University of Aarhus
Department of Computer Science
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark

9th May 2005

Generation and compression of endgame tables in chess with fast random access using OBDDs

Made by: Jesper Torp Kristensen, 19993197
Supervisor: Peter Bro Miltersen
This thesis contains 153 enumerated pages.

Contents

1 Introduction

In this thesis it is investigated whether ordered binary decision diagrams (OBDDs) can be used as a compact representation for chess endgames. A glossary is provided in appendix H. On my home page the program can be downloaded and links to most of the references are given.

`http://www.daimi.au.dk/~doktoren/master_thesis/handin/`

The most widespread endgame format today is that of Nalimov. The Nalimov endgame files can be used both in a block compressed form or uncompressed. When a chess program probes these tables, the disk access causes a huge overhead. Even though a LRU caching improves this situation, it is still expensive to probe in high depth searches.

If the block compressed tables are moved into main memory, the probing speed is still orders of magnitudes slower than the time needed to evaluate a chess position.

This thesis demonstrates that OBDDs are able to achieve a compression ratio very close to that of Nalimov's, while the probing costs are kept down. This requires of course that the OBDDs are kept in the main memory, but as all the 2-4 men endgames take up only 30 MB of memory this is reasonable.

Experiments are also performed where the distance to mate information is reduced to simply saying whether the position is won, lost or drawn. Here OBDDs totally outperform run-length encoding, which was used in the checkers program Chinook. All 2-4 men endgames now reduce to a single MB. A recently tried representation using decision trees grown with an ID3-style algorithm is able to compress slightly better than the OBDDs in this thesis, but at the expense of a slower probe speed.

The compression achieved results from combining a compact representation of OBDDs with heuristic mapping of don't care entries and reordering using the sifting algorithm. Also, choosing a non standard enumeration of the board squares reduces the size of the OBDDs.

The endgames generated in this thesis differ slightly from the work of others. I have included the positions with castling, which are generally ignored as not being worth the trouble. Oppositely, my implementation is able to recognise most of the unreachable positions. By labelling these "don't care", the compressibility of the OBDDs improves.

Finally, a new theoretical result is shown, stating that minimisation of OBDDs using don't cares is NP-complete. This has previously been shown for the case where the input is represented as an OBDD. However, if we change the input model to a table, then this proof is invalidated by an exponential blowup of the input description. Oppositely, the NP-hardness shown here, for the problem with input given as a table, immediately gives the other result.

The proof is a reduction from the Minimum Clique Partition problem (or equivalently; the Graph Colouring problem). Very good inapproximability results are known for this problem, and they translate in a weaker form to the problem of finding an optimal mapping of don't care entries.

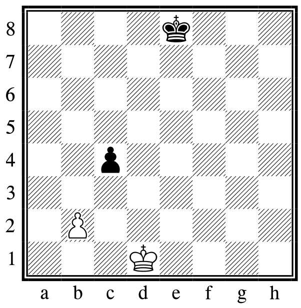
1.1 Rules of chess

The reader probably know the basic rules of chess.[1] However, this thesis is about chess endgames and I would like to be able to assume the familiarity with even the more subtle details:

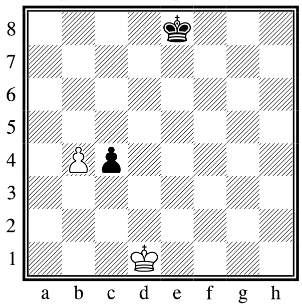
Castling: The king and rook performing the castle may not previously have been moved. Also the king may not be in check or pass a threatened square.

En passant: If the last move was moving a pawn 2 squares forward then this piece can be captured "en passant" by an enemy pawn as illustrated below.

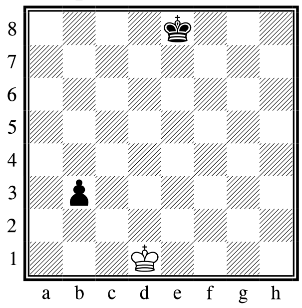
Before:



En passant possible:



En passant taken:



50 move rule: If there have been no pawn move or capturing move among the last 50 moves by each side, then the game is a draw.

3. repetition rule: If the current position has occurred twice before, the game is a draw. A position is determined by the pieces on the board, the side to move, en passant if possible, the side to move and the castling capabilities of the kings.

Check: The side-to-move may not leave its king in check (this may seem like a superfluous rule as a player breaking it would immediately lose the game. This rule however turns out to be quite useful in the process of compressing endgame tables as it reduces the number of legal positions).

1.2 What is an endgame table?

The size of the state space in chess have been estimated between 10^{43} and 10^{50} [3] and we are no where close to solving it. What have been done is to divide this

[1]If not, a full description is given in appendix D.

state space up into one subclass for each combination of pieces left in the game. The subclasses with few pieces are manageable in size and feasible to solve using a retrograde algorithm.

1.2.1 Naming conventions

Each of the subclasses is conventionally named by concatenating one letter from each piece. As an example, the name KQKN refers to "white **K**ing and **Q**ueen versus black **K**ing and k**N**ight". The letters belonging to white pieces are placed first. The order is **K**ing, **Q**ueen, **R**ook, **B**ishop, k**N**ight and **P**awn.

There is no KNKQ endgame — at least not in Nalimov's format or mine. Instead the KQKN endgame has 2 halves, one for white to move and one for black to move. The reason is that the KQKN endgame can be used to probe KNKQ positions, it is simply a matter of switching the colours of the pieces and mirror the board horizontally.[2] Endgame tables are also referred to as tablebases.

1.2.2 Distance to mate (DTM)

Traditionally, an endgame table provides a mapping from position to distance to mate values. A distance to mate value is either the value drawn, a win in $n > 0$ moves or a loss in $n \leq 0$ moves. The distance to mate information combined with trying every move in a position, allows you to select a move that brings you one step closer to the actual mate.

The endgame KBBKN is a general win for white. However, before this endgame table was constructed it was considered a draw by most players [22]. It has no intuitive winning strategies, so without the distance to the mate information even the best players or programs have difficulties making progress towards the mate.

This is in contrast to e.g. the situation in checkers. The world champion program *Chinook* uses endgame tables saying only whether the position is won, drawn or lost [19]. *Chinook* has never reached a won position, where it couldn't enforce the actual win.

The 50 move rule has an unintended impact on e.g. the KBBKN endgame. The 50-move rule was originally designed to enforce a draw in positions where nobody wanted to take the initiative. This was before the invention of endgame tables and before having identified any position to be a win in more than 50 moves. Hence, the effect of this rule, making some otherwise won positions drawn, is actually undesired. This is why the distance to mate tables ignores this rule.

[2]The horizontal reflection is made necessary by the movements of the pawns. With no pawns, it is not needed.

1.2.3 Distance to conversion (DTC)

Thompson's tablebases takes the 50 move rule into account by using the distance to conversion measurement instead. A conversion is either a pawn promotion or a piece capture. To illustrate why this is useful, consider a KBBKN position that is won in e.g. 60 moves. Unfortunately, every optimal line of play (ignore the 50 move rule) might start out by doing at least 52 non pawn, non capturing moves and enforce a draw. But there might be a winning strategy in which less than 50 moves are made before the knight is captured, but which require a larger total number of moves.

By far the most widespread format is the distance to mate. This is easiest to use in a chess program, and is due to Nalimov also the format in which most endgames are available. At present time the construction of nearly all 6 men endgames have completed.

1.2.4 Win, draw, loss information (WDL)

This information alone is generally not enough for a chess program to win a won position. However, only 2 bits are needed per position. If a chess program has proven capable of winning certain endgames, then memory can be saved by storing only this information.

1.3 The use of endgame tables

As I see it, the use of endgame tables can be divided into 3 categories — each of which sets different demands to respectively size and access time.

Human aid: You are playing e.g. a game of correspondence chess and have reached a position with only 6 pieces left. You are not playing fair, so you wish to consult an endgame database. For this purpose an ideal solution would be to consult a chess endgame server on the internet so you wouldn't need to download several GB just for this endgame.[3]

Computer program having reached a winning position: For each legal move find its value by making it and indexing the endgame tables. The best move is guaranteed to make progress. A disk access for each legal move is acceptable.

Computer program in a position with few pieces left: The program performs a search from the current position. A state-of-the-art program is able to investigate around 2 million positions per second on a 3 GHz pc. If the endgame tables are probed on a hard disk for even a tiny fraction of the positions, then it will totally kill the performance.

[3]One 3-5 men endgame server is `http://www.lokasoft.nl/uk/tbweb.htm`

The focus of this thesis is entirely on the last category.

It is hard to state the requirements to access times and the size of the endgame tables precisely. The major aim is to make the chess program stronger, but how many resources is it reasonable to reserve for this specific purpose?

Appendix A gives a short overview of the common building blocks of nearly every chess program. An important part is the use of transposition tables. This is a data structure used to cache previously evaluated positions whereby the search tree is pruned. Obviously, by caching more positions, more pruning will be possible and the program will be able to search deeper in the same time, hereby making it stronger. Hence, if we reserve more memory for the use of endgame tables, we do it at some expense. Equivalently; if one probe in the tablebases takes as long as searching n positions, how low should n be before it pays off?

In the checkers program Chinook, endgame tables play an important role. In the article [19] describing the work on these endgame tables, the authors mention that:

> This work have been applied to the domain of computer checkers (8 x 8 draughts), which is an interesting point of comparison for computer chess. In checkers, since there are only checker and king pieces, all games play into a limited set of endgame classes. Also, the lower branching factor of checkers trees and the force captures of the game result in deeper search trees than in chess. Although the root of the tree may be far from the endgame, the leaf nodes already may be in the databases. Consequently, the utility of the endgame databases is higher in checkers than chess.

In chess, the game has usually been decided before the point where the endgame tables become useful. Hence it is unwise to use more than a small part of the memory, say 10%, on the representation of endgame tables.

1.4 Related work

Ken Thompson[4] was a pioneer in the area of generating endgame tables for chess. Later tables generated by S. J. Edwards were also available, but today the most commonly used tables are those of E.V. Nalimov. The reasons are many — they enjoy the most compact representation and are available for nearly all 6 men endgames by now. Just as important, a LRU caching system greatly reduces the number of disk accesses when probed during a search.

[4]Better known as the key designer of the UNIX operating system

1.4.1 Endgame construction

An endgame is constructed by a retrograde analysis. This will be described in section 3. At an abstract level the algorithm is quite simple — most of the challenge lies in how to implement things like move generation.

Much of the work have had as goal to push the limits on the size of the endgames that could be constructed. This includes finding ways of decomposing each endgame into smaller fractions that each can be solved with a small amount of interaction with the other fractions (see e.g. [22]).

Another focus have been on the how to produce efficiently computable mappings from chess endgames to indexes in the endgame tables. The article *Space-efficient indexing of chess endgame tables* [21] from september 2000 gives an excellent review of this area.

Neither the endgame construction or indexing strategy used in this thesis are at the level of the research frontier at these points.

However they are necessary as building blocks for the main focus with this thesis — the compression using OBDDs. In this connection, my endgame construction algorithm is *not* the weakest link in the chain. Also, for reasons that will become clear, the indexing strategy is as good as it needs to be.

In the book `Scalable Search in Computer Chess` [20] the author mentions that the chess program RETRO, made by Dekker in 1989, should have included both positions with en passants and castling rights in its endgame tables. He's however unaware of any publication about it, and to my knowledge, this is the only prior attempt to include positions with castling rights in the endgame tables. In section 4 I present a way of encoding positions with castling rights as otherwise illegal positions.

1.4.2 The work of Ernst A. Heinz

What got me started on the topic of compression of endgames was reading the book "Scalable Search in Computer Science" by Heinz [20].[5]

He describes how it has been possible to reduce the memory requirements for all 3-4 men endgames by a factor of 11 down to 15 MB, such that they would easily fit into the main memory. By comparison, these endgames takes up 30 MB in Nalimov's compressed format.

This optimisation has been achieved without the use of actual compression. Instead he has analysed how much information can be thrown away without compromising the strength of his world class chess program DarkThought. The endgames have been divided into 4 categories:

Trivial won: E.g. KRRK. No table is needed, just some code to recognise the stale mate positions.

[5] My first aim was to use data mining techniques to train a neural net or the likes.

Won or drawn: E.g. KBPK. One bit per position tells whether white has won or it's a draw.

Won, drawn or lost: E.g. KQKQ. 2 bits are needed per position

Distance to mate: This full information is only used for the small KRK, KQK and KPK endgames.

When distance to mate information is thrown away, the game theoretical value has to be combined with some heuristic evaluation of the position to ensure that the mate is eventually reached.

In the end Heinz proposes how the 5 men endgames can similarly be reduced, but that even further optimisations would be needed to make it feasible:

> Standard techniques for general-purpose data compression (especially Huffman, Lev-Zimpel-Welch, and run-length encoding) look like promising candidates for future trials to overcome the size limitations.

I found it a bit strange that he didn't use any compression at all. He has put quite an effort into the design of a space efficient indexing of positions. He could have gained a lot more by using the simple "compression" described below, at the cost of introducing only a few extra instructions in the probing code.

Simple compression of win/draw/loss values: We are given the endgame table

$$t : [0..N[\mapsto [0..4[$$

The interpretation of the values could be $0 \mapsto$ `drawn`, $1 \mapsto$ `won`, $2 \mapsto$ `lost` and 3 $\mapsto$ `illegal`. Let c be a listing of the M different tuples $t[8k..8 \cdot (k+1)[$ in the table t.

$$c : [0..M[\mapsto [0..4[^8$$

We introduce a table t' to index c. Together t' and c define t:

$$t' : [0..\lceil N/8 \rceil[\mapsto [0..M[$$
$$t[i] = c[t'[\lfloor i/8 \rfloor]][i \mod 8]$$

The compression achieved depend on the number of different tuples M. In many endgames, one value is clearly dominant (e.g. white wins). By reassigning any `illegal` value to this the dominant one we get at most 3^8 different tuples, but it is likely much lower. If we assume $M = 243$, we get the following compression ratio

$$\frac{\lceil \log(M) \rceil}{8 \cdot 2} = \frac{\lceil \log(243) \rceil}{16} = \frac{8}{16} = 0.5$$

that is, if we disregard the size of c:

$$243 \cdot 8 \cdot 2 \text{ bits} = 3888 \text{ bits} = 486 \text{ bytes}$$

1.4.3 Infallible rule-based endgame play

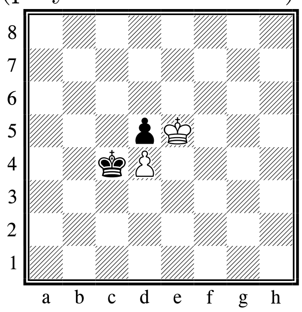
Many individual endgames have been analysed with the purpose of extracting rules producing an optimal strategy. This would avoid the need of having all values tabulated. Most automatic rule derivations are based on decision trees. In [20] a lot of references is given, but every reference seems to focus on some small subset of endgames.

A recent attempt by Johan Melin [23] uses decision trees grown with an ID3-style algorithm combined with run-length encoding of the exceptions (positions misclassified by the decision tree). It achieves a very good compression. However, judging from his brief description it seems that the probing speed varies a lot, and that the decision trees may even yield "don't know" values.

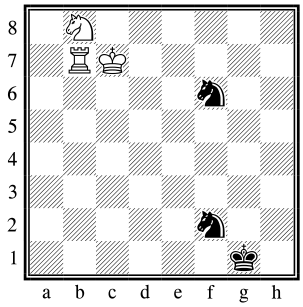
1.4.4 Endgame mining

This is basically the search for chess positions that are interesting in some way. Examples include:

Full point mutual zugzwang
(player to move loses)



Longest known mate -
white moves and wins in 262



For this thesis a position is interesting if it is illegal. Because then there is no need to store it and we save space. In [6] they conclude that it is non-trivial to decide whether a position is legal in the sense that it is reachable from the initial position. In section 12 this definition of legal is approximated by one based one local reachability.

1.5 Goal of this thesis

The compression of endgame tables using OBDDs is an attempt to give a solution for the problem left open by Heinz. Nalimov's tablebases fulfil their purpose excellent, but they have not been designed for use in high depth searches. Here it is my hope that the representation using OBDDs will provide a good alternative. The aim is not only to examine whether the OBDD approach has this potential, but also to use it to construct the following endgames.

- All 3-4 men endgames with DTM information.

- All 5 men endgames with WDL information.

I assume that the 5 men endgames with WDL information reach the outer limit of how much memory is reasonable to use for endgames.

1.6 How to navigate this thesis

Section 2: This is a case study of the KBBK endgame. It provides an illustrative example of how the trivial indexing scheme can be improved. Also, it demonstrates that domain specific knowledge can be used to order the data in a way that makes it more compressible.

Section 3: Present two slightly different algorithms for generating endgame tables and compares their performance.

Section 4: Describes how to map each set of endgame positions to a compact index range. The exploitation of symmetries requires a few modification to one of the algorithms for constructing endgames. The new way of encoding castling capabilities is presented.

Section 5: Explains why the OBDD representation was chosen. This includes going into details about some of the implementation. Also, a less space efficient indexing scheme is presented. This index scheme assures that each bit used to index an OBDD translates to some simple property of the chess position. Hereby the OBDD should be able to recognise more structure in the data and achieve better compression.

Section 6: This is an attempt to reduce the size of an OBDD. A heuristic algorithm is presented. The algorithm permutes the enumeration of the nodes of each layer in the OBDD. Afterwards many nodes will have the property that the index of their right child is 1 higher than the index of their left child. This halves the size of the description of each such node and results in an overall saving of 20 percent.

Section 7: The index scheme used by the OBDDs contains don't care entries. Several articles have described how to minimise the size of OBDDs using a heuristic assignment of don't care entries. In this thesis the circumstances are somehow different. On one hand, it is feasible to give the input in the simple form as a table and not as an OBDD. On the other hand, the tables are still too large for a n^2 algorithm to work in reasonable time. A faster algorithm is presented. Also, it is demonstrated on a concrete example that this algorithm has a miserable worst case performance.

Section 8: The minimisation of OBDDs that was solved using a heuristic in section 7 is shown NP-complete. This has previously been shown for two other input models, but these results do not translate to the case where the input is simply given as a table. Oppositely, the result shown here immediately gives the other way. Also, an inapproximability result is shown. This states under the assumption P $\neq$ NP and NP $\subsetneq$ coRP that the problem can't be approximated within $N^{\frac{1}{4}-\epsilon}$ for any $\epsilon > 0$.

Section 9: In the experiments with the KBBK endgame in section 2 it was possible to improve the compression by using another enumeration of the squares of the board. This section shows that compression achieved by OBDDs depends in a less predictable way by this enumeration. All algorithmically attempts failed, but a hand coded enumeration similar to the Z curve turned out to be superior to the standard enumeration.

Section 10: This section presents the sifting algorithm. The sifting algorithm reduces the number of nodes by doing a local search on the ordering of the variables in the OBDD. Interactions with the mapping of don't cares can degrade the performance, but in average a clear improvement is made.

Section 11: This section examines whether clustering can be used to split up an OBDD into several which together will be smaller in size. Initial tests showed improvements, but this seems to have been restrained by the other optimisations. Presently it is only able to improve compression in few cases.

Section 12: No definition of a legal position is given by the official chess rules. By defining a position to be legal if and only if it is reachable from the starting position would render most positions illegal. The definition used by e.g. Nalimov covers less illegal positions but is easy to implement. This section examines how many more positions will be rendered illegal, if we insist that a legal position should be reachable from a legal position (in Nalimov sense) in at most n moves. Less legal positions means less to compress, but it turns out that only a fractional improvement is made.

Section 13: This section compares the performance of final version of the OBDDs with the work of Nalimov and Melin. About the same level of compression is achieved, but the OBDDs are much faster to index.

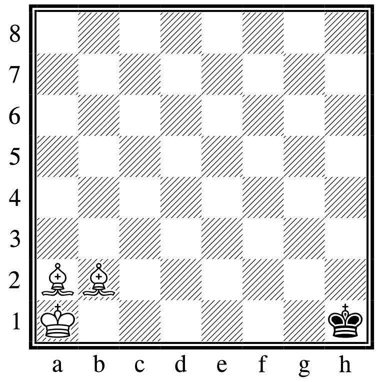
Section 14: Concludes the thesis.

2 Experimenting with the KBBK endgame and gzip

Before I decided finally to write about compression of endgame tables I did some experiments with the KBBK endgame. The objective was to examine to what extend using domain specific knowledge could improve the compression of the table.

2.1 Trivial representation of KBBK endgame

In this case study we are using the distance-to-mate values. The position below is an example of a position that is won in 15 moves if it is white to move (assuming perfect play by both sides).



In the KBBK endgame a position is determined uniquely by the positions of the 4 pieces and the side-to-move (no en passant or castling will be possible). Hence a (too) simple solution is to store the data in a 5 dimensional array. 4 of the dimensions are the positions of the pieces on the board (each of size 64) and the last is the side-to-move (size 2). The table will be indexed like this:

KBBK[white king][white bishop 1][white bishop 2][black king][current player]

“current player” will be either 0 (white to move) or 1 (black to move). The squares on the board are enumerated in the a1-a8-h8 order:

	a	b	c	d	e	f	g	h	
8	56	57	58	59	60	61	62	63	8
7	48	49	50	51	52	53	54	55	7
6	40	41	42	43	44	45	46	47	6
5	32	33	34	35	36	37	38	39	5
4	24	25	26	27	28	29	30	31	4
3	16	17	18	19	20	21	22	23	3
2	8	9	10	11	12	13	14	15	2
1	0	1	2	3	4	5	6	7	1
	a	b	c	d	e	f	g	h	

If one byte is used to store the distance-to-mate information for each position, this scheme will take up $2 * 64^4$ bytes = 32 Mbytes of data!

2.2 Fixing white king to a1-d1-d4 triangle

An improvement can be made by using the fact that a lot of positions are just mirrors or rotations of one another (with no pawns or castling capabilities left the directions are indistinguishable).
By using a combination of vertical, horizontal and diagonal mirrorings of the board it is always possible to fix the position of a piece within the a1-d1-d4 triangle.[6] White king is (arbitrarily) chosen.

	a	b	c	d	e	f	g	h	
8	-	-	-	-	-	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	-	-	-	-	-	-	-	-	6
5	-	-	-	-	-	-	-	-	5
4	-	-	-	9	-	-	-	-	4
3	-	-	7	8	-	-	-	-	3
2	-	4	5	6	-	-	-	-	2
1	0	1	2	3	-	-	-	-	1
	a	b	c	d	e	f	g	h	

This reduces the size to $2 * 10 * 64^3$ bytes = 5 Mbytes.

2.3 Cutting away trivial part

In this particular endgame the game will be unconditionally drawn if the 2 bishops occupy the same colour. Hence the table can be restricted to contain only the positions with 2 bishops of different colours. By arbitrarily deciding that the first bishop occupy a white square and the second bishop occupy a black one, a space reduction by a factor of 4 can be achieved.

Index of white bishop on white squares:

	a	b	c	d	e	f	g	h	
8	28	-	29	-	30	-	31	-	8
7	-	24	-	25	-	26	-	27	7
6	20	-	21	-	22	-	23	-	6
5	-	16	-	17	-	18	-	19	5
4	12	-	13	-	14	-	15	-	4
3	-	8	-	9	-	10	-	11	3
2	4	-	5	-	6	-	7	-	2
1	-	0	-	1	-	2	-	3	1
	a	b	c	d	e	f	g	h	

Index of white bishop on black squares:

[6] This will be described in further detail in section 5.

	a	b	c	d	e	f	g	h	
8	-	28	-	29	-	30	-	31	8
7	24	-	25	-	26	-	27	-	7
6	-	20	-	21	-	22	-	23	6
5	16	-	17	-	18	-	19	-	5
4	-	12	-	13	-	14	-	15	4
3	8	-	9	-	10	-	11	-	3
2	-	4	-	5	-	6	-	7	2
1	0	-	1	-	2	-	3	-	1
	a	b	c	d	e	f	g	h	

This leaves us with $2 * 10 * 32^2 * 64$ bytes $= 1.25$ Mbytes to compress.

2.4 Data format

In the endgame KBBK the best the lone king can hope for is a draw. Each position in the (`C++`) array

```
unsigned char KBBK[10][32][32][64][2];
```

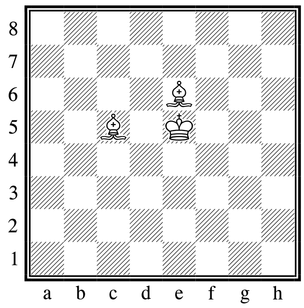
is stored as an unsigned char with the following interpretation:

- A value of 255 represents an illegal position (e.g. a position with at least 2 pieces on the same square).
- A value of 254 represents a drawn position.
- A value $n \in [0 \ldots 38]$ represents a position won by white in n half moves.
- The values $39 \ldots 253$ do not occur.

Assuming that the chess engines probing this table will perform the necessary checks for illegal positions, the table entries with value 255 will never be indexed. We can exploit this by reassigning these entries with values that make the tables more compressible. This is the topic of section 7.

2.4.1 Example

To give an example of how the data looks like, consider this incomplete position, where black king is missing:



The 64 entries below shows the values for this position if black king is inserted at the appropriate position (and it is black to move).

	a	b	c	d	e	f	g	h	
8	16	18	18	18	16	16	16	16	8
7	16	20	20	20	16	16	16	16	7
6	18	254	254	255	255	255	18	18	6
5	18	254	255	255	255	255	18	18	5
4	18	254	254	255	255	255	22	20	4
3	20	22	24	22	24	22	22	20	3
2	20	22	24	24	24	22	22	20	2
1	20	20	22	22	22	22	20	18	1
	a	b	c	d	e	f	g	h	

- The 10 values of 255 (representing an invalid position) are the positions which are too near the white king or on top of the lone bishop.
- The 5 values of 254 (representing draw) are the squares next to lone bishop (from here black king can capture this bishop and enforce a draw).
- Notice that all the remaining numbers (representing mate in n) are even, because n is the number of half moves.

2.5 Compressing the endgame table using gzip

2.5.1 Streaming order

In the first experiment I tested the $5! = 120$ different ways of streaming the 5 dimensional table to a file and then used gzip to compress it. The table below shows the default, the worst and the best way of streaming the table regarding how well gzip performs (for the complete table, see appendix E).

filename	size	ratio
Uncompressed table	1310720	-
Average compression	294058	0.224348
KBBK_01234.dat.gz	327420	0.249802
...	...	...
KBBK_21043.dat.gz	384161	0.293092
...	...	...
KBBK_43120.dat.gz	208731	0.159249
...	...	...

The numbers 0-4 in the above file names represent in which order the index ranges are iterated:

0 Index of white king (size 10)

1 Index of White bishop on white square (size 32)

2 Index of white bishop on black square (size 32)

3 Index of black king (size 64)

4 Side-to-move (size 2)

As an example, the file KBBK_43120.dat was serialised like this:

```
for (side_to_move = 0; side_to_move<2; side_to_move++)
  for (black_king = 0; black_king<64; black_king++)
    for (white_bishop_on_white_square = 0; wbows<32; wbows++)
      for (white_bishop_on_black_square = 0; wbobs<32; wbobs++)
        for (white_king = 0; white_king<10; white_king++)
          output (uchar)KBBK[white_king][wbows][wbobs][black_king][side_to_move];
```

As the table above illustrates, the impact of rearranging the streaming order has a great impact on how well gzip performs. The optimal streaming order results in 29% better compression than choosing a streaming order at random.

2.6 Permutation of board square enumeration

The natural next step to rearrange the table to gain higher compression is to modify the enumeration of the squares on the board. An exhaustive search though the 64! enumerations is clearly infeasible, so we have to use some other approach. Since we have made a restriction of the legal squares for each of the 4 pieces and these restrictions differ, we are searching a separate enumeration for each piece (also it is likely that an optimal ordering for a knight is not optimal for a queen).

The idea I have pursued is the following. We are given a random position (in the KBBK endgame) and we remove the piece for which we are seeking an improved enumeration. Now, if it is likely that the value of the resulting position will be the same if we insert the piece on either square s_1 or square s_2, then these squares are similar. Intuitively gzip's (Lempel-Ziv) dictionary encoding should perform better if similar data are kept close. Hence, the produced enumeration of squares $\langle s_0, \ldots, s_{63} \rangle$ should have a high similarity between s_i and s_{i+1}.

2.6.1 Algorithm

I have implemented an algorithm that performs a greedy bottom up construction of the enumeration of the board squares, based on the above intuition. The pseudo code below is for finding an improved enumeration of the squares for the black king (the only piece which is allowed to occupy all 64 squares). Only slight modifications are needed for the other pieces.

First a fully connected graph is constructed where each vertex i represents the square i in the default enumeration:

$$G = (V, E), V = \{0, \ldots, 63\}, E = \{(i, j) \mid i \neq j\}$$

The informal "similarity" between squares is formalised by assigning a weight function $w : E \to \mathbb{N}$

$$w(d,d') = \|\{(a,b,c,e) \mid KBBK[a][b][c][d][e] = KBBK[a][b][c][d'][e]\}\|$$

$w(d,d')$ is the number of positions in which the outcome would remain the same if you put black king on square d' instead of d. Notice that $w(d,d') = w(d',d)$. The goal will be to maximise the objective function f:

$$f(\langle s_0,\ldots,s_{63}\rangle) = \sum_{i=0}^{62} w(s_i, s_{i+1})$$

Two vertex disjoint chains $c = \langle v_1,\ldots,v_k\rangle$ and $c' = \langle w_1,\ldots,w_l\rangle$ is said to bind with value $\max(\{w(v_1,w_1), w(v_1,w_l), w(v_k,w_1), w(v_k,w_l)\})$. This binding value reflects how well these chains can be merged together according to the similarity measure.

I will refer to a sequence $\langle v_1,\ldots,v_k\rangle$ as a chain. The algorithm described below gradually refines a partial solution given as a set of chains. The final enumeration will consist of these chains glued together at the ends.

Algorithm: ComputeEnumeration
Input: $(G = (V,E),\, w)$
Output: Enumeration of squares (bijective mapping $[0..64[\ \to [0..64[$)
$C = \{\langle v_i\rangle | v_i \in V\}$.
while $\|C\| > 1$ do begin
    Remove c_1 and c_2 from C with largest binding value.
    Construct new chain c by merging c_1 and c_2 "appropriately".
        (I.e. if $w(v_1, v'_1) = \max(\{\ldots\})$ then $c = \langle v_k,\ldots,v_1,v'_1,\ldots,v'_l\rangle$)
    Add c to C
end
return $C[0]$

Now $C = \{\langle p_0, p_1, \ldots, p_{63}\rangle\}$ gives the hopefully improved ordering of the squares (old square index p_i is mapped to index i)

2.6.2 Improving a non-optimal solution

Let's assume that we have interrupted the while loop at a point where we have the following:

$$C = \{\langle 0,1\rangle, \langle 2,3\rangle, \langle 4,5\rangle\}$$

$$w(v_1,v_2) = \begin{cases} 4 & \text{if } (v_1,v_2) \in \{(0,1),(1,0),(2,3),(3,2),(4,5),(5,4)\} \\ 3 & \text{if } (v_1,v_2) \in \{(2,4),(4,2)\} \\ 2 & \text{if } (v_1,v_2) \in \{(0,2),(2,0),(3,5),(5,3)\} \\ 0 & \text{otherwise} \end{cases}$$

$$f(C) = w(0,1) + w(2,3) + w(4,5) = 4+4+4 = 12$$

The next iteration would merge $\langle 2, 3\rangle$ and $\langle 4, 5\rangle$ to $\langle 3, 2, 4, 5\rangle$ and increase the objective function by 3. But from the resulting set

$$C = \{\langle 0, 1\rangle, \langle 3, 2, 4, 5\rangle\},$$
$$f(C) = 4 + (4 + 3 + 4) = 15$$

the final merging would add nothing to the objective function. An optimal solution is

$$C = \{\langle 1, 0, 2, 3, 5, 4\rangle\},$$
$$f(C) = 4 + 2 + 4 + 2 + 4 = 16$$

What went wrong? No solution with vertex 2 and 4 adjacent in the enumeration is optimal. I have made a simple extension of the greedy algorithm described before to overcome this problem. First the greedy algorithm is applied as before. Then it is applied once more for each pair of adjacent vertices in the computed enumeration, but this time with the requirement that these 2 vertices may not be adjacent in the new solution. Of course the algorithm still produces sub-optimal solutions, but tests verify that they often improve.

Algorithm: ComputeEnumeration2
Input: $(G = (V, E), w)$
Output: Enumeration of squares (bijective mapping $[0..64[\to [0..64[$)
$c =$ ComputeEnumeration(G, w)
For each pair of adjacent vertices (v_1, v_2) in c do begin
    let $w' = ((v_1, v_2) \mapsto 0)\ /\ w$
    $c' =$ ComputeEnumeration(G, w')
    if $(w(c') > w(c))$ then $c = c'$
return c

2.6.3 Resulting enumerations

The result of running the ComputeEnumeration2 algorithm on the KBBK endgame is shown below.

New index of white king (not changed):

	a	b	c	d	e	f	g	h	
8	-	-	-	-	-	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	-	-	-	-	-	-	-	-	6
5	-	-	-	-	-	-	-	-	5
4	-	-	-	9	-	-	-	-	4
3	-	-	7	8	-	-	-	-	3
2	-	4	5	6	-	-	-	-	2
1	0	1	2	3	-	-	-	-	1
	a	b	c	d	e	f	g	h	

New index of white bishop on white squares:

	a	b	c	d	e	f	g	h	
8	22	-	9	-	7	-	29	-	8
7	-	23	-	8	-	30	-	14	7
6	4	-	24	-	31	-	13	-	6
5	-	5	-	25	-	12	-	0	5
4	6	-	26	-	18	-	11	-	4
3	-	27	-	17	-	19	-	10	3
2	28	-	16	-	2	-	20	-	2
1	-	15	-	1	-	3	-	21	1
	a	b	c	d	e	f	g	h	

New index of white bishop on black squares:

	a	b	c	d	e	f	g	h	
8	-	29	-	7	-	9	-	22	8
7	14	-	30	-	8	-	23	-	7
6	-	13	-	31	-	24	-	4	6
5	0	-	12	-	25	-	5	-	5
4	-	11	-	18	-	26	-	6	4
3	10	-	19	-	17	-	27	-	3
2	-	20	-	2	-	16	-	28	2
1	21	-	3	-	1	-	15	-	1
	a	b	c	d	e	f	g	h	

New index of black king:

	a	b	c	d	e	f	g	h	
8	63	52	51	50	49	48	47	46	8
7	62	53	22	23	24	25	26	45	7
6	61	54	21	34	33	32	27	44	6
5	60	55	20	35	36	31	28	43	5
4	59	56	19	11	37	30	29	42	4
3	58	57	13	10	38	39	40	41	3
2	17	15	14	16	8	7	6	5	2
1	0	9	12	18	1	2	3	4	1
	a	b	c	d	e	f	g	h	

Notice the interesting patterns in the new enumerations for the bishops and that they are reflections of each other.

2.6.4 Compression

Using gzip again to compress the endgame table, we get an improvement no matter which streaming order we apply (for the complete table, see appendix E:

filename	old size	new size	diff	old ratio	new ratio
Average	294058	255645	38413.3	0.224348	0.195041
KBBK_01234.dat.gz	327420	266378	61042	0.249802	0.20323
...	...	...	...	...	...
KBBK_21043.dat.gz	384161	347045	37116	0.293092	0.264774
...	...	...	...	...	...
KBBK_43120.dat.gz	208731	181353	27378	0.159249	0.138361
...	...	...	...	...	...

We improve the compression ratio from the best streaming order from 0.159249 to 0.138361 — an improvement of 13.12%.

However these numbers are based on an algorithm that works solely for the KBBK endgame (the square enumerations are hardwired in the model). Hence if we should generalize the approaches in this section to arbitrary endgame we would have to include the square enumerations in the description of an endgame hereby making it 64*(number of pieces) bytes larger. But this is rather insignificant.

2.7 Conclusion

I have found the results of this section quite encouraging. Applying the domain specific knowledge reduced the compressed table from the original size of 327420 bytes down to 181353 bytes which is a total improvement of 44.61%.

It is not straight forward to compare the results of this section with the size of the endgame in Nalimov's compressed format. The reason is that our simple way of mapping chess positions resolved in a table with $2*10*32^2*64 = 1310720$ entries, whereas the uncompressed Nalimov format contains $789885 + 873642 = 1663527$ entries (for white to move and black to move).

The Nalimov endgame tables are block compressed using a program called Datacomp. The KBBK endgame is compressed to $249216 + 205549 = 454765$ bytes — 38.89% more than our original gzipped size and 150.76% more than the best gzipped size!

However, the fact that gzip and domain specific knowledge greatly outperforms the size of Nalimov's KBBK endgame is not in itself interesting as we can't use gzip for fast random access. What I find more interesting is that when the domain specific knowledge has such a big impact on how well gzip performs, then hopefully it can also be used to improve the compression achieved by using binary decision diagrams.

3 Generating endgame tables

The basic idea behind generating an endgame table is simple. Start by identifying all positions with a check mate, then all positions with mate in one can be found, and so on. Eventually all lost or won positions will be identified and when this happens the remaining positions must be drawn.

An indexing scheme provides a function for each endgame that maps a position to a value within some fixed index range. The task of making this mapping space efficient (i.e. minimising the index range) while at the same time efficiently computable is the topic of the next section.

The only property of a position that the endgame construction algorithms need to know about, is the side to move. To reflect this, we use a 2-dimensional index range $\{0,1\} \times [0 \dots N[$. The first component gives the side to move (0 for white and 1 for black).

3.1 Notation

It will ease the reading of this section to introduce some naming conventions.

e: A chess endgame (like KQRK).

p: A chess position.

P_e: The set set of all chess positions in the endgame e.

P: The set set of all chess positions.

N_e: An integer representing the size of the index range for each side to move.

m_e: The index function for endgame e:

$$m_e : P_e \mapsto \{0,1\} \times [0 \dots N_e[$$

m_e^{-1}: The inverse index function:

$$m_e^{-1} : \{0,1\} \times [0 \dots N_e[\mapsto P_e$$

It does not need to hold that $m_e^{-1}(m_e(p)) = p$ for all p, but the position $m_e^{-1}(m_e(p))$ should have the same value as p.

s: Side to move. Is either 0 (meaning white to move) or 1 (black to move).

x: A move.

$p \xrightarrow{x} p'$: Position p' is reached by making the legal move x from position p.

`-M`n, `M`n: The distance to mate values lost in n respectively won in n.

3.2 Dependency between endgames

If we wish to solve some specific endgame, say KQKR, then we can't do it in isolation. Optimal moves may bring us to other endgame classes, and then we need to know their game theoretical values too. In this case the endgames KQK and KRK suffice (as either the rook or queen can be captured). Notice that we only need the endgames that can be reached in a single move.

Luckily this dependency is finite. We can define a partial order "$<$" = "$<^\infty$":

$$e_1 <^n e_2 \iff e_1 \leq^n e_2 \wedge e_1 \neq e_2$$

$$e_1 \leq^n e_2 \iff \text{There exists a position } p_1 \in P_{e_1} \text{ than can be reached from a position } p_2 \in P_{e_2} \text{ in at most } n \text{ moves.}$$

Notice that $\forall e$: KK $\leq e$. Only 2 kinds of moves bring a position to another endgame class — namely capture moves and pawn promotions (a move can be both). Let's take the endgame KRKP as an example:

Capture moves: KPK, KRK $<^1$ KRKP

Pawn promotions: KRKN, KRKB, KRKR, KQKR $<^1$ KRKP

Both the above: KNK, KBK, KRK, KQK $<^1$ KRKP

Hence, to construct the KRKP endgame we first need to construct the above 9 endgame tables (KRK is listed twice).

3.3 Level of ambitiousness

As mentioned in the first section, work in the topic of endgame construction has tried to make it feasible to solve even larger endgames.

In this thesis however, I have chosen to restrict the domain to endgames with at most 5 pieces. Compression with fast random access has the main focus. 6 men endgames each take up several GB in Nalimov's compressed format. This makes them rather unsuited for being stored in the memory.

The restriction to ≤ 5 men endgames also simplifies the algorithm. 32 bit addressing space suffices and we do not need to use any memory external optimisation.

However, even a 5 men endgame has a considerable size ranging from 40 MB to 700 MB (or millions of positions). Therefore we have to keep the memory usage at a minimum — even the use of a single pointer per position will make it infeasible.

There are 2 ways of implementing the retrograde analysis. The "forward" approach uses a normal move generator, while the "backward" approach requires the generation of all moves leading to a position. When I started on this thesis, I had already implemented a chess program. Hence the normal move generator was ready to use, so I started implementing the "forward" algorithm.

3.4 "Forward" algorithm

The forward algorithm works by continuously iterating through all positions. For each position, where the game theoretical value has not yet been decided, it does a ply[7] 1 search to check whether sufficient information is available to determine its value.

[7]A ply is a move for only one of the players (a half move)

For each iteration, all positions that are won or lost in a specific number of moves are found. This number increases by one for each iteration. When no more progress can be made, the remaining positions can be judged drawn, because any winning strategy would have been identified by then.

Some care has to be taken though. Just because no position was identified as a win or loss in n moves, it does not necessarily mean that no position is won or lost in m*more* than n moves. An example is the KBPKNN endgame. It contains 6 positions with white to move that are won for white in respectively 141, 148, 164, 195, 206 and 221 moves. These 6 positions are the only ones that are won or lost in more than 140 moves! The problem occurs when the best move in a position is a capture or pawn promotion that brings this position into another endgame class. This endgame class might contain positions that are won or lost in more moves than is the case for the endgame class under construction.

```
Algorithm: ForwardGenerate
// First identify all check mate, stale mate and illegal positions:
For each (s,i) ∈ {0,1} × [0..N_e[ do begin
    p = m_e^{-1}(s,i);
    If legal_position(p, s) then begin
        If check_mate(p, s) then t_e[(s,i)] = -M0;
        Else if stale_mate(p, s) then t_e[(s,i)] = drawn;
        Else t_e[(s,i)] = unknown;
    End else t_e[(s,i)] = illegal_position;
End;

// Now identify all wins/loses incrementally starting from depth 1:
bool more_to_do = true;
int optimal_win = 1;
While more_to_do do begin
    more_to_do = false;
    For each s = 0 to 1 do begin
        // I: All wins/losses in strictly less than optimal_win
        // have been identified identified for the player 1 − s.
        For each i =0 to N_e − 1 such that t_e[(s,i)] = unknown do begin
            Let Q = {t_e'[m_e'(p')] | ∃x,p',e' : (m_e^{-1}(s,i) --x--> p') ∧ (p' ∈ e')}
                (Q is the set of values relative to the opponent, after each
                of the legal moves have been made. As this endgame is under
                construction, some of these values may be unknown).
            Modify the values in Q so that they become relative to the current
                player. That is; a win in n becomes a loss in n, and a loss in
                n becomes a win in n + 1. unknown and drawn values remains.
            Let v be the "best" value in Q (defined the obvious way).
            If unknown ∉ Q then begin
                more_to_do = true;
                t_e[(s,i)] = v;
            End else if v is a win then begin
                more_to_do = true; // Solves problem with e.g. the KBPKNN endgame.
                If v is a win in exactly optimal_win moves then
                    t_e[(s,i)] = v;
            End;
        End;
    End;
    optimal_win = optimal_win+1;
End;

// The remaining positions must be drawn:
For each (s,i) ∈ {0,1} × [0..N_e[  with  t_e[(s,i)] = unknown  do t_e[(s,i)] = drawn;
Return t_e;
```

That the invariant is satisfied can be realised by looking at the diagram below:

optimal_win = 1:
$s = 0$ (white to move) : B(`-M0`) $\Rightarrow$ W(`M1`) $s = 1$ (black to move) : W(`-M0`) $\Rightarrow$ B(`M1`), W(`M1`) $\Rightarrow$ B(`-M1`),
optimal_win = 2:
$s = 0$ (white to move) : B(`-M1`) $\Rightarrow$ W(`M2`), B(`M1`) $\Rightarrow$ W(`-M1`) $s = 1$ (black to move) : W(`-M1`) $\Rightarrow$ B(`M2`), W(`M2`) $\Rightarrow$ B(`-M2`),
...

