Strong 1-dimensional clobber

Fatemeh Tavakoli, Taylor Folkersen, Zahra Bashir

1. Introduction

Clobber is a partisan game invented by Albert, Grossman, Nowakowski, and Wolfe in 2001[2]. This 2-player game consists of black and white stones. Each player can "clobber" an adjacent opposite-colored stone, thereby, removing one of the opponent's stones and substituting the opposing stone with its stone. The well-recognized form of Clobber consists of a 2-dimensional grid graph with stones placed on the nodes. This project aims to work on the 1-dimensional format of Clobber which clearly consists of a row of stones; each can be Black, White or Empty. The Clobber game is all-small with infinitesimal values since there is an equal opportunity for each player to play at each state. Notably, Clobber does not have nice canonical forms which make solving it challenging.

1.2. Problem statement

This project aims to answer the following question.
How to create a strong solver for a 1-dimensional Clobber game utilizing CGT techniques?
A solver's strength is measured in terms of runtime and the number of visited nodes in the game tree search. This project verifies the hypothesis about the result of some games that follow specific patterns for large boards in a reasonable amount of time.

We are reporting detailed ablation results on different improvements that are not limited to CGT techniques. This project encompasses a wide variety of techniques applied in database creation and game tree search.
The baseline is an iterative deepening solver without any GCT techniques (explained in section 5.1) which is later improved several times using different approaches. The effectiveness of improvements has been analyzed and compared.

Up to the authors' knowledge, there is no previous work on 1-dimensional clobber testing different approaches as we do in this project. Also, there is little documentation and previous work on Clobber (especially the 1-d case). Additionally, we are not aware of any faster solver for 1-d Clobber which can verify the outcome of hypotheses and conjectures mentioned in [2] for larger values of n. This project pushes the boundaries of verifying $1 \times n$ clobber patterns conjectures in [2], and addresses the aformentioned problems which show the significance of our project.

2. literature review

The first paper published for applied combinatorial game theory in Clobber was released by [2] in 2005. In this paper, GT values of some boards (such as $1 \times n$ and $2 \times n$ boards) are computed. Also, atomic weights are introduced as a way of preventing complicated infinitesimals.
In 2007, Siegel in [6] developed a useful tool for analyzing Amazons, Domineering, and Clobber boards. This tool which is known as "CGSuite" computes the CGT values (in the canonical form) and the atomic weight of different boards. Also, Classen in [3] improved a clobber playing engine using some exact values incorporated in MCTS in his thesis in 2011.

Anther related work has been done by Griebel and Uiterwijk in [7]. They created an endgame database of the exact values of 2-dimensional clobber subgames of size 8 and less. Exact games values are calculated using the CGSuite package. They showed a 75% reduction in the number of visited nodes during the search.

According to the mentioned literature review, most of the previous work is done on 2-d Clobber, and there is not enough documentation on the 1-d Clobber. Furthermore, there is no previous work that incorporates a variety of techniques on 1-d Clobber while measuring their effectiveness on runtime and the number of visited nodes separately.

Kristopher De Asis in [4] worked on designing a CGT-based Monte Carlo tree search (MCTS) clobber player which resulted in an acceptable win rate for the CGT player against the baseline MCTS player. J Fernando Hernandez in [1] created a database of left and right stops using sub-zero thermographs of boards up to size 8. This improvement significantly reduced the number of visited nodes.

3. Overview

As a baseline solver for Clobber, we designed an iterative deepening search algorithm utilizing a transposition table with a heuristic score for non-terminal leaf nodes. We decided to consider an acceptable and relatively fast solver as our Non-CGT baseline. CGT techniques are applied in different ways and are not just limited to game values or just game outcomes. The CGT solver starts by first simplifying the board (removing zero games, removing games and their negatives,

etc) which is then split into several subgames. Several steps are taken to take advantage of the CGT techniques for solving the game based on these subgames. Before running the solver, an endgame database is filled with the outcome class of the connected boards, (boards without empty tiles), for up to 16 tiles. Using the outcome classes from the database, the number of the visited nodes dropped significantly. Further improvements are applied by trying some other approaches that are explained briefly in the following paragraph.
In the next step, the database is extended to save the dominated moves for both the black and white players. This information is used in the search tree and has led to a reduced number of options and a smaller tree.
Another approach tried is saving some exact game values in the database. Exact values are saved in the form of the number of ups, downs, and stars.

