Formal Verification of Chess Endgame Databases

Joe Hurd[⋆]

Computing Laboratory
Oxford University
joe.hurd@comlab.ox.ac.uk

Abstract. Chess endgame databases store the number of moves required to force checkmate for all winning positions: with such a database it is possible to play perfect chess. This paper describes a method to construct endgame databases that are formally verified to logically follow from the laws of chess. The method employs a theorem prover to model the laws of chess and ensure that the construction is correct, and also a BDD engine to compactly represent and calculate with large sets of chess positions. An implementation using the HOL4 theorem prover and the BuDDY BDD engine is able to solve all four piece pawnless endgames.

1 Introduction

The game of chess with the modern rules came into existence in Italy towards the end of the 15th century [7]. The half millennium since then has witnessed many attempts to analyze the game, and in the last half century computers have naturally been used to extend the range of human analysis. One such approach uses computers to enumerate all possible positions of a certain type in an endgame database, working backwards from checkmate positions to determine the number of moves required to achieve checkmate from any starting position.

A survey paper by Heinz [6] cites Ströhlein's Ph.D. thesis from 1970 as the earliest publication on the algorithmic construction of endgame databases, and today endgame databases exist for all positions with five or fewer pieces on the board. Nalimov has started construction of the six piece endgames, but it is estimated that the finished database will require at least 1 terabyte of storage.

As an aside, it is still unclear whether or not access to endgame databases improves the strength of chess playing programs. However, they have found other uses by problemists in aiding the creation of endgame studies, and also by experts intepreting the computer analysis and writing instructional books for human players [10].

The attitude towards correctness of endgame databases is summed up by the following quotation in a paper comparing index schemes [9]:

> The question of data integrity always arises with results which are not self-evidently correct. Nalimov runs a separate self-consistency phase on each

[⋆] Supported by a Junior Research Fellowship at Magdalen College, Oxford.

> [endgame database] after it is generated. Both his [endgame databases] and those of Wirth yield exactly the same number of mutual zugzwangs [. . .] for all 2-to-5 man endgames and no errors have yet been discovered.

Applying computer theorem provers to chess endgame databases has two potential benefits:

- verifying the correctness of an endgame databases by proving that it faithfully corresponds to the rules of chess; and
- reducing the storage requirements by employing a more efficient representation than explicitly enumerating all possible positions.

For analyzing chess endgames this paper advocates the use of a higher order logic theorem prover integrated with a BDD engine. Higher order logic is very expressive, and it is possible to encode the rules of chess in a natural way, as an instance of a general class of two player games. On the other hand, BDDs can compactly represent sets of positions that have been encoded as boolean vectors, and the BDD engine can perform efficient calculation on these sets. The theorem prover ensures that the results of the BDD engine are faithfully lifted to the natural model of chess, and that all the reasoning is valid.

This methodology has been used to solve all four piece pawnless chess endgames, the product of which is a set of 'high assurance' endgame databases that provably correspond to a natural definition of chess. For example, given a chess position p in which it is Black to move, the theorem prover can fully automatically derive a theorem of the form

$$\vdash_{\mathsf{HOL+BDD}} \ \mathsf{win2\_by\ chess}\ 15\ p\ \wedge\ \neg(\mathsf{win2\_by\ chess}\ 14\ p)\ ,$$

which means that after any Black move from p, White can force checkmate within 15 moves but not within 14 moves. The $\vdash_{\mathsf{HOL+BDD}}$ symbol indicates that this theorem has been derived only from the inference rules of higher order logic and some BDD calculations. The only constants in this theorem are $\mathsf{win2\_by}$, which has a natural definition in the theory of two player games (see Section 2.1), and chess, which is a natural model of chess in higher order logic (see Section 2.2).

The primary contribution of this paper is a demonstration of the novel approach of verifying the correctness of an endgame database by proving its correspondence to a natural definition of chess, as opposed to testing its correspondence to another endgame database.

A secondary contribution of this paper is an investigation into the efficiency of BDDs to represent and calculate with sets of chess positions. Preliminary results in this area have already been obtained by Edelkamp, who calculated the number of BDD nodes to be 5% of the number of winning positions [3].

The structure of the paper is as follows: Section 2 presents a natural model of chess in higher order logic, which makes use of a general theory of two player games; Section 3 describes how an endgame database can be constructed in the theorem prover by rigorous proof; Section 4 presents the results; and Sections 5 and 6 conclude and look at related work.

2 Formalizing Chess in Higher Order Logic

The rules of chess are formalized in higher order logic in two phases. The first is a formalization of the theory of two player games, which is general enough to cover a large class of two player zero sum games with perfect information, including human games such as chess, checkers and go, and logic games such as Ehrenfeucht-Fraïssé pebble games.

The second phase defines the legal positions, move relations and winning positions of chess. Putting these into the two player game framework yields the crucial sets containing all positions in which White has a forced checkmate within n moves.

2.1 Two Player Games

The two players of the game are conventionally called *Player I* and *Player II.* In the general theory of two player games layed out in this section the positions have higher order logic type α, a type variable. This means that when the theory is applied to a specific game the type of positions can be instantiated to any concrete representation type.

A two player game G is modelled in higher order logic with a four tuple

$$(L, M, \overline{M}, W) \ ,$$

where L is a predicate on positions that holds if the position is legal. M is a relation between pairs of legal positions that holds if *Player I* can make a legal move from the first position to the second. Similarly, $\overline{M}$ is the move relation for *Player II.* Finally W is a predicate on legal positions that holds if the position is won for *Player I* (e.g., checkmate in chess). A game G is said to be well-formed (written $\mathsf{two\_player}\ G$) if the move relations and winning predicate are always false when given an illegal input position.

