SEGClobber - A Linear Clobber Solver

Taylor Folkersen(✉)[0009−0004−8147−903X], Zahra Bashir[0009−0000−9491−0694], Fatemeh Tavakoli[0000−0002−3562−8928], and Martin Müller[0000−0002−5639−5318]

University of Alberta, Edmonton, Canada
{folkerse,zbashir1,tavakol1,mmueller}@ualberta.ca

Abstract. We develop SEGClobber (Simplest Equal Game Clobber), a solver for the one-dimensional form of Clobber known as linear Clobber. We briefly review previous work on this game and related concepts about combinatorial game theory. We describe our basic computational approach, a series of enhancements, and the experimental evaluation. We experimentally verify two conjectures by Albert et al. about linear Clobber up to values of $n = 50$ for $(●○)^n$ and $n = 28$ for $(●●○)^n$. This greatly exceeds previous computational results for such positions.

Keywords: Linear Clobber · Combinatorial Games · Minimax Search

1 The Game of Clobber and Linear Clobber

Clobber is a two player, partizan, perfect information game, created in 2001 by Albert et al. [1]. The game can be played on arbitrary graphs, but is traditionally played on a two-dimensional grid comprised of squares, which can be either empty, or filled with a black or a white stone belonging to players *Black* and *White* respectively. On a player's turn, their only legal moves are to move one of their own stones to *clobber* (replace) a neighboring opponent stone. A player unable to move on their turn loses.

Our work focuses on linear Clobber, played on a one-dimensional board. For example, from the position ●●○_○, Black's only move is to ●_●_○, leaving no more moves, and White's only move is to ●○__○, leaving one move for either player. Clobber is an *all-small* game: whenever one player has a move, so does the other player.

Clobber positions can be easily split into independent subproblems called *subgames*. Each group of contiguous stones makes up a single subgame. A nice property of linear Clobber is that positions break into subgames more quickly than in two-dimensional Clobber, as every move on the inside of a subgame splits it into two. Reasoning about subgames allows positions to be solved orders of magnitude faster than traditional full board search, and exploiting this key concept of combinatorial game theory (CGT) is at the core of our solver.

1.1 Related Previous Work on Clobber

Clobber was invented at the Games-at-Dal meeting 2001. In the standard starting position, black and white stones alternate in a checkerboard pattern. In

linear Clobber, this is written as $(●○)^n$. For example, $(●○)^2 = ●○●○$. Albert et al. proved important properties of this game, such as the NP-hardness of determining wins [1]. They also stated two conjectures:

Conjecture 1. Boards of the form $(●○)^n$ are first player wins for all $n \neq 3$.

Conjecture 2. $(●●○)^n = \lfloor (n+1)/2 \rfloor \cdot \uparrow$.

Here, $k\cdot\uparrow$ represents k copies of the well-known game *up*, $\uparrow = \{0|*\}$. At the time, these conjectures were computer verified up to $n = 19$ for Conjecture 1 and $n = 17$ for Conjecture 2 [1]. Using our solver, we verify these conjectures up to $n = 50$ and $n = 28$ respectively. Chen et al. recently proposed a mathematical proof of Conjecture 1 [3].

Uiterwijk et al. developed an alpha-beta Clobber solver which uses a database of *canonical forms* of all endgame positions up to 8 stones [7]. In contrast, our minimax solver avoids potentially expensive canonical form calculations. The work described here extends two graduate projects at the University of Alberta: a final course project [6] and a capstone report [5].

2 Basic Concepts of Combinatorial Game Theory

Our solver applies many ideas from *combinatorial game theory* (CGT). This section briefly reviews the most relevant concepts. More detailed explanations are given in textbooks such as [2].

One of the most important concepts used by our solver is that of a *sum* of *subgames*. A position with empty points such as $G = ●●○\_●○$ can be split into independent subgames, in this case $G_1 = ●●○$ and $G_2 = ●○$. The overall position is *equal* in the sense of CGT to the sum of these two subgames: $G = G_1 + G_2$. This implies that with optimal play by both, the sum game has the same winner as the original game.

In the *inverse* $-G$ of a game G the roles of both players are swapped. For example, $-G1 = -●●○ = ○○●$. Subtraction is defined by $G - H := G + (-H)$.

