View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Bath Research Portal

Citation for published version:
Caulfield, T 2004, Acquiring and using knowledge in computer chess. Computer Science Technical Reports, no. CSBU-2004-17, Department of Computer Science, University of Bath.

Publication date:
2004

Link to publication

©The Author May 2004

University of Bath

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 12. May. 2019

Department of
Computer Science

Technical Report

Undergraduate Dissertation: Acquiring and Using Knowledge in Computer Chess

Tristan Caulfield

Technical Report 2004-17 May 2004

ISSN 1740-9497

Copyright ©May 2004 by the authors.

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Acquiring and Using Knowledge in Computer Chess

Tristan Caulfield
BSc (Hons) Computer Science
University of Bath

May 2004

Abstract
Traditional computer chess methods use a high-speed heuristic-based search to evaluate positions and decide which move to play. This project examines the use of knowledge in chess by both humans and existing computer programs and then implements an addition to GNU Chess that uses the generalized generalized hebbian algorithm to acquire and use knowledge while playing. It is found that this method works in some cases and with further research could be extended to become more powerful.

Acquiring and Using Knowledge in Computer Chess

submitted by Tristan Caulfield

COPYRIGHT

Attention is drawn to the fact that the copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of this work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university of institution of learning. Except where specifically acknowledged, it is the work of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purpose of consultation.

Signed: ....

Acknowledgments
I would like to thank Joanna Bryson for her wonderful supervision, and my brother Adrian and friend Richard for their proofreading efforts.

Contents

1 Introduction **1**

2 How Humans Play **2**
2.1 de Groot's Study . . . 2
2.2 Chunking . . . 3

3 Traditional Computer Chess **4**
3.1 A Brief History of Computer Chess . . . 4
3.2 The Minimax Tree . . . 5
3.2.1 α-β Pruning . . . 6
3.2.2 Transposition Tables . . . 7
3.2.3 Progressive and Iterative Deepening . . . 7
3.2.4 Drawbacks to Minimax . . . 7
3.3 Evaluation Algorithms . . . 8
3.3.1 Quiescence . . . 8

4 Knowledge in Chess **10**
4.1 Why Knowledge? . . . 10
4.2 Plans . . . 11
4.2.1 Planner . . . 11
4.2.2 PARADISE . . . 12
4.3 Acquiring Knowledge . . . 13
4.3.1 Human Input . . . 13
4.3.2 Automatic Acquisition . . . 13

5 Clustering **15**
5.1 Principal Component Analysis . . . 15
5.2 Generalized Hebbian Algorithm . . . 16
5.3 Generalized Generalized Hebbian Algorithm . . . 16

6 Implementation **21**
6.1 GNUChess . . . 21
6.2 What Was Implemented . . . 21
6.2.1 The GGHA Algorithm . . . 21
6.2.2 Evaluation . . . 22
6.2.3 ChunkQuery . . . 23

6.3 Problems 23

7 Experiments and Results **25**
7.1 Testing the GGHA Implementation 25
7.2 Reconstruction of Positions 25
7.3 ChunkQuery 27
7.4 Evaluation 28

8 Conclusions **30**

List of Figures

3.1 The Horizon Effect 6
3.2 Alpha-Beta Pruning 7

4.1 *Nuchess* (white) to play, against *Cray Blitz* 11
4.2 PARADISE finds mate in 19 ply 12

5.1 The error in the network decreases with each epoch. 18
5.2 3D projection after 5 epochs. 18
5.3 3D projection after 15 epochs. 19
5.4 3D projection after 15 epochs, showing feature vectors. 20

7.1 Position in a game. 26
7.2 Reconstruction of Figure 7.1 using 4 of 100 chunks and a threshold value of 0.6. 26
7.3 Reconstruction of Figure 7.1 using 4 of 100 chunks and a threshold value of 0.5. 27
7.4 Scores 29

Chapter 1

Introduction

Humans have been playing chess for thousands of years while computers have been playing for less than a hundred, yet there exist chess programs which can rival the greatest minds in the game. They achieve this by using their strength — raw computational power — to analyze millions of moves and select the best one. Humans, not blessed with custom hardware for chess, only analyze a couple hundred positions at most and must rely on knowledge rather than brute force to win the game.

Most of the effort towards improving computer chess engines has gone to finding ways to speed up and increase the depth of the computer's search. This focus on an increase in speed and depth has produced significant advances, but there is potential for equally significant progress by using knowledge to aid and eventually replace search.

Most programs to date that have attempted to use knowledge have had their knowledge defined for them by a human. This work has shown that knowledge can be a very powerful tool, but has not done very much more than that because of the dependency on human masters for knowledge input. Attempts to automatically gather this information have mostly tried to emulate human thought, and most of the programs that do this only seek to create a model human cognition, not store and utilize knowledge.

These programs or models for the most part follow the chunking theory, started by de Groot (1965) and continued by Chase & Simon (1973) among others. This is usually implemented with some sort of EPAM-like model (Feigenbaum & Simon 1961) with a discrimination net to learn and store chunks, the groups of pieces that serve as indices into a players library of knowledge. An alternative method, proposed by Hyötyniemi & Saariluoma (1998), follows the basic ideas of the chunking theory but uses clustering to identify and recognize chunks.

This method is used here to analyze master-level games and break them down into chunks. Knowledge is then associated with the chunks and its effect on the performance of a chess engine is examined.

Chapter 2

How Humans Play

The human mind is incredibly slow at mathematical calculations when compared to a computer, but it is very good at remembering and recognizing patterns and their significance. A study by de Groot (1965), started investigations into the way that humans store and use knowledge, particularly in chess. If a chess program can use knowledge in a similar manner then it might be possible to increase its performance and skill.

2.1 de Groot's Study

de Groot (1965) performed a series of experiments on chess players varying in skill from amateur to grand master to try and examine the human thought process during a game of chess. In one of the experiments, chess players were asked to look at a position on the board and consider the next move to be made, while speaking their thoughts out loud. Surprisingly, the expert players did not consider or evaluate more moves than the amateur players, but the moves they considered were in general stronger. Additionally, the expert players tended to think about the board in terms of groups of pieces, while the amateur players thought about pieces individually.

A second experiment had the players of different ability reconstructing a position from a game unknown to them after being shown it for a small amount of time. de Groot (1965) found that the master-level players had an almost perfect ability to reconstruct the position, while the ability of the amateurs was considerably less. This can be explained by the fact that the expert players think of pieces in groups — they have less to memorize than the amateur players who must memorize the location of every piece individually.

That expert players think in terms of groups of pieces, or *chunks*, also explains why the experts considered only stronger moves when asked to examine a position. When they are presented with an unfamiliar position, the experts can identify groups of pieces on the board that are familiar to them and provide insight as to the best moves to consider. The expert has a large library of knowledge and the chunks provide a means to quickly identify and retrieve the needed knowledge.

2.2 Chunking

The ideas proposed in de Groot (1965) were verified and expanded upon by Chase & Simon (1973), who developed the theory of *chunking*. They re-created de Groot's reconstruction experiments where players were shown a position from a game for a short amount of time and asked to replace the pieces, and confirmed that experts are a lot better at reconstructing the position. Importantly, they also showed the players random positions not taken from any game and found that the experts were no more successful than the amateurs in recalling these random positions. When shown a position from a game, the experts recognize chunks which they can store in short-term memory and use to reconstruct the board. Amateurs without a vast collection of chunks at their disposal must try and remember the position of each piece individually, too much information to store in their short-term memory. When the position is random, however, the experts can not find any chunks and, like the amateurs, must try and remember individual piece locations.

