Set-Based Retrograde Analysis: Precomputing the Solution to 28-card Bridge Double Dummy Deals

Isaac Stone[1] , **Nathan R. Sturtevant**[1,2] , **Jonathan Schaeffer**[1]
[1]University of Alberta
[2]Alberta Machine Intelligence Institute (Amii)
{istone,nathast,jonathan}@ualberta.ca

Abstract

Among the most popular games played worldwide, Bridge stands out for having had little AI progress for over 25 years. Ginsberg's Partition Search algorithm (1996) was a breakthrough for double-dummy Bridge play, allowing a program to reason about sets of states rather than individual states. Partition Search supports the current state of the art for both bidding and cardplay. In the time since, virtually no progress has been made in Bridge bidding. Inspired by Ginsberg's idea, this paper presents *Setrograde Analysis*, a new set-based algorithm for perfectly solving Bridge hands. Using this approach, we have solved all 7-trick (28-card) hands – 10^{30} states, which can be reduced to 10^{17} unique states using preexisting techniques. This was done by considering five orders of magnitude fewer sets than the traditional state-based Retrograde Analysis algorithm. This work suggests that the entire 13-trick (52-card) state space can be solved with modern technology using this new approach. The 7-trick computation represents the largest endgame database to date in any game.

1 Introduction

Some of the early high-performance game-playing programs relied on Retrograde Analysis and endgame databases [Ströhlein, 1970] for strong play. The most notable example is Checkers, where 39 trillion (4×10^{13}) endgame positions, all those with 10 or fewer pieces, were used as part of the CHINOOK program [Schaeffer *et al.*, 1992], and for solving Checkers [Schaeffer *et al.*, 2007]. Endgame databases are also used widely in Chess programs [Chess, 2024], as well as in many other games (e.g., for solving Awari [Romein and Bal, 2003]).

Endgame databases are most effective in games where there are far fewer positions at the end of the game than elsewhere. As a result, they have not been applied in games that do not have this property. For instance, Sturtevant (2003) noted that in 3-player Chinese Checkers a winning arrangement of a single player's pieces in the game has approximately 10^{23} possible permutations of the other player's pieces, making it infeasible to store all the variations of even a single winning configuration. While in Chinese Checkers each player has a unique endgame configuration (the other side's piece locations are irrelevant), in Go the locations of both side's pieces in a terminal state are important. Hence these games require significantly different analysis [Berlekamp and Wolfe, 1994]. In a 4-player trick-based card game such as Bridge, the last two tricks have $\binom{52}{2}\binom{50}{2}\binom{48}{2}\binom{46}{2} = 1.9 \times 10^{12}$ possible deals of the cards, but only 16 ways for each deal to play out. It is trivial to solve but storing all states (as done in Checkers) is difficult.

These numbers suggest it might be impractical to build an effective Bridge endgame database for, say, 7 tricks (10^{30} states; 10^{17} unique states). This statement is only true under the assumption that every unique endgame state must be computed and stored independently. The contribution of this paper is to show how to avoid this assumption by representing an endgame state as a member of a set. This idea, along with other symmetry reduction techniques, makes it feasible to use Retrograde search to compute all 28-card double-dummy (DD) endgames — and store the solutions in just 1.4TiB of storage — something that was historically hard to imagine.

This paper describes our set-based approach to endgame databases, making the following contributions.

- We present a new set-based Retrograde Analysis algorithm, *Setrograde Analysis*, inspired by the ideas in Ginsberg's Partition Search [Ginsberg, 1996]. Whereas standard Retrograde Analysis computes a value for every state, Setrograde Analysis generalizes a state into a set where all members have the same game value. The powerful generalization step allows the algorithm to solve fewer states. Replacing states with sets leads to a large degree of problem reduction, by mapping a large endgame state-space to a smaller set-space.
- Setrograde Analysis is demonstrated using 28-card Bridge deals. The set database contains 5 orders of magnitude (OOM) fewer sets than there are states in a traditionally generated database. The database was constructed using 3 OOM less computing resources than would be needed for a traditional 28-card database. The 7-trick Setrograde database represents 2×10^{17} (200 quadrillion) unique states stored in 1.4TiB. Consider retrograde computations found in the literature. The 7-

piece chess endgame databases have 4×10^{14} positions using 18TiB [Chess, 2024]. Pentago, with 3×10^{15} positions, was strongly solved with 3.7TiB used to store just the top 17 ply of the tree [Irving, 2014]. Rubik's Cube pattern databases with 5.8×10^{12} entries were stored using 2.6TiB of disk [Hu and Sturtevant, 2019]. This Bridge computation is the largest endgame database computed to date.