Intuitively, *Player I* wins a position if and only if it can be forcibly driven into a position satisfying W (within a finite number of moves). Given a well-formed game G, the following definitions make this intuition precise by carving out the set of legal positions that are eventually won for *Player I.* One way that *Player I* can fail to win is by reaching a non-winning position in which no moves are possible (e.g., stalemate in chess). This motivates the following two definitions:

$$\begin{aligned} \mathsf{terminal1}\ G \ &\equiv\ \{p \mid L_G(p) \ \wedge\ \forall p'.\ \neg M_G(p, p')\} \ ; \\ \mathsf{terminal2}\ G \ &\equiv\ \{p \mid L_G(p) \ \wedge\ \forall p'.\ \neg \overline{M}_G(p, p')\} \ . \end{aligned}$$

A position with *Player II* to move is won for *Player I* within zero moves if the predicate W is true of it:

$$\mathsf{win2\_by}\ G\ 0 \equiv \{p \mid W_G(p)\} \ .$$

A position with *Player I* to move is won for *Player I* within n moves if *Player I* can make a move to reach a position that is won for *Player I* within n moves:

$$\mathsf{win1\_by}\ G\ n \equiv \{p \mid \exists p'.\ M_G(p, p') \ \wedge\ p' \in \mathsf{win2\_by}\ G\ n\} \ .$$

Finally, a position with *Player II* to move is won for *Player I* within $n+1$ moves if it is won within n moves, or (i) it is not a terminal position and (ii) every move that player *Player I* makes will reach a position that is won for *Player I* within n moves:

$$\begin{array}{l}\mathsf{win2\_by}\ G\ (n+1) \equiv \\ \quad \mathsf{win2\_by}\ G\ n\ \cup \\ \quad \{p \mid L_G(p)\ \wedge\ \forall p'.\ \overline{M}_G(p,p')\ \Rightarrow\ p' \in \mathsf{win1\_by}\ G\ n\} - \mathsf{terminal2}\ G\ .\end{array}$$

Also of interest is the set of all positions that are eventually winning for *Player I*, which is defined separately for the cases of *Player I* to move and *Player II* to move:

$$\begin{array}{rcl}\mathsf{win1}\ G & \equiv & \{p \mid \exists n.\ p \in \mathsf{win1\_by}\ G\ n\}\ ; \\ \mathsf{win2}\ G & \equiv & \{p \mid \exists n.\ p \in \mathsf{win2\_by}\ G\ n\}\ .\end{array}$$

The preceding definitions provide all the theory of two player games that is necessary to interpret theorems resulting from a query of a verified endgame database.

2.2 Chess

The authoritative version of the laws of chess is the FIDE[1] handbook [4]. Section E.I of the handbook is entitled *Laws of Chess*, and in a series of articles describes the object of the game, the movement of the pieces and how the players should conduct themselves. For example, Article 1 is entitled *The nature and objectives of the game of chess*

> **Article 1.1.** The game of chess is played between two opponents who move their pieces alternately on a square board called a 'chessboard'. [...]

which confirms that chess is an instance of the general class of two player games formalized in the previous section.

The first design choice that occurs in the formalization of chess is to decide which higher order logic type will be used to represent chess positions. The results in this paper cover only pawnless endgames in which castling is forbidden, so the only information that needs to be tracked by the position type is the side to move and the location of the pieces on the board. The key types used to represent chess positions are:

$$\begin{array}{rcl}\text{side} & \equiv & \mathsf{White} \mid \mathsf{Black}\ ; \\ \text{piece} & \equiv & \mathsf{King} \mid \mathsf{Queen} \mid \mathsf{Rook} \mid \mathsf{Bishop} \mid \mathsf{Knight}\ ; \\ \text{square} & \equiv & \mathbb{N} \times \mathbb{N}\ ; \\ \text{position} & \equiv & \text{side} \times (\text{square} \rightarrow (\text{side} \times \text{piece})\ \text{option})\ .\end{array}$$

Sides and pieces simply enumerate the possibilities. In the context of the two player game of chess, this paper will follow the convention of referring to *Player I*

[1] FIDE (Fédération Internationale des Échecs) is the World Chess Federation.

as White and *Player II* as Black. Squares are pairs of natural numbers, and a position is a pair of the side to move and a partial function from squares to pieces. For convenience and readability, a few basic functions are defined for examining positions:

$$\begin{aligned}
\mathsf{opponent}\ s &\equiv \mathsf{case}\ s\ \mathsf{of\ White} \rightarrow \mathsf{Black} \mid \mathsf{Black} \rightarrow \mathsf{White}\ ; \\
\mathsf{to\_move}\ (s, \_) &\equiv s\ ; \\
\mathsf{on\_square}\ (\_, f)\ sq &\equiv f\ sq\ ; \\
\mathsf{empty}\ p\ sq &\equiv (\mathsf{on\_square}\ p\ sq = \mathsf{NONE})\ ; \\
\mathsf{occupies}\ p\ s\ sq &\equiv \exists v.\ \mathsf{on\_square}\ p\ sq = \mathsf{SOME}\ (s, v)\ .
\end{aligned}$$

Once the type representing the game state is fixed, what remains to apply the general theory of two player games is a higher order logic encoding of the legal positions, move relations and winning positions of chess. Such an encoding is a routine formalization, and the remainder of this section demonstrates how naturally the laws of chess can be represented in higher order logic.

Article 2 of the laws of chess in the FIDE handbook describes the geometry of the chessboard:

> **Article 2.1.** The chessboard is composed of an 8×8 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.
> **Article 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'.

