

Comparing Lossless Compression Methods for Chess Endgame Data

Dave Gomboc^{a,*}, Christian R. Shelton^a, Andrew S. Miner^b and Gianfranco Ciardo^b

^aUniversity of California, Riverside

^bIowa State University

ORCID ID: Dave Gomboc <https://orcid.org/0009-0008-8051-9115>, Christian R. Shelton
<https://orcid.org/0000-0001-6698-7838>, Andrew S. Miner <https://orcid.org/0000-0002-7737-6888>,
Gianfranco Ciardo <https://orcid.org/0000-0002-4906-6145>

Abstract. Chess endgame tables encode unapproximated game-theoretic values of endgame positions. The speed at which information is retrieved from these tables and their representation size are major limiting factors in their effective use. We explore and make novel extensions to three alternatives (decision trees, decision diagrams, and logic minimization) to the currently preferred implementation (Syzygy) for representing such tables. Syzygy is most compact, but also slowest at handling queries. Two-level logic minimization works well when the full compression algorithm can be run. Decision DAGs and multiterminal binary decision diagrams are comparable and offer the best querying times, with decision diagrams providing better compression.

1 Introduction

Endgame tables (EGTs) support increased skill in play by storing precomputed exact (non-heuristic) information about game positions for convergent games, which are games where the size of the state space decreases as the game approaches its end. An EGT serves as a map from game state to relevant information. EGTs are typically calculated via retrograde analysis [15]. For the game of Checkers, the precomputation of EGTs combined with forward search allowed the determination of the game-theoretic result of its initial position [12].

We focus on Chess EGTs, the first of which were computed by Ströhlein [14]. The largest complete sets of such EGTs currently are for seven pieces [16], and require at least tens of trillions of bytes to generate and store, even when compressed. We study lossless compression methods for Chess EGTs, quantifying their compression and probing speeds.

1.1 State-of-the-art

Syzygy [5] is currently the predominant Chess EGT storage implementation: its representation is more compact than widely available alternatives, it is acceptably efficient to query, and its data files and source code are freely available for download and use.

Syzygy, and virtually all of its predecessor Chess EGT implementations, provides a separate table for each material balance: the set

of pieces remaining for each player. There is a table for positions where White has a king, pawn, and rook, and Black has a king and queen, there is another table for positions where White has a king and two rooks and Black has a king and pawn, and so on. For each material balance, there is both a win/cursed win/draw/blessed loss/loss (WCDBL) table that indicates the game-theoretic value of positions, and a distance to zeroing-move (DTZ) table.

Cursed wins and blessed losses are positions where the official over-the-board rules of Chess allow a player to successfully claim a draw once 50 moves by each side in succession without a capture or pawn push would be made in a position that would otherwise lead to a decisive result with perfect play. WCDBL tables are also referred to as win/draw/loss (WDL) tables in the literature because no piece configurations with four or less pieces in the absence of move history are cursed wins or blessed losses.

In this paper, we focus on WCDBL: the use of sufficiently large WCDBL tables alone during lookahead is sufficient to avoid reaching a game-theoretically suboptimal position. DTZ tables also store additional information that permits optimal play once an EGT position has been reached.

1.2 Problem definition

We encode the coordinate of any location on the chessboard using three bits for the file and three bits for the rank. One additional bit is used to encode the side to move. A pawn that may be captured en passant is encoded as being on the first or eight rank instead of the fourth or fifth rank where it actually is. Thus, if we assume that castling is not possible (which we do, as Syzygy does), then when probing some particular five-piece material balance, we can express the input as a bit vector of length $(3 + 3) \times 5 + 1 = 31$.

The mapping from the input to one of five outcomes is partial: not all inputs are possible. For instance, two pieces cannot occupy the same location. We also exploit symmetry to reduce further the number of stored positions. For example, when no pawns are present (and castling is not possible), all board rotations are equivalent. Flipping the player to move along with the colour of all pieces is also symmetric. We only store and query one canonical position for each set of symmetric positions. Consequently, over 90% of possible inputs are not associated with any particular output.

The goal is to produce a data structure, and suitable querying code,

* Corresponding Author. Email: dave_gomboc@acm.org.

that returns the proper game-theoretic value for any canonical encoding of a Chess position. While it is not strictly necessary to do so, we continue the historic practice of building separate data structures for each material balance. We build our alternative compressed representations from a list of each canonical Chess position of the relevant material balance, paired with its associated game-theoretic outcome.

1.3 Research Goal

We aim to explore alternatives to the Syzygy Chess EGT format. Syzygy is a complex system that by design exploits properties specific to Chess such as not encoding the correct values for positions containing legal captures, instead relying on identifying these positions at probing time and uses a capture-based minimaxing search to back-up the values from other tables.

However, it is not our goal to exhaustively assess any of the once-common Chess EGT formats such as the Nalimov [9] tables. These tables do not provide full WCDBL information, yet require seven times the space of Syzygy, so they are no longer used by the top tier of current Chess-playing programs.

Rather, this paper aims to quantify what is possible using non-Chess-specific compression techniques. While we do modify some standard compression methods, these modifications target general properties of EGTs. We explore three alternatives that, while exploiting basic symmetries of Chess, are nonetheless general-purpose compression and query techniques. They have simple querying code and straightforward interpretations. We believe these alternatives could also be helpful in other convergent games in which symmetries exist, such as Xiangqi and International Draughts.

