CHUNKING FOR EXPERIENCE

Michael George
Jonathan Schaeffer

Department of Computing Science
University of Alberta
Edmonton, Alberta
CANADA T6G 2H1

Abstract

Human game players rely heavily on the experience gained by playing over the games of masters. A player may recall a previous game to either obtain the best move (if he has previously seen the identical position) or suggest a best move (if similar to others seen). However, game-playing programs operate in isolation, relying on the combination of search and programmed knowledge to discover the best move, even in positions well-known to humans. At best, programs have only a limited amount of information about previous games. This paper discusses enhancing a chess-playing program to discover and extract implicit knowledge from previously played grandmaster games, and using it to improve the chess program's performance. During a game, a database of positions is queried looking for identical or similar positions to those on the board. Similarity measures are determined by *chunking* the position and using these patterns as indices into the database. Relevant information is subsequently passed back to the chess program and used in its decision making process. As the number of games in the database increases, the "experience" available to the program improves the likelihood that relevant, useful information can be found for a given position.

1. Introduction

For many subject areas requiring skills, a large body of previous experience is usually gathered which can be used as a resource to be studied for improving one's performance. Unfortunately, most attempts at having machines mimic human behavior are primarily concerned with trying to solve the problem in isolation (usually a difficult enough task), without exploiting the available wealth of others' experience. For many disciplines, this information is not available in machine readable form. Other than using simple memory tables, few programs attempt to take advantage of the acquired wealth of human experience on a particular subject.

For most games, a player's strength can be increased by re-playing and analyzing hundreds of master-level games. Besides the obvious skills that this can improve, a significant side-effect is the human's ability to remember the games or important fragments from them. During a game, strong players are frequently able to relate the position on the board to an identical or *similar* one seen in a game played by themselves or by others.

This allows the player to draw on the experience of (possibly) more experienced players. From these game fragments, one may recall the best move (if identical to a position seen previously) or suggest a move to play (if similar to other games seen). From the computer game-playing point of view, the finding of identical positions is not interesting; opening books used by chess programs are an example of how this can be achieved. However, the ability to discover similar positions has the potential for providing a significant improvement in a game-playing program's strength. From a set of similar positions, a program may be able to extract such information as the move(s) played, the plan or strategy followed by the opponents, potential traps or threats, and pitfalls to avoid. If this information could be reliably extracted, a program could build on the experience gained by others. Of course, this problem is just a paradigm of the more general problem of learning from experience.

With the introduction of chess database programs (such as *ChessBase* and *New in Chess Base*), thousands of recent grandmaster games are now available in machine readable form. For each game, the data consists of the moves played, the names of the players, and the result. Unfortunately, there is no indication of whether moves were good or bad. Given this large, growing body of data, there must be some way of extracting useful information from it.

Building on the pioneering work of Chase and Simon (1973a,b) in chess cognition, the notion of chess position similarity is achieved through *chunking*, breaking positions apart into their basic familiar patterns. By emulating the methods used by humans for pattern recognition, chess programs can take advantage of human experience, and apply a more knowledge-based approach to their decision making process. This is in stark contrast to the current brute-force search tendency.

This paper describes the design of *MACH*, (a *M*aster *A*dvisor for *CH*ess). Briefly, *MACH* uses a knowledge base of user-defined chunks to access a collection of grandmaster games. The chunks have been defined by masters and correspond to familiar, frequently occurring chess patterns. All the positions in the available grandmaster games have been broken into their constituent chunks and organized into a database for quick retrieval. When a chess program encounters a new position, *MACH* chunks it, and then references the database looking for similar positions (as defined by the chunks present). From the database, *MACH* can extract relevant information on what the grandmasters did in these positions; this information can be as trivial as the move(s) played or as sophisticated as the plans or strategies followed.

This work has attempted to create a prototype process that will interface with a chess program to give it a more human-like and informed direction to playing chess. To play chess well requires knowledge and the abilities to search and reason with that knowledge. Current systems are largely search based; their reasoning processes are quite simple. *MACH* is an attempt to strike more of a balance between them.

