Lossless compression via two-level logic minimization: a case study using Chess endgame data

Dave Gomboc Christian R. Shelton
Dept. of Computer Science and Engineering, University of California, Riverside, CA 92521 USA
dave_gomboc@acm.org cshelton@cs.ucr.edu

Abstract

The utility of processing techniques long in use within the electronic design automation community is underappreciated within the artificial intelligence community. We update and use the ESPRESSO logic minimizer in order to generate an exact, readily-queryable, and succinct representation of voluminous Chess endgame data.

1 Introduction

Chess play has been a relevant topic of academic study for over a century, decades before computing science itself existed as a discipline [Zermelo, 1913; v. Neumann, 1928; Shannon, 1950; Turing, 1953; Newell et al., 1958; Bellman, 1965; Quinlan, 1983; Campbell et al., 2002]. Today, machines are substantially stronger Chess players than top human experts, and the same can be said regarding many other similar traditional human games (e.g., Backgammon, Checkers, Go, Othello, Scrabble, Shogi, and Texas Hold 'em).

1.1 Game-theoretic error-free endgame play

A Chess endgame table (“EGT”) is a precomputed, known-correct source of information about Chess endgame positions. The first Chess EGTs were computed by Ströhlein [1970]; seven-piece EGTs were first computed by Zakharov et al. [2013] on the Lomonosov supercomputer, using tens of tebibytes. Chess engines that reach such pre-tabulated positions from within their heuristic searches can propagate back an exact score, which can be used either to improve on-line game play or to improve the accuracy and efficiency of offline reinforcement learning [Silver et al., 2018] via tablebase rescoring [killrducky, 2018].

1.2 State-of-the-art Chess endgame data compression

Syzygy endgame tables are the current standard Chess EGT format used, because these tables are more compact than any widely-available alternative, while also being acceptably efficient to query. By their design, it is required that the probing software possesses considerable infrastructure of a Chess engine: in particular, positions with legal captures may be recorded using a misleading value that achieves better compression. The querying program must actually perform a complete capture-based quiescence search and minimax to determine the correct result.

The open-source program Fathom [Falsinelli et al., 2015] assists in querying Syzygy EGTs in the absence of a Chess engine. We used a locally-modified version of this program to query existing Syzygy tables to obtain source data. Accordingly, our computations have the same limitations as the Syzygy EGTs: for example, it is always assumed that neither side may castle their king.

For each covered combination of pieces, the Syzygy format provides a win-draw-loss (“WDL”) table and a distance-to-zeroing-move (“DTZ”) table. The ability to search from the starting position until the game result of every position in the search frontier can be looked up within the WDL tables alone would be sufficient for a player to never play into a game-theoretically-suboptimal position, as has previously been achieved in Checkers [Schaeffer et al., 2007]: the DTZ tables can merely be helpful when exploiting any mistakes made by a fallible opponent. Accordingly, we will not consider the DTZ tables further herein.

1.3 Objectives

The exponential growth of tablebase size as the piece count increases ensures that the storage of even eight-piece tables on commodity hardware will not be feasible in the near future in the absence of an improved compression algorithm. We wish to sharply reduce the storage space required to represent Chess EGT WDL information, while also eliminating any requirement to perform any amount of game-tree search in order to successfully probe. We also hope that

<table>
<thead>
<tr>
<th>Piece count</th>
<th>Required space</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤ 5</td>
<td>78.1 MiB</td>
</tr>
<tr>
<td>= 6</td>
<td>67.8 GiB</td>
</tr>
<tr>
<td>= 7</td>
<td>8.5 TiB</td>
</tr>
</tbody>
</table>

Table 1: Syzygy win-draw-loss endgame table sizes
realizing substantial simplification or improvement in an al-
ready thoroughly-studied domain will encourage other sci-
entists and engineers to consider whether processing the data
related to their applications of interest in a similar manner
would be beneficial.

We provide a fresh, accessible introduction to two-level
logic minimization in Section 2, and report on our experi-
ments in Section 3.

2 Two-level logic minimization

Let us consider a partial function \( P: \{0, 1\}^n \rightarrow \{0, 1\}^m \).
An equivalent total function \( T: \{0, 1\}^n \rightarrow \{0, 1, X\}^m \) ex-
ists, where an output of \( X \) indicates that we do not care which
truth value is assigned to that output. Naturally, the straight-
forward tabular representation of \( T \) would always contain \( 2^n \)
rows. Succinctness is desirable, so the matrix representation of
the function \( M: \{0, 1, X\}^n \rightarrow \{0, 1, X\}^m \), where an in-
put of \( X \) indicates that the row is applicable regardless of the
instantiated truth value of that input, can be used to reduce the
number of rows in the tabular representation: a single row of
matrix \( M \) with \( k \) inputs set to \( X \) is equivalent to specifying
the \( 2^k \) compatible rows of the tabular representation of \( T \).