1.4 Contributions

We consider and compare three alternative methods to Syzygy for storing a WCDBL EGT: decision directed acyclic graphs (DAGs), indexed two-level logic minimization (TLLM), and multiterminal reduced ordered binary decision diagrams (MTBDDs). All methods construct a tree or DAG whose leaves indicate either individual outcomes or a small set of cube-outcome pairs to scan linearly. Each of these methods uses different heuristics to form a compact representation of the EGT data.

We provide necessary improvements to each method for efficient use on this application. For decision trees, the modification is minor: we build a decision DAG instead of a tree. For logic minimization, we significantly rework main operations of the ESPRESSO algorithm to enable operating upon tables of this size. For decision diagrams, we extend the concretization methods for fully-reduced binary decision diagrams (BDDs) to MTBDDs. The modifications for logic minimization and decision diagrams are novel contributions to these areas.

Each method approaches the compression problem differently. We outline each method, draw connections amongst them, and compare them empirically with each other and with Syzygy on three-, four-, and five-piece EGTs.

2 Decision DAGs

We use a standard binary decision tree learning algorithm [10] to compress each EGT. The set of canonical positions of each table are used as the training examples, each viewed as an input of binary features and an output as one of five classes reflecting the game-theoretic

value of the position. When building, we employ greedily the standard information gain metric [11]. Decisions at internal nodes correspond to checking a single bit of the input.

Our method differs only slightly from typical machine learning decision tree construction. Because we seek only compression, not generalization (our training set for each EGT already includes every position of interest), we exhaustively build the tree and perform no pruning. Second, to save space, we construct a DAG rather than a tree by merging common subtrees during the construction process.

3 Two-level logic minimization

We also compare with the recent endgame compression method of Gomboc and Shelton [7], which leveraged the ESPRESSO [2] two-level logic minimizer to perform minimization on three- and four-piece tables. ESPRESSO represents functions as pairs of input and output vectors. The input vector may be represented using the alphabet $\{0, 1, *\}$, respectively meaning that the corresponding input must be 0, must be 1, or may be either 0 or 1. The output vector may be represented using the alphabet $\{f, r, d, \sim\}$, respectively meaning that the corresponding output must be 1, must be 0, may be either 0 or 1, or is unconstrained by the paired input vector.

ESPRESSO repeatedly iterates through three major operations. EXPAND attempts to replace 0s and 1s in each input vector with *s (known as “raising”) such that as many other input vector are subsumed and eliminated as possible, while ensuring that no output vector specification conflicts occur. IRREDUNDANT identifies and removes input vectors whose output constraints may be deduced from combinations of other remaining input vectors without adjusting the values of any input vectors. REDUCE attempts to replace *s in each input vector with 0s and 1s without newly permitting any output bit to take on any value that was previously precluded by some combination of mapping constraints.

3.1 Representation simplification

As discussed in Section 1.2, lookups for the vast majority of input vectors will never be performed. Nonetheless, ESPRESSO explicitly represents D , the multiset of input vectors that map any output to d , because its implementation of both IRREDUNDANT and REDUCE depend upon D ’s availability. When attempting to process five-piece EGTs using ESPRESSO, representing D causes main memory (256 GiB) exhaustion. To address this problem, we devised alternative algorithms for performing these two operations that do not require D to be provided as an input.

In our application of TLLM, we never map any output bit to $\sim$, and furthermore, either every element or no element of each individual output vector is mapped to d . Thus, in addition to not representing D , we also need not represent R , the multiset of input vectors that map any output to r , separately from F , the multiset of input vectors that map any output to f , which results in further memory savings.

In this paper, we refer to just the input vector associated with an output as a cube. We initialize the EGTs prior to minimization by specifying the output value for each minimum product term (a.k.a. unit cube) about which we care.

As arguments to Algorithms 1 and 2, we provide two cube sets, which we denote as B (for baseline) and E (for expanded). B represents an earlier representation of F prior to expansion: it can, but does not have to be, the original definition of F prior to any minimization. E represents some expansion of B . In both, we use indices

(see Section 3.3), priority queues, and caching to make them more efficient than a direct implementation of the pseudo-code would be.

3.1.1 Efficient D -less irredundancy

Algorithm 1 computes an irredundant cover E' from B and E . We add cubes from E , starting with all cubes that uniquely cover some cube in B , then greedily select additional cubes which cover the most additional not-yet-covered cubes of B .

Algorithm 1 IRREDUNDANT

Require: B : Baseline cubes (modifies copy)

Require: E : Expanded cubes that covers B (modifies copy)

Ensure: E' : Irredundant subset of E that covers B

```

1  $E' = \{e \mid \exists b \in B, \forall e' \neq e, \neg \text{contains}(e', b)\}$ 
2  $B = B - \{b \in B \mid \exists e \in E', \text{contains}(e', b)\}$ 
3  $E = E - E'$ 
4 while  $E \neq \{\}$ 
5    $e = \arg \max_{e \in E} \sum_{b \in B \mid \text{contains}(e, b)} 1$ 
6    $E' = E' \cup \{e\}$ 
7    $B = B - \{b \in B \mid \text{contains}(e, b)\}$ 
8    $E = E - \{e\} - \{e' \in E \mid \nexists b \in B, \text{contains}(e', b)\}$ 
9 return  $E'$ 
```

3.1.2 Efficient D -less reduction

Algorithm 2 computes a reduced cover E' from B and E . We repeatedly select expanded cubes uniformly at random without replacement, shrinking them as much as possible while ensuring that B remains covered.