B(`-M1`) $\Rightarrow$ W(`M2`) should be read: All remaining positions, which has white to move and which white has won in 2, are determined using the positions, in which it is black to move and in which black has lost in 1.

3.4.1 Running time

The running time of the `ForwardGenerate` algorithm is dependent on the highest distance to a mate, as this is the number of times the outer while loop is iterated.

The number of legal moves in any legal position is less than 1000.[8] Hence, the work needed to determine Q can be viewed as a constant. The theoretical running time is

$$TIME(\texttt{ForwardGenerate}) \in O(max\_mate\_depth * N_e)$$

In practice each iteration of the outer loop does not take time proportional with $2N_e$, but with the number of entries in t_e that still have the `unknown` value.

This is reflected in the experimental results shown below. The KBBKN endgame takes exceedingly long time to generate because more than half of the positions are drawn. Many of these positions will be examined over and over, because they will retain their `unknown` value until the last few lines of the algorithm. Also, this endgame has a quite long longest mate of 78.

Tested on a 3 GHz Pentium 4 with 1 GB of memory

Endgames constructed	Number of positions	time	time/pos.
All 2 and 3 men	411382	0m19.564s	47.6μs
All K+2 vs K	78617952	64m59.912s	49.6μs
All K+1 vs K+1	78822912	179m5.106s	136.3μs
KQQQK	38497536	5m24.448s	8.4μs
KBBKN	119218176	832m41.147s	419.1μs

[8]The piece with most movability is the queen. From the centre it can reach 6+7+7+7 = 27 squares. With at most 32 pieces this gives a trivial upper bound of 27*32 = 864 < 1000.

3.5 "Backward" algorithm

We saw that the forward algorithm wasted time by examining the same positions over and over. Its weakness is that the algorithm does not know whether sufficient information is available to determine the value, so it has to try. The backward algorithm does not suffer from this weakness as each move is being examined at most twice.

First all check mated, stale mated and illegal positions are identified. All other positions have as value the value of the best move examined so far — initially this is defined to -M0 (lost).

To be able to determine when all moves in a position have been checked, we need to count the number of untried moves for each position. An unsigned char suffices for this purpose (but still, this doubles the memory requirements).

The algorithm now performs a 1 ply search from every position. Each move, that brings the position to a simpler endgame, updates the value of the position accordingly (this can be done now, as the values of all positions in the simpler endgames are already known). For each other move, the counter attached to this position is simply incremented by one, meaning that we are not yet able to determine the implications of this move.

The backward part of the algorithm is done next. Like in the forward algorithm we iterate through all positions for each mate depth. The good thing is that the real work is only done for few of the positions — namely those which are won or lost in a specific number of moves. For each of these positions we undo every move that keeps us within the current endgame. We know the values of the moves that we take back, so they can be used to update the values of the positions that we reach.

The algorithm makes the intuitively correct assumption, that the number of moves that can be made in a position is equal to the number of ways this position can be reached by taking back a move from other positions. We will see in the next section that there are cases where this does not hold.

A technical detail about this algorithm is that we only need a simplified retro move generator as we are not interested in captures or pawn promotions. This removes some of the many special cases, but even in this form it is more complicated to implement than a normal move generator.

Algorithm: `BackwardGenerate`

// Invariant: K gives the highest n for which t_e contains the value Mn or -Mn.
$K = 0$;
unsigned char $c[2][N_e]$;
Initialise all entries in c to 0;
// Identify all check mate, stale mate and illegal positions. All
// other positions are initialised with worst possible value, -M0.
For each $(s, i) \in \{0, 1\} \times [0..N_e[$ do begin
  Let $p = m_e^{-1}(s, i)$;
  If legal_position(p, s) then begin
    If check_mate(p, s) then $t_e[(s, i)] =$ `-M0`;
    Else if stale_mate(p, s) then $t_e[(s, i)] =$ `drawn`;
    Else begin
      Let $Q = \{t_{e'}[m_{e'}(p')] \mid \exists x, p', e' : (e' <^1 e) \wedge (m_e^{-1}(s, i) \xrightarrow{x} p') \wedge (p' \in e')\}$
        (Q is the set of values relative to the opponent, after each
        of the legal moves reaching a simpler endgame have been made.
      Modify the values in Q so that they become relative to the current player.
      $t_e[(s, i)] =$ best value in Q
      If $t_e[(s, i)]$ is Mn or -Mn for some $n > K$ then $K = n$

      For each x such that $(m_e^{-1}(s, i) \xrightarrow{x} p') \wedge (p' \in e)$ do
        $c[(s, i)] \mathrel{+}= 1$; // Initialise c to count the number of moves
    End;
  End else $t_e[(s, i)] =$ `illegal_position`;
End;

// Now we are done using the simpler endgames
$n = 0$;// It is the positions that are won or lost in n that are begin inspected.
While $n \le K$ do begin
  // I: All wins/losses in at most n moves have been identified.
  For each $s = 0$ to 1 do for each i =0 to $N_e - 1$ do begin
    If ($t_e[(s, i)] \ne$ `illegal_position`) and
      (($t_e[(s, i)] =$ Mn) or (($t_e[(s, i)] =$ -Mn) and ($c[(s, i)] = 0$))) then begin
      For each x such that $\exists p' : (p' \xrightarrow{x} m_e^{-1}(s, i))$ do begin
        $c[m_e(p')] \mathrel{-}= 1$;
        Let v be the value of p' relative to the opponent, of making move x.
        If v is better than $t_e[m_e(p')]$ then $t_e[m_e(p')] = v$
        If ($t_e[m_e(p')] =$ Mn) or ($c[m_e(p')] = 0$) then begin
          // The value of $t_e[m_e(p')]$ is final.
          If $t_e[(s, i)]$ is Mn or -Mn for some $n > K$ then $K = n$
        End;
      End;
    End;
  End;
End;

For each $(s, i) \in \{0, 1\} \times [0..N_e[$ where $c[(s, i)] > 0$ and $t_e[m_e^{-1}(s, i)]$ is a loss do
  $t_e[m_e^{-1}(s, i)] =$ `drawn`

3.6 Comparison and conclusion

The running time of the backward algorithm was tested on the same endgames as the forward algorithm.

Tested on a 3 GHz Pentium 4 with 1 GB of memory

Endgames constructed	Number of positions	Forward algorithm time	Forward algorithm time/pos.	Backward algorithm time	Backward algorithm time/pos.
All 2 and 3 men	411382	0m19.564s	47.6μs	0m3.572s	8.6μs
All K+2 vs K	78617952	64m59.912s	49.6μs	17m37.094s	13.4μs
All K+1 vs K+1	78822912	179m5.106s	136.3μs	11m28.075s	8.7μs
KQQQK	38497536	5m24.448s	8.4μs	10m30.698s	16.4μs
KBBKN	119218176	832m41.147s	419.1μs	20m25.832s	10.3μs

The results are as expected. The backward algorithm achieves good stable performance. The variance of a factor of 2 in time per position can be explained by the fact that the algorithm only takes back moves for positions that are eventually judged won or lost. Hence, endgames with many drawn positions require less work. This is in contrast with the forward algorithm. It is interesting that the forward algorithm, despite a worse worst case run time, is fastest at the construction of the KQQQK. Of course the reason is that very few iterations are needed, as the longest win is in 3 moves.

The endgame tables used in the checkers program *Chinook* were created by using a combination of the forward and backward algorithm [19]. On the first few iterations through the positions, the forward approach was able to find the value of by far the most of the positions. For the remaining positions, the backward approach was used. This combination united the fast construction by the backward approach with the low memory requirement for the forward approach (no counters were needed for the large fraction of positions that were solved using the forward approach).

A similar approach is not worth the trouble in chess. If you have time to wait for the forward algorithm, implement this. Otherwise go for the more challenging backward algorithm (with most of the challenge being concentrated in the implementation of the retro move generator).

3.7 Tables generated

This is to give some idea of the size of the tables. Notice that the construction of the largest 5 men endgames have failed. However, it is “just” a matter of having a computer with 2 GB of memory rather than 1 GB.

Endgame	Size in bytes (wtm+btm)	Nalimov’s uncompressed size in bytes (wtm+btm)
KK	462	-
KNK	29568 + 29568	26282 + 28644
KBK	29568 + 29568	27243 + 28644
KRK	29568 + 29568	27030 + 28644
KQK	29568 + 29568	25629 + 28644
KPK	86688 + 86688	81664 + 84012
KBBK	931392 + 931392	789885 + 873642
KRBK	1892352 + 1892352	1594560 + 1747284
KQKQ	1892352	1563735 (+ 1563735)
KNKP	5548032 + 5548032	4931904 + 4981504
...	...	...
KNNNK	19248768 + 19248768	13486227 + 17472840
KRRKQ	59609088 + 59609088	46658340 + 46912050
KRNKQ	121110528 + 121110528	92308740 + 93824100
KQQKP	174763008 + 174763008	131170128 + 149445120
...	...	...
KRPKQ (failed)	355074048 + 355074048	286777440 + 288610560

3.8 Verification

The endgames have been verified against Nalimov’s endgame tables. A position for position match have been performed for a few selected ones (such as KPKP and KPPK), the rest only by their signature (number of positions with mate in n etc.).

4 Indexing scheme

The problem of finding space efficient, fast computable mappings from chess endgames to index ranges has already been studied by several people. Again I refer to the article *Space-efficient indexing of chess endgame tables* [21]. It is not the aim of this thesis to improve this.

However, section 2 showed that significant improvements can easily be obtained compared to the trivial mapping that maps to $[0..2 * 64^n[$ for an n-piece endgame.

I have chosen to implement only the techniques that incur the highest reduction of size and whose impact at the same time remains when the table is compressed using OBDDs.

4.1 Using a reduced position description

In the design of the index scheme, which features of the position should be taken into account. To achieve correct play, a chess program needs all the following information about a position:

Pieces: Which pieces are left and which squares do they occupy?

Side-to-move: Is it white or black to move?

En passant: Is an en passant move possible, and if so; where?

Castling: Have the kings lost their right to castle (short/long)?

Position history: Needed to examine if a position has occurred twice before (3. repetition rule).

Non-progressing moves: It is necessary to keep track of how far back the last pawn move or capturing move was performed to obey the 50 move rule.

Most indexing schemes ignores the last 3 kinds of information. The reasons are:

Castling: It is unlikely that any serious chess game has ever been played, where a player was able to castle in an endgame position.

Position history: From no won or lost position will the optimal line of play reach the same position twice. This is quite obvious as the values of the positions would be $\ldots$-Mn, Mn, -M$(n-1)$, $\ldots$, -M0.

Non-progressing moves: The 50-move rule was originally designed to enforce a draw in positions where nobody wanted to take the initiative. This was before the invention of endgame tables, and before having identified any position to be a win in more than 50 moves. Hence, the effect of this rule, making some otherwise won positions drawn, is actually undesired and that's probably the main reason why it is being ignored.

In contrast, the indexing scheme used in this thesis includes the mapping of positions with castling. Positions with either castling capabilities or en passant are encoded as otherwise broken positions and do not add to the size of the representation of the endgame.

4.2 Broken positions

An entry in an endgame table that is not mapped by any legal position is called a broken position. Taking the KBBK endgame from section 2 as an example, 2 kinds of broken positions appear. Those in which the side-to-move can capture the opponent king and those in which 2 pieces are placed on the same square.

To design a space efficient indexing strategy is equivalent to eliminating the different kinds of broken positions and to use canonical representations. The use of canonical representations (such as the restriction of white king to the a1-d1-d4 triangle in section 2) has the highest impact.

The use of OBDDs with heuristic mappings of don't cares (broken positions) as our mean of compression has the effect that a reduction of index space by means of eliminating broken positions do not necessarily result in a smaller OBDD.

4.3 Indexing method

The trivial mapping from an n piece endgame to a table of size $2 * 64^n$ was introduced in section 2. The improvements used in this thesis are listed below. They are all part of Nalimov's and Heinz index schemes as well, while Edwards and Thompson's schemes are simpler and less efficient. Together they define what I will refer to as basic indexing.

Enumeration of legal king positions: First consider the group of pawnless endgames. Enumerating the legal king positions extends the idea of restricting white king to the a1-d1-d4 triangle and contains 2 improvements. If white king is on the a1-d1 half diagonal, then by symmetry black king can furthermore be restricted to the a1-h1-h8 triangle. Also, many placements of the 2 kings result in broken positions — in any legal position the two kings must be separated by at least one square (it is not allowed to move the king to a square where it can be captured). All initial 64*64 placements of kings reduce to 462 (with only white king restricted to the a1-d1-d4 rectangle, the number would be 640).

The above optimisations depend only on the position of the 2 kings, so it is obvious to represent it as a table. Each entry must also contain information about whether horizontal, vertical and/or diagonal reflection is used to map the kings to their canonical positions, because the remaining pieces must be transformed similarly. Below -1 represent an illegal placement of the 2 kings while T is the set of transformations of the board (i.e. reflections and rotations).

$$K_{-p}\text{: } [0..64*64[\rightarrow [-1..462[\text{ x } T$$

For endgames with pawns we lose 2 of the 3 degrees of freedom because the movements of pawns only have vertical symmetry. The same trick is still performed but the size of the enumeration increases almost 4-fold to 1806.

$$K_{+p}\text{: [0..64*64[} \rightarrow \text{[-1..1806[x } T$$

Restrict pawns to rank 2-7: Pawns can't occur on rank 1 or 8. Hence the number of squares is reduced to 48.

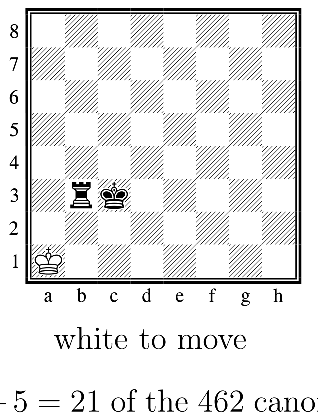
Saving of factor k! with k like men: In an endgame like KRRK the position remains the same if we swap the 2 rooks. Hence we can arbitrarily decide that the position of the first rook must be (strictly) less than the position of the second rook (according to some fixed enumeration of the squares). This reduces the number of ways of placing the rooks from 64^2 to $\sum_{i=0}^{i=63}(i)$ $= \frac{64*63}{2}$. Again this is implemented by a table.

Only white-to-move for endgames with equal pieces: Only store entries for white-to-move. If we wish to find the value of a position with black-to-move we simply swap the white pieces with the black pieces. Obviously a factor of 2 is achieved this way. Using this trick requires a minor modification of the algorithms for constructing endgames.

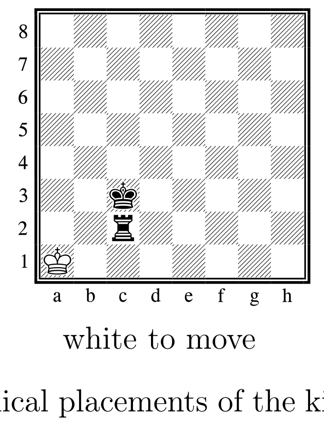
In the end of this section it is demonstrated how the above improvements can be combined into an indexing scheme.

4.4 Further symmetry

Even though we exploit the symmetry in the king enumerations, there are still positions that are symmetrical, but are given different indexes. Example (both positions satisfy the restrictions as white king is in the a1-d1-d4 triangle and black king is in the a1-h1-h8 triangle):



white to move



white to move

In the $6+5+5+5 = 21$ of the 462 canonical placements of the kings, both kings are placed on the same diagonal and this symmetry is unused. Furthermore the reduction caused by exploiting this symmetry would be a reduction in the number of legal positions in the table — not just a reduction of the number of broken positions. We will later see, that when it comes to compressing the table using an OBDD this is what really matters — the broken positions wont contribute much to the size.

However, it is much simpler just to let the king positions uniquely decide the transformation using a lookup table. Hence I have not tried to exploit the extra symmetry. Also, this is the approach taken by Nalimov. The verification of endgame tables by signatures described in section 3.8 will only be possible by making the same choice.

4.5 Unique representation

The index scheme presented so far has this property, but let's review the last section for a moment to see if the algorithms really need it. Assume we have $p, p' \in e$ and $p' \xrightarrow{x} p$, where p' doesn't have a unique representation, i.e. $p' = m_e^{-1}(s, i') = m_e^{-1}(s, i'')$:

$$\begin{tikzcd} (s', i') \arrow[r, squiggly, no head] \arrow[dr, "x \circ m_e^{-1}"'] & (s'', i'') \arrow[d, "x \circ m_e^{-1}"] \\ & p \end{tikzcd}$$

The forward algorithm would just find the value for both representations of p' and hence still work. Also, if p had more than one representation, then they would be given the same value and it wouldn't matter which one was used.

The backward algorithm however gets into trouble. When it takes back the move $p' \xrightarrow{x} p$ and perhaps updates the value of p', it does so for only one of its representations. From this point on, the value of the other representations of p' will be wrong.

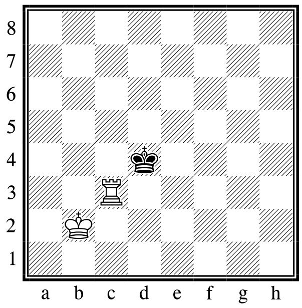
One might ask if this really matters as the representation produced by $m_e(p')$ do have the right value. The real problem emerges when a representation of p', having a wrong value, is used to infer the values of other positions.

Hence unless we modify the backward algorithm to take these exceptions into account, we must insist that our index scheme provides a unique representation for each position.

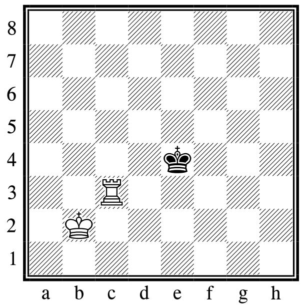
4.6 Canonical representation

Remember that in a pawnless endgame, we only allow 462 possible placements of the kings. We can use any of the 8 combinations of reflections and rotations to transform any position to this canonical form. However, in some cases the 8 transformations of a canonical position only produce 4 different ones. As an example, the position shown below to the left is the canonical representation of 4 different positions, while the one to the right represents 8 different positions. Notice that this is the case whether or not we had used the further symmetry described in the last subsection and hence is an inherent problem by allowing diagonal reflection. In the endgames with pawns where only vertical reflection is allowed, there is no problem.

Black to move



White to move



The problem is illustrated by trying the moves Kd5 and Ke4 from the left position. After the move Kd5, we have to do a reflection of the board in the a1-h8 diagonal to bring it back on canonical form. This means that both moves result in the right position! By a counting argument this makes sense. If we consider all transformations of the above boards, then we have $4 \cdot 2$ moves in the left position and $8 \cdot 1$ in the right position.

Lets see how this influences the algorithms from the last section. The forward algorithm is not affected. In the backward algorithm we count the number of moves for which the outcome is still uncertain. In view of the above discussion we have to make some adjustments to this part of the code.

The canonical representation has white king restricted to the a1-d1-d4 triangle. Furthermore if white king is on the a1-d4 half diagonal, then black king is restricted to the a1-h1-h8 triangle. But if both kings are placed on the same diagonal line then no further symmetry is used. Hence the canonical representations with both kings on the same diagonal only represent 4 different positions while the others represent 8. Let $\sharp p$ denote this number.

Line 19: Replace the line

$$c[(s,i)] \mathrel{+}= 1;$$

with

$$c[(s,i)] \mathrel{+}= (\sharp m_e^{-1}(s,i) > \sharp p')\ ?\ 2 : 1;$$

Line 31: If $\sharp p' = 4$ then repeat the lines 32-38 for the position p' reflected in the a1-d8 diagonal.

4.7 En passant

In [20] two different approaches for handling en passants is presented. The first approach is based on encoding en passant positions as otherwise broken positions. The other possibility is to add a separate index range for the en passant positions.

Nalimov has chosen the last strategy, but probably only because he had to. In his compact index scheme there is no entry that corresponds to a positions with 2 pieces on the same square. It is this kinds of broken positions the encoding is based on.

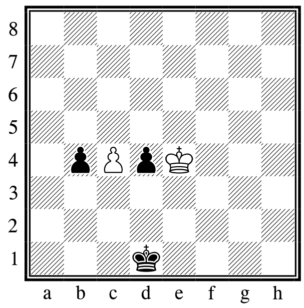
With my index scheme I have the possibility of applying both approaches. I have chosen the first one based on two reasons. First, the table remains the same size, and secondly, the implementation is easier.

The mapping between en passant positions and positions with 2 pawns on the same square has been designed such that it has the nice property that it commutes with vertical reflection.

In a position with an en passant we have at least one pawn of each colour — one pawn that just moved 2 forward and one pawn currently standing next to it ready to exploit the en passant. It is these two pawns that we reassign to the same square and thereby encodes the en passant.

When there are 2 pawns that can capture the en passant pawn, we get 2 different encodings of the same position, as either of them can be used together with the en passant pawn in the encoding. To keep the representation unique we choose the pawn closest to a d or e file. Example:

Whites last move was c4 $\Rightarrow$ en passant on c3:
Use black pawn closest to d or e file, i.e. the d pawn, to encode the en passant.



The encoding and decoding of an en passant is implemented efficiently using the conversion tables shown below. Let's demonstrate how to encode and decode the en passant from the example above. By indexing the left table below with the columns c and d we get the position c3. Hence, we move the 2 pawns to this square. Going the other way we first detect that the 2 pawns occupy the same square. Then we index the next two tables with their identical position c3 and retrieve their original positions.

Notice that no other piece can occupy the square we use for the encoding — it is either the square of the en passant pawn or the square it has just passed. This property is not essential, but it makes the implementation simpler.

Encoding tables telling on which square to place the pawns:

White pawn just moved 2 forward:

White pawn column ↓	Black pawn column: a	b	c	d	e	f	g	h
a	-	a3	-	-	-	-	-	-
b	b4	-	b3	-	-	-	-	-
c	-	c4	-	c3	-	-	-	-
d	-	-	d4	-	d3	-	-	-
e	-	-	-	e3	-	e4	-	-
f	-	-	-	-	f3	-	f4	-
g	-	-	-	-	-	g3	-	g4
h	-	-	-	-	-	-	h3	-

Black pawn just moved 2 forward:

Black pawn column ↓	White pawn column: a	b	c	d	e	f	g	h
a	-	a6	-	-	-	-	-	-
b	b5	-	b6	-	-	-	-	-
c	-	c5	-	c6	-	-	-	-
d	-	-	d5	-	d6	-	-	-
e	-	-	-	e6	-	e5	-	-
f	-	-	-	-	f6	-	f5	-
g	-	-	-	-	-	g6	-	g5
h	-	-	-	-	-	-	h6	-

Decoding tables giving the original positions of the pawns:

En passant pawn:

	a	b	c	d	e	f	g	h	
8	-	-	-	-	-	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	a5	b5	c5	d5	e5	f5	g5	h5	6
5	-	b5	c5	d5	e5	f5	g5	-	5
4	-	b4	c4	d4	e4	f4	g4	-	4
3	a4	b4	c4	d4	e4	f4	g4	h4	3
2	-	-	-	-	-	-	-	-	2
1	-	-	-	-	-	-	-	-	1
	a	b	c	d	e	f	g	h	

Capturer pawn:

	a	b	c	d	e	f	g	h	
8	-	-	-	-	-	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	b5	c5	d5	e5	d5	e5	f5	g5	6
5	-	a5	b5	c5	f5	g5	h5	-	5
4	-	a4	b4	c4	f4	g4	h4	-	4
3	b4	c4	d4	e4	d4	e4	f4	g4	3
2	-	-	-	-	-	-	-	-	2
1	-	-	-	-	-	-	-	-	1
	a	b	c	d	e	f	g	h	

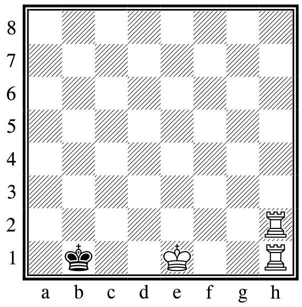
With en passant a new kind of broken positions occur — we do not allow a piece to occupy one of the two squares the en passant pawn should just have passed.

4.8 Castling

As mentioned earlier, only the program RETRO, made by Dekker in 1989, have included positions with castling capabilities in its endgames. The reason is that it is only theoretically possible to reach an endgame position with castling rights — in practise it will never occur.

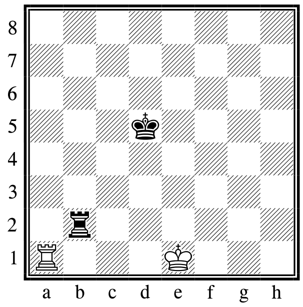
However I've found an encoding of castling rights similar to and combinable with the en passant. To make the tables complete I've implemented this. As a further argument, the ability to castle actually decides the outcome of a few positions like the ones below:

White can castle short



white to move

White can castle long



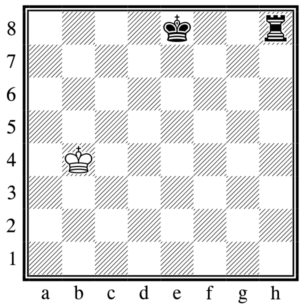
white to move

The encoding goes like this: If white can castle short, then reassign the h file rook to the square of whites king. If white can castle long, then reassign the a file rook to the square of blacks king. Similar for black.

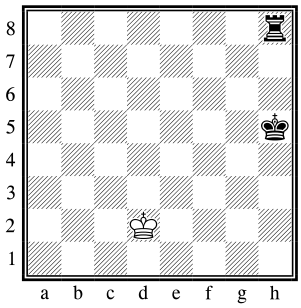
At a first glance the decoding looks just as easy, but the restriction of white king to either the a1-d1-d4 triangle or the a1-d8 rectangle complicates things — with castling rights no transformation is allowed.

As an example let's consider how the position below is transformed to place white king in the a1-d1-d4 triangle:

Black can castle short



Before transformation



After transformation

In the encoding of the castling, black rook will be moved to the square of the black king. It is irrelevant whether this happens before or after the restriction of white king to the a1-d1-d4 triangle. When we are going the other way, we interpret a black rook on black king as a short castle for black, but the problem is that black king is not placed on its original position. We solve this issue by transforming the board back to its original orientation. This transformation is uniquely determined by the position of the black king.

Before we continue it is necessary to make the concept of reflection more precise.

4.8.1 Reflection

When we are restricting the white king to either the a1-d1-d4 triangle or to the a1-d8 rectangle we need to be able to perform 3 kinds of reflections of the board to achieve this.

1 Vertical reflection (a1 $\leftrightarrow$ h1, a8 $\leftrightarrow$ h8)

2 Horizontal reflection (a1 $\leftrightarrow$ a8, h1 $\leftrightarrow$ h8)

4 Diagonal reflection along the a1-h8 diagonal

It is not an error that the diagonal reflection is labelled 4, because now we can determine an arbitrary transformation of the board by a sum of these labels. They will be represented by t_n for $n \in [0..7]$. If more than one reflection are used then they are applied in the order given above. An application of a transformation t_n on a position p will be written $t_n(p)$.

Note that in general $t_n(t_n(p)) \neq p$, but we do have that $t_n(t_n^{-1}(p)) = t_n^{-1}(t_n(p)) = p$, where

$$t_n^{-1} = \begin{cases} t_6 & \text{if n=5} \\ t_5 & \text{if n=6} \\ t_n & \text{otherwise} \end{cases}$$

4.8.2 Decoding positions with castling

The decoding of a position with castling is a 2 step process. First the rook should be returned to its original position. The 2 tables below determine its original position. They should be indexed with the position of the king owning the rook. If the entry is undefined, then no transformation can bring the king back to its original position and we have a broken position.

Rook placed on own king:

	a	b	c	d	e	f	g	h	
8	-	-	-	a8	h8	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	-	-	-	-	-	-	-	-	6
5	a8	-	-	-	-	-	-	h8	5
4	a1	-	-	-	-	-	-	h1	4
3	-	-	-	-	-	-	-	-	3
2	-	-	-	-	-	-	-	-	2
1	-	-	-	a1	h1	-	-	-	1
	a	b	c	d	e	f	g	h	

Rook placed on opponent king:

	a	b	c	d	e	f	g	h	
8	-	-	-	h8	a8	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	-	-	-	-	-	-	-	-	6
5	a1	-	-	-	-	-	-	h1	5
4	a8	-	-	-	-	-	-	h8	4
3	-	-	-	-	-	-	-	-	3
2	-	-	-	-	-	-	-	-	2
1	-	-	-	h1	a1	-	-	-	1
	a	b	c	d	e	f	g	h	

When the positions of all pieces have been determined it is time to apply the transformation to bring back the king to its original position. Once again the tables below should be indexed with the position of the king having the right to castle.

Transformation based on white king:

	a	b	c	d	e	f	g	h	
8	-	-	-	3	2	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	-	-	-	-	-	-	-	-	6
5	4	-	-	-	-	-	-	5	5
4	6	-	-	-	-	-	-	7	4
3	-	-	-	-	-	-	-	-	3
2	-	-	-	-	-	-	-	-	2
1	-	-	-	1	0	-	-	-	1
	a	b	c	d	e	f	g	h	

Transformation based on black king:

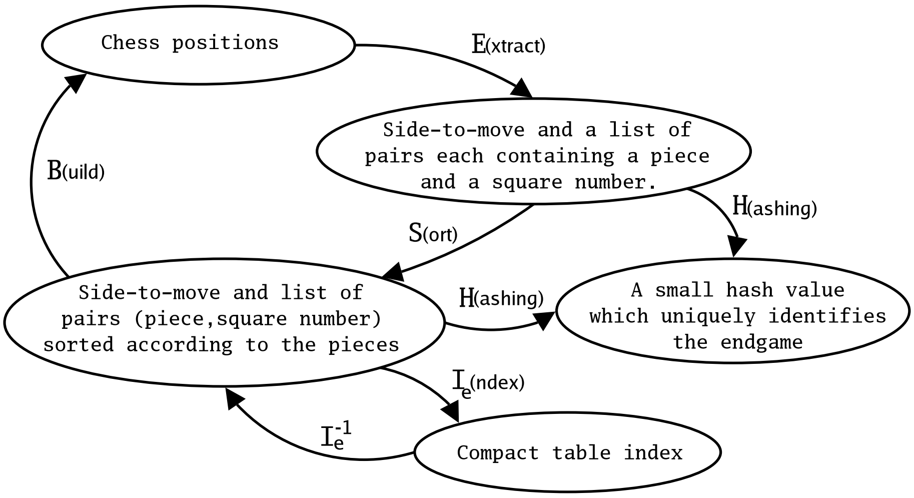
	a	b	c	d	e	f	g	h	
8	-	-	-	1	0	-	-	-	8
7	-	-	-	-	-	-	-	-	7
6	-	-	-	-	-	-	-	-	6
5	5	-	-	-	-	-	-	4	5
4	7	-	-	-	-	-	-	6	4
3	-	-	-	-	-	-	-	-	3
2	-	-	-	-	-	-	-	-	2
1	-	-	-	3	2	-	-	-	1
	a	b	c	d	e	f	g	h	

As an example, if white king is placed on d8, then the number 3 represents a vertical reflection mapping it to e8 followed by a horizontal reflection mapping it to e1.

If it's an endgame with pawns and the transformation is not 0 or 1 then we have a broken position. This is also the case if both black and white have castling capabilities, but they require different transformations of the board.

4.9 Implementation

The figure below shows how the functions m_e and m_e^{-1} have been implemented ($m_e = I_e \circ S \circ E$, $m_e^{-1} = B \circ E^{-1}$).



B This function restores the board position given the side-to-move and the list of pieces. The positions of the pieces are decoded if they contain information about en passant or castling.

E This function reduces the representation of a chess position to the side-to-move and a list of the pieces and their square numbers. Any castling capability or en passant has been encoded in the positions of the pieces.

H Given a set of pieces in a position we need an efficient way of determining which endgame this position belong to. The approach chosen here is to construct a collision free hash table, where each entry contains the necessary information.

Below is shown the hash function h. This is clearly collision free when we restrict ourself to endgames with at most n pieces.

$$\begin{aligned} h(\{p_i\}_{0\leq i<m}) &= \textstyle\sum_{i=0}^{m-1} v(p_i), \text{ where} \\ v(\texttt{white pawn}) &= (n+1)^0, \\ v(\texttt{white knight}) &= (n+1)^1, \\ \ldots &= \ldots \\ v(\texttt{black king}) &= (n+1)^{11} \end{aligned}$$

But clearly we are interested in minimising the size of the hash table. It is an easy observation to see that we can remove the 2 kings from our set of pieces as they contribute with no information (a legal chess position always has exactly one white and one black king). The maximal index computed will then be $(n-2)\cdot((n-2)+1)^{(11-2)}$ — ie. 39366 for n=4 and 3145728 for n=5.

However we can do even better. A simple greedy algorithm has been used to find the values below:

max 4-men endgame
$h(\{p_i\}) \in [0..2\cdot 96+1[$

	white	black
pawn	1	30
knight	3	44
bishop	7	65
rook	12	80
queen	20	96
king	0	0

max 5-men endgame
$h(\{p_i\}) \in [0..3\cdot 744+1[$

	white	black
pawn	1	124
knight	4	218
bishop	13	375
rook	32	572
queen	71	744
king	0	0

I_e This is not a single function but rather a class of functions — one for each kind of endgame. If we let X and Y be substitutable for a knight, bishop, rook or queen, then we need a function for each of the following kinds of endgames:

KK, KXK, KPK, KXXK, KXYK, KXKX, KXKY, KPPK, KXPK, KPKP, KXKP, ...

I_e^{-1} As in I_e there is a function for each kind of endgame.

S The list of pieces is sorted so that they appear in a canonical form where they are sorted in the order K_w, Q_w, R_w, B_w, N_w, P_w, K_b, Q_b, R_b, B_b, N_b,

P_b, where white is the strong side. If black is the strong side (e.g. black king and queen and white king) then the colours of the pieces have to be swapped. The effect of swapping the colours of the pieces is undone by also changing side-to-move. If it is an endgame with pawns, it is furthermore necessary to mirror the position because of the asymmetric movements of pawns. If it is a symmetric endgame and the position has black-to-move then the colours will also have to be swapped, because only the positions with white-to-move are stored.

4.10 Specification of index functions

The transformations needed to restrict the white king to the a1-d1-d4 can be tabulated. Along the diagonal positions the transformation is not unique.

T:

	a	b	c	d	e	f	g	h	
8	2 (6)	2	2	2	3	3	3	3 (7)	8
7	6	2 (6)	2	2	3	3	3 (7)	7	7
6	6	6	2 (6)	2	3	3 (7)	7	7	6
5	6	6	6	2 (6)	3 (7)	7	7	7	5
4	4	4	4	0 (4)	1 (5)	5	5	5	4
3	4	4	0 (4)	0	1	1 (5)	5	5	3
2	4	0 (4)	0	0	1	1	1 (5)	5	2
1	0 (4)	0	0	0	1	1	1	1 (5)	1
	a	b	c	d	e	f	g	h	

4.10.1 King enumerations

When white king occupies a diagonal square, the square of the black king determines which of the 2 transformations should be used. First the positions of both kings are transformed according to T (In the "x (y)" entries, x is used). Then black king is restricted to the a1-h1-h8 triangle — i.e. if the position of black king is not in the a1-h1-h8 triangle, then a further diagonal reflection is added (hence in that case the y of "x (y)" will be used).

Let's for a moment assume that we are dealing with pawnless endgames. Let k_{-p} be a function that computes the canonical position of white and black king and also gives the number of the necessary transformation:

$$\begin{aligned} k_{-p}(w,b) &= (t_n(w), t_n(b), n), \text{ where} \\ \text{n'} &= T(w), \\ \text{n} &= \begin{cases} n'+4 & \text{if } t'(w) \in \text{a1-h8 diagonal} \wedge \\ & \quad t'(b) \notin \text{a1-h1-h8 triangle} \\ n' & \text{otherwise} \end{cases} \end{aligned}$$

We can use k_{-p} to define K_{-p}. K_{-p} computes the index in the lexicographically sorted list of canonical placements of the kings. It also gives the transformation needed to reach this canonical representation.

$$
\begin{aligned}
K_{-p}(w,b) = {} & (i,n), \text{ where} \\
& (w',b',n) = k_{-p}(w,b), \\
& (i = \|\{(w'',b'') \mid dist(w'',b'') \geq 2 \;\; \wedge \\
& \qquad\qquad k_{-p}(w'',b'') = (w'',b'',0) \;\; \wedge \\
& \qquad\qquad (w'' < w' \vee (w'' = w' \wedge b'' < b'))\}\|)
\end{aligned}
$$

The king enumeration with pawns is simpler as the transformation is dependent on the position of white king alone, and is either 0 or 1.

$$
\begin{aligned}
k_{+p}(w,b) = {} & (w',b',n), \text{ where:} \\
& (t(w) = w' \wedge t(b) = b') \wedge \\
& (w' \in \text{a1-d8 rectangle}) \wedge n \in \{0,1\} \\
K_{+p}(w,b) = {} & (i,n), \text{ where} \\
& (w',b',n) = k_{+p}(w,b) \wedge \\
& (i = \|\{(w'',b'') \mid dist(w'',b'') \geq 2 \;\; \wedge \\
& \qquad\qquad k_{+p}(w'',b'') = (w'',b'',0) \wedge \\
& \qquad\qquad (w'' < w' \vee (w'' = w' \wedge b'' < b'))\}\|)
\end{aligned}
$$

4.10.2 Enumeration of k like pieces

This is somewhat simpler as there is no symmetry involved. As I have restricted my focus on endgames with at most 5 pieces of which 2 are kings, we will only need enumerations of 2 and 3 pieces. The definitions below are only valid when the input are sorted in ascending order. I.e. i<j $=>$ $s_i < s_j$ is assumed in the definition of X^n.

Intuitively, a sequence of the positions of k like pieces $(s_1, \ldots, s_k)$ is mapped to the index in the lexicographic sorting of $\{(s_1, \ldots, s_k) \mid i < j \Rightarrow 0 \leq s_i < s_j < 64\}$

$$
X^2(s_1,s_2) = \begin{cases} X^2(s_1, s_2-1)+1 & \text{if } s_1+1 < s_2 \\ X^2(s_1-1, 63)+1 & \text{if } s_1 > 0 \wedge s_1+1 = s_2 \\ 0 & \text{if } s_1 = 0 \wedge s_2 = 1 \end{cases}
$$

$$
X^3(s_1,s_2,s_3) = \begin{cases} X^3(s_1, s_2, s_3-1)+1 & \text{if } s_2+1 < s_3 \\ X^3(s_1, s_2-1, 63)+1 & \text{if } s_1+1 < s_2 \wedge s_2+1 = s_3 \\ X^3(s_1-1, 62, 63)+1 & \text{if } s_1 > 0 \wedge s_1+2 = s_3 \\ 0 & \text{if } s_1 = 0 \wedge s_2 = 1 \wedge s_3 = 2 \end{cases}
$$

It is possible to express the above enumerations as polynomials by counting the number of ways to place the pieces in a lexicographically smaller way. I will show how to do this for $X^2(s_1, s_2)$, and simply state the result for the other enumerations.

$$
\begin{aligned}
X^2(s_1, s_2) &= [\textstyle\sum_{i=0}^{s_1-1} \sum_{j=i+1}^{63} 1] + [\sum_{i=s_1+1}^{s_2-1} 1] \\
&= [\textstyle\sum_{i=0}^{s_1-1}((63+1) - (i+1))] + [((s_2 - 1) + 1) - (s_1 + 1)] \\
&= [\textstyle\sum_{i=0}^{s_1-1}(63 - i)] + [s_2 - s_1 - 1] \\
&= [(((s_1 - 1) + 1) - 0) * 63 - \textstyle\sum_{i=0}^{s_1-1} i] + [s_2 - s_1 - 1] \\
&= [63 * s_1 - \tfrac{1}{2}((s_1 - 1) * ((s_1 - 1) + 1))] + [s_2 - s_1 - 1] \\
&= [\tfrac{1}{2}(127 * s_1 - s_1^2)] + [s_2 - s_1 - 1] \\
&= \tfrac{1}{2}(125 * s_1 - s_1^2) + s_2 - 1
\end{aligned}
$$

$$
X^3(s_1, s_2, s_3) = -64 + \tfrac{1}{6}(11531 * s_1 - 186 * s_1^2 + s_1^3) + \tfrac{1}{2}(125 * s_2 - s_2^2) + s_3
$$

As the pawns may not occur on rank 1 or 8, the legal positions are restricted to $\{8, \ldots, 55\}$. We need special enumerations for this piece.

$$
\begin{aligned}
P^2(s_1, s_2) &= \begin{cases} P^2(s_1, s_2 - 1) + 1 & \text{if } s_1 + 1 < s_2 \\ P^2(s_1 - 1, 55) + 1 & \text{if } s_1 > 8 \wedge s_1 + 1 = s_2 \\ 0 & \text{if } s_1 = 8 \wedge s_2 = 9 \end{cases} \\
&= -413 + \tfrac{1}{2}(109 * s_1 - s_1^2) + s_2
\end{aligned}
$$

$$
\begin{aligned}
P^3(s_1, s_2, s_3) &= \begin{cases} P^3(s_1, s_2, s_3 - 1) + 1 & \text{if } s_2 + 1 < s_3 \\ P^3(s_1, s_2 - 1, 55) + 1 & \text{if } s_1 + 1 < s_2 \wedge s_2 + 1 = s_3 \\ P^3(s_1 - 1, 54, 55) + 1 & \text{if } s_1 > 8 \wedge s_1 + 2 = s_3 \\ 0 & \text{if } s_1 = 8 \wedge s_2 = 9 \wedge s_3 = 10 \end{cases} \\
&= -10480 + \tfrac{1}{12}(17494 * s_1 - 324 * s_1^2 + 2 * s_1^3) \\
&\quad + \tfrac{1}{2}(109 * s_2 - s_2^2) + s_3
\end{aligned}
$$

In my implementation the piece enumerations for 2 like pieces are simply tabulated. For 3 like pieces such a tabulation would use 1 MB of memory and I decided to use the polynomial expression instead. The polynomial expression can be written $f_1(s_1) + f_2(s_2) + f_3(s_3)$ and can hence be optimised by tabulating f_1, f_2 and f_3.