This is encoded into higher order logic with the following definitions:

$$\begin{aligned}
\mathsf{files} &\equiv 8\ ; \\
\mathsf{ranks} &\equiv 8\ ; \\
\mathsf{file}\ (f, r) &\equiv f\ ; \\
\mathsf{rank}\ (f, r) &\equiv r\ ; \\
\mathsf{board} &\equiv \{sq \mid \mathsf{file}\ sq < \mathsf{files} \land \mathsf{rank}\ sq < \mathsf{ranks}\}\ ; \\
\mathsf{same\_file}\ sq\ sq' &\equiv (\mathsf{file}\ sq = \mathsf{file}\ sq')\ ; \\
\mathsf{same\_rank}\ sq\ sq' &\equiv (\mathsf{rank}\ sq = \mathsf{rank}\ sq')\ ; \\
\mathsf{same\_diag1}\ sq\ sq' &\equiv (\mathsf{file}\ sq + \mathsf{rank}\ sq = \mathsf{file}\ sq' + \mathsf{rank}\ sq')\ ; \\
\mathsf{same\_diag2}\ sq\ sq' &\equiv (\mathsf{file}\ sq + \mathsf{rank}\ sq' = \mathsf{file}\ sq' + \mathsf{rank}\ sq)\ ; \\
\mathsf{diff}\ m\ n &\equiv \mathsf{if}\ m \leq n\ \mathsf{then}\ n - m\ \mathsf{else}\ m - n\ ; \\
\mathsf{file\_diff}\ sq\ sq' &\equiv \mathsf{diff}\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq')\ ; \\
\mathsf{rank\_diff}\ sq\ sq' &\equiv \mathsf{diff}\ (\mathsf{rank}\ sq)\ (\mathsf{rank}\ sq')\ .
\end{aligned}$$

Notice that the presentational aspect of white and black squares is not included in the higher order logic encoding, only the logically important aspect of the board being an 8×8 grid of squares.

Article 3 is entitled *The moves of the pieces*:

> **Article 3.2.** The bishop may move to any square along a diagonal on which it stands.
> **Article 3.3.** The rook may move to any square along the file or the rank on which it stands.

Article 3.4. The queen may move to any square along the file, the rank or a diagonal on which it stands.
Article 3.5. When making these moves the bishop, rook or queen may not move over any intervening pieces.
Article 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.
Article 3.8. There are two different ways of moving the king, by:

1. 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.
2. or 'castling'. [...]

The moves are encoded into higher order logic in three steps. In the first step the basic moves of the pieces are defined:

$$\begin{aligned}
\mathsf{bishop\_attacks}\ sq_1\ sq_2 &\equiv (\mathsf{same\_diag1}\ sq_1\ sq_2 \lor \mathsf{same\_diag2}\ sq_1\ sq_2) \land sq_1 \neq sq_2\ ;\\
\mathsf{rook\_attacks}\ sq_1\ sq_2 &\equiv (\mathsf{same\_file}\ sq_1\ sq_2 \lor \mathsf{same\_rank}\ sq_1\ sq_2) \land sq_1 \neq sq_2\ ;\\
\mathsf{queen\_attacks}\ sq_1\ sq_2 &\equiv \mathsf{rook\_attacks}\ sq_1\ sq_2 \lor \mathsf{bishop\_attacks}\ sq_1\ sq_2\ ;\\
\mathsf{knight\_attacks}\ sq_1\ sq_2 &\equiv ((\mathsf{file\_diff}\ sq_1\ sq_2 = 1) \land (\mathsf{rank\_diff}\ sq_1\ sq_2 = 2)) \lor\\
&\quad\ ((\mathsf{file\_diff}\ sq_1\ sq_2 = 2) \land (\mathsf{rank\_diff}\ sq_1\ sq_2 = 1))\ ;\\
\mathsf{king\_attacks}\ sq_1\ sq_2 &\equiv \mathsf{file\_diff}\ sq_1\ sq_2 \leq 1 \land \mathsf{rank\_diff}\ sq_1\ sq_2 \leq 1 \land sq_1 \neq sq_2\ .
\end{aligned}$$

To improve clarity, the definition of the basic moves is closer to an explanation typically found in a beginner's chess book rather than the letter of the articles. For example, the queen is explicitly defined to move like a rook or a bishop, and the definition of the knight move follows the traditional L-shape explanation rather than the article's more geometric explanation of "[nearest square] not on the same rank, file or diaganal".[2]

The second step formalizes the no-jumping requirement of Article 3.5 by defining the concept of a clear line from a square: all the squares that can be reached horizontally, vertically or diagonally without jumping over any intervening pieces:

$$\begin{aligned}
&\mathsf{between}\ n_1\ n\ n_2\ \equiv\ (n_1 < n \land n < n_2) \lor (n_2 < n \land n < n_1)\ ;\\
&\mathsf{square\_between}\ sq_1\ sq\ sq_2\ \equiv\\
&\ \ \mathsf{if\ same\_file}\ sq_1\ sq_2\ \mathsf{then\ same\_file}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{rank}\ sq_1)\ (\mathsf{rank}\ sq)\ (\mathsf{rank}\ sq_2)\\
&\ \ \mathsf{else\ if\ same\_rank}\ sq_1\ sq_2\ \mathsf{then\ same\_rank}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{file}\ sq_1)\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq_2)\\
&\ \ \mathsf{else\ if\ same\_diag1}\ sq_1\ sq_2\ \mathsf{then\ same\_diag1}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{file}\ sq_1)\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq_2)\\
&\ \ \mathsf{else\ if\ same\_diag2}\ sq_1\ sq_2\ \mathsf{then\ same\_diag2}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{file}\ sq_1)\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq_2)\\
&\ \ \mathsf{else}\ \bot\ ;\\
&\mathsf{clear\_line}\ p\ sq_1 \equiv \{sq2 \mid \forall sq.\ \mathsf{square\_between}\ sq_1\ sq\ sq_2 \Rightarrow \mathsf{empty}\ p\ sq\}
\end{aligned}$$

The definition of square_between formalizes the notion of a square lying strictly between two others in a straight line: the verbosity is a normal consequence of using algebraic formulas to capture an essentially geometric concept.

[2] A more succinct definition that illustrates the L-shape even better is

$$\mathsf{knight\_attacks}\ sq_1\ sq_2 \equiv (\{\mathsf{file\_diff}\ sq_1\ sq_2, \mathsf{rank\_diff}\ sq_1\ sq_2\} = \{1, 2\})\ ,$$

but this has the drawback of requiring a moment's thought to see that it is correct.