The union of the input vectors where any of the outputs is
assigned to either 0, 1, or \( X \) is considered to be part of the
ON-cover (or \( F \), for function), the OFF-cover (or \( R \), for reverse),
or the DC-cover (or \( D \), for don’t care), respectively. Each such cover is the sum of clauses; each clause (or “cube”,
by tradition, though a clause can represent a hyperrectangle)
is the product of individual inputs. Two-level logic minimiza-
tion is the task of, having been provided with some matrix
\( M \) that is consistent with \( P \), identifying a matrix \( M' \) that is
also consistent with \( P \) whose covers of interest have minimum
cardinality.

We first discuss a few important algorithms from the elec-
tronic design automation (“EDA”) literature, though we re-
fer the reader to Coudert [1994] for coverage of additional
historically-important logic minimization techniques. We
then describe the mapping from Chess endgame table data
to \( \{0, 1, X\} \)-vectors, and we finish the section by discussing
the enhancements that we have made to ESPRESSO.

2.1 MINI

The MINI logic minimizer [Hong et al., 1974] introduced the
heuristic approach of iteratively improving cover cardinality
via repeated cube expansion and reduction.

Positional cube notation

In positional cube notation (“PCN”), each specific value of
an input variable \( v \) is mapped to a tuple of bits whose length
is the cardinality of the domain. So, a multiple-valued input
variable of the domain \{ant, bee, cat, dog\} could be mapped
as follows: ant \( \rightarrow 1000 \); bee \( \rightarrow 0100 \); cat \( \rightarrow 0010 \); dog
\( \rightarrow 0001 \). Here, 1111 would be used to represent “don’t
care”.

For each binary input variable \( v \), PCN reduces to a bit
pair \( \{0v: 0 \rightarrow 10; 1 \rightarrow 01; X \rightarrow 11\} \). Cube intersection is
efficiently performed via bitwise and: when two cubes with
incompatible variable assignments are so intersected, a zero-
tuple occurs for each such conflicting variable.

Distance-one merging

In their paper, Hong et al. report that MINI performed
distance-one merging for “computational advantage”. An in-
dividual distance-one merge operation permits two product
clauses of a cover to be combined when they disagree in a
single \{0, 1, X\}-input variable, thereby reducing cover cardi-
nality, as in the following three examples:

<table>
<thead>
<tr>
<th>before</th>
<th>0x011 000</th>
<th>100x010</th>
<th>xo11 001</th>
</tr>
</thead>
<tbody>
<tr>
<td>after</td>
<td>0x11 000</td>
<td>10x010</td>
<td>xo1 001</td>
</tr>
</tbody>
</table>

MINI iterates over each such input variable once, updating
the sorted ordering of \( M' \) prior to processing each variable
to ensure that the clauses are ordered to permit all potential
merges involving that variable via a linear scan through the
product clauses.

Expansion

Distance-one merging is a particular form of cube expansion,
which is the process of enlarging a cube so that it (hopefully)
includes as many as possible of the minimum product terms,
or minterms, of \( M' \) that must be covered, while avoiding cov-
ering any product terms that must not be covered (the collec-
ction of which constitutes the blocking cover). An expanded
cube may newly encompass one or more other cubes: when
this happens, these other cubes are no longer necessary to re-
tain in order to accurately represent \( P \), and so are discarded.

Reduction

Once expansion has occurred, many cubes that partially o-
verlap may cover the same minterms. Cube reduction is the pro-
cess of shrinking a cube while ensuring that it continues to
cover all minterms not already covered by any other cube.

2.2 ESPRESSO

Brayton et al. [1984] famously introduced the unate recur-
sive paradigm in their book, and their C implementation of
ESPRESSO was open-sourced under a liberal licence, assist-
ing its wide adoption. Unfortunately, that version has re-
mained in relative statis for the past quarter-century.

Irredundancy

While expansion alone can eliminate many cubes, it does
not eliminate any cube that does not end up completely en-
compassed by a single other cube. The irredundancy pass
within ESPRESSO’s expansion-irredundancy-reduction main
loop exists to prioritize the cardinality minimization of \( M' \)
via the detection and removal of such cubes that are nonethe-
less redundant with respect to multiple other cubes in advance
of performing any reduction that could cause an available op-
portunity for cube removal to be forfeited.

