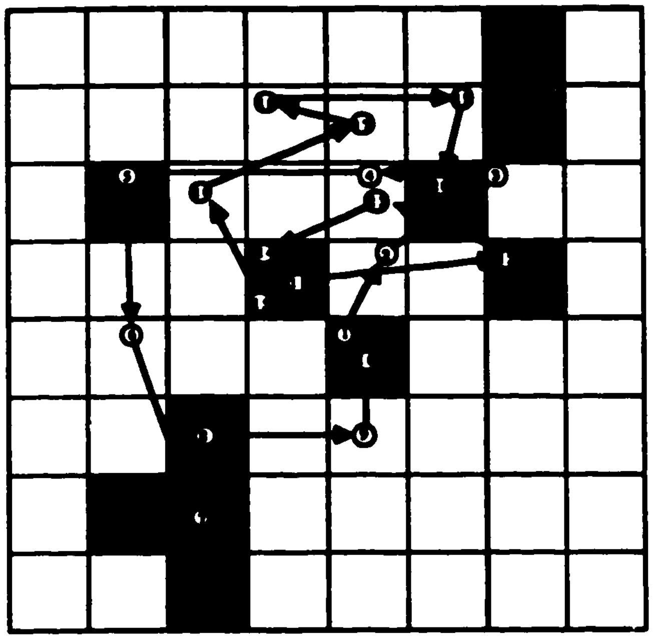


Figure 2.3 Eye movements of the expert player by Tikhomirov and Poznyanskaya. The squares occupied by the most active pieces are shaded.

Using the result that a player's reconstructive ability (his pattern recognition processes) depends upon his chess proficiency and whether or not the position is meaningful, Simon and Gilmartin took PERCEIVER one step further, subjecting it to the same board reconstruction experiment that the human players faced. To do this, however, required a number of changes to their first approach. They concluded that since short-term memory can hold only "seven plus-or-minus two" chunks at one time (see [Mil65]) and not more than one chunk could be transferred from short-term memory to long-term memory in a period of 5 seconds (recall the memory test), the information needed to reproduce the board after only 5 seconds must reside in short-term memory; as well, it must be that this information can be encoded in no more than 9 chunks. To comply with these limitations, they greatly modified PERCEIVER into a system called MAPP (Memory-Aided Pattern Perceiver).

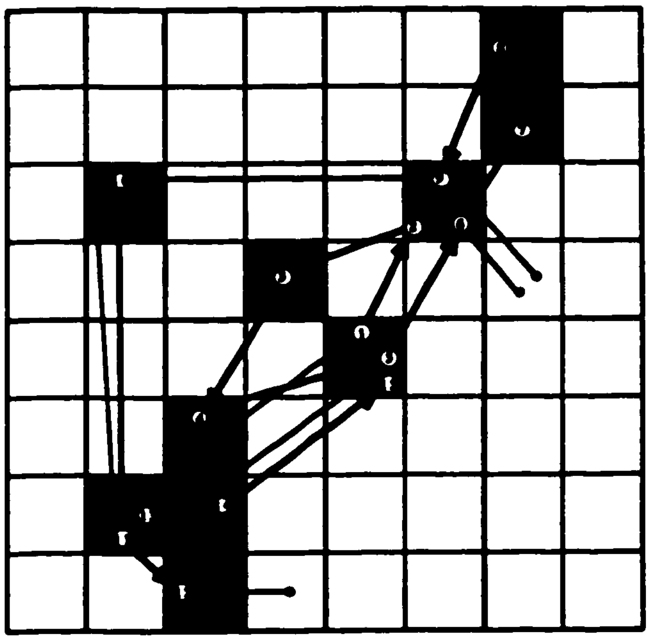


Figure 2.4 Simulated eye movements made by PERCEIVER.

The thick line represents eye movements and the thin lines represent relations noticed peripherally. The squares occupied by the most active pieces are shaded.

Two years previous to their work, Gregg and Simon [Gre67] developed a theory of discrimination (built from a system called EPAM - Elementary Perceiver and Memorizer [Fei61]) to explain how the encoding of chunks takes place (they actually worked with rote-verbal learning of nonsense syllables but the principles were the same). In short, the theory states that when stimuli (or chess pieces) match in certain relations or characteristics of previously recognized stimuli, they are replaced in short-term memory by a single chunk; this chunk, with which is usually associated a label or name, acts as a pointer into the location in long-term memory where the actual components (or pieces) are stored. Depending on the complexity of the chunks, those in short-term memory may actually point to other chunks which, in turn, point to the components of the chunk. This hierarchical approach allows the human to effectively "store" the 20 odd pieces of the position in short-term memory. Interestingly, it is usually the case that many pieces are contained within more than one chunk. This redundancy helps increase the number of relations between the pieces, leading to more accurate recollections of the positions. As De Groot outlined, the average number of pieces within each chunk is somewhere between 3 and 4, while the more "familiar" or frequently encountered chunks have as many as 6 pieces [DeG65]. It should now be clear how the master could reproduce positions with 93% accuracy.

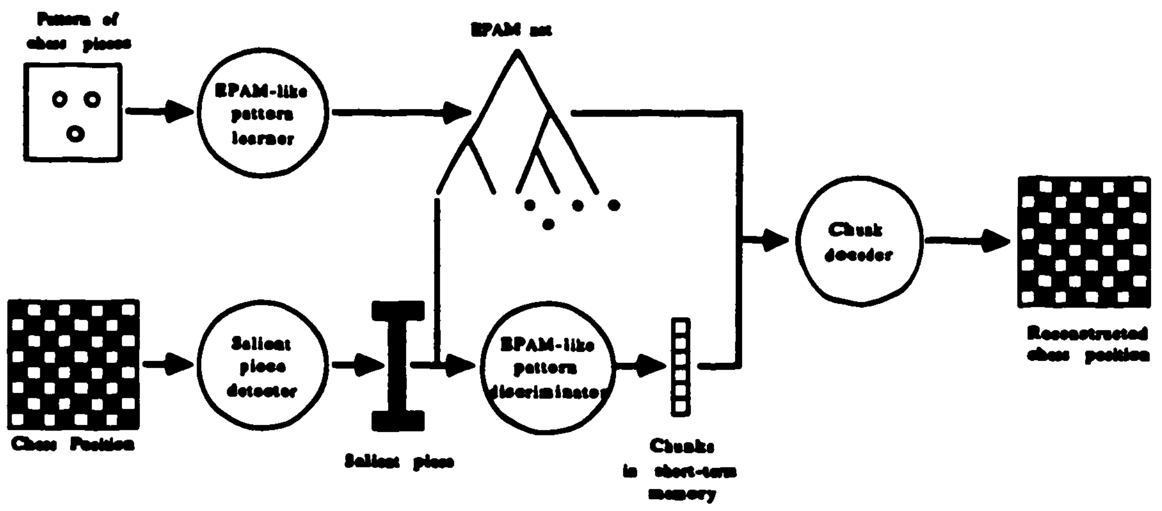

Pattern of chess pieces
EPAM-like pattern learner
EPAM net
Chunk decoder
Reconstructed chess position
Chess Position
Salient piece detector
Salient piece
EPAM-like pattern discriminator
Chunks in short-term memory

Figure 2.5 A schematic representation of the principal components of MAPP.

Using this theory of discrimination, Simon and Gilmartin designed their MAPP system around a learning component and a performance component (see Figure 2.5); this, no doubt, helped Simon develop his learning model, outlined in the previous chapter.

The learning mechanism was responsible for simulating "long-term memory" in which the chess chunks could be stored; the performance component, on the other hand, had to detect the salient pieces in the position, determine which patterns were present on the board and store their labels in short-term memory, and later decode these labels to reconstruct the board (for a more complete description of MAPP, consult [Sim73b]). To provide comparison with human performance, MAPP was tested on the same positions as were the humans. However, its repertoire of patterns numbered only slightly more than a thousand. Table 2.1 illustrates the results.

Percentage of Pieces Correctly Placed			
	Master	MAPP	Class A
5 positions*	62	43	34
9 positions†	81	54	49

* Positions after the 20th move
† Positions where the player to move can secure a decisive tactical advantage

Table 2.1 Percentage of pieces correctly placed by a Master, a Class A player and MAPP.

Not surprisingly, with such a small collection of chunks, MAPP did not perform as well as the master players (the masters placed about 50% more pieces correctly than MAPP) but functioned better than the Class A players. More importantly, though, MAPP missed only 30% of the pieces that the masters missed, indicating that MAPP's patterns were not unlike those of the masters.

From their experiments, Simon and Gilmartin found that not all patterns occur with equal frequency; as they pointed out, it is most likely a highly skewed distribution (e.g. harmonic distribution), similar to the distribution of words in natural language. Consequently, they attempted to estimate the number of patterns required for perfect reconstruction.

As an example, MAPP reproduced a board 20 moves into the game with 55% accuracy using a knowledge base of 1144 chunks. If π_{20} represents the number of patterns required for 100% accuracy on boards after 20 moves, then (based on the harmonic distribution),

$$\frac{\log \pi_{20}}{\log 1144} = \frac{1}{.55}$$

Solving for π_{20} we get 363,000. In addition, boards after 10 moves were reconstructed with 73% accuracy. Carrying out a similar calculation we get $\pi_{10} = 15{,}500$.

It would seem that the more patterns one has, the better one's reconstructive abilities. In particular, MAPP was tested on the same positions using 894 chunks and 1144 chunks. In these test positions, MAPP placed 160 pieces correctly with the smaller knowledge base compared with 168 using the larger one. The ratios of the knowledge base sizes and the number of correct pieces seem to match, as well: log 1144/log

894 = 1.07, 168/160 = 1.05.

The important result to recognize here is not the actual numbers but rather the trends that are evident. The more chunks a player has at his/her disposal, the more accurate he/she will be in reproducing a chess position from memory, which, of course, means the more positions the player will recognize as being similar. However, due to the highly skewed nature of the frequency with which these patterns occur, it is probably the case that doubling the size of one's chunk knowledge base will not double one's reconstructive abilities - the law of diminishing returns begins to take affect. For example, it may be that 10,000 patterns will yield 80% accuracy but to achieve an additional 19% may require 40,000 new patterns. Hence, the effort required to create these 40,000 patterns may not be worth the 19% increase in performance.

2.5. Some Experiments

Since this chunking hypothesis was first developed, a number of experiments have been conducted. Bratko, Tancig and Tancig [Bra86], for example, sought to devise a method for detecting positional patterns in chess, similar to what Chase and Simon had done. What the trio wished to do was derive chunks based solely on psychological studies of the master's behaviour, not artificially define them as Berliner had done using his CHUNKER program [Ber83]; their method required the use of probability and statistics.

They divided their twenty subjects into two groups: one group contained players all rated over 2300 (four of which were grandmasters) and the other group had players rated between 1800 and 2100. Each of the two groups were shown 24 middle-game positions for periods of 2, 5 and 9 seconds and, as in Chase and Simon's experiment, were asked to reconstruct them.

Their experiments measured two concepts: a *reconstruction factor* and *collective reconstruction* The *reconstruction factor* (i.e. the success of reconstruction) was defined to be,

$$F = \frac{P \cap R}{P \cup R}$$

where P is the set of pieces in the original position and R is the set of pieces in the reconstru[illegible] positi[illegible] The term *piece* also includes location. As a result, not only is the player rewarded for placing the pieces on the correct squares but also is penalized for placing pieces on the wrong squares, to eliminate guessing; the experiments conducted by Chase and Simon and De Groot did not include location in this manner.

A *collectively reconstructed* position is the position that results from the set of the most frequently placed pieces taken from the subjects' reconstructed positions. In other words, a specific piece rests on a specific square in the *collectively reconstructed* position only if the majority (some threshold, typically 50%) of the subjects' reconstructed positions contained that piece on that square.

Based on these two tests, Bratko et al. were able to define the derived chunks as follows: if *R* is the set of pieces in the *collectively reconstructed* position and *Sub(x)* is the set of subjects who placed piece *x* into their reconstructed position, then a derived chunk is each subset *C* of *R* such that

(i) S(C) = |intersection over C of Sub(x)| > T2 * N

and (ii) there is no C' which also satisfies this criterion such that $C \subset C'$.

Here, *T2* is some threshold between 0 and 1 and *N* is the number of subjects.

Indeed, this measure could produce chunks that the subjects are not actually using. Nevertheless, their results are rather interesting. The mean *reconstruction factor*, F, was between 0.44 and 0.78 (1.00 is a perfect score, of course) for Group 1 players but only 0.15 and 0.36 for those in Group 2. Table 2.2, taken from [Bra86], shows the mean reconstruction factors for each piece-type; note the relative importance of the pawns.

Using three differents exposure times, Bratko et al. found that 2 seconds was the most appropriate time frame for chunk detection whereas the 9-second exposure allowed the subjects (more notably those in Group 1) time to "think" about the position and to examine attack and defense properties, as well as to pick out features specific to the position.

The *collective-reconstructed* experiment illustrated some fascinating results, as well. Figure 2.6 shows two examples of the experiment. It was found that the mean reconstructive factor for these positions

F-values by piece-type		
Piece-type	Group 1	Group 2
King	0.785	0.354
Pawn	0.565	0.255
Queen	0.464	0.143
Rook	0.462	0.274
Bishop	0.439	0.082
Knight	0.383	0.107

Table 2.2 Mean reconstructive factors over all positions and for each group by piece-type.

was significantly higher than the average individual performance - 0.68 versus 0.55. Remarkably, the average number of derived chunks per position turned out to be 7.54 (recall the "seven plus-or-minus two" rule).

As one can see, the results generally concur with the hypothesis proposed by Chase and Simon; i.e. the stronger the player, the better his reconstructive abilities. In addition, their methods readily derived chunks that seem "quite natural" [Bra86]. However, properly setting the thresholds, *T1* and *T2*, remains to be rigorously determined.

Taking a different approach, Kopec et al. [Kop86] conducted experiments to see how the performance of humans and chess programs varied with time. A number of experiments were performed, two of which provide additional insight to the chunking hypothesis.

Pairs Rating Improvement	
Rating	Est. Improvement
1300-1599	250
1600-1799	100
1800-1999	200
2000-2199	250
2200-2399	100

Table 2.3 Rating improvement of subjects working in pairs compared to working individually.

The first experiment (called the Pairs Experiment) was designed to see if humans performed better in pairs than as individuals. Results from [Bra86] suggested that this is true. From using a sample of fifty-eight positions, it was shown that the rating for a pair was typically in the category above that for each of the individuals for the pair; e.g. two Class C players would perform as well as a Class B player. Table 2.3 illustrates their findings; the average improvement seems to be near 200 rating points. Using the F-test, they concluded that, with 99% confidence, the ratings were statistically improved.

One of the other experiments used, called the *Time Sequence* Experiment, was designed to see how human performance varied with time. Subjects, rated from 1600 to 2300, were shown four sets of 10 positions for periods of 30 seconds, 1 minute, 2 minutes, 4 minutes and 8 minutes. Results from the 4 and 8 minute test indicated that those players rated 2000 and above scored as much as 20% higher than those rated below 2000; on the shorter test, however, the increase was less than 8%. This shows that, given enough time, stronger players can distinguish themselves from weaker ones, developing a greater understanding of the problem at hand; perception and recognition - that is the key.