For the next improvement, game value bounds are computed and stored in the database by playing many difference games. Then the result of the board can sometimes be determined according to the lower and upper bound values. A modified version of the alpha-beta pruning algorithm is used to prune the game tree search space based on the bounds.
Another decent improvement is to replace each game with its smallest equivalent game. To do so, we saved a link to the simplest equivalent entry in the database.

The correctness for all these approaches is tested using our testing pipeline. More details on the algorithms and experiments, implementation and each of the techniques' effectiveness on the solver improvement and results is provided in sections (4), (5), and (6). The project claims to improve the baseline solver significantly for different test cases.

4. Theoretical analysis

4.1 Background

In combinatorial game theory, each game has a value (numbers, hot games, or infinitesimals). If a game has a number as a value, the number indicates how many more moves one player can have compared to its opponent. However, in clobber, for each black move, there is a corresponding white move. Therefore, clobber is an all-small game with infinitesimal values, meaning that the value of each possible clobber board is in the (1, -1) interval. Each game value can be a combination of star(*), up(↑), and down(↓).

4.2 Outcome classes

There are four outcome classes for a game as described in table 1.

Class	Value	Result
N	G||0	Next Player Wins
P	G=0	Previous Player Wins
L	G>0	Left(Black) Wins
R	G<0	Right(White) Wins

Table 1: Some Rules and Game Values.

Our database contains one of these outcomes for each board. The outcome class of a board is determined by playing the game with the Black and White players. Given the outcome classes of the subgames, the following rules are defined and used to indicate the winner of a board according to the current player.

$Consider\ G = G_1 + G_2 + ... + G_n$:

- $If\ G_1,\ G_2,\ ...G_n > 0 \rightarrow all\ games\ are\ L \rightarrow \sum_{i=0}^{n} G_i > 0 \rightarrow the\ game\ is\ a\ L\ wins.$
- $If\ G_1,\ G_2,\ ...G_n < 0 \rightarrow all\ games\ are\ R \rightarrow \sum_{i=0}^{n} G_i < 0 \rightarrow the\ game\ is\ a\ R\ wins.$
- $If\ G_1 = N\ and\ G_2,\ ...G_n = L\ and\ L\ to\ play \rightarrow the\ game\ is\ a\ L\ wins.$
- $If\ G_1 = N\ and\ G_2,\ ...G_n = R\ and\ R\ to\ play \rightarrow the\ game\ is\ a\ R\ wins.$

4.3 Game values

A challenge with all-small games is that game values are infinitesimals and it would be exponential to compute large board values, therefore some precomputed game values are calculated in our database. These values are computed from some verified game patterns that are found in [2] and [5].

For some of these game values, it is necessary to compute the left and right values. Based on the comparison rules shown in Figures 1 and 2, we can compare the left options values and find the maximum of them. For the right options, we do the same and keep the minimum value. Note that

all of these can be done if the values are comparable and we do have some predefined rules for them.
These game values will be saved according to the following predefined rules in Table 2.

$$\uparrow\uparrow\ >\ \uparrow + \uparrow^2\ >\ \uparrow + \uparrow^3\ >\ \uparrow\ >\ \uparrow^2\ >\ \uparrow^3\ >\ 0\ >\ \downarrow^3\ >\ \downarrow^2\ >\ \downarrow\ >\ \downarrow + \downarrow^3\ >\ \downarrow + \downarrow^2\ >\ \downarrow\downarrow.$$

Figure 1: The ordering of the game values compared to zero[1]

$$\uparrow\uparrow *\ >\ \uparrow *\ >\ *\ >\ \downarrow *\ >\ \downarrow\downarrow *.$$

Figure 2: The ordering of the game values compared to star[1]