MACH has been implemented and shown to provide useful advice to a performance chess program. Ultimately, the success of grandmaster chess programs will depend on their ability to build on others' experience. To work in isolation, computing moves for positions well known to humans, enhances the chance for errors. To know what has been played before in identical or similar positions is a tremendous advantage for helping decide on the move to play. Although our prototype implementation of *MACH* does not run in real-time for tournament use by a performance chess program, our production version of the program will achieve this.

2. Background

Humans are very good at learning board games. After given a brief introduction to the rules of the game, they usually play reasonably well. In fact, the moves made by a beginner are almost always legal, most often poor but usually never random (Eisenstadt and Kareev 1973). From the beginning, even inexperienced players generate primitive strategies (e.g. "capture pieces").

In the chess world, part of the strength of the chess master is his ability to determine the important features and characteristics of a position and recognize that they are similar to those of other games he has played or seen. These refined abilities allow him to extract relevant patterns, or *chunks*, from each position 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 method of play.

This chunking hypothesis was first developed by de Groot (1965) and later enhanced by Chase and Simon (1973a,b). Players were given 10 seconds to look at meaningful chess positions containing 20 to 25 pieces and asked to reconstruct them. Masters' re-constructive abilities were 93% accurate while beginners managed only 33%. These same players were later shown random positions, with pieces haphazardly strewn across the board. Surprisingly, the scores of all players dropped to 20%, illustrating that not only were the superior players subject to the same short-term memory constraints as the weaker players, but also when faced with meaningful subject material, they truly outperformed the less experienced.

By devising a *perception task*, Chase and Simon probed further into the thought processes of the master. Players were asked to reconstruct another set of positions but were allowed to look at each position as often as needed. Their head movements were used to segment the chunks of the position. Pieces within a single chunk are closely related in terms of attack/defense properties of the pieces as well as common color, type and proximity. Moreover, these pieces were placed significantly quicker than those not in chunks.

From these results, it is evident that the first few seconds after any move are important. Posnyanskaya and Tikhomirov (1969) monitored the eye movements of a chess expert as he examined a position, recording 20 different fixations that rested on pieces important to the position. In a similar vein, Simon and Barenfeld (1969) developed a program, called *PERCEIVER,* that was able to duplicate the eye movements of this expert by adhering to the simple relations of attack and defense. Their results clearly illustrate that an experienced chess player first examines a position for relevant information before applying a particular strategy.

Extending *PERCEIVER* into a system called *MAPP,* Simon and Gilmartin (1973) reinforced the chunking hypothesis by subjecting *MAPP* to the same board reconstruction experiment that the human players faced. By determining the patterns present on the board, and restricted to the same short-term memory constraints as humans (seven plus-or-minus two patterns per position (Miller 1956)), *MAPP* was able to reconstruct positions with 73% accuracy.

A key issue involved with these ideas is the derivation of chunks. Bratko, Tancig and Tancig (1986) sought to devise a natural method for detecting positional patterns in chess. By combining the most frequently placed pieces from reconstructed positions of a group of masters into a *collectively reconstructed* position, Bratko et al., found an average of 7.54 chunks per position. Campbell and Berliner (1984) were also able to derive chunks from positions, though it was done artificially and not based on psychological results.

The first chess program to apply the chunking theory and model human problem solving more closely was developed at USC by Zobrist and Carlson (1973). It first searched a given board position for all instances of certain patterns, recorded them internally into a *snapshot*, and searched ahead using these snapshots to evaluate board positions. Little additional research has followed up the USC system, though a number of chess cognition experiments have been conducted (for example, Bratko and Michie (1980) and Kopec, Newborn and Yu (1986)).

In the context of chess programs that use experience, two vastly different usages can be found. Hartmann (1987a,b) has done some work on using a database of grandmaster games, but only as a source of test data for tuning a program's evaluation function. Slate (1987) used transposition tables as a means of learning through experience. However, the experience used is only the program's and is useful only when identical positions can be found.

The usage of chunking in human chess play has been firmly established but, to date, implementations have been simple systems (Campbell's (1984) program playing king-and-pawn endgames being a notable exception, but without real-time constraints).