2.6. Results & Direction

In any event, it should be evident that a master or grandmaster plays near perfect chess as a result of the thousands of patterns (somewhere near 50,000)[4] at his/her disposal and the only way to acquire such a large number is through hours and hours of practice. By the same token, it seems reasonable that if a chess program could somehow access these 50,000 patterns with their associated game plans, it might play significantly better chess and bring it one step closer to achieving grandmaster status.

By providing chess programs with grandmaster advice, we believe, is the only sure way of creating a world champion - one that rivals the top human players.

[4] Considering that highly literate people have vocabularies of 50,000 words or more, Simon and Gilmartin concluded this estimate to be reasonable.

Chapter 3

System Design

3.1. Introduction

For MACH to be considered a useful system, it must possess an effective similarity measure and a fast search procedure. Its main task is to quickly find the most similar positions in a large database to each one it encounters over the course of a game. Then, once these similar positions are retrieved, further tests must be conducted to see which are the best matches in terms of material balance, piece location, etc. In other words, through the use of similarity criteria, we wish to extract readily only a small subset (perhaps a hundred or so) of positions out of the millions in our database, so that more extensive filtering can take place.

In this chapter, we will detail the design of MACH which will include a discussion of its components and an outline of some options for an efficient search routine. To begin, we will describe the necessary qualities that our database of games (called the *GAMETREE*) must possess to be useful; one of the deciding factors in its design will involve its enormous size. The next item that will be considered is the development of efficient access methods for this database. Creating what we call a *FEATUREndx*, developed from our similarity measures, will help in this regard. The creation of this *FEATUREndx* will be briefly outlined, saving the details of its contents for chapter 4; this will be followed by a survey of related work on partial-matching algorithms and an analysis of three approaches to the partial-matching problem. To summarize the chapter, a diagram illustrating MACH's file creation procedures will be presented.

3.2. The GAMETREE

A key element that has lead to our development of a chess experience teacher is the ability to have grandmaster chess games easily accessible in machine readable format. Suddenly, all the chess knowledge one would ever need is contained in one package; the trick, however, is to extract this knowledge properly and use it to one's advantage. Consequently, the structuring of the *GAMETREE* becomes of paramount importance.

Given that there are, on average, 80 positions per game and approximately 6000 new grandmaster games are played each year, it is immediately obvious that there exists almost a half million positions each year! In actual fact, the number of new positions is probably closer to 350,000 since many games open in similar fashions. Notwithstanding, one can easily forecast a database of millions of positions if games are included for the last half century. Clearly, it is evident that efficient storage of these positions must be of great concern. Moreover, we must keep in mind the principles involved to attain *cognitive economy* [Lea79].

3.2.1. A Data Structure

A chess game proceeds in a serial fashion, with each side alternating moves. Hence, one can construct *position-j* from *position-i* (for $i < j$) by executing the moves after *position-i* that resulted in *position-j*; i.e. given any position, one can compute any other following position by simply knowing the moves involved. Since the starting position of a game is always the same, all one requires is the set of moves of a game to construct every position that occurred in that game. This suggests that it is not necessary to remember positions of games, only the moves. So our database of positions need only be a database of moves and pointers linking these moves. This fact helps our concern for storage efficiency - a 6-bit move is certainly preferable to a 64-square chess board.

Another notion which can help reduce storage use is to have the games represented with no repeated positions. A common opening, such as *Pd2d4* (written in algebraic notation), need not be stored for each game in which it resulted; rather, a single occurrence is all that is required and subsequent moves (from different games) can be linked to this common opening move. This idea can be repeated throughout the database, as well. Unfortunately, identical positions which occur at different points throughout individual games (e.g. at ply #10 in game #24 and ply #34 in game #455) will be repeated. Linking these moves is possible, certainly, but the effort expended in maintaining these links, we believe, is not cost-efficient for our test system. Consequently, only those identical positions which occur at the same point in different games are represented only once.

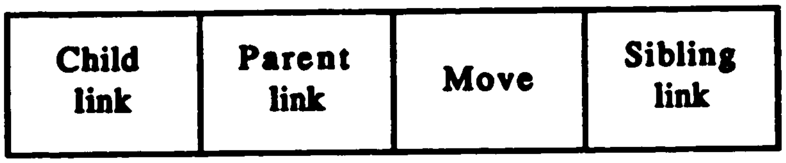


Figure 3.1 Data structure for a node in the *GAMETREE*.

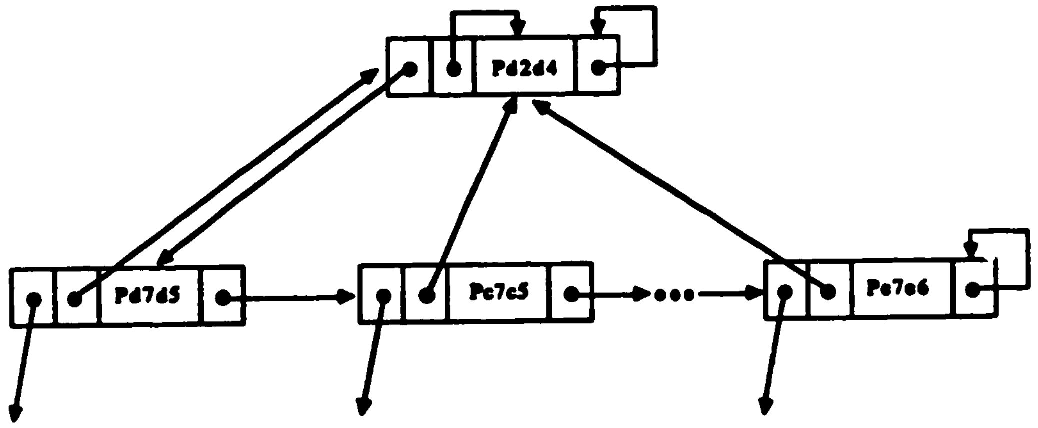


Figure 3.2 A typical *GAMETREE* configuration.

Given that only moves are stored for each game in our database, they must be properly linked to each other in order to reconstruct each of the positions. As a result, each entry (or node) in our database should contain the move played along with a pointer to the previous move (its parent node) and a pointer to the next move (its child node). Once all the links are in place, the database of games becomes simply a large tree (hence, the name *GAMETREE*). As will be apparent later, another pointer is required linking each node with its sibling. Hence, all moves from a game will be linked using a doubly-linked list while all child nodes of each parent will be linked using a singly-linked list. Figure 3.1 illustrates our data structure for each node in the *GAMETREE* and Figure 3.2 describes a simple *GAMETREE* configuration. As illustrated, each node occupies only 14 bytes of storage.

3.2.2. Adding Games

```
PROCEDURE ADD_A_GAME()

BEGIN
   IF the GAMETREE is empty THEN
      ADD_&_LINK()
   ELSE
      Position at root_node in GAMETREE
      WHILE current_move matches current_node DO
         Advance to next_move in game and next_node in GAMETREE
      END WHILE
      ADD_&_LINK()
   ENDIF
END
RETURN

PROCEDURE ADD_&_LINK()

BEGIN
   WHILE there are moves left in the current_game DO
      Create a new move_node
      Add (next) move from current_game to current_node
      Link current_node to parent_node
      Link parent_node to current_node
   END WHILE
END
RETURN
```

Figure 3.3 Pseudo-code describing how to add a game to the *GAMETREE*.

With this data structure intact, adding moves (and games) to our *GAMETREE* becomes relatively straightforward. Figure 3.3 illustrates our *ADD_A_GAME()* and *ADD_&_LINK()* procedures. Of course, games must first be added to the database before positions can be retrieved.

There is one major flaw in this *ADD_A_GAME()* routine that may not be obvious. To illustrate, let our *GAMETREE* contain the nodes of the sample configuration of Figure 3.2 and let the game we wish to add contain the opening sequence of moves, "*Pd2d4*, *Pc7c5*, *Pc2c4*...". Since the *GAMETREE* in Figure 3.2 is not empty, we immediately proceed to check for identical openings. The opening move of the game, *Pd2d4*, matches with the root node of the *GAMETREE*; so the next move of the game, *Pc7c5*, is read and the child pointer of the root node in the *GAMETREE* is followed. At this point, the current node, *Pd7d5*, does not match the current move in the game. However, its sibling pointer does point to the move *Pc7c5*. Consequently, the *WHILE* in *ADD_A_GAME()* would fail to find this fact and proceed to duplicate a move unnecessarily. To correct this, all nodes at the level where the first match failed must be examined. If no more nodes exist at that level, only then can the checking stop. As a result, our *ADD_A_GAME()* routine must be modified into the algorithm described in Figure 3.4.

```
PROCEDURE ADD_A_GAME()

BEGIN
  IF the GAMETREE is empty THEN
    ADD_&_LINK()
  ELSE
    Position at root_node in GAMETREE
    WHILE current_node matches game_move OR sibling_ptr is not NULL DO
      IF match THEN
        Advance to next_move in game and node in GAMETREE
      ELSE
        Advance to next_sibling_node in GAMETREE
      ENDIF
    END WHILE
    ADD_&_LINK()
  ENDIF
END
RETURN
```

Figure 3.4 Improved pseudo-cr 'e describing how to add a game to the *GAMETREE*.

It should now be obvious why a sibling pointer for each node is required: we need a way of linking a variable number of nodes at each level in the tree.

3.2.3. Constructing Positions

Now that we have routines to add games to our *GAMETREE*, we must be capable of constructing a chess position, given an arbitrary node in the tree (since it is really the positions we will be matching). This task is actually easier than adding games simply because each node is linked to its parent node and its child node. Starting at a given node, *s*, we recurse back up the tree using the parent pointers until the root (the starting position) is reached; then we traverse back along the same path, making each move as we go, until node *s* is reached.

```
PROCEDURE CONSTRUCT_POSITION()

ARRAY moves[]

BEGIN
   total_moves = 0
   WHILE parent of current_node <> NULL DO
      current_node = parent_node
      total_moves = total_moves + 1
      moves[total_moves] = current_move
   END WHILE
   FOR i = 0 to total_moves DO
      MAKE_MOVE(moves[i])
   NEXT
END
RETURN
```

Figure 3.5 Pseudo-code describing how to construct a position.

Figure 3.5 outlines the *CONSTRUCT_POSITION()* algorithm in detail. As each move is encountered, it is stored in an array for rapid processing; once the root is reached, we simply process the array in reverse. The *MAKE_MOVE()* routine mentioned in the algorithm is a routine that any chess program would use to make a move; this allows us to maintain compatibility and eliminate duplicate coding.

3.3. The FEATUREndx

It is should now be clear that once we are positioned at a node in the *GAMETREE*, we can easily reconstruct its corresponding position. Unfortunately, finding a set of similar positions is not so easy. PP In order to match positions, we must be able to discern the important features of them. Consequently, it seems reasonable to assign some sort of *feature tag* to each position in the *GAMETREE* that would indicate the important features of the position. We could create an index file or group of index files, referred to as the *FEATUREndx*, whose records could contain a feature tag of a position along with a pointer to the corresponding position in the *GAMETREE*. Then, given any position, we could compute its feature tag, search the *FEATUREndx* for identical or similar tags, and follow the pointers into the *GAMETREE*; following that, for each candidate position, a more thorough similarity computation could be performed to find the best match.

At a conceptual level, the task is very simple; implementing efficient procedures to carry out the task is not. It is apparent that two problems must be addressed:

• calculating this *feature tag*

• rapidly searching the *FEATUREndx* for matches.

From our discussion of chunking in the previous chapter, it seems appropriate that determining what chunks existed in a position would produce a reasonable measure indicating the important features in the position; our feature tag could simply be a list of the existing chunks. So, given a position in an actual game, we could *chunk* the position, creating a list of its chunks, and then scan the *FEATUREndx* for a similar list - a position does not necessarily require the same chunks to be considered similar.

For now, let us assume that we can accurately compute the important features of a position and readily store it in the *FEATUREndx* so that we may turn our attention to examining ways of retrieving similar lists from this, presumably, huge file (or set of files).

3.3.1. Accessing the FEATUREndx

Our *FEATUREndx*, as already outlined, is the main link into our *GAMETREE*. Given a current position, we wish to develop a method to scan this index file as quickly as possible, retrieving those links to positions in the *GAMETREE* which are likely to be similar. Each *FEATUREndx* record contains a list of the chunks that are present in a particular position in addition to a pointer to that position in the *GAMETREE*. Our method should pick out those records whose chunks match some subset (proper or improper) of the chunks of the current position (i.e. the one for which we want similar positions). Requiring only a subset of the chunks to match is, ultimately, what makes finding an efficient search procedure so difficult.

We require only a subset of chunks to match (as opposed to an exact match) for a number of reasons. The most significant reason involves our addition of knowledge to MACH. Our chunks are defined by an master-level chess player using a simple chess language (this language is described in the following chapter). To pick out the chunks, MACH needs to know what to look for; however, our chunk descriptions, since defined by a human, are subject to errors and omissions. By not restricting ourselves to exact matches, we can overcome some of these problems. In addition, since we have not yet developed a method for determining a chunk's importance in a position, we do not know which chunk will cause a position to be dissimilar. By setting a tolerance for our matching, we can, in some ways, circumvent the importance problem.

3.3.2. The Partial-Matching Problem and Related Work

The general partial-matching problem can be stated as follows: given some key, k, with n attributes (or characters, for simplicity), we wish to examine an index file (or files) containing r records ($r >> 0$), retrieving those records that match m attributes of k ($m \leq n$). The relative positions of the m matching attributes is immaterial; e.g. the string "2679" should match the key "12689", if $m = 3$ since the 2, 6 and 9 values match.

Unfortunately, the current literature surrounding this partial-matching problem is far from complete. For the most part, only special cases of the general problem are touched upon. Rivest [Riv76] developed hash-coding and tree-search algorithms for searching a file to find all occurrences that match *s* letters of a *k*-letter key. For instance, his algorithms were able to quickly find all records in a file that match the word "W*RD", where the "*" is a "don't care" character. Upon first glance, this appears to solve our problem. However, his method requires the *W*, *R* and *D* to remain in exactly the 1st, 3rd and 4th positions. While we could easily find the words, *WORD* and *WARD*, we could not find the word *SWORD*, since the word does not begin with a W.

Donald Morrison [Mor68] developed his PATRICIA system to retrieve information coded in alphanumeric. An attractive feature of his approach is that the effort required to find a key is dependent solely on the length of the key; i.e. even if the size of the search file doubles or triples, the search time to retrieve a particular key remains the same. His unique use of indices linking beginnings and endings of words as well as substrings of words falls short of finding words within words, unless the word you are looking for begins with the correct letter. Fredkin's use of trie memory [Fre61] lacks the same necessary ingredient; if the string "BEAD" is stored in his trie memory configuration, it is possible to find the string "BEA" but not the string "HEAD".

A recent publication by Blumer et al. [Blu87] goes farther but their function *find(w)* which returns the longest prefix of *w* that occurs in a database does just that - finds only the prefix.

3.3.3. Three Approaches

Faced with this dilemma, we sought a simple path to, at least, demonstrate that MACH works with *Phoenix* and works reasonably well. Though not adequately solving the general search problem, we will examine three different data structures. They are (1) inverted files, (2) tries, and a hybrid of these we will call (3) restricted enumeration. Note that a more "efficient" data structure or algorithm would not alter the correctness or validity of our approach; rather, only an increase in execution speed and/or a reduction in storage would be realized.

3.3.4. Inverted Files

This first method is, perhaps, the most naive approach of the three presented. However, its simplicity allows for minimal programming effort, producing easy implementation and quick results.