The game $0 = \{|\}$ is the neutral element in addition: for all games G, $G + 0 = G$, and $G - G = 0$. In CGT tradition, Black is also called Left and is associated with positive values, while White is called Right and prefers negative values. A move by either player leads to an *option*, which is another combinatorial game.

2.1 Outcome Classes and Dominance

Games can be partitioned into four *outcome classes*, shown in Table 1 below, based on who wins a game G, considering each player to play first. The outcome class also describes how a game G relates to 0. In the table, the symbol $\gtrless$ means "not comparable to" (G is not equal to, nor larger or smaller than 0).

The outcome class of a game G can be determined using two minimax searches. The relations $\leq$ and $\geq$ are defined in the usual way, so $G \leq 0 \iff$

Table 1: Outcome classes and G's relation to 0.

Outcome Class	Winner If Black First	Winner If White First	Relation To 0
$\mathscr{P}$ (previous)	White	Black	$G = 0$
$\mathscr{L}$ (left)	Black	Black	$G > 0$
$\mathscr{R}$ (right)	White	White	$G < 0$
$\mathscr{N}$ (next)	Black	White	$G \gtrless 0$

$G < 0$ or $G = 0$. Search of the *difference game* $G - H$ can be used to determine how two games G and H relate to each other: $G - H < 0 \iff G < H$.

Comparing games in this way enables pruning of *dominated moves* from positions, simplifying search. For example, if a game G with Black to play has two black options G_1 and G_2, and $G_1 \leq G_2$, then G_1 is dominated and Black can prune it. Similarly, if G_1 and G_2 are white options, White can remove G_2. Additionally, for Black options if $G_1 < G_2$, then G_1 is *strictly dominated*.

Definition 1. *For a player P and game G, a move is* sensible *if it is not strictly dominated by another of P's moves.*

Definition 2. *A set of* nondominated *moves for a player P and game G is a maximal subset of P's sensible moves, where no two moves have equal value.*

3 Search Techniques

This section describes the search algorithm of SEGClobber and its many components and optimisations. Given the current player `toplay` of a game, the recursive minimax `search()` function tries to find a winning legal move - a move to a position where the opponent loses. If `toplay` has no winning move, then the `opponent` wins. A *transposition table* stores win/loss results and heuristic values of previously-visited positions, and a pre-computed database provides information about subgames that dramatically speeds up search. The SEGClobber source code[1] is available under an MIT license.

3.1 A Normal Form for Linear Clobber

To avoid searching equivalent representations of a Clobber game multiple times, we impose a *normal form* on games, not to be confused with the *canonical form* of combinatorial game theory. The normal form is a unique, equivalent representation of a linear Clobber board. We exploit several observations about the properties of Clobber.

We define the *shape* of a game as the sequence of the lengths of its subgames. For example, the shape of ●○●_●●○●_ _ _○_○●_ is $(3, 4, 1, 2)$. Subgames of length 1 have no moves, so they can be omitted. Subgames can also be mirrored and/or re-ordered without changing the play of a game. For example,

[1] SEGClobber can be downloaded at: https://github.com/tfolkersen/SEGClobber

○○●_●○●● is equivalent to ●○●●_●○○. A single empty square is sufficient to separate subgames.

We define an ordering over subgames of the same length by encoding ● = 0, ○ = 1, and interpreting the resulting string as a binary number. We define a game to be in *normal form* if all the following five properties hold, and in *relaxed normal form* if at least 1, 2, and 4 hold:

1. There are no subgames of length 1
2. All redundant empty squares have been removed
3. Each subgame is greater-or-equal to its mirror image in the ordering above
4. Subgames are sorted in decreasing order of the numbers in the shape
5. Equal length subgames are ordered by the binary number ordering above

3.2 Database: Overview

We precompute and use a database containing all relaxed normal form games of lengths 2 through 16 inclusive. This covers all $3^{16} = 43,046,721$ boards of length 16, using only $866,924$ entries on disk, amounting to 42.2 MB. These numbers could be further reduced by storing only normal form games, and unifying entries for games and their negatives, but this would make database lookups algorithmically slower and more complex. The database contains both single subgames and sum games with a total length within the limit. A database entry contains the following information for a game:

- Outcome class ($\mathscr{L}$, $\mathscr{R}$, $\mathscr{N}$, $\mathscr{P}$)
- The nondominated moves for each player
- Lower and upper bounds on the game's value (see Section 3.5)
- A complexity score (see Section 3.3)
- A link to the *simplest equal game* (SEG) in the database (see Section 3.4)
- The game's shape, and its stones encoded in a fixed width binary number
- Index of a simplest (by complexity score) sensible move for each player

Using this data speeds up search dramatically, as shown in the experiments of Section 4.2. When outcome classes or bounds are known for all subgames, then the outcome of a position can sometimes be determined without further search. Dominated moves are removed, pruning branches of the search space. Games are replaced with *simpler* games, as determined in part by the *complexity score*.