4.10.3 Index functions

Let X and Y denote a knight, bishop, rook or queen. X and Y may denote the same piece, but the most specific index function should be used (ie. use KXXK instead of KXYK for the KRRK endgame).

$$
\begin{aligned}
I_{\text{KK}}(k_1,k_2) &= i, \\
&\quad \text{where } (i,\_) = K_{-p}(k_1,k_2) \\
\hline
I_{\text{KXK}}(k_1,x,k_2) &= 64 \cdot i + t_n(x), \\
&\quad \text{where } (i,n) = K_{-p}(k_1,k_2) \\
I_{\text{KPK}}(k_1,p,k_2) &= 48 \cdot i + (t_n(p) - 8), \\
&\quad \text{where } (i,n) = K_{+p}(k_1,k_2) \\
\hline
I_{\text{KXXK}}(k_1,x_1,x_2,k_1) &= (63 \cdot 64/2) \cdot i + X^2(t_n(x_1), t_n(x_2)), \\
&\quad \text{where } (i,n) = K_{-p}(k_1,k_2) \\
I_{\text{KXYK}}(k_1,x,y,k_2) &= 64 \cdot 64 \cdot i + 64 \cdot t_n(x) + t_n(y), \\
&\quad \text{where } (i,n) = K_{-p}(k_1,k_2) \\
I_{\text{KXKY}}(k_1,x,k_2,y) &= 64 \cdot 64 \cdot i + 64 \cdot t_n(x) + t_n(y), \\
&\quad \text{where } (i,n) = K_{-p}(k_1,k_2) \\
I_{\text{KXKP}}(k_1,x,k_2,p) &= 64 \cdot 48 \cdot i + 48 \cdot t_n(x) + (t_n(p) - 8), \\
&\quad \text{where } (i,n) = K_{+p}(k_1,k_2) \\
I_{\text{KPKP}}(k_1,p_1,k_2,p_2) &= 48 \cdot 48 \cdot i + 48 \cdot (t_n(p_1) - 8) + (t_n(p_2) - 8), \\
&\quad \text{where } (i,n) = K_{+p}(k_1,k_2) \\
I_{\text{KXPK}}(k_1,x,p,k_2) &= 64 \cdot 48 \cdot i + 48 \cdot t_n(x) + (t_n(p) - 8), \\
&\quad \text{where } (i,n) = K_{+p}(k_1,k_2) \\
I_{\text{KPPK}}(k_1,p_1,p_2,k_2) &= (47 \cdot 48/2) \cdot i + P^2(t_n(p_1), t_n(p_2)), \\
&\quad \text{where } (i,n) = K_{+p}(k_1,k_2) \\
\hline
I_{\text{KXXKY}}(k_1,x_1,x_2,k_2,y) &= 64 \cdot (63 \cdot 64/2) \cdot i + 64 \cdot X^2(t_n(x_1), t_n(x_2)) + t_n(y), \\
&\quad \text{where } (i,n) = K_{-p}(k_1,k_2) \\
&\ldots
\end{aligned}
$$

4.11 Some results

The tables below compares the index scheme presented in this section with the work of others for some representative endgames. The column "Number of positions" refers to how many are judged legal by Nalimov's definition (which is also the one adapted here).

The indexing scheme is not as compact as Nalimov's or Heinz's, but it is considerable better than Thompson and Edwards.

	Number of positions			This thesis			Nalimov		
Endgame	wtm+btm	=	total	wtm+btm	=	total	wtm+btm	=	total
KK	462	=	462	462	=	462	-	=	-
KNK	26282+28644	=	54926	2·462·64	=	59136	26282+28644	=	54926
KBK	24736+28644	=	53380	2·462·64	=	59136	27243+28644	=	55887
KRK	22497+28758	=	51255	2·462·64	=	59136	27030+28644	=	55674
KQK	18492+28644	=	47136	2·462·64	=	59136	25629+28644	=	54273
KPK	81664+84012	=	165676	2·1806·48	=	173376	81664+84012	=	165676
KBBK	650792+873642	=	1524434	2·462·(64·63)/2	=	1862784	789885 + 873642	=	1663527
KRBK	1203515+1754238	=	2957753	2·462·64·64	=	3784704	1594560 + 1747284	=	3341844
KQKQ	1145588	=	1145588	462·64·64	=	1892352	1563735	=	1563735
KNKP	4704266+4981504	=	9685770	2·1806·64·48	=	11096064	4931904 + 4981504	=	9913408
KNNNK	13486227+17472840	=	30959067	2·462·(62·63·64/6)	=	38497536	13486227 + 17472840	=	30959067
KRRKQ	32818990+35165947	=	67984937	2·462·(63·63/2)·64	=	117325824	46658340 + 46912050	=	93570390
KRNKQ	76966752+70038498	=	147005250	2·462·64·64·64	=	242221056	92308740 + 93824100	=	186132840
KQQKP	66292042+149445120	=	215737162	2·1806·(63·64/2)·48	=	349526016	131170128 + 149445120	=	280615248

	Heinz			Thompson			Edwards		
Endgame	wtm+btm	=	total	wtm+btm	=	total	wtm+btm	=	total
KNK	2·462·62	=	57288	2·462·64	=	59136	2·10·64·64	=	81920
KBK	2·462·62	=	57288	2·462·64	=	59136	2·10·64·64	=	81920
KRK	2·462·62	=	57288	2·462·64	=	59136	2·10·64·64	=	81920
KQK	2·462·62	=	57288	2·462·64	=	59136	2·10·64·64	=	81920
KPK	2·3612·24	=	173376	2·24·64·64	=	196608	2·32·64·64	=	262144
KBBK	2·462·(61·62/2)	=	1747284	2·462·64·64	=	3784704	2·10·64·64·64	=	5242880
KRBK	2·462·62·61	=	3494568	2·462·64·64	=	3784704	2·10·64·64·64	=	5242880
KQKQ	462·62·61	=	1747284	462·64·64	=	1892352	10·64·64·64	=	2621440
KNKP	2·3612·24·61	=	10575936	2·24·64·64·64	=	12582912	2·32·64·64·64	=	16777216
KNNNK	2·462·(60·61·62/6)	=	34945680	2·462·64·64·64	=	242221056	2·10·64·64·64·64	=	335544320
KRRKQ	2·462·62·(60·61/2)	=	104837040	2·462·64·64·64	=	242221056	2·10·64·64·64·64	=	335544320
KRNKQ	2·462·62·61·60	=	209674080	2·462·64·64·64	=	242221056	2·10·64·64·64·64	=	335544320
KQQKP	2·3612·24·(60·61/2)	=	317278080	2·24·64·64·64·64	=	805306368	2·32·64·64·64·64	=	1073741824

5 Compression using OBDDs

There are several reasons why I have chosen to use ordered binary decision diagrams as the mean of compressing endgame tables:

- They support fast random access
- It is a trivial problem to construct a minimal OBDD that represents any given table.
- The trivial indexing scheme maps an n men endgame to a table of size 64^n for each side to move. That means that when indexing an OBDD the first bit decides which half of the board the first piece is placed inside. The next 5 bits complete this binary search. The next 6 bits decide the position of the next piece and so on.

  Each bit hence has a clear interpretation on the board and this gives hope that the OBDD will be able to recognise some of the structure in the endgame table by representing it with few nodes.

  Even though the index scheme presented in the last section does not enjoy this direct interpretation of the bits, it can be padded to the size of the trivial mapping to regain this property.
- The endgame table contains entries with the broken position value. These entries will never be indexed and they can be replaced by whichever makes the OBDD compress better. This will be the topic of section 7.
- They can be represented using few bits per node.

An OBDD can be viewed as a restricted form of a deterministic finite automaton (DFA). Given a DFA D there is an efficient algorithm to find an equivalent DFA D' with a minimal number of nodes.

Hence I might as well have chosen to use DFA's, but after running the minimisation algorithm, the DFA would presumable be very similar to a minimal

OBDD (as all the accepted strings have the length $6n$). It would not be worth saving a few nodes at the expense of using more bits to represent each node.

5.1 Selecting the best suited variant of BDD's

There are several different variants of BDD's and I will assume that the reader is familiar with those.[9] In this thesis they are used for compression and hence it is a key feature that the OBDDs should be representable in a compact form. Another important aspect is that the computations needed to determine a minimal OBDD should remain feasible for table sizes up to $32 \cdot 64^3 = 2^{23}$ (produced by 4-men endgames with pawns), and preferably for even larger tables.

Given the above constraints and the fact that the input to the BDD constructor is a table of size 2^n, it is obvious to restrict the BDD to be ordered and to choose the ordering $b_{n-1} < b_{n-2} < ... < b_0$.

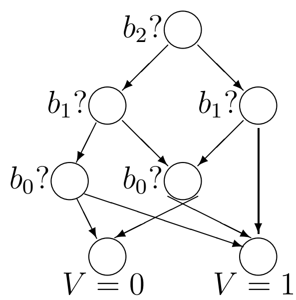
A minimal OBDD (also called reduced) satisfies the two requirements:

Uniqueness: No two distinct nodes define identical sub-OBDDs.

No redundant tests: No node has 2 identical children.

As an example, let's consider representing the 8 entry table shown below as a binary decision diagram. The minimal OBDD is shown to the right of it. A convention that will be used throughout this thesis is that each left arrow points at the successor for b_i=0 and each right arrow points at the successor for b_i=1.

Index in binary	Result
000	0
001	1
010	1
011	0
100	1
101	0
110	1
111	1



I have chosen to replace the requirement of no redundant tests with a requirement of exhaustive tests — i.e. that all variables b_i's are tested on each path from the root to a leaf. An OBDD satisfying this requirement is said to be complete. Pros and cons of this choice include:

+ : The number of legal successor nodes is reduced. Hence less bits are needed to define each node (in the example below each node is labelled by its index).

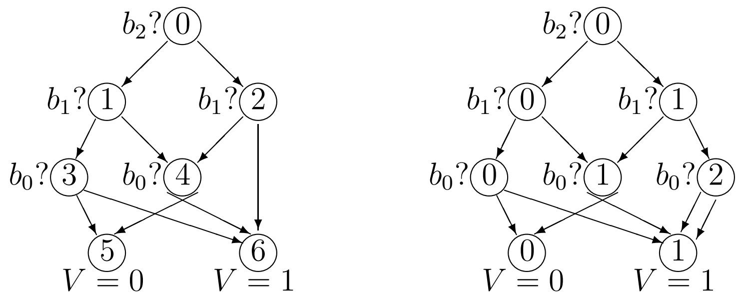
[9]If not, see e.g. [24]

+ : Implementations becomes simpler and faster as the OBDD is now separated into n layers — one for each b_i, and every successor points to a node in the layer below.

− : The OBDD is no longer minimal. In the example below one extra node will be introduced.

From this point on, every time an OBDD is mentioned, it has the requirement of exhaustive tests.

No redundant tests: Exhaustive tests:



5.2 Representation of OBDDs

Let $n_{i,j}$ be the j'th node in the i'th layer counted from the root. An h-layered OBDD can be defined by the following properties:

1 The height of the tree, h.

2 For each layer i, the number of nodes c_i in that layer.

3 For each internal node $n_{i,j}$ 2 indexes $0 \leq s_0, s_1 < c_{i+1}$

4 No information is stored for the leaf nodes — the value of a leaf node is simply its index (a separate conversion table is hence needed to convert this value to a draw, a broken position, a mate in 3 etc.).

The OBDDs constructed from endgame tables are huge. This means that the representation of properties 1 and 2 above are negligible in size compared with property 3.

We can represent each internal node on layer i with $2 * \lceil log(c_{i+1}) \rceil$ bits. In this coding scheme we waste $2 * (\lceil log(c_{i+1}) \rceil - log(c_{i+1}))$ bits per node. We could get close to optimal by using one bit less for each children having index in $[0 \ldots 2^{\lceil log(c_{i+1}) \rceil} - c_{i+1}[$.[10] However, when nodes no longer have a fixed size you can't access them by doing arithmetic on their size, and a linear scan is too slow for our purpose.

[10] Take $c_{i+1} = 6$ as an example. We can use the representation $0 \rightarrow 00$, $1 \rightarrow 01$, $2 \rightarrow 100$, $3 \rightarrow 101$, $4 \rightarrow 110$, $5 \rightarrow 111$.

5.3 Implementation

Below is shown the `C++` code for indexing an OBDD. The class `CompressedList` implements an array of n bits unsigned integers. The method `operator[]` makes a couple of assumptions. First, the n bits must be available in 4 consecutive bytes. The least n for which this is not the case is $1+8+8+8+1 = 26$. However this situation will never occur as 2 divides both 8 and 26 but not 1. The method will hence be valid for any $n \leq 26$. The second assumption is that a small endian byte order is used. The cast from a char pointer to an integer pointer relies on this. The second assumption is not that nice, but it does achieve a significant speedup compared to an older version of the method without this assumption.

```
class CompressedList {
...
  uint operator[](int index) {
    index *= bits_per_value;
    // ONE_BITS[v] = (1 << v)-1;
    return ONE_BITS[bits_per_value] &
           (*((uint *)(&(mem[index >> 3]))) >> (index & 7));
  }
...
  int bits_per_value;
  int size;
  uchar *mem;
}

class BinaryDecisionDiagram {
...
  uint operator[](int index) {
    int i=0;
    for (int layer=log_size-1; layer>=0; layer--) {
      int next_bit = (index>>layer)&1;
      i = nodes[layer][(i<<1) + next_bit];
    }
    return i;
  }
...
  int log_size;
  CompressedList *nodes;
}
```

5.4 Adjusting index scheme for OBDD

The index scheme introduced in last section has the unfortunate property that it spoils the direct relationship between individual bits of the index and the position of the pieces. To retain this relationship the table is expanded back to something close to the trivial indexing.

The example below shows how to compute the index for the KRRKP endgame. After this example it should be clear how to compute the index for any endgame, and I will therefore not be going int any further details. In the example below it is assumed that an eventual en passant or castling capability has already been encoded in the position of the pieces.

The input to the index function is given as a list of pieces sorted in the order $K_wQ_wR_wB_wN_wP_wK_bQ_bR_bB_bN_bP_b$. In this example we are given

$$\langle k_w, r1_w, r2_w, k_b, p_b \rangle$$

Remove the kings from this list, compute their index in the king enumeration and put this index in front of the list. Also, apply the required transformation to the remaining pieces. We then get:

$$\langle n, r1_w, r2_w, p_b \rangle, \text{ where } (i, n) = K_{+p}(k_w, k_b)$$

With multiple pieces of the same king and colour, arrange those in a descending order.

$$\langle n, \max(r1_w, r2_w), \min(r1_w, r2_w), p_b \rangle, \text{ where } (i, n) = K_{+p}(k_w, k_b)$$

Subtract 8 from each pawn position.[11]

$$\langle n, \max(r1_w, r2_w), \min(r1_w, r2_w), p_b - 8 \rangle, \text{ where } (i, n) = K_{+p}(k_w, k_b)$$

What remains is simply to view the entries in the above list as the digits in the base-64 system.

$$2^{3 \cdot 6} n + 2^{2 \cdot 6} \max(r1_w, r2_w) + 2^6 \min(r1_w, r2_w) + (p_b - 8), \\ \text{where } (i, n) = K_{+p}(k_w, k_b)$$

Even though this indexing scheme is less space efficient as the one presented in the last section, it is only the number of broken entries that have increased.

In the next section it is shown how we can improve the compression of the OBDD by remapping these entries to other values.

Another thing to notice is that the above indexing scheme does not provide the direct relationship with the bits of an index and the position of the kings. I did not want to blow up the size of the table this much. Furthermore, it is the most significant bits in the index that encodes the positions of the kings. The ordering chosen for the OBDD has these bits placed closest to the root of the OBDD and hence they have only very little impact on the size of the OBDD.

[11] I thought that using the square numbers [0 . . . 48[instead of [8 . . . 56[would make the OBDD perform slightly better. I haven't tested this though, and I have regretted this choice as it makes these index functions slower and less uniform.

5.5 Compression ratio achieved

The first results are listed below. After having implemented the OBDDs I tested them on the endgames shown below. At this point, the implementation was slightly different as each OBDD represented an endgame for *both* sides to move. Therefore only one size is shown for each endgame below. It shows that the endgame tables are compressed to roughly one half of the size of the uncompressed tables using the index scheme from the last section. However, using gzip on these uncompressed tables outperforms the OBDDs by a factor of 2. An endgame like KQKR is particularly disappointing.

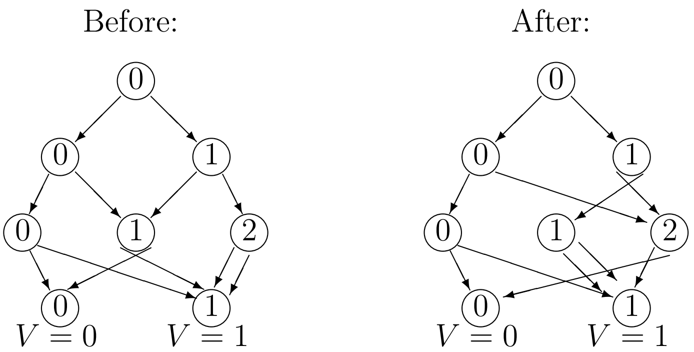
Endgame	Number of positions	Number of different values	OBDD size	OBDD bits/pos.	gzipped size	gzipped bits/pos.	WDL OBDD size	WDL bits/pos.
KRK	51255	34	33259	5.19	14630	2.28	6896	1.08
KQK	47136	22	30303	5.14	13125	2.23	8404	1.43
KPK	165676	57	90362	4.36	35009	1.69	30052	1.45
KNNK	1608946	3	97016	0.48	77069	0.38	97016	0.48
KBBK	1524434	40	832013	4.37	362087	1.90	123836	0.65
KRRK	1417452	25	686574	3.87	285449	1.61	90108	0.51
KQQK	1235796	16	535801	3.47	312591	2.02	116184	0.75
KBNK	3138965	68	2436117	6.21	1044635	2.66	494068	1.26
KRNK	3026203	34	1701451	4.50	755834	2.00	383828	1.01
KRBK	2957753	34	1761171	4.76	755982	2.04	396056	1.07
KQNK	2802307	21	1295834	3.70	677757	1.93	367916	1.05
KQBK	2733056	20	1439065	4.21	722843	2.12	393452	1.15
KQRK	2642086	24	1115569	3.38	572585	1.73	366328	1.10
KNKN	1603202	3	147844	0.74	98619	0.49	147844	0.74
KBKB	1514236	3	147688	0.78	104930	0.55	147688	0.78
KRKR	1390127	40	564435	3.25	264156	1.52	228652	1.32
KQKQ	1145588	27	612876	4.28	326456	2.80	268008	1.87
KBKN	3117438	3	323200	0.83	209740	0.54	323200	0.83
KRKN	2994304	82	1664647	4.45	686465	1.83	647608	1.73
KRKB	2905006	60	1003585	2.76	459190	1.26	522884	1.44
KQKN	2748790	44	2681333	7.80	1170053	3.41	441736	1.29
KQKB	2659824	36	2545905	7.66	1079884	3.25	471948	1.42
KQKR	2534931	72	3399869	10.73	1361639	4.30	450784	1.42

Notice that whenever the size of an OBDD is shown, it is the size it has when it has been saved to a file. Currently a number of calls to `malloc` cause a slight overhead when it is has been loaded. This problem can easily be fixed, but it has not been given top priority.

In their current form, OBDDs are no match against a program like gzip. Luckily there are ways to make them perform better. This is what the rest of the thesis will focus on.

6 Improving the representation of an OBDD

As mentioned in section 5.2, most of the information needed to describe an OBDD is used to determine the children of each internal node.
To represent a layer of nodes we used m bits for each left and the right child of a node, where the layer below contained n nodes and $2^{\log(m)-1} < n \leq 2^{\log(m)}$. This representation gives no guidelines for how to enumerate the nodes in each layer of an OBDD. Actually, given a representation of an OBDD we can perform an arbitrary permutation of each internal layer of nodes as long as we remember to update the indexes in the layer above accordingly.
If we take as example the OBDD from the last section, we can make a permutation of the second lowest layer by swapping node 1 with node 2:



In general, if an OBDD contains k layers of nodes and each layer has size n_i, then we can permute the layers in $\prod_{i=0}^{k-1} n_i!$ different ways without affecting its interpretation.
Having a description with such a high degree of freedom is expensive. No matter which enumerations of the nodes are chosen, the size of the representation remains fixed. This means that in theory we could save

$$\begin{aligned}
\log(\textstyle\prod_{i=0}^{k-1} n_i!) &= \textstyle\sum_{i=0}^{k-1} \log(n_i!) \\
&\approx \textstyle\sum_{i=0}^{k-1} (n_i \cdot \log(n_i) - n_i) \\
&= \textstyle\sum_{i=0}^{k-1} (n_i \cdot \log(n_i)) - \sum_{i=0}^{k-1} (n_i)
\end{aligned}$$

bits by choosing some canonical representation. The interesting question is how much we can save and still have a feasible representation.

To get some idea about the size of the above expression, let's do some further calculations. Each node has exactly 2 children, hence $2 \cdot n_i \geq n_{i-1}$. This gives

$$\begin{aligned}
\textstyle\sum_{i=0}^{k-1} (n_i \cdot \log(n_i)) - \sum_{i=0}^{k-1} (n_i) &\geq \textstyle\sum_{i=0}^{k-1} (n_i \cdot \log(\frac{1}{2} n_{i-1})) - \sum_{i=0}^{k-1} (n_i) \\
&\geq \textstyle\sum_{i=0}^{k-1} (n_i \cdot (\log(n_{i-1}) - 1)) - \sum_{i=0}^{k-1} (n_i) \\
&\geq \textstyle\sum_{i=0}^{k-1} (n_i \cdot (\log(n_{i-1}) - 2))
\end{aligned}$$

If we assume $\forall i$: $n_i \geq 64 \Leftrightarrow \log(n_i) \geq 6$ then

$$\begin{aligned}\textstyle\sum_{i=0}^{k-1}(n_i \cdot (\log(n_{i-1}) - 2)) &\geq \textstyle\sum_{i=0}^{k-1}(n_i \cdot (\log(n_{i-1}) - \frac{1}{3}\log(n_{i-1}))) \\ &= \textstyle\frac{1}{3}(2 \cdot \sum_{i=0}^{k-1}(n_i \cdot \log(n_{i-1})))\end{aligned}$$

Of course the assumption above does not hold, but we are dealing with large OBDDs, so the assumption will more than hold on the big layers — those who really contribute to the above sum.

Compare the last expression with the number of bits we used to define the children of all nodes:

$$2 \cdot \textstyle\sum_{i=0}^{k-1}(n_i \cdot \log(n_{i-1}))$$

We see that at least $\frac{1}{3}$ of the bits are wasted to determine the permutation of the nodes.

6.1 Improved representation

How can we find a more efficient representation then? With the current representation the cost of computing the index from a chess position is about the same as the cost of performing a lookup in the OBDD with this index. Hence any slowdown caused by a more memory efficient representation will be noticeable. The aim is to achieve a total lookup time in the same order of the time it takes a program to search a position. Hence, with $\frac{1}{2}\mu s$ as the goal, anything above a minor slowdown is not satisfactory.

I pursued the idea of producing a lot of nodes with the following property.

Definition: *A node is called ideal if the index of the right child is exactly one higher than the index for the left child.*

If a node is ideal, we can cut down the representation from $2m$ to m bits by only storing the index of the left child. As an example, this property is satisfied for respectively 4 and 2 of the nodes in the OBDDs from the examples above. However 2 problems occur in this approach:

1 How do we know which nodes are ideal or not?

2 If some nodes require $2 \cdot m$ bits and others only m bits, then we can't use simple arithmetic to index them.

These problems can be answered at the cost of adding an integer for each layer. This integer gives the number of nodes in the that layer are *not* ideal. The non-ideal nodes are placed first in the list (each occupying two entries of m bits), and the ideal nodes are placed last (each occupying a single m bits entry).

6.2 Modified indexing algorithm

The indexing code is only made slightly more complex by using the idea just presented. The reader is invited to compare with the old version from section 5.3. The implementation below shows that the choice of placing the ideal nodes last was not arbitrary.

```
class BinaryDecisionDiagram {
...
  uint operator[](int index) {
    int i=0;
    for (int layer=log_size-1; layer>=0; layer--) {
      int next_bit = (index>>layer)&1;
      if (i >= index_split[layer]) {
        // An ideal node
        i = nodes[layer][index_split[layer] + i] + next_bit;
      } else {
        // A non-ideal node
        i = nodes[layer][(i<<1) + next_bit];
      }
    }
    return i;
  }
...
  int log_size;
  CompressedList *nodes;
  int *index_split;
}
```

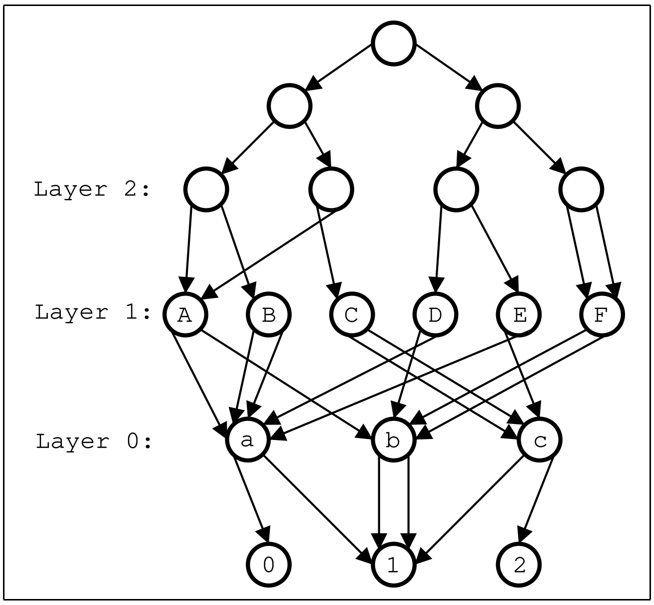
6.3 Algorithm

We are faced with the problem of reordering the nodes in each layer with our objective as maximising the total number of ideal nodes.

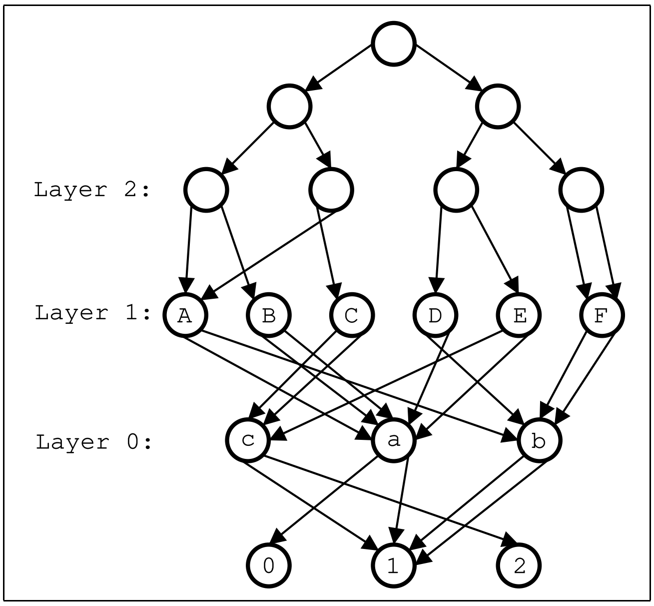
Let's enumerate the layers in an OBDD from $n-1$ which contains the root node down to layer 0, which is the lowest layer of internal nodes. This is a natural choice. When indexing an OBDD we then use the i'th bit of the index to choose the left or right child of a node in layer i. Remember that our OBDD description does not include the leaf nodes, as their values are simply their index.

The heuristic algorithm presented here tries to maximise the number of ideal nodes in one layer at a time starting from the bottom. We increase the number of ideal nodes in a layer by reordering the nodes in the layer below. As we don't have any representation of the leaf nodes, we can't do any reordering of them and hence can't improve the number of ideal nodes in layer 0. Therefore we start at layer 1.

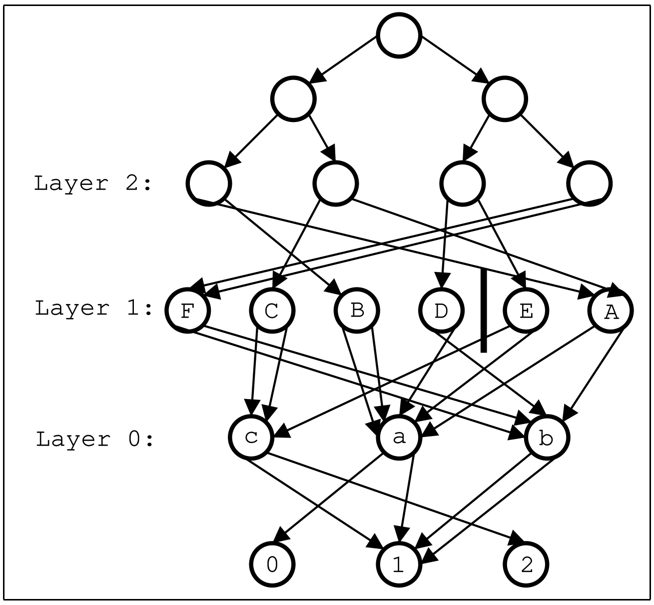
Below is shown an example of how the number of ideal nodes can be improved from 1 to 2 for layer 1.



In layer 1 of the above OBDD only A is an ideal node. If we reorder the nodes in layer 0 and updates the child indexes in layer 1 correspondingly then E will also become an ideal node:



Finally we have to rearrange the nodes in layer 1 such that the ideal nodes are placed last (the solid line represent the split between non-ideal nodes and the ideal nodes). This includes updating the child indexes in layer 2 correspondingly.



The reader might have noticed that a problem will occur when this procedure is repeated for layer 2. We are no longer free to reorder the nodes in layer 1 as we wish, because every ideal node has to be placed to the right of any non-ideal node.

This constraint obviously reduces the number of ideal nodes produced henceforth. However, it does not make the implementation any harder. A simple solution is to only run the algorithm on 2 subsets of the nodes in layer 2 — those who have no ideal children and those who have 2. This guarantees that the reordering in layer 1 will not cross the boundary between the ideal and non-ideal nodes. The nodes with one ideal and one non-ideal child are a lost case. The third node in layer 2 is actually ideal even though it is of this type, but this is the only ideal node we can have of this kind and I've chosen to ignore this tiny optimisation.

6.4 Finding an optimal reordering is NP-complete

The above description ignores the fact of how to reorder the nodes, such as to maximise the number of ideal nodes. Once we have formalised this problem, it is a simple matter to show that it is NP-complete. Notice however that this do not necessarily imply that the problem of maximising the number of ideal nodes in the entire OBDD is NP-hard.

Maximum Ideal Nodes (MIN):

Instance: Set N of pairs of integers. Each pair represents a node in the layer in which we seeks to maximise the number of ideal nodes. A pair of integers defines a node by the indexes of its left and right child.
An integer n gives the number of nodes in the layer below.

Solution: A permutation p of the n nodes in the layer below.
Measure: Number of nodes made ideal by applying the permutation p.

The decision version of Maximum Ideal Nodes is:

$$L_{MIN} = \{((N, n), k) \mid \text{The MIN instance } (N, n) \text{ has a solution of size } \geq k\}$$

Proposition: *L_{MIN} is NP-complete.*

Proof. Let $G = (V = \{0, \ldots, n-1\}, E = N)$ be a directed graph. We then have that $(i, j) \in E$ iff a nodes has i as a left child and j as a left child. The following lemma is usefull.

Lemma: *A cover of V by k node disjoint paths can be used to construct a permutation that makes $n-k$ nodes ideal — and vice versa.*

Proof of lemma.

"$\Rightarrow$": Each path of length m covers $m+1$ nodes. This means that the total length of all k paths must be $n-k$, which is also the numbers of edges used. By traversing the k paths one at a time, the order of which the nodes are visited is how they should be ordered. Now each edge contained in some path will correspond exactly to a node that is made ideal.

"$\Leftarrow$": Assume that the permutation $\langle p_i \rangle_i$ produces $n-k$ ideal nodes. Construct k node disjoint paths by cutting the sequence $\langle p_0, \ldots, p_{n-1} \rangle_i$ between every p_i and p_{i+1} for which no ideal node has p_i as left child and p_{i+1} as right child (or equivalently, for which there is no edge (p_i, p_{i+1})). This is the case for $(n-1) - (n-k) = k-1$ values of i and hence produces the k node disjoint paths.

The lemma says in particular that to solve the hamiltonian path problem for the graph G is equivalent with making $n-1$ nodes ideal. Given any graph G', it is easy to construct an instance of the MIN problem such that $(V = \{0, \ldots, n-1\}, E = N) = G'$. This shows that MIN is NP-hard. Also, since MIN is trivially in NP this concludes the proof. □

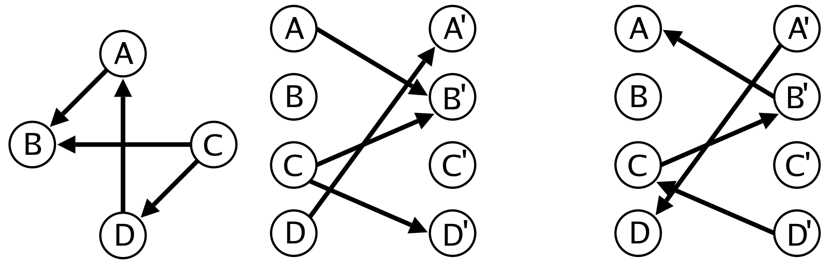
6.5 Using maximum matching to find an ordering

This algorithm is easiest described as a solution to the equivalent problem of covering the nodes in a graph with fewest node disjoint paths. When G contains no cycles the problem can be solved to optimality by reducing it to the problem of finding a maximum match in a bipartite graph.

However, when G contains cycles, the interpretation of the solution to the maximum matching problem fails. What I have done is to modify the Hopcroft-Karp matching algorithm ([40]) by imposing restrictions on which alternating paths must be chosen. I will assume that the reader knows about this algorithm. The restriction makes it likely that the interpretation holds in the end.

I will start by presenting an algorithm that works for undirected graphs and then add the modifications afterwards. Construct from G a bipartite graph $G' = (V', E')$. For each node b in V we have 2 nodes b, b' in V'. For each edge (a, b) in E we have a node (a, b') in E'. Consider the example below.

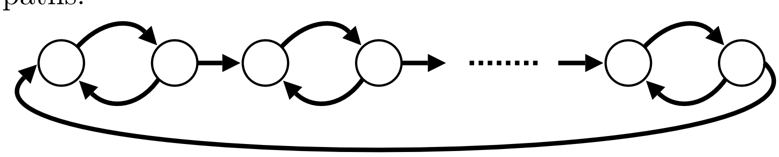
G: G': After having found alternating paths (each left arrow determines a match). G'':



From G'' we can read of the paths as follows. If we delete any right-directed edge from G'' and merge every pair of nodes x and x', then the remaining edges defines the paths in the node cover. Hence, the paths can be read of the graph G'' like this. Each x with no incoming edge define the start of a path. If x' has an outgoing edge (x', y) then add y to the end of the path. Now repeat this procedure with y and continue as long as possible.

As the left directed edges in G'' represents a solution to the bipartite matching problem, each left node in G'' has at most one incoming edge and each right node has at most one outgoing edge. This means that the paths defined in the procedure above must be node disjoint.

The fact that G does not contain cycles assures us that that we can find a x with no incoming edge as the start of each path. Without this property we could still use the result from the matching algorithm by detecting and breaking up the cycles into paths. However the relationship between the size of the matching and the number of paths we derive from it would disappear. Let's say we have the graph below with $2k$ nodes and we wish to cover its nodes with a minimal number of paths.

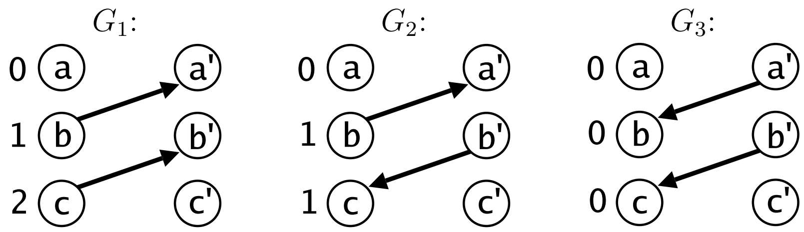


By using the above approach, we would find a perfect matching. This would represent a covering of the graph with cycles, which we could then break up into paths. However the solution that represents one big cycle containing all nodes is just as optimal as the solution that has k small cycles of 2 nodes.

6.5.1 Modified maximal matching algorithm

This is a modification to the algorithm using alternating paths. Only improving paths which do not introduce any cycles under the above interpretation are allowed.

Add a label to each left node. We wish to maintain the invariant that each left node has the same label as the first node in the path of which it is a member (according to the before given interpretation of the match). The purpose of this invariant is to make it easy to decide, whether reversing a path will introduce a cycle. An example is shown below.



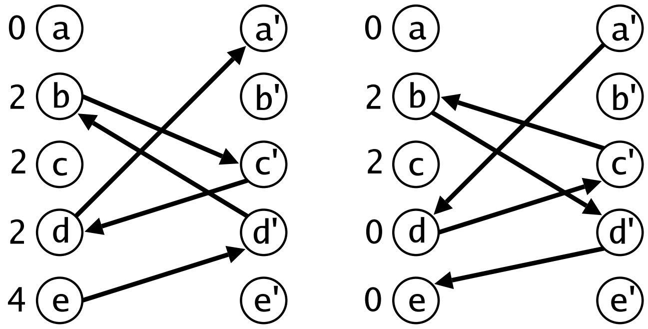
Remember that it is the left directed edges that represent a solution and hence those that determines the paths. In the above example the final solution is the single path $\langle a, b, c\rangle$.

To maintain the invariant we have to relabel some nodes for every right directed node we reverse. In the step from G_1 to G_2 above, the edge (c, b') is reversed to (b', c). This has the effect that (in the interpretation of the match) the node c is added to the path ending at b. Therefore, the node c must be given the same label as the node b. The next step from G_1 to G_2 is similar. (b, a') is reversed to (a', b) and the node b is given the same label as the node a. However b is not the last node in the path covering it, so any nodes following it must also be relabelled. In this case c is labelled 0.

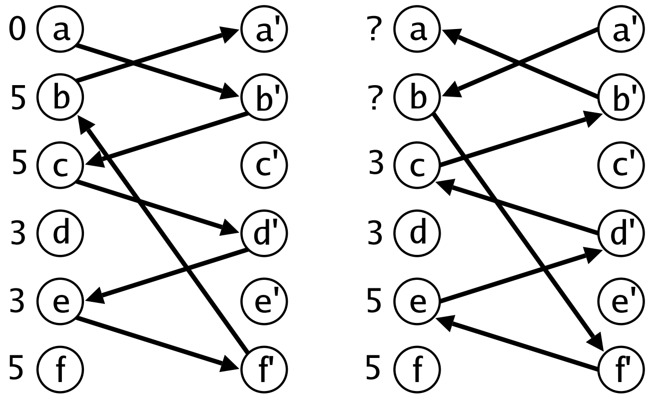
In the search for augmenting paths we rely on the assumption that reversing a right directed edge (x, y') where x and y share labels will introduce a cycle. The algorithm is therefore not allowed to choose paths in which such an edge occur.

The assumption is true in all simple cases. Before reversing a right directed edge (x, y') where x and y share labels, there must be already be some path from x to y. If this path is not affected by reversing the augmenting path that includes (x, y'), then the path from x to y will become a cycle by adding the edge (y, x).

However, it is possible to find examples in which the augmenting path interacts with the paths already defined. Below is shown an example where reversing the non-allowed edge (b, c') will not introduce any cycles after all.



On the other hand, the example below shows that an augmenting path may introduce a cycle even if the path don't contain any right directed edge between equally labelled nodes. The cycle is made between the a and b node.



However, the examples above don't occur that often in practice. Without having any constraints in the search for augmenting paths, too many of them would end up introducing a cycle and would have to be rejected. The constraint of not including any right directed edge between equally labelled nodes minimises this waste at the cost of overlooking only a few good augmenting paths.

When an augmenting path has been found all the edges on its path are reversed, and the nodes are relabelled as described earlier. If a cycle is introduced, then we have to undo the changes and continue the search for augmenting paths. If two augmenting paths sharing the starting node have been rejected in a row, then continue the search from a new starting node.

6.6 Results

It would be natural to compare the performance of the algorithm just described with the performance of an unmodified max matching algorithm, where we simply unfold the cycles into paths.

However, I did not carry out this experiment. Instead I just assumed that the solution presented here was superior. The modification to Hopcroft-Karp's algorithm avoids increasing the size of the match if this improvement doesn't translate to a node cover with fewer paths. Hence, by saves this potential it might be that another improvement of the match *does* help us.

In all of the few results shown below, the representation has decreased about 20% in size. The calculation in the beginning of this section showed that, in theory, at least a reduction of $\frac{1}{3}$ could be achieved. We have utilised about half of this potential at the cost of a slight slow down of the lookup operation.

Perm	KRK		KQK		KPK		KBBK		KRNK	
	Before	After	Before	After	Before	After	Before	After	Before	After
8	23995	19373	17507	13725	53942	42235	331589	261466	-	804841

7 Mapping of don't cares (broken positions)

In this thesis the OBDDs are constructed from endgame tables that generally contain plenty of broken positions. Some of these were already contained in the index scheme presented in section 4 and denote e.g. positions with a rook and a king on the same square. But most of the broken positions were introduced when expanding this format into the simpler indexing scheme used by OBDDs.

To illustrate the idea of this section let's consider the endgame KBKN. This endgame is generally drawn and it is only in some rare situations that a check mate can occur. This means that the table will consist of a few entries encoding a check mated position or a mate in one. Most of the entries correspond to a drawn position and about one third of the entries are broken positions.

If we require that the chess program will never probe the endgame tables for any broken position then we can assign any value we wish to these entries. The task is thus to find a mapping for these "don't care" entries such that the minimal OBDD will decrease in size. In the example with the KBKN endgame we could make a great improvement by simply mapping all the broken positions to the value encoding a drawn position. This should be clear as the 0'th order entropy would drop to almost 0.

7.1 Related work

The endgame tables used by the checkers program *Chinook* [19] only says whether a position is won, drawn or lost. A run-length encoding is used such that each endgame is stored as a number 1 K-byte compressed blocks. A separate index table contains the index number starting each 1 K-byte block. To do a lookup, first a binary search is performed in the index table and then the relevant block is decompressed until the requested value has been found.

The index scheme used for checkers endgames is not perfect either — some entries represent a broken position. The programmers of *Chinook* have chosen a very simple approach for reassigning these value. Simply count the number of positions with each of the values won, drawn and lost. The broken positions are then given the most frequently occurring value.

We will see that this simple solution improves the compression of the OBDDs as well. However, superior heuristics are known for minimising OBDDs by reassigning unused entries. In the literature, a value that may be changed is called a don't care value. I will stick to this convention. The heuristics given in the articles [27], [28] and [29] are all very similar to what has been done in this thesis.

An important distinction between the work in this thesis and the work of others is the representation of the input to the don't care minimisation algorithm. In all the related work I have examined, the input is given as a, ordered or not, BDD. Normally this compact representation is necessary, as unfolding it into a table with 2^n entries would be infeasible. However, the algorithm for don't care minimisation becomes much simpler when the input is simply given as a table. Therefore I won't present any of the work in articles [27], [28] and [29] as they basically present some simple heuristics that are complicated by the use of OBDDs as their input model.

It has been shown in [25] that the problem of minimising an OBDD using don't cares is NP-complete. Furthermore, a strong inapproximability result is given. These results are however only valid for the problem using OBDDs as the input model. The problem instances that are used to show the hardness would be blown up by an exponential factor if they were changed to the simple representation as a table.

However, by using a more complicated reduction, it can be shown that both hardness results still hold. This proof will be given in the next section.

7.2 Examples

Let's refer to the mapping of don't care entries to the most frequently occurring value as simple. The mapping used in this thesis will be called a top-down mapping (for reasons that will become clear).