The expert acquires his knowledge through years of study, building up a vast database of information about the game. As observed by de Groot (1965), players recognize familiar patterns on the board and remember information about when that pattern was used before. If a computer chess program can decompose master-level games down into chunks, it can then begin building up knowledge about what those chunks indicate when they are present. When a chunk is recognized during game play, the computer can use the information it has stored about that chunk.

Chapter 3

Traditional Computer Chess

3.1 A Brief History of Computer Chess

The first mechanical chess engine was Baron Wolfgang von Kempelen's 'Automaton Chess Player' in 1770 (Hayes & Levy 1976). This was later revealed to be a hoax–the machine actually had a man hidden inside–but since then incredible progress has been made and computers are capable of challenging the world's top chess players. Around 1900, Torres Quevedo invented a mechanical machine that could win endgames of a King and a Rook against a King from any position on the board. The machine, which could actually move the pieces as well as compute the moves, was supposed to have been able to force mate in 50 moves; it was later proved that some cases required more, but this is still quite impressive.

Towards 1950, the foundations of computer chess as we know it today began to emerge. In 1948 Alan Turing designed several programs for playing simple games of chess, and independently, Claude Shannon (Shannon 1950) wrote one of the most influential papers on the subject. The ideas in his paper, which will be discussed in detail later, are still used in computer chess engines today. After this paper, chess programs were written that incorporated its ideas, and programs of this type were soon the most common, though there were plenty of others. The first computer against computer match took place in 1967, when a program developed at MIT and Stanford played against a program written in the Soviet Union (which won the match). Also in 1967, the program MacHACK gained a lot of attention for performing particularly well. It went on to defeat a human, Dr. Hubert Dreyfus, who was a great critic of artificial intelligence.

Computer chess tournaments became more common and the computers running them became faster as time went on. In 1970 the first tournament was held by the Association of Computing Machinery and this competition has been held annually since then. Each year more competitors entered and achieved greater and greater levels of play.

In the mid-1980s through the mid-1990s, the strongest chess computers around were the 'Deep' series from IBM. They defeated many grand masters and, at their peak, Deep Blue defeated world champion Gary Kasparov. Although the circumstances of this match are sometimes questioned, it is clear that computers can now at least equal the best human

players.

3.2 The Minimax Tree

Most chess programs today (and many other game programs as well) use some form of a minimax tree, as originally described by Shannon (1950). The tree represents the moves possible in the game. At the first layer are possible moves by white, at the second layer moves by black, then white again, and so on. Each of the possible positions resulting from a move is assigned a value according to an evaluation function. When it is white's turn to choose the move, white should choose the move that maximizes the score; when it is black's turn to move, black should choose the move to minimize the score. This method assumes that the opponent always makes the best move, the one that will give them the highest score while minimizing your own. As Shannon (1950) writes in his article (M is a possible move and $f(P)$ is the evaluation function for position P):

> A deeper strategy would consider the opponent's replies. Let $M_{i1}, M_{i2}, ..., M_{is}$ be the possible answers by Black, if White chooses move M_i. Black should play to minimize $f(P)$. Furthermore, his choice occurs *after* White's move. Thus, if White plays M_i Black may be assumed to play the M_{ij} such that $f(M_{ij}M_iP)$ is a *minimum*. White should play his first move such that f is a maximum after Black chooses his best reply. Therefore, White should play to maximize on M_i the quantity $min(f(M_{ij}M_iP))M_{ij}$.

Chess is a game of perfect information and known rules, and so in theory one could compute the entire tree of moves, select a winning leaf at the edge of the tree, and then follow the moves to get to that point and always win. However, the sheer number of possible moves in chess prevents this, and is the main limiting factor in computer chess programs of this type. According to Shannon, There are an average of 30 legal moves at any point in a chess game, and an average game lasts about 40 moves until one player resigns. This means that there around 10^{120} moves to be calculated in order to see the entire tree! Even after a few levels of the tree, the number of possible moves becomes extremely large and takes a long time to calculate. Early chess programs were only capable of looking a few ply[1] ahead because they lacked the processing power and memory to be able to do this in a reasonable time. Even today's best chess computers still have a limit to how far they can see into the game.

Because computers have limited resources and can not see far into the future, they suffer from the Horizon Effect. In Figure 3.1, taken from Frey (1977), even a beginner can see that white has an obvious win. If the computer was limited to looking 3 ply ahead, it would not be able to see the possibility for pawn promotion. Its evaluation function would give the position a rating in favor of black, who has more pieces. If the computer looked 5 ply ahead, it would still not be able to see the pawn promotion. If offered a draw in this position, a computer playing white would accept, believing itself to be at a disadvantage. The computer

[1] A ply is half of a complete turn, so one move for either black or white.

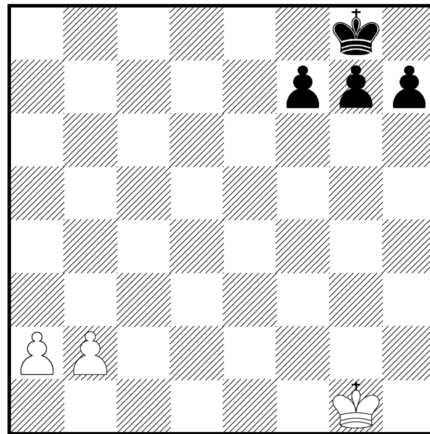


Figure 3.1: The Horizon Effect

needs to look ahead 9 ply in order to even realize that the Rook Pawn can reach the back rank. Even modern chess engines capable of looking many ply into the game suffer from this effect. It is entirely possible for a computer to miss a checkmate or fail to recognize a good move because it searched one ply too few.

3.2.1 α-β Pruning

Because the minimax tree is so important in chess engines, there have been many methods and algorithms developed to improve its performance. The α-β pruning algorithm improves the minimax tree by significantly reducing the number of nodes that have to be searched. Figure 3.2 shows a small tree with nodes numbered in the order that they are evaluated. The search is depth-first, so the tree is expanded to either a natural or artificial (created by a limit on search depth) leaf. Node 3 is the first leaf node encountered in this example, and the evaluation function assigns it a value of +1 (in white's favor). Node 4 is the next leaf evaluated, with a score of +2. Since all of the children of node 2 have been evaluated, node 2 gets assigned the value of its lowest child, +1. The next leaf that is encountered is node 6. When evaluated, this gives a value of +0. At this point, the computer knows that it does not have to evaluate any of the other children of node 5. This is because white already knows it can get a value of +1 by choosing node 2, and that if it chose node 5, black could choose node 6, giving white a lower score. The algorithm is called α-β because α is used to represent the best move for white and β is used to represent the best move for black found so far in the search. This algorithm reduces the number of nodes in the tree and produces exactly the same results as a full minimax search.

When using α-β pruning, the order that the moves are evaluated can make a large difference to the total number of nodes that need to be evaluated. If a stronger move is evaluated first, the number of other moves that can be discarded is greater. Heuristics are used on many chess engines to attempt to find the stronger moves so they can be evaluated first.

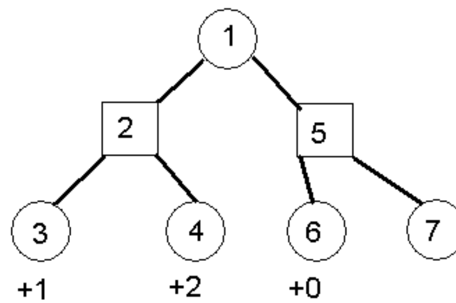


Figure 3.2: Alpha-Beta Pruning

3.2.2 Transposition Tables

When searching a large tree of moves, it is possible that some of subtrees or moves will be similar or the same. A transposition table serves three purposes (Marsland 1991). The tables store the merit (or value), best move, and status of nodes and subtrees. The first purpose of the table is to narrow α and β values of the search based on the value of an entry in the table. The second purpose is to allow the best moves from a subtree to be run first if a subtree is re-encountered. The last and most important use of a transposition table is to remember subtrees that have been searched already, preventing repetition in the search.