There are a number of definitions describing inverted files in the literature, some being more specialized than others. For our purposes, though, we define the following: for each chunk (1,...,c), we construct a file, each containing pointers to all those positions in the *GAMETREE* in which the chunk exists. For example, the records of *file_i* would simply consist of pointers to all those positions in the *GAMETREE* in which chunk i was present.

If the position in question for which we want similar positions contains chunks (i, j, k), we simply merge *file_i*, *file_j*, and *file_k* and find all records in common (i.e. retrieve any duplicate pointers to the *GAMETREE*). Since there are c chunks and p positions in the *GAMETREE*, there are a maximum of cp entries. As a result, doubling the number of chunks or positions merely doubles the amount of storage needed, in the worst case. Unfortunately, as the size of these files increase, the time required to merge f files and search for duplicates also increases.

It is obvious this use of inverted files is unacceptable in a real-time application since response time is the most critical factor. Conceivably, p could equal 1,000,000 and c could be 10,000 or more. Merging five or six files of this magnitude and searching for duplicates would certainly yield unacceptable performance, at least as far as a chess program is concerned. Nevertheless, this approach is guaranteed to find all positions in common - a more efficient implementation could not improve on this result. Moreover, its attractiveness lies in its minimal programming effort and linear storage requirements. For test purposes, though, we can keep p and c small.

3.3.5. Tries

As a data structure, tries were introduced almost thirty years ago. A trie is a form of m-way tree ($m \geq 2$) which stores records at its leaves. The order in which the records are stored has a significant effect on the search time. Unlike the inverted file approach, the benefit of this type of structure lies in its fast retrieval times; notwithstanding, what we gain in speed, we lose in storage requirements. Specifically, this structure grows exponentially, $O(2^n)$ in the worst case.

To demonstrate how we can adapt tries to our scenario, consider a simple example with c, the number of chunks, equal to 4. For each position in the *GAMETREE*, we know (hopefully) there exists a non-empty list of chunks. To help matters, we can insist that this list is to be kept in sorted order and each entry in the list is to be unique (i.e. chunk i can occur at most once). With these constraints, we arrive at the tree in Figure 3.6. The circles represent chunks and the squares represent pointers to positions in the *GAMETREE*.

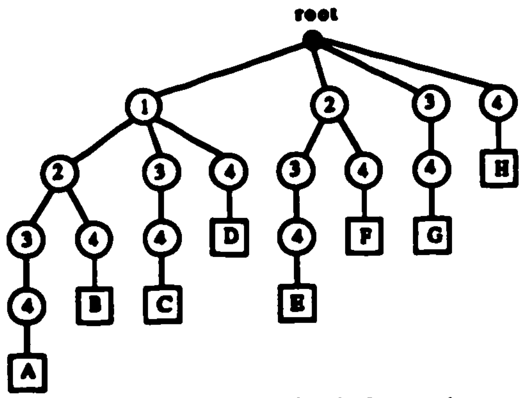


Figure 3.6 A simple trie for c = 4.

Thus, position E contains chunks (2, 3, 4). Note that the sons of each node are always greater than the node itself. This is a result of the sort restriction. In general, for each node i at level l, the maximum number of sons at level $l+1$ is c-i, where c is the number of chunks.

To find positions containing chunks (i, j, k), we can immediately ignore those paths from the root beginning with values less than i. However, if we wish to find positions containing only 2 of the 3 chunks listed, our search procedure is a bit more complicated. In particular, we must look for positions containing

(*, *i*, *j*), (*, *i*, *k*), (*, *j*, *k*), (*i*, *, *j*), (*i*, *, *k*), (*j*, *, *k*), (*i*, *j*, *), (*i*, *k*, *), and (*j*, *k*, *), where "*" is a don't-care symbol - it can assume any value from 1 to *c* as long as the list is in sorted order. Though this procedure can become quite involved for positions with six or seven chunks, it is still superior to the inverted file method in terms of speed. Unfortunately, it is much more difficult to program and the storage needs are expensive.

To increase the number of chunks in our previous example from 4 to 5 means doubling the number of nodes in the worst case (see Figure 3.7). In actual practice, this number is, likely, smaller given that some combinations might be impossible due to the semantics of chess; e.g. chunk 3 may never occur with chunk 1 due to the way in which each are defined - hence, any (1, 3, ...) combinations will not be represented in the tree (refer to the branches shaded in the figure). Nevertheless, it is the potential explosive growth that discourages practical use of this approach. For values of *c* in excess of 20, our trie becomes unmanageable.

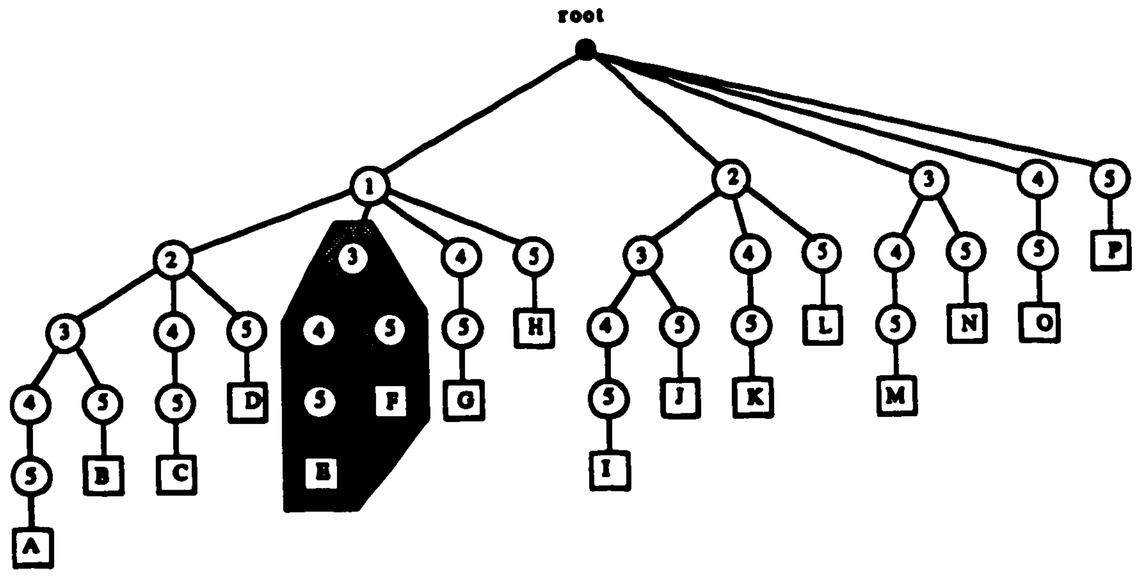


Figure 3.7 A trie with c = 5.
The shaded branches would not occur if any (1,3,...) combinations were impossible.

3.3.6. Restricted Enumeration

It appears there is no easy way to stop the exponential growth of the problem. But if there were some way to inhibit this growth, it may be a viable alternative. By capitalizing on some of the domain-dependent properties of the problem, we can achieve reasonable file sizes and relatively quick access times; as a result, we arrive at a method called *restricted enumeration* which will be described below.

The one feature that we know remains small is the number of chunks per position. Simon and Chase [Sim73a] have already shown through their memory experiments that the maximum number of chunks a GM recognizes on a particular board position is in the neighbourhood of 7, plus or minus 2. This, as you recall, is consistent with the notion of a limited short-term memory. By restricting the number of chunks/position to 9, we have accounted for a great majority of positions while limiting the number of combinations possible. It is true that restricting the number of chunks/position will shrink the size of the tries, as well. However, we will show that our hybrid takes advantage of both tries and inverted files to yield a better solution.

Another parameter we can reduce is the number of files needed. We know the some chunks are definitely instances of others (recall the PERCEIVER implementation of the previous chapter). As a result, it is probably worthwhile to take a hierarchical approach; thus, each chunk will belong to some equivalence class of chunks or chunk-class (hereafter called a *c-class*). Each *c-class* would contain chunks having similar descriptions (e.g. castled patterns) and would be specified by a human expert. By doing this, we can, hopefully, reduce the number of different files from tens of thousands to 50 or 60. So now we can say a position contains c-classes instead of chunks.

To continue, let the total number of c-classes equal c and the number of c-classes/position equal r. Hence, the number of possible combinations is $\begin{bmatrix} c \\ r \end{bmatrix}$. In addition, we will impose two other constraints: (1) all positions must have no more than 9 c-classes and (2) given a position with r c-classes, a potentially similar position is one which matches at least r-1 of these classes. Chapter 5 will illustrate some test results indicating why r-1 was chosen as opposed to r or r-2. Again, because our chunk descriptions are subject to errors and omissions, we do not want to restrict ourselves to perfect matches. We believe, though, that these constraints will give us the maximum return for our effort. Our intuition tells us that few positions will contain more than 9 c-classes (assuming our chunks are adequately constructed and implemented).

To get a handle on the values of $\binom{c}{r}$, we refer to Table 3.1 (for obvious reasons, we chose our c values to be powers of 2). You can easily witness the exponential growth: by doubling c from 16 to 32 with $r = 4$ increases the number of combinations by a factor of almost 20 and by a factor of over 800 with $r = 8$; doubling r, on the other hand, has a much less effect.

3.3.6.1. Setting up the Index Files

For each of the 9 values of r, let us assign a file, the maximum number of records that could be used in each corresponding to the entries from Table 3.1. For example, if we limited ourselves to 16 c-classes, then our file representing the positions with exactly 4 c-classes/position could point to a maximum of 1820 positions, one record for each of the 1820 combinations. For those positions that mapped into a record already occupied, they would be stored in an overflow file. Conceivably, we could have a file of arbitrary size and place all collisions into the next available spot; however, we wish to search each c-class file as quickly as possible and duplicates would degrade the search time.

For the next step in our description, let us assume that there are 900,000 positions (representing 10-15,000 games) we wish to place in our *GAMETREE* and that these are evenly distributed across our 9 files; i.e. there are 100,000 positions that contain exactly 1 c-class, 100,000 positions that contain exactly 2 c-classes, and so on.

Let us now consider the values in Table 3.2. For values of c and r in Table 3.1, based on 900,000 positions, we have determined the percentage of the records occupied in each of the 9 files along with the average number of collisions per record (this number assumes each c-class has the same probability of occurring, namely $1/c$, which, in actual practice, is probably not true since certain patterns in chess occur much more frequently than others).

Combinations			
r	c		
	16	32	64
1	16	32	64
2	120	496	2016
3	560	4960	41,664
4	1820	35,960	635,376
5	4368	201,376	7,624,512
6	8008	906,192	74,974,368
7	11,440	3,365,856	621,216,192
8	12,870	10,518,300	4,426,165,368
9	11,440	28,048,800	27,540,584,512

Table 3.1 Number of $\left[{c \atop r}\right]$ combinations.

From Table 3.2, we can see that as more c-classes are present, the less the number of collisions that occur. More importantly, though, with a relatively large number of c-classes, we can achieve a greater granularity for better filtering of unwanted, dissimilar positions, which is the sole reason why we need a feature index.

File Status						
	c					
	16		32		64	
r	% full	avg # collisions	% full	avg # collisions	% full	avg # collisions
1	100.0	6250.0	100.0	3125.0	100.0	1562.5
2	100.0	833.3	100.0	201.6	100.0	49.6
3	100.0	177.6	100.0	19.2	100.0	2.4
4	100.0	54.9	100.0	2.8	15.7	0.2
5	100.0	22.9	49.7	0.5	1.3	0.01
6	100.0	12.5	11.0	0.1	0.1	0.001
7	100.0	8.7	3.0	0.03	0.02	0.0002
8	100.0	7.8	1.0	0.01	0.002	0.00002
9	100.0	8.7	0.4	0.004	0.0004	0.000004

Table 3.2 Percentage of records occupied and average number of collisions.

To continue, we shall take this analysis one step further. We now want an efficient way of storing a c-class combination list so that it is easily calculated and can be quickly found.

There are two easy ways to represent a string of numbers such as (2, 4, 5, 6, 14); think of this string as a list of c-class identifiers that could be stored in the file containing exactly 5 c-classes. The first method assigns a bit for each different number in a certain range, combines these bits into a string, and sets each corresponding *i*th bit named in the string; we will call this method *simple-bit-assignment* or *SBA*. The second method determines the binary representation of each element in the string, stores it in an *n*-bit block, and combines each of the *m* elements into one *mn*-bit string; this method will be referred to as *bit-shifting-assignment* or *BSA*). For example, using simple-bit-assignment on the above string of numbers (assuming a range of 16 different numbers were allowed) yields the following bit representation: 0101110000000100 - here, the 2nd, 4th, 5th, 6th and 14th bits are set. Using bit-shifting-assignment we get a totally different representation: 0010010001010[illegible] here, since four bits are needed to represent a number from 0 to 15, each group of four bits represents one of the numbers; e.g. the first four bits, 0010, corresponds to the number 2, the next four, 0100, represents the number 4, and so on.

For this simple example, 16 bits are needed to represent a string of 5 numbers using *SBA* whereas 20 bits are needed to represent the same string using *BSA*. Obviously, we would chose the method which requires the least amount of storage. However, the size of the range can make *BSA* more space efficient than *SBA*. Table 3.3 illustrates our point. When we have as many as 64 different numbers in our string, the *BSA* method outperforms *SBA* in all respects.

Bits needed to represent *r* integers from *0* to *c-1*						
	c					
	16		32		64	
r	SBA	BSA	SBA	BSA	SBA	BSA
1	16	4	32	5	64	6
2	16	8	32	10	64	12
3	16	12	32	15	64	18
4	16	16	32	20	64	24
5	16	20	32	25	64	30
6	16	24	32	30	64	36
7	16	28	32	35	64	42
8	16	32	32	40	64	48
9	16	36	32	45	64	54

Table 3.3 Comparison of storage needs for SBA and BSA.

3.3.6.2. Using the Index Files

Given 900,000 positions in our *GAMETREE* and a representation for coding each of the c-class combinations, we wish to get a feel for the sizes of these index files. Since each of the nine files can experience overflow, it is necessary to associate an overflow file for each for a total of 18 files. We will use the naming convention *file_r* and *file_ro* (for each of the 9 values of *r* in Table 3.1) for each data file and their corresponding overflow file, respectively.

Each record in each of the 9 data files will consist of a combination list of c-classes in a certain position along with a pointer to either that position in the *GAMETREE* or a record in the overflow file or both. Each overflow record will contain a pointer to a position in the *GAMETREE* as well as a pointer to the next collision that occurred from the same c-class list.

Initially, records are added to each data file one after another using some form of an insertion sort; for large values of *c*, records are stored using the *BSA* coding. This method can become very slow for large files but it is used only to add positions, a task that is performed occasionally. On the other hand, since each file remains sorted, any record can be quickly retrieved using a binary search. If a record collides with an existing one during an *add* operation, it is simply written to the next location in the overflow file and the pointer from the data file would hold the location of this overflow record; multiple collisions from the same record would subsequently be linked together in the overflow file. Of course, the order in which records are added to the overflow files is immaterial since each group is linked by pointers.

Since the data files are sorted, finding matches is straightforward. If a given position contains 6 c-classes, a binary search would be conducted on *file_6* to find the specific c-class combination list of the given position. If it were found, the pointer attached to the matched record would be followed into the *GAMETREE* where further similarity tests, which will be outlined in the next chapter, could be used on the candidate position found there. As well, if the overflow pointer attached to this matched record were not null, meaning multiple positions contained this c-class list, it, too, would be followed into the overflow file and the list of these overflow pointers would each be used to trace through the *GAMETREE* to find additional candidate positions. If we required only *r-1* c-classes to match, *file_5* could be searched in a similar manner to match the subsets of 5 c-classes of the c-class list of the position.