This work improves on Retrograde Analysis, with computation and storage reductions allowing endgame database technology to scale to unprecedented levels. Using Setrograde with modern compute, our plan is to solve all 52-card endgames. These databases can be used to solve DD problems, but that is not our goal. Setrograde databases have the advantage that large sets of DD problems can be queried in a single operation, accelerating the exhaustive evaluation of state spaces that would be otherwise intractably large. Once complete, these databases open the door to evaluating and improving bidding strategies for double-dummy Bridge.

2 Background and Related Work

Double-dummy problems are the perfect-information variant of Bridge card play. A deal of n cards (with n as a multiple of 4) has an exact integer evaluation under perfect play, ranging from 0 to $\frac{n}{4}$ — the number of tricks won. See Fig. 1 (left) for a one-suit, two-card deal; the four players around the table are referred to as North, South, East and West.

The idea behind Retrograde Analysis is to solve a game from the end towards the start. In the example of Bridge, this works by enumerating all deals where each player has one card (1 trick), solving them, and then storing the results. Then one can move backwards in the game to consider all deals where each player has two cards (2 tricks). For a given 2-trick deal, taking the maximum result of all successor 1-trick deals (already computed) produces the correct result. Given sufficient computational and storage resources, one could continue to obtain the 3-trick results, and so on. Most often the algorithm is expressed as solving depth d (where d is the number of tricks in Bridge or the number of pieces on the board as in Chess) given the precomputed results from depth $d-1$. The computational cost and storage requirements typically grow exponentially in d. There are numerous enhancements to the basic algorithm that can improve its performance [Sturtevant and Saffidine, 2017]. Retrograde Analysis is typically used to compute and store the value for all states up to a given depth d, producing a comprehensive database (often called an endgame database or tablebase).

Since evaluations are discrete, n-card deals may be grouped into *consistent* sets, meaning that all states in the set share the same evaluation. Partition Search [Ginsberg, 1996] is a forward minimax solver with a set-based backup heuristic that generates a consistent set at each state in the search. Ginsberg showed a branching factor reduction by using these sets as transposition table entries. DD solvers based on Partition Search remain a state-of-the-art tool for modern mechanical Bridge players.

Bridge deals are colloquially described by the number of cards in each suit, and the ranks of relevant high cards. A low-valued card, with rank that does not affect the result, can be referred to as *x*. This notion is formalized in the set representation we use. Any card denoted *x* is interchangeable with any other *x*, and is strictly lower than a ranked card. For example, Fig. 1 (right) shows a set containing $6!/(2!)^3 = 90$ states that Partition Search might discover by generalizing Fig. 1 (left).

```
   N  |       ♠98                   ♠98
W    E|♠54          ♠32 → ♠xx              ♠xx
   S  |       ♠76                   ♠xx
```

Figure 1: ♠98 wins two tricks regardless of the locations of lower cards in the perfect-information deal (left) and set of deals (right); *x* refers to any low card. Play proceeds clockwise starting with North (N) until each player has played one card (East, South, West). The highest card wins. The winning player is next-to-play.

Partition Search maintains consistent sets by heuristically backing up cards that can be proven to never win by rank. Available DD solvers rely on expert heuristics for move ordering, set generation, and early search cutoffs. For Retrograde searching, this paper presents a more general algorithm for proving consistent sets and a framework in which the need for expert knowledge is eliminated.

There have been previous attempts at set-based search algorithms in games, but they bear little resemblance to our approach and they have not been applied (or are even relevant) to backward search. Setrograde Analysis is designed to strongly solve large state-spaces (finding the perfect play result for all reachable states) using Retrograde Analysis. Previous approaches have aimed at weakly solving state-spaces using a top-down approach. Retrograde search has seen success in board games including Checkers [Schaeffer *et al.*, 2007], Chinese Checkers [Sturtevant, 2020], and Chess [Chess, 2024]. Retrograde Analysis has been applied to a subset of Skat endgames [Furtak, 2013]. Most of the set-wise approaches that we know of, including Proof-Set Search [Müller, 2003], Method of Analogies [Adelson-Velsky *et al.*, 1988], and even Partition Search [Ginsberg, 1996], have been designed for top-down search, and targeted toward weak solutions [Haglund and Hein, 2014; Beling, 2017]. Set-based search has been applied successfully in planning [Edelkamp *et al.*, 2015] and concurrent work on board games [Considine, 2025] using Binary Decision Diagrams.

3 Overview

This section provides a high-level overview of a set-based approach to Retrograde Analysis. The ideas are illustrated using examples from Bridge.

3.1 State-Space Reduction

This section is specific to the game of Bridge, but is important for illustrating the search-space reductions that are possible in Bridge and in set-based search.

A 28-card endgame database contains the solution to all $\binom{52}{7}\binom{45}{7}\binom{38}{7}\binom{31}{7} = 2 \times 10^{29}$ ways to distribute the cards from a standard deck. Additionally, there are 5 trump suits to consider (clubs, diamonds, hearts, spades, and no trump). Hence there are 10^{30} deals represented in a 28-card database.