3.2.3 Progressive and Iterative Deepening

When performing a study of chess masters and their thought processes, de Groot (1965) noticed that the masters chose to expand certain parts of the tree selectively. He called this progressive deepening, as the tree was evaluated to a certain depth and then the best branch chosen as a new root node and expansion continued from there. This has been implemented in several programs (**?**), starting with a search of 1 ply, then 2, and so on. It seems to be a fairly efficient method.

3.2.4 Drawbacks to Minimax

Programs that are tree-based inevitably have some sort of artificial limit to the depth of their search. Theoretically, given infinite processing power and storage, they are perfect programs, capable of seeing to all of the leaves of the tree and selecting moves which always lead to a win. However, because they have an arbitrary limit due to limitations in processing power, it is always possible that they will fail to see a good move, or fail to notice an attack because they are essentially blind to the significance that moves might have a long way into the future. Techniques such as progressive deepening, transposition tables, and α-β pruning help to make it possible to increase the depth of searches, and this provides a little relief from the problem, but is not a solution (Schaeffer 1983).

Berliner, Kopec & Northam (1990) notes that programs using minimax without knowledge might not know the significance of positions on the board from the position itself, but that the strength or weakness of the position will become evident as the tree is searched. However, the depth required to see this advantage or disadvantage may be quite high; as an example, Berliner et al. (1990) gives the doubled isolated pawn: "a program that does not understand the weakness of a doubled isolated pawn will probably have to search to depths of up to 40 ply to discover this fact from more primitive features such as material." Using knowledge the weakness would be seen much sooner.

3.3 Evaluation Algorithms

In Shannon (1950), a very general algorithm for evaluating a position is described:

$$f(P) = 200(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + (P - P') - .5(D - D' + S - S' + I - I') + .1(M - M') + ...$$

in which:-

- K,Q,R,B,N,P are the number of White kings, queens, rooks, bishops, knights and pawns on the board.
- D,S,I are doubled, backward and isolated White pawns.
- M= White mobility (measured, say, as the number of legal moves available to White).
- Primed letters are the similar quantities for Black.

This simple function weights the different pieces with different values, the queen being worth the most, down to pawns being worth the least. Penalties are specified for pawns in bad positions, and a value of 200 is assigned to mate. Evaluation functions have advanced since this, and they now include many more heuristics for calculating the value of a position. However, these functions evaluate the position only on what they see before them, not with regards to any long term plan. This can mean that their goals change from move to move, and that moves are only reacting to the situation.

As the search is conducted, each position encountered is evaluated with this function. If it were possible to fully expand the tree, so the leaves were endings to the game, no evaluation function would be needed because the value of all the nodes are already known. Since it is impossible to fully expand the tree, the evaluation function must estimate the value of a node, making it a critical part of the program. If things are evaluated incorrectly then increasing the search depth will just increase error. However, a complex evaluation function takes longer to calculate, slowing the program down.

3.3.1 Quiescence

When evaluating nodes in the tree, it is possible to come up with very incorrect values for a position if the position is a dynamic one. As a trivial example, say the tree search reaches

its maximum depth right after white has captured black's queen in a trade. The algorithm will not check any further nodes, and thus evaluate the position as very good because it believes that white is up a queen, when in reality black will take white's queen in the next move. The evaluation function works its best when applied to positions that are static, or *quiescent* (Frey 1977). In order to fix this problem, quiescence checks were added to the search algorithms. When a terminal node of the tree is reached, a check is carried out to see if the position is a dynamic one. A dynamic situation will have pieces in the middle of movements or possibly capturable. In these cases, the search evaluates the sub-nodes of the dynamic node, until it reaches a static node where it can perform a proper evaluation.

Chapter 4

Knowledge in Chess

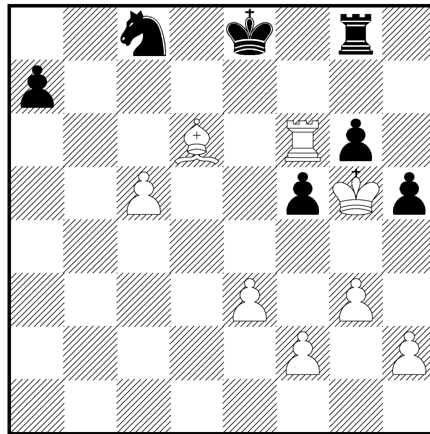
4.1 Why Knowledge?

At the most basic level, knowledge can help to make the evaluation of a position more correct, which reduces errors and makes the moves selected stronger. Knowledge of the game can also be used in deciding which lines of play to search and which to ignore, allowing most of the time spent searching to be concentrated on moves that are likely to yield a good result. At the highest level, knowledge can be used to create and follow plans in the game, using strong combinations of moves created to achieve a goal rather than discovered by search.

Marsland (1987) discusses how knowledge can be used, and gives Figure 4.1 as an example of what might happen when it is not. *Nuchess* gains a pawn, but doing so allows *Cray Blitz* to advance its pawn and promote it. A human player looking at the board would see that black would have a passed pawn (a pawn that has passed all of the pieces in its way and is free to advance to the back rank) if Rxg6 was played, realize that this meant black would be able to promote the pawn, and make a different move to prevent this from happening; the computer did not have knowledge of passed pawns or what they meant in the game and did not search deeply enough for the mistake to manifest itself in the tree. Modern programs now have knowledge about this particular situation in the form of passed pawn routines which are called when a position is evaluated, but there are still many cases where potential threats or advantages are not seen by the program because the search does not get deep enough to reveal their effects.

Adding extra knowledge about the passed pawns and other situations into the evaluation function was the response used by most programs that rely on a minimax search. It does help compensate for the inability to see the full effect of moves through search, but knowledge can be better used not augmenting the search but reducing the number of moves that have to be searched at each position. If instead of examining all possible lines of play from a location knowledge is used to select only a few of the best, this will greatly reduce the combinatorial explosion cause by chess's very high branching factor.

Understanding the significance of certain features within the game can help guide the



45.♖f6xg6? ♖g8xg6+ 46.♔g5xg6 ♘c8xd6 47.Pc5xd6

Figure 4.1: *Nuchess* (white) to play, against *Cray Blitz*

search, avoid problems, and utilize advantages, but there is a lot more that can potentially be done with that knowledge, such as creating and following plans.

4.2 Plans

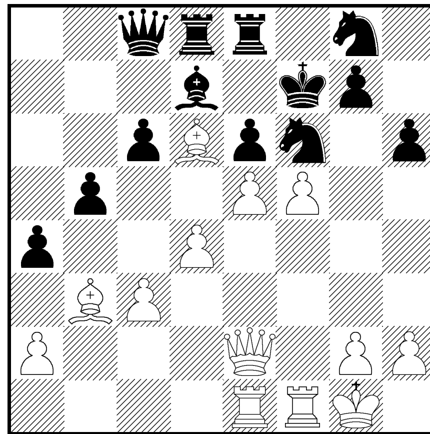
Planning is perhaps the most advanced use of knowledge in chess possible. It requires a complex analysis and understanding of the game not just at a single point but spanning many moves. When to adopt, continue with, and abandon a plan are all decisions that require knowledge.