In the third and final step, the basic moves of the pieces and clear lines are brought together to define the set of squares attacked from a square.

$$
\begin{array}{l}
\mathsf{attacks}\ p\ sq \equiv \\
\quad \mathsf{board} \cap \mathsf{clear\_line}\ p\ sq \cap \\
\quad (\mathsf{case\ on\_square}\ p\ sq\ \mathsf{of} \\
\quad\quad \mathsf{NONE} \to \emptyset \\
\quad\quad \mid \mathsf{SOME}\ (\_, \mathsf{King}) \to \{sq' \mid \mathsf{king\_attacks}\ sq\ sq'\} \\
\quad\quad \mid \mathsf{SOME}\ (\_, \mathsf{Queen}) \to \{sq' \mid \mathsf{queen\_attacks}\ sq\ sq'\} \\
\quad\quad \mid \mathsf{SOME}\ (\_, \mathsf{Rook}) \to \{sq' \mid \mathsf{rook\_attacks}\ sq\ sq'\} \\
\quad\quad \mid \mathsf{SOME}\ (\_, \mathsf{Bishop}) \to \{sq' \mid \mathsf{bishop\_attacks}\ sq\ sq'\} \\
\quad\quad \mid \mathsf{SOME}\ (\_, \mathsf{Knight}) \to \{sq' \mid \mathsf{knight\_attacks}\ sq\ sq'\})\ .
\end{array}
$$

Having defined the moves of the pieces, it is straightforward to formalize the set of legal positions. According to the laws of chess, a position is legal if the side that has just moved is not in check:

> **Article 3.9.** 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 are constrained from moving to that square because they would then leave or place their own king in check. No piece can be moved that will expose its own king to check or leave its own king in check.

In addition to this, the type of chess positions makes it necessary to require that all of the pieces are on the board. Without this extra requirement, the formalization would capture the game of chess being played on an infinite board!

$$
\begin{array}{l}
\mathsf{in\_check}\ s\ p \equiv \\
\quad \exists sq_1, sq_2. \\
\quad\quad (\mathsf{on\_square}\ p\ sq_1 = \mathsf{SOME}\ (s, \mathsf{King})) \wedge \\
\quad\quad \mathsf{occupies}\ p\ (\mathsf{opponent}\ s)\ sq_2 \wedge sq_1 \in \mathsf{attacks}\ p\ sq_2\ ; \\
\mathsf{all\_on\_board}\ p \equiv \forall sq.\ \neg \mathsf{empty}\ p\ sq \Rightarrow sq \in \mathsf{board}\ ; \\
\mathsf{chess\_legal}\ p \equiv \mathsf{all\_on\_board}\ p \wedge \neg \mathsf{in\_check}\ (\mathsf{opponent}\ (\mathsf{to\_move}\ p))\ p\ .
\end{array}
$$

Using everything that has been defined so far, it is easy to formalize the move relations chess_move1 (for the White pieces) and chess_move2 (for the Black pieces). In a nutshell, a move is either a simple move of a piece to an empty square, or a capturing move of a piece to a square occupied by an opponent's piece. For the full details of how this is formalized, please refer to Appendix A.

Finally, all that remains is to define the set of positions that are winning for the player of the White pieces. This is covered back in Article 1, *The nature and objectives of the game of chess*:

> **Article 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. The player who achieves this goal is said to have 'checkmated' the opponent's king and to have won the game. [. . .]

This wordy article can be concisely formalized in higher order logic:

$$
\begin{array}{rcl}
\mathsf{game\_over}\ p & \equiv & \mathsf{chess\_legal}\ p \wedge \forall p'.\ \neg \mathsf{chess\_move}\ p\ p'\ ; \\
\mathsf{checkmated}\ p & \equiv & \mathsf{game\_over}\ p \wedge \mathsf{in\_check}\ (\mathsf{to\_move}\ p)\ p\ ; \\
\mathsf{chess\_win}\ p & \equiv & (\mathsf{to\_move}\ p = \mathsf{Black}) \wedge \mathsf{checkmated}\ p\ .
\end{array}
$$

Finally, the legal positions, move relations and winning positions are put together to define the two player game of chess:

$$\mathsf{chess} \equiv (\mathsf{chess\_legal}, \mathsf{chess\_move1}, \mathsf{chess\_move2}, \mathsf{chess\_win}) \ .$$

The remainder of this paper presents a method for automatically constructing endgame databases that are formally verified with respect to this theory of the laws of chess. However, it is also possible to prove theorems interactively in the theorem prover, such as the result that a player with only a King can never win. Given a ternary relation $\mathsf{has\_pieces}\ s\ l\ p$ (defined in Appendix A) that holds whenever the side s has precisely the list of pieces l on the board in the position p, it is straightforward to prove the desired theorem

$$\vdash_{\mathsf{HOL}} \forall p.\ \mathsf{all\_on\_board}\ p \wedge \mathsf{has\_pieces\ White\ [King]}\ p \Rightarrow \neg \mathsf{chess\_win}\ p$$

by manually directing the theorem prover to apply standard proof tactics.

3 Constructing Formally Verified Endgame Databases

Recall from Section 2.1 that $\mathsf{win2\_by\ chess}\ n$ is a set of legal chess positions with Black (i.e., *Player II*) to move. The set contains all positions such that however Black moves White can force a checkmate within n moves. By convention the set $\mathsf{win2\_by\ chess}\ 0$ contains all positions where White has already won (i.e., Black is checkmated). Similarly, $\mathsf{win1\_by\ chess}\ n$ is a set of legal positions with White to move. This set contains all positions where there is a White move after which the resulting position lies in the $\mathsf{win2\_by\ chess}\ n$ set: in the chess jargon a position in the $\mathsf{win1\_by\ chess}\ n$ set is called a *mate in* $n+1$.