3.3 Complexity Score

During minimax search, subgames and sums can be replaced by simpler but equal games, having a lower *complexity score* (CS) (with some caveats discussed in Section 3.4). There are several intuitive choices for CS:

1. CS_1: The length (number of stones and empty squares) of G
2. CS_2: The number of moves available to each player

3. CS_3: The number of sensible moves available to each player

Empirically, CS_2 works better than CS_1, and CS_3 works better than both of the others. In addition to these primitive complexity scores, we define a fourth, recursive score CS_4 based on CS_3:

$$CS_4(G) \coloneqq CS_3(G) + \sum_{L \in LO(G)} CS_4(L) + \sum_{R \in RO(G)} CS_4(R) \tag{1}$$

LO and RO are the sensible Left and Right options of G, and the sum over an empty set is defined to be 0. This score, which evaluates the whole subtree below G, is better than the other scores (see experiments in Section 4.2).

3.4 Using Complexity Score

Given two games G_1, G_2 with $G_1 = G_2$ and $CS(G_1) > CS(G_2)$, we would like to substitute G_2 for G_1, however, this must be done with caution so as to avoid causing substitution cycles during search: Consider the game $G = \circ\bullet\circ\circ\circ\bullet\bullet\circ\circ\bullet$. White's first available move is to $G^R = \_\circ\circ\circ\circ\bullet\bullet\circ\circ\bullet$. From here, Black's third available move is to $G^{RL} = \_\circ\circ\circ\circ\bullet\bullet\circ\bullet\_$, with $G = G^{RL} = \downarrow *$. While G has two sensible moves, G^{RL} has four. G has fewer sensible moves, but substituting G^{RL} by G results in an infinite loop in the search $G \to G^R \to G^{RL} \xrightarrow{replace} G \ldots$

We avoid cycles in two ways. First, for G_2 to be eligible to replace G_1 during search, G_2 must come before G_1 in the order of database entry creation (Section 3.5). This means that G_2 cannot be longer than G_1 (in terms of the number of empty squares and stones). Additionally, CS_3 and CS_4 are based on sensible moves, rather than nondominated moves (the latter causes cycles). For a game G, we define its simplest equal game $SEG(G)$ to be an equal game of minimal complexity score which comes before G in the database ordering. Such a game may not exist for a given G.

3.5 Database Generation

A subgame database is generated offline, then used for solving problems. This process uses the SEGClobber solver together with the partial database generated so far. The order in which games are added to the database has a large impact on performance. All relaxed normal form games are considered in order of increasing length, with ties broken first in order of decreasing number of subgames, second by bit pattern. For example, all games of shape $(3, 2, 2)$, with two empty squares, are computed before games of shape (9).

Database entry generation takes several passes over the data. The first two are done only for normal form games: outcome classes are determined, and lower and upper bounds are found along two scales: Multiples of *up* in the range $[-31\uparrow, 31\uparrow]$ and multiples of *up* plus *star* in the range $[-31\uparrow\star, 31\uparrow\star]$.

A third, final pass over all games computes missing outcome classes and bounds, and the nondominated moves for each game G are found. Among equal sensible moves, only the move leading to the simplest option according to the CS_4 score is kept. The CS_4 score is then computed for G. For each player, the index of a simplest sensible move is found. Last, $SEG(G)$ is found subject to the rules described in Section 3.4 (if such a game exists).

3.6 Finding a Simplest Equal Game Efficiently

Given G, efficiently finding $SEG(G)$ is non-trivial. For a large database $\mathscr{D}$, comparing G to all games coming before it is impractical.