4.2.1 Planner

In Schaeffer (1983), a chess program called *Planner* was discussed. This program was based off an existing chess engine, and modified to use planning. In this implementation, there were 13 different plans available to choose from: attack king-side, attack queen-side, attack center, defend king-side, defend queen-side, and defend center were general plans. Attack pawn weakness, occupy open file, pawn break, advance passed pawn, defend pawn weakness, and block passed pawn were specific plans. The final plan was wait, which was invoked when another plan could not be decided upon.

The plans were selected by a complex analysis routine, which examines the position of the board using a lot of chess knowledge. This differs from standard evaluation functions, which are written to be quick so that they can be used in massive tree searches. The plan is chosen and activated, and then information about the plan is stored, so that the plan may be ended if it goes outside certain conditions.

When a plan is active in this engine, it serves to bias the choices of the standard minimax



1.♕h5+ ♘xh5 2.fxe6+ ♔g6 3.♗c2+ ♔g5 4.♖f5+ ♔g6 5.♖f6+ ♔g5 6.♖g6+ ♔h4 7.♖e4+ ♘f4 8.♖xf4+ ♔h5 9.g3 ♖f8 10.♖h4++

Figure 4.2: PARADISE finds mate in 19 ply

tree towards moves that are consistent with the plan. The plan is broken down into sub-plans, and these sub-plans into moves. If a plan starts to go wrong, or is not making progress, then it is replaced with the wait plan until another plan seems appropriate.

This chess engine was made to play against an unmodified version of the original engine it was built on. The version with planning won 8 games, while the original version won only 2.

4.2.2 PARADISE

Another chess program that uses planning is PARADISE (Wilkins 1981). PARADISE is more sophisticated than *Planner*, and uses a large collection of rules to suggest plans. Instead of being based on a minimax tree and using the plans to guide the search, PARADISE uses production rules that suggests moves which get stored in a database. The moves in the database are then analyzed to find the best one. The program is capable of finding very deep combinations because no artificial limit is placed on its search depth. It searches for moves as long as a plan is continuing to work.

The rules for PARADISE were obtained by solving chess puzzles from a puzzle book and writing down rules that could be used to solve the puzzles in a similar manner. The chess engine interprets these rules, which match various patterns on the chess board. If a pattern matches, then it posts its goals and moves into the database for further consideration. The actual tree that the system builds is very small, only a few hundred nodes, but it is incredibly deep and does not suffer from lack of breadth because the direction was dictated by the rules. PARADISE follows a plan until it fails or a significantly better one is found.

4.3 Acquiring Knowledge

In order to be able to use knowledge, it must first be acquired. Simon & Schaeffer (1992) state that "there is good evidence that it takes a minimum of ten years of intense application to reach a strong grand master level in chess," and that "presumably, a large part of this decade of training is required to accumulate and index the 50,000 chunks." Perhaps this is a reason why there has not been extensive use of knowledge in chess programs–it would simply take too long to provide the program with enough knowledge to make it play a strong game in all cases.

4.3.1 Human Input

Most programs that use knowledge to some degree of sophistication have the knowledge input by human experts. This works fine for small amounts of data, but it is not really possible to have a grand master sit down and tell a computer everything he knows about chess. Another *knowledge acquisition bottleneck* (George & Schaeffer 1990) is the fact that often the programs are written by scientists and not by chess players. The interface for adding knowledge, designed by the scientist, may be difficult for the chess expert to use. George & Schaeffer (1990) proposed a graphical interface for MACH as a solution to this, but also say that "if MACH were 'intelligent' enough to define and re-define its own set of chunks, this bottleneck would greatly diminish."

Despite the large amount of time and difficulty required for a human to provide knowledge to a program, there is one obvious advantage–it is possible to describe high-level concepts, such as plans, to the program. With human input it is easy to define goals and steps required to achieve them; with automated systems it is hard to acquire anything beyond very basic knowledge.

4.3.2 Automatic Acquisition

There have been a few attempts to create programs that are capable of learning concepts and rules of the game, mostly using explanation-based learning and case based reasoning. However, these systems are only capable of learning simple combinations that are present in the move tree and are not capable of understand more interesting positions and ideas (Fürnkranz 1996).

More work has gone into learning chess knowledge by modelling the way humans acquire, store, and use knowledge. The chunking theory is the basis for most of these models, which attempt to classify chunks on the board and associate information about the game with these chunks using a model of human cognition. The models are based on EPAM (Elementary Perceiver And Memorizer) (Feigenbaum & Simon 1961, Feigenbaum & Simon 1962), which is a system designed to simulate human verbal learning. EPAM has mechanisms to simulates short- and long-term memory. It uses a discrimination network to store data, where each non-leaf node is a test and each leaf node represents something that has been stored in the network. The long-term memory is simulated by the discrimination net; when learning,

the input is sorted through the tests in the network and stored as a leaf node.

One of these models is CHREST (Chunk Hierarchy and REtrieval STructures) (Gobet & Lane 2001). CHREST adds a few features to EPAM which let it model complex data more easily. It was applied to chess by using virtual eyes that mimicked human eyes scanning a chess board. The chunks the eyes observed were added to the discrimination net. There is a program, CHUMP, that is based on CHREST and plays chess by using pattern matching only, without searching.

Hyötyniemi (1997) proposed an alternative method of finding chunks in positions that does not try to imitate the way humans learn. This method uses the generalized generalized hebbian algorithm to extract features from board positions and group them.

Chapter 5

Clustering

In order to associate knowledge with different groups of pieces, it is first necessary to define what those groups of pieces are. This can be achieved by using clustering, a form of unsupervised learning, to extract information about the positions of pieces in a set of games. The generalized generalized hebbian algorithm (Hyötyniemi 1996) is well suited for this task.

5.1 Principal Component Analysis

Principal Component Analysis (PCA) is a standard statistical method that can be used for analysis and dimension reduction. PCA takes a set of data and returns a new set of axes, called principal components. The first axis is aligned in the direction of the highest variance in the data. The second is aligned, orthogonal to the first, in the direction of the second highest amount of variance, and so on. Examining the data along the first axis shows the most amount of information in the data possible on one axis. If the first two principal components are used then the amount of information shown from the data is the most possible for two axes. The principal components are ordered; the first is capable of showing the most information, the last the least. This is useful for visualizing high-dimensional data — displaying a two- or three-dimensional graph using the first principal components maximizes the amount of information that can be seen, making it easier to analyze. It is also good for dimension reduction because the greatest amount of information possible is preserved in the lower-dimensions space.

The principle components are calculated by finding the eigenvectors of the covariance matrix of the data. The eigenvector with the greatest eigenvalue is the first principal component, and the eigenvector with the lowest eigenvalue is the last principal component.

There is another way to calculate the principal components which does not explicitly calculate the covariant matrix. It uses a neural network and is called the Generalized Hebbian Algorithm.

5.2 Generalized Hebbian Algorithm

The Generalized Hebbian Algorithm (GHA) is named after a theory of learning behavior proposed by Donald Hebb. His theory described a manner in which the relationship between cells in the brain becomes stronger over time. The more often the firing of one cell contributes to the firing of another, the stronger the chance the second cell will fire after the first becomes. The term Hebbian can be used to describe most forms of unsupervised learning or clustering, where the weights of the network are adapted to fit the input data.

Sanger (1989) described a method for using a single linear neuron for extracting the first principle component of input data.

$$w(t+1) = w(t) + \eta(t)\left[y(t)x(t) - y^2(t)w(t)\right]. \tag{5.1}$$

In that equation, w is the vector containing the weights, x is the input vector, y is the output vector and η is the learning weight. The weight vector is being updated every epoch to better model the input data. The weights in Hebbian learning can grow without bounds without normalization, so the term $y^2(t)w(t)$ is needed to normalize w and keep $|w|$ around 1. When the network is trained, the weight vector is adapted and becomes equal to the first principle component, the eigenvector with the largest eigenvalue from the covariance matrix, without calculating the covariance matrix at all.