Cards	States	Upper Bound	Setrograde	Lower Bound
4	3×10^7	2×10^2	7×10^1	2×10^1
8	9×10^{12}	8×10^4	8×10^3	5×10^2
12	4×10^{17}	3×10^7	5×10^5	7×10^3
16	3×10^{21}	1×10^{10}	3×10^7	6×10^4
20	7×10^{24}	3×10^{12}	1×10^9	4×10^5
24	5×10^{27}	8×10^{14}	2×10^{10}	2×10^6
28	1×10^{30}	2×10^{17}	7×10^{11}	7×10^6
32	6×10^{31}	3×10^{19}	-	2×10^7
36	1×10^{33}	4×10^{21}	-	4×10^7
40	5×10^{33}	4×10^{23}	-	4×10^7
44	4×10^{33}	4×10^{25}	-	3×10^7
48	3×10^{32}	2×10^{27}	-	9×10^6
52	3×10^{29}	1×10^{28}	-	2×10^6

Table 1: State-space size and reductions for Bridge.

A well-known optimization used in card-game transposition tables is to represent cards using relative ranks instead of absolute ranks [Haglund and Hein, 2014]. If there are 8 spade cards in play, the lowest one is always represented as a 2, the next lowest as a 3, and so on. Thus, $\binom{13}{8}$ possible deals (1,287) are reduced to 1 representative deal. This does not affect card-play mechanics, but it reduces the number of 24-card states by roughly 12 OOM. For the 28-card database, several minor symmetry-related optimizations (not discussed here) can be applied to remove approximately one more OOM, leaving 2×10^{17} (200 quadrillion) unique states that must be evaluated and stored.

Table 1 provides the number of states given the number of cards remaining. *Upper Bound* is the number of states a state-based Retrograde solver would need to both solve and store (after rank and symmetry reductions). *Setrograde* is the number of *sets* stored in our set-based database. *Lower Bound* is a bound on the number of sets in a complete DD database, assuming an approach that partitions the state-space by the distribution of suits between the players. The data in this table is discussed in more depth later in the paper.

3.2 Set-Space Reduction

Retrograde Analysis (Alg. 1) works by iterating over all states at depth d (lines 3-4, 12-13), computing a state's value based on the successor states at depth $d - 1$ (line 5), and storing it in the database (line 7). Applying the same approach to 28-card Bridge would require repeatedly iterating over more than 10^{17} states, requiring storage of $\sim 10^{17}$ bytes or 100 petabytes depending on possible compression approaches.

Our set-based approach, *Setrograde Analysis*, builds a database of consistent sets. Alg. 1 shows a naive version of the algorithm. As with Retrograde Analysis, it iterates through all states at depth d (lines 3-4, 12-13). Each state is queried in the database to see if it matches any of the sets that have already been computed (line 5). If such a set is found, the state's value is known. Otherwise, a search routine finds a consistent set containing the new state (line 9), and adds it to the database (line 10).

We highlight here some of the challenges in creating a fast Setrograde algorithm:

Algorithm 1 Retro/Setrograde Analysis

```
1:  for $d \leftarrow 1..D$ do
2:      $EDB_d \leftarrow \emptyset$
3:      $s \leftarrow \mathit{firstState}(d)$
4:      while $s \neq \mathit{null}$ do
5:          $v \leftarrow databaseLookup(s, d)$
6:          if $\mathit{algorithm} = \mathit{retroGrade}$ then
7:              $EDB_d[s] \leftarrow v$
8:          else if $EDB_d[s] = \mathit{undefined}$ then
9:              $t \leftarrow \mathit{generalizeToSet}(s, v)$
10:             $EDB_d[t] \leftarrow v$
11:         end if
12:         $s \leftarrow nextState(s, d)$
13:     end while
14: end for
```

1. Generalization. This additional routine (line 9) is performed at every state. The cost of this operation must be less than the cost of performing the Retrograde operation for each of the states encompassed by the set returned.
2. Querying. It is more complex to look for a state in a database of *sets* than to look for a state in a database of *states* (line 5). $databaseLookup$ performs multiple queries at each state. Without a fast and scalable implementation, queries become a computational bottleneck.
3. Iteration. Iterating over all members of a large state-space will be expensive even if the cost per state is small. To scale computation, we must be able to iterate through the set-space without considering each state in the (much larger) state-space.

Each of these are briefly described. Generalization and iteration are then illustrated using Bridge.

Generalization: In Retrograde Analysis, when a state is reached that is not in the database, it will have its value computed and added to the database. Instead, Setrograde Analysis finds a generalization of the state (a set) in which all states have the same value (consistent). It does this using a generate-and-test approach, where sets are generated until the best consistent set (along some metric) is found. In Bridge, this is done by replacing some cards with *x*'s.

