Full Chess Retrieval

Peter McKenzie

October 1994

Abstract

Most chess programs utilise simple memory-based techniques that compliment the brute force search. These techniques utilise stored positions to increase program performance. The ability to efficiently store and retrieve positions is critical to the success of the memory-based approach.

The use of data compression to increase the capacity of memory-based systems is investigated. A number of methods for compressing and indexing chess positions are presented and analysed.

Contents

Chapter 1

Introduction

Game playing has been a major focus of AI research since the days of Shannon (Shannon, 1950) and Turing (Turing *et al.*, 1953). Games provide well defined problems that have traditionally required intelligence to solve. They are, however, typically of lower complexity than 'real world' problems. They therefore provide an excellent testbed for the development of AI techniques.

Of the various games, chess has proven to be a particularly suitable problem domain. There are a number of reasons for this. It has widespread popularity, so there is no shortage of expertise on the subject. There is also a great deal of literature on chess, that has been built up over a period of more than 200 years.

Furthermore, chess is of sufficient richness for there to be a wide range of proficiency levels. The level of proficiency can be easily measured owing to the existence of an accurate rating system. This gives researchers a simple and objective way of measuring the success of their ideas.

The challenge of designing a program that plays at a level comparable to that of the human world champion has been an overriding goal for researchers in the field. While early programs played a passable amateur game, it is only now, after more than 40 years of research, that the best programs are getting close to playing at world championship level. However the $100,000 Fredkin prize, for the first program to attain world champion status, still remains unclaimed.

By far the most successful approach to computer chess has been the so called 'brute force' approach. Programs based on this approach will examine thousands, or even millions of positions per second. In contrast, experiments have shown that human players typically consider at most a few hundred positions in the course of a three minute think (de Groot, 1965). The crucial factor being that the skilled human player is able to rule out many irrelevant possibilities. Attempts to program such selectivity have so far yielded poor results.

The brute force approach is of little interest to those AI researchers who are interested in emulating human style thinking. To others however, it is an excellent example of solving a complex problem that has traditionally been associated with intelligence.

Brute force based programs focus on utilising the great computational speed of the computer. These programs often incorporate techniques designed to

increase the effectiveness of the search by making full use of the, typically large, memory capacity of the computer. I will refer to such techniques as *memory based techniques.*

There are a number of memory based techniques in common use, all of which require the storage of information about individual chess positions. In general, the utility of these techniques increases as information about more positions is stored. In fact, their effectiveness is ultimately limited by the total number of positions that can be stored. The major focus of this project therefore, is increasing the number of chess positions that can be stored in a given space.

The use of data compression techniques to increase the capacity of memory based techniques, particularly rote learning, is investigated. The main problem here is the compression of chess positions. A number of methods for compressing collections of chess positions are implemented and tested.

An additional problem is that the collection of chess positions must support fast random access. An indexing technique is developed to support this. This puts constraints on the types of compression techniques that can be used. In particular, adaptive compression becomes less applicable.

Memory based techniques are examined in more detail in Chapter 2. Following this, the aims and motivations of the project are developed and stated in Chapter 3. Related work is covered in Chapter 4.

Discussion of my own work begins in Chapter 5 where a basic indexing system is described. Following this a number of compression methods are described in Chapter 6. The results from experiments involving these methods are described in Chapter 7. Conclusions are drawn in Chapter 8. A special mention should be made of appendix A which should be read first by those unfamiliar with algebraic chess notation.

Chapter 2

Memory Based Techniques

An overview of the basic concepts is presented in section 2.1. Following this, the various memory based techniques are examined in sections 2.2 to 2.5.

2.1 Overview

Most game playing programs work by searching a game tree using a derivative of the minimax algorithm, usually augmented with alpha-beta pruning. Terminal nodes in the tree are given a score by a heuristic based evaluation function. Complimentary to this approach are memory based techniques, which are is used in various forms by many successful game playing programs.

In their simplest form, memory based techniques work as follows: the program will have a number of stored positions together with information about each position. That information will typically contain a score and best move for the position. During the tree search, the program checks at each node to see whether the current position is the same as any of the stored positions. If it is, then the score for that node is returned immediately, without any further searching. Otherwise the search continues normally.

2.1.1 Benefits

There are a number of benefits from the memory based approach as described above. Firstly, the scores for the stored positions may be of a level of accuracy that the program could not achieve under the time constraints of tournament conditions. This is particularly the case for opening books (see section 2.4), where the scores for each position are often based on years of experience by human grandmasters.

Another benefit is the saving of time. The simplest case is where the current game position has been previously memorised. If the correct move for that position was also memorised, then that move could be played almost instantly (assuming fast retrieval). This time saving allows the program to spend more time on later moves in the game, thereby raising the performance level.

Time savings are also possible while searching the game tree. Any node that can be evaluated using a memory based technique will not need to be searched

to a greater depth , resulting in a time saving. This saved time can be used to search other parts of the tree to a greater depth.