Algorithm 2 REDUCE

Require: B : Baseline cubes (modifies copy)

Require: E : Expanded cubes that cover B (modifies copy)

Ensure: E' : Cover of B , each cube is a subcube of some cube in E

```

1  $E' = \{\}$ 
2 foreach  $e \in E$  (random order)
3    $E = E - \{e\}$ 
4    $B' = \{b \in B \mid \text{contains}(e, b)\}$ 
5    $B'' = \{b \in B' \mid \nexists e' \in E, \text{contains}(e', b)\}$ 
6   if  $B'' \neq \{\}$ 
7      $E' = E' \cup \text{supercube}(B'')$ 
8    $B = B - B''$ 
9 return  $E'$ 
```

3.2 Accelerating expansion

Application of Algorithms 1 and 2 permit us to no longer immediately exhaust main memory, yet we can still only completely process several of the simplest five-piece EGTs. The performance of ESPRESSO, both before and after the above changes, is dominated by the 99% time spent in EXPAND.

Though the time complexity of a single EXPAND operation is referred to as quadratic in the size of its input within ESPRESSO's

source code, we found that it actually scales less well. As is, EXPAND could not be applied even a single time to the vast majority of five-piece EGTs, even when given weeks of running time. We were able to address this problem by introducing two innovations.

3.2.1 Distance- n merging (for $n > 1$)

Both ESPRESSO and its predecessor MINI [8] support distance-one merging, which is the merging of two cubes that disagree on only a single variable. Such merges are always safe to perform.

We introduce a generalization, distance- n merging, that can be used to merge cubes that disagree on multiple variables simultaneously. Unlike the distance-one case, the blocking cover must now be consulted. Given v input variables, Algorithm 3 sweeps over all $\binom{v}{n}$ combinations of input variables and attempts to merge any set of cubes that differ only in these variables. Distance- n merging is not intended to be a replacement for the more general EXPAND operation of MINI and ESPRESSO. Rather, it is particularly effective and intended for use when $n \ll v$ and there may be many small cubes present within a cover that is to undergo expansion. By sorting the cubes based on all other variables, we can quickly find groups of cubes to be merged.

Algorithm 3 DISTANCE-N-MERGE

Require: M : Cover to which merging should be applied

Require: B : Baseline cubes

Require: n : number of variables to merge

Ensure: M' : Cover equivalent to M , where $|M'| \leq |M|$

```

1  $M' = M$ 
2 foreach subset,  $V$ , of variables of size  $n$ 
3    $\mathcal{G} = \text{groupby}(M, V)$ 
4   (groups where all variables except  $V$  are the same)
5   foreach  $G \in \mathcal{G}$ :
6     if output,  $o$ , same for cubes in  $G$ 
7        $c = \text{supercube}(G)$ 
8        $c[V] = *$ 
9        $D = \{b \in B \mid \text{contains}(c, b) \wedge \text{output}(b) \neq o\}$ 
10      if  $D = \{\}$  (check not necessary if  $n = 1$ )
11         $M' = (M' - G) \cup \{c\}$ 
12 return  $M'$ 
```

3.2.2 Random expansion

ESPRESSO's [2] implementation of the EXPAND algorithm [8] tracks which cubes cover which other cubes in order to attempt to grow cubes in ways that encompass as many other cubes as possible. While highly effective at small problem sizes, this tracking is also the root cause of its performance bottleneck when applied to the much larger minimization problems we consider, which render the algorithm infeasible for our use.

Instead, we propose the RANDOM-EXPAND algorithm (Alg. 4). Each time RANDOM-EXPAND attempts to expand any cube, it randomly selects an input variable ordering, then attempts to *raise* each input variable (replace the specific 0 or 1 for this variable with $*$) one after the other in that chosen order. Every raise that does not result in a conflict is accepted. Consequently, unlike with EXPAND as used by ESPRESSO, RANDOM-EXPAND discards no cubes whatsoever. Instead, we exclusively rely on IRREDUNDANT for cube elimination.

Algorithm 4 RANDOM-EXPAND

Require: B : Baseline cubes**Require:** L : Input dimension count of B **Require:** F : Cover of B , where $|F| \leq |B|$ **Ensure:** E : Cover of B , where $|E| = |F|$, such that each cube in F is a subcube of some cube in E

```
1  $E = F$ 
2 foreach  $c \in E$  (random order)
3    $e = c$   $D = 1 \dots L$ 
4   foreach  $d \in D$  (random order)
5      $e' = \text{raise}(e, d)$ 
6     if ( $e'$  does not contradict  $B$ )
7        $e = e'$ 
8    $E = E - \{c\} \cup \{e\}$ 
9 return  $E$ 
```

3.3 Cube list indexing

Naïve lookup of a position in the resulting cube list would require scanning all cubes to find the one(s) that match the input position's bit vector, and then returning the associated output. This is too slow, so we build an index tree over the cube list to accelerate lookup at table load time.