We simplify the work required by defining a partitioning of all games in the database, according to a 5-tuple of outcome class, and lower and upper bounds along both scales. Given G's tuple t, it is only necessary to search the corresponding partition P_t. At the start of the database generation process, every P_t is initialized to be the empty set. G is compared to every game in the current instance of P_t. If no equal game is found, then G is inserted into P_t, and $SEG(G)$ does not exist. If a match G_2 is found, and $CS_4(G) > CS_4(G_2)$, then $SEG(G) = G_2$ (this is the only case where $SEG(G)$ exists). If instead $CS_4(G) < CS_4(G_2)$, then G_2 is removed from P_t, and G takes its place. Lastly, if $CS_4(G) = CS_4(G_2)$, then G_2 remains in P_t.

3.7 Sum Game Simplification

Each sum game $G = G_1 + \cdots + G_n$ encountered in minimax search is first simplified in a multi-step process, which includes replacing groups of subgames with their SEG, deleting zero games including pairs of games G_i and their inverses $(G_i + -G_i)$, and normalising the final sum game.

1. Subgames G_i that are not in $\mathscr{D}$ (whether not generated yet, or too large), are set aside temporarily.
2. Zero subgames $G_i = 0$, identified by database lookup, are deleted.
3. The remaining subgames are sorted in order of decreasing length, and sets of small subgames are replaced by simpler games as detailed in Section 3.4.
4. Using the string representations of all subgames, some are eliminated: single subgames containing no moves, and pairs of a game G_i and its inverse $-G_i$. This step is also done for the larger subgames set aside in step 1. For example, even without a database, in the game ●●○_●○_●○○, the pair of inverse subgames ●●○ and ●○○ = −●●○ would be recognised and deleted.
5. Finally, the resulting (sum) game is converted to normal form.

Game Replacement in a Sum Game These steps are done for the smaller subgames, excluding the too-large subgames outside of $\mathscr{D}$.

1. The sorted games are viewed as a single board (separated by single spaces), and a sliding window of length 16 is moved over the board.

2. If three or more subgames are within the window, then their sum S is searched in the database. If $\mathrm{SEG}(S)$ is known, it replaces S in G. If $\mathrm{SEG}(S)$ is not known, the rightmost subgame is removed from the window and the new resulting sum is searched again. This repeats until either SEG is known for the sum, or the window contains less than 3 subgames.
3. Next, all pairs of subgames are searched in the database, and possibly replaced by their SEG.
4. Finally, all single subgames G_i are searched and possibly replaced.

At each step above, games $G_i = 0$ are removed directly, instead of being replaced by $\mathrm{SEG}(G_i) = 0$. After each replacement or removal, the new simplified version of G is used for subsequent replacements, including any new subgames. New subgames are appended to the right side of G, and the process resumes without returning to a previous step.

3.8 Transposition Table

A *transposition table* `TT` with 4×2^{27} entries is used to remember outcomes of previously solved games, and to store best move information according to a heuristic for unsolved games. *Zobrist hashes* [8] are used to represent games and to index into `TT`. The Zobrist hash of a game is built from a persistent array `Z` of random 64 bit integers, with one random number `Z[location][color]` encoding each possible $(location, color)$ pair on a board. In Clobber, the three possible colors *empty*, *black*, and *white* require three integers per board location. The hash of a game is the XOR of the `Z` entries corresponding to all $(location, color)$ pairs of the board, plus one more code for `toplay`.

Of the 64 bit code, 27 bits are used to index into `TT`. The full code is stored in each table entry to avoid most hash collisions. Each index stores a group of four entries. A replacement policy keeps the most valuable entries. An entry value for position p is defined as $age \times depth^2$, where $depth$ is the depth of p in the search tree, and age is a number between 1 for the most recently used entry and 4 for the least recently used one. A new entry with a lower score replaces one with largest score among the group of four entries.

3.9 Move-Ordering Heuristic

The heuristic value of a position G is based on move counts. Let C_x be the number of moves for player x which decrease the remaining move count by more than 1. Then, the heuristic value $h(G)$ is given by Equation 2.

$$h(G) \coloneqq depth \times (C_{toplay} - C_{opponent}) + C_{toplay} \tag{2}$$

This value is updated, when playing moves, to the maximum of the negative of the heuristic of each position reached by a single move. The move corresponding to this value is played early in the move ordering in future searches of this position. A move resulting in a proven loss for the current player is never considered a best move.