Generalization is illustrated in Bridge in Section 3.3. If all possible ways of generalizing a state are considered, this function would be very expensive. In practice the cost is reduced by considering only a subset of generalizations.

Querying: In Retrograde Analysis, a ranking or perfect hash function is used to map every state to a unique number. States are not explicitly stored; the offset in memory uniquely identifies a state. In Setrograde Analysis, sets must be explicitly stored. Thus, looking up a state in the database (line 5) involves matching a state to a set, a more complicated operation. Querying can be done efficiently using an appropriate data structure. One such data structure is discussed in [Stone, 2025]. The approach used lets the values of multiple states be retrieved in a single query.

Iteration: A state can be bypassed if it is a member of a set already in the database. Similar to how backjumping is used in DPLL search [Gaschnig, 1979] to avoid searching ir-

```
  a)  ♠98        |  b)  ♠98        |  c)  ♠98
♠3x       ♠54    | ♠xx       ♠54   | ♠xx       ♠5x
      ♠76        |      ♠76        |      ♠76
-----------------+-----------------+-----------------
  d)  ♠98        |  e)  ♠98        |  f)  ♠98
♠xx       ♠xx    | ♠xx       ♠xx   | ♠xx       ♠xx
      ♠76        |      ♠7x        |      ♠xx
-----------------+-----------------+-----------------
  g)  ♠9x        |  h)  ♠xx        |       N
♠xx       ♠xx    | ♠xx       ♠xx   |  W         E
      ♠xx        |      ♠xx        |       S
```

Figure 2: Sets generated by replacing 0 or more low-rank cards with *x*'s. East is on the lead.

relevant portions of a tree, it is possible to identify portions of the space that have already been solved, and jump past them. The exact approach is dependent on the state representation being used. One low-cost iterator is illustrated in Bridge in Section 3.4.

3.3 Example 1: Generalization in Bridge

In the following deal, East is on lead. North and South will take 2 tricks with the two highest spades:

```
      ♠98      |    N
♠32        ♠54 | W     E
      ♠76      |    S
```

To generalize this deal we must produce candidate sets. Here we use the 8 candidates (a-h) shown in Fig. 2 that were generated by replacing one or more low-rank cards with *x*'s. It can quickly be verified that sets a-f contain only states in which North and South take 2 tricks. For example, consider verifying set f using minimax search. East, South, and West each have one legal move (play an *x*), after which North plays either the 9 or the 8, winning the trick. The resultant set using relative card ranks is:

```
    ♠5
♠x      ♠x
    ♠x
```

Having taken one trick, North and South must take one more trick in this resultant set. A search for this set in the 4-card database returns a value of one trick for North and South. Therefore this candidate set is a valid generalization. Note that in general, the exact set being looked for may not be in the database. Instead the overlap of several entries may be equivalent to the set looked for. The value of the overlapped sets would be the minimum of their values.

Sets g and h each contain at least one deal in which North and South take only one trick, for instance:

```
     ♠96
♠87        ♠54
     ♠32
```

Consider trying to verify the correctness of g. Similar to the analysis of f above, East, South, and West again have one legal move and North has 2. North can play an *x* in which case North may fail to win the current trick. While a complex analysis can be performed when it is unclear which card wins a trick, it is sufficient here to note that in any case where North and South fail to win this trick, North and South will be unable to take 2 total tricks. Therefore, if North plays an *x* on best play, set g must not be consistent. North therefore plays the 9, producing the following resultant set:

```
    ♠x
♠x      ♠x
    ♠x
```

North and South must take one more trick in this resultant set. Searching the 4-card database finds the following entry, which overlaps the set being looked for. Here North and South take 0 more tricks:

```
    ♠x
♠5      ♠x
    ♠x
```

This means on North's action (the ♠9) set g is not consistent. Since North has no legal actions that leads to consistent resultant positions, set g is not consistent and cannot be added to the database.

More details can be found in [Stone, 2025].

3.4 Example 2: Low-Cost Iteration in Bridge

To reduce computation, we skip over states that are members of the set just added to the database. We only solve and generalize *independent* states — states not yet represented in the database. Naively, this can be done as follows: each time a set is added to the database, we generate one or more potential next-states. These states are constructed by applying minor changes to the previous set that give rise to states that are provably not represented by that set. The next states are placed into an open list (as in the A* algorithm [Hart *et al.*, 1968]) for further consideration. From the previous example, if we add to the database the set:

```
      ♠98      |    N
♠xx        ♠xx | W     E
      ♠xx      |    S
```

it would be redundant to evaluate any other state in which North holds ♠98. Therefore an independent state can be produced efficiently by trading the ♠8 for one of the ♠x cards. For example, we could produce the following set (one of three such possibilities):

```
     ♠9x
♠xx        ♠xx
     ♠8x
```