Distance-one merging

Like MINI, the ESPRESSO implementation used does sup-
port the ability to apply distance-one merging across multiple
variables of the ON-cover in sequence. Though this capabil-
ity is neither alluded to anywhere in the book nor happens by
default, the espresso(1) manual page suggests its use.
2.3 Pupik

The Pupik logic minimization algorithm [Fišer et al., 2008; Fišer and Toman, 2009] hails from the same research group as the BOOM-II heuristic logic minimizer [Fišer and Kubátová, 2006]. Pupik is based on processing ternary trees [Fišer and Hlavíčka, 2001] that compactly represent Boolean functions. Pupik repeatedly performs single-variable absorption \((a + ab = a)\) and complementation \((\bar{a} + \bar{ab} = \bar{a})\) to combine adjacent cubes with identical outputs.

Unfortunately, the algorithmic and experimental performance analyses performed by Fišer et al. [2008] and Fišer and Toman [2009] consider neither the possibility of maintaining \(M'\) in sorted order nor the use of ESPRESSO’s distance-one merging capability, respectively. In fact, exploiting single-variable absorption and complementation together is exactly the same operation as a single distance-one merge operation, and performing the full procedure described in Fišer et al. [2008] is precisely equivalent to distance-one merging over \(F\).

Furthermore, asymptotic analysis is not the whole story: performing repeated accesses over a tree lacks the memory locality behaviour of comparing vectors that are juxtaposed in memory\(^1\), and assessing each individual binary bit access within a table as a distinct operation is also unrealistic. Today, commodity processors provide registers that support operating on 256 or even 512 bits at a time: it is difficult to say how many tens of thousands of binary inputs might be required before performance actually increased from using a tree structure without either access to Pupik’s source code or reimplementing it from scratch.\(^2\)

2.4 A simple position encoding scheme

Many different ways to encode Chess positions for subsequent logic minimization exist: we have chosen ours with two major criteria in mind. We retain the traditional top-level division of Chess endgame positions by their material balance. Even though another strategy could be superior, making this choice permits straightforward comparison of our experimental results with what has been prior practice for a half-century.

More importantly, the position encoding scheme has been selected to be as absolutely uninformed about Chess as possible. Not only do our input vectors contain no machine-learned features, they also fail to manually capture basic Chess notions such as whether the player to move is in check\(^3\), and which covers will be provided. Such directives are indirect indexing schemes that include all legal positions for a material balance, but as few additional illegal positions as possible [van den Herik and Herschberg, 1985; Thompson, 1986; Heinz, 1999; Nalimov et al., 2000]. It is also common for multiple indexing order permutations to be attempted for each material balance: once it is determined which variant turns out to yield the smallest file size after a subsequent layer of block compression is applied, the necessary data required to select which scheme is to be used for decompression is recorded near the beginning of the file. By instead relying upon logic minimization to combine adjacent cubes with compatible outputs, we avoid considerable tedium.

This representation also permits labelling large blocks of positions with the same output vector \(a\text{ priori}\). For example, all positions where a black pawn is on the eighth rank are illegal. We could specify that we do not care about any such positions within the \(\mathcal{W}\mathcal{Q}\mathcal{R}\mathcal{A}\) table using a single matrix row: \(\begin{array}{cccccccc} x & x & x & x & x & x & x & x \end{array}\). Thus, enhancing EGTs to also accommodate positions where castling rights have not been lost becomes straightforward:\(^4\) add a binary input for each relevant castling status, and whenever one is enabled, set to don’t care the three outputs for all positions when either the relevant king or rook that would be castled with has already moved.

2.5 Modifications of ESPRESSO

We have made several local improvements to ESPRESSO. Most of the changes we described are to ensure the correctness of and/or to simplify the implementation, though compaction itself provides a substantial performance improvement, as we shall see in Section 3.

\(^1\)In practice, a suffix array implementation exhibits an approximately 5x performance advantage over an equivalent suffix tree implementation, for this very reason.

\(^2\)We attempted to but did not succeed in establishing communication with the relevant research authors.

\(^3\)Castling support would be useful: Chess studies presume the legality of castling, unless it can be proven that an encountered position could not have been reached without having forfeited the right to castle.
We perform distance-one merging against all \{0, 1, X\}-inputs over each of \(F\), \(D\), and \(R\), but using ESPRESSO’s existing data structures. Compaction can occur in the presence of multiple-valued inputs and/or multiple binary outputs. Unlike MINI’s and ESPRESSO’s older distance-one merging capabilities, we do not cease to iterate after visiting each input variable once. Instead, we continue iterating until no further distance-one merges are available.