1. $(BW)^n \mid\mid 0$ $(first\ player\ wins,\ for\ n \neq 3)$ $[up\ to\ n = 19]$	$\boldsymbol{n} \cdot \uparrow\ = \{0 \mid (\boldsymbol{n}-1)\ \cdot \uparrow *\}\quad if\, \boldsymbol{n} \geq 1$ $\boldsymbol{n} \cdot \uparrow * = \begin{cases} \{0 \mid (\boldsymbol{n}-1) \cdot \uparrow\} & if\, \boldsymbol{n} > 1 \\ \{0, * \mid 0\} & if\, \boldsymbol{n} = 1 \end{cases}$
2. $(BBW)^n = \left[\frac{n+1}{2}\right].\uparrow$ $[up\ to\ n = 17]$	
3. $B^m W^n = 0\ ,\ for\ m, n \geq 2$	
4. $WB^n = (n-1).\uparrow\ +\ n.*$	
$\begin{array}{llll} \uparrow & = \{0 \mid *\} & \uparrow * & = \{0, * \mid 0\} \\ \Uparrow & = \{0 \mid \uparrow *\} & \Uparrow * & = \{0 \mid \uparrow\} \\ \text{⤊} & = \{0 \mid \Uparrow *\} & \text{⤊} * & = \{0 \mid \Uparrow\} \\ \text{⟰} & = \{0 \mid \text{⤊} *\} & \text{⟰} * & = \{0 \mid \text{⤊}\} \end{array}$	$n \cdot \downarrow\ = \{(n-1) \cdot \downarrow * \mid 0\}$ $n \cdot \downarrow * = \begin{cases} \{(n-1) \cdot \downarrow \mid 0\} & if\, n > 1 \\ \{0 \mid 0, *\} & if\, n = 1 \end{cases}$

Table 2: Some Rules and Game Values.

4.4 Conjectures

Rules 1 and 2 in Table 2 are two conjectures verified in [2] up to some values of n. This project could successfully push the boundaries of n values in these rules. All the results with detailed runtimes are brought in section 6.2.2.

4.5 Generating dominated moves

For each board, we save the dominated moves in the database for both Left and Right players. Then, while solving our game, after getting the list of possible moves and before playing the game, we remove the dominated moves according to the database and keep the non-dominated moves. We are storing the dominated moves before solving the game and in the database creation stage, so we do not need to solve the dominated games, and that improves performance significantly.

To determine if a move is dominated or not, our solver generates the list of moves for a player at a state and compares all pairs of moves (i, j) and their respective resulting games I, J, and check if a single player wins I - J regardless of who goes first. If this happens, one of moves i and j are marked as dominated. We store dominated moves for all connected boards from sizes 1 through 12.

4.6 Bounds

For each game, a lower and an upper bound is defined which shows that the game's value remains within that range.The algorithm to generate these bounds is as follows:
By playing the difference game between the board and some games that we know their values, we can get a good estimate for the lower and upper bound. To do so, we used the game values in fig 4 which have a known game value. The solver sums up all the subgames lower bounds and upper bounds to have a new bound for the whole game. If the final lower bound is positive or the upper bound is negative, then, the result is a win for one Black or White respectively.

$WWWWWB$	$-5 = 4\downarrow *$
$WWWWB$	$-4 = 3\downarrow$
$WWWB$	$-3 = 2\downarrow *$
WWB	$-2 = \downarrow$
WB	$-1 = *$
	$0 = 0$
BW	$1 = *$
BBW	$2 = \uparrow$
$BBBW$	$3 = 2\uparrow *$
$BBBBW$	$4 = 3\uparrow$
$BBBBBW$	$5 = 4\uparrow *$

Figure 3: Pattern of game values used for computing bounds.

4.7 Substituting games with smaller games

Part of our board simplification process is replacing games with equivalent, but “simpler” games. Database entries have links to simpler games, where games with smaller sums of the number of black and white moves are considered simpler. During board simplification, these simpler subgames are substituted in place of the original subgames. Only games in the database which have bounds computed for them are candidates for substitution, both in terms of being considered for replacement and being the game substituted. This is because it would be impractical to compare all games in the database, so only games having the same upper and lower bounds and outcome classes are compared to each other.

5. Implementation Details

5.1 ID search, TT, and heuristic

This solver uses an iterative deepening approach, with a heuristic score for non-terminal leaf nodes. The heuristic score of a node is the number of moves the opponent with respect to that node has, times -1. Our iterative deepening starts with a maximum depth of 1 and increases by 1 each iteration. We observed that for larger max depths which don’t include a proof or disproof of the root, the solver would needlessly evaluate all of the nodes up to the current max depth, which was very wasteful. To address this, the solver will only visit up to a certain number of leaf nodes at its current max depth, before stopping the search and increasing the max depth. The maximum number of paths allowed to reach the max depth also increases with each increase to max depth, by 50, and after the max depth reaches 30, this limit on the maximum number of leaf nodes is removed, and the search is allowed to complete uninterrupted. The transposition table stores the heuristic value, or actual outcome of a node, if known, as well as the best move for the player at that node, according to the heuristic. The heuristic aims to maximize the current player’s moves at the next state considered. The transposition table has a fixed amount of space, and only one element per hash. States closer to the root node get priority in the table. This lets the search follow a (hopefully) good path, according to the heuristic, from the root. This combination of limited depth and limited paths to leaf nodes allows for quick and increasingly accurate updates to the heuristic value of states. The transposition table is somewhat limited due to storing each tile of the board as a byte, to verify hits/misses for table lookups. We could achieve better performance by packing 4 board tiles into 1 byte, decreasing the size of each table entry and allowing for more entries, but for some unknown reason, doing this packing changes the behaviour of the program resulting in worse performance.