2.1.2 Different Techniques

The different techniques are typically most useful in different phases of the game. Their implementations will tend to differ in the key areas of the methods used for storage and retrieval, and the methods used for generating the stored information. Apart from rote learning, all of these techniques are practically mandatory for a tournament strength chess program.

2.2 Transposition Tables

The transposition table (sometimes called the trans table) is maintained over the course of a single game. Whenever the program calculates a score for a position in the search tree, this score is stored in the transposition table. Usually some additional information is also stored, including the best move, and the depth that the position was searched to.

Simply put, the transposition table is the program's short term memory. It gets its name because it helps the program deal with transpositions. For example, from the starting position in a game of chess, the moves **1.e4 e6 2.d4** and **1.d4 e6 2.e4** (see appendix A for an explanation of this notation) result in the same position.. On reaching the position for a second time, a program would see the position in the transposition table and return the value stored there.

The transposition table is also useful for remembering search information between moves. For example, consider a chess program that chooses a move after performing a six ply full width search. After the opponents reply, the program must again choose a move. The current position will be in the transposition table, as it occurred while searching the previous move. In this case the program will not use the score in the transposition table, as it is effectively from a four ply search while the program wants to do a six ply search. In this case the transposition table will be used for *move ordering.* The program still performs a six ply search, but will examine the move from the transposition table first. As this move is effectively the result of a four ply search on the current position, it is likely to be a good move. The effect of examining good moves first can result in dramatic time savings as a result of using the alpha-beta algorithm.

Because a transposition table is accessed and updated thousands of times per second, it is essential that it is implemented using the fastest possible data structure. Therefore, a hash table (in primary memory) is always used. The Zobrist hashing function (Zobrist, 1970) has been specially designed for chess positions. Typically, a hash code of between 32 and 48 bits is used.

2.3 Rote Learning

Rote learning is perhaps the simplest form of machine learning. It was first studied by Arthur Samuel (Samuel, 1959) who conducted a number of classic experiments with a program that played checkers. Despite the fact that Samuel's program showed impressive learning capabilities, his ideas were not investigated by computer chess researchers until relatively recently.

Rote learning is, in principle, quite similar to the transposition table. The key difference is that positions are memorised on a permanent basis, so that information learnt about a position may be utilised in a different game. If the transposition table is the programs short term memory, then the rote learning system can be seen as its long term memory. Recent work on applying rote learning to chess has focussed on adding a degree of persistency to the transposition table.

David Slate, a dominant figure in computer chess during the 1970s, conducted a number of experiments with a chess program that incorporated rote learning (Slate, 1987). These experiments were extended by Tony Scherzer, with his Bebe program (T.Scherzer *et al.*, 1989). Both Slate and Scherzer implemented rote learning by saving selected entries from the transposition table to disk. These entries were then loaded into the transposition table before a move was searched.

Both Slate and Scherzer reported promising results from their experiments. However the experiments were of a limited nature, and their techniques have yet to gain wide acceptance.

2.4 Opening Books

All competitive chess programs contain an opening book. The purpose of the opening book is to allow the program to quickly play the opening moves, without falling into any well known traps. The opening book is constructed by the programmer, who will often consult chess books or expert chess players for the appropriate knowledge.

An opening book will typically contain several thousand moves. A difference between an opening book and other memory based techniques is that there may be a number of different moves specified for a given position. This is because a certain amount of variation is desirable in order to reduce the predictability of a program. It is also due to the fact that there are often a number of moves of similar merit available in a position.

The opening book is usually stored in a textual format that can be easily edited by the programmer. When the program is loaded, this text file will be loaded into separate hash table for fast access by the program.

2.5 Endgame Databases

While the opening book attacks the game from the start, endgame databases attack it from the other end. This approach has been very successful in the

game of checkers, where the CHINOOK program has recently won the world championship (Schaeffer *et al.*, 1993). CHINOOK's endgame database contains every checkers position with seven or fewer men, plus a large number of the eight men positions, in excess of 70 billion positions in all.

The greater variation of pieces in chess means that the memory requirements for storing all endgames with fewer than seven men would be astronomical. Recent efforts, pioneered by Ken Thompson, have gone into constructing perfect play databases for all interesting endings of five or fewer pieces (including kings) (Thompson, 1986; Thompson, 1991).

Endgame databases are starting to find their way into tournament chess programs. They are typically stored in a compressed format, but still require considerable storage and therefore reside on disk. The program will only access the database if an endgame of the appropriate type is encountered.

Chapter 3

Aims

The general aim of this project is to investigate the viability of using data compression to increase the capacity of memory based systems in computer chess. I have narrowed this aim to focus on the rote learning systems. I will examine the motivation for this aim in the following section, while in section 3.2 I will develop a set of concrete aims.