Our algorithm for building the cube index is similar to that of building a decision DAG: The index is a tree with internal nodes corresponding to checking one bit of the input. It is built greedily and recursively. However, there are a few differences.

Because the cubes to be indexed are not bit vectors, but rather $\{0, 1, *\}$ -vectors, constructing a binary tree would require the duplication of cubes: If an internal node tests bit i , for any cube for which bit i is $*$, it would need to be duplicated on the 0 and 1 subtrees. This would negate the advantages of the two-level logic minimization that produced the cubes in the first place. Instead, we build a ternary tree with three branches at each internal node. Any cube so indexed belongs to only one of the 0, 1, or $*$ branches.

We use a modified Gini impurity score to score a potential internal node. If n_0 , n_1 , and n_* are the number of cubes that would be sorted into the 0, 1, and $*$ branches, respectively, we use the score of $n_0(n_0 + n_*) + n_1(n_1 + n_*) + 2n_*^2$. The first two terms represent the number of items to be scanned ($n_0 + n_*$ and $n_1 + n_*$) times the relative frequency with which they would be scanned. The last term penalizes large number of examples in the common subtree.

We do not build the index structure all the way to single leaves. Rather, empirical tests indicate that performing a linear scan of 10 to 20 cubes is faster than refining the index further, so we stop when the number of remaining cubes is 16 or fewer. A lookup consists of recursively descending the tree, checking either the 0 or 1 branch first, followed by the $*$ branch if necessary. Whenever a leaf is reached, the cubes associated with that leaf are scanned linearly. Because each leaf has a unique set of cubes, it is not possible to convert this tree into a DAG.

4 Multiterminal reduced ordered binary decision diagrams

A fully-reduced, ordered BDD (FBDD) [3], the most common type of BDD, encodes a Boolean-valued function in a DAG with the two terminal nodes 0 and 1. The fixed order in which variables are encountered on all paths from the root to any terminal node (unlike

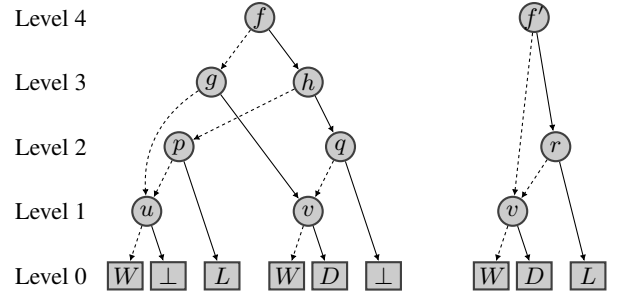


Figure 1. MTBDD encoding f (left), and a concretization f' of f (right).

the decision DAGs of Section 2) allows for both efficient operations combining BDD-encoded functions (not used in this work) and greater opportunity to merge common subDAGs.

We could encode partial function f described by an n -piece EGT using five FBDDs encoding $f_i : \mathbb{B}^{6n+1} \rightarrow \mathbb{B}$, for $i \in \{W, C, D, B, L\}$, where $f_i(x) = 1$ iff $f(x) = i$, so that the *don't care set* $\mathcal{D} = \{x : f(x) = \perp\}$ would be implicitly given by $\{x : \forall i, f_i(x) = 0\}$. Instead, we use a single *multiterminal reduced ordered binary decision diagram* (MTBDD) [4], a generalization of FBDDs that allows an arbitrary set of terminal nodes. MTBDDs encode total, rather than partial, functions, so we treat f as a total function of the form $f : \mathbb{B}^{6n+1} \rightarrow \{W, C, D, B, L, \perp\}$. Probing an MTBDD once is both simpler than probing multiple FBDDs in parallel and, in expectation, faster than probing multiple FBDDs serially.

Formally, a K -variable MTBDD is an edge-labelled DAG where the terminal nodes belong to an arbitrary finite set $\mathcal{R}$, and are at level 0, while each nonterminal node p is at a level $p.\text{level} = k \in \{1, \dots, K\}$ and has a 0-child, $p[0]$, and a 1-child, $p[1]$, satisfying $k > \max\{p[0].\text{level}, p[1].\text{level}\}$ and $p[0] \neq p[1]$ (i.e., there are no *redundant* nodes). MTBDD node p at level k encodes function $f_p : \mathbb{B}^k \rightarrow \mathcal{R}$, defined recursively as $f_p(i_1, \dots, i_k) = f_{p[i_k]}(i_1, \dots, i_{k-1})$ if $k > 0$, otherwise $f_p = p$. Fig. 1 shows two example MTBDDs, encoding functions $f, f' : \mathbb{B}^4 \rightarrow \{W, D, L, \perp\}$. In the figure, dashed edges point to the 0-child, solid edges point to the 1-child.

An advantage of the BDD approaches we discuss is that we naturally store multiple BDDs of the same type in a single *forest* with shared nodes, especially at lower levels. Thus, the node count of the forest storing *all* n -piece EGTs together cannot exceed, and is in practice less than, the *sum* of the node counts needed to store each n -piece EGT individually.

4.1 Concretization

In our application, we never evaluate $f(x)$ for $x \in \mathcal{D}$. Thus, the MTBDD size may be reduced by *concretizing* f , that is, changing some of the values of f from $\perp$ to (appropriate) values $i \in \{W, C, D, B, L\}$.