To determine the maximum file sizes of each of these 18 files, we will consider them separately, for each of the 3 values of *c* in Table 3.1 (16, 32, & 64). The sizes are determined according to the following simple formulae:

$$file\_r\_size = max\_\#\_records * (c{-}class\_size + 2 * pointer\_size)$$
$$file\_ro\_size = max\_\#\_records\_in\_file\_r * avg\_\#\_collisions * 2 * pointer\_size$$

Pointers generally occupy 4 bytes but may expand to 8 bytes when the number of positions grows beyond 4 billion (i.e. 2^{32}).

Table 3.4 illustrates the space requirements for the 18 files. For $c = 16$, note that the overflow file is considerably larger than the corresponding data file. This is due to the small number of different records that can be represented. On the other hand, for $c = 64$, the size of the overflow files is virtually 0 for *file_7o* to *file_9o*, meaning there are very few collisions; this is a result of a large number of combinations. The sizes of all of the overflow files are rounded up to be divisible by eight since each record is typically eight bytes long.

It would seem that the more c-classes you have the better. Not only have you increased granularity for weeding out dissimilar positions but also have made better use of storage space. The 3.5% increase in total storage from $c = 32$ to $c = 64$ is almost negligible. As well, using slightly more than 10Mbytes of storage to index 900,000 positions, we believe, is a reasonable tradeoff.

File Sizes (bytes)			
	c		
	16	32	64
file_1	144	288	576
file_1o	800,000	800,000	800,000
file_2	1080	4960	20,160
file_2o	800,000	800,000	800,000
file_3	5600	49,600	458,304
file_3o	800,000	800,000	800,000
file_4	18,200	395,560	1,100,000
file_4o	800,000	800,000	125,912
file_5	48,048	1,200,000	1,200,000
file_5o	800,000	397,272	10,496
file_6	88,088	1,200,000	1,300,000
file_6o	800,000	88,288	1072
file_7	137,280	1,300,000	1,400,000
file_7o	800,000	23,768	136
file_8	154,440	1,300,000	1,400,000
file_8o	800,000	7608	24
file_9	148,720	1,400,000	1,500,000
file_9o	800,000	2856	8
TOTAL	7,801,600	10,570,200	10,916,688

Table 3.4 File sizes for the 18 files that make up the FEATUREndx.

3.4. Current State of MACH

It is clear that each of the three methods discussed have their merits: inverted files are simple to test and implement, tries have fast retrieval times, and restricted enumeration makes efficient use of storage with reasonable search times (a binary search is all that is needed to search each of the files for the first occurrence of c-class list).

Our objective is to test whether MACH helps a chess program play better chess and to achieve results as quickly as possible. Consequently, we chose the inverted file approach since it was the easiest method to implement. Our hybrid would probably perform better but its superior performance would not improve the advice from MACH, only find it quicker. Indeed, an even more efficient search procedure, no doubt, can be found; so there is no point wasting time implementing a more advanced algorithm if MACH cannot provide useful advice in the first place. This search problem remains a topic of further study.

Below is Figure 3.8, illustrating the file creation procedures of MACH. MACH processes each game in the file *GAMES* using the *BUILD* program. This program first extracts the moves and stores them in a list. At the same time, each position is *chunked* (i.e. determine the chunks that are present) and the c-class in which each chunk belonged is added to a c-class list; this c-class information is part of each chunk's description. When all the moves have been gathered, the move list is added to the *GAMETREE* using the *ADD_A_GAME()* routine discussed previously and then, for each c-class, the appropriate c-class index files are updated to contain the pointers of the positions in the *GAMETREE* in which each c-class occurred.

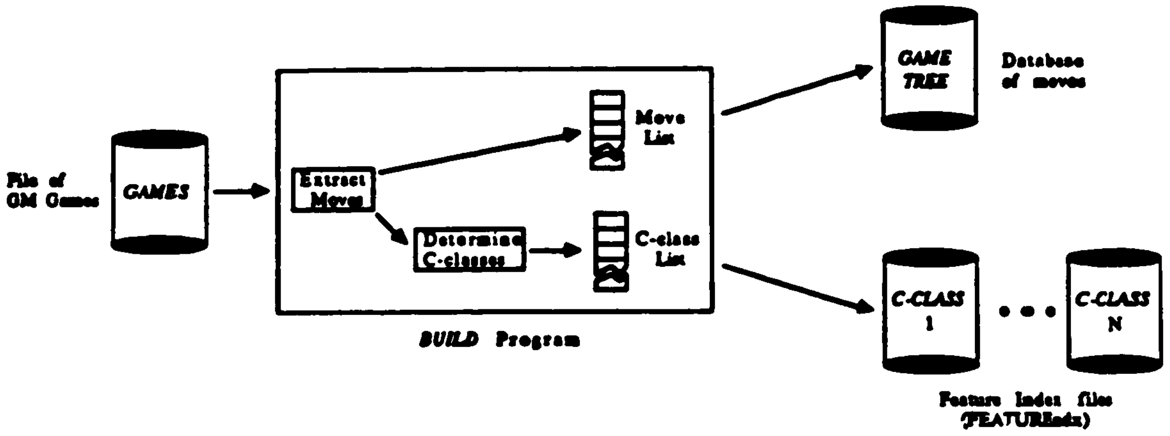


Figure 3.8 File creation procedures of MACH.

Chapter 4

Similarity and Chunking

4.1. Introduction

Until now, we have glossed over the details of how to *chunk* a position; i.e. extract its relevant features. Instead, we have concentrated our efforts on properly constructing the *GAMETREE* and developing a method to search through it for a desired group of positions. In this chapter, however, we will turn our attention to the difficult task of developing a reliable similarity measure - indeed, the issue at the heart of this project.

The study of cluster analysis in the field of statistics is able to supply an application-independent algorithm for measuring similarity; in fact, Van der Meulen discussed this in his development of proper weight assessments for evaluation functions [Van87]. Indeed, its methods are based in well-established theories but they lack the pieces of application-dependent knowledge necessary to handle the intricacies of chess. For example, two positions can be identical in every manner except for the position of one king, and that can be significant; on the other hand, the difference in only one pawn is usually not enough to consider the positions dissimilar. It is the relative importance of chess specific knowledge that discourages the use of numerical methods such as cluster analysis.

Consequently, given the psychological evidence that humans *chunk*, it will be useful to express a chess position in terms of those chunks in order to distinguish important features that will lead to the development of a similarity measure for the position. Hopefully, by doing so, a lot of irrelevant information will be eliminated from the position. To accomplish this effectively, however, will require considerable chess knowledge (of course, our *GAMETREE* is abundant in this knowledge but we must have a way to extract it). To follow the philosophy of Zobrist et al. [Zob73], we will abandon the use of expert programmers and head toward methods that allow chess masters to easily express this knowledge to us.

4.2. Language and Grammar Development

By far, natural language is the medium through which most humans are most comfortable communicating (chess masters, included). While the study of natural language understanding is certainly a complex topic in itself, we can, nevertheless, design a simple language specific to chess to capture much information (this may not be the most efficient way for the masters to communicate but alternate methods, such as a graphical interface, will be outlined later).

The quality and quantity of the features of the language will determine how useful and accurate the master-defined chunks will be. To help in deciding what features would be useful in a chunk language, we can, once again, draw on the results of Simon, Chase and Gilmartin and their PERCEIVER and MAPP systems. Recall that PERCEIVER was able to duplicate the eye movements of a master by simply adhering to the attack and defense properties of the pieces as well as to commonality of piece-color, type and proximity. Given these fundamental position features as a basis for the language, it is reasonable to assume that, for each chunk that can defined, there should exist some minimum set of pieces on specific locations that must be present for the chunk to be present; i.e. if pieces *p* and *q* are on squares *s* and *t*, then chunk *c* is present. By the same token, additional pieces, other than those in this minimum set, may be a part of a chunk and, if present, should be included in its makeup; their absence, though, would not affect the chunk's existence.[5] Consequently, our language should define a set of *required* pieces and a set of *optional* pieces necessary to form a chunk.

An additional feature which is necessary arises from our discussion of c-classes. We found it helpful to structure chunks logically in a hierarchical fashion, since many chunks can be defined in terms of others. For example, we could define *open_file* as a chunk whose definition would require no black or white pawns on a given file. In addition, a chunk called *file_control* could be defined whose makeup would include the conditions of an open file. Hence, our language should be flexible enough to allow one chunk to be defined

[5] Though these optional pieces are not truly utilized by our current system, their presence can be used to determine the pieces involved with each chunk, thereby discovering what pieces were missed by the chunking process.

in terms of other, less complex, chunks so that a respecification of the smaller chunks is not neces[illegible]

This *open_file* scenario brings up another point: namely, that of instantiation or parameterization. It is certainly preferable to specify a simple generic chunk, like *open_file*, and supply a parameter detailing which file is open rather than creating a separate chunk for each of the possible eight open files that could arise. It should be evident that by combining instantiation with the hierarchical facilities just mentioned, a large number of chunks can easily be created.

For all chunks, their presence can be noted from either the attacking side or defending side or more precisely, from white's point-of-view or black's point-of-view; conceptually, it is the same chunk so our knowledge base of chunks need not include copies of each point-of-view. In a similar vein, many chunks can be present on either the king's side or the queen's side of the board (castling is a good example). Consequently, only one side need be specified for our chunk knowledge base. As well, it is possible for a chunk to be expressed with respect to (wrt) color *and* board side (of course, if a certain pattern is in the centre of the board, it need be expressed only wrt white or black). It should be clear from this discussion that our language must be capable of handling the specification of chunks with respect to one of these four areas (white-kingside, white-queenside, black-kingside, and black-queenside) thereby minimizing coding effort for the master while increasing chunk-data integrity. With this in mind, our chunk-detecting mechanism must be able to transpose a specified chunk into each of these four areas implicitly.

Another way to minimize coding effort is to include special conjunctive and disjunctive symbols to handle conditions. In most programming languages, to specify the disjunction of two or more conditions, each condition must be completely expanded to avoid ambiguity; e.g. to determine if *A* is equal to 1, 2 or 4, one must say,

IF A = 1 OR A = 2 OR A = 4 THEN ...

Our chunk language, however, need not be so verbose. Specifying pieces and locations for a chunk can quickly become tedious. To see if a white pawn (WP) is on squares *F2* or *F3* or *G3* or *G4*, we can say,

REQUIRE WP ON F2 / F3 / G3 / G4

Here, the "/" symbol is a special disjunction symbol to distinguish between the logical disjunction of conditions. To form the latter, we shall use a vertical bar, "|". For example,

REQUIRE (WP ON F2 / G3) | (WB ON G2)

means the chunk requires a white pawn to be located on either *F2* or *G3* or a white bishop to be situated on *G2*.

To briefly summarize, we see, at the very least, that our chunk language must include:

- positional features
  - *attack/defense properties*
  - *location information*
  - *piece-color and type information*
- *required* information
- *optional* information
- hierarchical facilities
- instantiation facilities
- ease of expression
  - *wrt* specifications
  - special *and* and *or* symbols

These features are similar to those found in *frame* or *prototype* languages of AI.

4.3. Qualitative Concerns

So far, we have discussed concepts that can be easily evaluated in quantitative terms for a chess position. For the most part, a feature is either present or not; that is to say, a white pawn cannot be *partially* on a square. However, there are a number of qualitative issues that can affect the usefulness of a particular chunk.

The first issue concerns the relative importance of a chunk, a clearly prevalent theme throughout this thesis (for chess, in general, in fact). As mentioned in the introduction of this chapter, the presence of a king on a certain square is, no doubt, more significant than that of a pawn. By the same token, a chunk may be important in one situation but completely irrelevant in another. For example, an open file is certainly important if its presence can lead to a back-rank mate but it can be quite meaningless if the next move in the game nullifies its presence. This suggests that our similarity measure should be intelligent enough to include a chunk only if it is important in the given position and that its importance may change when it is associated with certain other chunks.

Another qualitative issue that should be examined is the notion of advice; it can provide the means for supplying general rules of thumb when a certain chunk is present. Returning to our *open_file* example, we could specify the advice "Put major pieces on this file" when the chunk was present. This information could be passed back to the chess program along with information gathered from the *GAMETREE* to aid in the selection of the best move.

4.4. Implementing the Language

In the current implementation of the chunk language used by MACH, we have made use of the Unix utilities Lex [Les75] and Yacc [Joh75] to parse each chunk description and invoke the actions to determine a chunk's presence. Consequently, we were able to incorporate the positional features of location, piece-color and type information, the so-called *required* and *optional* facilities along with "wrt" handling and the special *and* and *or* symbols. To some extent, we have made use of the *advice* notion, though only in a "comments" role. The chess program using MACH must have the capability to interpret the advice and use it to its advantage.

However, a direct result of this project is the development of plans within the chess program *Phoenix*. In specific cases, *Phoenix* applies its own "chunking" methods to recognize a limited number of patterns and implements a number of plans accordingly. In fact, the *advice* portion of our chunk descriptions could be used to incorporate these plans to guide *Phoenix* in specialized situations.

Unfortunately, the notion of *importance* has not been implemented. Proper development of the necessary mechanisms and heuristics to implement this idea is certainly non-trivial and remains a topic of further research. In addition, the issues concerning attack and defense properties and the instantiation and hierarchical facilities have not been used simply because of our use of Lex and Yacc and their sequential processing of the chunks. Clearly, to work with a real-time chess application, all these issues must certainly be used.

Nevertheless, the features that have been implemented are powerful enough to yield interesting results and are reasonable, we believe, for a first-attempt. Indeed, much experimentation is needed to develop a good understanding of the influence of each of these issues to provide an effective similarity measure.

4.4.1. Lex and Yacc

As already mentioned, we implemented our language using Lex and Yacc, in keeping with our "minimal coding" philosophy. We needed a simple way of testing our language and Lex and Yacc provided the means. Letting these utilities worry about the parsing and lexical details allowed us to concentrate on developing the actions that were to be taken to test the presence of a chunk and recording its c-class for later use.

To determine a chunk's presence, we simply tested each clause of the chunk description for its truth value. Included in this test was the transposition mechanism previously mentioned above to determine if the chunk were present in one of its mirror images. Each action simply returned *true* or *false* and a chunk's presence was indicated if all *REQUIRE* clauses evaluated to *true*. A complete listing of the our grammar is provided in Appendix A1.

4.4.2. An Example

To better illustrate our work, Figure 4.1 presents an example of a chunk description for an *castle* chunk and one for a *lopez* opening chunk.