3.1 Motivation

I have focussed on rote learning systems for a number of reasons. Firstly, it is the least developed of the memory based techniques. The other techniques (see chapter 2) have all been accepted as useful by the computer chess community, while rote learning is not in common use. While it is possible that rote learning is ultimately of little value, the work of Slate and Scherzer suggests that further development of the idea is required in order to make a valid judgement. Also, I see rote learning as potentially the most exciting of the memory based ideas. Certainly the idea of a program improving its play without human intervention is appealing. Finally, there is an obvious application for data compression in current rote learning systems, as explained below.

The rote learning systems developed by Slate and Scherzer both stored transposition table entries onto disk. The problem with this is that the entries do not contain an exact representation of the position, but merely a hash code for the position. To quote David Slate, talking about his experimental rote learning program Mouse:

> "So far, Mouse retains only trans table entries. These are 'anonymous', in the sense that they lose connection with the positions they represent. But the complete position could be stored with each entry. This would permit more intelligent management of the data base as well as some ability to load selectively only those positions relevant to the given stage of the game to avoid clogging the trans table. Samuel employed several heuristics to classify and refine his rote learned checkers data base; similar ones could be used with chess"

Scherzer also notes that the stored entries would be invalidated by any change in the hash key algorithm. Clearly there are significant advantages to storing the complete position. These positions could then be converted into hash codes when they are read into the transposition table from disk.

The systems developed by Slate and Scherzer loaded the entire database of positions into the transposition table. There are a number of problems associated with this approach. Firstly, the capacity of the database could not exceed the capacity of the transposition table. This is quite serious as the database is disk based while the transposition table capacity is limited by available RAM. Also, if the database entries occupy a significant portion of the transposition table there will be 'clogging' problems caused by excessive hashing clashes when the search engine is updating the transposition table. Slate's suggestion of only loading the relevant positions from the database into the transposition table is designed to alleviate these problems.

Although information other than the position is stored in the database, I have not concentrated on this. Typically the position is by far the largest piece of information stored, while the exact nature of the other information will vary between implementations. Chapter 4 section 4.3 provides a brief discussion of move compression.

3.2 Concrete Aims

- Develop a method for compressing chess positions that has the following qualities:
  - high compression,
  - fast decompression,
  - allows random access.

  The goal of high compression is the most obvious, any memory based system is ultimately limited by its total capacity so it makes sense to maximise that capacity. Fast decompression is required because it will be used during the game. Compression, on the other hand, need not be particularly fast as positions to be stored can be batched and compressed after the game when time is not so critical. Random access is desirable to support systems that need to quickly load only the part of the database that is currently relevant.

- Develop a prototype indexing system for chess positions. I decided that this was necessary in order to be able to properly assess the compatibility of the compression methods with random access techniques.

It should be noted that, while the above aims are focussed towards rote learning systems, they are also relevant to other memory based techniques. Of the other techniques, I see opening books as being the closest to rote learning in terms of implementation requirements, and hence they are most likely to benefit from any work developed here. Other application with similar requirements that

could potentially benefit are chess databases of the type used by serious chess players for studying the game.

Chapter 4

Previous Work

A thorough search has failed to locate any published work on the general compression of chess positions. A certain amount of related work was found however, and that is described in this chapter.

It is interesting to note that some of the best programs now run on personal computers and are commercial products. It is likely that these programs contain new and innovative techniques however these are treated as trade secrets and rarely published.

4.1 Compressing Endgame Databases

As discussed in chapter 2, there has been a considerable amount of work done in this area. Unfortunately it is difficult to apply the techniques developed in this area to the compression of positions in a more general sense. The reason for this is that these databases are exhaustive, meaning they contain all possible legal positions for a particular endgame of a particular material balance (eg. K+P vs K). The storage of these endgames is, to my mind, more of a pure encoding problem than a compression problem. The key factor being that the information to be stored in the database is constant and predetermined.

4.2 Entropy of Chess Positions

An interesting experiment attempting to determine the entropy of chess positions from master games was conducted by Nievergelt (Nievergelt, 1977). The experiment involved the playing of a guessing game to estimate the information content of a position. The guesser is allowed to ask multiple choice answers about the position, which the informants will answer to the best of their ability. These questions may be subjective in nature, for example: "Would you say White has a won position, yes or no?". The game continues until the guesser actually guesses the position.

A question with k possible answers is counted as giving $\log_2 k$ bits of information. The information content of a position is taken to be the sum of the number of bits of information given by each question.

This experiment estimated the entropy of positions from master games to be approximately 70 bits, although this figure is of limited accuracy as it is based on only twenty trials. The guessers and informants were either expert or master chess players. It would have be interesting to vary this as experiments have shown a correlation between playing strength and the ability to recall chess positions (de Groot, 1965). Players of grandmaster level may well demonstrate a lower entropy.

4.3 Compression of Chess Moves

The compression of chess moves has been investigated, resulting in an elegant solution (Althöfer, 1991). The method used was based on predicting the most likely moves using a fast deterministic chess program. The three best moves were encoded using a small number of bits, while the remaining moves were encoded using a flat encoding.