3. Similarity and Chunking

It is evident that the existence of the chunking hypothesis presupposes the ability to extract relevant patterns from a position leading to a determination of a similarity measure. The study of cluster analysis is able to supply an application-independent algorithm for measuring similarity but the intricacies of chess and chess-specific knowledge discourages use of such methods. Consequently, we will follow the philosophy of Zobrist and Carlson (1973), abandoning the use of expert programmers, and strive towards methods which allow chess masters to easily express chunks to us.

By designing a language specific to chess, it becomes easier to express and, therefore, capture chess specific information. Further, if that language can be expressed graphically, in terms of familiar chess patterns, then human experts can be used as a source of information. Computer chess researchers have always had the difficulty of obtaining useful research data from human experts. This *knowledge acquisition bottleneck* occurs because the scientists talk one language and the domain experts, another. Since our chunking project requires defining human familiar chess patterns, a graphical interface would allow the experts to converse with the program in their language (chess) and have the interface translate this into a machine understandable form. The following paragraphs describe the textual computer language used by *MACH*. Although thought has gone into the design of a graphical interface, it remains unimplemented.

To determine what features the language should entail, we can use the results of *PERCEIVER* and *MAPP*. At the very least, the positional features adhered to by these systems (namely, attack/defense properties and commonality of piece-color, type and proximity) should be included in our language. Along the same lines, there should exist a set of *required* pieces as well as a set of *optional* ones. Of course, those in the required set must exist in a position for the chunk to exist. On the other hand, those in the optional set, if present in a given position, should be included as part of a chunk's makeup. The incorporation of these two sets can easily identify the pieces that were missed by the chunking process.

Some non-positional attributes a chunk should possess include hierarchical and instantiation features. For example, our environment should be flexible enough to allow one chunk to be defined in terms of other, less complex, ones so that a re-specification of the smaller chunks is not necessary. In addition, chunks should be able to have parameters, reducing the need for repeating specifications.

Not only does our similarity measure rely on the quantitative characteristics just mentioned but it also depends on a number of qualitative features - importance and advice, in particular. The presence of some chunks may be more important than others in determining the dominant characteristics of a position. An importance measure attached to a chunk may help eliminate irrelevant, unimportant features when determining similarity. Assuming a chunk's presence is relevant, the system allows the user to specify *advice* that is passed back to the chess program. This facility assumes that the chess program has the capability to interpret the advice.

Figure 1 shows one of the chunks used for obtaining the results in section 5.

4. Data Structures

Given the need for real-time constraints, it is important to have the database of grandmaster games chunked, position by position, and stored in a database that allows quick access. Following is a brief discussion of the data structures used; further details can be found in George (1988).

The database of games is represented as a trie (the *GameTree*), with nodes representing positions and branches, moves. All redundant opening sequences of moves have been eliminated to reduce the size of information to be processed. No attempt has been made to eliminate transpositions (identical positions reached by different move sequences). Conventional trie usage is for search efficiency, to look for a datum by following a path from the root to terminal node. For our application, index files (discussed below) are used to point into the middle of the data structure. From an arbitrary node in the tree, the position it represents can be re-constructed by following parent nodes to the root. Then, from the initial starting position of a chess game, re-tracing the path back down the tree from the root and playing the moves along the branches will yield the desired position. Thus, our trie need only store moves (12 bits per move) rather than

```
# The following lines describe a sample chunk where each '#' symbol designates
# a comment line. This chunk, named 'qgdcenter', is described with respect to
# (wrt) white.  An additional program will check for the mirror images of the
# chunk.  Chunks are organized into classes; this chunk is classified as being
# a member of the 'queens_pawn_opening' group of chunks.  The 'require' state-
# ments specify the piece/square combinations that must exist for the chunk to
# be present.  Statements without a predicate correspond to 'optional' piece-
# square combinations.  The symbols, '&' and '|', refer to the logical 'and'
# and 'or' operators respectively, while the symbols, '+' and '/', refer to
# special conjunction and disjunction operators for piece-square combinations.
# The 'advice' statement specifies plans frequently followed when the chunk is
# present.

chunk qgdcenter wrt white.
classification queens_pawn_opening.
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.
advice          PutRooks(white,C1,D1), Outpost(white,E5), Outpost(black,E4).
end qgdcenter.
```

Figure 1. Sample Chunk.

positions (roughly 20 bytes, depending on the sophistication of the encoding scheme).