Function cover consistency

While a function’s covers need to be self-consistent, ESPRESSO’s checking has been stricter than is necessary. Self-consistency does require that the union of a function’s ON-cover, OFF-cover, and DC-cover must be the universe. However, overlap between its ON-cover and its DC-cover is permitted. Likewise, overlap between its OFF-cover and its DC-cover is permitted. Consequently, overlap between a function’s ON-cover and OFF-cover that are simultaneously undergoing minimization should actually be permitted, so long as the entirety of their overlap remains within the function’s DC-cover. In other words, the intersection of the OFF-cover and the DC-cover should not be part of the blocking cover when operating on the ON-cover, and likewise, the intersection of the ON-cover and the DC-cover should not be part of the blocking cover when operating on the OFF-cover.

ESPRESSO does not retain either \(F - D\) or \(R - D\) in memory: making such an improvement would permit simpler covers to be identified whenever both covers could take advantage of flexibility provided by using the same don’t cares. Currently, we compute these as part of our improved consistency checking when vetting cover information read in from a data file, as in the following operation.

Function consistency

Verifying that two matrices \(A\) and \(B\) actually represent the same partial function \(P\) is essential to ensure that operations on \(M^P\) have been correctly performed. Therefore, we added the capability to check that all of the following conditions hold for two matrices read in from their corresponding data files:

\[
\begin{align*}
D_A &= D_B \\
F_A - D_A &= F_B - D_B \\
R_A - D_A &= R_B - D_B
\end{align*}
\]

Technical debt repayment

In furtherance of our aim to make additional algorithmic and technical improvements, including but not limited to supporting multiprocessing and SIMD-enablement via the usage of in-memory compressed Boolean vectors [Lemire et al., 2018], we: have sharply reduced ESPRESSO’s use of not only global variables and structures, but also raw memory accesses; have upgraded all of ESPRESSO’s code, which was a mixture of K&R C and ANSI C89/ISO C90, to C++17; now use the CMake build system, permitting parallel compilation.

### 3 Experimentation

We conduct experiments to explore the trade-off between minimization time and minimization quality. As can be discerned from Table 2, the two-piece table has \(2^{13}\) possible positions (where, as explained above, a position not only encompasses the mapping of pieces to squares but also includes the side to move), while each three-piece table has \(2^{19}\) possible positions. Any position where a piece would be superimposed on another is considered to be invalid. The notion of canonicalization referred to in the third column of Table 2 will be described below.

The machine that we used for all timed computations has an eight-core 17-9700 CPU that nominally runs at 3.0GHz (though the frequency adapts dynamically) and has 32 GiB of RAM. Each job was given the use of a single core and 4 GiB of RAM. Up to seven jobs were permitted to run simultaneously, in an attempt to avoid any resource oversubscription that might cause undesirable timing variability.
Table 3: Two- and three-piece endgame table results, with and without canonicalization and distance-one merging

<table>
<thead>
<tr>
<th>acanonical as don’t care</th>
<th>material balance</th>
<th>$T$ clauses in $F$</th>
<th>just expansion clauses in $F$</th>
<th>full ESPRESSO clauses in $F$</th>
<th>just compaction clauses in $F$</th>
<th>compaction, then just expansion clauses in $F$</th>
<th>compaction, then full ESPRESSO clauses in $F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td></td>
<td>7224</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>492</td>
<td>0.1</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>368452</td>
<td>21319.9</td>
<td>1275</td>
<td>22524.9</td>
<td>1123</td>
<td>91.7</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>24851</td>
<td>121.8</td>
<td>196</td>
<td>4772</td>
<td>184</td>
<td>19.1</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>399112</td>
<td>30169.6</td>
<td>846</td>
<td>30660.9</td>
<td>746</td>
<td>58.4</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>27212</td>
<td>177.1</td>
<td>161</td>
<td>426.6</td>
<td>155</td>
<td>19.1</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>417228</td>
<td>119.0</td>
<td>1</td>
<td>120.3</td>
<td>1</td>
<td>124.5</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>28215</td>
<td>10.2</td>
<td>1</td>
<td>15.6</td>
<td>1</td>
<td>14.0</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>429440</td>
<td>41.6</td>
<td>1</td>
<td>42.7</td>
<td>1</td>
<td>44.0</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>29060</td>
<td>9.3</td>
<td>1</td>
<td>14.7</td>
<td>1</td>
<td>12.9</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>331352</td>
<td>8032.3</td>
<td>5407</td>
<td>175921</td>
<td>4632</td>
<td>122.6</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>85018</td>
<td>1368.4</td>
<td>2198</td>
<td>85296</td>
<td>1992</td>
<td>31.0</td>
</tr>
</tbody>
</table>

3.1 Two- and three-piece endgame tables

Our first experiment manipulates three processing conditions while processing the two- and three-piece tables:

1. Whether compaction is or is not performed;
2. Whether the full ESPRESSO algorithm is or is not executed;
3. Whether the full ESPRESSO algorithm is or is not executed.