Let's compare the simple mapping with the top-down mapping on a concrete example. The 32 characters below represent the entries in the table we will be working on in this example (spaces are inserted to ease readability). The characters `*`, `d`, `0`, `1` represents a don't care, a draw, a check mate (-M0) and a mate in one (M1) respectively.

```
*dd0 d**d 0*dd **dd  d*d0 **0d 01dd dd**
```

Without applying any mapping of don't cares the size (counted in number of nodes) of the minimal OBDD would be (l_i denotes the set of nodes in layer i):

$$
\begin{aligned}
\|l_5\| &= \|\{\texttt{*dd0 d**d 0*dd **dd d*d0 **0d 01dd dd**}\}\| = 1 \\
\|l_4\| &= \|\{\texttt{*dd0 d**d 0*dd **dd, d*d0 **0d 01dd dd**}\}\| = 2 \\
\|l_3\| &= \|\{\texttt{*dd0 d**d, 0*dd **dd, d*d0 **0d, 01dd dd**}\}\| = 4 \\
\|l_2\| &= \|\{\texttt{*dd0, d**d, 0*dd, **dd, d*d0, **0d, 01dd, dd**}\}\| = 8 \\
\|l_1\| &= \|\{\texttt{*d, d0, d*, 0*, dd, **, 0d, 01}\}\| = 8 \\
\|l_0\| &= \|\{\texttt{d, 0, 1, *}\}\| = 4 \\
\sum_{i=0}^{5} \|l_i\| &= 4 + 8 + 8 + 4 + 2 + 1 = 27
\end{aligned}
$$

Now, let's try to preprocess the table by mapping all don't cares to the most frequently occurring character d.

Before mapping	:	`*dd0 d**d 0*dd **dd  d*d0 **0d 01dd dd**`
After	:	`ddd0 dddd 0ddd dddd  ddd0 dd0d 01dd dddd`

This reduces the size of the minimal OBDD by 8 nodes:

$$
\begin{aligned}
\|l_5\| &= \|\{\texttt{ddd0 dddd 0ddd dddd ddd0 dd0d 01dd dddd}\}\| = 1 \\
\|l_4\| &= \|\{\texttt{ddd0 dddd 0ddd dddd, ddd0 dd0d 01dd dddd}\}\| = 2 \\
\|l_3\| &= \|\{\texttt{ddd0 dddd, 0ddd dddd, ddd0 dd0d, 01dd dddd}\}\| = 4 \\
\|l_2\| &= \|\{\texttt{ddd0, dddd, 0ddd, dd0d, 01dd}\}\| = 5 \\
\|l_1\| &= \|\{\texttt{dd, d0, 0d, 01}\}\| = 4 \\
\|l_0\| &= \|\{\texttt{d, 0, 1}\}\| = 3 \\
\sum_{i=0}^{5} \|l_i\| &= 3 + 4 + 5 + 4 + 2 + 1 = 19
\end{aligned}
$$

The strategy used in the top-down mapping is this: First try to make as many big blocks[12] identical by mapping some of the don't cares. Then continue with smaller and smaller blocks until they eventually have size 1. By applying this strategy we get the mapping

Before	:	`*dd0 d**d 0*dd **dd  d*d0 **0d 01dd dd**`
After	:	`ddd0 dd0d 01dd dddd  ddd0 dd0d 01dd dddd`

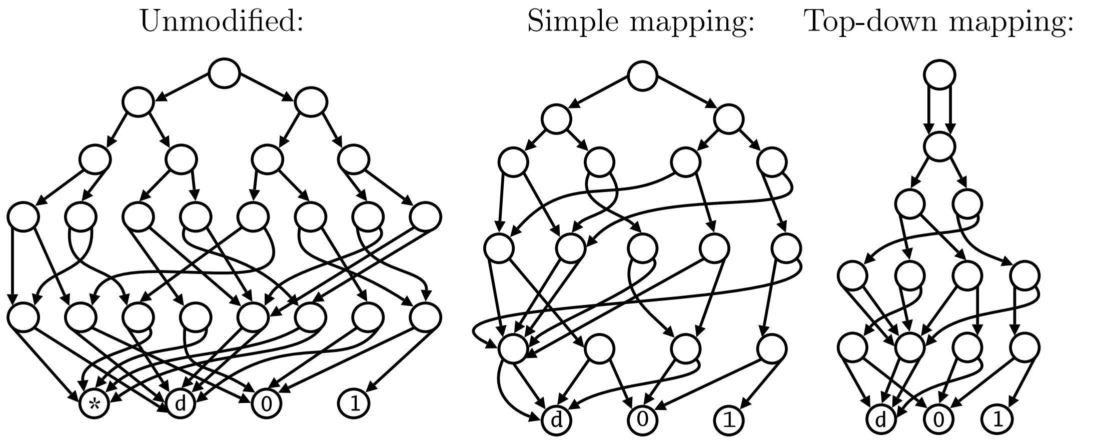
Compared to the simple mapping, the OBDD shrinks by another 4 nodes:

$$
\begin{aligned}
\|l_5\| &= \|\{\texttt{ddd0 dd0d 01dd dddd ddd0 dd0d 01dd dddd}\}\| = 1 \\
\|l_4\| &= \|\{\texttt{ddd0 dd0d 01dd dddd}\}\| = 1 \\
\|l_3\| &= \|\{\texttt{ddd0 dd0d, 01dd dddd}\}\| = 2 \\
\|l_2\| &= \|\{\texttt{ddd0, dd0d, 01dd, dddd}\}\| = 4 \\
\|l_1\| &= \|\{\texttt{dd, d0, 0d, 01}\}\| = 4 \\
\|l_0\| &= \|\{\texttt{d, 0, 1}\}\| = 3 \\
\sum_{i=0}^{5} \|l_i\| &= 3 + 4 + 4 + 2 + 1 + 1 = 15
\end{aligned}
$$

Finally, let's compare the OBDDs visually. The OBDD produced by applying the top-down mapping consists of much less nodes near the root. It can be

[12] I will use the word block for a subsequence of the table that can be represented by node in the OBDD — ie. a subsequence of the form $t[n \cdot 2^k \ldots (n+1) \cdot 2^k[$ for integer valued k and n.

very costly (counted in number of don't cares that we have to reassign) to unify two nodes near the root of the OBDD. The reason it pays off is that it usually involves unifications of nodes in the lower layers at the same time. Remember the convention that the left child is indexed with a 0 and the right child is indexed with 1.



7.3 Implementation of top-down mapping

The top-down mapping algorithm can be implemented quite simple.

Algorithm: `SimpleTopDownMapping`
Input: $t[0 \ldots 2^n[$ of integers — 0 represents a don't care
Output: $t[0 \ldots 2^n[$ with no zero valued entry
for $m = n - 1$ down to 0 do begin
    for $k_1 = 0$ to $2^n - 2 \cdot 2^m$ with step 2^m do begin
        for $k_2 = k_1 + 2^m$ to $2^n - 2^m$ with step 2^m do begin
            if the blocks $t[k_1 \ldots k_1 + 2^m[$ and $t[k_2 \ldots k_2 + 2^m[$ can be
                made identical by mapping some of the don't cares contained in
                $t[k_1 \ldots k_1 + 2^m[\cup t[k_2 \ldots k_2 + 2^m[$ then do so
        end
    end
end;
return t;

Let's examine the running time of the above algorithm. It takes time linearly in the size of the blocks to check whether 2 blocks are uniteable or not. This gives

$$
\begin{array}{llll}
 & \sum_{m=0}^{n-1}(\sum_{i=0}^{2^{n-m}}(\sum_{j=i+1}^{2^{n-m}}(2^m))) & = & \sum_{m=0}^{n-1}(\sum_{i=0}^{2^{n-m}}((2^{n-m}-i)\cdot 2^m)) \\
= & \sum_{m=0}^{n-1}(\sum_{i=0}^{2^{n-m}}(2^n - i\cdot 2^m)) & = & \sum_{m=0}^{n-1}(2^{n-m}2^n - 2^m\sum_{i=0}^{2^{n-m}}(i) \\
= & \sum_{m=0}^{n-1}(2^{2n-m} - 2^m(2^{n-m}(2^{n-m}+1)/2)) & \approx & \sum_{m=0}^{n-1}(2^{2n-m} - 2^m(2^{2\cdot(n-m)-1})) \\
= & \sum_{m=0}^{n-1}(2\cdot 2^{2n-m-1} - 2^{2n-m-1}) & = & \sum_{m=0}^{n-1}(2^{2n-m-1}) \\
= & 2^n\sum_{m=0}^{n-1}(2^{(n-1)-m}) & = & 2^n\sum_{m=0}^{n-1}(2^m) \\
= & 2^n(2^n-1) & \approx & (2^n)^2
\end{array}
$$

A quadratic running time is not acceptable when it has to work for the inputs of size 2^{23} that are produced by 4 men endgames with pawns. Preferably it should be feasible even for the tables generated by 5 men endgames and this extra piece adds a factor of 64!

7.3.1 Improving running time

The algorithm described above can be improved by cutting down on the number of blocks that need to be compared. To illustrate the idea let's consider an algorithm that finds duplicates in an array by doing all $\theta(n^2)$ comparisons. This algorithm can be improved to an $O(n \cdot log(n))$ algorithm by first doing a sort of the entries and then only compare neighbour entries.

Unfortunately this optimisation can't be translated directly to our domain, since we can't impose a useful ordering on the blocks. If we define $x \cong y \Leftrightarrow$ x and y can be made identical by a mapping of don't cares, then our problem basically is that this relation is not transitive as the example below shows.

$$
\begin{array}{rrcl}
 & [*,2,1,2] & \cong & [1,2,*,2] \\
 & [1,2,*,2] & \cong & [1,*,2,2] \\
\text{but} & [*,2,1,2] & \ncong & [1,*,2,2]
\end{array}
$$

The solution presented below first performs a lexicographical ordering of the blocks and then recursively compares the blocks one entry at a time. The trick is that some blocks sharing a prefix are placed in succession of each other due to the ordering. Hence the comparisons with other blocks is only needed once for the entire group up to this prefix. Before presenting the refined algorithm, it is convenient to define the cost of making 2 blocks identical:

$$
C(b_1, b_2) = \begin{cases} k & \text{if } b_1 \text{ and } b_2 \text{ can be made identical by mapping } k \text{ wildcards} \\ \infty & \text{otherwise} \end{cases}
$$

The subroutine presented below should be viewed as a replacement for the 2 inner for loops in the simple implementation. `ComputeMatches` returns a list of all pairs of blocks that can be unified by a mapping of don't cares. Its running time is highly dependent on the number of pairs it finds.

Algorithm: `ComputeMatches`

Input: b consisting of lexicographically sorted blocks of size n.
    Integers i_0, i_1, j_0, j_1 and p such that:
        The blocks $b[i_0 \ldots i_1[$ agree on the first p entries
        The blocks $b[j_0 \ldots j_1[$ agree on the first p entries
    Integer $c = C(b[i][0 \ldots p[,\ b[j][0 \ldots p[)$
        (for arbitrary $i_0 \leq i < i_1$ and $j_0 \leq j < j_1$)

Output: $\{(i, j, c) \mid 0 < C(b[i], b[j]) = c < \infty\}$

Set p' to the least integer such that $b[i_0 \ldots i_1[$
    or $b[j_0 \ldots j_1[$ are no longer identical on this prefix.
I.e. set p' to the least value such that
$$b[i_0][p'] \neq b[i_1 - 1][p'] \quad \vee \quad b[j_0][p'] \neq b[j_1 - 1][p']$$
If no such value less than the block size n exists then begin
    // $b[i_0 \ldots i_1[$ are identical blocks and so are $b[j_0 \ldots j_1[$,
    // and not just on the first p entries but everywhere.
    If $C(b[i_0][p \ldots n[, b[j_0][p \ldots n[) = k < \infty$ and $c + k > 0$ then
        return $\{(i, j, c + k) \mid i_0 \leq i < i_1 \wedge j_0 \leq j < j_1\}$
    Else return $\{\}$
End else begin
    // Divide the 2 groups of blocks into a minimal number of new
    // groups such that each group will agree on the first p' entries.
    Let $i'_0 < \cdots < i'_k$ and $j'_0 < \cdots < j'_l$ be such that
        $\forall 0 \leq a, b \leq k$: $b[i'_a]$ and $b[i'_b]$ are identical on the first p' entries
        $\forall 0 \leq a, b \leq l$: $b[j'_a]$ and $b[j'_b]$ are identical on the first p' entries
        k and l are minimal satisfying the above.
    Let $c_{a,b} = c + C(b[i_a][p \ldots p'[,\ b[j_b][p \ldots p'[)$
    Return $\bigcup_{\substack{0 \leq a < k, \\ 0 \leq b < l, \\ c_{a,b} < \infty}}$ `ComputeMatches`$(i'_a,\ i'_{a+1},\ i'_b,\ i'_{b+1},\ c_{a,b},\ p')$
End;

The simple implementation combined 2 blocks whenever a match was found. The order of which the blocks are combined is crucial for how well the algorithm performs. Consider the small example below:

`00** **11 000* 00*0 *111 1*11`

If we start by unifying `00**` with `**11` then the we will eventually reach the solution with 3 different kinds of blocks:

`0011 0011 0000 0000 1111 1111`

An optimal solution only has the 2 kinds of blocks `0000` and `1111`. A heuristic is used in the algorithm below to determine this order. It works by combining the 2 blocks that require remapping a minimal number of don't cares.

Because $\cong$ is not a transitive relation, we have to check whether 2 blocks are still combinable after a number of other blocks have been combined.

Algorithm: `TopDownMapping`
Input: $t[0 \ldots 2^n[$ of integers — 0 represents a don't care
Output: $t[0 \ldots 2^n[$ with no zero valued entry
For $m = n - 1$ down to 0 do begin
    Divide t into 2^{n-m} blocks $b[0 \ldots 2^{n-m} - 1][0 \ldots 2^m - 1]$.
    Sort the blocks b using a lexicographic ordering
        (0 represents a don't care, all other values are strictly positive).
    Remove any duplicates from b.
    Let $m =$ `ComputeMatches`(b); // $m = \{(i, j, c) \mid 0 < C(b[i], b[j]) = c < \infty\}$
    While m is not empty do begin
        Remove a (i, j, c) with least c from m
        Let $c' = c(b[i], b[j])$
        If $c' = c$ then
            Make $b[i]$ and $b[j]$ identical by reassigning c don't cares.
        Else if $0 < c' < \infty$ then
            Add (i, j, c') to m
    End
    Copy the changes back to table
        - this includes the duplicates that were removed.
End
Return t;

A problem with the above implementation is that even if `ComputeMatches` returns far less than $(2^{n-m})^2$ pairs, this number may still be more than what is feasible with respect to the memory usage. This problem has been solved by giving `ComputeMathces` an extra argument specifying an upper bound on the allowed cost of the matches found. If the upper bound given results in too many matches, then it is simply reduced. Once `ComputeMathces` has been called without the need of rejected any matches due to a too high cost, we have identified all the unifiable pairs.

The test results below show that applying the top-down mapping clearly improves the compression of the OBDDs. The gain is proportional with how big a fraction of the entries are don't cares, so it varies from endgame to endgame. The improvement is much greater if we cut down the information to merely saying whether the position is won, drawn or lost. The reason has already been illustrated with the KBKN example but is even more clear with the endgame KRK for white to move. Every position is a win for white. If we assign all broken positions the value won the OBDD will become trivial.

With distance to mate information the simple mapping only makes a marginal improvement. With only win/draw/loss information one value is usually much more dominant than the others and the improvement by the simple mapping becomes significant. However it is still clearly outperformed by the top-down mapping.

Using full distance to mate information

Mapping used	KPK	KQK	KRK	KQKR
No mapping	25811+19331	7496+9878	10386+12696	1176788+1205010
Simple mapping	24935+17983	6736+9274	10046+12196	1127220+1194410
Top-down mapping	**19567**+**15307**	**4584**+**6954**	**7598**+**10092**	**849764**+**1093770**

Representing only whether a position is won, drawn or lost

Mapping used	KPK	KQK	KRK	KQKR
No mapping	2148+2248	1955+1756	1291+1380	24977+30013
Simple mapping	1540+1892	**551**+1340	**551**+984	7161+26353
Top-down mapping	**1164**+**1336**	**551**+**1092**	**551**+**704**	**4313**+**14693**

7.4 Modifying the heuristic used by the top-down mapping

In the top-down mapping algorithm just presented, a heuristic was used to determine the order in which to unify the blocks. Given 2 matching blocks b_1 and b_2 a cost function simply computed the number of don't cares that would have to be mapped in b_1 and b_2. The applied heuristic was to unify the pairs of blocks with lowest cost first.

I later realised that this heuristic behaved in an unintended way. In the example it would unify the two first blocks to `01**`.

`0*** *1** 1111 0000`

Instead we could get rid of the two first for free by unifying `0***` with `0000` and `*1**` with `1111`. By "for free" I refer to the fact that don't cares are only mapped in one of the blocks which is then made identical to the other.

Hence, it would be more appropriate to define the cost of unifying two blocks based on how much more "restricted" the unification of the 2 blocks is compared to each of them. The new cost function C' reflects this:

$$C'(b_1, b_2) = \begin{cases} k_1 \cdot k_2 & \text{if } b_1 \text{ and } b_2 \text{ can be made identical by mapping } k_1 \\ & \qquad\quad \text{don't cares in } b_1 \text{ and } k_2 \text{ in } b_2. \\ \infty & \text{otherwise} \end{cases}$$

The top-down mapping algorithm using the above cost function will be referred to as top-down'.

7.4.1 5 men endgames

The hack that was described to limit the number of pairs returned by `ComputeMatches` made the algorithm feasible to use for the table sizes of up to 2^{23} produced by 4 men endgames.

However, this approach is not feasible for 5 men endgames. Therefore a slightly different version of the top-down algorithm has been implemented. Instead of identifying all unifiable pairs at once, it focuses at one block at a time according to their lexicographic ordering. It identifies all possible unifications for this block and then applies them in the order specified by the cost function C'.

The result is that the order in which the blocks are unified is no longer the global best one in terms of the cost function. It is only the best order locally to the block being examined. The good news is that the number of unifiable pairs blocks is far smaller when one of the blocks has been fixed.

To avoid making too costly unifications I have tried to specify an upper bounds on the allowed cost. For each block size the algorithm is repeated a few times where the allowed cost is increased for each run. When the block size is n, the cost function C' produces values in $[0 \dots n^2[\bigcup \{\infty\}$. A cost of 0 gives a free unification just like in the example showed earlier. "Modified, t-d' $\langle 0, n, \infty \rangle$" will refer to this algorithm with 3 repeated runs of allowed costs 0, n and ∞.

7.4.2 Results

The table below compares the performance of the different versions of the top-down mapping algorithm. In the column "KQKR time" the time used by the algorithms is shown. Without any mapping of don't cares the construction of the KQKR OBDDs take 4 minutes and 35 seconds. This time has been subtracted from the times shown.

Algorithm version	KPK	KQK	KRK	KQKR	KQKR time
Top-down	19567+15307	4584+6954	7598+10092	849764+1093770	4m09s
Top-down'	19479+**15259**	4532+6806	7606+10064	**824888**+1090350	4m29s
Modified t-d', $\langle \infty \rangle$	19563+15491	4556+6998	7666+10196	861440+1098842	4m24s
Modified, t-d' $\langle 0, \infty \rangle$	**19467**+15419	4552+6978	7630+10144	845620+1093158	3m52s
Modified t-d', $\langle 0, n, \infty \rangle$	**19467**+15399	**4528**+6982	**7586**+10156	842228+1093330	5m55s
Modified t-d', $\langle 0, n \rangle$	19875+15439	4688+6766	7786+**9660**	836124+**1077626**	4m29s
Modified t-d', $\langle 0, n, n^{3/2} \rangle$	**19467**+15427	4572+**6662**	7666+10100	831132+1093370	6m21s

The above results are rather inconclusive about which version performs best. The differences are however quite small. I have therefore chosen to use the version "Modified t-d', $\langle 0, n \rangle$" in the implementation.

7.5 Bad performing example for the top-down algorithm

Even though the top-down mapping algorithm seems to work well in practice it is possible to construct simple inputs on which it performs extremely poor. A prime example is the 256 entries table shown below. Notice that the heuristics tested earlier don't play any role here.

```
***0 ***1 ***0 ***1 ***0 ***1 ***0 ***1 ***0 ***1 ***0 ***1 ***0 ***1 ***0 ***1
**0* **0* **1* **1* **0* **0* **1* **1* **0* **0* **1* **1* **0* **0* **1* **1*
*0** *0** *0** *0** *1** *1** *1** *1** *0** *0** *0** *0** *1** *1** *1** *1**
0*** 0*** 0*** 0*** 0*** 0*** 0*** 0*** 1*** 1*** 1*** 1*** 1*** 1*** 1*** 1***
```

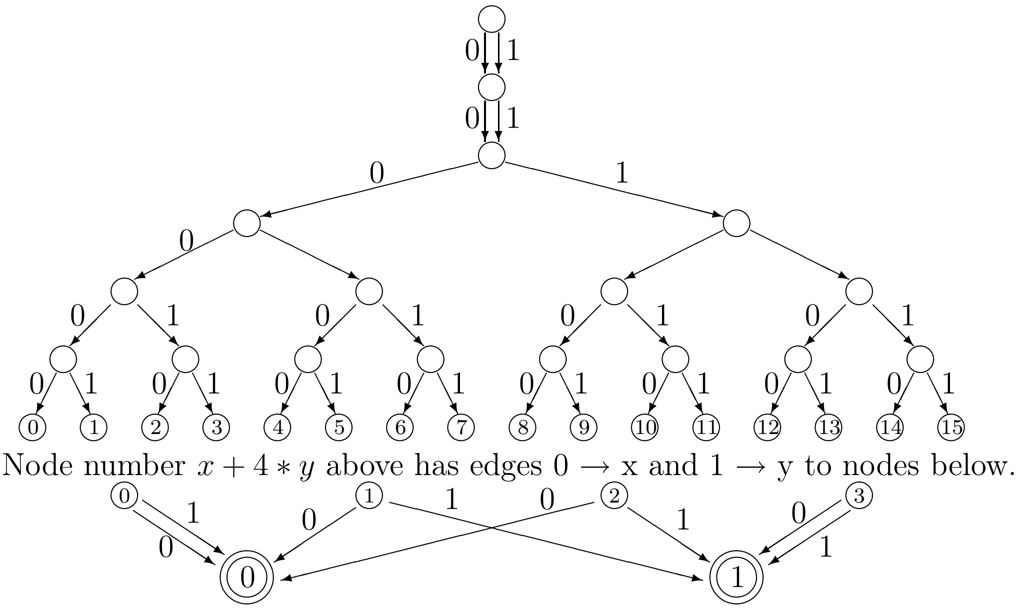
First the two 128 sized blocks are made identical:

```
*0*0 *0*1 *0*0 *0*1 *1*0 *1*1 *1*0 *1*1 *0*0 *0*1 *0*0 *0*1 *1*0 *1*1 *1*0 *1*1
0*0* 0*0* 0*1* 0*1* 0*0* 0*0* 0*1* 0*1* 1*0* 1*0* 1*1* 1*1* 1*0* 1*0* 1*1* 1*1*
*0*0 *0*1 *0*0 *0*1 *1*0 *1*1 *1*0 *1*1 *0*0 *0*1 *0*0 *0*1 *1*0 *1*1 *1*0 *1*1
0*0* 0*0* 0*1* 0*1* 0*0* 0*0* 0*1* 0*1* 1*0* 1*0* 1*1* 1*1* 1*0* 1*0* 1*1* 1*1*
```

Next, all 64 sized blocks are made identical:

```
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
```

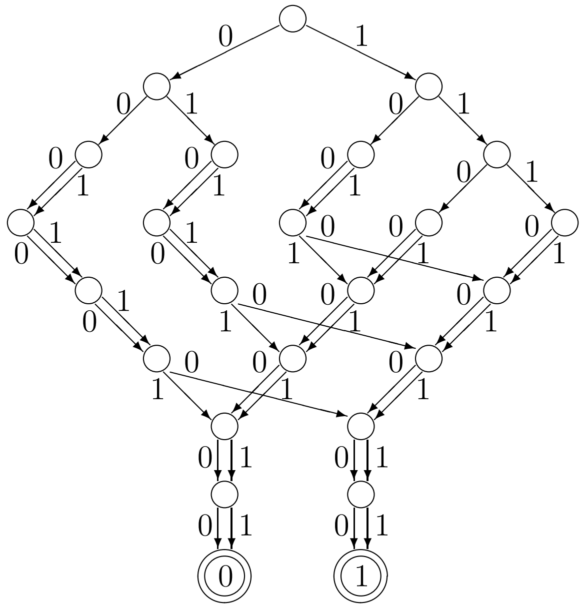
Consequently, the resulting OBDD will include nodes representing each of the possible 4 bit integers. This is shown below.



For comparison, an optimal mapping of the wildcards is

```
0000 1111 0000 1111 0000 1111 0000 1111 0000 1111 0000 1111 0000 1111 0000 1111
0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111 0000 0000 1111 1111
0000 0000 0000 0000 1111 1111 1111 1111 0000 0000 0000 0000 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111 1111 1111 1111 1111
```

and the equivalent OBDD looks like this:



7.5.1 Generalising

The above example is easily generalised to arbitrary large table sizes on the form $N = (2^n)^2 \cdot 2^{(2^n)}$. The table generated from the top-down mapping of such an input would consist of 2^n identical parts, each listing the binary representations of all numbers less than $2^{(2^n)}$. The equivalent OBDD tightly reflects this and it is easy to compute its size. Each addend in the first expression counts the number of nodes in a layer. There is in total $\log((2^n)^2 \cdot 2^{(2^n)}) + 1 = 2n + 2^n + 1$ layers (let's include the leaf nodes in the count even though they are not included in our representation of an OBDD). Size of top-down mapped OBDD:

$$\begin{array}{rcl} s' & = & \sum_{i=1}^{n} 1 + \sum_{i=0}^{i=2^n-1} 2^i + \sum_{i=0}^{i=n} 2^{2^i} \\ & = & n + (2^{2^n} - 1) + \sum_{i=0}^{i=n} 2^{2^i} \\ & = & (2^{2^n} + (n-1)) + (2^{2^n} + \sum_{i=0}^{i=n-1} 2^{2^i}) \\ & \geq & 2^{2^n+1} \end{array}$$

Size of optimal mapped OBDD:

$$\begin{array}{rcl} s & = & \sum_{i=0}^{n} 2^i + \sum_{i=2}^{2^n+1} i + \sum_{i=1}^{n} 2 \\ & = & (2^{n+1} - 1) + ((2^n + 1) \cdot ((2^n + 1) + 1)/2 - 1) + (2n) \\ & = & 2^{n+1} + (2^{2n} + 3 \cdot 2^n + 2)/2 + 2n - 2 \\ & = & 2^{n+1} + 2^{2n-1} + 3 \cdot 2^{n-1} + 2n - 1 \\ & = & 2^{2n-1} + 7 \cdot 2^{n-1} + 2n - 1 \end{array}$$

For $n \geq 4$ we can make the estimation $s = 2^{2n-1} + 7 \cdot 2^{n-1} + 2n - 1 < 2^{2n}$

7.5.2 Measuring performance

The worst case performance is usually defined as the ratio between the size of the solution computed by the algorithm and the size of the optimal solution. This is a minimisation problem and we therefore seek a ratio as close to 1 as possible. However, the example from before shows us that our algorithm has a worst case performance of at least

$$\frac{s'}{s} \geq \frac{2^{2^n+1}}{2^{2n}} = 2^{2^n-2n+1}$$

For sufficiently large n (i.e. $n \geq 6$) it holds that

$$s = 2^{2n-1} + 7 \cdot 2^{n-1} + 2n - 1 \leq (2^n - 2n + 1)^2$$

We conclude that the solution produced by our algorithm ratio is exponentially bigger than optimal

$$\frac{s'}{s} \geq 2^{2^n-2n+1} \geq 2^{\sqrt{s}}$$

The inapproximability ratio can also be described in terms of the size of the input N. For any $\epsilon > 0$ it holds for sufficient large n (or equivalently, N) that

$$\frac{s'}{s} \geq 2^{2^n-2n+1} \geq 2^{(2^n+2n)(1-\epsilon)} = \left(2^{2^n+2n}\right)^{1-\epsilon} = N^{1-\epsilon}$$

7.5.3 What makes the algorithm perform so poorly?

In the last example given, the top-down algorithm performed very poorly because the algorithm greedily used all don't cares to unify the few blocks of size 128 and 64. Minimizing the number of nodes in one layer can hence have a negative effect on the other layers — the optimal mapping have no unified blocks of size 128 or 64.

Therefore an algorithm producing a minimal OBDD can't be made by optimising one layer at a time. However, even if this was the case, we would still face an NP-complete problem. Actually it is quite easy to show that the problem of minimizing the number of nodes in one layer is NP-hard. The intuitively harder problem of minimizing the size of the entire OBDD is a lot more difficult to prove NP-hard. This result is interesting in its own right as it generalises the other hardness results shown for OBDD minimization using don't cares. Therefore it will be described separately in the next section.

Also, an inapproximability result is shown stating that OBDD minimization using don't cares can't be approximated within $N^{\frac{1}{4}-\epsilon}$ unless P=NP or NP $\subseteq$ coRP. Hence, it wouldn't help much to improve the algorithm presented here, such that it could do well on the example that otherwise killed its performance. There would always be some other problematic instance.

8 Minimisation of OBDDs using don't cares is NP-hard

In [25] and [26] it is shown that the problem of constructing a minimal OBDD that is consistent with an incompletely specified boolean function is NP-hard. Also it is shown that approximation within N^{ϵ} is impossible if P $\neq$ NP.

In concrete problems, an instance typically have to represented in some compact form. Hence in [25] the hardness is shown with the input modelled either as an OBDD taking values from $\{0, 1, *\}$, where $*$ stands for "don't care" or equivalently as 2 boolean valued OBDDs with one defining the don't care set. In [26] the input is given as the 2 disjoint sets of the positive and negative entries.

If the input model is changed to a truth table then it is far from obvious that the problem remains NP-hard. It may require exponentially more space to represent an input as a truth table than in one of the other input formats. The proofs presented in [25] and [26] shows the hardness by the using exactly such instances. Hence, these proofs will not work with the input modelled as a truth table. On the other hand it is a simple observation to see that once the NP-hardness has been shown with the truth table model, it immediately gives the NP-hardness of the other versions of the problem.

This section presents a proof showing that the minimisation of OBDDs using don't cares is NP-hard when the input is given as a truth table. It is based on a reduction from minimum clique partition — or equivalently, the graph colouring problem.

A recent result from [37] states that under the assumption NP $\subsetneq$ coRP the graph colouring problem is not approximable within $N^{1-\epsilon}$ for any ϵ. The reduction used here translates this into a $N^{\frac{1}{4}-\epsilon}$ inapproximability result for the problem of minimising an OBDDs using don't cares.

8.1 Definitions

The OBDDs used in this section are a bit more restricted in that they are boolean valued. This makes the set of instances smaller and the result slightly more general. We still require that the OBDDs are complete, i.e. that all variables b_i's are tested on each path from the root to a leaf.

We say that a string s is covered by s' if they have the same length and $\forall i : s[i] \neq * \Rightarrow s[i] = s'[i]$. A set of strings S is covered by s' if all $s \in S$ are covered by s'. The first definition below is taken from [32].

Minimum Clique Partition (MCP):
Instance: Graph $G = (V, E)$
Solution: A clique partition for G, i.e., a partition of V into disjoint subsets $V_1, V_2, \ldots, V_k$ such that, for $1 \leq i \leq k$, the subgraph induced by V_i is a complete graph.

Measure: Cardinality of the clique partition, i.e., the number of disjoint subsets V_i.

Minimum String Cover (MSC):
Instance: Set S of equal sized strings in the alphabet $\{0, 1, *\}$
Solution: A partition of S into disjoint subsets $S_1, S_2, \ldots, S_k$ such that each subset has a cover.
Measure: Number of disjoint subsets S_i.

Minimum Consistent OBDD (MCO):
Instance: String t of size 2^k with entries from $\{0, 1, *\}$
Solution: An OBDD representing a cover of t
Measure: Size of the OBDD with measure being the number of nodes.

The decision variants of the above problems L_{MCP}, L_{MSC} and L_{MCO} are defined the obvious way. With the definitions in place, let's phrase the result formally.

Theorem 1: *L_{MCO} is NP-hard. Furthermore, under the assumption NP $\subsetneq$ coRP MCO can't be approximated within $N^{\frac{1}{4}-\epsilon}$ for any $\epsilon > 0$, where N is the size of the table.*

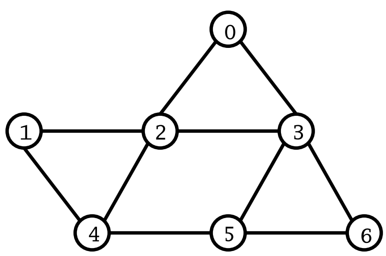
It will be shown by combining a reduction from MCP to MSC with a reduction from MSC to MCO.

8.2 MCP reduces to MSC

The reduction r' from L_{MCP} to L_{MSC} that will be described here does not change the size of an optimal solution:

$$\forall G, k : (G, k) \in L_{MCP} \Leftrightarrow (r'(G), k) \in L_{MSC}$$

The reduction will be demonstrated from an example. To the left we are given our graph for which we wish to find a minimum clique partition. To the right this graph has been converted to a kind of adjacency matrix. The diagonal contains 1's. Otherwise the entry $M_{i,j}$ contains a $*$ if $(i, j) \in E$ and 0 otherwise.



M	0	1	2	3	4	5	6
0	1	0	$*$	$*$	0	0	0
1	0	1	$*$	0	$*$	0	0
2	$*$	$*$	1	$*$	$*$	0	0
3	$*$	0	$*$	1	0	$*$	$*$
4	0	$*$	$*$	0	1	$*$	0
5	0	0	0	$*$	$*$	1	$*$
6	0	0	0	$*$	0	$*$	1

By considering each row as a string we have the set S of strings in the alphabet $\{0, 1, *\}$.

Now the claim is that G has a clique partition $\{V_i\}$ of size at most k iff there exists an unification S' of S with size at most k.

Proof:

- $\Rightarrow$ We are given a clique partition $\{V_i\}_i$.
  Construct $s'_i[j] = \begin{cases} 1 & \text{if } v_j \in V_i \\ 0 & \text{otherwise} \end{cases}$
  Clearly s'_i covers the strings $\{s_j | v_j \in V_i\}$, and hence $\bigcup\{s'_i\}$ covers the set S.

- $\Leftarrow$ We are given a set S' of strings covering the set S.
  To get a clear relationship to the MCP problem, we first have to prune this set, while maintaining that it is a covering of S. First replace each $s'_i[j] = *$ with $s'_i[j] = 0$. Then repeat the following as long as progress is made.

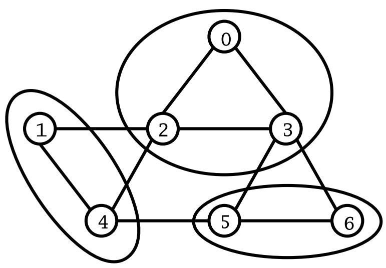
  - If $s'_i[k] = 1$ and $s'_j[k] = 1$ for $i \neq j$ then set $s'_i[k] = 0$
  - If $\forall j : s'_i[j] \neq 1$ then throw away s'_i (this will make our clique partition even smaller).

  By the above 2 steps, we are guaranteed that each s_i is covered by a unique s'_j and that each s'_i covers at least one s_j.

  Now, construct $V_i = \bigcup\{v_j | s'_i[j] = 1\}$. Suppose that V_i is not a clique, then $\exists j, k \in V_i$ such that $(j, k) \notin E$. By definition then $s_j[k] = 0$ and $s_k[k] = 1$. Also, the pruning gives us that s_j and s_k are uniquely covered by s'_i. But this is not possible as they differ on entry k.

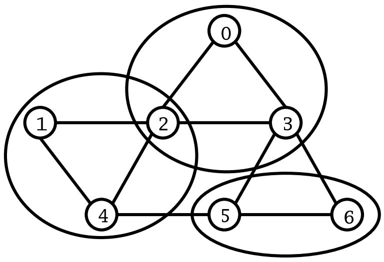
Continuing the example, an optimal solution is $V_0 = \{0, 2, 3\}, V_1 = \{1, 4\}, V_2 = \{5, 6\}$. Hence we have a cover s'_0 for s_0, s_2 and s_3 (and similar for s'_1 and s'_2).

$$s'_0 = \texttt{1011000},\ s'_1 = \texttt{01*0100},\ s'_2 = \texttt{000*011}$$



$s_0 = $ 10**000	is covered by s'_0
$s_1 = $ 01*0*00	is covered by s'_1
$s_2 = $ **1**00	is covered by s'_0
$s_3 = $ *0*10**	is covered by s'_0
$s_4 = $ 0**01*0	is covered by s'_1
$s_5 = $ 000**1*	is covered by s'_2
$s_6 = $ 000*0*1	is covered by s'_2

When $s'_1 = $ 01*0100 is a cover of s_1 and s_4, so is 0100100 and 0110100. By choosing $s'_1 = $ 0110100, the direct interpretation of $\{s'_i\}_i$ would give us



I.e. the sets V_i would no longer be disjoint. This is why the pruning was needed.

8.3 MSC reduces to MCO

Since we have established the NP-hardness of MSC by a reduction from MCP, the reduction from MSC to MCO only need to work in the case of 2^n strings each of length 2^n. The reason is that if we are given a graph G with the number of vertices not being a power of 2, then we can construct a graph G' by simply adding maximum connected vertices until the number reach a power of 2. This won't increase the size of the minimum clique partition as the added vertices are free to join any clique. Obviously, it wont decrease the size either. Let r'' be a reduction that expands a graph G to G' in the way described above.

$$\forall G, N : (G = (V, E), N) \in L_{MCP} \quad \Leftrightarrow \quad (r''(G) = G' = (V', E'), N) \in L_{MCP}$$
$$|V'| \leq 2|V|$$

In the description of the reduction from MSC to MCO, $n = 2^k$ will denote the number of strings s_i in the set S.

8.3.1 Goal

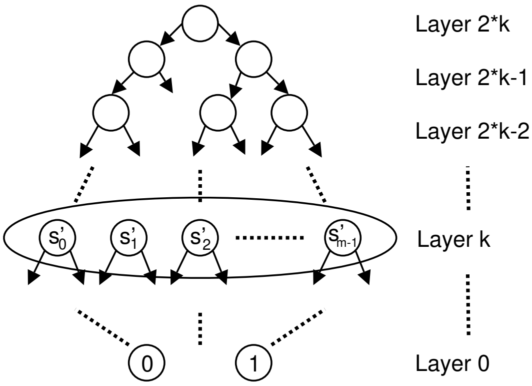
To show that MSC reduces to MCO, we must find polynomial time computable r, f such that

$$\forall S(|S| = 2^k \wedge \forall i : |s_i| = 2^k) \Rightarrow$$
$$[\forall N : (S, N) \in L_{MSC} \iff (r(S), f(N)) \in L_{MCO}]$$

First I will present an obvious way of constructing the reduction r and argue why this doesn't work. Then I'll show how to modify the reduction, so that the bug is fixed.

8.3.2 First attempt

Define $r(S) = s_0 s_1 \ldots s_{n-1}$. Consider how a minimal OBDD for this table could look like:



In layer k, the nodes $\{s'_i\}_i$ represent a cover for S. This follows from the fact that each string $s \in S$ has size $n = 2^k$.

Now, the problem is that minimising the size of the entire OBDD does not imply that the number of nodes in each individual layer is minimised. Hence the cover given by layer k is not necessarily optimal.

8.4 Revised reduction

We can modify the reduction such that in a minimal OBDD, the covers are no longer represented by single nodes, but by subtrees. By making these trees big enough compared to the other parts of the OBDD, we can enforce that their number will necessarily be minimal in a minimal OBDD.

The new reduction looks like this

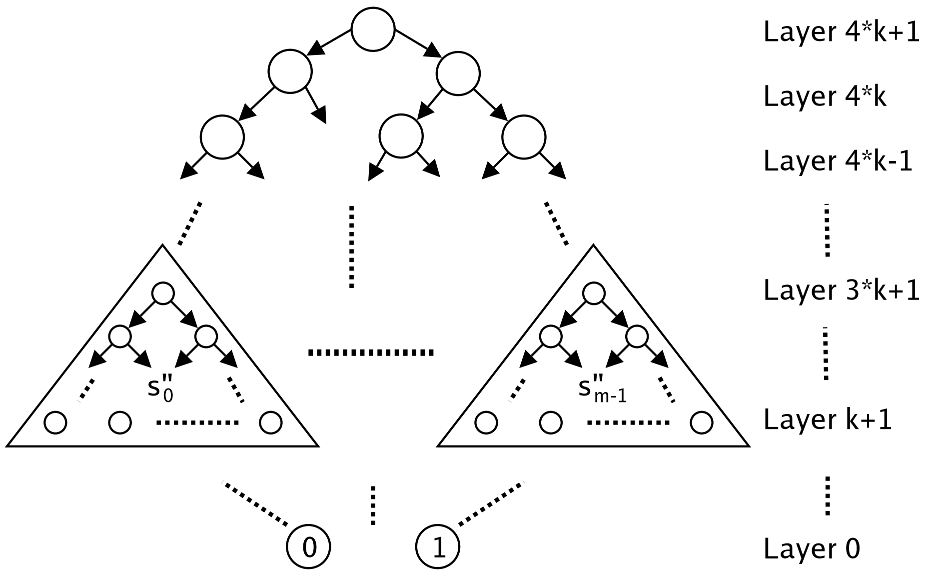
$$\begin{aligned} r(S) = {} & b_0 s_0 b_1 s_0 b_2 s_0 \dots b_{n^2-1} s_0 \\ & b_0 s_1 b_1 s_1 b_2 s_1 \dots b_{n^2-1} s_1 \\ & \dots \\ & b_0 s_{n-1} b_1 s_{n-1} b_2 s_{n-1} \dots b_{n^2-1} s_{n-1} \end{aligned}$$

where $|b_i| = n = |s_i|$ and $b_i = b(k, i, 2k)$. The function $b(x, y, z)$ returns a binary representation of the number y satisfying $0 \leq y < 2^z$ and padded with 0's between the digits to make its length 2^x. Formally it is defined as

$$b(x,y,z) = \begin{cases} '0' & \text{if } x = y = 0 \\ '1' & \text{if } x = 0 \wedge y = 1 \\ b(x-1, \lfloor y \cdot 2^{-z''} \rfloor, z'') b(x-1, \lfloor y \bmod 2^{z'} \rfloor, z''), & \\ \qquad \text{where } z' = \lfloor \frac{z}{2} \rfloor \text{ and } z' + z'' = z & \text{otherwise} \end{cases}$$

As an example $b(4, y, 5) = 00000y_4 0 y_3 000 y_2 0 y_1 0 y_0$, with $y_4 y_3 y_2 y_1 y_0$ being the usual binary representation of y.

Now, a minimal OBDD representing $r(S)$ will be of the form



where $s''_i = b_0 s'_i b_1 s'_i b_2 s'_i \dots b_{n^2-1} s'_i$.