If this set is consistent then it will be turned into a state by replacing the *x*'s with low values (one of 720 possibilities):

```
     ♠97
♠32        ♠54
     ♠86
```

This new state goes onto the open list and eventually gets generalized using the process shown in Example 1. This process repeats until all deals in the state-space are represented by a set in the database. Through this process, a Setrograde solver can consider far fewer states than it would with the standard iteration over all states, reducing computation costs. Further

Algorithm 2 General Setrograde Analysis

```
 1: for $d \leftarrow 1..D$ do
 2:     $\mathbf{T}_d \leftarrow \emptyset$
 3:     $EDB_d \leftarrow \emptyset$
 4:     $s \leftarrow$ nextIndependentState($\mathbf{T}_d$)
 5:     while $s \neq$ null do
 6:         $v \leftarrow$ databaseLookup($s, d$)
 7:         $t \leftarrow$ generalizeToSet($s, v$)
 8:         $\mathbf{T}_d \leftarrow \mathbf{T}_d \cup t$
 9:         $EDB_d[t] \leftarrow v$
10:         $s \leftarrow$ nextIndependentState($\mathbf{T}_d$)
11:     end while
12:     compactEDB($EDB_d$)
13: end for
```

detail on this approach can be found in [Stone, 2025]. Iteration can be performed more generally and cheaply by maintaining a set of solved or a set of unsolved positions from which independent states may be directly constructed.

4 Setrograde Analysis

Now we turn to a more complete description of Setrograde Analysis, identify further bottlenecks, and describe how these are implemented efficiently.

These definitions are used to describe Setrograde Analysis:

- D: maximum Retrograde distance — in Bridge, number of tricks
- d: Retrograde distance $d \in 1..D$ (that is, distance to terminal state)
- $\mathbf{S}_d$: all states at distance d (from terminal state)
- $\mathbf{T}_d$: a set of sets of states such that all encapsulated states have a Retrograde distance d

Setrograde Analysis (Alg. 2) parallels its state-wise predecessor with three key modifications. First, a set $\mathbf{T}_d$ is maintained to track solved states (which are not stored explicitly but found in the union of all sets in $\mathbf{T}_d$). Second, at each iteration, a state $s \in \mathbf{S}_d | s \notin \mathbf{T}_d$ is evaluated — that is, states that are already solved are not re-evaluated. While re-evaluation does not occur in the state-wise formulation, it could occur in the set-wise formulation if the states were evaluated iteratively without validating independence. Finally, each solved state is generalized to a consisistent set t. Set t is stored in the database instead of the individual state.

Alg. 3 provides the helper functions needed for Alg. 2. The generalization process from a state to a consistent set can be done in many ways. One method is a backup heuristic, such as the one used in most modern Double Dummy solvers [Ginsberg, 1996]. The heuristic produces a single consistent set *t*. This approach opts for simplicity, ignoring generality at each step and incurring a compounding performance penalty in both computation and storage. For our Setrograde solver, we use a generate-and-test approach. Multiple candidate sets are produced, which may or may not be consistent. Each is evaluated for consistency. Any metric (for instance $|t|$) can be used to select which consistent set to add to the database.

Algorithm 3 Setrograde Helper Functions

```
# Returns value of a state.
function databaseLookup(state $s$, distance $d$)
    return $\max_{s' \in \text{succ}(s)} EDB_{d-1}[s']$
end

# Returns a state that is not yet present in the EDB.
function nextIndependentState(set of set of states $\mathbf{T_d}$)
    if $\exists s \in \mathbf{S}_d | s \notin \mathbf{T_d}$ then
        return $s$
    end if
    return null
end

# Returns whether all states in a set share a given value.
function ORACLE(set of states t, value v)
    return $\nexists s \in t | databaseLookup(s) \neq v$
end

# Returns a collection of candidate sets.
function generateCandidates(state s)
    return $\{t \in 2^{\mathbf{S}_d} | s \in t\}$
end

# Returns a consistent set.
function generalizeToSet(state s, value v)
    $t_{ret} \leftarrow s$
    $candidates \leftarrow generateCandidates(s)$
    while $\exists t_c \in candidatates\ s.t. |t_c| > |t_{ret}|$ do
        if $ORACLE(t_c, v)$ then
            $t_{ret} \leftarrow t_c$
        end if
    end while
    return $t_{ret}$
end
```

Setrograde Analysis is not guaranteed to produce the smallest possible database by any metric. The order in which states are evaluated can affect which sets are added to the database. The metric used to select which consistent candidate set is added to the database (or indeed whether multiple sets are added) can affect the composition of the final database. Some metrics (including $|t|$) can result in ties, and the tiebreak can affect the composition and size of the database. Anything that affects the size of the database may also affect computation time since speed is positively correlated with the number of states that are evaluated and generalized. Future work might establish stronger performance guarantees or tighter bounds.