The program, which was set to use one second of time in predicting each move, predicted sensible moves often enough for the method to be effective. Testing showed the method to require about 3.9 bits/move while a flat encoding required about 5.2 bits/move. This type of method could be used for storing moves in a rote learning system, although decompression speed may be a problem.

Chapter 5

Indexing

The purpose of indexing the database of learned positions is to be able to quickly load the positions that are relevant to the current game position before performing the search. In this context, positions are relevant if they will occur in the search tree that is created while choosing the move. This problem of loading all relevant positions is far from trivial, and I do not propose to develop an optimal solution. However, a prototype indexing method is developed that should be useful for finding relevant positions, and may form the basis of a more advanced method.

Positions are indexed by the combination of pieces on the board, with the colour of the pieces being significant. For example, every position in which white has K + P, and black has K + B, would have the same *index key*. However a position where white has K + B, and black has K + P, would have a different key. All positions with the same index key are stored adjacently in the file, and such a group of positions is referred to as a block. To support random access there is a synchronisation point at the start of every block, which will be on a byte boundary. An index file contains an array of index records, each of which contains an index key and a pointer to the start of the corresponding block in the database. The index records are sorted by index key to facilitate binary searching.

To load positions relevant to a game position, a program would load all positions having the same key as the game position. This would at least guarantee that the current game position is loaded if it were stored, as well as positions obtained by non-capture moves. It would also be desirable to load positions that have material balances that correspond to removing one or more pieces from the game position. These positions would allow for capture moves occurring in the search tree. The exact number of captures that should be allowed for would depend on the depth that the program is expected to search to.

Chapter 6

Compression Techniques

There are a potentially bewildering array of techniques in use in the data compression field (Bell *et al.*, 1990; Witten *et al.*, 1994). In general I have tried to avoid anything too complicated due to having limited time for implementation work. My general philosophy has been to use experimentation to get an idea of the potential of a particular method, and not necessarily to find an optimal implementation.

6.1 Simple Encoding

It is estimated that there are in the order of 10^{43} possible chess positions (Shannon, 1950), meaning that about 143 bits are required to represent any given position. Unfortunately there is no known simple and efficient encoding scheme that achieves this performance.

I will examine a simple technique for encoding a chess position. This will provide an overview of the information that must be stored and also a conservative upper bound on the number of bits require to store this information.

A chess position consists of chess pieces arranged on a chess board, along with some associated state information. The state information must include the side to move, castling flags, and *en passant* information. There are four castling flags to represent whether each side has the potential for king-side and queen-side castling. The *en passant* information stores whether the previous move involved a pawn moving two squares. If the previous move was a double pawn move, the *en passant* information must also contain the file on which the pawn moved.

To fully describe a position in a chess game, it is also necessary to store two further pieces of information. Namely the number of moves since the last pawn move or capture, and the number of times the position has occurred previously in the game. This information is necessary because of the rules that allow draw by three-fold repetition of position and draw after 50 moves without capture or pawn move. For the purposes of this project, I have ignored this extra information, mainly because the major source of my test data did not include it. The loss is not serious however, as the missing information is of little importance in the context of rote-learning.

To represent the arrangement of pieces on the board, a simple approach is to use an integer for each of the 64 squares. A square may be empty, or contain a piece of which there are six types and two possible colours, 13 different possibilities in all. Thus a four bit binary number can be used to represent the contents of a single square. The contents of the entire board can then be represented in $4 \times 64 = 256$ bits.

To store the state information, a single bit is required to store the side to move, plus four bits are needed for the castling flags. The *en passant* information requires one bit to indicate whether the previous move was a double pawn move, and another three bits to indicate the file of the pawn move, four bits at most. The state information can therefore be represented in $1 + 4 + 4 = 9$ bits. So an upper bound for the memory required to represent a chess position is $256 + 9 = 265$ bits.

6.2 A Symbolwise Approach

The simple encoding discussed in section 6.1 can be loosely described as a symbolwise approach as it treats the arrangement of pieces on the board as a succession of 64 symbols. This approach, which was chosen for its conceptual simplicity, was used as the basis for a number of experiments described in this section. All of the symbolwise compression methods described below read the pieces from the board in the same order. This ordering is from **a8** to **h8**, then **a7** to **h7**, etc. The choice of this ordering was somewhat arbitrary, although no compelling reason was seen for trying other orderings.

The methods in the following subsections use successively more powerful models to generate probabilities for each symbol. These probabilities are used to encode the symbols by means of a 14-bit arithmetic coder.

6.2.1 Order Zero Model

The order zero method was the simplest of all those implemented. Separate order zero models are used for the pieces, side to move flag, each castling flag, and the *en passant* information. A semi-static approach is used, meaning that two passes through the file are required. On the first pass, counts are made of the various symbols. It should be noted that only a single model was maintained for the pieces regardless of which square they were on, not a single model for each square. The results of this method, and those in following sections, will be discussed in chapter 7.