The search for similar positions in the trie must be done in an efficient manner to achieve real-time performance. Thus, all positions in the tree can be chunked beforehand and stored in index files that allow for easy access. The simplest way of doing this (although not the most efficient) is to have a file for each chunk which contains pointers into the *GameTree* for each position containing that chunk. However, since there may be an enormous number of chunks in a full-fledged system, it was decided to order the chunks into a hierarchy. A group of high-level *features* (or *chunk classes*) were defined, with each chunk classified as being a representative of a feature. For example, one of the features is called "castled king-side". Under that heading, all chunks that represent a king castled on the king-side would be grouped. In Figure 1, the *qgdcenter* chunk was classified as being part of the *queens_pawn_openings* feature. The list of these features is called the *FeatureIndex.* For each feature *M*, a file *(C-ClassM)* was created containing pointers into the *GameTree* to all positions that contained that feature.

The program *build* takes a database of grandmaster games, creates the *GameTree,* and chunks each position. For each chunk found, a pointer into the *GameTree* database is made in the corresponding *C-Class* file, identifying the position containing that chunk. This is illustrated in Figure 2.

Figure 2. ***Build*****ing the Databases.**

During a game, it is necessary to chunk the current position and identify positions in the *GameTree* that are *similar* to the current one. Similarity would be easy if two positions were considered similar only if they contained identical chunks. Unfortunately, any similarity measure must be more robust. Hence, if there are M features present in a position, our system considers a position similar if it has $M - R$ of the same features. By having a sub-set of the chunks to match, we allow for possible errors in chunk definitions. When $R > 0$, one must be concerned with considering all possible subsets of the $M - R$ chunks, introducing a combinatorial complexity. In our system, $R = 1$ and $R = 0$ have been experimented with, demonstrating that robustness ($R = 1$) is necessary.

The simplest way of finding all positions that have $M - R$ features in common is to take the M feature files for the features identified, merge them, and flag all positions that occur $M - R$ times in the merged file. Although in terms of execution speed this method is not desirable, it has the advantage of being simple to implement. For our initial implementation, this method was chosen because it allowed us to quickly get a working version of the program. The issue of identifying matching subsets is an efficiency issue and does not affect the correctness of *MACH.*

The alternative is to pre-compute all the possible chunk combinations beforehand, so that when needed, it is easy to retrieve the desired combination of chunks. Although this allows for fast retrieval, it may require storage exponential with respect to the number of possible chunks. We have designed data structures that address this issue but they have not been implemented. More details can be found in George (1988).

Figure 3 illustrates the process used to access the databases during a game. The current position is chunked and L chunks are identified in the position. The chunk features indicate which $C - Class$ files to access. Note that more than one chunk could be in the same feature class, so $M \leq L$ files are accessed. These M files are merged and positions occurring $M - R$ times are identified. This is the first filter of the system. These positions match the current position in terms of global features. The second filter matches the positions to the specific chunks found in the position. The positions remaining after the first filter are compared for the presence of the L chunks identified as being present in the current position. From these, a smaller set of positions is identified that match at least $L - R$ chunks. Finally, the remaining candidate positions pass through a third filter. The final filter contains some *ad hoc* similarity measures to identify positions with gross defects and throws them away. For example, if a position matches in every respect except king position, we may still want to reject this position. If the king is supposed to be on $e1$ but is in fact on $h8$ then there is no similarity at all! In effect, this final filter attempts to identify inadequacies in the first two filter phases. Given sufficient chunk information in filters 1 and 2, this third phase would be unnecessary.

From this final set of positions, *MACH* determines the advice to give. This advice could be as simple as the moves made in that position or as detailed as the plans used by the opponents in the subsequent play. Currently, *MACH* only provides the list of moves made in the similar positions. Of course, some of those moves may even be illegal in the current position. Increasing the sophistication of the advice requires more chess knowledge for identifying items of interest in the subsequent play from the similar positions.

Figure 3. Finding Similar Positions.

5. Results

MACH's abilities to find similar positions was tested on a variety of data including a test suite of 20 positions, 10 each arising from well known openings†. The positions were generally 7 to 15 moves into the game. Twelve chunk definitions were used which, with reflections, gave 34 different chunks. 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 the roughly 60,000 positions we had available (from 850 games).