8.4.1 Estimating the size of the OBDD

We need to make some estimates on the size of the minimal OBDD. This will be split up in 3 parts:

$$\begin{aligned} X_{low} &: \text{Denotes the number of nodes in the lowest } k+1 \text{ layers.} \\ X_{high} &: \text{Denotes the number of nodes in the highest } k \text{ layers.} \\ X_{subtrees} &: \text{The remaining part consisting of the subtrees.} \end{aligned}$$

It is our goal to show that the difference caused by adding or removing a single subtree outweighs the contributions made by X_{low} and X_{high}. Let's begin with X_{high}, which is rather easy

$$0 < X_{high} \le \sum_{i=3k+2}^{4k+1} 2^{4k+1-i} = \sum_{i=0}^{k-1} 2^i = 2^k - 1 < n.$$

In the estimate of X_{low} we first calculate the contributions of the blocks b_i with i ranging from 0 to n^2-1. Call this X_b. Looking at the definition of b, we see that the nodes from layer $k-m$ define exactly the substrings on the form $b(k-m, i, \lceil \frac{2k}{2^m} \rceil)$ for $0 \le i < 2^{\lceil \frac{2k}{2^m} \rceil}$. Notice that at layer k this gives one node for each $b_i = b(k, i, 2k)$.

$$\begin{aligned} X_b &= \sum_{i=0}^{k} 2^{\lceil \frac{2k}{2^{k-i}} \rceil} \ge 2^{\lceil \frac{2k}{2^{k-k}} \rceil} = 2^{2k} = (2^k)^2 = n^2 \\ X_b &= \sum_{i=0}^{k} 2^{\lceil \frac{2k}{2^{k-i}} \rceil} = 2^{\lceil \frac{2k}{2^{k-k}} \rceil} + 2^{\lceil \frac{2k}{2^{k-(k-1)}} \rceil} + \sum_{i=0}^{k-2} 2^{\lceil \frac{2k}{2^{k-i}} \rceil} \\ &\le 2^{2k} + 2^k + (k-1)2^{\lceil \frac{2k}{2^{k-(k-2)}} \rceil} \le n^2 + n + k2^{\frac{k}{2}+1} \\ &\le n^2 + n + 2k(2^k)^{\frac{1}{2}} \le n^2 + 2k\sqrt{n} \\ &\le n^2 + 2n \end{aligned}$$

The last inequality holds when $\sqrt{n} \geq 2k \iff n \geq 4 \cdot \log(n)^2 \iff n \geq 256$. Next, the contributions of the s_i's are estimated. Call this X_s. As a lower bound we are satisfied with $0 \leq X_s$. The calculation of the upper bound is based on 2 facts: At layer k, there is at most n nodes (one for each s_i). This number can at most increase with a factor of 2 for each layer down. At layer i there can't be more than 2^{2^i} distinct nodes, as the OBDD contains at most 2 leaves. This gives:

$$\begin{aligned} X_s &\leq \textstyle\sum_{i=0}^{k} \min\{n \cdot 2^{k-i}, 2^{2^i}\} \leq 2^{2^0} + 2^{2^1} + \sum_{i=2}^{k} n \cdot 2^{k-i} \\ &= 2 + 4 + n \textstyle\sum_{i=0}^{k-2} 2^i = 6 + n \cdot (2^{k-1} - 1) \\ &= 6 + n \cdot (\frac{n}{2} - 1) \leq n^2 \end{aligned}$$

The last inequality holds whenever $n \geq 3$ (for integer n). Now we are able to compute X_{low}. In the lower estimate keep in mind that the nodes representing parts of s_i may be shared with nodes representing parts of b_j, hence the use of max.

$$\begin{aligned} X_{low} &\geq \max(X_b, X_s) \geq \max(n^2, 0) = n^2 \\ X_{low} &\leq X_b + X_s \leq (n^2 + 2n) + (n^2) = 2n^2 + 2n \end{aligned}$$

The reduction creates a table including all the pairs $\{b_i s_j\}_{i,j}$. Hence for each $0 \leq i < n^2$, layer k must contain a set of nodes representing $\{b_i s'_j\}_j$ with $S' = \{s'_j\}_j$ being a cover of S. Furthermore, as the b_i's are different and without don't care entries, these sets must be disjoint. This gives $n^2 K$ as a lower limit for the number of nodes in layer k. Remember that K is the size of the minimal clique partition of G'. As the number of nodes decrease by a factor of 2 for each layer we ascend, we have

$$\begin{aligned} X_{subtrees} &= \textstyle\sum_{i=k+1}^{3k+1} \frac{n^2 K}{2^{i-(k+1)}} = n^2 K \sum_{i=0}^{2k} \frac{1}{2^i} \\ &= n^2 K \cdot (2 - \frac{1}{2^{2k}}) = n^2 K \cdot (2 - \frac{1}{n^2}) = 2n^2 K - K \end{aligned}$$

8.5 Proof

With r given earlier and $f(K) = (2K + 3)n^2$, we are now ready to show that

$$\forall S(|S| = 2^k \wedge \forall i : |s_i| = 2^k) \Rightarrow [(S, K) \in L_{MSC} \iff (r(S), f(K)) \in L_{MCO}]$$

Proof:

- $\Rightarrow$ S has a cover of size K. Construct $r(S)$ but with each s_i replaced by its cover (and with any remaining don't care symbols replaced by 0). The minimal OBDD will have have one subtree representing $b_0 s'_i b_1 s'_i \ldots b_{n^2-1} s'_i$ for each s'_i in the cover S'. Its total size is:

$$\begin{aligned} \text{Size of OBDD} &= X_{subtrees} + X_{low} + X_{high} \\ &= (2n^2 K - K) + X_{low} + X_{high} \\ &\leq (2n^2 K - K) + (2n^2 + 2n) + (n) \\ &= (2K + 2)n^2 + 3n - K \leq (2K + 2)n^2 + 3n \\ &= (2K + 3)n^2 = f(K) \end{aligned}$$

The last inequality holds when $n^2 \geq 3n \iff n \geq 3$

$\Leftarrow$ We show that $(S, K) \notin L_{MSC} \Rightarrow (r(S), f(K)) \notin L_{MCO}$

S has no cover of size N, i.e. a minimal string cover has size $K' \geq K + 1$.

$$\begin{aligned}\text{Size of OBDD} &= X_{subtrees} + X_{low} + X_{high} > X_{subtrees} + X_{low} + 0 \\ &\geq (2n^2K' - K') + (n^2) \geq 2n^2(K+1) - (K+1) + n^2 \\ &\geq (2K+3)n^2 = f(K)\end{aligned}$$

By combining all the reductions we get

$$(G, K) \in L_{MCP} \iff (r''(G), K) \in L_{MCP} \iff$$
$$(r'(r''(G)), K) \in L_{MSC} \iff (r(r'(r''(G))), (2K+3)|V'|^2) \in L_{MCO}$$

The last relation holds because $r'(r''(G))$ consists of 2^k strings each of length 2^k. This concludes the proof. $\square$

8.6 Inapproximability result

The performance ratio $R_A(I)$ is defined as the ratio between the size of the solution computed by the algorithm A and the size of the optimal solution. The worst case performance ratio is defined as

$$R_A(n) := \sup\{R_A(I) \mid I \text{ input instance of size } n\}.$$

Proposition: *If MCO has an approximation algorithm A with $R_A(t) \leq \alpha(|t|)$ for all strings t, then MCP has an approximation algorithm A' with $R_{A'}(G) = \frac{5}{2}\alpha(32|V|^4)$* for all graphs $G = (V, E)$.

Proof: Choose $A' = I \circ A \circ R$, where $R = r \circ r' \circ r''$. I is an algorithm that extracts a clique partition from the OBDD produced.

Let $G = (V, E)$ be given. Let $r''(G) = (V', E')$ and $n = |V'|$. Let $P_{A'}$ be the size of the solution produced by A' on input G and $P_{\mathtt{opt}}$ be the size of an optimal clique partition of G. Let $D_{\mathtt{opt}} = \min\{|D'| \mid D' \text{ represent a cover of } R(G)\}$.

Assume $|A(R(G))| \leq 2n^2N$. Layer $k+1$ in the OBDD $A(R(G))$ can't consists of more than n^2N nodes — these are $\bigcup_{i=0}^{n^2-1}\{b_i s'_{i,j}\}_j$. There must be some i for which $|\{b_i s'_{i,j}\}_j| \leq N$. The set $\{s'_{i,j}\}_j$ gives a cover for $\{s_i\}_i$ of size at most N. Hence, given an OBDD $A(R(G))$ the algorithm I identifies such an i and then reconstructs a clique partition from the cover $\{s'_{i,j}\}_j$ as specified in the proof of the reduction from MCP to MSC. The size of is the clique partition is at most N. The reduction R produces a string of length $2n^4$. This gives

$$P_{A'} \leq \frac{|A(R(G))|}{2n^2} \leq \frac{\alpha(2n^4) \cdot D_{opt}}{2n^2}$$

We know that

$$(G, K) \in L_{MCP} \iff (R(G), (2K+3)n^2) \in L_{MCO}$$

This gives

$$\begin{array}{rcl} (G, P_{opt}) \in L_{MCP} & \Rightarrow & (R(G), (2 \cdot P_{opt} + 3)n^2) \in L_{MCO} \\ & \Rightarrow & (2 \cdot P_{\texttt{opt}} + 3)n^2 \geq D_{\texttt{opt}} \\ & \Rightarrow & P_{\texttt{opt}} \geq \frac{1}{2}(\frac{1}{n^2} D_{\texttt{opt}} - 3) \end{array}$$

Hence

$$R_{A'}(G) = \frac{P_{A'}}{P_{\texttt{opt}}} \leq \frac{\frac{\alpha(2n^4) \cdot D_{opt}}{2n^2}}{\frac{1}{2}(\frac{1}{n^2} D_{\texttt{opt}} - 3)} = \frac{1}{n^2} \frac{\alpha(2n^4) \cdot D_{opt}}{\frac{1}{n^2} D_{\texttt{opt}} - 3}$$

WLOG we can assume $P_{\texttt{opt}} \geq 2 \Rightarrow D_{\texttt{opt}} \geq 5n^2 \Rightarrow \frac{3}{5}\frac{1}{n^2} D_{opt} \geq 3$.

$$\begin{array}{c} R_{A'}(G) \leq \ldots = \frac{1}{n^2} \frac{\alpha(2n^4) \cdot D_{opt}}{\frac{1}{n^2} D_{\texttt{opt}} - 3} \leq \frac{1}{n^2} \frac{\alpha(2n^4) \cdot D_{opt}}{\frac{1}{n^2} D_{\texttt{opt}} - \frac{3}{5}\frac{1}{n^2} D_{opt}} \\ = \frac{1}{n^2} \frac{\alpha(2n^4) \cdot D_{opt}}{\frac{2}{5}\frac{1}{n^2} D_{opt}} = \frac{5}{2}\alpha(2n^4) \leq \frac{5}{2}\alpha(2 \cdot (2|V|)^4) = \frac{5}{2}\alpha(32|V|^4) \end{array}$$

The above proposition implies that any $N^{x-\epsilon}$ inapproximability result for the graph colouring gives an $N^{\frac{x}{4}-\epsilon}$ inapproximability result for MCO. In [38] it is shown that if NP $\not\subseteq$ coRP (or equivalent NP $\not\subseteq$ ZPP) then the graph colouring problem can't be approximated within $N^{1-\epsilon}$ for any $\epsilon > 0$. We conclude that under the same assumptions, MCO can't be approximated within $N^{\frac{1}{4}-\epsilon}$. Without the assumption of NP $\not\subseteq$ coRP only the weaker $N^{\frac{1}{7}-\epsilon}$ inapproximability result is known for graph colouring.

8.7 Reduced OBDD

The results also holds if we remove the requirement of the OBDD being complete. This can only reduce the size of the minimal OBDD. I will describe the few details that needs to be fixed for the proof of theorem 1 — it is equally easy to fix the inapproximability result.

We only need to check that "$\Leftarrow$" in the proof still holds. It is easy to see that the subtrees representing the b_i's won't change. The estimation of X_{low} is based on the n^2 distinct nodes representing b_i and hence still holds. With X_{high} we can do no better than $X_{high} \geq 0$, but it is easy to introduce the ">" with X_{low} instead. $\square$

9 Enumeration of board squares

In section 2 we were able to improve gzip's compression with 13% by changing the enumeration of the squares. The idea was to seek an enumeration $\langle s_i \rangle_i$ for each piece, so that the value of a position was often the same, no matter if that piece was interchanged between the squares s_i and s_{i+1}. This worked well with gzip's (Lempel-Ziv) dictionary encoding, but we will see in this section that the relation between the square enumeration and the compression ratio achieved by an OBDD is more obscure.

9.1 Algorithms for determining square enumeration

I have done experiments with a couple of algorithms. I didn't really believe that the algorithm from section 2 would improve the performance of the OBDD, so I started implementing a genetic algorithm. After realising that this gave poor results, I revisited the algorithm from section 2 and tried making some modifications to it. This didn't work well either and I gave up trying.

However, while implementing the algorithms I also tested some hand coded square enumerations and found that an enumeration following a Hilbert curve through the board gave better results.

It should be noticed that to ease implementation I only tried to modify the enumeration for the piece occupying the low 6 bits of the OBDD index (which means the non-king piece for each of the endgames KPK, KQK and KRK and one of the bishop for the KBBK endgame).

I will now present the two algorithmically attempts.

9.1.1 Genetic algorithm

The genetic algorithm (GA) has some similarity with the algorithm presented in section 2. It does a bottom up construction where it combines chains of initial length 1 to finally end up with a chain of length 64 representing the enumeration of the squares.

Each node in an OBDD represents a block of 2^n consecutive elements. This property makes it natural to require that the length of these chains are restricted to being powers of 2. Hence the outermost part of the algorithm looks like this:

> Let $C = \{\langle 0\rangle, \ldots, \langle 63\rangle\}$.
> While $|C| > 1$ do
>   Let $C =$ `combine_pairwise`(C)
> $c = C[0]$ is the square enumeration (square c_i is given new index i).

The while loop will be processed $\log(64) = 6$ times. We will see in a while that there is a direct relationship between each solution provided by `combine_pairwise` and the organisation of each of the 6 lowest layers in an OBDD representing this endgame — the better a solution `combine_pairwise comes` up with, the fewer nodes will be needed in that layer of the OBDD.

The function `combine_pairwise` is implemented as a genetic algorithm. Each genome is rather obviously being represented as a set of chains.

We say that a genome g is valid if each square number is contained in at most one chain at most once. A genome is complete if it contains each square number $\in [0 \ldots 63]$ in some chain.

With this representation it is rather easy to implement the operations of a GA such as cross over and mutation. The problem is more how to pick the weights. How big a percentage of the population should survive each generation? How many new genomes should be made by cross overs and how many should just be copies of the best individuals? In my implementation I picked some values that seemed reasonable. A pseudo code for `combine_pairwise`, `Shuffle` and `CrossOver` is shown below. The interesting part — the fitness function — will be presented underneath.

Algorithm: `combine_pairwise`
Input: Genome $C = \{c_i\}_{i=0}^{n-1}, c_i = \langle s_0^i, \ldots, s_{\frac{64}{n}-1}^i \rangle$
Output: Genome $\{c_{a_i} c_{b_i}\}_{i=0}^{\frac{n}{2}-1}$

Genomes $g[population\_size]$
$g[0] = \bigcup_{i=0}^{\frac{n}{2}-1} \{\langle c_{2i}, c_{2i+1} \rangle\}$
// $g[0]$ is valid and complete
$\forall i : g[i] = g[0]$, do a complete shuffle of $g[i]$
While best genome has not been improved for specified number of generations do
Begin
    Kill the worst 75% of the genomes:
        25% is replaced by cross overs of the best 25%
        50% is replaced by random copies of the best 25%
    For each genome excepts the best one and the cross overs do
        With 50% possibility do an n-shuffle:
            With 10% possibility, pick $n = 2$
            With 60% possibility, pick $n = 3$
            With 15% possibility, pick $n = 4$
            With 8% possibility, pick $n = 5$
            With 3% possibility, pick $n = 6$
            With 2% possibility, pick $n = 7$
            With 2% possibility, pick $n =$ all
End;
Return best genome;

Algorithm: Shuffle
Input: Genome g and integer n giving number of pairs to shuffle
Output: Genome
Choose and remove n random pairs $\{c_{a_i} c_{a_{n+i}}\}_{i=0}^{n-1}$ from g.

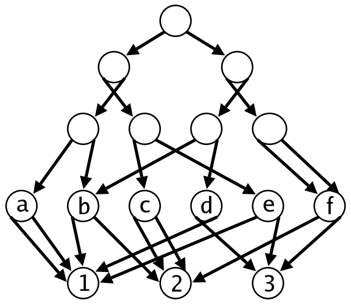
Connect the $2n$ elements $\{c_{a_i}\}_{i=0}^{2n-1}$ into n new arbitrary pairs and add then to g. Return the result. // Is valid and complete

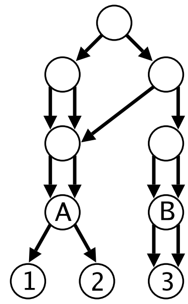
Algorithm: CrossOver
Input: Genomes g_1 and g_2
Output: A genome that is a randomised mix of the pairs in g_1 and g_2
Genome $g = \emptyset$
While $\exists c_i c_j \in g_1 \cup g_2$ such that $g = g \cup \{c_i c_j\}$ is valid
    Pick a random such $c_i c_j$ and let $g = g \cup \{c_i c_j\}$
While g not complete do
    Pick random c_i, c_j such that $g \cup \{c_i c_j\}$ is valid and let $g = g \cup \{c_i c_j\}$
Return g; // g is valid and complete

Definition of fitness function: Before I start getting into details with the definition of the fitness function, let's consider a very simplified example in which the board has only $4 \cdot 4$ squares and there is only a single piece. The figures below show the values for each possible position of the piece. The left figure uses the standard enumeration of the squares while the right figure uses an optimal enumeration in the sense that the corresponding OBDD is minimal among all enumerations. These OBDDs are shown below the figures.

	a	b	c	d	
4	1_{12}	2_{13}	3_{14}	1_{15}	4
3	2_{8}	3_{9}	2_{10}	3_{11}	3
2	1_{4}	1_{5}	1_{6}	2_{7}	2
1	2_{0}	2_{1}	1_{2}	3_{3}	1
	a	b	c	d	

	a	b	c	d	
4	1_{0}	2_{11}	3_{13}	1_{10}	4
3	2_{1}	3_{12}	2_{9}	3_{14}	3
2	1_{2}	1_{4}	1_{8}	2_{7}	2
1	2_{3}	2_{5}	1_{6}	3_{15}	1
	a	b	c	d	





Let's assume that our algorithm produced the above optimal enumeration using the pairwise combinations of chains shown below:

$$
\begin{gathered}
\{\langle a1\rangle,\langle a2\rangle,\langle a3\rangle,\langle a4\rangle,\langle b1\rangle,\langle b2\rangle,\langle b3\rangle,\langle b4\rangle,\langle c1\rangle,\langle c2\rangle,\langle c3\rangle,\langle c4\rangle,\langle d1\rangle,\langle d2\rangle,\langle d3\rangle,\langle d4\rangle\}\\
\Downarrow\\
\{\langle a2,a1\rangle,\langle a4,a3\rangle,\langle b2,b1\rangle,\langle b3,c4\rangle,\langle d4,b4\rangle,\langle c1,d2\rangle,\langle c2,c3\rangle,\langle d3,d1\rangle\}\\
\Downarrow\\
\{\langle a4,a3,a2,a1\rangle,\langle b2,b1,c1,d2\rangle,\langle b3,c4,d3,d1\rangle\ \langle c2,c3,d4,b4\rangle\}\\
\Downarrow\\
\{\langle a4,a3,a2,a1,b2,b1,c1,d2\rangle,\langle c2,c3,d4,b4,b3,c4,d3,d1\rangle\}\\
\Downarrow\\
\{\langle a4,a3,a2,a1,b2,b1,c1,d2,c2,c3,d4,b4,b3,c4,d3,d1\rangle\}
\end{gathered}
$$

By evaluating each square number to its value in the result of the first call of `combine_pairwise`, we see that we get only 2 different valued pairs:

$$
\begin{gathered}
\mathrm{eval}(\{\langle a2,a1\rangle,\langle a4,a3\rangle,\langle b2,b1\rangle,\langle b3,c4\rangle,\langle d4,b4\rangle,\langle c1,d2\rangle,\langle c2,c3\rangle,\langle d3,d1\rangle\})=\\
\{\langle 1,2\rangle,\langle 1,2\rangle,\langle 1,2\rangle,\langle 3,3\rangle,\langle 1,2\rangle,\langle 1,2\rangle,\langle 1,2\rangle,\langle 3,3\rangle\}=\{\langle 1,2\rangle,\langle 3,3\rangle\}
\end{gathered}
$$

There is a direct correspondence between the pair $\langle 1,2\rangle$ and the A node in the OBDD above, and the pair $\langle 3,3\rangle$ and the B node.

Since the size of the set returned by `combine_pairwise` is fixed, reducing the number of different (evaluated) pairs is equivalent to maximising the number of equal valued pairs. This is the goal of the fitness function f below:

$$
\begin{array}{l}
equal(a,b)=\begin{cases}1 & \text{if } a=b\\ 0 & \text{otherwise}\end{cases}\\
f(g)=\sum_{v\in\langle\mathbb{N},\ldots,\mathbb{N}\rangle}(\sum_{c_i\in g} equal(eval(c_i),v))^2
\end{array}
$$

Below this fitness function is applied on $\{\langle a1,a2\rangle,\langle a3,a4\rangle,\ldots,\langle d3,d4\rangle\}$ (the first step towards the standard enumeration) and on the result of the first call of `combine_pairwise` from the example above:

$$
\begin{array}{rcl}
f(g_{standard}) & = & (\#\langle 1,1\rangle)^2+(\#\langle 1,2\rangle)^2+(\#\langle 2,2\rangle)^2+\\
 & & (\#\langle 3,1\rangle)^2+(\#\langle 1,3\rangle)^2+(\#\langle 2,3\rangle)^2\\
 & = & 1^2+2^2+1^2+1^2+1^2+2^2=12\\
f(g_{optimal}) & = & (\#\langle 1,2\rangle)^2+(\#\langle 3,3\rangle)^2=6^2+4^2=40
\end{array}
$$

$g_{optimal}$ gives a much higher value as desired. The fitness function above can easily be adjusted to work with several pieces. Each value of h $\in [0\ldots\frac{\text{table size}}{64}[$ corresponds to some fixed position of the other pieces.

$$
\begin{array}{rcl}
v(i) & = & \text{Value of position with index i}\\
eval(h,\langle p_0,\ldots,p_i\rangle) & = & \langle v(2^6\cdot h+p_0),\ldots,v(2^6\cdot h+p_i)\rangle\\
f(g) & = & \sum_{v\in\langle\mathbb{N},\ldots,\mathbb{N}\rangle}(\sum_{c_i\in g}\sum_h equal(eval(h,c_i),v))^2
\end{array}
$$

Extensions of the GA: As mentioned earlier, the GA was not as successful as I had hoped it to be. Very often the entire population got stuck to some quite bad local optimum. Therefore I tried a couple of strategies to avoid this behaviour, which both involved having several levels of populations.

Separate populations with occasional migration: The idea is that when a single population gets stuck, the occasional interaction with the other populations will provide the necessary "noise" to get it out of its local optimum.

Recursive combination of populations: Start with a large number (e.g. 2^6) of small populations that evolve for a short time. Then combine these pairwise into slightly bigger populations that are allowed to evolve for a bit longer. Continue until only one population is left. The idea is that only the best local optimums will survive the merging processes.

Results: The table below show the size of the OBDD for the square enumeration computed by the different variants of the genetic algorithm. Also the result for a few hand coded enumerations are shown.[13]

Method	KPK	KQK	KRK	KBBK
Standard enumeration	72520	20713	27617	581664
Standard genetic algorithm	71313	23054	31650	-
8 separate populations with occasional migration	75433	23530	32694	-
As above, but allowed more processing time	72629	23050	31322	-
Recursive combination of populations	75865	23190	31706	-
As above, but allowed more processing time	70341	23246	32678	-
Pawn enumeration	75640	22713	31321	-
Bishop enumeration	88108	25857	41293	502031
Bishop enumeration 2	-	-	-	499671
Hilbert curve	71180	22289	30381	-
Z curve	71056	21845	29757	-

It should be noticed that the GA gave varying results for each time it was executed (with different rand seed of course), but the above values are quite representative.

The high computational cost of running the GA combined with my finite patience restricted the tests to the 3 non trivial 3 men endgames.

The only highlighting result was the discovery of the enumeration below which reduced the size of the KPK endgame to 70341 bytes and thereby defeated every hand coded enumeration.

[13] These enumerations are listed in appendix G

Square enum. for pawn found by GA

	a	b	c	d	e	f	g	h	
8	55	56	62	53	51	48	49	57	8
7	54	59	58	52	50	60	63	61	7
6	14	12	15	13	5	7	4	6	6
5	10	8	11	9	1	3	0	2	5
4	24	25	28	29	45	44	40	41	4
3	26	27	30	31	47	46	42	43	3
2	19	17	23	21	37	35	34	33	2
1	18	16	22	20	36	39	38	32	1
	a	b	c	d	e	f	g	h	

9.1.2 Algorithm from KBBK experiment

This algorithm was based on the idea that "similar" squares should be enumerated close to each other. If we review the simplified example from the description of the GA, we see that this is *not* the case for the shown optimal enumeration. For this enumeration the order of the values is:

$$\langle 1, 2, 1, 2, 1, 2, 3, 3, 1, 2, 1, 2, 1, 2, 3, 3 \rangle$$

Remember that the algorithm from the KBBK experiment worked by greedily combining the initial set of chains $\{\langle 0 \rangle, \ldots, \langle 63 \rangle\}$ until a single chain remained and represented the enumeration of squares. Each progress was made by linking together the 2 chains with largest binding value (bv).

$$\begin{aligned} w(p, q) &= |\{h \mid v(2^6 \cdot h + p) = v(2^6 \cdot h + q)\}| \\ bv(\langle p_0, \ldots, p_{m-1} \rangle, \langle q_0, \ldots, q_{n-1} \rangle) &= \max(w(p_0, q_0), w(p_0, q_{n-1}), \\ &\qquad w(p_{m-1}, q_0), w(p_{m-1}, q_{n-1})) \end{aligned}$$

In this section 3 variants of this algorithm are tested:

Chain merger: The algorithm presented in section 2 (without the improvement to escape non-optimal solutions).

Binary chain merger: The same algorithm, but where it is only allowed to merge chains of equal length (hence all chains will have length 2^n for some n).

Modified binary chain merger: The binding value between 2 chains is no longer determined solely by the values at the ends of the chains, but by the entire chains

$$bv(\langle (p_0, \ldots, p_{n-1} \rangle, \langle (q_0, \ldots, q_{n-1} \rangle) = |\{h \mid \textstyle\bigwedge_{i=0}^{n-1} v(2^6 \cdot h + p_i) = v(2^6 \cdot h + q_i)\}|$$

Results: The table below show that the results are just as useless as expected. An improved enumeration was only found for the KPK endgame, but the enumeration found by the GA was even better. At this point I gave up solving the problem of enumerating the squares algorithmically.

Enumeration	KRK	KQK	KPK	KBBK	KRNK
Standard enumeration	27553	20713	72500	516423	1291817
Chain merger	38817	25689	80332	563639	1493153
Binary chain merger	33545	23841	71784	588035	1385689
Modified binary chain merger	31141	24457	71832	599835	1312937

9.2 King positions

In section 4 the arbitrary choice of restricting white king instead of black king was made. Surprisingly we will see that this was a bad choice, as it makes the OBDD compress less.

Let the adjective "bound" refer to the king which due to symmetry is restricted to the a1-d1-d4 triangle or the a1-d8 rectangle. Similarly the adjective "free" refer to the other king.

All the index functions described in section 4 and 5 remain valid — we just have to replace each occurrence of white king with bound king and each occurrence of black king with free king.

The results in the table below speak for themselves — in almost all cases binding black king instead of white improved the compression.

Comparison of OBDD sizes with white king bound versus black king bound:

	KRK		KQK		KPK		KBBK		KRNK	
Enumeration:	w. k. b.	b. k. b.	w. k. b.	b. k. b.	w. k. b.	b. k. b.	w. k. b.	b. k. b.	w. k. b.	b. k. b.
Standard e.	27553	**27249**	20713	**19929**	72500	**71764**	516423	**434771**	1291817	**1235305**
Pawn e.	31257	**31141**	22713	**21129**	75640	**75052**	549023	**445223**	1307505	**1231481**
Bishop e.	41229	**39621**	25857	**23977**	88108	**88056**	502031	**418327**	1358073	**1292085**
Hilbert c.	30317	**27945**	22289	**19737**	71180	**70336**	455971	**383155**	1276493	**1241733**
Z curve	29693	**28653**	21845	**19677**	71056	**70208**	496583	**407875**	1291697	**1220901**
Bishop e. 2	43025	**42165**	26513	**24669**	98876	**98824**	499671	**421883**	1434777	**1398857**
Pawn e. 2	34433	**33821**	24169	**21109**	**69648**	69948	540543	**441151**	1316989	**1252309**
Chain m.	**38817**	40721	25689	**24465**	80332	**76060**	563639	**474699**	1493153	**1440245**
B. chain m.	33545	**27689**	23841	**21389**	**71784**	74228	588035	**435499**	1385689	**1234601**
M. b. chain m.	31141	**27297**	24457	**21581**	**71832**	71896	599835	**475631**	1312937	**1225013**

Why it decreases the size of the OBDD up to 20 % to bind black king instead of white king I have no clear answer to. However it must have something to do with the convention that the first pieces in an endgame name belongs to white and the rest to black — in all the endgames tested above this makes white the strong side and black the weak side.

This suggest that an even better strategy would be for each endgame to bind the king belonging to the strong side. This will assure that white king is bound in an endgame like KBPKQ where black is the strong side.

However having fixed the bound king simplifies the implementation and avoids overheads like

```
inline Position get_bound_kind() {
    return white_king_bound ? white_kind : black_king;
}
```

9.2.1 Decompress enumeration of legal king positions

Lets review how to index an OBDD given an endgame position. Of all the space optimising techniques that was introduced in section 4 only the enumeration of the kings was kept as part of the index format for the OBDDs. The rest was undone by padding with don't care values. The idea was to ensure that each bit used to index the OBDD had a simple interpretation on the indexed position (such as "is the rook placed inside the a1-h4 rectangle?"). Intuitively this would make the data less random and hence more compressible. The reason why the king positions were not used directly as substrings of the OBDD index was mainly the blowup this would incur on the table size. The tables below illustrate that we would need $\lceil \log(27 + 1) \rceil = 5$ bits for a king bound to the a1-d1-d4 triangle and $\lceil \log(59 + 1) \rceil = 6$ bits for a king bound to the d1-d8 rectangle. This blows up the size for pawnless endgames with a factor of $2^{\frac{5+6}{9}} = 4$ and $2^{\frac{6+6}{11}} = 2$ for endgames with pawns.

	a	b	c	d	e	f	g	h	
8	56	57	58	59	60	61	62	63	8
7	48	49	50	51	52	53	54	55	7
6	40	41	42	43	44	45	46	47	6
5	32	33	34	35	36	37	38	39	5
4	24	25	26	**27**	28	29	30	31	4
3	16	17	**18**	**19**	20	21	22	23	3
2	8	**9**	**10**	**11**	12	13	14	15	2
1	**0**	**1**	**2**	**3**	4	5	6	7	1
	a	b	c	d	e	f	g	h	

Pawnless endgame

	a	b	c	d	e	f	g	h	
8	**56**	**57**	**58**	**59**	60	61	62	63	8
7	**48**	**49**	**50**	**51**	52	53	54	55	7
6	**40**	**41**	**42**	**43**	44	45	46	47	6
5	**32**	**33**	**34**	**35**	36	37	38	39	5
4	**24**	**25**	**26**	**27**	28	29	30	31	4
3	**16**	**17**	**18**	**19**	20	21	22	23	3
2	**8**	**9**	**10**	**11**	12	13	14	15	2
1	**0**	**1**	**2**	**3**	4	5	6	7	1
	a	b	c	d	e	f	g	h	

Endgame with pawns

However this blowup in size is mainly cause by an enumeration of the squares that is unsuitable to the bound king. If we use an enumeration such that all squares in the a1-d1-d4 triangle have 4 bit indexes and all squares in the d1-d8 rectangle have 5 bit indexes, then we can reduce the overhead by a factor of 2.

New OBDD index scheme: The index functions from section 5.4 only changes slightly. The king enumeration index n is simply replaced by $2^6 \cdot \mathrm{K}_{bound} + \mathrm{K}_{free}$. It requires 4 or 5 bits to determine the position of the bound king while all other pieces uses exactly 6 bits. By letting the bound king be determined by the most significant bits of the index we have a solution that works for both endgames with and without pawns.

Unfortunately I have no test results indicating whether or not this change had a positive impact on the compressibility of the OBDD or not. The reason is that

I implemented the change as a part of a major reorganisation of the program. This involved changes to the format of the OBDD which affecting its size.

9.3 Comparing hand coded enumerations

All the hand coded enumerations are given in appendix G

Only 3 of the enumerations satisfy the requirement that the squares in the a1-d1-d4 triangle should be given 4 bit indexes and that the squares in the a1-d8 rectangle should be given 5 bit indexes. These are the Hilbert curve, the Z curve and the modified Z curve.[14] For each non-king piece I have chosen the simplest non-trivial endgame for the test.

Results for KRK endgame:

	Enumeration for kings:		
Enumeration for rook:	Hilbert curve	Z curve	Modified Z curve
Standard enumeration	8882+13692	7846+11844	7786+11728
Pawn enumeration	8914+13672	8010+11756	7954+12108
Bishop enumeration	8926+13776	7874+11792	7838+12164
Hilbert curve	8890+13676	7810+11796	7786+12092
Z curve	8874+13708	7842+11876	7766+11732
Bishop enumeration 2	9018+13848	7906+11796	7870+12212
Pawn enumeration 2	8898+13724	7866+11756	7842+12072
Whirl enumeration	8862+13684	7838+11756	7818+12068
Modified Z curve	8842+13676	7838+**10004**	**7750**+11636

Results for KQK endgame:

	Enumeration for kings:		
Enumeration for queen:	Hilbert curve	Z curve	Modified Z curve
Standard enumeration	6036+10718	5744+9122	5320+9098
Pawn enumeration	5944+10766	5396+9386	5244+9762
Bishop enumeration	6032+10822	5512+9386	5396+9774
Hilbert curve	5952+10738	5468+9358	5304+9694
Z curve	6040+10722	5764+9150	5348+9082
Bishop enumeration 2	6052+10826	5512+9386	5368+9790
Pawn enumeration 2	5944+10710	5428+9362	5356+9691
Whirl enumeration	6044+10754	5532+9382	5436+9686
Modified Z curve	6068+6886	**4692**+**6798**	4696+6842

Results for KPK endgame:

[14] The king "standard enumeration" also satisfies the requirement, but was added after these tests.

Enumeration for pawn:	Enumeration for kings: Hilbert curve	Z curve	Modified Z curve
Standard enumeration	29583+22891	21191+16551	21015+16375
Pawn enumeration	30171+22987	21151+17227	20747+16815
Bishop enumeration	30127+24723	21175+17195	20803+16863
Hilbert curve	30059+23303	21195+17235	20731+16879
Z curve	29515+22887	21191+16571	20983+16339
Bishop enumeration 2	30219+24727	21199+17211	20695+16819
Pawn enumeration 2	30087+23723	21191+17215	20759+16867
Whirl enumeration	29231+22091	21167+17251	20755+16863
Modified Z curve	29043+21419	**20575**+16487	20999+**16143**

Results for KBBK endgame:

Enumeration for bishops:	Enumeration for kings: Hilbert curve	Z curve	Modified Z curve
Standard enumeration	152550+133499	110838+137359	161389+137919
Pawn enumeration	108399+151415	110746+142235	116482+143487
Bishop enumeration	144741+207336	173806+196895	176126+187915
Hilbert curve	147744+213364	158529+204603	172114+210095
Z curve	152650+133475	106339+136979	161389+137919
Bishop enumeration 2	131519+156936	169618+180483	183018+179099
Pawn enumeration 2	121678+225871	170922+191459	177066+149647
Whirl enumeration	102426+140179	102298+133719	171902+149091
Modified Z curve	**101218**+**127435**	101554+130659	107650+143651

Results for KBNK endgame (bishop use the modified Z curve enumeration):

Enumeration for knight:	Enumeration for kings: Hilbert curve	Z curve	Modified Z curve
Standard enumeration	615756+752681	605988+875657	809764+861605
Pawn enumeration	615616+752509	745216+875549	750800+861489
Bishop enumeration	615888+752385	744904+875865	750940+861705
Hilbert curve	615940+752905	744880+875969	750652+861653
Z curve	615772+752701	605988+875677	809732+861305
Bishop enumeration 2	616184+752585	745104+876049	750932+861869
Pawn enumeration 2	615752+752673	745044+875849	751168+861405
Whirl enumeration	615808+752741	608012+875565	750752+861325
Modified Z curve	615800+645537	574364+**623897**	**574112**+845285

The results above are easily summarised — for each of the 5 non-king pieces the modified Z curve is the winner. For the king enumeration the picture is more diffuse, but it still has a clear winner — namely the Z curve.

I find these results quite encouraging. They indicate that we can choose a fixed enumeration for each piece kind which is close to optimal for every endgame.

Hence, if we are satisfied with this close to optimal performance, there is no need to compute or hand code it for every endgame.

The tables show that even small changes to the enumerations can have a big impact on the OBDD compressibility. As an example the KBNK endgame with black to move gets 25% smaller by choosing the Z curve instead of the modified Z curve for the kings.

9.4 Implementation

In the implementation I have removed the algorithms for computing square enumerations. Instead it provide the choice of which hand coded enumeration to use for each kind of piece. This enumeration is stored together with the description of the OBDD and hence takes up 64 bytes of space for each piece.

Another option would be to fix the enumerations. Always choose the Z curve for a king piece and always choose the modified Z curve for any other piece. This way we could save $64 \cdot n$ bytes for each n piece endgame. However, I think there are still improvements to be made and this would be a limiting choice.

10 Permutation of bit order, sifting algorithm

The variable ordering used in the OBDD defines the positions of the pieces one piece at a time. I.e. first the position of the bound[15] king is determined, then the position of the free king, etcetera.

The size of the minimal OBDD depends highly on this variable ordering. An OBDD with depth n has $n!$ different variable orderings, so doing an exhaustive search for the best one is infeasible. Also, in [31] this problem is shown to be NP-complete.

The article “Dynamic Variable Ordering for Ordered Binary Decision Diagrams” [30] is the first one to present an efficient algorithm that is not based on any application specific heuristics.

This algorithm works by performing a local search. For each variable, it freezes the ordering of all the other variables and then tries every insertion place. The variable is moved to the position where the size of the OBDD was minimal. The first time this procedure has been applied to all variables without any improvement, a local optimum is reached. This local optimum is the computed variable ordering.

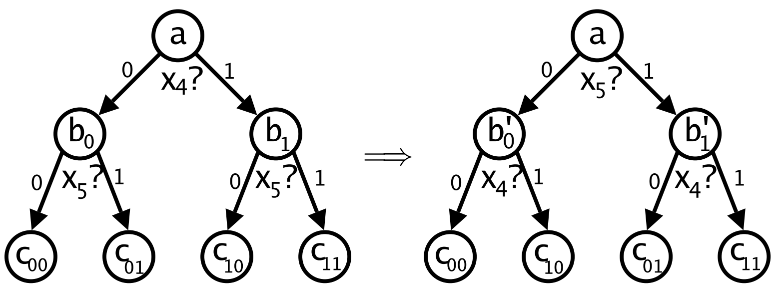
It is not necessary to rebuild the OBDD for each insertion place that is tested. If the variable ordering is only changed by swapping two successive variables, then the minimal OBDD can be updated locally.

[15]Remember that the bound king refers to the king which is being restricted to the a1-d1-d4 triangle or a1-d8 rectangle.

Example (taken from [30]): We have the variable ordering $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ and wish to find the best position for x_4. First x_4 is swapped all the way to the first or last position (depending on what is closest), and then to the other end. Now all positions have been tried, and x_4 is then swapped back to the position that gave the least number of nodes. In this example the 3'rd position.

$x_1, x_2, x_3, x_4, x_5, x_6, x_7$	initial
$x_1, x_2, x_3, x_5, x_4, x_6, x_7$	swap(x_4, x_5)
$x_1, x_2, x_3, x_5, x_6, x_4, x_7$	swap(x_4, x_6)
$x_1, x_2, x_3, x_5, x_6, x_7, x_4$	swap(x_4, x_7)
$x_1, x_2, x_3, x_5, x_6, x_4, x_7$	swap(x_7, x_4)
$x_1, x_2, x_3, x_5, x_4, x_6, x_7$	swap(x_6, x_4)
$x_1, x_2, x_3, x_4, x_5, x_6, x_7$	swap(x_5, x_4)
$x_1, x_2, x_4, x_3, x_5, x_6, x_7$	swap(x_3, x_4)
$x_1, x_4, x_2, x_3, x_5, x_6, x_7$	swap(x_2, x_4)
$x_4, x_1, x_2, x_3, x_5, x_6, x_7$	swap(x_1, x_4)
$x_1, x_4, x_2, x_3, x_5, x_6, x_7$	swap(x_4, x_1)
$x_1, x_2, x_4, x_3, x_5, x_6, x_7$	swap(x_4, x_2)

To illustrate how the `swap` operation is implemented consider swap(x_4, x_5) above. For each node in the x_4 layer we apply the following transformation:



Hence it is only the number of nodes in the earlier x_5 layer that can change. If there will be less different b' nodes than there were b nodes, then we have made an improvement.

10.1 Combining with minimisation using don't cares

A natural question is whether the sifting algorithm should be run before or after the mapping of don't cares. The top-down mapping of don't cares works on the input given as a table. The sifting algorithm on the other hand takes an OBDD as input and produces an OBDD. Hence, it would be natural to do the things in this order:

- Expand the endgame table into the simpler index scheme used by OBDD's.

- Rearrange the entries according to which enumerations of the board squares that have been used.
- Apply the top-down mapping of don't cares.
- Construct the OBDD.
- Apply the sifting algorithm.
- Compress the representation of the OBDD.

If the sifting algorithm was to be applied first, then we would afterwards have to convert the OBDD to a table, apply the top-down mapping and reconstruct the OBDD again. This wastes some time. I tried both approaches and got the results shown below:

Endgame	No sifting	Top-down mapping first	Sifting first
KRK	9138+13204	9130+12672	**9070**+**12020**
KQK	6032+10714	**6004**+10578	6100+**10266**
KPK	28715+23635	25551+20555	**19651**+**15219**
KBBK	**151014**+198531	151150+197807	177282+**194147**
KQKR	**861800**+1473554	861884+1471618	864528+**1104306**

If the don't cares were mapped before running the sifting algorithm, then the sifting algorithm improved very little. It seems that the optimisation performed by the top-down algorithm is tightly connected to the variable ordering. The improvement caused by mapping the don't cares is a lot higher for the variable ordering that was present when running the top-down algorithm. This is likely to freeze the variable ordering and therefore the sifting algorithm should be used first.

Unfortunately, when sifting is used before don't care minimisation it might degrade the compression. The KBBK endgame with white to move has increased in size from 151014 to 177282 by applying sifting. The easy solution would be to construct the OBDD both with and without sifting and keep only whichever has the least size. I have chosen the other simple solution of always performing the sifting first.[16] In average this choice works quite well — just look at the endgames KPK and KQKR with white to move.

10.2 If sifting powerful enough?

To test how likely it is for the local search to get stuck in some suboptimal solution, I decided to do a little experiment. For each of the endgames KRK, KQK, KPK and KBBK the sifting algorithm was tried on several different initial variable orderings. In each case the same solution was produced. I therefore

[16] The implementation let's you decide whether you wish to perform sifting first or not. See appendix B for more details.