Finding a concretization f' of f with minimal size (which can be shown to be *full*, i.e., $f'(x) \neq \perp$ for all $x \in \mathbb{B}^K$) is NP-hard, so we limit ourselves to greedy (suboptimal) heuristics. We use three increasingly expensive and increasingly general heuristics originally proposed for FBDDs [13], but with notable differences: in Shiple et al. [13], a partial Boolean-valued function f is encoded using *two* FBDDs: one to encode f , and one to encode “don't care” set $\mathcal{D}$ (or, equivalently, the “care set” which is the complement of $\mathcal{D}$). The

heuristics must then consider the two FBDDs simultaneously to perform the concretization. Our versions of the heuristics, instead, use a single MTBDD where the “don’t cares” are encoded as terminal node $\perp$. The heuristics proceed top-down in the MTBDD and attempt to remove each nonterminal node p by making it redundant as follows (let p' and p'' be the two children of p , in either order):

- *Restrict*: If $p' = \perp$, change p' to p'' . When applied to function f in Fig. 1, *restrict* would eliminate nodes u (replaced by W) and q (replaced by v).
- *One-sided-match*: If p' is a (not necessarily full) concretization of p'' , change p'' to p' . When applied to function f in Fig. 1, *one-sided-match* would eliminate nodes g (replaced by v), q , and u .
- *Two-sided-match*: If p' and p'' admit a common (least, thus not necessarily full) concretization q , change p' and p'' to q , which may be an existing node or a new node. When applied to function f in Fig. 1, *two-sided-match* would produce function f' : node r concretizes both nodes p and q , while node v concretizes node u ; thus node h is eliminated and replaced by r , and node g is eliminated and replaced by v .

As these are heuristics, none is guaranteed to be best. However, experiments we performed using five-piece EGTs showed that *restrict* and *one-sided-match* tend to have similar node savings w.r.t. no concretization, while *two-sided-match* was almost always best, sometimes by a substantial factor. Five-piece EGTs concretized with *two-sided-match* require only 44.2% of the nodes versus using no concretization. When storing these EGTs in a forest, the node counts are only 78.4% (with no concretization) and 91.1% (with *two-sided-match*) of the total node counts for the individual EGTs. Thus, using forests does not help as much in conjunction with concretization, but the forest with *two-sided-match* still requires only 59.9% of the nodes of the forest with no concretization.

Finally, we observe that, while BDD implementations may vary widely in the way they store edges (pointers vs. indices) and levels (the size of the integer types), once the total count T of (terminal or nonterminal) nodes in the forest is known, it is a simple matter to encode the entire MTBDD forest using

$$T \cdot (2 \lceil \log_2 T \rceil + \lceil \log_2 K + 1 \rceil)$$

bits, where K is the number of variables, while still providing efficient query time.

5 Experimentation

We measure and discuss data preparation, the size of the data of the various methods, then measure and discuss their probing efficiency.

5.1 Methodology

We downloaded the Syzygy WCDBL tables[1], and updated the code of Fathom [6] to return an appropriate result whenever an invalid or illegal position is probed. Syzygy tables do not include game-theoretic values for positions where either side has not already permanently lost their right to castle.

We probed every bit vector of each three-, four-, and five-piece EGT, and recorded the result returned by Fathom in a binary file ordered by the numerical value of the bit vector. The bit vector representation we use accommodates piece coordinates, the side to move, and the right to capture en passant. To represent a white (black) pawn that is at risk of being captured en passant, we encode it as being

placed on the first (eighth) rank instead of the fourth (fifth) rank where it actually resides.

Subsequently, we used this extracted information as our training data for decision DAG, TLLM, and MTBDD EGT construction. All 145 decision DAG and 145 MTBDD EGTs were constructed within a few hours. However, the construction of TLLM EGTs requires substantially more time.

For each EGT, we iterated through $n = 1 \dots N$, performing two operations: a single distance- n merge sweep immediately followed by an IRREDUNDANT pass. The application of IRREDUNDANT reduces the cube count of the working set as quickly as possible, so that more challenging functions can be minimized using a given amount of computational effort.

For three-piece EGTs, we set $N = 11$ (higher N introduces no further changes). For four-piece and five-piece EGTs, we used $N = 9$ and $N = 5$, respectively.

Subsequently, we performed a single REDUCE pass, a single RANDOM-EXPAND pass, then attempted to perform a single further IRREDUNDANT pass, and one final REDUCE pass (which, while not reducing cube count, tends to improve the compressed size on disk).

5.2 Data size

We first consider the size of the compressed data on disk, then give some observations about runtime memory usage.

5.2.1 Disk space

Of the four methods, decision trees provide the least compact disk representation (see Table 1 and Figure 2). One practical advantage of the MTBDD representation is that its tables may be stored together, rather than as 145 distance MTBDDs (one for each of the material balances with three, four, or five pieces), without adjusting the encoding scheme: this yields a reduction in overall space usage of a few percent. TLLM uses less space than MTBDD even when this is taken into account. However, Syzygy’s disk representation is clearly the most compact.

5.2.2 Memory usage

The MTBDD, decision tree, and TLLM methods all require roughly a constant factor across tables more for memory versus disk space. However, in the case of TLLM, that constant factor is relatively higher, because indexing of the cube lists is required to probe the TLLM tables efficiently (these indices are constructed rapidly at data load time). In contrast, the decision tree and MTBDD structures do not require additional augmentation after in-memory decompression of the on-disk data. Of these three methods, MTBDD is most efficient from a memory usage perspective.