In the *castle* chunk description, we see the use of the phrase "...wrt white kingside." The "wrt" verb details how the chunk is conceived: the pattern describes the pieces of castled king for white on her kingside of the board. Implicitly, the pattern for a castled king on white's queenside as well as black's queenside and kingside is also present; the transposition facility of the interpreter makes this possible. The second statement of this description (and, indeed, of all descriptions) specifies that this chunk belongs to a specific c-class; in this case, it is the *kings* c-class that contains most patterns involving the king. The number *1* following the "c_class" predicate simply refers to the c-class number that is handled by MACH in its

```
chunk castle wrt white kingside.
classification kings c_class 1.
require WP on G2.
require WP on H2 / H3.
require WP on F2.
require WK on G1 / H1.
        WN on F1 / F3.
        WR on F1 / E1.
end castle.
```

```
chunk lopez wrt white.
classification centre c_class 2.
require WP on E4 + D4.
require BP on E5 + D6.
WP on C3.
        BP on C5.
        WN on F3.
        BN on F6.
end lopez.
```

Figure 4.1 A typical chunk description of a castled pattern and Ruy Lopez opening.

similarity routines. The next four statements, beginning with the *require* verb, describe the pieces that must be present for the castled pattern to be present - namely, that there should be a white pawn (WP) on squares *G2*, *H2* or *H3* and *F2* as well as the white king (WK) on squares *G1* or *H1*. The next two statements without a verb indicate the *optional* pieces that can be present in the chunk. If there is a white knight (WN) on squares *F1* or *F3* and a white rook (WR) on squares *F1* or *E1*, then these pieces should be included in the chunk's makeup; their absence, of course, would not affect the chunk's presence. Notice the use special disjunctive symbol '/'.

For the *lopez* chunk, we see it is specified in terms of white only. This means that its presence is meaningful only on the centre of the board. There are two items of importance in this description. The first involves the classification. Since this chunk describes a centre pattern, it is assigned to the *openings* c-class; it is not at all related to the *castle* chunk. The other item of interest is the use the '+' symbol to represent the conjunction of squares (of course, this could have been used in the *castle* chunk for squares *G2* and *F2* by writing, "require WP on G2 + F2."; we intentionally omitted this to further illustrate its benefits). Both the special conjunctive and disjunctive can be mixed in any *require* or *optional* statement with the corresponding logical symbols, '&' and '|'.

4.5. Finding Similar Positions

As you can see, a list of simple conditions can produce a host of complex patterns to be sought for in a position. If qualitative issues such as importance or advice could be readily manipulated, this language's power of expression would significantly improve and produce a more accurate filter for dissimilar positions.

Nevertheless, we currently have a good basis to develop a reasonable similarity measure to quickly hone in on similar positions. Figure 4.2 illustrates our methodology.

To begin, a given position is *chunked* to produce a list of c-classes (recall from Chapter 3 that the *GAMETREE* is indexed by c-classes). Then, we search through the *GAMETREE* finding positions that have at least *n-1* of these c-classes in common. Since a chunk and its transposes belong to the same c-class, two positions may contain the same c-classes but have totally different chunks. As a result, we must weed through this first group of positions discarding those that do not match at least *n-1* of the actual chunks present in the given position. To do this requires us to *chunk* the group of positions once again to record the actual chunks.

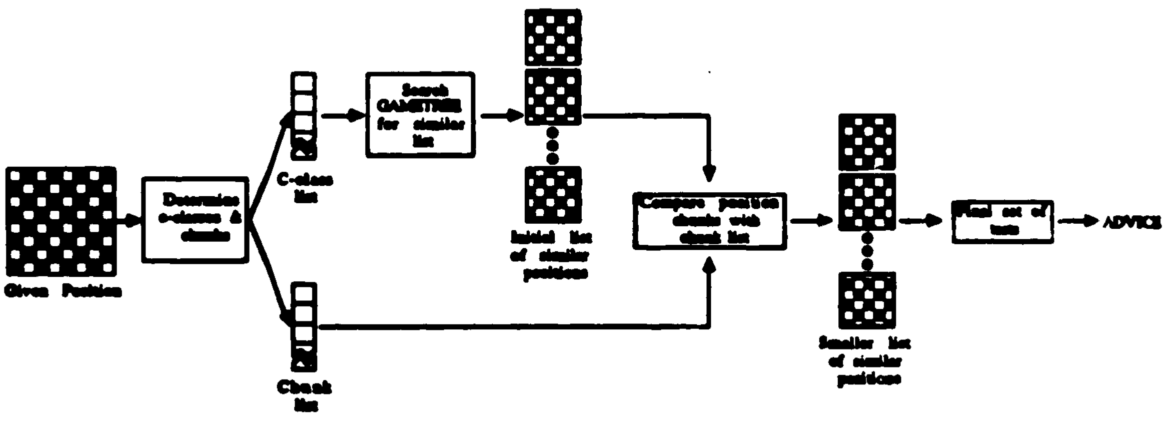

Given Position
Determine c-classes & chunks
C-class list
Chunk list
Search GAMETREE for similar list
Initial list of similar positions
Compare position chunks with chunk list
Smaller list of similar positions
Final set of tests
ADVICE

Figure 4.2 Methodology used to find a group of similar positions.

From this second set of positions, we apply a number of tests, examining each position on the basis of material balance, piece-type and piece location. For each of these tests, we have assigned ad-hoc scores to compare similarity. A position that receives a score of 140 represents a perfect match. Our tests and score assignment criteria are certainly naive but are adequate for our purposes. As with most first-attempt., there is obvious room for improvement.

The first test involves material balance (i.e. the net difference in chess material between white and black). Any position that differs by more than 100 points (i.e. one pawn-value) in material balance from the given position is deemed not similar and is discared. Those positions that have the same amount of material are given a similarity score of 20; those off by 100 piece points receive only 10 similarity points.

This test, however, has no way to determine match piece-type, only whether the total material is the same. For example, one position may have three pawns totaling 300 points and another position may have a bishop which also is valued at 300 points. Consequently, the actual material on the board must be checked. We have determined that any position which differs by more than one in the number and type of pieces of each color gets discarded. A perfect match scores 20 similarity points, a difference of only one piece from either side receives 15 points and a difference of one piece from each side gets 10 points.

In a similar way, a position may have the same pieces but these pieces could be situated on different squares, yielding a position that is not at all similar. If a position passes the material balance and piece-type tests, it is then subjected to a location test. A score is tabulated based on the number of matched pieces and locations with the given position (recall the memory test of Chase and Simon). Scores range from 0 to 100, according to the simple formula,

$$Location\_score = \left\{100 * \left[\frac{\#\_matches}{\#\_pieces}\right] - T\right\} * \frac{100}{100 - T}$$

where T is some threshold between 0 and 100.

For our tests, T was set to 66 2/3, meaning that at least 2/3 of the pieces must lie on the same squares for the position to be considered similar. Any score less than 0 is reset to 0.

This location test seems to work well when there are many pieces on the board. In the endgame, however, when the number of pieces is small, a difference of only 1 piece can mean a dramatic drop in score.

To be more useful, this test should be sensitive to the actual number of pieces on the board and adjust the similarity points accordingly. Until we get a better indication for what should comprise an effective similarity measure, we have ignored this detail.

If a position passes all our tests, the "best" move associated with it is extracted and written to an initial move list along with its similarity score. Once this list is complete, it is likely that it contains duplicate (i.e. similar positions sometimes produce the same "best" move) and illegal moves for our desired position, especially if the matches were not identical ones. As well, there may moves from the opposing side only; these, too, should be considered illegal (on the other hand, we could view these moves as advice on what our opponent might do). For example, we may be looking for white move but the move provided by the match was a black move.

Consequently, before any information is passed back to *Phoenix*, an extra process must be conducted to weed out all duplicate and illegal moves and create a new, more usable list. Therefore, one by one, each move from the initial move list is examined for its legality. If it passes this test, the new move list is checked for a matching entry. If none is found, it is written to this new move list along with its corresponding similarity score; if a match does occur, the higher similarity score gets written. Of course, if the move were illegal to begin with, it is immediately discarded. This new move list is then passed on to *Phoenix* to help in its choice of the best move.

4.6. Using the Language

Developing a comprehensive knowledge base of chunks is not an easy task. One method is to have an experienced chess player analyze a large number of positions and describe the important patterns in each. Ultimately, this method is very time-consuming. As well, forcing the master to describe each chunk in our primitive language would not only slow his progress further but, more importantly, would increase the chance of error.

This is because a master recognizes a chunk not by a 6 line description but rather from a visual image of the chess board. Each chunk is defined not only by the pieces within it but also in relation to other pieces on the board (i.e. they are not self-contained). Therefor [illegible] master could simply place the pieces of the chunk on a board and let a program covert the ima[illegible] board into a description, we could be much more effective and make significantly better use of the master's time.

We believe, to use the language efficiently, a front-end graphical interface describing the chess board should be used to define the chunks for our knowledge base. The processing of each image into our language could be performed behind the scenes; transforming location information is trivial for a program. For the inherent qualitative information of a chunk, such as advice and importance, the master to could be prompted enter any appropriate information.

Of course, our current implementation does not include this facility. Each description must be entered and passed to Lex and Yacc for syntax validation. With a graphical interface, the master need not worry of these tedious details.

4.7. Summary

Developing an effective similarity measure, we have found, is an extremely difficult task. Without good rules of thumb, much of the testing is simply trial and error. Assigning similarity is not nearly as difficult or time consuming as constructing useful chunks. Great care must be taken to create a chunk that is neither too specific, so that it rarely occurs, nor too general, so that it does not occur too frequently; either extreme produces an ineffective filter. The next chapter is devoted to a presentation of test results of MACH interfaced with *Phoenix*, showing where its advice aided and hindered *Phoenix* as well as useful rules of thumb that should be applied when developing chunk descriptions.

Chapter 5

MACH's Performance

5.1. Introduction

Chapters 3 and 4 have set forth MACH's overall design, including its data and file structures along with a specific language to represent chunks. From the outset, our philosophy has been to develop MACH only to the point of getting a feel for its usefulness and practicality. We have already outlined some of the difficulties faced in developing an efficient search procedure and an effective similarity measure. Nonetheless, some interesting results have come out of our initial system.

Since our chunk descriptions are certainly not all encompassing, there will exist a set of positions in which none of the defined chunks will appear. In this case, the chess program must rely on its own judgment since MACH would be unable to provide effective advice. However, in the areas for which we are targeting - namely, the opening, the early middlegame and late endgame - there do exist patterns that occur with unusually high frequency. It is these areas where MACH will prove the most beneficial and it is precisely these areas where chess programs can easily falter.

This chapter will show that MACH has successfully demonstrated that grandmaster games can be used to provide useful advice. As well, we will list a few good rules of thumb that should be followed to achieve well-developed chunk descriptions and summarize a number of unsolved items that require further experimentation. For all of the tests presented in this chapter, we made use of the chess system *Phoenix* [Sch86]. In addition, all of our test games are courtesy of Dap Hartmann [HaT86].

5.2. Advice Paradigms

MACH can provide two types of advice: good or bad. The chess program can do two things with this advice (use it or discard it) but it needs to avoid situations that enable it to use bad advice or ignore good advice. It is up to the chess program to determine when it should use the advice and when it should ignore it; MACH simply reports what it finds.

As outlined in Chapter 4, the final result from MACH's search and similarity evaluations is a list of moves and their associated similarity scores. Before *Phoenix* examines these moves, it performs a search of its own game tree to a predetermined ply. All terminal nodes from this search are then assigned a score; refer to Figure 5.1. Each of the moves leading to the terminal nodes are checked against MACH's move list. If any of the moves match, a score is assigned to them and added to the score of the terminal nodes. *Phoenix* then picks the path (move) with the highest score.

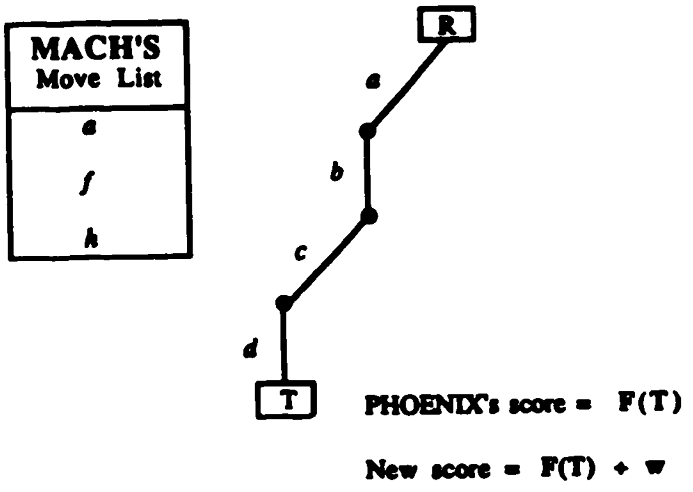


Figure 5.1 How Phoenix uses MACH's advice to choose a move.

In the figure above, *Phoenix* has generated a path from *R* to *T* by making the moves, *a*, *b*, *c*, and *d* and assigned the score *F(T)* to the terminal node *T*. MACH's move list includes the moves *a*, *f* and *h*. Since the move *a* is found both in MACH's list and the path leading to *T*, the score at *T* becomes, *F(T)* + *w*, where *w* is some positive weight. In this case, since the matched move was first on MACH's list, *w* was given a value of 20.

As is evident, the moves provided by MACH are not guaranteed to be choosen; rather, by adjusting their weights, the probability of one of them being choosen is simply increased. Since *Phoenix* has its own expertise on what constitutes a "best" move, it does not blindly use the moves supplied by MACH. However, if the scores of two nodes are close to each other, MACH's advice will swing the pendulum.

One difficulty in using advice from MACH is the integrity of that advice. Unfortunately, the *GAMETREE* will contain useful as well as useless information (we trust the former is more prevalent). The games are added to the *GAMETREE* as played, including all the good and bad moves made without annotations indicating the quality of a move. A program using this information, therefore, must be careful to filter the data and extract relevant information, erring, if necessary, on the side of conservatism.

One of the simplest applications resulting from the use of this advice is to employ the *GAMETREE* as a means for playing opening moves. However, unless each game is examined, there is no guarantee that the moves in the knowledge base are best. At least in an opening book, such as *Encyclopedia of Chess Openings* [Mat87], there is a high confidence in the analysis given and the justification for the right move (although, even then, many errors find their way in). Unfortunately, data from various sources will contain many moves of questionable quality and hence, *Phoenix* must apply what it *knows* about chess to avoid the negative feedback.

Over the next three sections, several examples will be presented to illustrate some of the problems *Phoenix* encountered interfacing with MACH.

5.3. Results from an Identical Match

Since many chess matches open in a predefined and limited number of ways, it is quite common to extract identical positions from the *GAMETREE*, sometimes as many as 20 moves into the game. Some of these identical matches are presented below. We will use algebraic notation for each of the moves presented, as done so in Chapter 3.

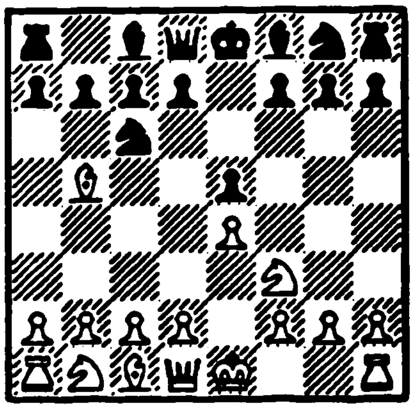


Figure 5.2 A sample position reached by Phoenix, 5 plies into a game.

Figure 5.2 shows a board position that *Phoenix* (as black) encountered after a game that opened with,

1. Pe2e4	Pe7e5
2. Ng1f3	Nb8c6
3. Bf1b5	?

Through its normal analysis, *Phoenix* determined that the best move to make was *Ng8f6*. MACH did, indeed, report this move to be superior; however, it also suggested *Pa7a6* and *Bd8d7*. The reason it supplied more than one move means that the *GAMETREE* included games that proceeded in at least three different ways (not surprisingly). But which way is best? We see here a problem of consensus.