This can be generalized to extract all of the principal components, and was done by Oja (1982), resulting in the GHA. This time a simple, one-layer neural network is used, with one neuron for each principle component to be found. In order to find the eigenvectors in decreasing order, the input vector should be modified after every individual neuron is trained to remove the contribution of that neuron.

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)\left[y_i(t)x_j(t) - y_i(t)\sum_{k\leq i} w_{kj}(t)y_k(t)\right]. \tag{5.2}$$

The summation term is responsible for removing the effects of previous neurons from the input vector. In terms of PCA, this means that the first weight vector becomes the axis aligned with the largest amount of variance, which is then removed from the input data, allowing the second weight vector to align with the second largest amount of variance. This continues for each neuron in the network. When the network has converged, the weights are the eigenvectors of the covariance matrix of the input data.

5.3 Generalized Generalized Hebbian Algorithm

The Generalized Generalized Hebbian Algorithm (GGHA) was presented by Hyötyniemi (1996) as a combination of clustering and principal component analysis. PCA assumes that all of the data in the input set belongs to one distribution, but GGHA assumes that there are multiple independent distributions in the data set. This is desirable for finding chunks because chunks are groups of pieces that are independent of each other on the board. If PCA was used to analyze a game, it would find features that represent the whole board.

Knowledge could be associated with these features, but could only be used when the entire state of the board matches. With GGHA, knowledge can be associated with each of the separate distributions, which represent the chunks, allowing the presence of a group of pieces instead of the state of the whole board to suggest moves.

GGHA is based on the Kohonen self-organizing map (SOM) (Hyötyniemi 1996), an unsupervised learning method that is often used for feature analysis. The SOM consists of N nodes, represented by a set of codebook vectors, θ_i where $1 \leq i \leq N$. The SOM is different from other networks because it has the idea of a neighborhood, where the locations of the nodes are important. When the network is adapted, the node that best fits the input data, the 'winning' node, is moved towards the input values. The other nodes are also moved, but how much they are moved depends on their proximity to the winning node — the closer the node to the winner, the more it is moved. If f is the vector of input data and c is the index of the node that best matches f, the adaptation function is:

$$\theta_i \leftarrow \theta_i + \gamma \cdot h(i, c) \cdot (f - \theta_i). \tag{5.3}$$

The neighborhood parameter, h, determines how close the ith node is to the winning c^{th} node and modifies the amount that the node is moved by. The parameter γ ensures that the network converges.

The SOM itself is not capable of feature extraction, only of organizing the data that it is given. GGHA is a modification to the standard SOM that performs feature extraction as the network is trained. When the network converges, each node of the network (θ_i) is a basis vector for the feature space. The GGHA assumes that each input vector f can be expressed additively as a sum of feature vectors, f_i, multiplied by weights ϕ_i:

$$f \approx \sum_{i=1}^{N} \phi_i \cdot f_i \tag{5.4}$$

In order for the nodes to converge into basis vectors for the feature space, feature vectors must be input to the SOM. A feature vector f_i can be extracted from the input vector f with:

$$f_i = (\theta_i^T f) \cdot \theta_i = \phi_i \theta_i, \tag{5.5}$$

and then the input vector should be updated:

$$f \leftarrow f - f_i = f - \phi_i \theta_i. \tag{5.6}$$

This process should be done starting with the θ_i that best matches the input vector (so the weight ϕ_i is maximum) and proceeding in descending order to the worst match.

The problem with this is that in order to extract the feature vectors from the input vector the basis vectors must already be defined, but the basis vectors are the codebook vectors of the SOM, which are created by training on the feature vectors from the input. In order to overcome this circular dependency it is necessary to initialize the SOM to a random state and then *assume* that the basis vectors have already been defined. If the data is applied iteratively, the basis vectors will eventually converge on the correct values. Figure 5.1 shows the error present in the SOM over time. It starts out very high and drops as the network converges.

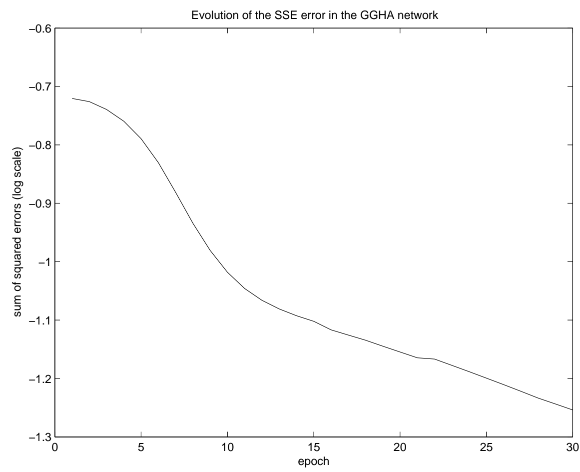


Figure 5.1: The error in the network decreases with each epoch.

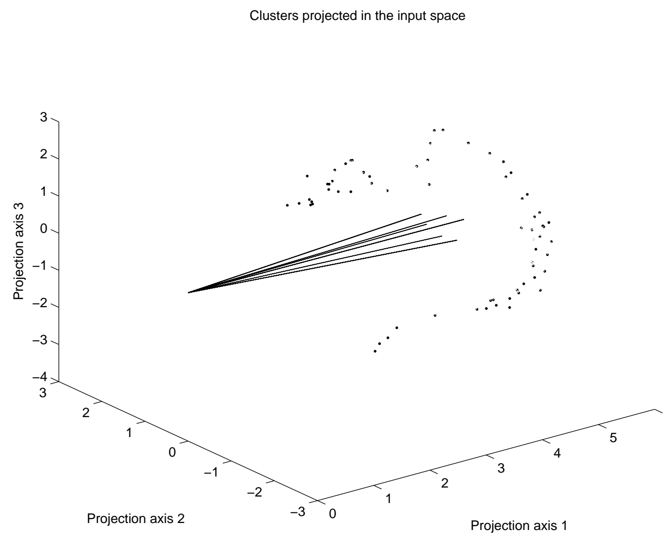


Figure 5.2: 3D projection after 5 epochs.

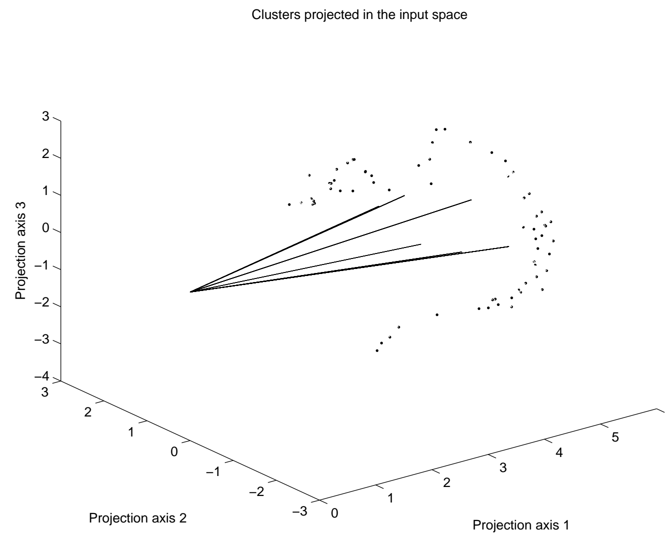


Figure 5.3: 3D projection after 15 epochs.