The memory usage of Syzygy substantially differs from the three other methods we explore, and is difficult for us to characterize. Syzygy EGTs are cleverly engineered to store blocks of compressed data, so that it is not actually necessary to unpack the entirety of its tables into memory to perform any single probe. However, because Syzygy’s data encoding requires that a minimax-based capture search be performed when any captures are available, it follows that compressed blocks from multiple tables may actually need to be probed to resolve the game-theoretic value of the particular position under consideration.

method subset	disk space used (MiB)				memory space used (MiB)				average query time (μ s)			
	3 pc	4 pc	5 pc	all	3 pc	4 pc	5 pc	all	3 pc	4 pc	5 pc	all
Flat file mmap	0.0	2.8	640.5	643.3					1147.0	1365.5	1178.8	1216.3
Decision DAGs	0.0	4.5	1372.5	1377.0	0.0	11.4	3038.5	3049.9	1221.8	1449.7	1312.1	1337.5
MTBDD	0.0	2.7	782.9	785.6	0.0	7.1	1763.5	1770.6	1198.1	1467.2	1513.4	1492.9
TLLM	0.0	2.0	634.8	636.8	0.1	13.2	4160.9	4174.2	1335.5	1804.2	3935.4	3404.8
Syzygy WCDBL	0.0	1.1	348.4	349.6					7271.5	12496.3	17160.8	15854.7

Table 1. Experiment results. Disk space is for files after compression with xz, using options `-T1 -lzm2=preset=9e,dict=1GiB,mf=bt4,mode=normal,nice=273,depth=1000`. Memory space includes the index for TLLM. Syzygy does not load the entire table into memory at once, but also requires additional tables for querying. Full distributions are shown in Figures 2 and 3.