The move *Pa7a6* is played almost 100% of the time in this situation while the other two moves are only occasionally played. Though a frequency count could be incorporated into each move, MACH has no way to know this fact. We could force MACH to favor the "most frequently" played moves; however, because a move is played most often does not mean it is the best; indeed, the possibility could arise where an uncommon move proved to be brilliant, opening a line of attack that no one had previously discovered. Until better criteria for selecting moves is defined, MACH unbiasedly reports all moves.

Since both *Phoenix* and MACH determined that *Ng8f6* was a superior move, it was given enough weight for it to be the one chosen.

3. ... Ng8f6
4. O-O ?

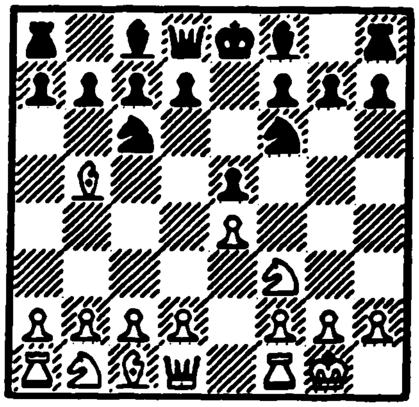


Figure 5.3 The position farther into the game before Phoenix uses MACH's advice.

Continuing the game to move #4, *Phoenix* arrived at the position in Figure 5.3; here, white had just castled. Searching to a depth of 5 ply, *Phoenix*, without MACH's advice, reasoned that *Pa7a6* was the best move to make; MACH, however, reported that *Bf8c5* was better. With MACH's advice, *Phoenix* agreed and proceeded to move her bishop. In fact, this advice actually prevented a blunder. If *Phoenix* chose to move its pawn from *a7* to *a6*, a search of the tree several plies deeper showed this move to cause *Phoenix* to lose a pawn, resulting in a lost position. This is a case where MACH provided good advice and *Phoenix* decided to follow it.

As the game progressed, *Phoenix*, by following MACH's advice, was forced to deviate from the standard opening and encountered unfamiliar territory; figure 5.3 illustrates this point.

4. ... Bf8c5
5. Nf3xe5 Nc6xe5

6. Pd2d4	Pa7a6	
7. Bb5a4	Nf6e4	- deviate from ECO[6]
8. Qd1e2	?	

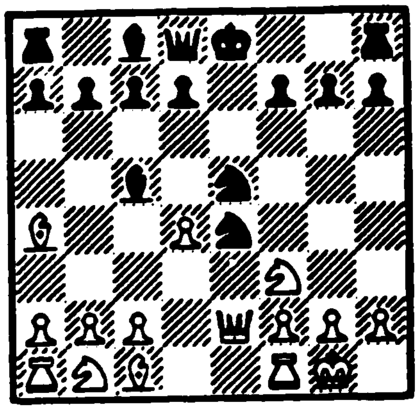


Figure 5.4 An unfamiliar position for Phoenix.

In this last set of moves, *Phoenix's* generated moves were the same as the ones provided by MACH. After white moved her queen from *d1* to *e2*, *Phoenix* began to lose control due to inadequate search depth. *Phoenix* decided that capturing white's pawn on *d4*, using her bishop, was the best move. MACH suggested retreating the bishop from *c5* to *e7*. Once again, MACH's advice proved to be superior several plies later. Unfortunately, MACH's advice was not sufficient to change *Phoenix's'* decision. By playing *Bc5d4*, *Phoenix* lost a piece and eventually the game. This demonstrates a situation where *Phoenix* discarded good advice. More importantly, though, it illustrates the need for *Phoenix* to recognize when it should use MACH's advice, especially when a board position is unfamiliar.

[6] The *Encyclopedia of Chess Openings*, [Mat87].

5.4. Results from a Similar Match

Though we have seen that MACH can provide useful advice when identical matches occur, it would be far more interesting and demonstrate MACH's real power if helpful information could be supplied from only similar matches. Indeed, it would illustrate how effective our similarity tests have been. This section will present several examples to show that MACH's similarity functions are effective enough to retrieve similar positions. The results presented are but a representative sample of the types of tests and positions that can be performed.

MACH's abilities to find similar positions were on a set of 20 chess positions, 10 each arising from well known openings[7]. The positions were generally 15-20 plies into the game. Unlike the previous section, an actual complete game was not tested; rather, the tested positions were those in which *Phoenix* did not play well. As a result, we did not interface MACH with *Phoenix* to see if the advice were followed. We used our own chess knowledge to determine whether the moves were useful and the positions similar. In figures 5.5 to 5.7, the test position is presented on the left with MACH's two most similar positions on the right.

From each of the test positions used in these examples, MACH returned an average of 5 similar positions. To speed up our testing, only those games that began with our two selected openings were added to the *GAMETREE*. This resulted in a total of 8526 positions out of some 60,000 positions we had available.

Our chunk descriptions (shown in appendix A1) used for our tests fell into 11 c-classes. Though only one description was provided for each c-class, using our transposition mechanism effectively 34 different chunks. Chunking each of the positions in the *GAMETREE* resulted in 64.4% (5490) of the positions containing at least one chunk. The other 35.6% not covered by our chunks consisted mainly of late middlegame and endgame positions.

Table 5.1 gives the experimental data for the 20 positions using the similarity criterion that *n-1*

[7] The Ruy Lopez and Queen's Gambit Declined.

chunks must be present for a match to occur. On average, each of the 20 test positions contained 6.1 chunks. This number is amazingly similar to the 5.75 chunks per position of Simon and Gilmartin's MAPP and confirms their results. Our 6.7 pieces/chunk is significantly higher than MAPP's 3.9 figure and De Groot's average of between 3 and 4 [DeG65]. This is a result of the *optional* piece specification, something that Simon and Gilmartin did not use. If we include the number of *required* pieces only, this value drops to 3.4.

On the memory test, MACH is able to reconstruct an average of 80% of the pieces on the board, close to the 73% figure of MAPP, again confirming their results.

Summary of MACH's Performance				
Position	# pieces in position	Pieces found	Chunks found	Suggested Moves
Ruy Lopez (10)	310	241	56	22
Queen's Gambit (10)	312	257	66	20
Total	622	498	122	42
Average	31.1	24.9	6.1	2.1

Table 5.1 MACH's performance on 20 positions.

Our first test position is presented in Figure 5.5. In this test, there were actually five positions supplied by MACH; the three positions not shown were not as similar as the two in the figure which indicates that our chunk descriptions are not detailed enough.

As you can see, the positions MACH found are both visually and structurally similar. This is a result of over 90% of the pieces residing on the same squares. Interestingly, our chunks found only 47% of the pieces. In fact, each position contained the same chunks: white's castled king, queenside's pawn formation, white's undeveloped queenside. One noticeable difference is that position *A* contains a castled king on black's kingside.

For this situation, it was white's turn to move. MACH supplied three moves with equal similarity

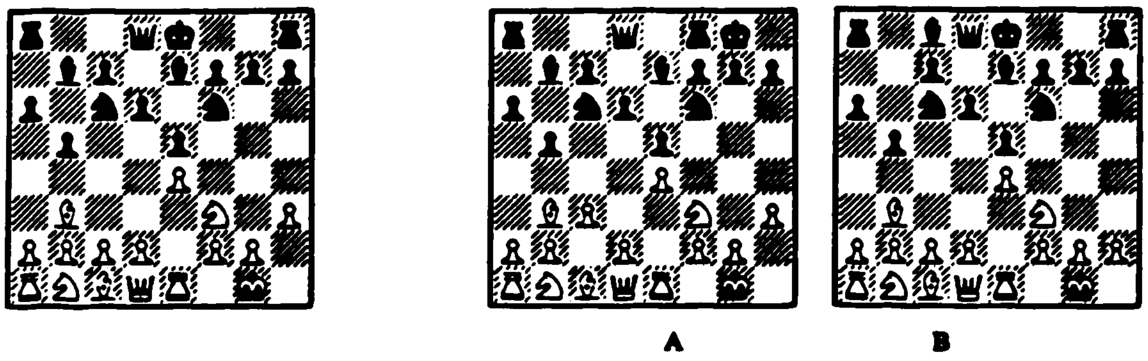


Figure 5.5 Ruy Lopez test position #1.

scores: *Pd2d4*, *Pc2c3* and *Pa2a4*. All are reasonable moves that most players would make.

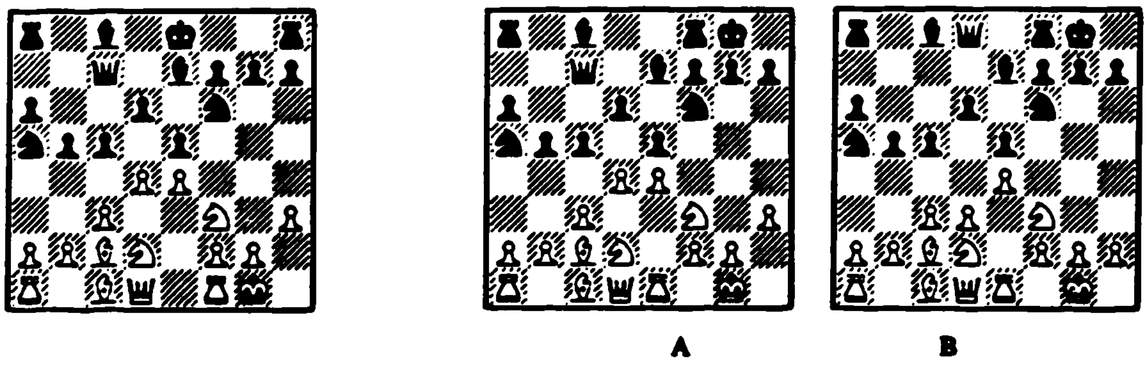
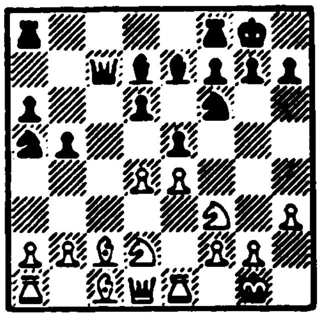


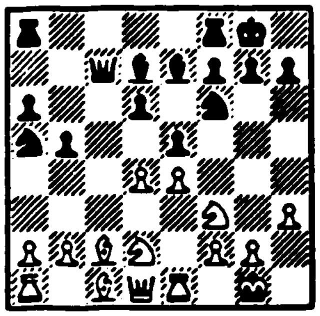
Figure 5.6 Ruy Lopez test position #2.

Figure 5.6 illustrates our second test position. Again, the untrained eye will admit that MACH's position are visually similar; in fact, position *A* can be considered structurally similar. Therefore, it is not surprising that position *A* received a higher similarity score than did position *B*. The main fault with position *B* is that too many pieces are on different locations. Some of the important structural features that matched in the first position include white's pawns on *h3* and *d4*, white's castled king and black's queen on *c8*. All of these features are not present in position *B*. This example illustrates that our chunk descriptions are useful for finding a similar position but not restrictive enough to weed out less helpful ones.

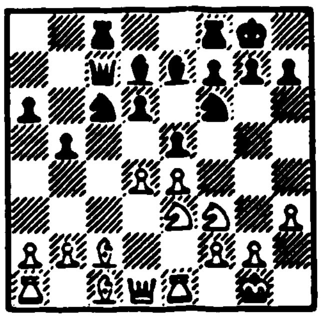
Since *Phoenix* was black in this position, the moves MACH suggested were *Pc5d4* and *Na5c6*, retrieved from positions *A* and *B* respectively. Both are good moves.

The final example comes from a position 14 plies into the game, presented in Figure 5.7. We see that position *A* is actually an exact match. Position *B*, however, is not so lucky; it has both a white and black knight misplaced as well as a black rook. Nevertheless, it is still visually similar and most of the structural qualities match, including the castled kings and the advanced pawn on *h3*.





A



B

Figure 5.7 Ruy Lopez test position #3.

From the exact match, MACH suggested retreating white's knight from *d2* to *f1* and advancing white's pawn to *d5*. Position *B's* move advised us to capture black's pawn on *e5* instead of advancing the pawn to *d5*.

These examples, we believe, are strong evidence in favor of the chunking theory as a similarity measure but more importantly show that MACH's advice is useful in similar situations. However, it is also true that the quality of the chunks descriptions determines the similarity's effectiveness, as shown in Figure 5.6. Later in this chapter guidelines will be presented to aid in this important problem. The positions presented are but a few representative examples that demonstrate the usefulness of a MACH-type system. By creating a wider variety of chunks and a faster chunking process, MACH could provide useful advice in many more situations; the larger its knowledge base, the more effective it can become.

5.5. Results Matching *n* Chunks

As outlined in chapter 3, we said that two positions could be similar if they had at least *n-1* chunks in common. We reasoned that since our chunk descriptions were subject to errors and omissions and we did not know whether a chunk was relevant in a position, matching only *n-1* chunks could, in some ways, increase the integrity of our tests.

To test this reasoning, we ran our five test positions through MACH once again but this time we force all chunks to match. As we might expect, in two instances, MACH provided less moves than it did using *n-1* matches. Unfortunately, three of the five tests returned the same set of legal moves, mostly because our chunk descriptions were too general.

Nevertheless, table 5.2 illustrates that using *n* chunks reduces the number of candidate positions by 23.4%, on average. Though we experienced only six reductions of move lists supplied by MACH, the possibility of missing similar positions is definitely increased.

# of Candidate Positions		
Opening	*n-1*	*n*
Ruy Lopez	319	245
Queen's Gambit	325	248
Total	644	493
Avg reduction	23.4%	7.6 positions

Table 5.2 Comparison of the number of positions found using *n-1* and *n* chunks

One other significant result from this test is the speed at which MACH ran. Though we have no real-time numbers, requiring *n* chunks to match instead of *n-1* greatly increased the speed of MACH. Since the chunking process (using Lex and Yacc) is the bottleneck, finding a smaller number of candidate positions cannot help but reduce the number of chunk comparisons.

5.6. Guidelines for Constructing Effective Chunk Descriptions

The slowest task involved with MACH's development was the addition of chunks; this is not surprising since knowledge acquisition is a difficult process. Without a great deal of experimentation, one has to rely on common sense more than anything else. Unfortunately, Simon et al. did not provide an insight into this area; their chunks comprised patterns that frequently occurred in "...games in the published literature" [Sim73b]. Other than mentioning that their patterns adhered to the relations of attack and defence, piece-type, piece-color, and proximity, no discussion was provided regarding the actual development of their chunks. From our experimentation, however, a number of conclusions can be drawn which can help to expedite future work.

One of the first results we found was that it was very easy to create chunk descriptions that were too general; these descriptions would tend to encompass patterns that were not relevant at all, producing an ineffective filter of dissimilar positions. By the same token, making these chunks too specific resulted in MACH overlooking positions that were similar. There seemed to be a narrow region between these two extremes.

However, incorporation of an *importance* factor might, indirectly, broaden this acceptable region. By assessing the relative importance of each chunk found in a position, general chunks may be weeded out since, typically, their broad descriptions tend to cover pieces that are unimportant. Further work is certainly needed in this area.

Notwithstanding, a chunk can be made less general if the pieces it contains are not spread across the board; that is, the pattern should be localized on a small portion of the board. Simon et al. used the rule that two pieces proximate each other if one of the pieces is within one of the eight adjoining squares of the other. Our chunk descriptions generally adhere to this rule except when they incorporate the use of the "or" operator. This, typically, multiplies the number of chunks that will match and has a tendency to spread pieces apart. One the other hand, overuse of the "and" operator will cause the opposite effect. We believe that use of both operators should be done sparingly.