For each equivalence class of positions defined by the available symmetries for a material balance, we can designate one in particular as the canonical representation for the particular canonical transformation made, potentially translating the rest data back in order to return the appropriate probe result.

One caveat is that we do not yet take advantage of an additional material balance that exists when White and Black have the same material, which limits performance in the and cases.

A full discussion thereof would be too astray: see Bromley et al. [1994] for details.
3.2 Two- through four-piece results

We now include four-piece endgames as we attempt to explore further the trade-off of minimization time versus the cardinality of $F$. The processing treatment that has been added to Table 4 is to apply compaction, expansion, and irredundancy, without performing the full ESPRESSO algorithm. Attempting to run ESPRESSO when using the $\mathcal{A} \mathcal{A} \mathcal{A}$ material balance results in an assertion failure being issued from within the irredundancy portion of ESPRESSO's code. The root cause is that the 16-bit field used to store cube indices while determining an irredundant cover is insufficiently capacious when processing sufficiently large data sets.

We assess the irredundancy pass to be both relatively quick and effective at further reducing the cardinality of $F$ after the expansion pass has been performed. When the full ESPRESSO algorithm also completes relatively quickly, it is in cases where it provides no substantial additional minimization over and above what expansion and irredundancy together achieve. In many other cases, running the full ESPRESSO algorithm is extremely time-consuming, and furthermore, there is every reason to expect it to remain prohibitively expensive as problem size increases.

The additional effort to continue to reduce cover cardinality undertaken by the full ESPRESSO algorithm may be particularly valuable in the electronics manufacturing context. For example, simpler circuits are associated with using either fewer lookup tables ("LUT’s), or less die space and less power. However, the extra effort of executing full ESPRESSO does not appear to be an efficient use of our limi-
ited processing power: given our intention to obtain usable compressed PCN versions of larger EGTs as economically as possible, applying compaction, expansion, and irredundancy appears to be our best trade-off.

### 3.3 Compression effectiveness

We now take each product clause in $F$ generated when using compaction, expansion, and irredundancy. Each clause can be represented in PCN in 64 bits with room to spare. We have generated a binary file per material balance containing each row in PCN. Table 5 reports the bytes consumed by the uncompressed PCN in memory (which is 8 bytes per product clause in $F$), the bytes consumed by the compressed PCN files that would be stored persistently, and the size of the (already compressed) Syzygy WDL tables that are currently stored persistently when used with Chess engines.

The final row of Table 5 shows that the WDL information for all successfully processed endings requires 948 292 bytes in the compressed PCN format, versus 1 017 456 bytes for the same data in the Syzygy WDL format. Given that there has been a half-century of endgame table technology development leading to the Syzygy format, and that numerous opportunities to improve compression results using this novel method remain, we are comfortably claiming that this method of lossless compression has promise.

### 4 Contributions

Logic minimization techniques have previously been applied widely within EDA, and also within image processing-like and stream compression contexts [Yang et al., 2006; Amarú et al., 2014]. We have provided a top-level explanation of essential two-level logic minimization algorithms, clarified some relationships between techniques described within its literature, and shown experimentally that logic minimization may be an effective technique for compressing Chess endgame tables.

### Acknowledgments

We thank the University of California, Riverside for access to computational resources. The first author is also an employee of Google LLC, however, this research has been performed with neither awareness of any insider Google LLC knowledge that might exist nor use of resources associated with that employment. Any statements and opinions expressed do not necessarily reflect the position or the policy of neither Google LLC nor the University of California; no official endorsement should be inferred.

### References


### Table 5: Endgame table sizes

<table>
<thead>
<tr>
<th>material balance</th>
<th>uncompressed PCN size (B)</th>
<th>compressed PCN size (B)</th>
<th>Syzygy WDL size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kk Kk</td>
<td>8</td>
<td>64</td>
<td>n/a (80)</td>
</tr>
<tr>
<td>KBk KBk</td>
<td>264</td>
<td>1232</td>
<td>131 720</td>
</tr>
<tr>
<td>KBq KBq</td>
<td>37448</td>
<td>13652</td>
<td>7056</td>
</tr>
<tr>
<td>KRk</td>
<td>175120</td>
<td>5776</td>
<td>2320</td>
</tr>
</tbody>
</table>
...