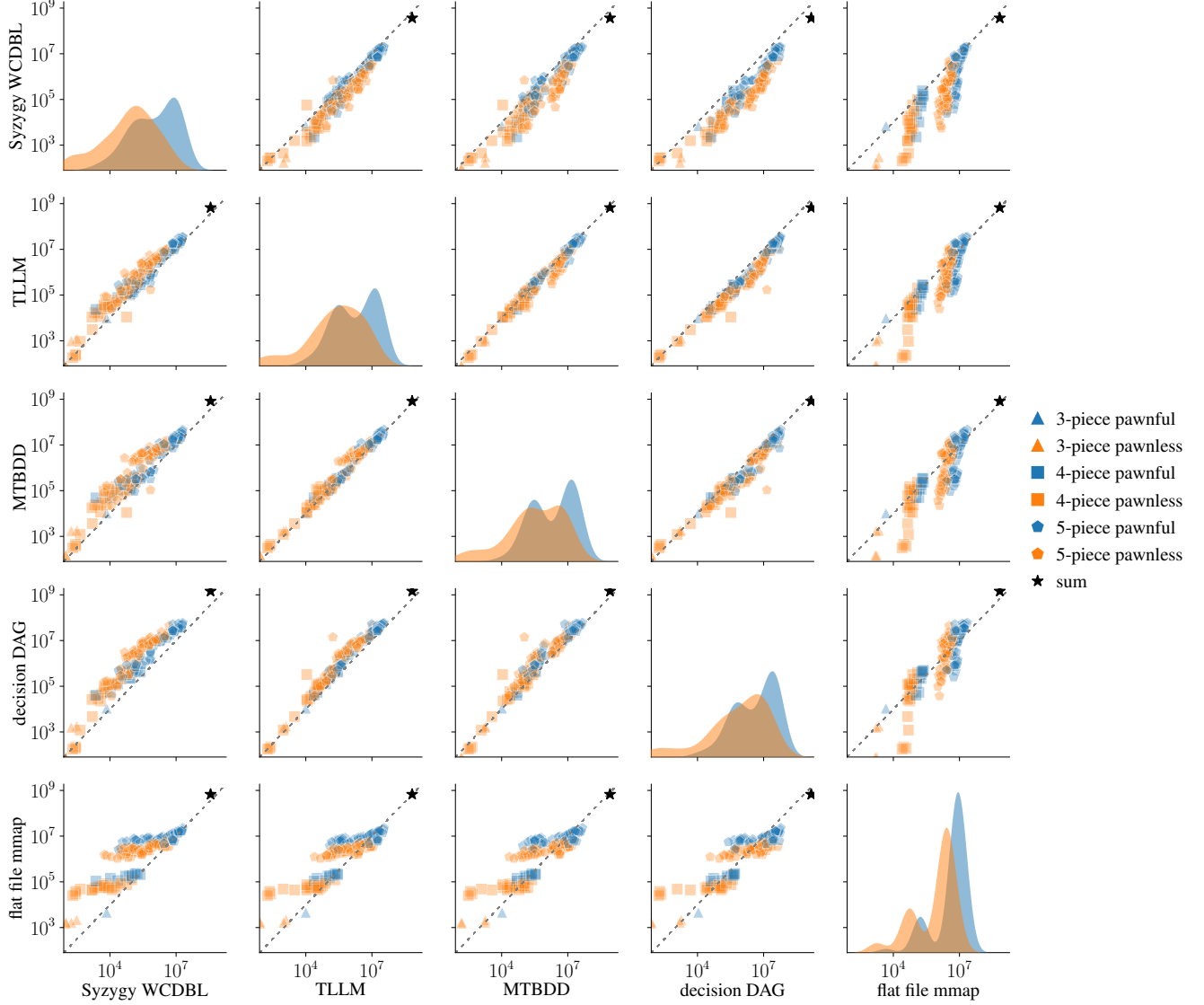


Figure 2. Pair plot of bytes required to store compressed WCDBL data on disk. Each point is one material balance. Orange points are pawnless material balances; blue points are those with at least one pawn. Triangles are 3-piece tables; squares are 4-piece tables; pentagons are 5-piece tables. Off-diagonal plots are scatter plots comparing two methods. Diagonal plots are (kernel-smoothed) distributions over the space per table.

5.3 Probe timing

For each of the 145 material balances, we sampled two million positions with replacement by drawing each of the n pieces, then drawing

the side to move. Every invalid or illegal position thus selected is immediately discarded and redrawn. Because en passant positions are encoded as illegal positions, none are present in the probe timing set. Fathom provides the quiescence search necessary to query Syzygy

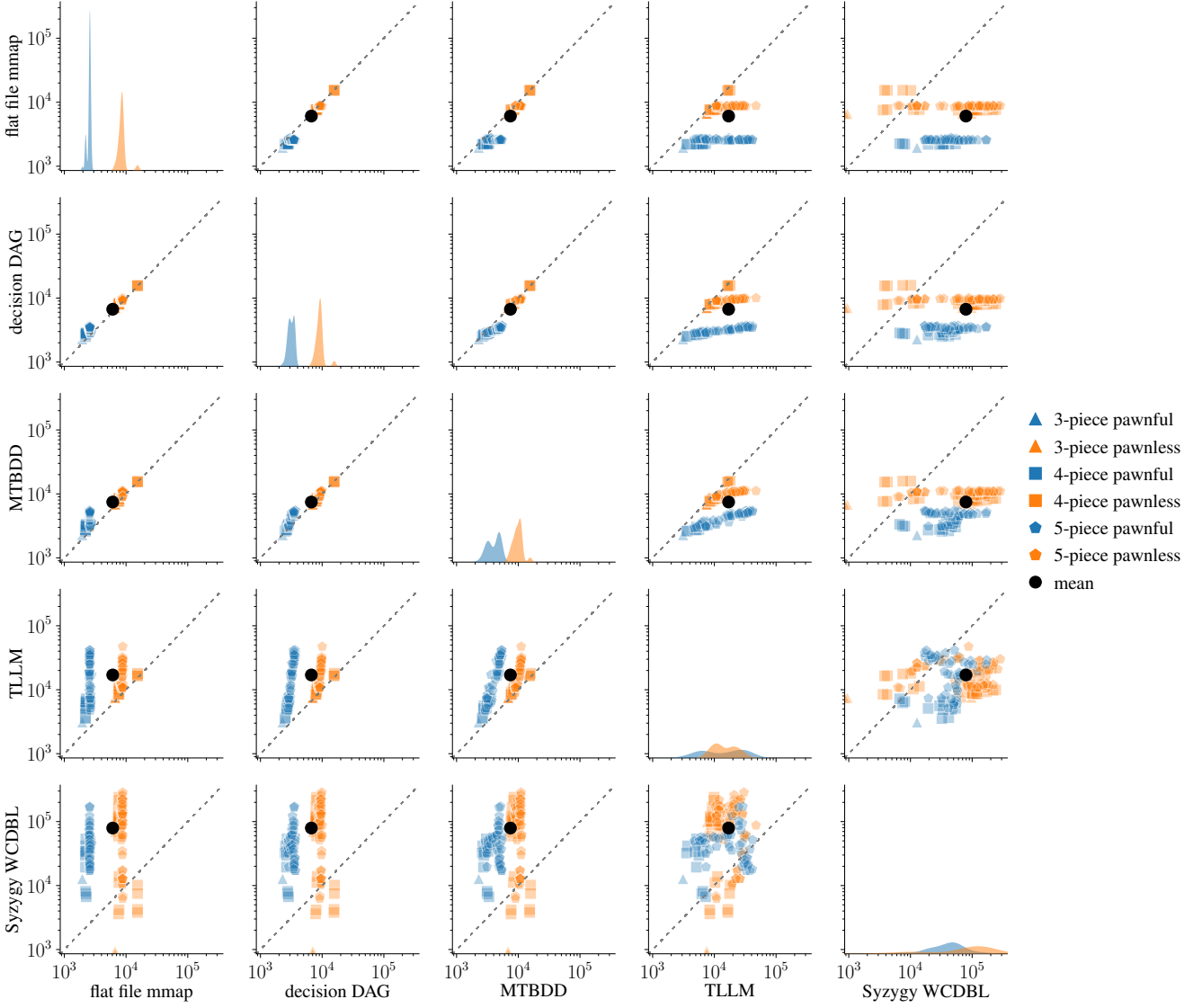


Figure 3. Pair plot of mean nanoseconds required to probe WCDBL data. Interpretation is the same as 2, but represent the (mean) querying time of the compressed representation, once loaded into memory.

EGTs.