For each of the test positions used in this experiment, *MACH* found an average of 5 similar positions in the *GameTree*. Chunking each of the *GameTree* positions 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.

Figure 4 gives the experimental data for the 20 positions using the similarity criterion that $M - 1$ 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* (1973) and confirms their results. Our 6.7 pieces/chunk average is significantly higher than *MAPP*'s 3.9 figure and De Groot's average of between 3 and 4 (de Groot 1965). This is a result of the *optional* piece specification, something that Simon and Gilmartin did not use. If we check the number of *required* pieces only, this value drops to 3.4.

On the memory test, *MACH* is able to re-construct 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

Figure 4. MACH's Performance on 20 Positions.

MACH has been used to give advice to the chess program *Phoenix*†, and has been useful in overcoming a well-known problem in computer chess. The first few moves in a game made by a chess program come from a file or *book*. Once these moves are exhausted, the program is on its own and usually has no idea how to proceed. Often at this juncture, programs make critical (even fatal) mistakes. *MACH* eliminates this problem by providing useful advice that is consistent with the previous play.

† The Ruy Lopez and Queen's Gambit Declined.

† A frequent competitor in the world computer chess championship.

Phoenix has played game fragments incorporating *MACH*'s advice. To do this, *Phoenix*'s position evaluation function was modified to take into consideration the list of moves suggested by *MACH*. If a position is reached via a suggested move, the evaluation function score for that position is increased, making it more desirable. The moves provided by *MACH* are not guaranteed to be chosen; rather the probability is simply increased. Since *Phoenix* has its own expertise on what constitutes a good move, and there is no assurance that *MACH*'s suggestions are worthwhile, the program should not blindly follow the advice.

Figure 5. ***Phoenix*** **using** ***MACH*****'s Advice.**

The relationship between *MACH* and *Phoenix* is shown in Figure 5. *MACH* has identified similar positions and returned the set of moves made in these positions to *Phoenix*. During *Phoenix*'s search of the current position, if it plays a move (such as *a*) that *MACH* suggests, then the leaf node evaluation of that line is incremented by a weight associated with that move (*w*). In this way, evaluations that are a result of moves suggested by *MACH* have higher scores and, therefore, a better chance of forming the principal variation.

MACH is frequently able to suggest moves better than what *Phoenix* would normally play, leading to a noticeable improvement in performance. We have several examples where *MACH*'s advice has even saved *Phoenix* from making a losing move.

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 and Gilmartin did not provide any insight into this area; their chunks comprised patterns that frequently occurred in "...games in the published literature" (Simon and Gilmartin 1973). Other than mentioning that their patterns adhered to the relations of attack and defense, 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.

It was 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 and Gilmartin (1973) 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 multiplies the number of chunks that will match and has a tendency to spread pieces apart. On the other hand, over use of the "and" operator will cause the opposite effect. Both operators should be used carefully.

The following simple feedback methodology was used for adding chunks:

(1) Define the chunks

(2) Set up test position(s)

(3) See which positions did and did not match

(4) If satisfied, stop, else go to step (1)

To a large degree, steps (2) and (3) can be automated. As well, step (1) can be improved by providing the chess master with a graphical interface. Since the image of the pattern is certainly more meaningful to the expert than a 6-line description in our language, this tool would place the expert in a more familiar environment in hopes of invoking the thought processes that are involved in an actual game.

Although the feedback process is slow, it is necessary for 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.

6. Conclusions and Future Work

The system, as it currently stands, requires three major software projects to make it a useful tool. First, the original design concentrated on functionality rather than efficiency. A re-implementation, using chunk recognition methods similar to those used by Simon and Gilmartin's (1973) *EPAM*-like pattern learner, is necessary to achieve real-time performance. Second, tools are needed to ease the chess masters' difficult task of expressing their knowledge to the system. As mentioned, a graphical interface is mandatory to allow the master to converse with the system in a manner natural to him. Finally, tools to analyze chunks for being too general, too specific, or too similar to existing chunks are needed to reduce the effort required to ensure chunk orthogonality. At present, the above ideas are in the analysis stage.