Figure 5.2 shows a series of positions from several games projected into 3D space. The axes for the projection were selected using PCA. The points in the figure represent the input vectors (the positions on the board), which were orignally 768-dimensional. The lines are from the origin to the center of the clusters. It is interesting to see how the different games diverge from one another; the points at the top of the figure are positions from the beginning of the game, and as the game progresses, the points move clockwise and spread out.

In Figure 5.2, which shows the network after 5 epochs, the centers of the clusters are quite close together, but in Figure 5.3, which shows the network after 15 epochs that have moved and are now spread out more. This reflects the fact that the network is learning the inputs better every epoch, and will eventually converge.

Figure 5.4 shows the same network at 15 epochs, but this time also shows the feature vectors for each point mapped into 3D space. The length of each line represents the weight ϕ of that particular feature. In this example, 4 features f_i were assumed to contribute to each input vector f. Although it is hard to see in the diagram, there are 4 feature vectors leading to each point. In some cases, the feature vectors only approach the input vector; this is expected because the network has not converged yet and there is still a large amount of error.

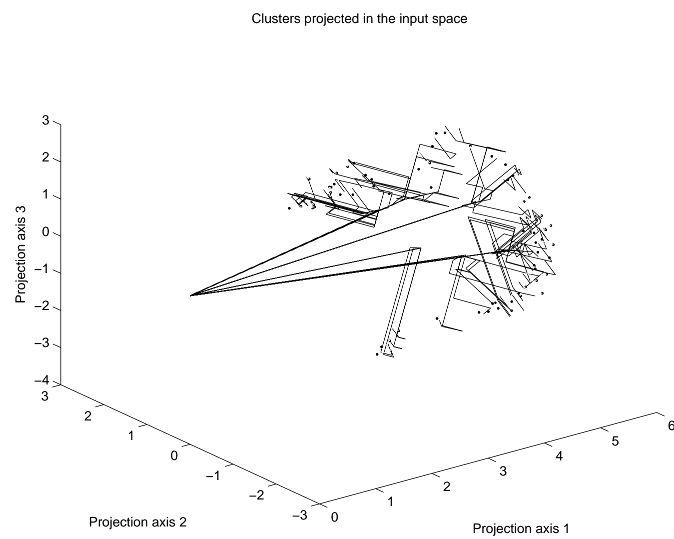


Figure 5.4: 3D projection after 15 epochs, showing feature vectors.

Chapter 6

Implementation

6.1 GNUChess

GNU Chess is a free, open-source chess engine capable of master-level play, using many modern ideas about computer chess. It uses bitboards to represent the board internally and generates moves using the rotated bitboard technique, a relatively new method of quickly generating moves by 'rotating' the order board squares are stored in the bitboard. For search it uses Principle Variation Search (Marsland 1983), which is a modification to alpha-beta search that can increase performance by using tighter alpha and beta bounds in the search. The program also uses a hash table to store moves that have been evaluated previously, increasing the search speed.

6.2 What Was Implemented

The main addition to GNU Chess this project made is, naturally, the GGHA clustering algorithm that enables both the chunking of existing master games and also the analysis of the game in progress. There are two different modes that can be used: the first is an altered evaluation function that increases a move's score if it is suggested by a chunk and the second mode automatically plays a learned move if a familiar position is recognized.

6.2.1 The GGHA Algorithm

The GGHA algorithm was implemented in C and integrated into GNU Chess. A command was added to the interface that reads in chess games from a file and then runs the clustering algorithm on them. The file is a PGN file, a standard format for recording chess games, and in which thousands of master games are available openly on the Internet.

For each position read in from the PGN file the move that was made from that position is recorded in a table, if it was played by the winning side. This helps make sure that the moves collected are not outright blunders. Since multiple games may be read in at once from

a PGN file a hash is calculated for each position and when the same position is encountered multiple times each of the different moves are added to the list of moves made from that position. The table also stores the number of times a move was made from a position. This information is used later as another measure for determining the quality of the move. Once this process is complete, the table is saved to a file.

As input, the GGHA algorithm takes the sample data as a matrix of real numbers (in this implementation, C's double type) where each column of the matrix is one sample, or position from a game. For chess, each sample has 768 dimensions, one value for whether each type of piece for each color is on each square of the board. If a piece is present the corresponding value in the input is one; if it is not present the value is zero. The input matrix thus has 768 rows and as many columns as unique positions that were read in from the PGN file.

After it has finished, the algorithm returns two matrices, one containing the centers of the clusters that were found and the other containing the feature model that was adapted, which are then saved to files on disk. The next step is to use these matrices and the sample data to construct a table that links specific chunks to positions and moves.

There is an analysis function, GGHAanal, which takes a data sample and the two matrices returned from clustering and returns in a vector the coordinates of that data sample in the feature space. The feature space has fewer dimensions than the original input data, the actual number depending on the number of presumed clusters. It is also a sparse representation, meaning that most of the components of the vector are zero. Each component of vector represents a chunk. If a component is non-zero then it means that specific chunk is present in the data sample.

The chunk-to-move table is the same length as the vector returned from the analysis function (because one entry is required for each chunk) and contains for each chunk a list of indices to the original table holding positions and moves. The table is constructed by running all of the original sample data through the analysis function and then, for each chunk present in the returned vector, adding the index of the data sample to the list of indices for that chunk. When it is necessary to look up which moves were played when a specific chunk was present, the moves can be found by looking at the list of positions where that chunk was present and then at the moves played for each of those positions.

Once this is complete, the table is saved.

6.2.2 Evaluation

After the initial run of the algorithm, the tables and matrices generated can be used in either of two different modifications to GNU Chess. The first of these is a modification to the program's evaluation function. The evaluation function looks at a position and gives it a score based on heuristics such as pawn position, material balance, and control of the board. The modification adds a small bonus to the score if the move to arrive at that position was one suggested by a chunk.

When a position is being evaluated the previous position is chunked and a list of moves is generated from the various tables. If the move that was used to arrive at the current position

is in the list of moves suggested by chunking then a bonus score of one third of a pawn is awarded to the current position. The bonus value was derived from experimentation–too large of a value and the program is likely to blunder if chunking suggest a bad move and too small of a value means that chunking is essentially insignificant.

6.2.3 ChunkQuery

The other modification is the ChunkQuery function, which acts similarly to the BookQuery function in GNU Chess. BookQuery uses a library of moves to automatically play based on position during the opening moves of a game. The opening book feature is quite common in modern chess engines as it allows the engine to make moves quickly at the beginning of a game, saving time for deeper thought about later moves in tournament conditions. ChunkQuery performs a similar function, but instead of playing moves based on exact position matches and only at the beginning of the game, ChunkQuery uses chunking to generate a list of moves and the best one of these is automatically played.

The best move is selected by several criteria, the most important being the number of chunks that suggested that move. If, for example, there are four chunks found in the current position, and all four of those chunks suggest the same move, this implies that the move is a better one for the current position than a move suggested by fewer chunks because the position more closely matches the position the move was originally made from. The next criteria is the number of times a move was made. This is only used when there are two or more moves suggested by the same number of chunks. The move that was used the most number of times in the sample games is the move that gets played.

The ChunkQuery function automatically plays the move, and no searching or other evaluation takes place. If the function fails to find any moves, then it falls back to a standard search.

6.3 Problems

There are a few problems with the implementation in its current state. Foremost among these is the fact that the GGHA algorithm is very slow for large sets of data. The algorithm involves a lot of matrix multiplication, which is implemented here as an $O(n^3)$ operation. The time is not that significant for small matrices, but when repeatedly multiplying matrices with hundreds of thousands of elements it can take an incredibly long time to run. Clustering around 500 moves took more than 24 hours to complete. A possible solution to this is to use the Strassen method (or something similar) for matrix multiplication, which reduces the number of operations required.