For each of the three groups of n -piece positions, two probing passes are performed. The correctness of the probe results returned is verified on the first pass; probe timings are captured on the second pass. We then compute the mean probe time for each material balance within the group of n -piece positions. The probe timings reported are from a system with only a solid-state drive. (When an electromechanical disk drive is used instead, the performance of each method deteriorates.)

As shown in Table 1 and Figure 3, the decision DAGs and MTBDDs are probed most quickly on average, with the decision DAGs exhibiting only a slight speed advantage. In comparison, the mean probing speeds of TLLM and especially Syzygy are poor.

6 Discussion

Syzygy is both the most space-efficient and the least runtime-efficient choice. Why is this so? By design, Syzygy stores incorrect values for Chess positions containing legal captures to achieve better compression. The trade-off for the reduction in space thereby achieved is that querying requires the use of a capture-based quiescence search and the minimaxing of the resulting values, which lengthens the time required to return the desired information. It would be interesting to explore removing this transformation from Syzygy and/or adding it to the other techniques to be able to discern with precision the effects of this particular choice.

Decision trees provide the fastest query times, but are substantially less space efficient than MTBDDs. When implementing from scratch, decision trees are the quickest route to achieving good query performance.

MTBDDs provide both good compression and efficient query

time. We believe that other BDD-based techniques are likely to perform similarly well and are worth exploration. While not considered in this paper, BDDs can also be used to efficiently perform operations on sets of positions, which could enable generating the Chess EGTs in compressed format without explicitly enumerating all canonical positions.

TLLM exhibits both high potential for future result improvement and high risk of future obsolescence, depending on whether or not further effective breakthroughs in the efficiency and effectiveness of the minimization process are discovered. The disk space and results reported upon herein are reasonable for five-piece EGTs. That said, the challenge of handling the exponential growth in data size as even larger n -piece EGTs are processed currently seems the most daunting for this method.

Acknowledgments

The work of A. Miner and G. Ciardo was supported in part by the National Science Foundation (NSF) under grant CCF-2212142. Many computations were performed using the computer clusters and data storage resources of the High Performance Computing Center of the University of California, Riverside, which were funded by grants from NSF (MRI-2215705, MRI-1429826) and the National Institutes of Health (1S10OD016290-01A1).

During most of the period in which this research was conducted, the first author was also an employee of Google LLC, however, this research was performed with neither awareness of any applicable 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 Google LLC, the University of California, Riverside, or Iowa State University; no official endorsement should be inferred.

References

- [1] <https://tablebase.sesse.net>, 2018.
- [2] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Norwell, MA, USA, 1984.
- [3] Randal E. Bryant, ‘Graph-Based Algorithms for Boolean Function Manipulation’, *IEEE Transactions on Computers*, **35**(8), 677–691, (1986).
- [4] E. M. Clarke, M. Fujita, P. C. McGeer, K. McMillan, J. C.-Y. Yang, and X. Zhao, ‘Multi-terminal binary decision diagrams: an efficient data structure for matrix representation’, in *Proc. Intl. Workshop on Logic Synthesis*, (May 1993).
- [5] Ronald de Man. Syzygy. <https://github.com/syzygy1/tb>, 2020.
- [6] Basil Falsinelli, Jon Dart, and Ronald de Man. Fathom. <https://github.com/jdart1/Fathom>, 2015.
- [7] Dave Gomboc and Christian R. Shelton, ‘Lossless compression via two-level logic minimization: a case study using Chess endgame data’, in *Advances in Computer Games: Seventeenth International Conference, ACG 2021, Virtual Event, November 23-25, 2021*, eds., Cameron Browne, Akihiro Kishimoto, and Jonathan Schaeffer, volume 13262 of *Lecture Notes in Computer Science*. Springer Nature, (2021).
- [8] S. J. Hong, R. G. Cain, and D. L. Ostapko, ‘MINI: a heuristic approach for logic minimization’, *IBM Journal of Research and Development*, **18**(5), 443–458, (September 1974).
- [9] E. V. Nalimov, G. McC. Haworth, and E. A. Heinz, ‘Space-efficient indexing of Chess endgame tables’, *The Journal of the International Computer Games Association*, **23**(3), 148–162, (2000).
- [10] John Ross Quinlan, *Learning efficient classification procedures and their application to Chess end games*, 463–482, Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.
- [11] J.R. Quinlan, ‘Induction of decision trees’, *Machine Learning*, **1**, 81–106, (1986).
- [12] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen, ‘Checkers is solved’, *Science*, **317**(5844), 1518–1522, (2007).
- [13] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton, ‘Heuristic minimization of BDDs using don’t cares’, in *31st Design Automation Conference*, pp. 225–231, (1994).
- [14] T. Ströhlein, *Untersuchungen über Kombinatorische Spiele* (“Investigations on Combinatorial Games”), Ph.D. dissertation, Fakultät für Allgemeine Wissenschaften der Technische Hochschule München (“Faculty of General Sciences of Munich Technical University”), 1970.
- [15] Ken Thompson, ‘Retrograde analysis of certain endgames’, *The Journal of the International Computer Chess Association*, **9**(3), 131–139, (1986).
- [16] V. Zakharov and V. Maknhychev, ‘Creating tables of Chess 7-piece endgames on the Lomonosov supercomputer’, *Superkomp’yutery* (“Supercomputers”), **15**, (2013).