3.10 Iterative Deepening

Iterative deepening helps guide search and improve move ordering [4]. Each iteration is depth-limited, starting with a limit of 1, and increasing by 1 in each search up to a maximum of 12. Non-decided positions at the depth limit are evaluated by the heuristic. The final search after reaching the maximum uses no depth limit. Each search has an additional limit on the number of *leaf nodes* that can be reached before the search is stopped. Leaf nodes are positions that are either at the depth limit, or are solved both without the transposition table and without a recursive call to `search`. The first iteration's limit is sufficient to play each move at the root position. Starting with the second iteration, the maximum leaf node count is 3, and is increased by a factor of 3 with every subsequent search.

3.11 The Search Algorithm of SEGClobber

The recursive minimax `search()` function implements many algorithmic improvements. A given board is first simplified according to the procedure described in Section 3.7. The board is searched first in the database, then the transposition table, and if found in neither, then several rules are attempted in order, to try and solve the position statically or quickly:

1. For every subgame, its outcome class and bounds are looked up in the database. If the outcome class of every subgame is known, then the win/loss result is solved in the following cases:
   - If all outcome classes are $\mathscr{L}$ ($\mathscr{R}$), Black (White) wins.
   - `toplay` wins if one subgame is in $\mathscr{N}$ and all others are wins for `toplay`.
2. If lower and upper bounds of all subgames are known, their sums are computed. If the sum of lower bounds is positive, then Black wins, and if the sum of upper bounds is negative, then White wins.
3. If Black is to play, and the database identifies a subgame $G_i > 0$, a separate search checks if Black can win $G - G_i$. If yes, since $G > G - G_i$, Black also wins G. Similarly, White to play wins G if $G_i < 0$, and White wins $G - G_i$.
4. If $G_i \in \mathscr{N}$, and the `opponent` loses $G - G_i$, then `toplay` wins G.

The *speculative subgame removal* in steps 3 and 4 is tried for all possible G_i where $G - G_i$ has at most 40 squares after simplification. Experimental results in Section 4.2 show that these smaller, less complex boards can sometimes be solved much more quickly, and may even be in the database. If the position is still unsolved, regular moves are generated until a win is found. If this fails and all options have non-heuristic values, then the position is a loss. Positions at the depth limit are evaluated by the heuristic. For move generation, first the known dominated moves are removed. If G contains multiple copies of the same subgame, then moves are generated only for one copy. The remaining moves are sorted and generated in the following order:

1. Subgames with unknown outcome, in the middle two quarters of their board

2. The best move according to the heuristic, if known
3. Simplest moves on lost subgames
4. Simplest moves on $\mathscr{N}$ positions
5. Remaining moves on lost subgames
6. Remaining moves on $\mathscr{N}$ positions
7. Remaining moves on subgames with unknown outcome class
8. Simplest moves on known wins for `toplay`
9. Remaining moves on known wins for `toplay`

In each subgame, moves are generated left to right. If `search()` returns a heuristic score, the transposition table is updated with the score and best move.

SEGClobber returns a winning move, so search at the root has simplified logic. Dominated moves are pruned, and a simpler move ordering is used: the best move according to the heuristic (if known), then other moves from left to right. Steps which would alter the board are omitted.

4 Experiments

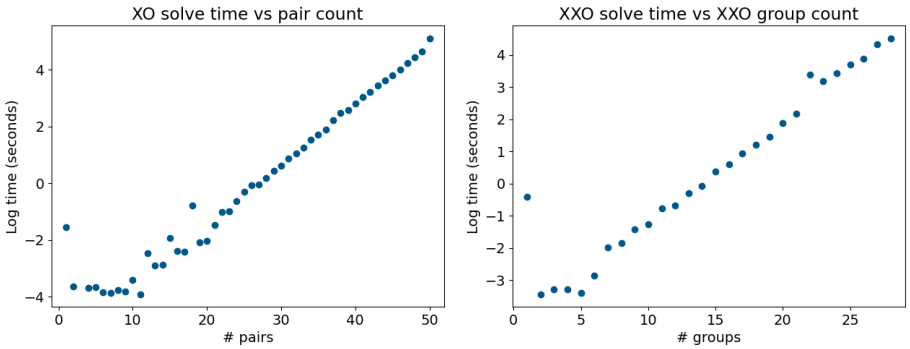
Experiments are run on a server with an Intel Xeon X5670 @ 2.93GHz CPU. We measure time instead of search node counts, so we run experiments sequentially to avoid resource contention between multiple SEGClobber instances. For verifying the conjectures, we use a transposition table with 4×2^{29} entries, which is approximately 38.65 GB. For our ablation studies, we use a transposition table with 4×2^{27} entries (approximately 11.27 GB). The conjecture experiments use a slightly more compact transposition table entry, saving 3 bytes per entry by using a smaller integer type for the depth value.