6.2.2 Using Rules

To improve the previous method, the order zero model for the pieces was augmented by using rules to alter the probabilities. The following rules are used:

- Pawns do not occur on the first and eighth ranks
- There is always one king of each colour

- There are never more than eight pawns of a given colour
- There are seldom more than two rooks, knights or bishops
- There is seldom more than one queen of each colour

All of the rules, excepting the first, refer to a single position. For example, after a white king has been seen in a position, the probability of a white king occurring on one of the remaining squares in that position becomes zero. The last two rules are not absolute but refer to events which are extremely unlikely. I use the arbitrarily low probability of 1/10000 for these events.

6.2.3 Using the Index

Because of the way the positions are indexed, the decompresser will actually know the exact combination of pieces present in the position. This information can be used to aid compression, and illustrates the fact that an index contains redundant information.

The information describing the number of pieces of each type present can be maintained by simply decrementing the count for a piece type whenever a piece of that type is encountered. This can have a dramatic effect on compression as illustrated by the example of figure 6.1. The first piece to be encoded in this example is the black king on **a8**. The compressor knows that the exact makeup of the 64 symbols that will occur, namely one black king, two black pawns, one white king and 60 empty squares. It also knows the rule that pawns cannot occur on the first or eighth ranks. Therefore, $\frac{1}{62}$ is used to encode the king being on **a8**. The black pawn on **a7** is encoded with probability $\frac{2}{56}$ because the compressor knew that of the remaining 56 symbols, two must be black pawns. The empty squares from **d7** are encoded with probability 1 because the compressor knows that there are only empty squares left.

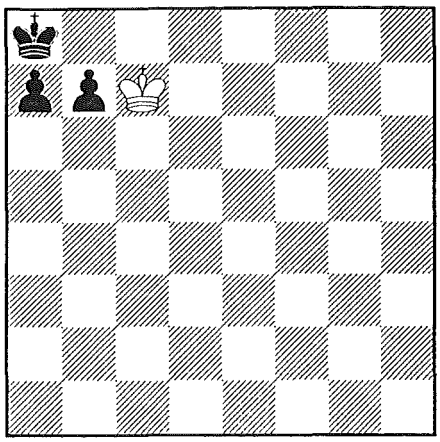
6.2.4 Additional Rules

I implemented an additional rule to improve the model used to predict the castling flags. This rule gives the compressor the knowledge that each castling flag can only be set if the appropriate king and rook are on their home squares.

There are many other possible rules that can be added, for example the two kings cannot appear on adjacent squares. A heuristic to reflect the territorial nature of the game is an area open to future research.

6.3 Differential Compression

While the symbolwise methods in section 6.2 focus on individual positions, differential methods put more emphasis on the similarities between positions in the same block. The first position in each block is encoded using a symbolwise method, however each successive position is described by a number of delta transformations from the previous position. The state information is compressed using the method described in section 6.2.4.



Probabilities to encode:
$\frac{1}{62}\ \frac{60}{61}\ \frac{59}{60}\ \frac{58}{59}\ \frac{57}{58}\ \frac{56}{57}\ \frac{55}{56}\ \frac{54}{55}$

$\frac{2}{56}\ \frac{1}{55}\ \frac{1}{54}\ 1\ 1\ 1\ \ldots$
(Encoded in 22.47 bits)

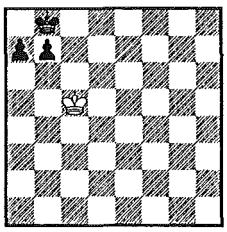
Figure 6.1: Symbolwise index driven compression example

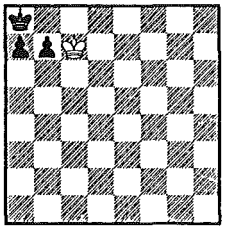
A delta transformation moves a piece from one square on the board to another. As the combination of pieces on the board is the same throughout the entire block, transformations for adding and deleting pieces are not required.

An example of delta compression is given in figure 6.2. The coding of the number of deltas is straightforward, although it should be noted that a position may need no deltas, the state information is enough to distinguish between positions. Encoding the delta to move the black king involves first encoding the selection of the black king. The deltas are encoded in order of the square of the moving piece, using the normal ordering of **a8** to **h8**, then **a7** to **h7**, etc. Because we are encoding the first delta of two, the last of the pieces in the ordering (white king) can't be the piece moving. Therefore the probability that it is any other piece is $\frac{1}{3}$. The destination square can be any square other than the piece's original square, or the destination square of a previous delta, hence the probability $\frac{1}{63}$. The second delta is coded using similar logic.

The differential method is also likely to benefit from the use of rules similar to those used by the symbolwise methods. A number of possibilities exist, although they were not implemented. For example, when coding deltas that move pawns, a rule containing the knowledge that the pawn cannot be placed on the first or eighth ranks would improve compression performance.