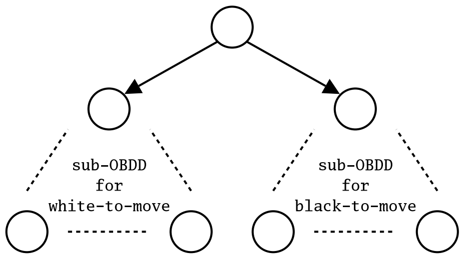
concluded that the sifting algorithm is strong enough and that there was no need to try to improve it (e.g. using application specific knowledge).

11 Splitting into several OBDDs using clustering

When I started the implementation of the compression algorithm I found it natural to represent each endgame table with only one OBDD. The most significant bit then determined the side-to-move. There is however two important reasons why a separate OBDD for each side-to-move is more appropriate:

1: If the cost of doing a 1 ply search from all positions with e.g. black-to-move is acceptable then we don't need the values for white-to-move (this is also exploited in Nalimov's endgame tables).

2: Consider the endgame KQK. If it is white-to-move then the value will be a mate in $1 \leq n \leq 10$, while if it is black-to-move then value will either be a draw or a loss in $1 \leq n \leq 10$ moves. The side-to-move hence splits the possible values in two disjoint sets. Assuming the variable order in the corresponding OBBD has not been modified, it will look like this:



Remember that each internal node is being represented by its left and possible its right child index. The number of bits needed for each index depends on the number of nodes in the layer below. Hence, by splitting up the OBDD in one for each side-to-move you can expect a saving of one bit per index.

It has just been argued that it is a good idea to have an OBDD for each side-to-move. But why stop here? We can try to reduce the number of bits per index even further by splitting up the endgame into several parts. If these parts are distinct enough, then the saving of using less bits per child index out pays the cost of having some sub-OBDDs defined more than once.

11.1 Principle of clustering

When dividing an endgame table we need some criteria that tells us how we are allowed to do so. Given any chess position, we must be able to determine where we should do the lookup. In the above example the side-to-move tells us which OBDD to use.

I made the assumption that using the positions of the kings in addition to the side-to-move would be a reasonable way to divide the table. Normally the position of a king has a better correlation with the value of the position than any of the other pieces. As an example, a check mate in a KRK or KQK endgame can only occur when the black king is placed on a border square.

This way an endgame gets divided into $n = 2 \cdot 462$ or $n = 2 \cdot 1806$ subsets — depending on whether it is pawnless or not.[17]

A clustering algorithm is then used to combine these n subsets into a number of parts that each will be represented by an OBDD. Each OBDD will still be using the same index scheme. All positions not covered by an OBDD will simply be given the value don't care. This makes the use of the top-down mapping algorithm crucial.

11.2 Example

The idea is sketched below. Let's say we have the (very simplified) endgame table t:

t	0101 001* 11*1 10*1 0000 010* 11** 11*1 0111 1*01

If we divide t into 5 equally sized subsets of successive entries, it might be a good idea to perform the clustering below. t_0 will then consist mainly of 0-entries and t_1 mainly of 1-entries. Measured in 0'th order entropy we have thus improved the situation. But we also need to introduce the table c that gives the mapping from the subsets to the clusters:

$$c = \langle 0, 1, 0, 1, 1 \rangle$$

t_0	0101 001* **** **** 0000 010* **** **** **** ****
t_1	**** **** 11*1 10*1 **** **** 11** 11*1 0111 1*01

We have the following relationship ($\lfloor \frac{i}{8} \rfloor$ gives the number of the subset the entry i belongs to):

$$t_{c[\lfloor \frac{i}{8} \rfloor]}[i] = t[i]$$

11.3 Clustering algorithm

We have divided the endgame into n subsets. From each of these we compute a pair (v, w). w is the number of care entries in the subset. Let k be the number

[17]Remember that the restriction of a king to the a1-d1-d4 triangle produced 462 legal placements for the 2 kings. Similarly the restriction to the a1-d8 rectangle produced 1806 legal placements for the 2 kings.

of different distance to mate values for this endgame, and enumerate these values from 0 to $k-1$. Let v' be the point in the k'th dimensional space you get by counting the number of occurrences of each distance to mate value in this subset. v is then the intersection between the k'th dimensional hyper sphere and the line through $\mathbf{0}$ and v'.

The clustering algorithm works on these (v, w) fingerprints. If 2 subsets produces fingerprints (v, w) and (v', w') where the euclidian distance $\|v - v'\|$ between v and v' is low, then they have a similar distribution of values. In this case it is likely best to include them in the same OBDD.

The algorithm below recursively splits up the endgame into smaller and smaller clusters. The algorithm is very expensive — every time the algorithm tries to do a binary split it constructs the corresponding OBDDs to see if it pays off. The k-mean clustering (see e.g. [39]) algorithm with k=2 is used to perform this binary split. Each fingerprint (v, w) is given the weight w.

Algorithm: `ComputeClusters`
Input: Endgame table t that has n defined subsets.
Output: A mapping $c : [0, \ldots, n-1] \mapsto [0, \ldots, n'-1]$,
where n' is the number of clusters produced.
Let $\mathcal{T}$ be an empty set of sets of integers
- each set of integers represents a cluster.
Let t_i denote the i'th of the n subsets of t
Add $\{0, \ldots, n-1\}$ to $\mathcal{T}$
// $\mathcal{T}$ now contains one cluster representing the entire endgame.
While T not empty do begin
    Remove a set S from $\mathcal{T}$
    Let s be the size of the OBDD covering the subsets $\{t_i \mid i \in S\}$
    Let A,B be the result or applying a 2-mean clustering
        on the fingerprints of $\{t_i \mid i \in S\}$
    Let a be the size of the OBDD covering the subsets $\{t_i \mid i \in A\}$
        and similar for b.
    If $a + b < s$ then begin
        // It pays of to divide the cluster S into A and B.
        Add the sets A and B to $\mathcal{T}$
    End else begin
        //The cluster S should not be split up any further.
        Add S to the solution.
    End
End

11.3.1 Initial results

The results given below was produced when the only other optimisation was the mapping of don't cares. The clustering improves the performance for all 5 endgames tested. Note however that both the positions with white-to-move and black-to-move was represented by a single OBDD before applying the clustering. Most of what has been gained comes from separating these two parts of the endgames.

Before and after adding clustering

KRK		KQK		KPK		KBBK		KRNK	
Before	After	Before	After	Before	After	Before	After	Before	After
27249	**24627**	19929	**18735**	71764	**65506**	434771	**388729**	1235305	**1088183**

11.4 Clustering revisited

Since the experiment described above was performed, the implementation has changed. Now there is an OBDD for each side-to-move. Also, various optimisations have been added. I therefore decided to repeat the experiment. The implementation requires that all OBDDs an endgame has been split into share the same variable ordering. This choice was partly made to make the implementation simpler and partly to make clustering algorithm faster. The sifting algorithm is executed only once and that is before `ComputeClusters`.

This time the clustering algorithm was tested on every 3 and 4 men endgame for each side to move. The table below shows only the successful cases — i.e. those in which more than one cluster was produced.

Endgame	Without clustering	With clustering	Saving	Number of clusters
KRPK - white to move	999754	960288	39466	3
KRKB - white to move	171964	168549	3415	2
KQRK - white to move	90796	77619	13177	4
KQNK - white to move	212251	202413	9838	3
KQKQ - white to move	197637	192764	4873	4
KQKR - white to move	861440	836140	25300	11
KQKN - white to move	508792	482372	26420	5
KQKN - black to move	1056193	980383	75810	11
KQKB - white to move	490204	457223	32981	8
KQBK - white to move	194934	194687	247	2
KNKP - black to move	1199859	1177844	22015	2
Total	5983824	5730282	253542	-

Considering that there are five 3-men endgames and thirty 4-men endgames, the above results are disappointing. The total saving is 253542 bytes — or less than 1% of the total size. This is not worth the slow down of the compression caused by the clustering algorithm. But why is it suddenly not able to perform any better? The main reason must be that the separation of side-to-move has already been done.

11.5 Using another separation into subsets

Instead of identifying each subset solely by the positions of the kings I have tried a different approach. Let's take the endgame KPK as an example. By summing the results of indexing each tables below with the corresponding position we get a value v. This is the number of the subset this position belongs to. There are in total $4 \cdot 6^2$ subsets.

Black king (bound)

	a	b	c	d	e	f	g	h	
8	0	0	36	36	-	-	-	-	8
7	0	0	36	36	-	-	-	-	7
6	0	0	36	36	-	-	-	-	6
5	72	0	0	36	-	-	-	-	5
4	72	72	0	0	-	-	-	-	4
3	72	72	72	72	-	-	-	-	3
2	108	108	108	108	-	-	-	-	2
1	108	108	108	108	-	-	-	-	1
	a	b	c	d	e	f	g	h	

White king

	a	b	c	d	e	f	g	h	
8	30	30	30	30	30	30	30	30	8
7	30	18	18	18	18	18	18	30	7
6	30	18	24	24	24	24	18	30	6
5	6	24	24	0	0	24	6	30	5
4	6	6	0	0	0	0	6	30	4
3	12	12	6	0	0	6	6	30	3
2	12	12	12	12	6	6	6	30	2
1	12	12	12	12	12	6	6	30	1
	a	b	c	d	e	f	g	h	

White pawn

	a	b	c	d	e	f	g	h	
8	-	-	-	-	-	-	-	-	8
7	4	4	4	4	4	4	4	4	7
6	3	2	3	3	3	3	3	3	6
5	2	2	2	2	2	1	1	1	5
4	0	0	0	2	1	1	1	1	4
3	0	0	0	0	5	5	5	5	3
2	0	0	0	5	5	5	5	5	2
1	-	-	-	-	-	-	-	-	1
	a	b	c	d	e	f	g	h	

Each subset covers a set of positions on the form

$$\{p_{\mathtt{K}_w\mathtt{PK}_b} \mid (\mathtt{K}_w \in A) \wedge (\mathtt{P} \in B) \wedge (\mathtt{K}_b \in C)\}$$

where A, B and C are some subsets of $\{0, \ldots, 63\}$ and K_w, K_b is white and black king.

The above example generalises to the use of $4 \cdot 6^{n-1}$ subsets for any n men endgame. The bound king has less available squares than the other pieces and these squares have therefore been divided into only 4 groups.

If the number of subsets was increased, then each subset would cover less positions. This more fine grained input to the clustering algorithm would produce better results. On the negative side, the mapping $c\colon\ [0, \ldots, n-1] \mapsto [0, \ldots, n'-1]$ produced by `ComputeClusters` would require more space to represent. The choice to group the squares of each piece into exactly 6 sets has just been based on my estimate of the best tradeoff.

The tables from the KPK example have been computed using a clustering algorithm. Taking the white pawn as an example, we split up the endgame into 64 subsets — one for each of its positions (16 of the subsets consist solely of don't care values as the pawn only has 48 legal positions). Then a 6-mean clustering algorithm using the same idea with the fingerprints has been applied.

The new way of defining the subsets has been tested on all 3 and 4 men endgames:

Endgame	Without clustering	With clustering	Saving	Number of clusters
KPPK - white to move	435415	419723	15692	2
KQKB - white to move	490204	487861	2343	2
KQPK - white to move	568604	547481	21123	2
KNPK - white to move	1277110	1209603	67507	2
KNPK - black to move	1383360	1265065	118295	2
KBPK - white to move	1161319	1077872	83447	2
KBPK - black to move	1341477	1224366	117111	2
Total	6657489	6231971	425518	-

Even though less endgames are improved, the total saving is higher — 425518 bytes compared to 253542. This brings the saving on the right side of 1%, but still this isn't very impressive.

11.6 Using knowledgeable scoring to define subsets

In this third attempt on choosing the subsets I have hand coded the functions that map a position to its subset number. But lets first jump to something completely different.

In the introduction it was mentioned how Ernst A. Heinz has been able to reduce all 3-4 men endgames to only 15 MB uncompressed data by throwing away unnecessary information. Take the KRKB endgame as an example. This endgame is either drawn or won for white. Only this 1 bit of information saved for each position. If a won position is indexed, then a specialised heuristic evaluation is applied. This function has been designed such that a position closer to a mate is rated higher than others. The information about the win is also used, but this is irrelevant here. The evaluation function is (*dist* and *edge_dist* is counted in king moves):

$$eval(K_w, R, K_b, B) = \frac{61}{16} + \frac{1}{4} \cdot dist(K_b, B) - \frac{1}{8} \cdot edge\_dist(K_b) - \frac{1}{16} \cdot dist(K_w, K_b)$$

Now, if Heinz designed this evaluation function properly then there should be a high correlation between the number of moves it takes for white to win and the value of the evaluated position.

Hence, by letting each evaluation value define a subset of the endgame we get a good split as the subsets have little in common. By changing the evaluation function slightly we can make it return a value in $[0, \ldots, 41]$:

$$13 + 4 \cdot dist(K_b, B) - 2 \cdot edge\_dist(K_b) - dist(K_w, K_b)$$

The endgame is divided into quite a few subsets compared to the first two approaches that was described. It has been tested on the endgames KRK, KQK, KBBK, KBNK, KRKB and KQKR but worked in none of the cases.

I find this a bit odd. If 0'th order entropy encoding was used, then the size would decrease about 10-20% for each of the endgames tested. This can be illustrated on the KQK endgame which has only 11 subsets. Each position belongs to the subset numbered $corner\_dist(K_b) + dist(K_w, K_b)$. The reader should notice that the contour of the non-zero numbers shows a clear correlation between the subset number and the distance to mate.

KQK, white to move (0'th order entropy = 51350.3 bits)

Subset:	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	0'th order entropy
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	99	36	0	0	0	0	0	0	0	0	112.947
3	68	190	283	61	0	0	0	0	0	0	1039.71
4	74	190	313	846	387	117	3	0	0	0	4209.96
5	71	166	344	767	1224	889	221	3	0	0	8758.5
6	0	67	191	554	875	1479	1136	201	0	0	10554.8
7	0	0	26	289	533	930	1571	921	152	0	10218.2
8	0	0	0	29	259	358	573	554	140	1	4315.54
9	0	0	0	0	35	288	331	238	80	0	1959.04
10	0	0	0	0	0	40	267	19	3	0	300.531
Sum	312	649	1157	2546	3313	4101	4102	1936	375	1	41469.2

KQK, black to move (0'th order entropy = 82422.8 bits)

Subset:	draw	-M10	-M9	-M8	-M7	-M6	-M5	-M4	-M3	-M2	-M1	-M0	0'th order entropy
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	24	0	0	0	0	0	0	0	0	71	74	17	326.626
3	69	0	0	0	0	4	40	188	364	144	50	9	1970.37
4	267	0	0	23	205	454	824	705	318	85	23	10	7721.23
5	559	0	0	160	1092	1821	1228	559	165	82	28	10	14547.9
6	783	0	123	931	2514	1621	748	256	77	15	0	0	17404.2
7	755	3	539	2457	1975	816	288	86	25	0	0	0	16294.1
8	312	3	413	1126	774	252	67	29	0	0	0	0	6762.48
9	144	2	278	675	285	69	35	0	0	0	0	0	3121.45
10	40	0	147	231	38	40	0	0	0	0	0	0	944
Sum	2953	8	1500	5603	6883	5077	3230	1823	949	397	175	46	69092.3

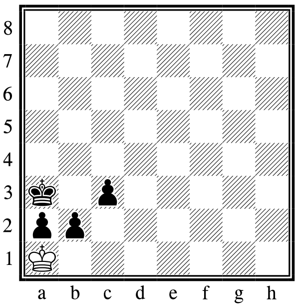
11.7 Conclusion

We have seen that using clustering to split up an endgame into several OBDDs can improve the compression achieved. In its current form I don't think the saving of 1.5% makes it worth using it, because it adds some extra complexity to the lookup operations. However, the tests described in this section were not exhaustive. There are still things to be tested and parameters that has not been optimised. Also, the compression algorithm currently fails on most of the 5-men endgames due to insufficient memory. Hence it might be necessary to break these endgames into clusters that are manageable in size even if we gain nothing no compression.

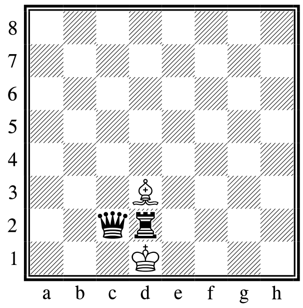
12 Legality of chess positions

In chess the concept of a legal position is not as clear as one might think. As it is not part of the official chess rules it is open for interpretation. Clearly, an endgame database should contain at least the positions that satisfies the strictest definition, but what about the positions below?

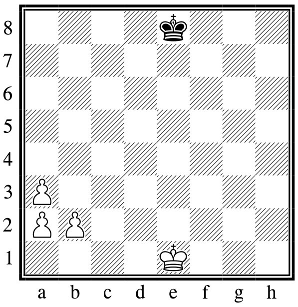
white to move



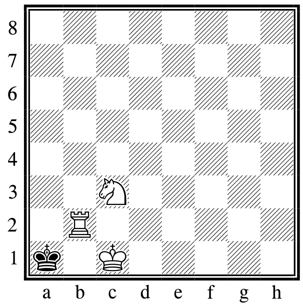
white to move



white to move

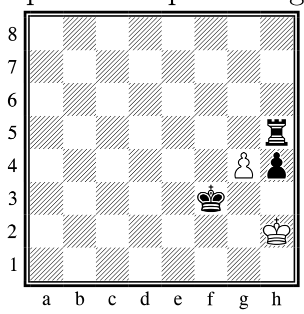


white to move

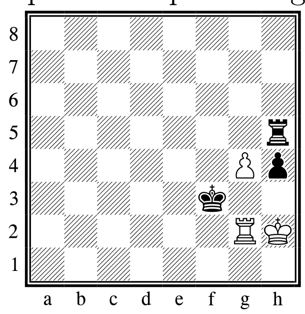


It is easy to conclude that the above positions are unreachable (hint[18] to first position). If the reader tries to index Nalimov's endgame tables with any of the above positions he will see that they are regarded as normal positions.[19] Now, lets try to index Nalimov's endgame database with the 2 unreachable positions below.

Black to move
En passant capture at g3



Black to move
En passant capture at g3



In both positions an en passant capture is possible. Therefore the last move must have been to move the white pawn 2 squares forward. In the first position, white

[18] If it were reachable, then the last move must have been b3-b2 (otherwise white king could have been captured). This is a non-capturing move, hence whites move before must have involved the king. But both the possibilities b1-a1 and b2-a1 originate in illegal positions □

[19] An online Nalimov tablebase server is available at http://www.lokasoft.nl/uk/tbweb.htm

pawn could have captured the black king from it earlier position at g2. The rules say that no move may expose the king belonging to side-to-move. This concludes that the first position is unreachable. In the second position it is simpler to conclude that the last move cannot have been g2-g4 — the g2 square is occupied. But conceptually there is no difference between the 2 positions.

The first position is contained in Nalimov's endgames, the second is not. The cleanest definition of a legal position is a position that is reachable from the initial position. But as the reader might have realised, such a legality check would be too difficult to implement. This is also the conclusion of the article "Legality of Positions of Simple Chess Endgames" [6].

Also, there would be no real gain by giving these positions the value of a broken position in Nalimov's endgame. The reason is that the value of a broken position is treated no different from the other values by the compression program. But with the OBDDs we have an effective way of dealing with broken positions (or don't cares as they're referred to then).

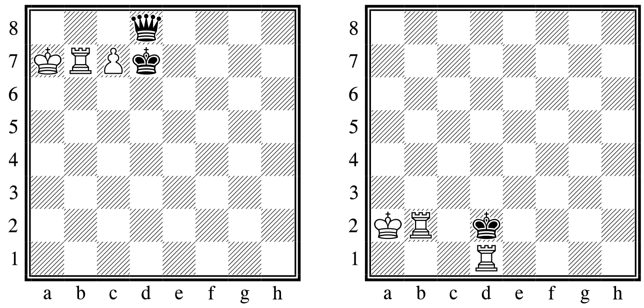
This section describes an attempt to identify as many unreachable positions as possible. The predicate legal will refer to legality in the sense of Nalimov — i.e. legal but not necessarily reachable.

Notice, that the endgame construction algorithm still uses the same definition of legal as Nalimov. The check for unreachability is performed when compressing an endgame as an OBDD.

12.1 Adjusting for symmetry

An important thing to notice is how symmetry is used to reduce the number of chess positions in the endgames. Each position actually represents (usually) 2 or 8 different transformations of that same position. If any of these versions are reachable from the initial position, then we can't label it as a don't care. Below is shown an example. After the pawn promotion, the position enters a pawnless endgame. If the position was afterwards looked up in the endgame table, then it would be mapped to the position shown to the right. This right position is unreachable. But as it also represents reachable positions, we are therefore not allowed to change its value.

White moves cxd8=R   Black to move



12.2 Statically identifiable unreachable positions

If a position is unreachable, it is almost always because the side-to-move king has been checked in some impossible way. The following checks are performed every time a chess position is being set up. If any of the properties are satisfied, then the position is unreachable. I could have added more special cases, but I wanted the check to be inexpensive.

- The number of checks is larger than 2
- The side-to-move king is checked from diametral opposite directions.
- The side-to-move king is checked by two non block able pieces.
- The side-to-move king is checked by one of the following pairs of pieces (R_i is a rook placed on a non-border square).

	P	N	B	R_i
P	x	x	x	
N	x	x		
B	x		x	
R_i				x

12.3 Trying to take back moves

A computationally more expensive solution is to take back a number of moves from the position in question. If this fails, then the position must be unreachable.

It is no easy task to implement such a retro move generator. The backward algorithm for constructing the endgames also used a retro move generator. However, this only had to return moves between positions in the same endgame. Therefore captures and pawn promotions could be ignored.

The reader doesn't need to be bothered with the details of the retro move generator. But it should be mentioned that to define an undoing move, we need to reconstruct any information that was destroyed by this move. I.e. was a piece captured, what was the castling rights, etc. Also, when the position is a representative for a number of transformed versions and a certain move can only be undone in another version, then we have to include this move also. If this move is tried undone, then we have to apply some transformation of the board first. Take the right position from section 12.1 as an example. The only move we can undo is cxd8=R, and this requires that we mirror the board horizontally first. Appendix C illustrates all the special cases that one has to be aware of when implementing a retro move generator.

12.4 Results

In appendix F the number of positions judged unreachable for every 2-5 men endgame is shown. These are summarised below.

Endgame	Illegal positions	Probably reachable	Statically decided unreachable	Undoing the 1'st move failed	Undoing the 2'nd move failed	Undoing the 3'rd move failed	Undoing the 4'th move failed
ALL 3 MEN	37.547	371.861	0	512	0	0	-
ALL 4 MEN	31.162.319	125.529.205	168.904	579.696	739	1	-
ALL 5 MEN	11.350.319.255	25.679.713.947	115.999.273	282.836.459	318.234	1.846	42

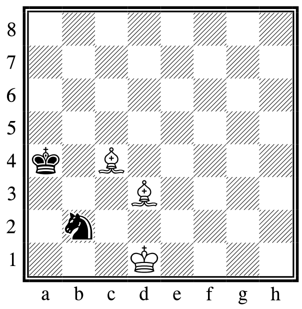
The computationally cheap static check is not as strong as I had hoped it to be. If the aim is to improve compression, then trying to take back more than one move would be a waste of time as more than 99% of the unreachable position have been found by then. The table below shows the proportion of the otherwise legal positions that was judged unreachable.

3 men endgames	4 men endgames	5 men endgames
0.137%	0.593%	1.53%

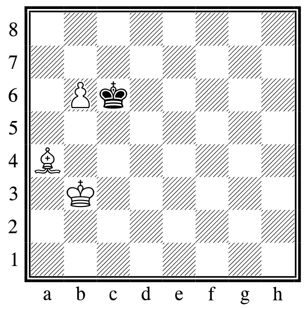
If we assume that the OBDDs will decrease in size by a similar amount, then the improvement is even smaller than for the clustering algorithm. However, the savings caused by detected unreachable positions come for free. No other part of the program need to be changed. In the best possible case — the endgame KQQQK with black to move, the size was reduced from 1189144 to 969960 bytes.

Personally I find the positions judged unreachable more interesting than the saving in size. A few examples is shown below. It is actually not easy to see how 2 moves can be taken back in the middle position and 3 moves can be taken back in the right position (hint[20])

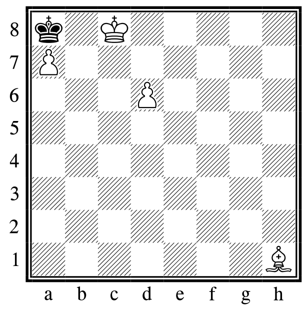
White to move
1'st undo failed



Black to move
3'rd undo failed



Black to move
4'th undo failed



[20]In both the middle and right position the last move was an en passant capture. In the middle position the last 2 moves was b7-b5, a5-b6 or b7-b5, c5-b6. In the right position the last 3 moves was e4-e5, d7-d5 and e5xd6.

13 Comparison with other compression methods

The work in this thesis has focused on compressing the endgames with distance to mate information. As illustrated with DarkThought a world class chess program will be optimised to achieve the perfect play with much less information. Therefore this section will also present comparisons where the distance to mate (DTM) information has been reduced to simply saying whether the position is won, drawn or lost (WDL).

13.1 Distance to mate

Nalimov's endgame tables are by far the most used. Also, it is the only format which allows the use of the tables in a compressed form. This makes it obvious to judge the performance of the OBDDs against Nalimov's tables. Lets start by focusing on the compression achieved and ignore the decompression speed.

13.1.1 Datacomp

Nalimov's endgame tables are compressed using the program Datacomp. Datacomp is an implementation of a limited-reference variant of the Lempel-Ziv algorithm. It divides the files into blocks of 2^n bytes and compresses each block separately. The block size can be chosen between 2^8 and 2^{13} bytes. Nalimov's endgames has been compressed using a block size of $2^{13} = 8192$ bytes. The compressed files support the decompression of an individual block. To avoid the cost of additional disk reads in the search for a specific compressed block, an array of pointers specifies the offset of each block. These arrays are read from the files into the memory during an initialisation process. Therefore, Nalimov's endgame tables actually takes up memory even though they stored on the hard disk. These 4 bytes per 8 K-bytes block is not a problem unless you go for all the 6 men endgames.

13.1.2 Datacomp versus gzip and bzip2

The table below shows how Datacomp performs compared to the program bzip2 and gzip. bzip2 is a compression program based on the Burrows-Wheeler transformation and Huffman coding. gzip uses Lempel-Ziv coding (LZ77). It is the KRK endgame in different representations that has been tested.

KRK endgame (w. t. m. + b. t. m.)	Raw data	gzip	bzip2	Datacomp
Tables generated	29568+29568	**5832**+**7582**	6248+7791	5910+7729
Indexing scheme used by OBDD	65536+65536	7744+9076	**7243**+**8526**	8073+11880
-||- after mapping of don't cares	65536+65536	6582+7842	**6157**+**7621**	10218+9028
Using Nalimov's indexing scheme	27030+28644	6129+**6997**	**5959**+7240	7059+7051

We see that Datacomp gives the worst compression ratio. Especially for the padded versions of the table (from which the OBDD is constructed). For the compact index schemes it is however quite close to gzip and bzip2. This is more important as this is its target area. Hence I will assume that Datacomp is a worthy representative for the standard compression algorithms.

I included the padded versions in the test because I wanted to see how much the mapping of don't cares would improve the performance of normal compression programs. Surprisingly, the file covering white-to-move is compressed less by Datacomp after the don't cares have been mapped. bzip2 on the other hand compresses this file to less than the one using the compact indexing scheme.

13.1.3 This thesis versus Nalimov

The table below presents the main result of this thesis. The reader should try to ignore the two rightmost columns for now, they will be introduced later. The OBDDs are able to represent all 3-4 men endgames in less than 30 MB — 29.85 MB to be precise. This is very close to the 29.15 MB for Nalimov's endgames.

The reader is invited to compare the final performance of the OBDDs shown below, with the initial results from the end of section 5. Even though it didn't look promising in the beginning, the variety of ways to make improvements finally resulted in a competitive compression ratio.

However, we know from section 2 that the performance of standard compression programs can similarly be improved. The table below shows that the KBBK endgame has been compressed to a mere 97658+128747 bytes — or less than half of what Nalimov uses. But 97658+128747 = 226405 is still not as good as the 181353 bytes that gzip produced in section 2.

Endgame	Distance to mate		Win/draw/loss	
	This thesis	Nalimov	This thesis	EgmProbe 2.0
KK	455	-	455	-
KPK	19563+**15491**	**17654**+16589	**1172+1336**	3962
KNK	551+551	**186+187**	551+551	**56**
KBK	**551**+551	1503+**187**	551+551	**56**
KRK	7666+10196	**7059+7051**	551+708	**92**
KQK	**4556**+6998	7605+**5961**	551+1076	**84**
KPPK	435415+**442704**	**384915**+610687	**6216+11488**	20392
KNNK	**816+688**	1358+1441	816+688	**230**
KBBK	**97658+128747**	249216+205549	672+4248	**230**
KRRK	**49533+172384**	135331+201764	647+1060	**68**
KQQK	**41050+116746**	141855+181425	647+5768	**68**
KPKP	1339738	**1116299**	**122913**	213592
KNKN	**649**	2449	649	**108**
KBKB	**689**	13989	689	**186**
KRKR	**143348**	189818	11357	**7034**
KQKQ	**197637**	257632	42189	**6316**
KNKP	**501759**+1199859	618587+**1089865**	56557+107217	**139878**
KBKP	**178135+665425**	369787+775143	20729+53273	**45912**
KRKP	1432954+2348010	**1208181+1479273**	**31197+59133**	148706
KQKP	1310330+1721549	**1103045+1353920**	3925+53025	**26736**
KBKN	**649+649**	14063+2451	649+649	**134**
KRKN	428740+281969	**324996+179806**	**63985+67817**	134856
KQKN	508792+1056193	**413756+582758**	916+34616	**1250**
KRKB	**171964+69321**	216606+84137	24152+26136	**27710**
KQKB	490204+748957	**401622+505366**	3332+21564	**2150**
KQKR	861440+1098842	**562344+641119**	4341+14837	**10970**
KNPK	**1277110**+1383360	1547889+**1227423**	**14860+34944**	108612
KBPK	**1161319**+1341477	1511181+**1290857**	**11904+39508**	62768
KRPK	**999754**+1249964	1197199+**1167451**	663+5660	**516**
KQPK	**568604+818736**	886731+913558	663+10548	**304**
KBNK	571560+632333	**555601+479960**	3304+13200	**6596**
KRNK	322710+452632	**308291+367838**	647+3124	**406**
KQNK	**212251**+379578	284437+**302140**	647+8008	**104**
KRBK	**266818**+416216	278694+**357372**	647+6960	**456**
KQBK	**194934**+344210	273890+**329501**	647+12668	**104**
KQRK	**90796**+305152	211121+**296182**	647+4492	**68**
All 3-4 men	**13890243**+17409488	14814890+**15768949**	1039436	**970710**

So far it has only been possible to construct the simplest 5 men endgames, by the results so far are very good:

Endgame	Distance to mate	
	This thesis	Nalimov
KBBBK	**2267775+3007899**	4599013+6450288
KNNNK	**4479412+4599705**	5127367+6831165
KQQQK	**217341+969960**	1620714+4302377
KRRRK	**244943+1321303**	1942079+4665085

13.1.4 Probe speed

The prime objective by using OBDDs was to assure a very fast lookup operation. The goal has been to match the time needed for a state-of-the-art chess program to search a position. This means in the order of $\frac{1}{2}\mu s$. Let's emphasise this point by conducting a mind experiment.

We have a chess program which is capable of searching 1 million positions per second. In an endgame position it typically takes the program 5 times longer to do a ply $n+1$ search than a ply n search. We have two different representations of endgames. In A the endgames have a size of 30 MB and it takes 10 μs to perform a lookup. In B the endgames have a size of 45 MB, but it only takes 1 μs to perform the lookup. Now, the question is which representation is best?

To give the answer we have to understand the usefulness of having a separate table for each side to move. Let's assume we use the B representation but throw away all endgames with black-to-move. This way the 45 MB will reduce to little more than half — say 25 MB.[21] Every time the program wishes to probe the endgames in a position with black-to-move, it instead does a 1 ply search and indexes in average 5 positions with white-to-move. This takes no more than 5 times 1+1 μs and is therefore no slower than doing a single probe in the A representation. Probing positions with white-to-move takes only 1 μs. On average, the B representation where we keep only the endgames with white-to-move is faster than the A representation despite the 1 ply searches. Also, 25 MB is less than 30 MB, so it is a win/win situation. In fact, the A representation should only be chosen if even 25 MB is more than acceptable.

The table below compares the probe speed in the OBDDs with the probe speed in Nalimov's compressed tables. The first two lines give the probe speed in the corresponding uncompressed tables. This shows the time used by the indexing function etc., which is not part of the actual compression. Notice that the Nalimov tables have been loaded to memory so the access times are not slowed down by any disk reads — just to keep the comparison fair.

The tests have been performed by first initialising a long list of random indexes that corresponds to legal positions. Both positions with white-to-move and black-to-move are indexed.

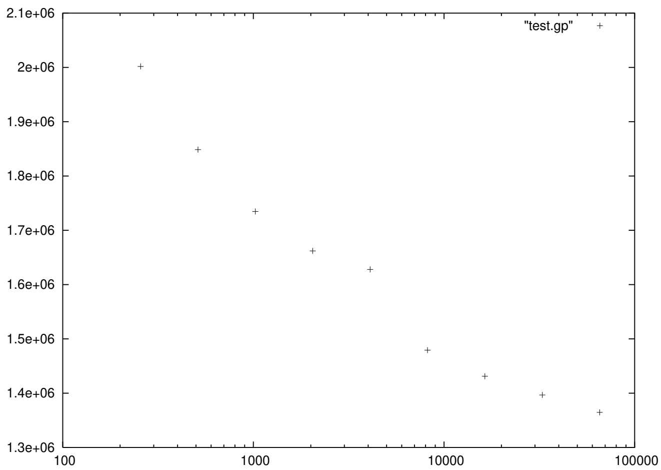
[21]Remember that in symmetric endgames like KQKQ we only represent the positions with white-to-move. Therefore we can't throw away anything for these endgames and the 25 MB reflects this by being a bit more than half of 45 MB.

Indexing	KBBBK	KRRRK	KRKP	KQKR	KQK
Uncompressed table	-	$0.28\mu s$	$0.28\mu s$	$0.24\mu s$	$0.20\mu s$
Nalimov, uncompressed	-	$0.82\mu s$	$0.74\mu s$	$0.46\mu s$	$0.29\mu s$
OBDD	$1.91\mu s$	$1.71\mu s$	$0.72\mu s$	$0.69\mu s$	$0.48\mu s$
Nalimov, compressed	$78.1\mu s$	$61.3\mu s$	$28.4\mu s$	$84.1\mu s$	-

The probing speed for the OBDDs are within range of the $\frac{1}{2}\mu s$. There is however a significant gap in the probing speed between the 4 and 5 men endgames. The only explanation I can think of is that a lot more cache fails are produced for the 5 men endgames. But the KRRRK endgame stored as an OBDD is actually smaller than the KQKR endgame. Also, it is unlikely that the difference should be more than 10 cache fails (the heights of the OBDDs representing the KRRRK and KQKR endgames are 28 and 22 respectively).

Combined with the level of compression the OBDDs achieve, the results are as good as I aimed for. It is about 40 times faster to index an OBDD than to decompress a 8 K-byte block. But what if we tried to change the block size used by Datacomp? By assuming that the decompression time is linearly dependent on the block size, the this should be reduced to 200 bytes to match the probe speed offered by OBDDs. I have tried to compress the KRKP endgame using different block sizes. The results are shown below.

Block size	KRKP
65536	1364599
32768	1396658
16384	1431193
8192	**1479273**
4096	1627879
2048	1662050
1024	1734596
512	1848682
256	2001733



A reasonable estimate is that the compressed files would blow up by 50% if the access time should match that of an OBDD. This means that even if this kind of compression was optimised like it was done in section 2, then it would unlikely be a match for the OBDDs.

13.2 Reducing to win/draw/loss information

For WDL endgames the mapping of don't care entries becomes much more important than was the case for DTM endgames. Programs like gzip, bzip2 and Datacomp do not exploit the don't care entries and are therefore not competitive.

For the checkers program *CHINOOK*, they used a block compression based on run length encoding. As mentioned in section 7 all broken positions (don't cares) was mapped to the frequently most occurring value. They tried other representations as well, but concluded that run length encoding worked best. They used a block compression to support a fast lookup operation and paired it with a LRU cache — much like the way Nalimov's endgame tables work.

I have implemented a run length encoding of the WDL endgames to compare with the approach. It is not a block compression and it does not support fast lookup, but this will only make it compress better. I have tested it in 4 different versions. In 2 of them don't cares are mapped to the frequently most occurring value (they are marked "+"). In 2 of them a Huffman encoding of the run length is used (these are prefixed "H") and in the 2 others an Elias coding is used.

The results below show that even the best version of run length encoding is nowhere near the performance of the OBDDs. For WDL endgames OBDDs work really well.

Compression method:	KRK	KQK	KPK	KRKP	KQKR
RLE-	5229+2856	7278+3019	12576+12231	-	-
HRLE-	2644+1841	4694+1945	8902+9280	-	-
RLE+	**4**+2297	**4**+2395	9874+10932	242315+729529	17326+309696
HRLE+	5+1213	5+1262	7413+8498	177521+553326	10430+199697
OBDD	551+**708**	551+**1076**	**1172**+**1336**	**31197**+**59133**	**4341**+**14837**

13.2.1 EgmProbe 2.0

EgmProbe is a recent program (2003-4) that uses decision trees to represent the endgames. I have only found the short description quoted below.

> Endgame models (egm) are used when there are 5 or less men on the board ... Egms are highly compressed win/loss/draw tables. On a machine with enough ram to fully load the tables they can be probed without any disk reads. On my 733MHz PIII when relevant egms have been loaded it can make 20-320 Kprobes/sec depending on model complexity.
>
> If there is too little ram a cache is used. It is possible to make an unverified "model" probe. ...
>
> Decision trees grown with an ID3-style algorithm is used to predict if a position is won, lost or drawn. In unclear cases small searches can be performed. An accuracy above 99% is reached for most endgames. A table with the exceptions is compressed with run length encoding and huffman codes. The exception table is probed on the fly, without generating any decompressed tables. Eugene Nalimov's EGTBs are used as an oracle during generation of the egms.

The table a few pages back gives a throughout comparison of all 3-4 men endgames. EgmProbe is able to compress all 3-4 men to 948 Kbytes. This is slightly less than the 1015 Kbytes used by the OBDDs. The 20 Kprobes/sec mentioned as the lowest probe speed translate to around 15 μs/probe on a 3 GHz Pentium 4. Also, it is not clear what he means by "In unclear cases small searches can be performed" or by "small". Anyway, even in its most optimistic interpretation this representation is still considerably slower than the OBDDs and only slightly smaller.

14 Conclusion

In this thesis I may have focused on too many different subjects. As my primary objective I wanted a working solution in the end. Therefore every possibility of optimisation was tested but not all of them thoroughly. In its current state the endgame tables that are compressed as OBDDs are very useful, but there are obvious places to continue the work.

14.1 Future improvements

We have seen that all 3-4 men endgames are representation about as compactly as OBDDs as in the best competitive format, while at the same time offering much higher probing speeds. A small task is to extract the necessary code to perform the probing operations, put it into a single source file and make it public available. I will however postpone this until I'm completely sure about the format. My doubt concerns mainly whether to use the clustering or not. Unless some new more encouraging results are produced, I will not include it.

So far the construction of the OBDDs only works for the simplest 5 men endgames if distance to mate information is stored. The primary problem is a lack of memory. But even if memory was not a problem, it would still take in the order of days (!) to compress an endgame like KRPKQ. I have worked quite hard to make the implementation efficient, but there is still work to be done.

14.2 Applicability

The 3-4 men endgames generated have such a small size that they can easily be adapted by any chess program. What is at least as interesting is that the potential of the 5 men endgames with WDL information. For the 3-4 men endgames there was a reduction in size of a factor of 30 when the DTM information was stripped to merely WDL information. If this is also the case for the 5 men endgames, then they will get down at around 250 Mbytes — or 125 MB if only one side to move is used. Even though a chess program would probably need more information for a few of the endgames, it would still be reasonable in size.

A Computer chess

This section outlines how a typical chess program would be implemented. This thesis tries to achieve a compact representation of chess endgames that is feasible to probe during high depth searches. The example from section 13.1.4 show that the trade-off between probe speed and the size is highly dependent on properties of the chess program — to construct useful endgame tables you have to know how they are going to be utilised.

Let's start by describing the very basic ingredients. It is common to speak of a chess engine instead of a chess program. A chess engine is the part of the program where the move to make is being computed. Everything else is of no interest here.

A chess engine work by estimating the outcome of each applicable move by performing a local search. The most trivial solution is to do a negamax search to some fixed depth as exemplified in section A.1. This can be improved by introducing transposition tables, alpha-beta search etc. which will be mentioned in section A.2.

For the leaf nodes in this search tree, where no game theoretical information is available, we have to invent some heuristic measure. An evaluation function computes this value — typically as a weighted sum of some easy recognizable features:

$$\begin{gathered} eval : P \mapsto \mathbb{Z} \\ |eval(p)| < \text{Value of a won position} \\ eval(p_1) > eval(p_2) : p_1 \text{ looks more promising than } p_2 \end{gathered}$$

The evaluation and search function together form the basic building block of almost any chess program, the rest is just optimisations of the search algorithm and fine tuning of the evaluation function.

A.1 Negamax search

A negamax search tries every sequence of moves to some fixed depth. Whenever this depth is reached, the heuristic evaluation is applied. In this case the evaluation function is merely a weighted sum of the pieces [22]. Chess is a 0-sum game — what is good for the opponent is bad for you. The function `Negamax` gives the value of searching the current position relative to the side-to-move. The value of the move can thus be decided by making it, calling `Negamax` and then remember to change sign to get the value relative to the player making the move.

[22] These values are a commonly accepted rough measure of piece material.

```
int PieceValue(Piece piece) {
    switch(piece) {
    case PAWN: return 1;
    case KNIGHT: return 3;
    case BISHOP: return 3;
    case ROOK: return 5;
    case QUEEN: return 9;
    case KING: return 0;
    }
}

int Evaluate() {
    int sum = 0;
    For all pieces p in current position do begin
        if (p belongs to side-to-move)
            sum = sum + PieceValue(p)
        else
            sum = sum - PieceValue(p)
    end;
    return sum;
}

int Negamax(int depth) {
    if (game over) then begin
        if (side-to-move lost)  return -WIN;
        if (drawn)  return 0;
        if (side-to-move won)  return WIN;
    end;
    if (depth = 0) return Evaluate()

    int value = -INF;
    For all legal moves m in current position do
        execute move m;
        value = max{value, -Negamax(depth-1)};
        undo move m;
    end;
    return value;
}

Move getMove(int calc_to_depth) {
    int value = -INF;
    Move best_move;
    For all legal moves m in current position do begin
        execute move m;
        int tmp = -Negamax(calc_to_depth);
        if (tmp > value) then begin
            value = tmp;
            best_move = m
        end;
        undo move m;
    end;
    return best_move;
}
```

A.2 Improving techniques

Even though the engine just presented is very simple,[23] it will not be trivial to beat for a novice chess player. In a typical mid game position there are around 35 legal moves. A ply 4 search (2 moves by both sides) would hence have the complexity of evaluating approximately $35^4 = 1500625$ positions which is feasible on a modern computer. This engine would play flawlessly in the sense that it would detect any potential traps (i.e. captures) within the 4 ply. However it would have no positional understanding whatsoever — any position with the same amount of material would be equally good.