Another problem, related to the first, is that when searching the tree, the program evaluates thousands of different positions. Running the analysis algorithm on each of the positions takes a long time. This means that it would really not be possible to run the program in a tournament under a time limit without finding a way to speed up the evaluation. Limiting the depth at which the modified evaluation takes place would substantially reduce the number

of times the function has to be run, because the number of nodes evaluated increases rapidly at each subsequent level of search.

The final problem is with the ChunkQuery function. Because the function plays a move immediately, without evaluating anything else, it is rather prone to error and serious blunder. If a position is similar enough to one that was learned then a chunk might match, even though the small difference could change the entire meaning of the position.

Chapter 7

Experiments and Results

7.1 Testing the GGHA Implementation

In order to be sure that the GGHA algorithm was implemented correctly it was necessary to test it. This was done by creating a large matrix of the form

$$\begin{pmatrix} 1 & 2 & 3 & ... & 200 \\ 201 & 202 & 203 & ... & 400 \\ ... & ... & ... & ... & ... \\ 19801 & 19802 & 19803 & ... & 20000 \end{pmatrix}$$

which is then chunked and reconstructed from the chunks. The reconstructed matrix was identical to the original, showing that the clustering worked properly. This test can be accessed inside the program with the command: chunk test.

7.2 Reconstruction of Positions

de Groot (1965) and Chase & Simon (1973) have established that the ability to reconstruct a position on a board is a good measure of the player's skill and library of knowledge. Examining the ability of the program to reconstruct positions based on chunks provides a rubric by which it is possible to gauge the success of the clustering method in defining and identifying chunks.

The output from the reconstruction is, like the input, a 768-dimensional vector representing the presence of each type of piece on each square of the board. Also like the input, the values are real numbers and as such can range from below zero to above one, which represent the absence and presence of a piece on a square, respectively. In order to decide whether or not a piece is present, thresholding is applied to the vector (Hyötyniemi 1997). Values above the threshold are treated as one, values below are treated as zero.

Figure 7.1 shows the board at the start of a game and Figure 7.2 shows the board as reconstructed using 4 chunks out of 100 possible, learned from the 1937 World Championship

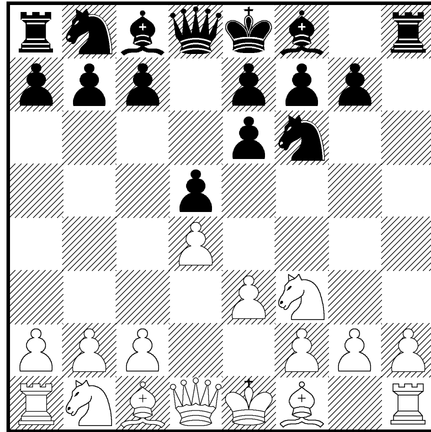


Figure 7.1: Position in a game.

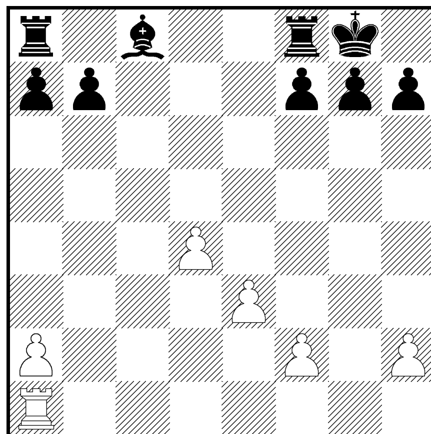


Figure 7.2: Reconstruction of Figure 7.1 using 4 of 100 chunks and a threshold value of 0.6.

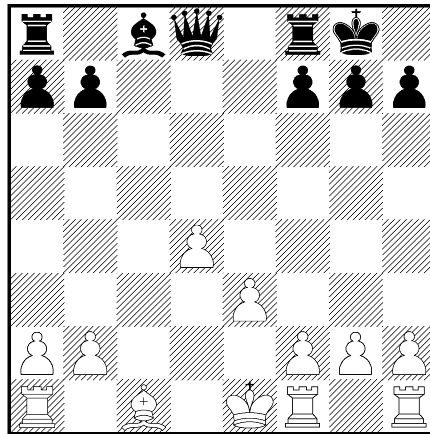


Figure 7.3: Reconstruction of Figure 7.1 using 4 of 100 chunks and a threshold value of 0.5.

games between Alekhine and Euwe. Most of the pieces are missing in the reconstruction, indicating that the threshold value of 0.6 is too high. In Figure 7.3 the threshold value has been changed to 0.5 and more pieces are visible, but the reconstruction is still incorrect. Pieces are missing or in the wrong location, the black rook and king are in a castled position, and there is an extra white rook. This occurs because there is no combination of chunks capable of exactly representing all of the pieces on the board. Changing the threshold value only changes which pieces are displayed, not which chunks are used in the reconstruction.

7.3 ChunkQuery

The position in Figure 7.1 was reached by playing against the program in ChunkQuery mode, where chunks are found in each position and one of the moves suggested by chunking is automatically played. In order to reach that position, the move e6 was played. This was one of 7 moves suggested by chunking, which are listed in Table 7.1.

Move	Number of Chunks	Number of Uses
Nc6	2	1
e6	3	7
Bf5	2	38
c6	1	4
c5	2	1
Nbd7	1	1
e5	1	1

Table 7.1: Moves suggest by chunking before Figure 7.1.

The move e6 was chosen because it was suggested by more chunks than any other move. If there had been two moves suggested by an equal number of chunks then the move with the

most uses would have been selected. In this case, the move selected was a reasonable one for the position, but quite often the move selected is completely inappropriate, giving away pieces and position for no benefit. There are a couple reasons why this can occur. First, all of the moves selected might be used in a very low number of chunks, implying that only a few similarities between the current game and the games the moves were learned from, and any move used on this advice is very likely to be incorrect. The presence of a chunk does not automatically mean that a move it suggests is the best one, or even a good one, which is why the move suggested by the most chunks is used. To alleviate this problem, a lower limit can be imposed on the number of chunks required for a move to be used. If a move is suggested by fewer chunks than the limit, then it is considered bad and not used. When this rule is applied, the quality of the game played by the program improves drastically. As a game progresses, the high branching factor means similarities between the actual position and learned positions become increasingly less likely. Imposing a required number of chunks on a move ensures that the move, if played, is at least somewhat relevant.

A second reason that incorrect moves are played is because the chunks are not precise enough to represent the exact positions of all the pieces, meaning that two similar situations will have the same chunks present and therefore suggest the same moves. The problem with this arises because consecutive positions in the master games used for training are represented by the same chunks, meaning that when they are recalled by ChunkQuery, both of the moves from those positions are suggested, and the move the occurred second in the master games may be made first. When move ordering is important, this can have disastrous effects–a move that relies on another piece being in position could be made before the other piece has moved. Increasing the number of chunks and the number of epochs in training would hopefully help define the chunks more precisely and reduce the problem, but this was not the case. Using more data when defining chunks does help a little, but only if the new data is similar; if it is not, then the chunks must represent a greater variety of positions, leading back to imprecision.

7.4 Evaluation

In order to test the modified evaluation function, the modified program was made to play an unmodified version of GNU Chess to see if there was any improvement in its play. Because the modified program runs considerably slower, it was necessary to control a lot of variables in order to make the games fair. There was no time limit set for the games, so the programs were allowed to think for as long as it took to select a move. Thinking during the opponent's time was also disabled, as this would allow the unmodified version a much larger amount of time. The depth of search was set to a constant level in both of the programs, ensuring that the only difference is the use of chunking in the modified program.