The order of positions within a block is not important in the context of the rote learning application because the application will be reading in the entire block. The compression techniques are therefore free to alter this ordering. Experimentation showed that the use of a simple greedy algorithm improved compression performance markedly. This algorithm does not reorder the first position to be compressed, however the following positions to be compressed





Probabilities to encode:
2 transformations, minimum of 0, maximum of 4: $\frac{1}{5}$
Move the black king: $\frac{1}{3}\ \frac{1}{63}$
Move the white king: $\frac{1}{3}\ \frac{1}{62}$

(Encoded in 17.10 bits)

Figure 6.2: Differential compression example

are chosen from those remaining in the block, the criteria for choice being to minimise the number of deltas in the next compression.

The problem of ordering the positions actually maps onto the problem of finding a minimal cost spanning tree in a graph. Kruskal's algorithm can be used to find minimal spanning trees (Kingston, 1990), although this was not implemented.

Chapter 7

Results

To test the methods described in chapter 6 a collection of 28459 unique positions was compiled (see appendix B). This file, which uses a text based format, requires 422 bits/position of storage. The collection was of a heterogeneous nature, in that positions from all phases of the game were included. The positions came from a number of sources, but most were ultimately the result of tournament praxis.

7.1 Compression Performance

To put the performance of my methods into perspective, a number of general purpose compression tools were run on the test data. The results of these tests are shown in figure 7.1.

The `compress` program is the standard Unix compression utility, while `bdmc` is an implementation of Dynamic Markov Compression (DMC). The `ppmC` program uses Prediction by Partial Matching (PPM), `zip` is a program compatible with the popular PKZIP (Phil Katz ZIP). The best performer, `gzip`, is distributed by the Gnu Free Software Foundation and is based on the LZ77 method. All programs were run on maximum compression settings where available.

The results for a number of the compression methods described in chapter 6 are shown in figure 7.2. The result of the best general purpose program, gzip, is included for ease of comparison. The size of the index is counted as contributing to the bits/position figure in the case of those methods that used the index to aid compression ie. ind1, ind2, diff.

The sym1 method is the simple order zero method as described in chapter 6 section 6.2.1. The sym2 method is the simple rule based method improvement on sym1 as described in chapter 6 section 6.2.2. The ind1 method is the index driven method described in chapter 6 section 6.2.3. The ind2 method is ind1 using the improved model for castling flags as described in chapter 6 section 6.2.4. The diff method is the differential method described in chapter 6 section 6.3.