The (unrealistic) lower bound on the number of sets required for a database (Table 1) is based on an ideal set representation in which all deals with the same game value are represented in a single set. In that case, we need exactly one set for each game value. In Bridge we partition the state-space based on the distribution of suits between the four hands. Since each partition requires at least one database entry (some partitions evaluate to a single game value), the number of partitions provides the tighter lower bound found in Table 1.

```
          ♠98          |          ♠xx
♠xx            ♠xx     |  ♠xx            ♠xx
          ♠xx          |          ♠98
-------------------------------
          ♠9x          |          ♠8x
♠xx            ♠xx     |  ♠xx            ♠xx
          ♠8x          |          ♠9x
```

♠9	♠8	♠7	♠6	♠5	♠4	♠3	♠2
1100	1100	x	x	x	x	x	x

Figure 3: Top: ♠98 wins two tricks regardless of the locations of lower cards in each set depicted. East is next-to-play in all diagrams. Bottom: Binary representation of the union of the sets above.

This partitioning of both the state-space and set-space also makes our implementation embarrassingly parallelizable, as each partition can be solved independently.

5 Implementation in Bridge

This section briefly mentions some of the important Bridge implementation details. Further information can be found in [Stone, 2025].

Set Representation: As discussed in the background section, and depicted in Fig. 1, representing sets of deals with low cards unspecified is not a new idea. This representation, with fixed cards and *x*'s is used in most (if not all) implementations of Partition Search. To reduce database size, we extend this methodology, by representing each card using four positional bits — one bit per player — indicating whether or not each player may hold this card. This representation allows for AND-OR conditions that are not possible with a simple x notation for low cards. We demonstrate the utility of this representation by example.

In each of the four sets in Fig. 3 (top), North and South can take two tricks by playing the highest two cards, one on each trick. Since the evaluation of each set is identical (North and South take two tricks on perfect play), the union of the four sets is itself a consistent set. The union could be expressed as (North holds the ♠9 *OR* South holds the ♠9) *AND* (North holds the ♠8 *OR* South holds the ♠8) *AND* (all lower cards are *x*'s). Using 4 positional bits per-card (in North-South-East-West order) we can represent that statement compactly. The ♠9 and ♠8 each have their bits set to true (1) corresponding to North and South holding those cards, and two bits false (0) corresponding to East and West set not holding those cards, as illustrated in Fig. 3 (bottom). Our databases consist of entries mapping from sets represented in this syntax to a bound on the number of tricks taken by each partnership.

Low-Cost Generalization: State generalization uses binary search. The lowest card in a suit can always be marked as x and it is possible that an entire suit (13 cards) could be xs. Hence for each suit a binary search is done on the number of xs. The program starts in the middle of the range and introduces that number of xs. Depending on whether the resulting set is consistent, the search either tries adding more or eliminating some xs. In some cases, the 4-bit representation used can compactly express the union of sets in the database, reducing database size. Compaction operations can be performed at insertion or in post-processing (Alg. 2, line 12).

Low-Cost Querying: State lookups, matching a state to a set to retrieve a value, are achieved using a depth-limited tree data structure. At each node, a bitwise AND operation can be used to determine whether a state is in a set. The tree structure lets us minimize the nodes that need to be tested. Each node contains partial information, and if a node does not provide a partial match, its subtree can be pruned. The tree benefits from locality, and we maintain relatively small independent trees for each distribution of suits between players (Lower Bound in Table 1).

Disk Storage: The 28-card database is partitioned into 7×10^6 independent pieces (Lower Bound in Table 1), each one reflecting a different distribution of cards (the deal's $shape$). Within each partition the sets are organized in a tree-like fashion. More details can be found in [Stone, 2025].

6 Experimental Results

Here we discuss Setrograde Analysis performance on Bridge database generation. The program is written in Julia, and compiled in version 1.8 or higher using the LLVM compiler. All code is compatible with version 1.11. The 24-card databases and some of the 28-card databases were computed on a machine with 48 cores, 187 GB of RAM, and 256 GB of swap using an Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz. The remainder of the 28-card databases were produced using several clusters provided by the Digital Research Alliance of Canada.

6.1 Performance of Setrograde Analysis

Table 1 shows one measure of Setrograde's performance. Upper Bound is the number of states that a Retrograde Analysis program would considered (in the reduced search space). Setrograde is the number of sets that were needed to capture the exact same information. For the 28-card database, there is a factor of 2.5×10^5 reduction. This is slightly too optimistic, as the cost to compute the value of a state is less than the cost of producing a consistent set.

In Table 2 the storage and computational resources for Retrograde and Setrograde Analysis are presented. Some of the Retrograde numbers were too costly to run and their values are extrapolated (indicated by a †). The Retrograde storage is pessimistic (two states per byte), given the potential for applying further data compression techniques. The Retrograde computation times are optimistic as they do not take into account performance loss due to scaling (e.g., loss of locality).