The only way we discovered this ideas was through our similarity testing. Although trial-and-error is certainly the least optimal method by which to create these chunks and until a more accurate and efficient method can be found, we propose the following methodology.

1) Define chunks
2) Set up test position(s)
3) See which positions did and did not match
4) If satisfied, stop, else goto step (1)

The first step in the process is to define a set of chunks, without much concern for generality or specificity. The next step is to set up a series of test positions for which we expect to find similar positions in the database. After invoking MACH's similarity tests, carefully examine the set of positions returned by MACH.

For each of these similar positions, check the similarity score and determine which chunks matched. If the position should not have been selected, it is likely that at least one of the matched chunk descriptions encompasses too many patterns (possibly, due to an overuse of the "or" operator). Each of the questionable chunks should be re-defined, perhaps, by splitting each into smaller, more specific chunks.

Though it is more difficult to do, it is worthwhile to see what positions, if any, in the *GAMETREE* did not match simply because some chunks were too restrictive.

By identifying these ineffective chunks, the likelihood of discovering all similar positions is increased. After restructuring the chunk knowledge base, the same process should be re-executed. Albeit slow, we believe this feedback process is necessary towards the development of a useful knowledge base to enable MACH to be truly effective.

Of course, if MACH were "intelligent" enough to define and re-define its own set of chunks, this bottleneck would greatly diminish. Positions could be supplied to MACH and it could pick out which patterns were relevant, piece by piece. Unfortunately, this realization is many years away and is another topic of further study.

5.7. Summary of problems

Though we have demonstrated that MACH can certainly provide good advice in most situations, more importantly, we have uncovered a number of problems that need to be addressed before MACH can seriously be integrated with a real-time chess program.

1) Discarding "good" advice

From our tests involving identical matches, we saw *Phoenix* discard good advice from MACH in favor of capturing a pawn. MACH had led *Phoenix* into an unfamiliar position and *Phoenix* did not know how to respond. To avoid further occurrences of this, we must either alter the way in which *Phoenix* uses the advice (simply by increasing scores) or, when it disagrees with MACH, search farther in the tree to see why MACH chose its moves.

These two solutions suggest that *Phoenix* must have more knowledge to appreciate good advice. Unless it is aware that its choice of move is inferior to that of MACH's, *Phoenix* will still discard good advice.

2) Different strategies used by different players

Grandmasters, over the years, have exhibited distinctive styles of play. Some players, like former World Human Champion Bob[illegible]her, tend to be aggressive in their play, attacking the opponent frequently, whereas other players adopt a more passive style. When MACH supplies advice, it makes no distinction between these styles of play. *Phoenix* has been designed to model an aggressive player and consequently, disagreements can result if MACH is providing the advice of a passive player.

3) *Chunking* is the most time consuming process

As outlined in the previous section, having MACH use *n* chunks instead of *n-1* caused it to run significantly faster. Indeed, chunking positions as a similarity test is an extremely slow process and is currently the bottleneck of our system. Of course, this is a direct result of incorporating Lex and Yacc since they parse each chunk description for each position.

Nevertheless, great care must be taken in developing an efficient chunking mechanism. Methods similar to those used by Simon and Gilmartin's MAPP system can serve as a basis for this development. They built a large tree (which they called their EPAM net), with each node corresponding to location check for a piece (like the *REQUIRE* statement of our grammar). The top nodes consisted of the salient pieces on the board. Each chunk was triggered by the discovery of its salient piece which directed the search for the chunk down a particular path. If the salient piece was not found on the position, it was deemed not to exist. Figure 5.8 illustrates what this tree might look like for our kingside pattern of Appendix A2.

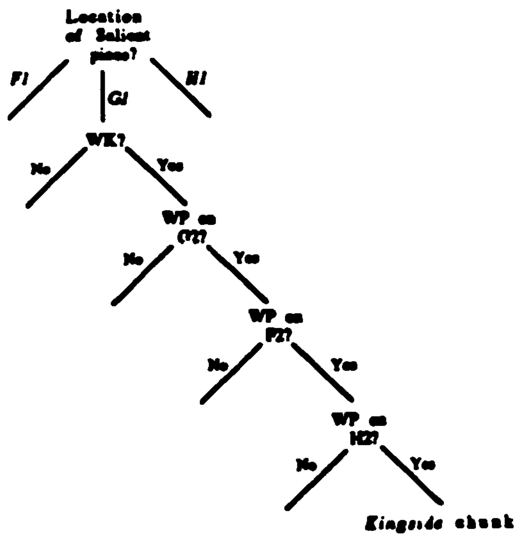


Figure 5.8 Segment of net that MAPP would use for our kingside chunk.

4) Dependence of the *GAMETREE* on chunk knowledge

Recall that as the *GAMETREE* was built, so was the *FEATUREndx*, indexing each position by its chunk list. If the chunk knowledge were ever changed (adding new chunks or redefining old ones), this *GAMETREE* and *FEATUREndx* must be rebuilt. While this was a relatively painless process for our 8500 positions, it can become a real annoyance if the *GAMETREE* contained a few million positions. Consequently, as a new partial-matching algorithm is sought, it must try to eliminate this complete dependence.

5) Defining chunks and c-classes

It is evident that defining chunks is a tedious process. Before a chunk is written to the knowledge base, it must be checked against all other chunks to see that it does not already appear. Assigning salient pieces to each chunk can overcome this somewhat but a more involved search must be conducted if a chunk contains more than one salient piece; this is similar to file having multiple keys.

By assigning each chunk to a c-class, we were able to reduce the number of comparisons on the first pass of our similarity tests from potentially tens of thousands down to fifty or sixty. However, it is not easy to define these c-classes; in addition, it is possible to assign one chunk to more than one c-class. In any event, improper definition of these c-classes can lead to the same problems characteristic of chunks, namely being too general or too specific.

We have attempted to outline some of the major problems involved in the development of a MACH-type system. Some are easier to solve than others. Indeed, more complications remain hidden that will surface only when attempts are made to combat these outstanding issues. Nevertheless, new ground has been forged towards the developing of a master advisor for chess.

Chapter 6

Conclusion

6.1. General Results

From the outset, our goal was to assess the feasibility of a master advisor for chess. To that end, we have been successful in demonstrating that such a system is worthwhile. Moreover, we have shown that to build a complete system is certainly a difficult task. However, through our analysis, we have uncovered a number of very interesting problems that are fundamental to many areas of computer science.

6.1.1. Partial-Matching Problem

The first problem we discovered was the so-called *partial-matching* problem. At first glance, this seemed like a relatively easy problem to solve. However, the combinatorial explosion quickly prohibited us finding a general solution.

Some research has been conducted in this area, as presented in chapter 3, but all studies have touched upon only special cases of this open problem. However, it should be noted that MACH's future development does not hinge upon a solution to this problem. Of course, an efficient partial-matching algorithm would certainly help in this regard. For the time being, though, by taking advantage of the domain-dependent properties of chess, our ***restricted enumeration*** methodology can provide fast retrieval times with reasonable storage requirements, effective enough to work with a real-time application.

6.1.2. Database Design

Related to this algorithm problem is that of database design, something which greatly affects the performance of any information system. As we mentioned in chapter 5, our *GAMETREE* and *FEATUREndx* are both dependent on the chunk knowledge base. Changing the latter results in a complete rebuilding of both databases. Indeed, a more flexible database design may provide insight into solving the *partial-matching* problem and vice versa. Exploring "Information Retrieval" theory may uncover some clues.

6.1.3. Learning

Another area investigated by this thesis is that of machine learning. We know that Samuel pioneered this area with his checkers playing program. His methods revealed some important notions that must not be overlooked.

First of all, his memorization of positions allowed for an increase of performance without an increase in search depth or time. As well, he enlightened us as to the subtleties of the relative importance of knowledge - where some ideas are relevant in one aspect of the game b[illegible] [illegible] other.

By applying these two fundamental ideas, a chess [illegible] can gr[illegible] into a more effective system. Th[illegible]gh it is primarily the responsibility of the chess program [illegible] MACH can certainly be called upon to learn to improve its advic[illegible] [illegible]rieval of it. By using the EPAM-like net described in [Sim73b], MACH can learn to recogn[illegible] [illegible] through its chunk descriptions. A testing mode can be established whereby MACH proces[illegible] [illegible]et of positions, displays the set of patterns it recognizes on the board and interacts with an expert in hopes of learning new patterns.

As far as the chess program is concerned, it can improve its performance with MACH by becoming more "cognitively economical." As the chess program encounters positions where MACH supplies it with better moves than it determined from its normal evaluation, the chess program could store these better moves for future situations or update its evaluation function so that it would next time choose these moves (similar to [Van87] method). By doing this, the system becomes *intelligently redundant.*

One of our examples in chapter 5 showed that MACH's advice can produce a problem of consensus. In particular, one of the moves supplied by MACH was one that would have been played 99% of the time while the others were played much less frequently. However, because MACH's *GAMETREE* contains knowledge from some of the greatest games ever played, one of those less frequent moves may be more effective than the most commonly played one, exploiting a line of attack never before discovered. By adapting th[illegible] chess program to discover these "surprise" moves, its focus of attention becomes much more astute.

Though not learning per se, MACH is able to provide the means by which a chess system can improve its performance; learning is simply the next logical step. Of the four types of learning, *taking advice* is the paradigm which best suits the current interface between *Phoenix* and MACH. Though *Phoenix* does not store any information about this advice, it is still able to improve its performance. In addition, if MACH supplied *Phoenix* with a masters' next *n* moves, it could examine further lines of play and lend itself to follow the example of the master. As already mentioned, this can be extremely important in the endgame where strategies are difficu'· to code into a chess program.

6.1.4. Knowledge Acquisition

The assumption made by all these ideas is the existence of a large usable chunk knowledge base - truly a big assumption. In the previous two chapter, we made reference to how slow and difficult this knowledge acquisition can be.

We have seen that the MAPP system of Simon and Gilmartin contained over 1000 different chunks. From their memory experiments, that was enough to allow MAPP to reconstruct only 73% of the board, at best. Indeed, their estimates of 50,000 patterns may, perhaps, be large but ce [illegible] ainly it is probably not the case that a database of chunks is ever too big. It is difficult to know when a knowledge base is adequate.

There are several other factors involved in this knowledge acquisition. The first, as outlined as a requirement for our chunk language, is the *relative importance* of a chunk. It sho·' ' now be obvious that a chunk's importance is not a static value. Determining this value, however, remains a topic of further study.

Related to this importance problem is the notion that different parts of the game have different requirements. Strategies for the opening game are much different from those of the middle and endgames. Indirectly, our chunking mechanism must able to adapt to the changing parts of the game.

One other related item not addressed in this thesis is the problem of what to do with pieces that are not included in any chunk. Certainly, in some cases, they can simply be ignored; the fact that a pawn is out of place usually does not have much meaning. However, that same pawn, when associated with other pieces, may be important to include in a few chunk descriptions. Improving our chunk descriptions may overcome this problem but mechanisms should still be in place to discover occurrences of this.

One way to speed up the knowledge acquisition process is to incorporate a graphical interface with our chunk language to enable the expert to define chunks more readily. Since the image of the pattern is certainly more meaningful to the expert than a 6-line description of the pattern, the graphical interface places the player in a more familiar environment in hopes of invoking the thought processes that are involved in an actual game.

6.1.5. Spinoffs from this work

As mentioned in our introduction, this MACH-typ: system can be extended into other areas such as medicine or law. In fact, any field of study that relies heavily on past performance and results can apply the principles involved with MACH. A. well, designing a system for other areas may uncover clues into the development of a general advisor.

6.2. mary

To summarize, we see the following list as open problems.

1) Partial-matching algorithm
2) Efficient database design for our games
3) Learning mechanisms for both MACH and the system with which it interfaces
4) Guidelines for effective knowledge acquisition

We believe this project has been a worthwhile experiment; not only have we demonstrated that *Phoenix* can improve its performance, we have also uncovered a number of interesting problems. As well, the ideas incorporated into the design of MACH are not restricted to the game of chess; rather, the underlying principles of human problem solving are applicable to any area of study that relies on advice from experts in the field and from past experience. Moreover, by attacking the "chess problem" from a more human-like point of view, we have made strides towards the creation of world champion chess machine.

References

[Bar81] Barr, A. and E. A. Feigenbaum (eds.), "*The Handbook of Artificial Intelligence*", Vol. 1, Morgan Kaufmann, Inc., Los Altos, CA, 1981.

[Bay66] Baylor, G.W. and H.A. Simon, "A chess mating combinations program", in *AFIPS Conference Proceedings*, 1966 Spring Joint Computer Conference, 1966, 28:431-47, Washington, D.C., Spartan Books.

[Bea86] Beal, D.F. (ed.), "*Advances in Computer Chess 4*", Pergamon Press Ltd., Willowdale, Ont., 1986.

[Ber73] Berliner, H.J., "Some Necessary Conditions for a Master Chess Program", *IJCAI 3*, 1973 1:77-85.

[Ber83] Berliner, H.J. and M. Campbell, "Using Chunking to Solve Chess Pawn Endgames", in *2nd Int. Conf on AI and Chess*, Milan, Italy, 1983.

[Ber86] Berliner, H.J., "Computer Chess at Carnegie-Mellon University", in *Advances in Computer Chess 4*, D. Beal (ed.), pp. 166-180.

[Blu87] Blumer, A., J. Blumer, D. Haussler, and R. McConnell, "Complete Inverted Files for Efficient Text Retrieval and Analysis", *JACM*, 1987, 34(3):578-595.

[Bra80] Bratko, I. and D. Michie, "A Representation for Pattern-Knowledge in Chess Endgames", in *Advances in Computer Chess 2*, M.R.B. Clarke (ed.), 1980, pp. 31-56.

[Bra86] Bratko, I., P. Tancig and S.Tancig, "Detection of Positional Patterns in Chess" in *Advances in Computer Chess 4*, D. Beal (ed.), pp. 113-126.

[Cha77] Charness, Neil, "Human chess skill", in *Chess Skill in Man and Machine*, Peter W. Frey (ed.), 1977, pp. 34-53.

[ChS73a] Chase, William G. and Herbert A. Simon, "Perception in Chess", *Cognitive Psychology*, 1973, 4:55-81.

[ChS73b] Chase, William G. and Herbert A. Simon, "The mind's eye in chess", in *Visual Information Processing*, W.G. Chase (ed.). Proceedings of Eighth Annual Carnegie Psychology Symposium, New York: Academic Press.

[Cla80] Clarke, M.R.B. (ed.), "*Advances in Computer Chess 2*", Edinburgh University Press, Great Britain, 1980.

[Cla82] Clarke, M.R.B. (ed.), "*Advances in Computer Chess 3*", Pergamon Press Ltd., Willowdale, Ont., 1982.

[Coh82] Cohen, Paul R. and Edward A. Feigenbaum (eds.), "*The Handbook of Artificial Intelligence*", Vol. 3, Morgan Kaufmann, Inc., Los Altos, CA, 1982.

[DeG65] De Groot, A.D., "Thought and Choice in Chess", The Hague: Mouton, 1965.

[DeG66] De Groot, A.D., "Perception and memory versus thought: Some old ideas and recent findings", in *Problem Solving: Research, Method and Theory*, B. Kleinmuntz (ed.), 1966, New York: John Wiley.