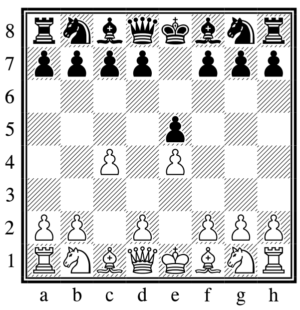
What follows now is a short description of the techniques that raises this simple engine to the level of a state-of-the-art engine. For a more throughout description I recommend the lecture notes of David Eppstein for the course "Strategy and board game programming" [13]. There are basically 2 ways to perform the search as described by Shannon in 1950 [17]:

Shannon Type A: Perform a full-width search to some fixed depth.

Shannon Type B: Perform a selective expansion of certain lines, using knowledge to prune uninteresting branches.

The algorithm just presented is a pure Type A strategy. There are basically 2 ways in which one can perform a theoretical sound pruning of this game tree, ie. one that guarantees that the move found is the same or an equally good one. These are:

Transposition table: Implemented by a hash table [13]. The position shown below can be reached in two different ways; either by the sequence "1. e2-e4 e7-e5 2. c2-c4" or by "1. c2-c4 e7-e5 2. e2-e4".



By remembering the values of the positions encountered, identical subtrees are evaluated only once. It has also proven useful to store partial information such as values of pawn structures [20].

[23] Implementing the move generator would be the hard part.

Alpha-beta search: This branch and bound technique described in [13] intuitively works by only considering the best move for one of the players.

Let's assume player A luckily starts by searching his best move, giving the value V_a. Now we don't need to know the exact value of the other moves. We are satisfied by knowing that they are no better than V_a. Hence instead of examining all B's replies to each of A's remaining moves, we only need to examine one sufficiently good one. If the moves are examined in the best-first order, then it will essentially allow the search tree to be twice as deep without affecting its size.

The catch is that you do not know what the best move is (otherwise there would be no need to perform the search). The alpha-beta search does help even if a random move ordering is used, but the close the move ordering is to the best-first order, the better. There has been a lot of research in how to select the move ordering:

Iterative deepening: The idea is that the best move returned by a depth n-1 search will likely also be the best move returned by a depth n search[24].

Killer moves: A good move in a certain depth is likely to be good again (see [20]).

History moves: A good move is likely to be good again. Give higher priority to moves searched deeper (see [20]).

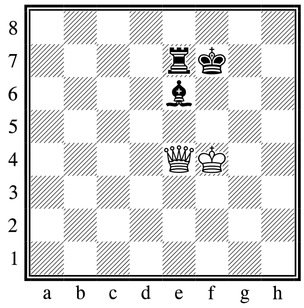
Aspiration search, MTD(f), PVS: These are modifications of alpha-beta that puts some assumptions on the value of the root position. This allows for some further pruning. If however these assumptions does not hold, a research is necessary. Therefore a transposition table is a must. See e.g. [13] or [18].

Over the last 10 years focus have changed slightly towards the Shannon Type B strategy as experimentally shown robust ways of performing this selectivity have emerged. These includes:

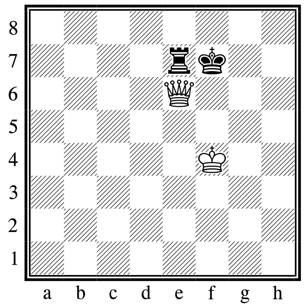
Forward pruning: A very popular forward pruning technique is called null move [20]. This technique greatly reduces the amount of time used to search stupid moves like Qxe6 below.

[24]This is studied in [20], where experiments suggest that the probability that the best move will change decreases from 35-40% at n=2 to 15% at n=14 (on a state of the art cpu engine)

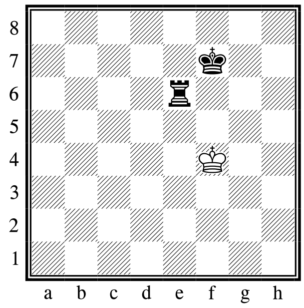
Before



Queen sacrificed



Queen captured



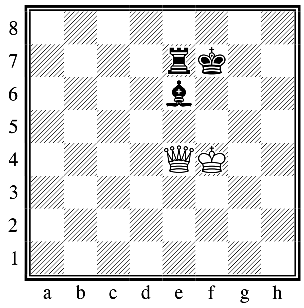
If there was some hidden trap in this sacrifice and if white had the ability to perform 2 consecutive moves from the last position he would almost for sure be able to exploit it. Hence if we temporarily allow white this double move in a search to a reduced depth[25] and he is still not able to achieve any compensation, then there is no need for the full depth n search.
Care should be taken in endgames where zugswang situations is not unlikely to occur.

Search Extensions: [13] If a player only have one legal move, then it can be argued that the position is interesting and should hence be searched one ply deeper. Also, the extra ply is partly "paid" by the fact that the tree doesn't split from this position. Also other interesting features may justify this extra ply.

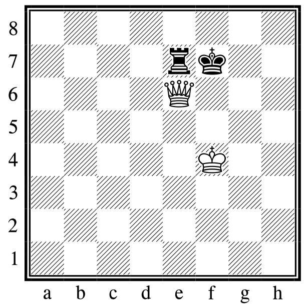
The general view upon the evaluation function is that it should be kept simple and hence very fast to evaluate [15]. With state of the art search techniques a four times slower evaluation function is traded for an extra ply, and this is hard to justify. Methods for improving evaluation function are:

Quiescence search: This is a solution to The Horizon Effect problem. Consider how the simple engine would evaluate the middle position below (black to move):

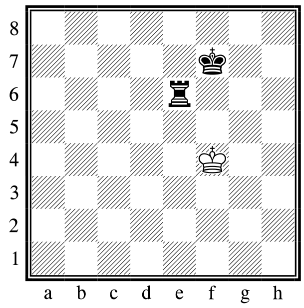
Before the last move:



Evaluated position:



The quiescence search stops at this position:



[25]Experiments suggest that reducing the depth with 2 ply gives the best balance between the saving in number of nodes and the incurred tactical weaknesses.

By just counting material the engine concludes that white is in the lead (white material is 9+0 versus 5+0 for black). Any decent human player would however judge the position lost for white, since black rook can capture white queen and that king and rook vs king is a general win.

One solution, the quiescence search, is to continue the search, but restricting it to "interesting" moves. These could e.g. be captures and checking moves, but only in some restricted form or otherwise the search tree would blow out of dimensions. Another counter measure would be to apply some approximate static exchange evaluation.

It should be noted that if this problem is not handled somehow, then the last move in the line of play would nearly always be a capturing move. If the search depth is even this implies that the engine is unlikely to put its pieces into play (then these could be hit in the last move).

Selecting relevant features: Which features of the position is computationally easy to recognise and relevant enough to contribute to the evaluation function? This includes piece material, ability to castle, passed pawns, pawn shelter, threats, etc[26].

Hand fitting constants: If you are a skilled player, then by playing against the engine you will found out e.g. how much bonus a pawn on the 6'th rank should receive.

Automated learning: Like hand fitting the constants. The known frameworks will not be able to make as intelligent adjustments, but it could do thousands of times more.

Other topics include:

Opening library: A lot of strategies for evolving the pieces in the early stage of the game have been developed. There is even an *Encyclopedia of Chess Openings* (ECO, [8]) containing a comprehensive list of opening listed from a00 to e99.

The early stage of the game is almost entirely based on gaining positional development — a concept which is difficult to learn a computer. It is easy just to let the engine follow one of the theoretical strong lines of play. Some care have to be taken however — it is not optimal to follow a line of play contradicting the intuition of the engines (i.e. evaluation function), otherwise you risk that the engine just takes back the moves when it runs out of the opening line (or even worse, don't know how to get its sacrificed pawn back).

[26] See e.g. the evaluation function used in Gnuchess for a simple example [10]

Endgame tables (also called tablebases): Given a set of pieces e let P_e be the set of positions containing exactly these pieces. Chess have a huge state space:

$$10^{43} \lessapprox \sum_e |P_e| \lessapprox 10^{50}$$

However when we restrict s to few pieces it becomes feasible to list the distance to mate value of all positions in P_e. The distance to mate value give more detailed information than merely stating if the position is won, lost or drawn — when combined with a 1 ply search it provides progress towards a mating position.

N	Number of endgames with N pieces ($\|\{s \mid \|s\| = N\}\|$)	Total size of endgames in Nalimov's compressed format
2	1	-
3	5	62 KB
4	30	29 MB
5	110	8 GB
6	365	estimated 1 TB
7	1001	-
8	2520	-
9	5720	-
10	12190	-

The effect of including n-men endgame tables that can be probed efficiently[27] is that

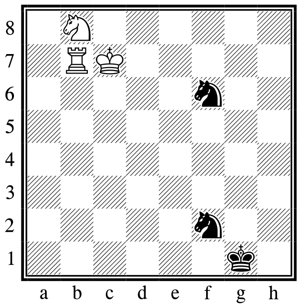
1 The engine will play flawlessly with at most n pieces left.

2 There is no need to search any further in a position that contains at most n pieces. This will cause a lot of pruning in positions with only slightly more than n pieces left.

Bitbases: Like endgame tables, but contains only information about if the position is lost, drawn or won. Two bits are needed per position, but some care has to be taken. As an extreme example the position below is a sure mate in 262 for white, but it is very unlikely that an engine could enforce a win only knowing if the positions are lost, drawn or won:

[27] Storing the tables on a hard disk at the cost of ≈10 ms per probe is `not` considered efficient.

White moves and wins in 262



Efficient implementation: This of cause also has to be taken into account. An increasingly popular choice is to use bitboards to represent the board [14]. This way all the possible moves for e.g. a queen can be found at once. Also, 64 bit processors are becoming increasingly common — especially after the introduction of Athlon 64. A board contains 8x8 squares, hence a 64 bit integer can be used to contain one bit of information of each square.

B The program

B.1 Getting started

I maintain a home page for my master thesis at
`http://www.daimi.au.dk/~doktoren/master_thesis/`
The program can be downloaded from the page
`http://www.daimi.au.dk/~doktoren/master_thesis/download.html`

B.1.1 Compiling

Change the value `Endgame_directory` in the file `default.set` to the directory where you want to store the endgame tables. The values in this file is read whenever the program starts and it is even possible to change them when the program is running.

The values which have an attached comment like `//NDEBUG = ...` are only used when the program has been compiled without defining NDEBUG.

No setting concerning endgames (those starting with `Endgame_`) affects the representation of the OBDDs. Hence an endgame stored as an OBDD with e.g. `Endgame_calc_sifting` turned on still works if it is later turned off.

Some further adjustable settings are located in `experimenting.hxx`. After having modified one of these it is necessary to recompile from the ground, and also to rebuild the endgame tables/OBBD's (the program will probably crash if you try to use the old endgame files).

B.1.2 The Makefile

Use `make` to compile the program.
Use `make clean`, `make` to recompile the program if you have changed the Makefile.

In the file `Makefile` you can switch the debugging mode and decide whether it should be compiled to use with XBoard.
If `CFLAGS` below is suffixed with `_DB` the debugging mode is on, and if it is suffixed with `_XB` it will be compiled for use with XBoard.

```
%.o: %.cxx
        $(CXX) -c $(CFLAGS) $< -o $@
```

B.2 Interaction

The interaction with the program depends on whether it was compiled with `XBOARD` defined.

B.2.1 With XBoard

With XBoard you can only use the program to play chess against. This has no relevance in connection with this thesis, but if you already have XBoard installed (it is at the university), then you can start it like this:
`xboard -size Medium -tc 1 -inc 1 -firstChessProgram ./chess &`
From now on I will assume that the program has not been compiled to use with XBoard.

B.2.2 Without XBoard

Use `./chess` or `./run` to run the program (`run` makes sure the program is properly killed if something goes wrong).

The interaction is unfortunately quite simple in the form of typing command lines.

Because the program was originally designed to work with XBoard, you have to type the command `load` as the first thing when the program starts. `load` reads the commands given in `default.com` — the first (mandatory) of these are identical to the interaction protocol with XBoard and are just some that the program expects to receive.

What happens next depends on how which extra commands are specified in the current version of `default.com`. If you type `help` you will see a small list of supported commands. Warning: Due to debugging etc. probably not all commands works or do what their description says.

I think its better to illustrate, how to use the program, by a number of small examples rather than to describe every command in more or less detail.

However one thing that should be mentioned already now is that the command `enter [structure]` redirects the commands to a specific part of the program. As an example `enter endgame database` redirects all commands to the endgame part of the program. Also, the command `leave` cancels this redirection and sends all commands to the main procedure. If a command is redirected to a structure that doesn't recognise it, it will go to the main procedure.

B.3 Examples

B.3.1 Playing chess

After you have started the program you will see the initial position of a chess board. Let's first view some available commands. Type

```
load
help
help command
help xboard
```

Often the commands can be shortened by only typing the first letter of each word with no spaces between. In the above case this reduces to (the initial `load` has to be typed exact).

```
load
h
hc
hx
```

However in this section I will consistently use the long versions.

The program expects the moves in the PGN format. If you are in doubt about the details of this format, the command `pml` will list all legal moves in the current position. Let's play Queen's gambit:

```
pml
usermove d4
usermove d5
usermove c4
```

`usermove` can be shortened to a dot. We undo the moves and repeat them using this shortened notation:

```
undo
,
,
.d4
.d5
.c4
```

Typing “,” repeats the last command. Getting tired from all this playing, let’s save the game and quit.

```
save greatgame.pgn
system cat greatgame.pgn
save greatgame.fen
system cat greatgame.fen
quit
```

The command `system` sends the rest of the line to the system prompt.

Start the program and type

```
load
load greatgame.pgn
undo 3
load greatgame.fen
undo
```

The last `undo` fails because the FEN description doesn’t include move history.

B.3.2 Playing against chess engine

The chess engine has been split into 2 parts — one performing the search and one performing the evaluation. Currently there is 3 versions of each:

Search version 1: Implements a simple alpha beta search.

Search version 2: Alpha beta search extended with transposition tables.

Search version 3: Uses most state-of-the-art techniques: Alpha beta search, transposition tables, iterative deepening, killer moves and move history. When compiled with debugging mode, most of its behaviour can be specified in the file `default.set`.

Similarly, 3 versions of the evaluation function can be used:

Evaluation version 1: Count pieces according to the weight specified in `piece_values.hxx`

Evaluation version 2: Counts pieces in a clever way — the value of a piece now depends on where it is positioned on the board. Also, the game is divided into 3 game phases — opening game, mid game and endgame.

Evaluation version 3: Considers some other features also. Namely castling capabilities, pawn formations (double pawns, passed pawns), measurements of which player “controls” the board — especially the centre and near the kings. Etc.

When you start the program it will inform you which combination of search and evaluation function is currently used. I.e. you will see something like

```
Eval_1 constructor called.
Search_1 constructor called.
```

Let's start a little game

```
load
search version 3
evaluation version 3
.d4
go
.a3
```

The command `go` sets the computer to play the current colour.

The last move was probably stupid (of course depending on how the cpu played), so we wish to retract it.

```
remove
.e4
```

We can also take back the move by using `undo` twice, but then we have to reactivate the computer.

```
undo
undo
.e4
go
```

If you start a new game, then the computer will automatically be set to play black. We can disable the computer by typing `force`.

```
new
force
.d4
.d5
```

B.3.3 Generating and indexing endgame database

First, make sure the value `Endgame_directory` in the file `default.set` is set correctly. This example assumes that the directory is empty. Let's build the KRK endgame

```
load
enter endgame database
help
load table KRK t t
```

The construction of the KRK endgame shouldn't take long — especially if debugging mode is off. Remember that the command `enter endgame database` redirects all following commands to the endgame database part.

The first `t` (true) argument to `load` expresses that we wish to construct the endgame table if it can't be found in the directory specified by `Endgame_directory`. The second says that we are only interested in the part of the table that represents white-to-move positions (the btm part is still constructed and saved though).

Now let's test this endgame table.

```
loadfen 8/8/8/8/k7/3RK3/8/8 w - -
index database
```

The command `index database` gives the value of this position and also the value of each of the legal moves. Notice that only the current position is found in the table, because only the wtm part of the table is loaded. Let's fix this.

```
load table KRK
index database
```

We have already constructed the KRK endgame for both sides, so this time the we only need to load the endgame.

These values can be compared with those given by Nalimov's endgames. This server contains all endgames with at most 5 pieces:

`http://mx2.lokasoft.com/tbweb/tbweb2.asp`

B.4 Examining index functions

Download the endgame KRRK and KPKP to your endgame folder. The first contains positions with castlings and the second positions with en passant.

```
load
load table KRRK
loadfen 8/8/8/8/8/8/7R/1k2K2R w K -
index database
```

If you try to index this position on the Nalimov server, you will see that the castling capability is ignored.

Let's apply the function $I_{\{K,R,R,k\}}$ on this position.

```
show table index
.O-O#
construct from table index KRRK 306688 0
```

(`show table index` can be shortened `sti` and `construct from table index` can be shortened `cfti`). It reports that the position is given the index 306688. We check mate black, and then use the index to reconstruct the position.

The reader can try to apply $I^{-1}_{\{K,R,R,k\}}$ on some random indexes — some of them do not correspond to any legal position.

```
construct from table index KRRK 123456 1
construct from table index KRRK 1234567 0
...
```

Let's try a position with en passant. This position is won by capturing the en passant pawn. Otherwise it would be lost.

```
loadfen 8/8/8/8/2PpK3/8/8/3k4 b - c3
index database
show table index
construct from table index KPKP 4112149 0
```

When you ask for the index of this position, it reports that side-to-move is white (0). This is because it is a symmetric endgame. The reconstructed position has been mirrored horizontally because of the change of side-to-move and the colour of the pieces. Also (if `SWAP_KINGS` is 1), the position has been mirrored vertically such that black king is in the a1-d8 rectangle.

B.5 Replicating the experiments

Most of the optimisations of the OBDDs can be turned on and off — many of them even without recompiling the program.

KBBK experiment: Implemented in the file `endgame_KBBK.cxx` which is no longer part of the program.

Compact OBDD representation: `BDD_USE_COMPRESSED_ENCODING` is located in `experimenting.hxx`. This setting controls whether the compact representation should be used. This setting affects the format of the endgames stored as OBDDs, so the existing ones will have to be deleted/removed, and new ones created.

Mapping of wildcards: Whether the wildcard mapping algorithm should be used when constructing an OBDD can be specified in `default.set`.

Enumeration of board squares: In the file `default.set`, the enumeration used for each kind of piece can be specified. Appendix G specifies the different possible enumerations.

Bit order, Sifting algorithm: Can be specified in `default.set`.

King positions: You can define which king should be bound to respectively the a1-d1-d4 triangle and the a1-d8 rectangle. Change the setting `BOUND_KING` in `experimenting.hxx`. It affects both the format of the uncompressed tables an those stored as OBDDs, so both kinds will have to be rebuild after deleting/moving the old ones.

Clustering of OBDDs: Can be specified in `default.set`.

The rest of the section will demonstrate how to do a small experiment for a couple of the features that can still be turned on and off.

B.5.1 King positions

Let's compare the size of the KPK endgame by restricting white respectively black king to the a1-d8 rectangle. Open the file `experimenting.hxx` and check whether `BOUND_KING` has the value 1. If not, set it to 1 and run `make clean`, `make`. If the KPK endgame is in your endgame directory either in compressed or uncompressed form, erase it. Now, let's build build and compress the endgame. `BOUND_KING` was 1, so black king will be restricted to the a1-d8 rectangle.

```
load
enter endgame database
build table KPK
build bdd KPK
quit
```

Now check the size in your endgame directory and delete KPK_*. Modify the value of `BOUND_KING` to 0, run `make clean`, `make` and repeat the above execution of the program. By comparing the size of newly generated KPK_*bdd the reader will see that it was better to restrict the black king.

B.5.2 Clustering of OBDDs

This experiment takes a bit time to perform. Open `default.set`, set `Endgame_clustering_method` to 0, and save. Delete KBPK_*bdd from your endgame folder.

```
load
enter endgame database
build bdd KPK
quit
```

Check the size of KBPK_*bdd and then delete or remove it. Open `default.set`, set `Endgame_clustering_method` to 2, and save. Repeat the above execution, but type the command `print` before `quit`. If the reader looks for KBPK, then he should hopefully find the text `(num c.=2)` for both `bdd[0]` and `bdd[1]` meaning that 2 clusters were produced.

C Generating retro moves

Below is given a list of the special cases one has to be aware of when implementing a retro move generator. The reader should have read section 12.3 before continuing below. The next 5 pages elaborates and shows examples.

Basic stuff

- Each retro move can have been a normal move or a capture of one of the opponents pieces, excluding the king. Remember that special capture rules apply for pawns, and that the pawn could not have been captured on rank 1 or 8.

Castling

- Undoing a castling move.
- Restricting king and rooks with castling rights against moving.
- Capturing a rook with castling rights
- Moving a king or rook that might previously had castling rights.

Pawnless position

- With no castling rights, an undone pawn capture must be duplicated into 4 variants — one for each transformation of the board.
- When castling rights and a pawn are introduces at the same time by undoing a move, only vertical reflection or none is allowed.

En passant

- Undoing an en passant move.
- When en passant is possible, no other move may be undone.
- Each move undone to a position, where en passant could have been possible, must also exist in a version for each possible en passant.

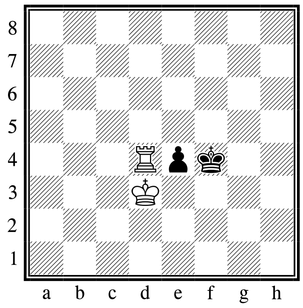
Pawn promotion

- Each piece that is different from a pawn and a king and is placed on the opponents back line, could just have been promoted from a pawn. In a pawnless position with no castling rights this pawn promotion could have taken place anywhere on the border (Remember the example from section 12.3).
- A piece on a corner square could have been promoted from a pawn in 2 differently transformed versions of the board (assuming a pawnless position with no castling rights).

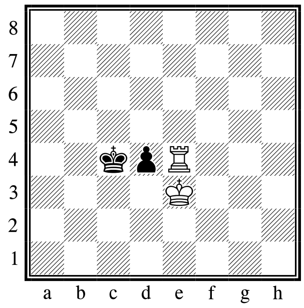
C.1 A move may incur more symmetry

When making a move, more symmetry can appear. As an example consider the 4 positions below — all with white to move. Each position have a pawn but no castling right is involved. Hence the position is only symmetric with respect to vertical reflection. However, if white rook captures the black pawn, then the resulting positions also gain symmetry according to horizontal and diagonal reflection.

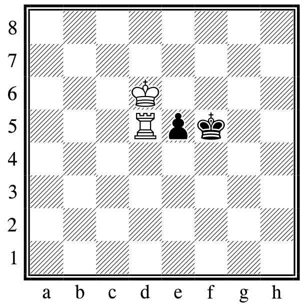
Whites moves Rxe4



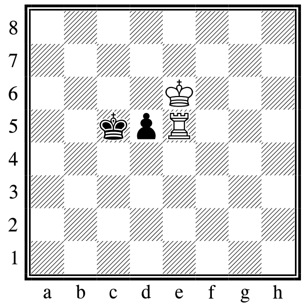
White moves Rxd4



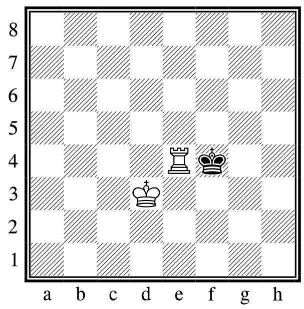
Whites moves Rxe5



White moves Rxd5



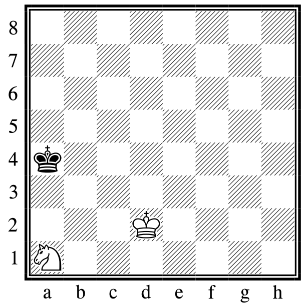
The effect is that the 4 above positions collapse to the position below after having made the capturing move (white king is now being restricted to the a1-d1-d4 triangle).



C.2 Much different positions combine

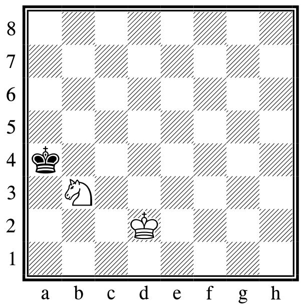
Consider the position below.

Black to move

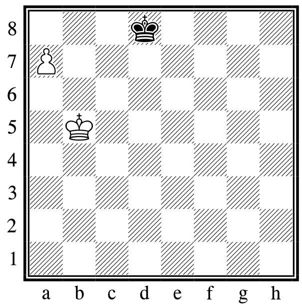


This position can be reached in a number of ways — a lot of them being less obvious. Below is shown some examples.

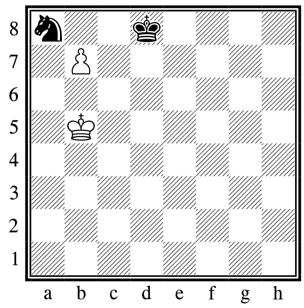
Whites moves Na1



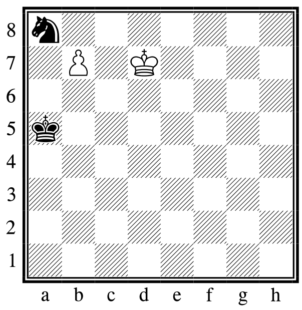
Whites moves a8=N



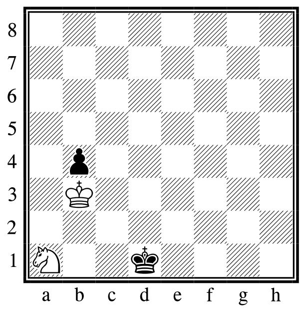
Whites moves bxa8=N



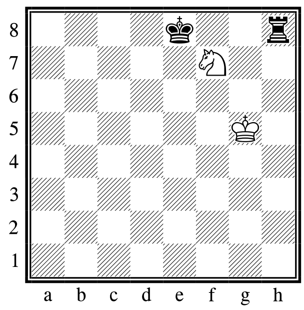
Whites moves bxa8=N



White moves Kxb4



White moves Nxh8



Note that there is 2 versions of the last position, one in which the black king has its short castling right maintained and one in which he don't.

C.3 Restrictions imposed by castling and en passant rights

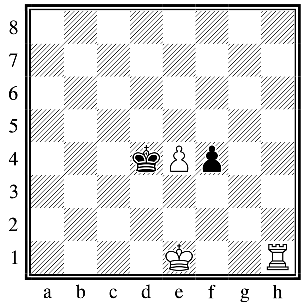
Consider the 2 kinds of moves; castling and an en passant capture of a pawn. These moves can be viewed as bonus moves that are available to the player, given that some requirements are satisfied:

castling: The king and pawn involved in the castling may not previously have been moved.[28]

en passant: The pawn being captured en passant must have moved 2 forward immediately before this move.

These requirements impose restrictions on the moves leading to a position where such a bonus move is possible. As an example, consider the position below.

Black to move.
White has its short castling right maintained.



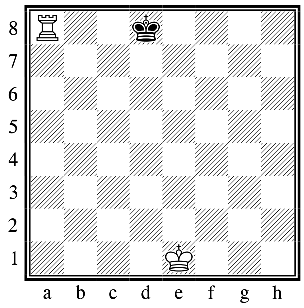
In the last move, white moved either its pawn, king or rook. White still has a castling right, so it can't have moved its king or rook. Hence the pawn must have been moved. It couldn't have moved e3-e4, because the pawn from e3 would be able to capture blacks king. Also, the move e2-e4 is rejected as this would give black the option to capture it en passant. Hence the 10 possible moves leading to this position is the 5 dxe4 captures and the 5 fxe4 captures — one for each non-king black piece.

[28]Also, the king is not allowed to pass any threatened squares, but that is not relevant for this discussion.

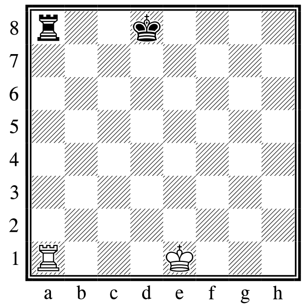
C.4 Castling is nasty

Some positions without castling capabilities can be reached by both a position in which white can castle and a position in which black can castle. Example.

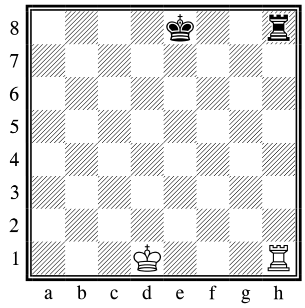
Original position
black to move.



White can castle,
but moves Rxa8+

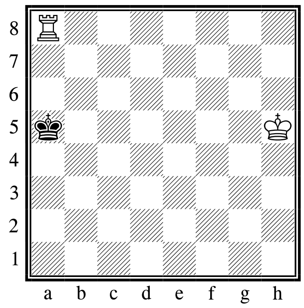


Black can castle.
White moves Rxh8+

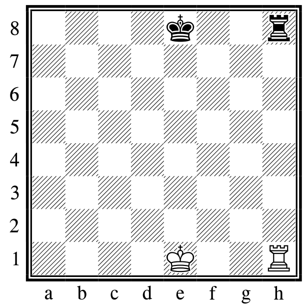


As each added castling capability depends on a unique transformation of the board, the added castling for white and black may be mutually exclusive as is the case above. Below is shown an example in which both white and black had their castling right before.

Original position



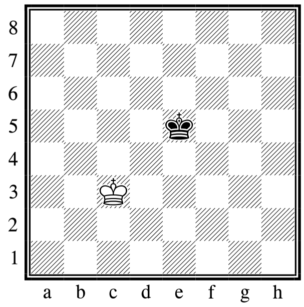
White and black can castle,
white moves Rxh8+



C.5 List of retro moves may contain duplicates

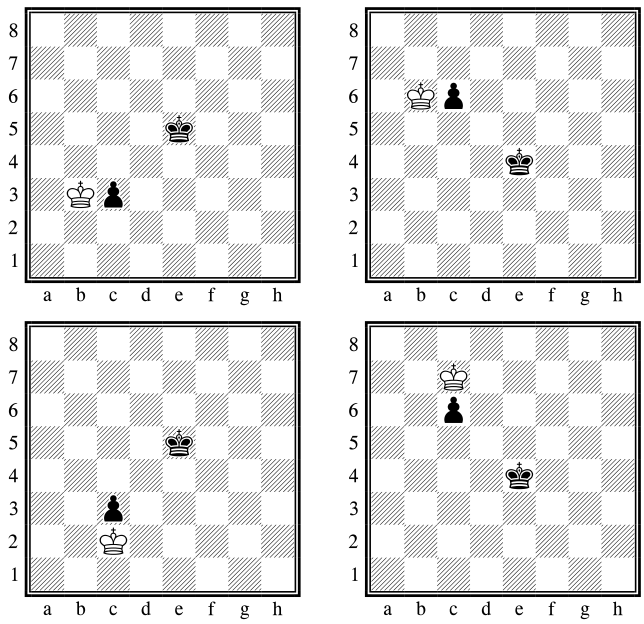
Consider the position below with black to move. Each last pawn capturing move for white king exists in 2^2 versions — each being related to a different transformation of the board.

Black to move.



The reason is that by putting a pawn back on the board, 2 of the 3 degrees of symmetries are lost. Hence the one position with just the 2 kings actually represents 8 different positions, while each position with a pawn represents just 2. Therefore we need 4 pawn positions to bring the count up to 8.

One of the possible pawn capturing moves to undo is Kb3xc3. This move, paired up with each of the 4 transformations of the board, gives the positions shown below.



Now, because the position with just the 2 kings is symmetric in the a1-h8 diagonal, the pawn capturing move Kc2xc3 will be identical to Kb3xc3 and result in the same 4 positions above.

By analysing the position further, these duplicate moves could be avoided. However, since they do no harm I have not bothered doing this.

D FIDE chess rules

This is a selection of the official FIDE chess rules [5]. Not all of the rules are relevant in connection with this thesis (e.g. a rule for the case where a player accidentally touches a piece), hence these rules are shortened to *irrelevant* here.

D.1 The nature and objectives of the game of chess

1.1 The game of chess is played between two opponents who move their pieces alternately on a square board called a 'chessboard'. The player with the white pieces commences the game. A player is said to 'have the move', when his opponent's move has been made.

1.2 The objective of each player is to place the opponent's king 'under attack' in such a way that the opponent has no legal move which would avoid the 'capture' of the king on the following move. The player who achieves this goal is said to have 'checkmated' the opponent's king and to have won the game. The opponent whose king has been checkmated has lost the game.

1.3 If the position is such that neither player can possibly checkmate, the game is drawn.

D.2 The initial position of the pieces on the chessboard

2.1 The chessboard is composed of an 8x8 grid of 64 equal squares alternately light (the 'white' squares) and dark (the 'black' squares). The chessboard is placed between the players in such a way that the near corner square to the right of the player is white.

2.2 At the beginning of the game one player has 16 light-coloured pieces (the 'white' pieces); the other has 16 dark-coloured pieces (the 'black' pieces): These pieces are as follows:

A white king, usually indicated by the symbol ♔
A white queen, usually indicated by the symbol ♕
Two white rooks, usually indicated by the symbol ♖
Two white bishops, usually indicated by the symbol ♗
Two white knights, usually indicated by the symbol ♘
Eight white pawns, usually indicated by the symbol ♙
A black king, usually indicated by the symbol ♚
A black queen, usually indicated by the symbol ♛
Two black rooks, usually indicated by the symbol ♜
Two black bishops, usually indicated by the symbol ♝
Two black knights, usually indicated by the symbol ♞
Eight black pawns, usually indicated by the symbol ♟

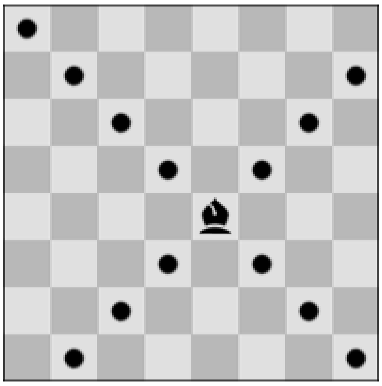
2.3 The initial position of the pieces on the chessboard is as follows:

2.4 The eight vertical columns of squares are called 'files'. The eight horizontal rows of squares are called ranks'. A straight line of squares of the same colour, touching corner to corner, is called a 'diagonal'.

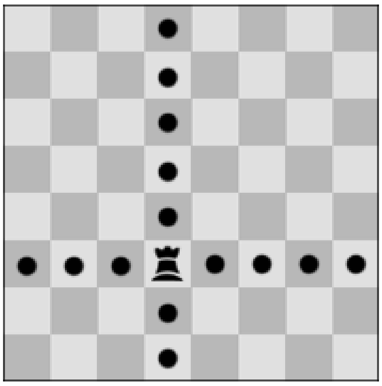
D.3 The moves of the pieces

3.1 It is not permitted to move a piece to a square occupied by a piece of the same colour. If a piece moves to a square occupied by an opponent's piece the latter is captured and removed from the chessboard as part of the same move. A piece is said to attack an opponent's piece if the piece could make a capture on that square according to Articles 3.2 to 3.8.

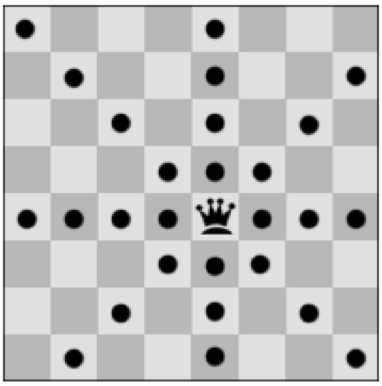
3.2 The bishop may move to any square along a diagonal on which it stands.



3.3 The rook may move to any square along the file or the rank on which it stands.

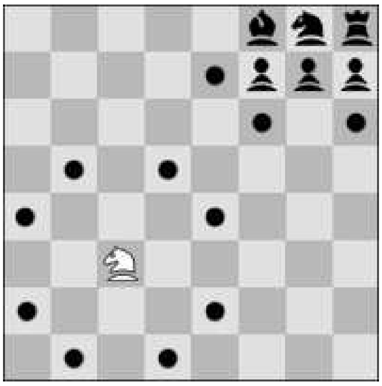


3.4 The queen may move to any square along the file, the rank or a diagonal on which it stands.



3.5 When making these moves the bishop, rook or queen may not move over any intervening pieces.

3.6 The knight may move to one of the squares nearest to that on which it stands but not on the same rank, file or diagonal.

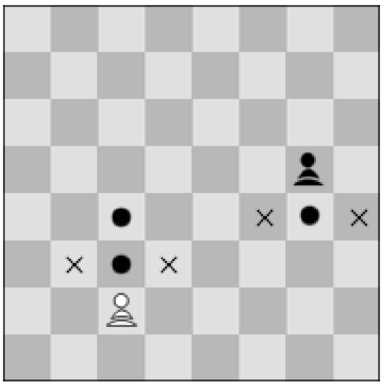


3.7

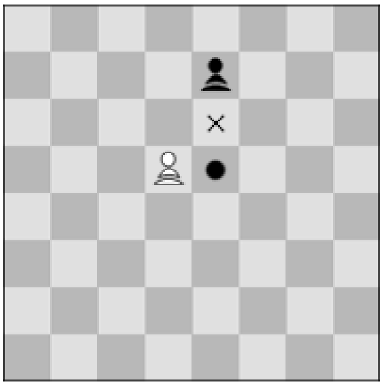
a. The pawn may move forward to the unoccupied square immediately in front of it on the same file, or

b. on its first move the pawn may move as in (a); alternatively it may advance two squares along the same file provided both squares are unoccupied, or

c. the pawn may move to a square occupied by an opponent's piece, which is diagonally in front of it on an adjacent file, capturing that piece.



d. A pawn attacking a square crossed by an opponent's pawn which has advanced two squares in one move from its original square may capture this opponent's pawn as though the latter had been moved only one square. This capture may only be made on the move following this advance and is called an 'en passant' capture.

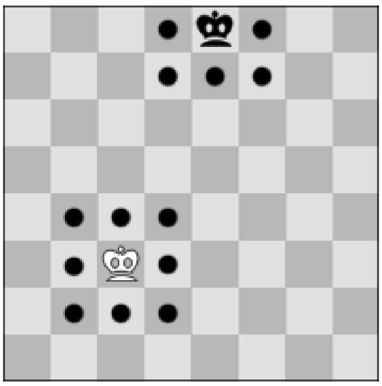


e. When a pawn reaches the rank furthest from its starting position it must be exchanged as part of the same move for a queen, rook, bishop or knight of the same colour. The player's choice is not restricted to pieces that have been captured previously. This exchange of a pawn for another piece is called 'promotion' and the effect of the new piece is immediate.

3.8

a. There are two different ways of moving the king, by:

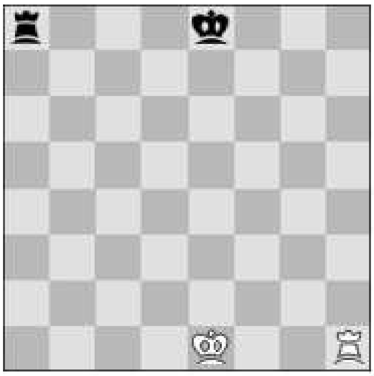
   i. moving to any adjoining square not attacked by one or more of the opponent's pieces.



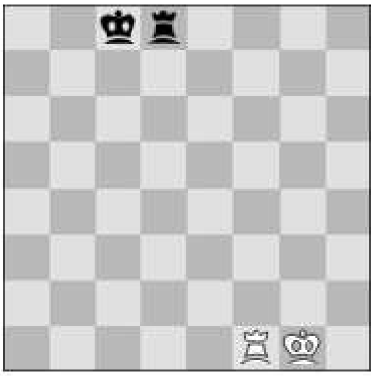
   The opponent's pieces are considered to attack a square, even if such pieces cannot themselves move.

   or

   ii. 'castling'. This is a move of the king and either rook of the same colour on the same rank, counting as a single move of the king and executed as follows: the king is transferred from its original square two squares towards the rook, then that rook is transferred to the square the king has just crossed.

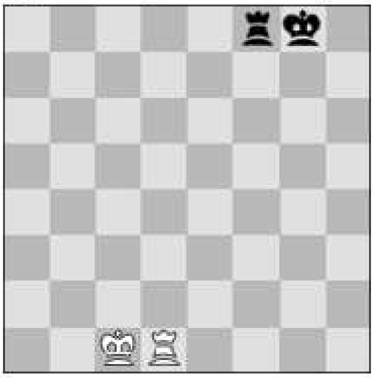


Before white kingside castling
Before black queenside castling



After white kingside castling
After black queenside castling

Before white queenside castling
Before black kingside castling



After white queenside castling
After black kingside castling

(1) Castling is illegal:
   a. if the king has already moved, or
   b. with a rook that has already moved

(2) Castling is prevented temporarily:
   a. if the square on which the king stands, or the square which it must cross, or the square which it is to occupy, is attacked by one or more of the opponent's pieces.
   b. if there is any piece between the king and the rook with which castling is to be effected.

b. The king is said to be 'in check', if it is attacked by one or more of the opponent's pieces, even if
such pieces cannot themselves move.
Declaring a check is not obligatory.

3.9 No piece can be moved that will expose its own king to check or leave its own king in check.

D.4 The act of moving the pieces

Irrelevant

D.5 The completion of the game

5.1

a. The game is won by the player who has checkmated his opponent's king. This immediately ends the game, provided that the move producing the checkmate position was a legal move.

b. The game is won by the player whose opponent declares he resigns. This immediately ends the game.

5.2

a. The game is drawn when the player to move has no legal move and his king is not in check. The game is said to end in 'stalemate'. This immediately ends the game, provided that the move producing the stalemate position was legal.

b. The game is drawn when a position has arisen in which neither player can checkmate the opponent's king with any series of legal moves. The game is said to end in a 'dead position'. This immediately ends the game, provided that the move producing the position was legal.

c. The game is drawn upon agreement between the two players during the game. This immediately ends the game. (See Article 9.1)

d. The game may be drawn if any identical position is about to appear or has appeared on the chessboard at least three times. (See Article 9.2)

e. The game may be drawn if each player has made the last 50 consecutive moves without the movement of any pawn and without the capture of any piece. (See Article 9.3)

D.6 The chess clock

Irrelevant

D.7 Irregularities

Irrelevant

D.8 The recording of the moves

Irrelevant

D.9 The drawn game

9.1 *Irrelevant*

9.2 The game is drawn, upon a correct claim by the player having the move, when the same position, for at least the third time (not necessarily by sequential repetition of moves)

a. is about to appear, if he first writes his move on his scoresheet and declares to the arbiter his intention to make this move, or

b. has just appeared, and the player claiming the draw has the move.

Positions as in (a) and (b) are considered the same, if the same player has the move, pieces of the same kind and colour occupy the same squares, and the possible moves of all the pieces of both players are the same.

Positions are not the same if a pawn that could have been captured en passant can no longer be captured or if the right to castle has been changed temporarily or permanently.

9.3 The game is drawn, upon a correct claim by the player having the move, if

a. he writes on his scoresheet, and declares to the arbiter his intention to make a move which shall result in the last 50 moves having been made by each player without the movement of any pawn and without the capture of any piece, or

b. the last 50 consecutive moves have been made by each player without the movement of any pawn and without the capture of any piece.

9.4 *Irrelevant*

9.5 *Irrelevant*

9.6 The game is drawn when a position is reached from which a checkmate cannot occur by any possible series of legal moves, even with the most unskilled play. This immediately ends the game.

D.10 Quickplay Finish

Irrelevant

D.11 Scoring

11.1 Unless announced otherwise in advance, a player who wins his game, or wins by forfeit, scores one point (1), a player who loses his game, or forfeits scores no points (0) and a player who draws his game scores a half point (1/2).