An open research problem is building a system that can generate chunks automatically. Perhaps by feeding a program hundreds of thousands of grandmaster games, such a system could automatically detect frequently occurring patterns.

The existence of a database of useful chunks could be applied to related types of reasoning. The use of analogies, a dominant feature in human play, has been largely ignored in real-time game programs (Adelson-Velsky, Arlazarov and Donskoy (1975) being an exception applied to chess tactical searches). The usage of chunking has the promise to use knowledge in a more meaningful way in game-playing programs, by substituting for the ever-dominant exhaustive brute-force searches that are prevalent today.

Ultimately, one must utilize experience to enhance program performance. To continually confront a program with common place scenarios, which it treats as a new situation every time, is counter-productive and error prone. Utilizing experience allows a program to improve its capabilities *without additional programming effort.*

Acknowledgments

Financial support from the Natural Sciences and Engineering Research Council of Canada is greatly appreciated. Dap Hartmann kindly provided the collection of grandmaster games used for the experiments.

References

Adelson-Velsky, G.M., V.L. Arlazarov and M.V. Donskoy (1975), "Some Methods of Controlling the Tree Search in Chess Programs," *Artificial Intelligence*, vol. 6, pp. 361-37.

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

Bratko, I., P. Tancig and S. Tancig (1986), "Detection of Positional Patterns in Chess," *Advances in Computer Chess 4,* D. Beal (editor), pp. 113-126.

Campbell, M.S. and H.J. Berliner (1984), "Using Chunking to Play Chess Pawn Endgames," *Artificial Intelligence,* vol. 23, pp. 97-120.

Chase, W.G. and H.A. Simon (1973a), "Perception in Chess," *Cognitive Psychology*, no. 4, pp. 55-81.

Chase, W.G. and H.A. Simon (1973b), "The Mind's Eye in Chess," *Visual Information Processing: Proceedings of Eighth Annual Carnegie Psychology Symposium*, W.G. Chase (editor), Academic Press, New York.

de Groot, A.D. (1965), *Thought and Choice in Chess*, The Hague, Mouton.

Eisenstadt, M. and Y. Kareev (1973), "Toward a Model of Human Game Playing," International Joint Conference on Artificial Intelligence, pp. 458-463.

George, M. (1988), "MACH - A Master Advisor for Chess," M.Sc. thesis, Department of Computing Science, University of Alberta.

Hartmann, D. (1987a), "How to Extract Relevant Knowledge from Grandmaster Games, Part 1: Grandmasters Have Insights - The Problem is What to Incorporate Into Practical Programs," *Journal of the International Computer Chess Association*, vol. 10, no. 1, pp. 14-36.

Hartmann, D. (1987b), "How to Extract Relevant Knowledge from Grandmaster Games, Part 2: The Notion of Mobility, and the Work of De Groot and Slater," *Journal of the International Computer Chess Association*, vol. 10, no. 2, pp. 78-90.

Kopec, D., M. Newborn and W. Yu (1986), "Experiments in Chess Cognition," *Advances in Computer Chess 4*, D. Beal (editor), pp. 59-79.

Miller, G.A. (1956), "The Magical Number Seven, Plus Or Minus Two: Some Limits On Our Capacity For Processing Information," *Psychological Review*, no. 63, pp. 81-97.

Posnyanskaya, E.D. and O.K. Tikhomirov (1969), "On the Function of Eye Movements," *Soviet Psychology*, no. 1, pp. 25-30.

Simon, H.A. and M. Barenfeld (1969), "Information-Processing Analysis of Perceptual Processes in Problem Solving," *Psychological Review*, vol. 76, no. 5, pp. 473-83.

Simon, H.A. and K. Gilmartin (1973), "A Simulation of Memory for Chess Positions," *Cognitive Psychology*, no. 5, pp. 29-46.

Slate, D.J. (1987), "A Chess Program That Uses its Transposition Table to Learn From Experience," *Journal of the International Computer Chess Association*, vol. 10, no. 2, pp. 59-71.

Zobrist, A.L. and F.R. Carlson, Jr. (1973), "An Advice-Taking Chess Computer," *Scientific American*, vol. 228, no. 6, pp. 92-105.