[Ebe87] Ebeling, C., "All the Right Moves: A VLSI Architecture for Chess", Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University.

[Edi87] Editor, "A Revolution in Chess", *New In Chess 1987*, 1987, 3:94-98.

[Eis73] Eisenstadt, Marc and Yaakov Kareev, "Toward a Model of Human Game Playing", *IJCAI 3*, 1973, 1:458-463.

[Ell71] Ellis, Stephen H. and William G. Chase, "Parallel processing in item recognition", *Perception & Psychophysics*, 1971, 10(5):379-84.

[Fei61] Feigenbaum, E.A., "The simulation of verbal learning behaviour", *Proceedings of the 1961 Western Joint Computer Conference*, 1961, pp. 121-132.

[Fin74] Finkel, R.A. and J.L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", *Acta Informatica*, 1974, 4:1-9.

[Frd61] Fredkin, Edward, "Trie Memory", *Comm. ACM*, 1961, 3(9):490-500.

[Fre77] Frey, Peter W., "An introduction to computer chess", in *Chess Skill in Man and Machine*, Peter W. Frey (ed.), 1977, pp. 54-81.

[Gre67] Gregg, L.W. and H.A. Simon, "An Information-processing Explanation of One-trial and Incremental Learning", *Journal of Verbal Learning and Verbal Behaviour*, 1967, 6:780-787.

[Gri74] Griffith, Arnold K., "A Comparison and Evaluation of Three Machine Learning Procedures as Applied to the Game of Checkers", *Artificial Intelligence*, 1974, 5:137-148.

[Har84] Harbison, Samuel P. and Guy L. Steele, Jr., "*C: A Reference Manual*", Englewood Cliffs, N.J., Prentice-Hall, 1984.

[Hat86] Hartston, W.R., "Artificial Stupidity", in *Advances in Computer Chess 4*, D. Beal (ed.), pp. 52-58.

[HaT86] Hartmann, D., "Computer Analysis of Grandmaster Games", Report, Leiden Observatory, 1986.

[Hea77] Hearst, Eliot, "Man and machine: Chess achievements . thinking", in *Chess Skill in Man and Machine*, Peter W. Frey (ed.), 1977, pp. 167-200.

[Hor86] Horspool, R. Nigel, "*C Programming in the Berkeley UNIX Environment*", Scarborough, Ont., Prentice-Hall, 1986.

[Hya86] Hyatt, R.M., A.E. Grover and H.L. Nelson, "Cray Blitz", in *Advances in Computer Chess 4*, D. Beal (ed.), pp. 8-18.

[Joh75] Johnson, S.C., "Yacc - Yet Another Compiler-Compiler", Comp. Sci. Tech. Report No. 32., Bell Laboratories, Murray Hill, N.J., 1975.

[Kop86] Kopec, D., M. Newborn and W. Yu, "Experiments in Chess Cognition", in *Advances in Computer Chess 4*, D. Beal (ed.), pp. 59-79.

[Len79] Lenat, Douglas B., Frederick Hayes-Roth and Philip Klahr, "Cognitive Economy in Artificial Intelligence Systems", *IJCAI 6*, 1979, 1:531-536.

[Les75] Lesk, M.E., "Lex - A Lexical Analyzer Generator", Comp. Sci. Tech. Report No. 39, Bell Laboratories, Murray Hill, N.J., 1975.

[Lev86] Levy, D.N., "Chess Master Versus Computer", in *Advances in Computer Chess 4*, D. Beal (ed.), pp. 181-194.

[Mat87] Matanovic, A., B. Rabar and M. Molerovic (eds.), *Encyclopedia of Chess Openings*, Chess Informant, Belgrade, Yugoslavia, 1987. In 5 volumes.

[Mic82] Michie, D., "Information and Complexity in Chess", in *Advances in Computer Chess 3*, M.R.B. Clarke (ed.), pp. 139-144.

[Mic86] Michie, D., "Towards a Knowledge Accelerator", in *Advances in Computer Chess 4*, D. Beal (ed.), pp. 1-7.

[Mil56] Miller, G.A., "The magical number seven, plus or minus two: some limits on our capacity for processing information", *Psychological Review*, 1956, 63:81-97.

[Mor68] Morrison, Donald R., "PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric", *JACM*, 1968, 15(4):514-534.

[Nit82] Nitsche, T., "A Learning Chess Program", in *Advances in Computer Chess 3*, M.R.B. Clarke (ed.), pp. 113-120.

[Not71] Noton, David and Lawrence Stark, "Eye Movements and Visual Perception", *Scientific American*, 1971, 224(6):34-43.

[Pit77] Pitrat, Jacques, "A Chess Combination Program Which Uses Plans", *Artificial Intelligence*, 8:275-321.

[Pos69] Posnyanskaya, E.D. and O.K. Tikhomirov, "On the Function of Eye Movements", *Soviet Psychology*, 1969, 1:25-30.

[Ric83] Rich, Elaine, "*Artificial Intelligence*", McGraw-Hill, New York, N.Y. 1983.

[Riv76] Rivest, Ronald L., "Partial-Match Retrieval Algorithms", *SIAM J. Comput.*, 1976, 5(1):19-50.

[Sam59] Samuel, A.L., "Some Studies in Machine Learning Using the Game of Checkers", *IBM* "*es. Dev.*, 1959, 11:601-617.

[Sam67] Samuel, A.L., "Some Studies in Machine Learning Using the Game of Checkers. II - R- : Progress", *IBM J. Res. Dev.*, 1967.

[Sch86] Schaeffer, Jonathan, "*Experiments in Search and Knowledge*", Ph.d. Thesis, Dep ent of Computer Science, University of Waterloo, Waterloo, Ont., 1986.

[Sha50] Shannon, C.E., "Programming a Computer for Playing Chess", *Philosophical Magazine [Series 7]*, 1950, 41.

[Sim69] Simon, Herbert A. and Michael Barenfeld, "Information-Processing Analysis of Perceptual Processes in Problem Solving", *Psychological Review*, 1969, 76(5):473-83.

[Sim73a] Simon, Herbert A. and William G. Chase, "Skill in Chess", *American Scientist*, 1973, 61:394-403.

[Sim73b] Simon, Herbert A. and Kevin Gilmartin, "A Simulation of Memory for Chess Positions", *Cognitive Psychology*, 1973, 5:29-46.

[Sla77] Slate, David J. and Lawrence R. Atkin, "CHESS 4.5 - The Northwestern University chess program", in *Chess Skill in Man and Machine*, Peter W. Frey (ed.), 1977, pp. 82-118.

[Sla87] Slate, D.J., "A Chess Program thet Uses its Transposition Table to Learn from Experience", *ICCA Journal, June 1987, 10(2):59-71.*

[Ten81] Tenenbaum, Aaron M. and Moshe J. Augenstein, "*Data Structures Using Pascal*", Englewood Cliffs, N.J., Prentice-Hall, 1981.

[Tho82] Thompson, K., "Computer Chess Strength", in *Advances in Computer Chess 3*, M.R.B. Clarke (ed.), pp. 55-56.

[Van87] van der Meulen, Maarten, "Weight assessment in evaluation functions", in *Advances in Computer Chess 5, Noordwijkerhout, Netherlands, 1987, 1-6.*

[Wag71] Wagner, Daniel A. and Martin J. Scurrah, "Some Characteristics of Human Problem-Solving in Chess", *Cognitive Psychology*, 1971, 2:454-78.

[Wie87] Wiederhold, Gio, "*File Organization for Database Design*", McGraw-Hill, 1987.

[Wil66] Williams, L.G., "The effect of target specification on objects fixated during visual search", *Perception & Psychophysics*, 1966, 1:315-18.

[Wil79] Wilkins, David, "Using Plans in Chess", *IJCAI 6*, 1979, 2:960-967.

[Zob73] Zobrist, Albert L. and Frederic R. Carlson, Jr., "An Advice-Taking Chess Computer", *Scientific American*, 1973, 228(6):92-105.

Appendix A1

Chunk Grammar

```
/* The following list contains the tokens used by Lex.
/* Alphabetically, they represent the following:
/*      ADVICE : the word "advice" that begins the designation of an advice string.
/*      BLACK : the word "black" that designates the black pieces.
/*      C_CLASS : the word "c-class" that begins the designation of a c-class.
/*      CID : a label indicating the c-class to which a chunk belongs.
/*      CHUNK : the word "chunk" that begins the designation of a chunk's description.
/*      CLASSIFICATION : the word "classification" that begins the designation of a
/*              chunk's type.
/*      END : the word "end" that signals the end of a description.
/*      FFILE : an entire file on the chess board; e.g. D_FILE would represent the
/*              squares d1 through d8 inclusive. Such a specification is represented
/*              by a column (from A to H followed by the string "_FILE".
/*      ID : any unique identifier for a chunk consisting of a followed by any number
/*              of letters, digits or underscores
/*      KINGSIDE : the word "kingside" used to designate that the chunk is described
/*              from the king's point-of-view.
/*      NOTHING : the word "nothing" which specifies that a particular square or file
/*              is empty.
/*      NOTON : the string "noton" which specifies that a piece is not located on a
/*              specific square.
/*      OCCUPANT : a string used to designate a particular chess piece.     It is represented
/*              by a "W" or a "B" (for a white or black piece, respectively) followed
/*              by a "PIECE"
/*      ON : the opposite of "NOTON".
/*      PIECE : a 1-character name for each chess piece: i.e. a "P", "K", "Q", "N", "B",
/*              or "R" to indicate a pawn, king, queen, knight, bishop or rook, respectively.
/*      QUEENSIDE : the word "queenside" used to designate that the chunk is described
/*              from the queen's point-of-view.
/*      REQUIRE : the key-word "require" used to designate a condition or set of conditions
/*              that must exist for the chunk to be present.
/*      SQUARE : a 2-character string consisting of a column (from A to H followed by a
/*              row (from 1 to 8).
/*      WHITE : the word "white" that designated the white pieces.
/*      WRT : the string "wrt" which specifies the point-of-view in which the chunk is
/*              described; e.g. "...wrt white kingside".

%token      CHUNK WRT WHITE BLACK KINGSIDE QUEENSIDE PIECE
%token      END ID OCCUPANT FFILE REQUIRE NOTHING CLASSIFICATION
%token      ADVICE SQUARE ON NOTON C_CLASS CID


%nonassoc '&' '|' ON NOTON
%nonassoc '!'

%%

program             : classifications chunk_list
            ;
```

```
classifications         : classifications class_list
                | class_list
                ;

class_list      : CLASSIFICATION id_list '.'
                ;

id_list         : id_list ',' ID
                | ID

chunk_list      : chunk chunk_list
                | chunk
                ;

chunk           : chunk_header chunk_group chunk_body chunk_end
                ;

chunk_end       : END ID '.'
                ;

chunk_header            : CHUNK ID '(' parameter ')' with_respect_to '.'
                | CHUNK ID with_respect_to '.'
                ;

parameter       : ID ':' instantiation
                ;

instantiation   : SQUARE
                | FFILE
                | PIECE
                ;

with_respect_to         : WRT colour side
                |
                ;

colour          : WHITE
                | BLACK
                ;

side            : KINGSIDE
                | QUEENSIDE
                |
                ;

chunk_group : CLASSIFICATION ID C_CLASS CID '.'
                ;

chunk_body      : chunk_body statement
                | statement
                ;

statement       : require expr '.'
                | ADVICE comments '.'
                ;
```

```
comments        : comments ID
                | ID
                ;

require                   : REQUIRE
                |
                ;

expr            : expr '|' term
                | term
                ;

term            : term '&' factor
                | factor
                ;

factor          : piece_sq
                | '(' expr ')'
                ;

piece_sq        : and_list
                | or_list
                | occupied
                ;

and_list        : and_list '+' square
                | occupied '+' square
                ;

or_list         : or_list '/' square
                | occupied '/' square
                ;

occupied        : occupant ON square
                | occupant NOTON square
                ;

occupant        : OCCUPANT
                | NOTHING
                ;

square          : SQUARE
                | FFILE
                ;
%%
```

Appendix A2

Chunks used for recognizing Ruy Lopez and Queen's Gambit Declined formations.

```
# These chunks also appear in other openings but not the same combination.

classification c1, c2, c3.

# Typical kingside pattern

chunk ks1 wrt white kingside.
classification c1 c_class 1.
require WP on G2.
require WP on H2/H3.
require WP on F2.
require WK on G1/H1.
        WN on F1/F3.
        WR on F1/E1.
end ks1.

# Pattern for regular Ruy Lopez center formation

chunk lopez wrt white.
classification c2 c_class 2.
require (WP on E4) & (WP on D4).
require (BP on E5) & (BP on D6).
        WP on C3.
        BP on C5.
        WN on F3.
        BN on F6.
end lopez.

# Pattern for a deferred Ruy Lopez center formation

chunk deferredlopez wrt white.
classification c3 c_class 3.
require (WP on E4) & (WP on D3).
require (BP on E5) & (BP on D6).
        WP on C3.
        BP on C5.
        WN on F3.
        BN on F6.
end lopez.

# Typical Lopez queen-side pawn formation

chunk lopezqs wrt white.
classification c4 c_class 4.
require (BP on A6) & (BP on B5).
require WP on A2/A3/A4.
require WP on B2/B3/B4.
      WP on C3.
        BN on A5/C6.
        BQ on D8/C7.
        BR on A8/C8.
        BB on B7/C8/D7.
end lopenqs.

# A side of the board that is undeveloped yet

chunk undevelopedq wrt white queenside.
classification c5 c_class 5.
require WR on A1.
require WN on B1.
require WB on C1.
      WP on A2/A3.
        WP on B2/B3.
end undevelopedq.

chunk undevelopedk wrt white kingside.
classification c6 c_class 6.
require WK on E1.
require WR on H1.
require WP on G2/G3.
require WP on F2/F3.
require WP on H2/H3.
        WB on F1.
        WN on G1.
end undevelopedk.

#
# Queen's Gambit Declined chunks.
#

chunk qgdcenter1 wrt white.
classification c7 c_class 7.
require (WP on D4) & (BP on D5).
require (WP on C4) & (BP on E6/C6).
require (WP on E2/E3).
        WN on C3.
        BN on F6.
        WN on F3.
end qgdcenter1.

chunk qgdcenter2 wrt white.
classification c8 c_class 8.
require (WP on D4).
require  BP on C6.
require (WP noton C_FILE).
require (BP noton D_FILE).
require (WP on E2/E3).
        WN on C3.
        BN on F6.
        WN on F3.
end qgdcenter2.

chunk qgdcenter3 wrt white.
classification c9 c_class 9.
require (WP on D4).
require  BP on B6.
require (WP noton C_FILE).
require (BP noton D_FILE).
require (W[illegible]n E2/E3).
        WN on C3.
        BN on F6.
        WN on F3.
end qgdcenter3.

chunk qgdcenter4 wrt white.
classification c10 c_class 10.
require WP on D4.
require BP on D5.
require WP noton C_FILE.
require BP noton E_FILE.
        WN on C3.
        BN on F6.
        WN on F3.
end qgdcenter4.

#
# Others
#

chunk bn5pin wrt white kingside.
classification c11 c_class 11.
require ( (WB on G5) & (BP on H7) ) | ( (WB on H4) & (BP on H6) ).
require BN on F6.
        BP on G7.
        BB on E7.
        BQ on D8.
end bn5pin.
```