4.1 Extended Verification of Conjectures 1 and 2

We revisit the Conjectures 1 and 2 of [1] discussed in Section 1.1. Figure 1 shows our computational results for both conjectures, with the time on a logarithmic scale (with a logarithm base of 10). Each n is solved sequentially, starting with $n = 1$, and the transposition table is kept between each n. For both conjectures, the time to solve $n = 1$ takes significantly longer than the next few values of n, due to startup cost.

The left diagram (a) shows the time to prove $(\bullet\circ)^n$ as a first player win for all $n \leq 50$ except $n = 3$. Due to symmetry, we only verify that Black wins when playing first. A hardcoded move is played at the root node: Black moves the 13th (from the left) stone to its right. The largest board has 100 stones. Over a wide range of values from $n = 23$ to $n = 49$, the slope is close to linear, and each added $\bullet\circ$ pair makes solving about $1.63\times$ as slow. The largest board at $n = 50$ takes significanty longer, being $2.98\times$ as slow as $n = 49$. Our transposition table's size may be insufficient starting with $n = 50$.

Diagram (b) shows our results for $(\bullet\bullet\circ)^n$ up to $n = 28$: $(\bullet\bullet\circ)^n - \lfloor (n+1)/2 \rfloor \cdot \uparrow = (\bullet\bullet\circ)^n + \lfloor (n+1)/2 \rfloor \cdot \downarrow$ is a second player win. The largest board



(a) Time to solve $(●○)^n$ for increasing n. (b) Time to prove value of $(●●○)^n$.

Fig. 1: Scaling studies for (a) solving $(●○)^n$ as a first-player win (except $n = 3$), and (b) for proving that $(●●○)^n = \lfloor (n+1)/2 \rfloor \cdot \uparrow$.

has 140 squares: 84 contiguous stones followed by 14 copies of (○○●). Each increment to n is about 2.12× as slow as the previous one.

We greatly exceed previous computational results. For perspective, using this same transposition table size, and with a clean startup with no prior transposition table data, SEGClobber verifies Conjecture 1 for $n = 19$ in 0.084 seconds, and for $n = 47$ in 40421.63 seconds. With this same setup, Conjecture 2 is verified for $n = 17$ in 12.72 seconds, and for $n = 26$ in 14441.92 seconds.

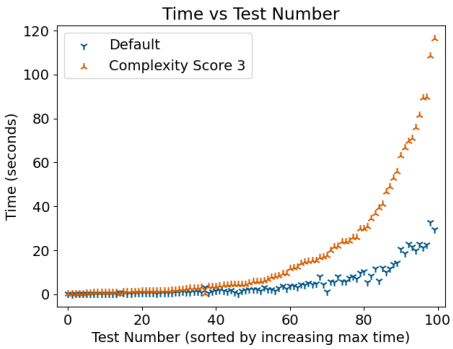
4.2 Ablation Studies

Figure 2 presents five ablations. Each study used a different set of 100 random boards with 60-66 stones and no empty squares, and case (d) uses 58. Each vertical segment of the graph represents the same board. If solving took 120 seconds or more for either of a board's two runs, then it was excluded and a new one took its place. SEGClobber was restarted for every run.

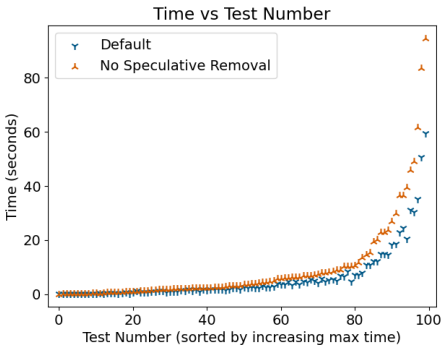
Diagram (a) demonstrates that the complexity score CS_4 is far superior to CS_3. Speculative subgame removal in (b) shows an improvement on the more difficult instances. Pruning dominated moves in (c) gives a modest but increasing gain. Substitution of subgames by their simplest equal game in (d) is a huge gain. Iterative deepening in (e) can give a noticable improvement, but only on a few boards. The heuristic evaluation function of Section 3.10 may only be well-suited to specific board patterns.