Constructing a formally verified endgame database consists of evaluating the $\mathsf{win1\_by\ chess}\ n$ and $\mathsf{win2\_by\ chess}\ n$ sets in the theorem prover. The first problem that occurs is that these sets are extremely large: even with just four pieces on the board, the total number of winning positions can be ten of millions. Thus it is not feasible to aim to prove a theorem of the form

$$\vdash_{\mathsf{HOL}} \mathsf{win1\_by\ chess}\ n = \{p_1, \ldots, p_N\} \ ,$$

where the p_i are an explicit enumeration of the positions in the winning set. Instead, the winning sets are represented symbolically using Binary Decision Diagrams [2], which provide a compact way to represent sets of boolean vectors. A theorem of the form

$$\vdash_{\mathsf{HOL+BDD}} \phi[B_1, \ldots, B_k] \in \mathsf{win1\_by\ chess}\ n \mapsto \mathit{\Delta} \tag{1}$$

is proved, where $[B_1, \ldots, B_k]$ is a vector of boolean variables that encode a position, ϕ is a decoding function from an encoding to a position, and $\mathit{\Delta}$ is a BDD representing a set of boolean vectors. The theorem asserts that for any assignment of booleans b_i to the variables B_i, the position $\phi[b_1, \ldots, b_k]$ is a forced win for White within n moves if and only if the vector $[b_1, \ldots, b_k]$ is in the set represented by the BDD $\mathit{\Delta}$.

The following two sections will discuss the encoding of positions as boolean variables, and the proof tools required to construct theorems of the above form in the theorem prover.

3.1 Encoding Positions as Boolean Variables

The formalization of the laws of chess presented in Section 2.2 is designed to be as natural as possible, so that a human reader (familiar with higher order logic) can be easily convinced that it is a faithful translation of the laws of chess. However, it fails to satisfy two basic requirements for encoding positions as boolean vectors:

1. The position type should be easy to encode as a vector of booleans. Although there are tools in the theorem prover to support boolean encoding of (bounded) numbers and lists, the function from squares to pieces in the position type would require a custom encoder to be written and proved correct.
2. Given a list of White and Black pieces, it should be straightforward to define the set of all positions that have precisely these pieces on the board, since that is how endgame databases are structured. Unfortunately, the square based nature of the position type makes it inconvenient to reason about the pieces on the board.

For both these reasons, the boolean encoding of positions makes use of an intermediate 'posn' type defined as follows:

$$\begin{aligned} \text{placement} &\equiv (\text{side} \times \text{piece}) \times \text{square} \; ; \\ \text{posn} &\equiv \text{side} \times \text{placement list} \; . \end{aligned}$$

Versions of the legal position predicates, move relations and winning position predicate are defined on type posn, and their definitions are designed for ease of boolean encoding. In addition, a function

$$\mathsf{abstract} : \text{posn} \to \text{position}$$

is defined that lifts elements of type posn to chess positions. With respect to the abstract function, the two versions of the legal position predicates, move relations and winning position predicates are identical: a useful check for both versions.

The new posn type also satisfies the requirement that positions should be easily categorized according to the pieces on the board. Define a category to be a side to move and a list of pieces on the board:

$$\text{category} \equiv \text{side} \times (\text{side} \times \text{piece}) \text{ list} \; .$$

For example

$$(\mathsf{Black},\ [(\mathsf{White}, \mathsf{King}),\ (\mathsf{White}, \mathsf{Rook}),\ (\mathsf{Black}, \mathsf{King})])$$

is the category of positions where it is Black to move, White has a King and Rook on the board, and Black has only a King. The set of all elements of the posn type in a category (s, l) can be defined as

$$\mathsf{category}\ (s, l) \ \equiv\ \{(s', l') \mid s' = s \ \wedge\ \mathsf{map\ fst}\ l' = l\}\ ,$$

where map is the standard list map function and fst is the function that picks the first component from a product.

For each category c, all the positions p in category c are encoded to booleans in the same way. The side to move and pieces in p are fixed, so the only 'state' left to encode as booleans are the squares that the pieces are on, which is a fixed length list of pairs of bounded natural numbers. Encoding this type is a relatively straightforward matter of plumbing together the standard boolean encoders for fixed length lists, products and bounded natural numbers that are already defined in the theorem prover [11]. Given a category c, this process yields a function encode_posn c for encoding posns in category c as a vector of booleans, and an inverse function decode_posn c for decoding a vector of booleans as a posn in category c.

For positions in a category c, the decoder function ϕ in Equation (1) can now be expanded to

$$\mathsf{abstract} \circ \mathsf{decode\_posn}\ c\ .$$

3.2 Proving Endgame Database Theorems

The verified endgame database is constructed category by category by symbolically evaluating the winning sets (i.e., calculating the BDDs Δ in Equation (1) for increasing values of n). When a fixed point is found, a stability theorem is proved which is lifted to the position type using to_move and has_pieces predicates. For example, the lifted stability theorem

$$\begin{array}{l} \vdash_{\mathsf{HOL+BDD}} \\ \quad \forall p. \\ \qquad \mathsf{all\_on\_board}\ p \wedge (\mathsf{to\_move}\ p = \mathsf{Black}) \wedge \\ \qquad \mathsf{has\_pieces}\ p\ \mathsf{White}\ [\mathsf{King}, \mathsf{Rook}] \wedge \mathsf{has\_pieces}\ p\ \mathsf{Black}\ [\mathsf{King}] \Rightarrow \\ \qquad (p \in \mathsf{win2\ chess} \iff p \in \mathsf{win2\_by\ chess}\ 16) \end{array}$$

states that for positions with Black to move, White having a King and Rook and Black having only a King, if a position is won at all for White then checkmate can be forces within 16 moves. In addition, a concrete position is lifted from the final BDDs to show that this bound is the best possible:

$$\begin{array}{l} \vdash_{\mathsf{HOL+BDD}} \\ \quad (\mathsf{Black}, \\ \quad \lambda sq. \\ \qquad \mathsf{if}\ sq = (0,0)\ \mathsf{then\ SOME\ (White, King)\ else\ if}\ sq = (5,6)\ \mathsf{then\ SOME\ (White, Rook)} \\ \qquad \mathsf{else\ if}\ sq = (3,6)\ \mathsf{then\ SOME\ (Black, King)\ else\ NONE)}\ \in\ \mathsf{win2\_by\ chess}\ 16\ \wedge \\ \quad (\mathsf{Black}, \\ \quad \lambda sq. \\ \qquad \mathsf{if}\ sq = (0,0)\ \mathsf{then\ SOME\ (White, King)\ else\ if}\ sq = (5,6)\ \mathsf{then\ SOME\ (White, Rook)} \\ \qquad \mathsf{else\ if}\ sq = (3,6)\ \mathsf{then\ SOME\ (Black, King)\ else\ NONE)}\ \notin\ \mathsf{win2\_by\ chess}\ 15\ . \end{array}$$

Calculating the sequence of BDDs representing winning sets for a category is implemented using the category-specific boolean encoding of the move relations and winning position predicate. The winning position predicate is converted to a BDD, and this becomes the first BDD in the sequence. The move relation is also converted to a BDD, and applied to the current winning set to find the set of positions that for which the current winning set is reachable in one White move (this new winning set consists of all the *mate in one* positions). The BDD resulting from this calculation is added to the sequence of BDDs, and becomes the current winning set. The BDD for the move relation is now applied again, but with a universal instead of an existential quantifier, to calculate the set of positions such that *all* Black moves result in a position in the current winning set. The BDD representing this winning set is added to the sequence of BDDs, and becomes the current winning set. This sequence of BDDs representing winning sets is continued until it converges to a fixed point (i.e., the winning set with Black to move is the same as the previous winning set with Black to move).

Since pieces may get captured during play, and this changes the category of the position, it is important to construct the endgame databases for the small categories first, so that captures always reduce to a previously solved position. The base case is two bare Kings on the board, and then different pieces are added to first solve all the three piece endgames, and then the four piece endgames.

There are potential pitfalls to symbolically calculating the winning sets that do not appear in the usual method of explicitly enumerating all positions, but the theorem prover ensures that the reasoning is sound and that no positions are left out. For example, consider the category

$$(\mathsf{White},\ [(\mathsf{White},\mathsf{King}),\ (\mathsf{White},\mathsf{Queen}),\ (\mathsf{White},\mathsf{Rook}),\ (\mathsf{Black},\mathsf{King})])$$

where from any starting position White needs at most six moves to force checkmate. Indeed, during construction of the sequence of BDDs they are seen to converge after six moves. However, because this category of endgame can reduce by a capture to the category where White has a King and Rook against Black's bare King, and because in this smaller category White sometimes needs 16 moves to force checkmate, it is logically necessary to extend the sequence of BDDs to 16 moves in the original category. At that point all the side conditions are satisfied and the stability theorem can be proved:

$$\begin{array}{l} \vdash_{\mathsf{HOL+BDD}} \\ \quad [\cdots] \Rightarrow \\ \quad p \in \mathsf{win1\ chess} \iff p \in \mathsf{win1\_by\ chess}\ 16\ . \end{array}$$

The final step is to prove that the official set of winning positions found after 16 moves is equal to the set of winning positions found after six moves, and thus conclude that the same stability theorem must also hold for six moves:

$$\begin{array}{l} \vdash_{\mathsf{HOL+BDD}} \\ \quad [\cdots] \Rightarrow \\ \quad p \in \mathsf{win1\ chess} \iff p \in \mathsf{win1\_by\ chess}\ 6\ . \end{array}$$

4 Results

The construction of verified endgame databases described in the previous section is implemented in the HOL4 theorem prover,[3] using the `HolBddLib` [5] interface to the BuDDy BDD engine.[4]

One thing that can make a big difference to the performance of a BDD calculation is the ordering of the boolean variables. Recall from Section 3.1 that the 'state' to be encoded as boolean variables is a list of squares on the board. This is exactly how the state breaks down into boolean variables B:

$$\begin{array}{rcl} \text{State} & \longleftarrow & \text{Square} \cdots \text{Square} \\ \text{Square} & \longleftarrow & \text{File Rank} \\ \text{File} & \longleftarrow & B\ B\ B \\ \text{Rank} & \longleftarrow & B\ B\ B \end{array}$$

To test the effect of variable ordering on performance the construction of the King and Rook versus King and Rook endgame database is used as a benchmark.[5] If the variables are ordered exactly as above then the endgame database takes 1,514 seconds to construct, and the BDD engine creates 165,847,971 nodes. If instead the variables for the state are formed by interleaving the variables for each square, then the endgame database takes 543 seconds to construct, and the BDD engine produces 16,413,512 nodes. If additionally the variables for each square are formed by interleaving the file and rank variables, the endgame database takes 835 seconds to construct and the BDD engine produces 84,019,830 nodes. Clearly the middle option is best, and this has since been confirmed on other benchmark tests.

Another BDD optimization that proved effective was to combine the quantification and logical connective that occurs when the move relation is applied to the current winning set. On a benchmark test of constructing all four piece endgames containing only Kings, Rooks and Knights, the time required fell 19% from 3,251 seconds and 222,122,342 nodes produced to 2,640 seconds and 144,441,858 nodes produced.

The final results for all four piece pawnless endgames are shown in Table 1. The first column shows the pieces on the board: first the White pieces using the standard abbreviations of K for King, Q for Queen, R for Rook, B for Bishop and N for Knight; next an underscore; and finally the Black pieces. The other columns are separated into positions with White to move and positions with Black to move. Within each, the columns are as follows: the **max** column shows the maximum number of moves required for White to force checkmate from a winning position, or a dash if there are no positions winning for White; the **%win** column shows the percentage of legal positions that are winning for White, a dash if there are none, or 'ALL' if every legal position is winning for White;

[3] HOL4 is available for download at `http://hol.sf.net/`.