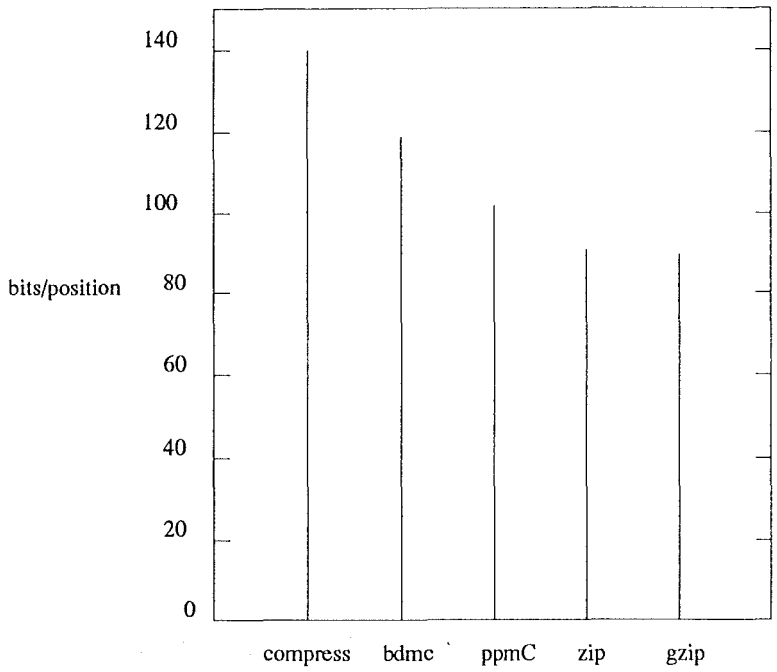


Figure 7.1: Compression Performance of General Compression Programs

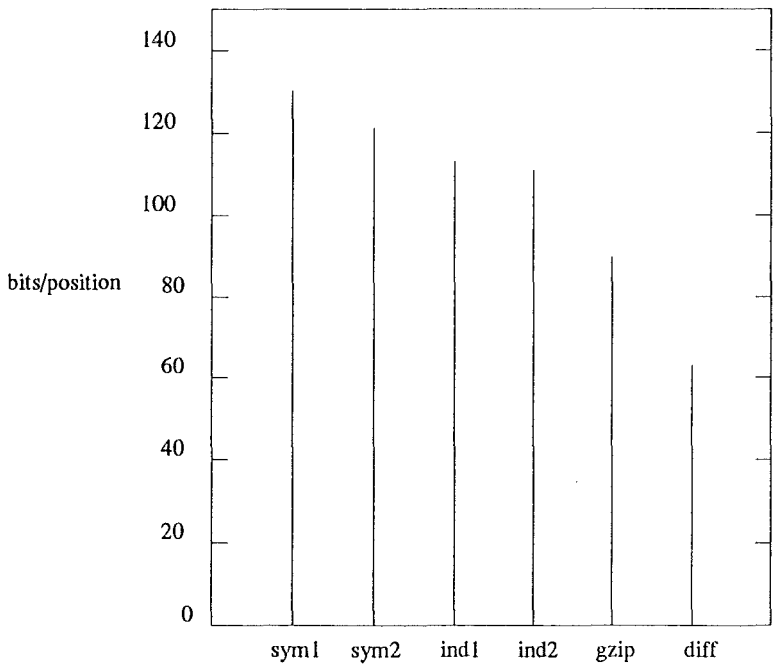


Figure 7.2: Compression Performance of Experimental Methods

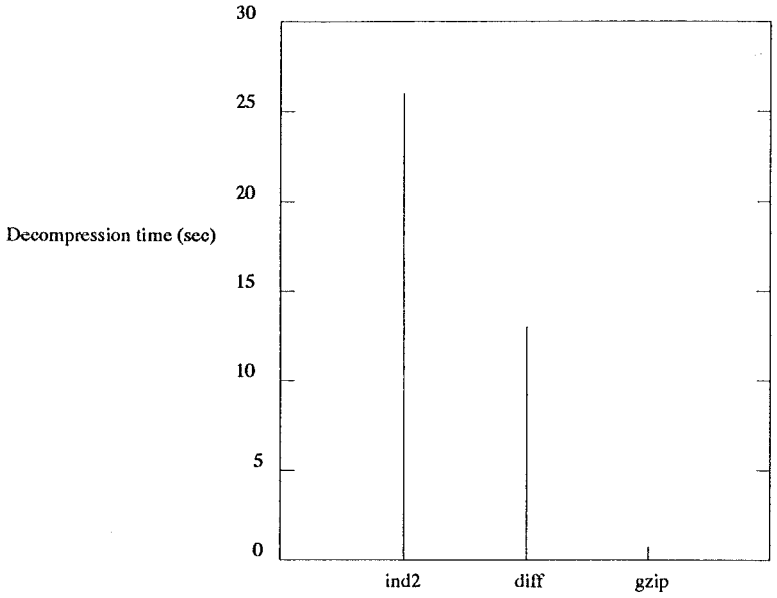


Figure 7.3: Decompression Speed of Selected Methods

7.2 Decompression Speed

Figure 7.3 shows decompression times for selected methods, on my entire test data file. The tests were performed on the department's SPARC 10 based file server, during off peak hours.

Chapter 8

Conclusion

The differential compression method comes closest to meeting the aims as set out in chapter 3, section 3.2. Its compression performance exceeds all other methods discussed in this report, and is even superior to the entropy figure given in chapter 4. This last comparison however, is not a fair one for two reasons. The first reason being that the entropy figure referred to positions selected at random from master games. The positions in my test file were most definitely not selected at random, in fact many of them were closely related, being from the same game. The second reason being that the differential method loses information by not preserving the original order of the file.

For the purposes of rote learning however, this second point above is not important. The first point too can be countered by observing that a rote learning system is likely to store positions from the same game. Hence the compression performance of the differential method was still a significant result in terms of the overall goals of this project.

Although the decompression speed of the differential method was superior to the symbolwise methods, it was out performed by an order of magnitude by `gzip`. Although further optimisation of the program is doubtless possible, more fundamental changes would be necessary be make large speed improvements. A possible area of investigation would be to bypass the arithmetic coder in favour of a faster coding technique, even at the expense of a small amount of lost compression.

The differential compression technique also proved to be compatible with random access requirements. This was evident by its compression performance, even taking into account the storage of the the index. The utility of the actual indexing technique however is difficult to evaluate without actually implementing a full rote learning system.

Chapter 9

Acknowledgements

I would like to thank the following people: Bruce McKenzie for his helpful suggestions, Shane Hudson for helping with TEX, and Carl Ansley for helping with implementation details.

References

Althöfer, Ingo. 1991. Data compression using an intelligent generator: the storage of chess games as an example. *Artificial intelligence*, **52**, 109–113.

Bell, T.C., Cleary, J.G., & Witten, I.H. 1990. *Text compression.* Englewood Cliffs, N.J.: Prentice-Hall.

de Groot, A.D. 1965. *Thought and choice in chess.* The Hague: Mouton.

Kingston, Jeffrey H. 1990. *Algorithms and data structures.* Addison Wesley.

Nievergelt, Jurg. 1977. Information content of chess positions. *Sigart newsletter*, Apr., 13–15.

Samuel, Arthur L. 1959. Some studies in machine learning using the game of checkers. i. *IBM journal of research and development*, **3**, 210–229.