5 Conclusions and Future Work

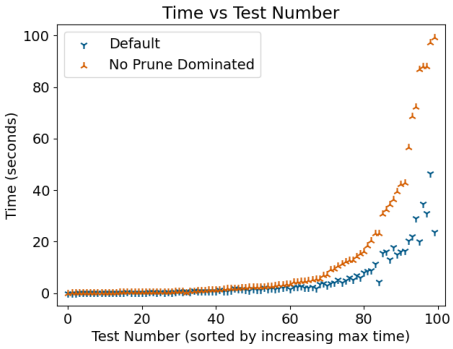
We developed a strong linear Clobber solver and far exceeded previous computational results. Key elements of this solver are based on combinatorial game theory concepts, taking advantage of subgame structure. The most effective are building



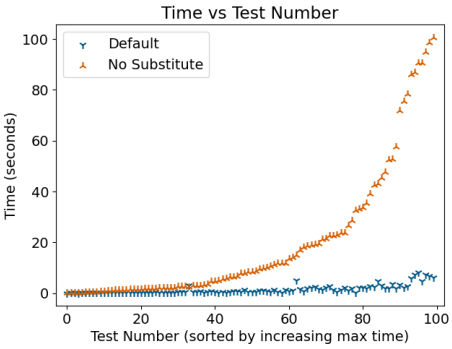
(a) Using Complexity Score CS_3



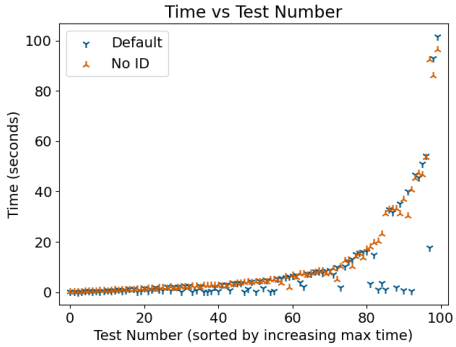
(b) No speculative subgame removal



(c) No pruning of dominated moves in database subgames



(d) No simplest equal subgame substitution



(e) No iterative deepening.

Fig. 2: Ablations - removing one component of SEGClobber.

a database of small sum games, substitution of subgames by their simplest equal game, an efficient transposition table based on a normal form of games, and using database information to simplify search by removing dominated moves.

The main limitations of our solver, to be addressed in future work, are a lack of parallel search, and the relatively small size of our database. The heuristic evaluation function may be improved, as it is likely only accurate for specific board patterns. Future exploration of complexity scores may yield even better results.

Acknowledgments. We gratefully acknowledge financial support from NSERC, the Natural Sciences and Engineering Research Council of Canada, and the Canada CIFAR AI Chair program.

Disclosure of Interests. The authors declare no competing interests for this article.

References

1. Albert, M., Grossman, J., Nowakowski, R., Wolfe, D.: An introduction to Clobber. Integers **5**(2), 12 pp. (2005), article A01, MR2192079
2. Albert, M., Nowakowski, R., Wolfe, D.: Lessons in Play: An Introduction to Combinatorial Game Theory. A K Peters, Wellesley, Massachusetts (2007)
3. Chen, X., Folkersen, T., Hasham, K., Hayward, R., Lee, D., Randall, O., Schultz, L., Vandermeer, E.: A proof of the 2004 Albert-Grossman-Nowakowski-Wolfe conjecture on alternating linear Clobber (2025), https://arxiv.org/abs/2509.08985
4. De Groot, A.D.: Thought and Choice in Chess. De Gruyter Mouton (1978)
5. Folkersen, T.: Linear Clobber solver (2022), Capstone report, University of Alberta
6. Tavakoli, F., Folkersen, T., Bashir, Z.: Strong 1-dimensional Clobber (2022), CMPUT 655 project report, University of Alberta
7. Uiterwijk, J., Griebel, J.: Combining combinatorial game theory with an alpha - beta solver for clobber: Theory and experiments. In: BNAIC. Communications in Computer and Information Science, vol. 765, pp. 78–92. Springer (2016)
8. Zobrist, A.: A new hashing method with application for game playing. Tech. Rep. 88, Univ. of Wisconsin (1970)