[4] BuDDy is available for download at `http://sourceforge.net/projects/buddy`.

[5] All the results were collected on a Pentium 4 3.2GHz processor with 1Gb of main memory and running the HOL4 theorem prover using Moscow ML 2.01.

the **#win** column shows the total number of positions winning for White; the **bdd** column shows the compression ratio of the number of BDD nodes required to store the winning sets divided by the total number of winning positions; the **#legal** shows the the total number of legal positions; and the final **bdd** column shows the BDD compression ratio for the legal positions.

Pieces	**White to move**						**Black to move**					
	max	**%win**	**#win**	**bdd**	**#legal**	**bdd**	**max**	**%win**	**#win**	**bdd**	**#legal**	**bdd**
K_K	—	—	0	0%	3612	1%	—	—	0	0%	3612	1%
K_KB	—	—	0	0%	223944	0%	—	—	0	0%	193284	0%
K_KBB	—	—	0	0%	13660584	0%	—	—	0	0%	10164056	0%
K_KBN	—	—	0	0%	13660584	0%	—	—	0	0%	10875504	0%
K_KN	—	—	0	0%	223944	0%	—	—	0	0%	205496	0%
K_KNN	—	—	0	0%	13660584	0%	—	—	0	0%	11499304	0%
K_KQ	—	—	0	0%	223944	0%	—	—	0	0%	144508	1%
K_KQB	—	—	0	0%	13660584	0%	—	—	0	0%	7698432	0%
K_KQN	—	—	0	0%	13660584	0%	—	—	0	0%	8245296	0%
K_KQQ	—	—	0	0%	13660584	0%	—	—	0	0%	5657120	0%
K_KQR	—	—	0	0%	13660584	0%	—	—	0	0%	6911296	0%
K_KR	—	—	0	0%	223944	0%	—	—	0	0%	175168	0%
K_KRB	—	—	0	0%	13660584	0%	—	—	0	0%	9366840	0%
K_KRN	—	—	0	0%	13660584	0%	—	—	0	0%	9905048	0%
K_KRR	—	—	0	0%	13660584	0%	—	—	0	0%	8325184	0%
KB_K	—	—	0	0%	193284	0%	—	—	0	0%	223944	0%
KB_KB	1	0%	416	0%	11832464	0%	0	0%	112	0%	11832464	0%
KB_KN	1	0%	16	0%	11832464	0%	0	0%	8	0%	12535256	0%
KB_KQ	—	—	0	0%	11832464	0%	—	—	0	0%	8952608	0%
KB_KR	—	—	0	0%	11832464	0%	—	—	0	0%	10780728	0%
KBB_K	19	49%	5007216	12%	10164056	0%	19	41%	5628080	8%	13660584	0%
KBN_K	33	100%	10822184	30%	10875504	0%	33	82%	11188168	19%	13660584	0%
KN_K	—	—	0	0%	205496	0%	—	—	0	0%	223944	0%
KN_KB	1	0%	40	0%	12535256	0%	0	0%	8	0%	11832464	0%
KN_KN	1	0%	40	0%	12535256	0%	0	0%	8	0%	12535256	0%
KN_KQ	—	—	0	0%	12535256	0%	—	—	0	0%	8952608	0%
KN_KR	1	0%	32	0%	12535256	0%	0	0%	8	0%	10780728	0%
KNN_K	1	0%	1232	0%	11499304	0%	0	0%	240	0%	13660584	0%
KQ_K	10	ALL	144508	19%	144508	1%	10	90%	200896	12%	223944	0%
KQ_KB	17	100%	8925252	19%	8952608	0%	17	77%	9097332	18%	11832464	0%
KQ_KN	21	99%	8894128	21%	8952608	0%	21	80%	10088688	21%	12535256	0%
KQ_KQ	13	42%	3737092	11%	8952608	0%	12	0%	40628	1%	8952608	0%
KQ_KR	35	99%	8863768	52%	8952608	0%	35	66%	7062680	35%	10780728	0%
KQB_K	8	ALL	7698432	11%	7698432	0%	10	91%	12379568	8%	13660584	0%
KQN_K	9	ALL	8245296	9%	8245296	0%	10	90%	12343856	7%	13660584	0%
KQQ_K	4	ALL	5657120	4%	5657120	0%	10	98%	13378232	6%	13660584	0%
KQR_K	6	ALL	6911296	4%	6911296	0%	16	99%	13519192	6%	13660584	0%
KR_K	16	ALL	175168	20%	175168	0%	16	90%	201700	18%	223944	0%
KR_KB	29	35%	3787160	11%	10780728	0%	29	3%	381888	5%	11832464	0%
KR_KN	40	48%	5210920	34%	10780728	0%	40	11%	1364800	23%	12535256	0%
KR_KQ	19	29%	3090088	5%	10780728	0%	18	0%	17136	0%	8952608	0%
KR_KR	19	29%	3139232	5%	10780728	0%	19	1%	72464	1%	10780728	0%
KRB_K	16	ALL	9366840	12%	9366840	0%	16	91%	12458920	10%	13660584	0%
KRN_K	16	ALL	9905048	11%	9905048	0%	16	91%	12406892	10%	13660584	0%
KRR_K	7	ALL	8325184	3%	8325184	0%	16	100%	13621424	6%	13660584	0%
	40	**29%**	**1.179E8**	**6%**	**4.033E8**	**0%**	**40**	**34%**	**1.355E8**	**6%**	**4.033E8**	**0%**

Table 1. Results for all four piece pawnless endgames.

Constructing the whole endgame database took 18,540 seconds (including 418 seconds spent on garbage collection), during which the HOL4 theorem prover executed 82,713,188 primitive inference steps in its logical kernel and the BDD engine produced 882,827,905 nodes.

5 Conclusions

This paper has shown how a theorem prover equipped with a BDD engine can be used to construct an endgame database that is formally verified to logically follow from the laws of chess.