Schaeffer, Jonathan, Treloar, Norman, Lu, Paul, & Lake, Robert. 1993. Man versus machine for the world checkers championship. *A.I. magazine*, 14(2), 28–35.

Shannon, C.E. 1950. Programming a computer for playing chess. *Philosophical magazine*, 41(7), 256–275. See also Levy, D.N.L (ed.) (1988), Computer Games I, Springer Verlag, New York, pp.81-88.

Slate, D.J. 1987. A chess program that uses its transposition table to learn from experience. *ICCA journal*, **10**(2), 59–71.

Thompson, K. 1986. Retrograde analysis of certain endgames. *ICCA journal*, **9**(3), 131–139.

Thompson, K. 1991. New results for KNPKB and KNPKN endgames. *ICCA journal*, 14(1), 17.

T.Scherzer, L.Scherzer, & D.Tjaden. 1989. Learning in bebe. *Chap. 12 of:* T.A.Marsland, Jonathan Schaeffer (ed), *Computers chess and cognition.*

Turing, A.M., Strachey, C., Bates, M.A., & B.V.Bowden. 1953. Digital computers applied to games. *Pages 286–310 of:* Bowden, B.V. (ed), *Faster than thought.* Pitman.

Witten, Ian H., Moffat, Alistair, & Bell, Timothy C. 1994. *Managing gigabytes.* New York: VNR.

Zobrist, A.L. 1970. *A hashing method with applications for game playing.* Tech. rept. 88. Computer Science Department, University of Wisconsin.

Appendix A

Chess Notation and Terminology

The basics of the algebraic chess notation used throughout this report are explained here. A small number of chess specific terms are also explained.

The algebraic notation has largely superseded the older descriptive notation, mainly because of its greater simplicity. Each square on the chess board has a unique name, as shown in table A.1. Chess diagrams are always shown from whites perspective, with the white pieces starting at the bottom of the diagram. This means the bottom left square is always **a1** and the top right one is always **h8**.

a8	b8	c8	d8	e8	f8	g8	h8
a7	b7	c7	d7	e7	f7	g7	h7
a6	b6	c6	d6	e6	f6	g6	h6
a5	b5	c5	d5	e5	f5	g5	h5
a4	b4	c4	d4	e4	f4	g4	h4
a3	b3	c3	d3	e3	f3	g3	h3
a2	b2	c2	d2	e2	f2	g2	h2
a1	b1	c1	d1	e1	f1	g1	h1

Table A.1: The names of the squares

The various ways of referring to the different chess pieces are shown in table A.2.

Chess moves are written by first writing the name of the piece that is moving, and then writing the name of the square it is moving to. The default piece is the pawn, so pawn moves are written by specifying just the destination square. Captures are sometimes emphasized by the use of the 'x' symbol.

As a example, consider a game starting with the moves **1.e4 Nf6 2.Bc4 Nxe4**. The position resulting from these moves is shown in figure A.1.

The chess terms *rank* and *file* are used in this report. File refers to a column of squares on the chess board. For example, the squares from **a1** through to

Symbol	Letter	Name
	K	King
	Q	Queen
	R	Rook
	B	Bishop
	N	Knight
	P	Pawn

Table A.2: Chess pieces and their representations

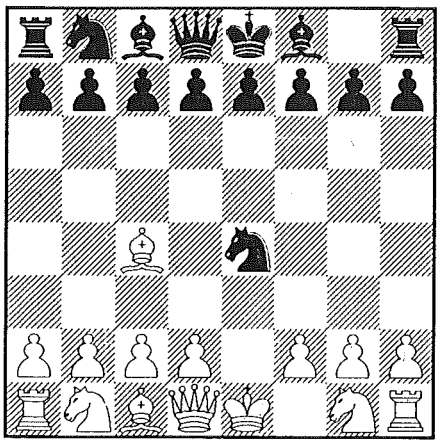


Figure A.1: An example position

a8 are called the 'a-file'. Rank refers to a row of squares on the chess board. For example, the squares **a1** through to **h1** are known as white's first rank, or black's eighth rank.

Appendix B

Test Data

The test data was obtained from the ftp site ics.onenet.net (164.58.253.10). This site is maintained by readers of the Usenet newsgroup rec.games.chess.

The data was in Extended Position Description (EPD) format, which is defined as part of the Portable Game Notation (PGN) standard. The PGN definition document is also available at the above ftp address.

The following is an example of EPD notation:
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -

The position described is the starting position. Lower case letters represent black pieces, upper case represent white pieces. The 'w' represents white to move, and the 'KQkq' represent all the castling rights as being intact. The final '-' signifies that the last move was not a double pawn move.

By for the largest part of my test file was made up of 22212 positions obtained by taking every unique position from a collection of the games of Vassily Smyslov. The remaining 6247 positions were obtained from various encyclopaedic collections of interesting positions. The total file size was 1501644 bytes.