D.12 The conduct of the players

Irrelevant

D.13 The role of the arbiter

Irrelevant

D.14 FIDE

Irrelevant

E Test results from gzipping KBBK

filename	size1	size2	diff	ratio1	ratio2
Average	294058	255645	38413.3	0.224348	0.195041
KBBK_01234.dat.gz	327420	266378	61042	0.249802	0.20323
KBBK_01243.dat.gz	368598	310895	57703	0.281218	0.237194
KBBK_01324.dat.gz	319284	251267	68017	0.243594	0.191702
KBBK_01342.dat.gz	293529	251977	41552	0.223945	0.192243
KBBK_01423.dat.gz	362849	305286	57563	0.276832	0.232915
KBBK_01432.dat.gz	288201	247030	41171	0.21988	0.188469
KBBK_02134.dat.gz	325149	265039	60110	0.248069	0.202209
KBBK_02143.dat.gz	365796	309700	56096	0.27908	0.236282
KBBK_02314.dat.gz	316458	250471	65987	0.241438	0.191094
KBBK_02341.dat.gz	292579	250736	41843	0.22322	0.191296
KBBK_02413.dat.gz	360831	303809	57022	0.275292	0.231788
KBBK_02431.dat.gz	286806	245242	41564	0.218816	0.187105
KBBK_03124.dat.gz	286212	223860	62352	0.218362	0.170792
KBBK_03142.dat.gz	261097	220396	40701	0.199201	0.168149
KBBK_03214.dat.gz	286597	224120	62477	0.218656	0.17099
KBBK_03241.dat.gz	263234	220749	42485	0.200832	0.168418
KBBK_03412.dat.gz	260647	217358	43289	0.198858	0.165831
KBBK_03421.dat.gz	261490	217348	44142	0.199501	0.165823
KBBK_04123.dat.gz	353349	295245	58104	0.269584	0.225254
KBBK_04132.dat.gz	278851	233194	45657	0.212746	0.177913
KBBK_04213.dat.gz	350934	295035	55899	0.267741	0.225094
KBBK_04231.dat.gz	277277	231797	45480	0.211546	0.176847
KBBK_04312.dat.gz	253524	211767	41757	0.193423	0.161565
KBBK_04321.dat.gz	254867	211518	43349	0.194448	0.161375

filename	size1	size2	diff	ratio1	ratio2
KBBK_10234.dat.gz	327646	299220	28426	0.249974	0.228287
KBBK_10243.dat.gz	369642	343405	26237	0.282014	0.261997
KBBK_10324.dat.gz	366996	305495	61501	0.279996	0.233074
KBBK_10342.dat.gz	353093	311963	41130	0.269389	0.238009
KBBK_10423.dat.gz	362842	337268	25574	0.276826	0.257315
KBBK_10432.dat.gz	343774	303787	39987	0.262279	0.231771
KBBK_12034.dat.gz	337331	303431	33900	0.257363	0.231499
KBBK_12043.dat.gz	380367	349536	30831	0.290197	0.266675
KBBK_12304.dat.gz	286811	278415	8396	0.218819	0.212414
KBBK_12340.dat.gz	281842	272532	9310	0.215028	0.207925
KBBK_12403.dat.gz	377157	344697	32460	0.287748	0.262983
KBBK_12430.dat.gz	276927	268230	8697	0.211279	0.204643
KBBK_13024.dat.gz	360428	300528	59900	0.274985	0.229285
KBBK_13042.dat.gz	350684	309079	41605	0.267551	0.235809
KBBK_13204.dat.gz	280157	268950	11207	0.213743	0.205193
KBBK_13240.dat.gz	279004	266756	12248	0.212863	0.203519
KBBK_13402.dat.gz	341600	301080	40520	0.26062	0.229706
KBBK_13420.dat.gz	271610	254754	16856	0.207222	0.194362
KBBK_14023.dat.gz	354345	330344	24001	0.270344	0.252032
KBBK_14032.dat.gz	339311	300055	39256	0.258874	0.228924
KBBK_14203.dat.gz	369309	339756	29553	0.28176	0.259213
KBBK_14230.dat.gz	273986	264057	9929	0.209035	0.20146
KBBK_14302.dat.gz	336704	296538	40166	0.256885	0.226241
KBBK_14320.dat.gz	263788	248576	15212	0.201254	0.189648

filename	size1	size2	diff	ratio1	ratio2
KBBK_20134.dat.gz	329439	296403	33036	0.251342	0.226138
KBBK_20143.dat.gz	371457	341524	29933	0.283399	0.260562
KBBK_20314.dat.gz	362201	299708	62493	0.276337	0.228659
KBBK_20341.dat.gz	350959	305511	45448	0.26776	0.233086
KBBK_20413.dat.gz	365882	335099	30783	0.279146	0.25566
KBBK_20431.dat.gz	340969	296710	44259	0.260139	0.226372
KBBK_21034.dat.gz	341010	299202	41808	0.26017	0.228273
KBBK_21043.dat.gz	384161	347045	37116	0.293092	0.264774
KBBK_21304.dat.gz	280492	268853	11639	0.213998	0.205119
KBBK_21340.dat.gz	273600	260561	13039	0.20874	0.198792
KBBK_21403.dat.gz	380888	342102	38786	0.290594	0.261003
KBBK_21430.dat.gz	269042	257188	11854	0.205263	0.196219
KBBK_23014.dat.gz	357147	295497	61650	0.272482	0.225446
KBBK_23041.dat.gz	349036	303019	46017	0.266293	0.231185
KBBK_23104.dat.gz	272114	260524	11590	0.207607	0.198764
KBBK_23140.dat.gz	269394	255749	13645	0.205531	0.195121
KBBK_23401.dat.gz	340781	296260	44521	0.259995	0.226028
KBBK_23410.dat.gz	264485	245297	19188	0.201786	0.187147
KBBK_24013.dat.gz	356722	328172	28550	0.272157	0.250375
KBBK_24031.dat.gz	336680	293326	43354	0.256866	0.22379
KBBK_24103.dat.gz	370863	337583	33280	0.282946	0.257555
KBBK_24130.dat.gz	263920	253463	10457	0.201355	0.193377
KBBK_24301.dat.gz	335979	291288	44691	0.256332	0.222235
KBBK_24310.dat.gz	256789	239068	17721	0.195914	0.182394

filename	size1	size2	diff	ratio1	ratio2
KBBK_30124.dat.gz	255042	203842	51200	0.194582	0.155519
KBBK_30142.dat.gz	234627	203310	31317	0.179006	0.155113
KBBK_30214.dat.gz	253760	201744	52016	0.193604	0.153918
KBBK_30241.dat.gz	234375	200582	33793	0.178814	0.153032
KBBK_30412.dat.gz	235473	201767	33706	0.179652	0.153936
KBBK_30421.dat.gz	234312	199270	35042	0.178766	0.152031
KBBK_31024.dat.gz	256553	205864	50689	0.195734	0.157062
KBBK_31042.dat.gz	237115	207078	30037	0.180904	0.157988
KBBK_31204.dat.gz	218405	200718	17687	0.16663	0.153136
KBBK_31240.dat.gz	222084	202075	20009	0.169437	0.154171
KBBK_31402.dat.gz	231398	201863	29535	0.176543	0.154009
KBBK_31420.dat.gz	218336	190633	27703	0.166577	0.145441
KBBK_32014.dat.gz	257885	204431	53454	0.196751	0.155968
KBBK_32041.dat.gz	238030	204335	33695	0.181602	0.155895
KBBK_32104.dat.gz	219826	201891	17935	0.167714	0.154031
KBBK_32140.dat.gz	223227	202605	20622	0.170309	0.154575
KBBK_32401.dat.gz	233732	200248	33484	0.178323	0.152777
KBBK_32410.dat.gz	220859	192634	28225	0.168502	0.146968
KBBK_34012.dat.gz	228913	196814	32099	0.174647	0.150157
KBBK_34021.dat.gz	228235	194356	33879	0.174129	0.148282
KBBK_34102.dat.gz	228238	198631	29607	0.174132	0.151543
KBBK_34120.dat.gz	213949	185938	28011	0.16323	0.141859
KBBK_34201.dat.gz	230306	197077	33229	0.17571	0.150358
KBBK_34210.dat.gz	216496	187615	28881	0.165173	0.143139

filename	size1	size2	diff	ratio1	ratio2
KBBK_40123.dat.gz	345263	291093	54170	0.263415	0.222086
KBBK_40132.dat.gz	267413	228515	38898	0.20402	0.174343
KBBK_40213.dat.gz	342596	290732	51864	0.26138	0.221811
KBBK_40231.dat.gz	265733	226739	38994	0.202738	0.172988
KBBK_40312.dat.gz	248314	208857	39457	0.189449	0.159345
KBBK_40321.dat.gz	249806	208753	41053	0.190587	0.159266
KBBK_41023.dat.gz	345755	296769	48986	0.26379	0.226417
KBBK_41032.dat.gz	318248	238641	79607	0.242804	0.182069
KBBK_41203.dat.gz	355438	304120	51318	0.271178	0.232025
KBBK_41230.dat.gz	259309	223387	35922	0.197837	0.170431
KBBK_41302.dat.gz	313028	234029	78999	0.238821	0.17855
KBBK_41320.dat.gz	252761	205874	46887	0.192841	0.157069
KBBK_42013.dat.gz	347783	296445	51338	0.265337	0.22617
KBBK_42031.dat.gz	316923	235213	81710	0.241793	0.179453
KBBK_42103.dat.gz	356530	303665	52865	0.272011	0.231678
KBBK_42130.dat.gz	250491	219611	30880	0.191109	0.16755
KBBK_42301.dat.gz	313601	232005	81596	0.239259	0.177006
KBBK_42310.dat.gz	246947	203513	43434	0.188406	0.155268
KBBK_43012.dat.gz	225224	194160	31064	0.171832	0.148132
KBBK_43021.dat.gz	224794	191914	32880	0.171504	0.146419
KBBK_43102.dat.gz	221232	193507	27725	0.168787	0.147634
KBBK_43120.dat.gz	208731	181353	27378	0.159249	0.138361
KBBK_43201.dat.gz	223507	192077	31430	0.170522	0.146543
KBBK_43210.dat.gz	211409	183250	28159	0.161292	0.139809

F Statistics on unreachable positions

Endgame	Illegal positions	Probably reachable	Statically decided unreachable	Undoing the 1'st move failed	Undoing the 2'nd move failed	Undoing the 3'rd move failed
KK - wtm	0	462	0	0	0	0
KPK - wtm	5024	81660	0	4	0	0
KPK - btm	2676	83622	0	390	0	0
KNK - wtm	3286	26280	0	2	0	0
KNK - btm	924	28644	0	0	0	0
KBK - wtm	4832	24734	0	2	0	0
KBK - btm	924	28644	0	0	0	0
KRK - wtm	7071	22497	0	0	0	0
KRK - btm	810	28644	0	114	0	0
KQK - wtm	11076	18492	0	0	0	0
KQK - btm	924	28644	0	0	0	0
ALL 3 MEN	37547	371861	0	512	0	0
KPPK - wtm	230497	1806477	0	188	6	0
KPPK - btm	124796	1894108	999	17265	0	0
KNPK - wtm	973307	4574084	0	620	21	0
KNPK - btm	423300	5089913	13687	21132	0	0
KNNK - wtm	196088	735177	0	123	4	0
KNNK - btm	57750	867898	5744	0	0	0
KBPK - wtm	1231417	4315842	0	598	175	0
KBPK - btm	423300	5073040	14163	37527	1	1
KBNK - wtm	500671	1391423	0	235	23	0
KBNK - btm	145068	1731416	7999	7869	0	0
KBBK - wtm	280600	650656	0	112	24	0
KBBK - btm	57750	863438	10198	6	0	0
KRPK - wtm	1604982	3942722	0	228	100	0
KRPK - btm	417918	5083371	8634	38109	0	0
KRNK - wtm	620387	1271804	0	114	47	0
KRNK - btm	138114	1729518	8703	16017	0	0
KRBK - wtm	688837	1203298	0	114	103	0
KRBK - btm	138114	1719429	5198	29611	0	0
KRRK - wtm	394592	536800	0	0	0	0
KRRK - btm	50740	855582	12920	12150	0	0
KQPK - wtm	2274335	3273468	0	200	29	0
KQPK - btm	423300	5068893	14222	41617	0	0
KQNK - wtm	837329	1054906	0	100	17	0
KQNK - btm	145068	1707566	16690	23028	0	0
KQBK - wtm	906580	985621	0	100	51	0
KQBK - btm	145068	1682500	12926	51858	0	0
KQKR - wtm	742017	1145588	0	4747	0	0
KQKR - btm	507756	1384595	0	1	0	0
KQQK - wtm	569238	362078	0	0	76	0
KQQK - btm	57750	788058	16563	69021	0	0
KPKP - wtm	421087	3721987	0	17950	0	0
KNKP - wtm	843766	4682313	0	21953	0	0
KNKP - btm	566528	4981390	0	114	0	0
KNKN - wtm	289150	1603202	0	0	0	0
KBKP - wtm	1102352	4424949	0	20706	25	0
KBKP - btm	566528	4963495	0	18005	4	0
KBKN - wtm	378116	1514236	0	0	0	0
KBKN - btm	289150	1603196	0	6	0	0
KBKB - wtm	378116	1514230	0	6	0	0
KRKP - wtm	1493392	4035840	0	18800	0	0
KRKP - btm	561374	4980905	0	5748	5	0
KRKN - wtm	507756	1384596	0	0	0	0
KRKN - btm	282644	1603202	0	6506	0	0
KRKB - wtm	507756	1384592	0	4	0	0
KRKB - btm	371942	1514236	0	6174	0	0
KRKR - wtm	502225	1384596	0	5531	0	0

Endgame	Illegal positions	Probably reachable	Statically decided unreachable	Undoing the 1'st move failed	Undoing the 2'nd move failed	Undoing the 3'rd move failed
KQKP - wtm	2177064	3355323	0	15645	0	0
KQKP - btm	566528	4981473	0	31	0	0
KQKN - wtm	746764	1145588	0	0	0	0
KQKN - btm	289150	1603201	0	1	0	0
KQKB - wtm	746764	1145588	0	0	0	0
KQKB - btm	378116	1514236	0	0	0	0
KQRK - wtm	1004504	887820	0	0	28	0
KQRK - btm	138114	1664154	20258	69826	0	0
KQKQ - wtm	746764	1145588	0	0	0	0
ALL 4 MEN	31162319	125529205	168904	579696	739	1

Endgame	Illegal positions	Probably reachable	Statically decided unreachable	Undoing the 1'st move failed	Undoing the 2'nd move failed	Undoing the 3'rd move failed	Undoing the 4'th move failed
KPPPK - wtm	5174872	26057088	0	4325	291	0	0
KPPPK - btm	2847860	27968939	44400	375374	0	3	0
KNPPK - wtm	30750149	99607125	0	20004	1474	0	0
KNPPK - btm	15636432	113160337	656868	925109	1	5	0
KNNPK - wtm	48763606	125966570	0	30194	2638	0	0
KNNPK - btm	21021048	151425381	1745072	571502	2	3	0
KNNNK - wtm	5762541	13482180	0	3754	293	0	0
KNNNK - btm	1775928	17145781	327054	5	0	0	0
KBPPK - wtm	36081881	94272521	0	19045	5305	0	0
KBPPK - btm	15636432	112778263	667967	1295949	42	98	1
KBNPK - wtm	116022638	238977762	0	57116	16532	0	0
KBNPK - btm	47590128	301244480	2918427	3320865	41	107	0
KBNNK - wtm	21118012	38478287	0	10608	2181	0	0
KBNNK - btm	7190568	51219992	782180	416346	1	1	0
KBBPK - wtm	62640937	112082244	0	26882	12945	0	0
KBBPK - btm	21021048	149721717	2485537	1534494	65	145	2
KBBNK - wtm	23394805	36201063	0	9934	3286	0	0
KBBNK - btm	7190568	50990495	1016341	411680	2	2	0
KBBBK - wtm	8033963	11210403	0	3080	1322	0	0
KBBBK - btm	1775928	16907038	564977	823	1	1	0
KRPPK - wtm	43714296	86648680	0	10570	5206	0	0
KRPPK - btm	15512070	112833256	436916	1596504	0	6	0
KRNPK - wtm	135730217	219291613	0	35058	17160	0	0
KRNPK - btm	47267208	300475976	2752436	4578427	1	0	0
KRNNK - wtm	24265271	35333352	0	6973	3492	0	0
KRNNK - btm	6981948	50979785	824138	823216	1	0	0
KRBPK - wtm	147063123	207943131	0	33916	33872	0	6
KRBPK - btm	47267208	298481404	2181986	7143138	57	255	0
KRBNK - wtm	51943017	69134809	0	16598	16098	0	6
KRBNK - btm	15583008	102280792	1152872	2093848	0	8	0
KRBBK - wtm	27986709	31608572	0	6402	7399	0	6
KRBBK - btm	6981948	50255127	871957	1500046	0	10	0
KRRPK - wtm	81390891	93359459	0	6402	6256	0	0
KRRPK - btm	20695485	147924437	2628514	3514572	0	0	0
KRRNK - wtm	29420487	30182528	0	3201	2872	0	0
KRRNK - btm	6769968	50237422	1248306	1353386	2	4	0
KRRBK - wtm	30900967	28698788	0	3201	6132	0	0
KRRBK - btm	6769968	49769704	1119806	1949610	0	0	0
KRRRK - wtm	10883387	8365375	0	0	6	0	0
KRRRK - btm	1563948	16364492	715599	604728	1	0	0
KQPPK - wtm	57918137	72449288	0	9324	2003	0	0
KQPPK - btm	15636432	112268056	682024	1792238	0	2	0
KQNPK - wtm	171407844	183627745	0	30844	7614	0	1
KQNPK - btm	47590128	296958096	4405924	6119896	0	4	0
KQNNK - wtm	30041389	29559862	0	6128	1709	0	0
KQNNK - btm	7190568	49941727	1259496	1217297	0	0	0

Endgame	Illegal positions	Probably reachable	Statically decided unreachable	Undoing the 1'st move failed	Undoing the 2'nd move failed	Undoing the 3'rd move failed	Undoing the 4'th move failed
KQBPK - wtm	182937164	172087596	0	29660	19622	0	6
KQBPK - btm	47590128	292360023	3778488	11344841	112	456	0
KQBNK - wtm	62889489	58196957	0	14538	9538	0	6
KQBNK - btm	16009968	100005758	1804478	3290317	0	7	0
KQBBK - wtm	33810975	25787847	0	5536	4724	0	6
KQBBK - btm	7190568	48432212	1287034	2699263	0	11	0
KQRPK - wtm	199107278	155940821	0	11320	14629	0	0
KQRPK - btm	47267208	288316553	4884501	14605734	50	2	0
KQRNK - wtm	68914666	52178594	0	5660	11608	0	0
KQRNK - btm	15583008	98655805	2359154	4512559	0	2	0
KQRBK - wtm	72518955	48570095	0	5660	15818	0	0
KQRBK - btm	15583008	96876564	2147313	6503629	0	14	0
KQRRK - wtm	38733907	20873174	0	0	2007	0	0
KQRRK - btm	6769968	46910489	1948789	3979842	0	0	0
KQQPK - wtm	110447888	64293145	0	4918	17057	0	0
KQQPK - btm	21021048	137106944	3728283	12906659	72	2	0
KQQNK - wtm	38851786	20747485	0	2459	7358	0	0
KQQNK - btm	7190568	45964886	2046292	4407342	0	0	0
KQQBK - wtm	40359173	19236850	0	2459	10604	0	2
KQQBK - btm	7190568	44809002	1943160	5666345	0	13	0
KQQRK - wtm	42457866	17140942	0	0	10280	0	0
KQQRK - btm	6981948	43882421	2420524	6324183	0	12	0
KQQQK - wtm	14623321	4617769	0	0	7678	0	0
KQQQK - btm	1775928	13407844	1127192	2937792	0	12	0
KPPKP - wtm	16419068	80962907	0	401980	109	0	0
KPPKP - btm	14057670	82882032	43726	800633	2	1	0
KPPKN - wtm	21978492	108394949	0	5309	2	0	0
KPPKN - btm	25011260	104357994	55127	954371	0	0	0
KPPKB - wtm	21978492	107968365	0	431720	175	0	0
KPPKB - btm	30351628	99070222	52240	904501	161	0	0
KPPKR - wtm	21864552	108373220	0	140750	230	0	0
KPPKR - btm	38719428	90785331	47671	826312	10	0	0
KPPKQ - wtm	21978492	108397253	0	3006	1	0	0
KPPKQ - btm	53542248	76103147	39874	693483	0	0	0
KNPKP - wtm	60645510	204909754	0	750111	161	0	0
KNPKP - btm	39768030	224559197	617095	1361214	0	0	0
KNPKN - wtm	80590548	274471424	0	12075	1	0	0
KNPKN - btm	72818088	280327948	755509	1172503	0	0	0
KNPKB - wtm	80590548	273397908	0	1085185	407	0	0
KNPKB - btm	87427246	265819512	718140	1109145	5	0	0
KNPKR - wtm	80301116	274405284	0	367206	442	0	0
KNPKR - btm	109747222	243656911	655247	1014660	8	0	0
KNPKQ - wtm	80590548	274475262	0	8227	11	0	0
KNPKQ - btm	149863392	203810362	551407	848887	0	0	0
KNNKP - wtm	45250816	128903644	0	608422	126	0	0
KNNKP - btm	25317888	148461742	970982	12396	0	0	0
KNNKN - wtm	15490848	44118235	0	5	0	0	0
KNNKN - btm	11513028	47778863	317187	10	0	0	0
KNNKB - wtm	15490848	44117443	0	797	0	0	0
KNNKB - btm	14026699	45280221	302162	6	0	0	0
KNNKR - wtm	15308448	44118078	0	182562	0	0	0
KNNKR - btm	17718056	41614962	276063	7	0	0	0
KNNKQ - wtm	15490848	44118039	0	201	0	0	0
KNNKQ - btm	24734207	34641851	233027	3	0	0	0

Endgame	Illegal positions	Probably reachable	Statically decided unreachable	Undoing the 1'st move failed	Undoing the 2'nd move failed	Undoing the 3'rd move failed	Undoing the 4'th move failed
KBPKP - wtm	71386167	194206915	0	710464	1990	0	0
KBPKP - btm	39768030	223621710	648339	2267251	59	147	0
KBPKN - wtm	95187089	259875938	0	11020	1	0	0
KBPKN - btm	72818088	279320880	768134	2166758	52	136	0
KBPKB - wtm	95187089	258862151	0	1024530	278	0	0
KBPKB - btm	87427246	264870665	725732	2050208	68	129	0
KBPKR - wtm	94912129	259816370	0	344652	897	0	0
KBPKR - btm	109747222	242789036	666702	1870904	59	125	0
KBPKQ - wtm	95187089	259879140	0	7806	13	0	0
KBPKQ - btm	149863392	203083826	557927	1568761	40	102	0
KBNKP - wtm	111812772	242400334	0	857202	3740	0	0
KBNKP - btm	53992128	297117179	1526262	2438404	74	1	0
KBNKN - wtm	37325467	83785055	0	6	0	0	0
KBNKN - btm	24918408	95326110	441320	424690	0	0	0
KBNKB - wtm	37325467	83783499	0	1555	7	0	0
KBNKB - btm	29945750	90342683	419498	402596	1	0	0
KBNKR - wtm	36977855	83784897	0	347763	13	0	0
KBNKR - btm	37328464	83029436	382672	369955	1	0	0
KBNKQ - wtm	37325467	83784738	0	320	3	0	0
KBNKQ - btm	51360766	69117887	321470	310404	1	0	0
KBBKP - wtm	59149402	115070168	0	540822	2616	0	0
KBBKP - btm	25317888	146727370	1661497	1056061	191	1	0
KBBKN - wtm	20271615	39337472	0	1	0	0	0
KBBKN - btm	11513028	47549485	544957	1618	0	0	0
KBBKB - wtm	20271615	39336736	0	737	0	0	0
KBBKB - btm	14026699	45067043	513786	1559	1	0	0
KBBKR - wtm	20107499	39337423	0	164166	0	0	0
KBBKR - btm	17718056	41415794	473835	1402	1	0	0
KBBKQ - wtm	20271615	39337353	0	120	0	0	0
KBBKQ - btm	24734207	34478718	395270	892	1	0	0
KRPKP - wtm	86436361	179220473	0	648338	364	0	0
KRPKP - btm	39531854	224243360	354486	2175653	183	0	0
KRPKN - wtm	116499704	238563928	0	10415	1	0	0
KRPKN - btm	72515872	279963054	476223	2118899	0	0	0
KRPKB - wtm	116499704	237640270	0	933568	506	0	0
KRPKB - btm	87139386	265478334	451915	2003975	438	0	0
KRPKR - wtm	116251926	238505581	0	316152	389	0	0
KRPKR - btm	109488610	243345676	409706	1830052	4	0	0
KRPKQ - wtm	116499704	238566953	0	7373	18	0	0
KRPKQ - btm	149640032	203558009	342999	1533003	5	0	0
KRNKP - wtm	132202056	222089757	0	779362	2873	0	0
KRNKP - btm	53679744	296674428	1674264	3045491	118	3	0
KRNKN - wtm	44143776	76966745	0	7	0	0	0
KRNKN - btm	24528048	95245591	480047	856842	0	0	0
KRNKB - wtm	44143776	76965402	0	1349	1	0	0
KRNKB - btm	29574322	90264071	456208	815917	10	0	0
KRNKR - wtm	43830280	76966458	0	313790	0	0	0
KRNKR - btm	36993676	82964503	414717	737632	0	0	0
KRNKQ - wtm	44143776	76966403	0	349	0	0	0
KRNKQ - btm	51072030	69069416	347605	621477	0	0	0
KRBKP - wtm	140953328	213118853	0	997583	4284	0	0
KRBKP - btm	55874568	293068262	880083	5251007	122	6	0
KRBKN - wtm	48052470	73058053	0	5	0	0	0
KRBKN - btm	24528048	94741251	286534	1554695	0	0	0
KRBKB - wtm	48052470	73056766	0	1292	0	0	0
KRBKB - btm	29574322	89792427	271801	1471968	10	0	0
KRBKR - wtm	47753767	73057917	0	298834	10	0	0
KRBKR - btm	36993676	82529302	246457	1341092	1	0	0
KRBKQ - wtm	48052470	73057776	0	282	0	0	0
KRBKQ - btm	51072030	68711880	205796	1120821	1	0	0

Endgame	Illegal positions	Probably reachable	Statically decided unreachable	Undoing the 1'st move failed	Undoing the 2'nd move failed	Undoing the 3'rd move failed	Undoing the 4'th move failed
KRRKP - wtm	78793540	95524795	0	444659	14	0	0
KRRKP - btm	25006117	145459059	2070812	2226732	285	3	0
KRRKN - wtm	26790098	32818988	0	2	0	0	0
KRRKN - btm	11119528	47188334	685313	615913	0	0	0
KRRKB - wtm	26790098	32818418	0	572	0	0	0
KRRKB - btm	13652294	44720292	652589	583903	10	0	0
KRRKR - wtm	26658596	32818856	0	131636	0	0	0
KRRKR - btm	17380553	41118276	585196	525063	0	0	0
KRRKQ - wtm	26790098	32818841	0	149	0	0	0
KRRKQ - btm	24443141	34237234	492588	436125	0	0	0
KQPKP - wtm	117206845	148561179	0	537417	95	0	0
KQPKP - btm	39768030	223547004	603110	2387392	0	0	0
KQPKN - wtm	155772635	199292938	0	8475	0	0	0
KQPKN - btm	72818088	279203726	784673	2267561	0	0	0
KQPKB - wtm	155772635	198517510	0	783488	415	0	0
KQPKB - btm	87427246	264756047	744162	2146591	2	0	0
KQPKR - wtm	155559321	199245204	0	269091	432	0	0
KQPKR - btm	109747222	242689111	676393	1961322	0	0	0
KQPKQ - wtm	155772635	199295327	0	6078	8	0	0
KQPKQ - btm	149863392	203002400	566459	1641797	0	0	0
KQNKP - wtm	168098394	186318129	0	656413	1112	0	0
KQNKP - btm	53992128	293368479	3199945	4513496	0	0	0
KQNKN - wtm	56878397	64232128	0	3	0	0	0
KQNKN - btm	24918408	94037342	920693	1234085	0	0	0
KQNKB - wtm	56878397	64231246	0	884	1	0	0
KQNKB - btm	29945750	89116592	875058	1173128	0	0	0
KQNKR - wtm	56607873	64232017	0	270638	0	0	0
KQNKR - btm	37328464	81917998	796808	1067258	0	0	0
KQNKQ - wtm	56878397	64231878	0	253	0	0	0
KQNKQ - btm	51360766	68183813	668566	897383	0	0	0
KQBKP - wtm	177612546	176630711	0	827628	3163	0	0
KQBKP - btm	56183808	287587664	2155716	9146705	148	7	0
KQBKN - wtm	60851038	60259489	0	1	0	0	0
KQBKN - btm	24918408	92774336	704779	2713005	0	0	0
KQBKB - wtm	60851038	60258672	0	818	0	0	0
KQBKB - btm	29945750	87934782	666838	2563158	0	0	0
KQBKR - wtm	60595743	60259415	0	255370	0	0	0
KQBKR - btm	37328464	80826270	608569	2347224	1	0	0
KQBKQ - wtm	60851038	60259299	0	191	0	0	0
KQBKQ - btm	51360766	67284015	508785	1956961	1	0	0
KQRKP - wtm	194562607	159765453	0	744770	1218	0	0
KQRKP - btm	55874568	283968472	3346663	11884111	234	0	0
KQRKN - wtm	66481507	54629019	0	2	0	0	0
KQRKN - btm	24528048	91883330	1089350	3609800	0	0	0
KQRKB - wtm	66481507	54628309	0	712	0	0	0
KQRKB - btm	29574322	87077862	1033679	3424655	10	0	0
KQRKR - wtm	66254784	54628920	0	226824	0	0	0
KQRKR - btm	36993676	80087936	932549	3096366	1	0	0
KQRKQ - wtm	66481507	54628801	0	220	0	0	0
KQRKQ - btm	51072030	66672572	778330	2587595	1	0	0
KQQKP - wtm	108470966	65981793	0	307924	2325	0	0
KQQKP - btm	25317888	135232854	2749969	11462297	0	0	0
KQQKN - wtm	37154155	22454933	0	0	0	0	0
KQQKN - btm	11513028	43622836	895510	3577714	0	0	0
KQQKB - wtm	37154155	22454739	0	194	0	0	0
KQQKB - btm	14026699	41346310	848771	3387308	0	0	0
KQQKR - wtm	37057596	22454903	0	96589	0	0	0
KQQKR - btm	17718056	38040801	769258	3080972	1	0	0
KQQKQ - wtm	37154155	22454857	0	76	0	0	0
KQQKQ - btm	24734207	31661545	642438	2570897	1	0	0
ALL 5 MEN	11350319255	25679713947	115999273	282836459	318234	1846	42

G List of hand coded enumerations

0: Standard enumeration:

8	56	57	58	59	60	61	62	63	8
7	48	49	50	51	52	53	54	55	7
6	40	41	42	43	44	45	46	47	6
5	32	33	34	35	36	37	38	39	5
4	24	25	26	27	28	29	30	31	4
3	16	17	18	19	20	21	22	23	3
2	8	9	10	11	12	13	14	15	2
1	0	1	2	3	4	5	6	7	1
	a	b	c	d	e	f	g	h	

1: Pawn enumeration:

8	56	57	58	59	60	61	62	63	8
7	48	49	50	51	52	53	54	55	7
6	38	36	39	37	45	47	44	46	6
5	34	32	35	33	41	43	40	42	5
4	22	20	23	21	29	31	28	30	4
3	18	16	19	17	25	27	24	26	3
2	6	4	7	5	13	15	12	14	2
1	2	0	3	1	9	11	8	10	1
	a	b	c	d	e	f	g	h	

2: Bishop enumeration:

8	63	16	60	23	55	28	48	31	8
7	9	61	17	56	24	49	29	41	7
6	62	10	57	18	50	25	42	30	6
5	4	58	11	51	19	43	26	36	5
4	59	5	52	12	44	20	37	27	4
3	1	53	6	45	13	38	21	33	3
2	54	2	46	7	39	14	34	22	2
1	0	47	3	40	8	35	15	32	1
	a	b	c	d	e	f	g	h	

3: Hilbert curve:

8	21	22	25	26	37	38	41	42	8
7	20	23	24	27	36	39	40	43	7
6	19	18	29	28	35	34	45	44	6
5	16	17	30	31	32	33	46	47	5
4	15	12	11	10	53	52	51	48	4
3	14	13	8	9	54	55	50	49	3
2	1	2	7	6	57	56	61	62	2
1	0	3	4	5	58	59	60	63	1
	a	b	c	d	e	f	g	h	

4: Z curve:

8	21	23	29	31	53	55	61	63	8
7	20	22	28	30	52	54	60	62	7
6	17	19	25	27	49	51	57	59	6
5	16	18	24	26	48	50	56	58	5
4	5	7	13	15	37	39	45	47	4
3	4	6	12	14	36	38	44	46	3
2	1	3	9	11	33	35	41	43	2
1	0	2	8	10	32	34	40	42	1
	a	b	c	d	e	f	g	h	

5: Bishop enumeration 2:

8	22	61	9	39	7	41	29	54	8
7	46	23	62	8	40	30	55	14	7
6	4	45	24	63	31	56	13	36	6
5	32	5	44	25	57	12	37	0	5
4	6	43	26	50	18	58	11	38	4
3	42	27	51	17	49	19	59	10	3
2	28	52	16	34	2	48	20	60	2
1	53	15	35	1	33	3	47	21	1
	a	b	c	d	e	f	g	h	

6: Pawn enumeration 2. Found by GA:

8	55	56	62	53	51	48	49	57	8
7	54	59	58	52	50	60	63	61	7
6	14	12	15	13	5	7	4	6	6
5	10	8	11	9	1	3	0	2	5
4	24	25	28	29	45	44	40	41	4
3	26	27	30	31	47	46	42	43	3
2	19	17	23	21	37	35	34	33	2
1	18	16	22	20	36	39	38	32	1
	a	b	c	d	e	f	g	h	

7: Whirl enumeration:

8	12	28	44	60	11	10	9	8	8
7	13	29	45	61	27	26	25	24	7
6	14	30	46	62	43	42	41	40	6
5	15	31	47	63	59	58	57	56	5
4	48	49	50	51	55	39	23	7	4
3	32	33	34	35	54	38	22	6	3
2	16	17	18	19	53	37	21	5	2
1	0	1	2	3	52	36	20	4	1
	a	b	c	d	e	f	g	h	

8: Modified Z curve:

8	16	18	24	26	58	56	50	48	8
7	17	19	25	27	59	57	51	49	7
6	20	22	28	30	62	60	54	52	6
5	21	23	29	31	63	61	55	53	5
4	5	7	13	15	47	45	39	37	4
3	4	6	12	14	46	44	38	36	3
2	1	3	9	11	43	41	35	33	2
1	0	2	8	10	42	40	34	32	1
	a	b	c	d	e	f	g	h	

9: King "standard enumeration":

8	28	29	30	31	60	61	62	63	8
7	24	25	26	27	56	57	58	59	7
6	20	21	22	23	52	53	54	55	6
5	16	17	18	19	48	49	50	51	5
4	12	13	14	15	44	45	46	47	4
3	8	9	10	11	40	41	42	43	3
2	4	5	6	7	36	37	38	39	2
1	0	1	2	3	32	33	34	35	1
	a	b	c	d	e	f	g	h	

H Glossary

In section 3.1 some endgame specific notation is introduced.

Block: A subsequence of a table that can be represented by node in the OBDD — ie. a subsequence of the form $t[n \cdot 2^k \dots (n+1) \cdot 2^k[$ for integer valued k and n. Introduced in section 7.2.

Broken position: The entries in an endgame table that no legal position maps to are called broken positions. See section 4.2.

Complete: A BDD is called complete if all its variables b_i's are tested on each path from the root to a leaf node. The OBDDs used in this thesis has this property.

Don't care: In section 7 the broken positions are referred to as don't cares. This designation is adapted to describe the problem in a more general context.

DTM: Abbreviates Distance To Mate. See section 1.2.2.

Engine: A chess engine refers to the core part of a chess program — the function that returns the move to be played.

File: File is the name of the columns of the board (a-h).

Illegal position: See broken position.

K[x]K[y] endgame: A position in which one side has the pieces x and the other side has the pieces y.

LRU: Abbreviates Least Recently Used. A LRU caching algorithm discards the least recently used items first.

Men: A man is a pawn, knight, bishop, rook, queen or king. n-men endgame refers to endgames with a total of n men left.

Merge: See unify.

OBDD: Abbreviates Ordered Binary Decision Diagrams. The OBDDs used in this thesis tests every b_i on each path from the root to a leaf node.

Opponent: Refers to the player which is not side-to-move.

Ply: By chess players n moves refer to n moves played by each side. However for a chess programmer, almost everything is counted in half moves (eg. each recursive call of the search is called with remaining depth reduced by one half move). Therefore the word **ply** has been invented to denote a half move.

Rank: Rank is the name of the rows (1-8) of the board. White player starts with all pawns on rank 2.

Tablebases: Another word for endgame tables.

Top-down (algorithm): Refers to the algorithm introduced in section 7 for mapping don't cares.

Unify: Two blocks in an endgame table is said to be unified if they are made identical by mapping a minimal number of don't cares.

WDL: Abbreviates Win/Draw/Loss. See section 1.2.2.

References

[1] *Deep Blue vs Kasparov* http://www.research.ibm.com/deepblue/

[2] *Specification of PGN and FEN formats*,
http://www.tim-mann.org/Standard

[3] Wikipedia: *Chess*,
Described at http://en2.wikipedia.org/wiki/Chess

[4] Wikipedia: *Chess rules*,
Described at http://en2.wikipedia.org/wiki/Rules_of_chess

[5] *FIDE rules* http://www.fide.com/official/handbook.asp?level=EE101

[6] Dietmar Lippold: *Legality of Positions of Simple Chess Endgames* (1996)

[7] ChessliB: http://www.chesslib.no/

[8] ECO openings list: http://www.persocom.com.br/bcx/votodasab.htm

[9] Nalimov tablebase server: http://www.lokasoft.nl/uk/tbweb.htm

[10] GnuChess evaluation function:
http://alumni.imsa.edu/ stendahl/comp/txt/gnuchess.txt

[11] *Creating Chess-Playing Artificial Neural Networks with Distributed Evolutionary Algorithms*. See http://neural-chess.netfirms.com/HTML/project.html

[12] NeuroChess: http://satirist.org/learn-game/systems/neurochess.html

[13] David Eppstein: *Strategy and board game programming*,
Lecture notes which can be found at
http://www.ics.uci.edu/∼eppstein/180a/w99.html

[14] Robert Hyatt: *Rotated bitmaps, a new twist on an old idea*
http://www.cis.uab.edu/hyatt/bitmaps.html

[15] Andreas Junghanns, Jonathan Schaeffer:
Search Versus Knowledge in Game-Playing Programs Revisited
http://www.cs.ualberta.ca/∼jonathan/Grad/Papers/svsk.ps

[16] Made by Tim Mann: *XBoard and WinBoard*,
Description and download at http://www.tim-mann.org/xboard.html

[17] Claude E. Shannon: *Programming a computer for playing chess*
Philosophical Magazine 41:256-275

[18] Aske Plaat: *Thesis on game trees*,
http://www.cs.vu.nl/∼aske/Papers/thesis.ps.gz

[19] Robert Lake, Jonathan Schaeffer, Paul Lu:
Solving Large Retrograde Analysis Problems Using a Network of Workstations
in Advances of Computer Chess 7, Maastricht Netherlands, 1994, 135-162.

[20] Ernst A. Heinz: *Scalable Search in Computer Chess*,
Most of the book can be found at http://supertech.lcs.mit.edu/~heinz/dt/

[21] E. V. Nalimov, G.M C. Haworth and E. A. Heinz:
Space-efficient indexing of chess endgame tables,
http://supertech.lcs.mit.edu/~heinz/ps/NHH_ICGA.ps.gz

[22] Lewis Stiller: *Multilinear Algebra and Chess Endgames*
http://free.madster.com/data/free.madster.com/1249/1stiller3.pdf

[23] Johan Melin: EgmProbe 2.0
http://www3.tripnet.se/~owemelin/johan/KnightDreamer.html

[24] Henrik Reif Andersen: *An Introduction to Binary Decision Diagrams*
http://www.itu.dk/people/hra/bdd97.ps

[25] Martin Sauerhoff and Ingo Wegener:
On the complexity of minimizing the OBDD size for incompletely specified functions

[26] Kouichi Hirata, Shinichi Shimozono and Ayumi Shinohara:
On the Hardness of Approximating the Minimum Consisten OBDD Problem

[27] Shih-Chieh Chang, Malgorzata Marek-Sadowska and TingTing Hwang:
Technology Mapping for TLU FPGA's Based on Decomposition of Binary Decision Diagrams in IEEE Trans. Computer-Aided Design, vol. 15, no. 10, october 1996

[28] Youpyo Hong, Peter A. Beerel, Jerry R. Burch and Kenneth L. McMillan:
Sibling-Substitution-Based BDD Minimization Using Don't Cares
in IEEE Trans. Computer-Aided Design, vol. 19, no. 1, january 2000

[29] Youpyo Hong, Peter A. Beerel, Luciano Lavagno and Ellen M. Sentovich:
Don't care-based BDD Minimization for Embedded Software
in Proceedings of the 35th annual conference on Design automation - Volume 00

[30] Richard Rudell: *Dynamic Variable Ordering for Ordered Binary Decision Diagrams*
in Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design, p. 42-47

[31] Ingo Wegener and Beate Bollig: *Improving the variable ordering of OBDDs is NP-complete*
in IEEE Transactions on Computers, Vol. 45, Issue 9 (September 1996), p 993-1002

[32] Pierluigi Crescenzi, Viggo Kann: *A compendium of NP optimization problems*
http://www.nada.kth.se/~viggo/problemlist/compendium.html

[33] *Computational Complexity of Games and Puzzles*
http://www.ics.uci.edu/~eppstein/cgt/hard.html#chess

[34] Bernhard Balkenhol: *An Approach to Data Compression in Coding Chess Positions*
http://www.mathematik.uni-bielefeld.de/ahlswede/pub/balkenhol/icca94.ps.gz

[35] Michael Burrows, David Wheeler:
A Block-Sorting Lossless Data Compression Algorithm

[36] Hopcroft, J. and Karp, R: *An Algorithm for Maximum Matching in Bipartite Graphs*
in SIAM J. Comput., 225-231, 1975.

[37] Bellare, M., Goldreich, O., and Sudan, M. (1998):
Free bits, PCPs and non-approximability - towards tight results
In SIAM J. Comput., Vol 27, no. 3, pp. 804-915, June 1998

[38] Uriel Feige, Joe Kilian: *Zero Knowledge and the Chromatic Number*
in IEEE Conference on Computational Complexity, 1996

[39] Dempster, A. P., Laird. N. M. and Rubin, D. B.:
Maximum Likelihood from Incomplete Data via the EM algorithm
in Journal of the Royal Statistical Society, Series B, 39(1), 1-38, 1977.

[40] J. E. Hopcroft and R. M. Karp: *A* $n^{5/2}$ *algorithm for maximum matching in bipartite graphs* in SIAM Journal on Computing, 2:225–231, 1973