The method has been implemented for all four piece pawnless positions, and the resulting endgame database can be used as a 'golden reference' for other implementors of endgame databases to check against. In addition, the verified endgame database has been used to produce a set of educational web pages showing the best line of defence in each position category.[6]

The approach used to augment standard theorem proving techniques with a tailor made BDD algorithm was found to be convenient for this application, combining the expressive power and high assurance of theorem provers with the compact representation and fast calculation of BDD engines. As seen in Section 3.2, the use of a theorem prover avoided some potential pitfalls that appear when symbolically processing sets of positions.

6 Related Work

The earliest example of applying BDDs to analyze a two player game is the attempt of Baldumus et. al. [1] to solve American Checkers by means of symbolic model checking.

Edelkamp [3] put forward the idea that BDDs are generally suitable for classifying positions in a wide range of two player games, including chess endgames. Edelkamp's encoding of chess positions also includes a bit for the side to move, but otherwise it is identical to the encoding in this paper. This paper can be seen as a continuation of Edelkamp's work, with the addition of a theorem prover to ensure the accuracy of the move encodings and winning sets.

Kristensen [8] investigated the use of BDDs to compress endgame databases, showing BDDs to be comparable to the state of the art in explicit enumeration for 3–4 man endgames, and better for some simple 5 man endgames.

Acknowledgements

This work was initiated by a conversation between the author and Tim Leonard, and the correct method of boolean encoding was hammered out during many discussions with Konrad Slind.

[6] Available at `http://www.gilith.com/chess/endgames`

References

1. Michael Baldamus, Klaus Schneider, Michael Wenz, and Roberto Ziller. Can American Checkers be solved by means of symbolic model checking? In Howard Bowman, editor, *Formal Methods Elsewhere (a Satellite Workshop of FORTE-PSTV-2000 devoted to applications of formal methods to areas other than communication protocols and software engineering)*, volume 43 of *Electronic Notes in Theoretical Computer Science*. Elsevier, May 2001.
2. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
3. Stefan Edelkamp. Symbolic exploration in two-player games: Preliminary results. In *The International Conference on AI Planning & Scheduling (AIPS), Workshop on Model Checking*, pages 40–48, Toulouse, France, 2002.
4. FIDE. *The FIDE Handbook*, chapter E.I. The Laws of Chess. FIDE, 2004. Available for download from the FIDE website.
5. Michael J. C. Gordon. Reachability programming in HOL98 using BDDs. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 179–196. Springer, August 2000.
6. E. A. Heinz. Endgame databases and efficient index schemes. *ICCA Journal*, 22(1):22–32, March 1999.
7. David Hooper and Kenneth Whyld. *The Oxford Companion to Chess.* Oxford University Press, 2nd edition, September 1992.
8. Jesper Torp Kristensen. Generation and compression of endgame tables in chess with fast random access using OBDDs. Master's thesis, University of Aarhus, Department of Computer Science, February 2005.
9. E. V. Nalimov, G. McC. Haworth, and E. A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, September 2000.
10. John Nunn. *Secrets of Rook Endings.* Gambit Publications, December 1999.
11. Konrad Slind and Joe Hurd. Applications of polytypism in theorem proving. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 103–119. Springer, September 2003.

A Formalized Chess (Omitted Definitions)

```
pushes $p\ sq \equiv$
   board $\cap$ clear_line $p\ sq \cap$
   (case on_square $p\ sq$ of
      NONE $\to \emptyset$
      | SOME (_, King) $\to \{sq' \mid$ king_attacks $sq\ sq'\}$
      | SOME (_, Queen) $\to \{sq' \mid$ queen_attacks $sq\ sq'\}$
      | SOME (_, Rook) $\to \{sq' \mid$ rook_attacks $sq\ sq'\}$
      | SOME (_, Bishop) $\to \{sq' \mid$ bishop_attacks $sq\ sq'\}$
      | SOME (_, Knight) $\to \{sq' \mid$ knight_attacks $sq\ sq'\}$) ;

sorties $p\ sq \equiv \{sq' \mid sq' \in$ pushes $p\ sq \land$ empty $p\ sq'\}$ ;

captures $p\ sq \equiv \{sq' \mid sq' \in$ attacks $p\ sq \land$ occupies $p$ (opponent (to_move $p$)) $sq'\}$ ;

simple_move $p\ p' \equiv$
   $\exists sq_1, sq_2.$
      occupies $p$ (to_move $p$) $sq_1 \land sq_2 \in$ sorties $p\ sq_1 \land$
      $\forall sq.$
         on_square $p'\ sq =$
         if $sq = sq_1$ then NONE
         else if $sq = sq_2$ then on_square $p\ sq_1$
         else on_square $p\ sq$ ;

capture_move $p\ p' \equiv$
   $\exists sq_1, sq_2.$
      occupies $p$ (to_move $p$) $sq_1 \land sq_2 \in$ captures $p\ sq_1 \land$
      $\forall sq.$
         on_square $p'\ sq =$
         if $sq = sq_1$ then NONE
         else if $sq = sq_2$ then on_square $p\ sq_1$
         else on_square $p\ sq$ ;

chess_move $p\ p' \equiv$
   chess_legal $p \land$ chess_legal $p' \land$
   (to_move $p'$ = opponent (to_move $p$)) $\land$
   simple_move $p\ p' \lor$ capture_move $p\ p'$ ;

chess_move1 $p\ p'$ = chess_move $p\ p' \land$ (to_move $p$ = White) ;

chess_move2 $p\ p'$ = chess_move $p\ p' \land$ (to_move $p$ = Black) ;

has_pieces $p\ s\ l \equiv$
$\exists f \in$ Bijection $\{n \mid n <$ length $l\}$ $\{sq \mid$ occupies $p\ s\ sq\}$.
   $\forall n.\ n <$ length $l \Rightarrow$ (on_square $p$ $(f\ n)$ = SOME $(s,$ nth $n\ l))$ .
```