Database Size: The Retrograde database sizes reported used 4 bits per state, indicating a deal's value $0..d$. This representation reflects having a function that maps a state to a unique storage location with no gaps. Although we did not use such a function (it was too slow), the results are reported as if we did. Although additional compression techniques (not done) could yield small gains, Retrograde database sizes scale linearly with the number of states in the state-space.

In contrast, Setrograde has to store variable size sets (for 28 cards, these range from 8 to 48 bytes, but compression gives a factor of 5 reduction). Despite the additional storage for a

	DB Size (GiB)		Gen Time (CPU Days)		States/Byte
Cards	Retrograde	Setrograde	Retrograde	Setrograde	Setrograde
4	1×10^{-7}	7×10^{-6}	—	—	2.6×10^{-2}
8	4×10^{-5}	3×10^{-5}	—	—	3.2×10^{0}
12	2×10^{-2}	1×10^{-3}	3×10^{-3}	1×10^{-4}	3.1×10^{1}
16	†5×10^{0}	5×10^{-2}	1×10^{0}	1×10^{-2}	1.9×10^{2}
20	†2×10^{3}	2×10^{0}	†4×10^{2}	2×10^{0}	1.6×10^{3}
24	†4×10^{5}	5×10^{1}	†1×10^{5}	3×10^{2}	1.5×10^{4}
28	†9×10^{7}	1×10^{3}	†2×10^{7}	5×10^{4}	1.4×10^{5}

Table 2: Database generation time and storage for Bridge.

set, our Setrograde database is almost 5 OOM smaller than its Retrograde counterpart (before investigating data compression techniques for the Retrograde counterpart).

Generation Time: The generation time is presented in number of CPU days. For the 48-core machine, one day of wall-clock time corresponds to 48 CPU days.

Through 12 cards, the execution times are too small to draw meaningful conclusions. For the 16-card calculation, Setrograde Analysis is 2 OOM faster than Retrograde Analysis. The 20 and 24 card Retrograde computations were not performed. Both Retrograde and Setrograde Analysis are embarrassingly parallelizable for Bridge.

For 28-cards, it is difficult to report the number of CPU years used for the computation due to the heterogeneous computing resources used. Our best estimate is a runtime of 140 CPU years (equivalent to 3 years on a 48-Core machine).

States/Byte: The number of states divided by the size of the resulting database is a measure of information density. A Retrograde solver would store roughly 2 states per byte (somewhat higher with appropriate compression), regardless of the value of d. For Setrograde the information density grows with d. Increasing d means that a set, in general, reflects a larger number of states. For 28 cards, each byte in the database represents over 100,000 states on average.

6.2 24- and 28-card Performance

Setrograde Analysis decreases both storage and computation costs, rather than trading one for the other. To understand where the computation and storage costs are being reduced, we can break down the performance of Setrograde Analysis. In the 28-card case there are 2×10^{17} unique states.

Over 90% of computation time is spent evaluating and generalizing positions. For the 28-card database, this is estimated to be 2×10^{12} independent states; the 24-card computation evaluated 4.6×10^{10} independent states. Each evaluation and generalization step results in adding a set to the final database. A post-processing phase (*compactEDB* in Alg. 2) scans the database to identify sets that can be combined to form a single, more general set. This small additional cost results in reducing the 28-card database by half to 7×10^{11} sets; the 24-card database was reduced to 2.4×10^{10} sets.

Setrograde's generalize step has no counterpart in Retrograde Analysis – and it is expensive. On average, it increases the cost of evaluating a state by 1 OOM. This is a worthwhile tradeoff as it leads to a 4 OOM reduction in the number of 24-card states considered, and a 5 OOM reduction in the number of 28-card states.

For the 28-card database, the sets range from including 1 to 2,885,762,880,000 (nearly 3 trillion) states. The median set contains almost 10^4 states.

6.3 Impact on Double-Dummy Search

Our long-term aim is to build the 52-card database. Most of the value of the databases is not used by independent DD searches. DD search computes the value of states one by one, whereas our databases can return the values for a set of many positions at once. This can be leveraged by a program that analyzes bidding strategies – the ultimate goal of this work.

As part of validation, we performed experiments with a DD solver. The 6-trick database eliminates roughly 75% of the search tree, and the 7-trick database eliminates 90%.

6.4 Validation

A standard Retrograde implementation was used to verify the accuracy of each state in the Setrograde database through 16-cards. To scale further, independent partitions of the 20 and 24-card databases were verified exhaustively against a Retrograde implementation modified to use validated 16-card and 20-card Setrograde Databases as standard endgame databases. Finally, 2,000 13-trick DD problems were solved with our Partition Search implementation using the Setrograde databases to truncate the last 20, 24, or 28 ply. The value of each position in the tree was validated against a standard alpha-beta solver. In total, approximately 1 billion positions were validated from the 24-card database. Statistically, this provides a 99% confidence that less than 3.7 million positions (or $<0.0000005\%$ of the databases) are incorrect.