Figure 7.4 contains the results of this experiment. The modified program was set to play against the unmodified version at search depths of 1, 2, and 3 with 10 games at each depth. As a control, the unmodified version played itself 10 times at each depth, too. One point was awarded for a win, zero points for a draw, and one point was taken away for a loss. The score reported for the control is the score white got in those games.

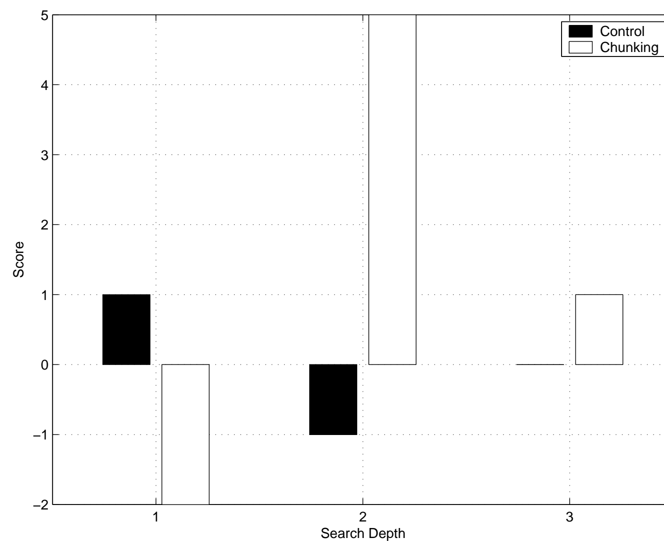


Figure 7.4: Scores

In the games with a search depth of 1, the modified program loses by 2 points to the unmodified version. This, however, is not as bad as it seems. With a search depth of only 1, both programs are just making the move that has the highest score and not even considering what the opponent's reply might be. Like this, the games played are only slightly better than those played by ChunkQuery.

With a search depth of 2, the modified version does significantly better than the unmodified version, winning by 5 points. In this case, the knowledge provided by chunking gives the modified program quite an advantage. However, this advantage does not last when the depth is increased to 3. In this case the modified program only wins by 1 point, which is not really significant considering that the control score was in the other cases 1 and -1.

It is possible that the network was not trained well enough or the positions in the games at depth 3 were not recognized by the chunking, but it is more likely that the score dropped because the limit of the knowledge being used was reached. The only knowledge used was the list of moves that were made from similar positions — perhaps this is not enough to be of any use when the search reaches a certain level.

Chapter 8

Conclusions

Even though it worked well only in certain caises, the program has shown that it has the beginnings of an ability to acquire and use knowledge. It analyzed master games, broke them down into chunks with a clustering algorithm, associated knowledge about moves with each chunk, and then used this knowledge when playing chess. These are the building blocks with which it might be possible to create a more advanced program that uses knowledge in a much more sophisticated manner.

The level of knowledge associated with chunks here is quite low — just the next move made from that position. The next step to take in this area is to associate greater amounts of knowledge. One possible idea would be to associate part of the game tree with each chunk, meaning that when the program is considering the moves suggested it can see what happened previously and after that move in the master game. This would allow it to better select a move that fits the current game and to choose the move with the best outcome. The more knowledge added, the less search needs to be performed. If part of the game tree is included then there is no need to search ahead — a move can be selected based on the game tree from the master games. Interestingly, providing the game tree would also help eliminate the problem with chunks matching too easily and moves being played out of order. If the next few moves are available from the game tree they can always be used in the correct sequence.

Once chunks serve as indices into game trees then it is possible to start adding even more advanced applications, such as planning, to the program. If the presence of a particular chunk indicates that a large material advantage is about to be won, say by capture of the opposing queen, then this may be set as the goal and moves may be chosen from the game trees that lead to this goal. In order for this to happen, knowledge about the state of the positions in the game tree, such as positional value and material value, must be added.

Of course, in order to generate a strong program, tens of thousands of chunks — as many as a grand master — must be defined. In humans it can take a decade to acquire all of this knowledge, and it is quite possible that it will take a similar time for a computer.

Bibliography

Berliner, H., Kopec, D. & Northam, E. (1990), A taxonomy of concepts for evaluating chess strength, *in* 'Proceedings of the 1990 ACM/IEEE conference on Supercomputing', IEEE Computer Society, pp. 336–343.

Chase, W. & Simon, H. (1973), 'Perception in chess', *Cognitive Psychology* **4**.

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

Feigenbaum, E. A. & Simon, H. A. (1962), 'Simulation of human verbal learning behavior', *Commun. ACM* **5**(4), 223.

Feigenbaum & Simon (1961), Forgetting in an association memory, *in* 'Proceedings of the 1961 16th ACM national meeting', ACM Press, pp. 23.201–23.204.

Frey, P. W. (1977), *An introduction to comptuter chess*, Vol. Chess Skill in Man and Machine of *Texts and Monographs in Computer Science*, Springer-Verlag, chapter 3.

Fürnkranz, J. (1996), 'Machine learning in computer chess: The next generation', *International Computer Chess Association Journal* **19**(3), 147–161.
*citeseer.ist.psu.edu/furnkranz96machine.html

George, M. & Schaeffer, J. (1990), 'Chunking for experience'.
*citeseer.ist.psu.edu/george90chunking.html

Gobet, F. & Lane, P. (2001), 'Chunking mechanisms in human learning', *Trends in Cognitive Sciences* **5**(6), 236–243.

Hayes, J. E. & Levy, D. N. (1976), *The World Computer Chess Championship*, Edinburgh University Press.

Hyötyniemi, H. (1996), Constructing non-orthogonal feature bases, *in* 'Proceedings of the International Conference on Neural Networks (ICNN'96)', pp. 1759–1764.

Hyötyniemi, H. (1997), On the statistical nature of complex data, *in* 'SCAI'97 — Sixth Scandinavian Conference on Artificial Intelligence: Research Announcements', pp. 13–27.

Hyötyniemi, H. & Saariluoma, P. (1998), Simulating chess players' recall: How many chunks and what kind can they be?, *in* 'Proceedings of the Second European Conference on Cognitive Modelling', Nottingham University Press, pp. 195–196.

Marsland, T. A. (1983), Relative efficiency of alpha-beta implementations, *in* 'Proc. of the 8th IJCAI', Karlsruhe, Germany, pp. 763–766.

Marsland, T. A. (1987), 'Computer chess and search', *ENCYCLOPEDIA OF ARTIFICIAL INTELLIGENCE* .
*citeseer.nj.nec.com/marsland89computer.html

Marsland, T. A. (1991), Computer chess and search, Technical Report TR 91-10.
*citeseer.nj.nec.com/marsland91computer.html

Oja, E. (1982), 'A simplified neuron model as a principle component analyzer', *Journal of Mathematics and Biology* **15**, 267–273.

Sanger, T. (1989), 'Optimal unsupervised learning in a single-layer linear feedforward neural networks', *Neural Networks* **2**(6), 459–473.

Schaeffer, J. (1983), Long-range planning in computer chess, *in* 'Proceedings of the 1983 annual conference on Computers : Extending the human resource', pp. 170–179.

Shannon, C. (1950), 'Programming a computer for playing chess.', *Philosophical Magazine* **Ser. 7, Vol 41**(314).

Simon, H. & Schaeffer, J. (1992), 'The game of chess'.
*citeseer.nj.nec.com/simon92game.html

Wilkins, D. (1981), 'Using patterns and plans in chess', *Readings in Artificial Intelligence* .