At this time, approximately 500 million positions from the 28-card database have been sampled and verified against a standard solver. 64-bit checksums are calculated for each independent database segment.

7 Conclusions

This paper introduces Setrograde Analysis, a generalization of Retrograde Analysis from states to sets. It has the potential to reduce the computational and storage needs by orders of magnitude. Other games for which Setrograde Analysis will be beneficial include Chinese Checkers and Skat.

Previously, Bridge endgame databases were not built because the massive search space and storage needs made it seemingly impractical. Setrograde makes this possible through 28 cards. The 32-card database is computable with today's technology. There are further improvements to Setrograde which we expect will reduce the number of sets created and reduce the computation time. It is now possible to imagine solving the entire 52-card deal space.

Acknowledgements

This research is generously supported by the Digital Research Alliance of Canada and the Canadian Natural Sciences and Engineering Research Council (NSERC).

References

[Adelson-Velsky *et al.*, 1988] Georgy M. Adelson-Velsky, Vladimir L. Arlazarov, and Mikhail V. Donskoy. *The Method of Analogy*. Springer, 1988.

[Beling, 2017] Piotr Beling. Partition search revisited. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):76–87, 2017.

[Berlekamp and Wolfe, 1994] Elwyn Berlekamp and David Wolfe. *Mathematical Go: Chilling Gets the Last Point*. AK Peters/CRC Press, 1994.

[Chess, 2024] Chess. Syzygy endgame tablebases. syzygy-tables.info, 2024. Accessed: 2025-06-12.

[Considine, 2025] Jeffrey Considine. Compressed game solving. In Michael Hartisch, Chu-Hsuan Hsueh, and Jonathan Schaeffer, editors, *Computers and Games*, pages 79–90. Springer, 2025.

[Edelkamp *et al.*, 2015] Stefan Edelkamp, Peter Kissmann, and Alvaro Torralba. BDDs strike back (in AI planning). In *AAAI Conference*, pages 4320–4321, 2015.

[Furtak, 2013] Timothy Furtak. *Symmetries and Search in Trick-Taking Card Games*. PhD thesis, University of Alberta, 2013.

[Gaschnig, 1979] John Gary Gaschnig. *Performance measurement and analysis of certain search algorithms.* PhD thesis, Carnegie Mellon University, 1979.

[Ginsberg, 1996] Matthew L. Ginsberg. Partition search. In *AAAI Conference*, pages 228–233, 1996.

[Haglund and Hein, 2014] Bo Haglund and Soren Hein. Search algorithms for a bridge double dummy solver, 11 2014. privat.bahnhof.se/wb758135/bridge/Alg-dds_x.pdf.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hu and Sturtevant, 2019] Shuli Hu and Nathan R. Sturtevant. Direction-optimizing breadth-first search with external memory storage. *IJCAI*, pages 1258–1264, 2019.

[Irving, 2014] Geoffrey Irving. Pentago is a first player win: Strongly solving a game using parallel in-core retrograde analysis. *CoRR*, abs/1404.0743, 2014.

[Müller, 2003] Martin Müller. Proof-set search. In Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson, editors, *Computers and Games*, pages 88–107. Springer, 2003.

[Romein and Bal, 2003] John Romein and Henri Bal. Solving the game of awari using parallel retrograde analysis. *IEEE Computer*, 38(10):26–33, 2003.

[Schaeffer *et al.*, 1992] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A World Championship Caliber Checkers Program. *Artificial Intelligence*, 53(2-3):273–290, 1992.

[Schaeffer *et al.*, 2007] 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, September 2007.

[Stone, 2025] Isaac Stone. Setrograde analysis: A set-based approach to fast computation and compact data representation. Master's thesis, University of Alberta, January 2025.

[Ströhlein, 1970] Thomas Ströhlein. *Untersuchungen über kombinatorische Spiele.* PhD thesis, Technical University of Munich, 1970.

[Sturtevant and Saffidine, 2017] Nathan R. Sturtevant and Abdallah Saffidine. A study of forward versus backwards endgame solvers with results in chinese checkers. In *Computer Game Workshop at IJCAI*, pages 121–136, 2017.

[Sturtevant, 2003] Nathan R. Sturtevant. *Multi-player games: Algorithms and approaches*. PhD thesis, UCLA, 2003.

[Sturtevant, 2020] Nathan R. Sturtevant. On strongly solving chinese checkers. In Tristan Cazenave, Jaap van den Herik, Abdallah Saffidine, and I-Chen Wu, editors, *Advances in Computer Games*, pages 155–166. Springer, 2020.