5.2 Endgame database structure

The last updated version of the database saves the following values for each board:

- Outcome (for up to 16 tiles boards)
- Dominated moves for both players 1 and 2. (for up to 12 tiles boards)
- Game value (for up to 16 tiles boards)
- Game bounds (for up to 16 tiles boards)
- Link to the simplest equivalent board in the database (for boards that have bounds computed in the database)

It takes about 53 minutes to make the database.

5.3 Solver Flow (pseudo code)

The board is a string of characters, and a subgame is a contiguous region of the board not containing empty tiles.

When visiting a node, the board is first simplified. The simplification process is as follows: First, each subgame is replaced with its simplest equivalent subgame, if this information is in the database. This is a simple string replacement. Next, subgames that can be determined to be negatives of each other just by looking at the strings, are deleted. Then other subgames that are P positions according to the database are deleted. The remaining subgames are then sorted to produce a board with a “normal form”.

After simplification, bounds for the game are determined by looking at each subgame and summing their lower bounds together, and summing their upper bounds together, to determine the lower and upper bounds respectively of the overall board. If a subgame doesn’t have bound information in the database, the overall bound is defaulted to a very negative lower bound and very positive upper bound.

If the lower bound is positive, or the upper bound is negative, the node is determined to be a win for Black or White respectively.

If the outcome isn’t determined yet, the board is looked up in the transposition table, which may have a stored outcome for this node, in which case this outcome is returned. Otherwise, static rules mentioned in section 4.2 are applied to try to determine the outcome based on outcome classes of the subgames.

Next, if the current player is Black (White), an L (R) position is removed from the board and the search continues recursively on this modified board, starting with the first step as described above, with the same player to play. If Black (White) can win without this L (R) position, then they can win on the unmodified board.

If this fails, the player must make a move on the unmodified board, but only if this is permitted by the iterative deepening parameters (max depth and max number of searches), otherwise a heuristic value is assigned to the node according to the number of the opponent's moves from this node, and this is stored in the transposition table and returned. Dominated moves are removed from consideration, and the remaining moves are ordered and played as follows: the best known move (if stored in the transposition table), is played first, and then all moves that aren't in a subgame whose outcome class agrees with the current player, in random order, are played (Black won't play on an L position at this point), and then all remaining moves, in random order, are played.

A move leading to a win is stored in the transposition table as the best move, otherwise the move with the best heuristic value returned by search is stored as the best move.

For each move played, the resulting board's bounds are passed back from the recursive search, and these bounds are used to do alpha beta pruning, in place of a precise value that would be used by an ordinary alpha beta implementation, as our solver doesn't compute exact values.

If all moves are exhausted, this board is a loss for the current player, and this is stored in the transposition table and returned.

At the root node, the search will not modify the board, so that a winning move can be returned.

6. Results

6.1 Correctness:

We conducted a pipeline for testing our solver based on the reference solver that we were given, which was used in the grading of assignments 1 and 2. We recognized the source of our few wrong results and debugged our code to fix them.

When generating a list of moves, our solver was considering one tile after the end of the board, which was generating one extra invalid move some of the time. The database building program also loaded from file when initializing the database, and it's possible that non null-terminated strings could have been a problem. Our solver was also outputting "0-0" instead of "None" in a few cases.
According to the testing pipeline, right now, our solver is correct and passes all the different test cases.
For correctness, we also tested the winning move to see if that leads to a win of the determined winner. For example, if the result indicates a win for B with the "3-5" move, that move is played and the resultant board is played with the next player W. Its result should again determine B as the winner.

6.1.1 Test cases

We have different groups of test files. Plenty of them are the ones that have been given to us by the instructor. Also, we provided more tests verified by the CGSuit tool. The test cases include W and B consecutive cases, the conjecture cases for already verified lengths, some randomly generated simple and hard boards with different lengths, etc.

6.2 Experiments

6.2.1 Experiment Setting

In the experiments we are mostly comparing some solvers. It is to be noted that the base solver for all of the following solvers is the Basic Solver. In order to compare different improvements and techniques, we have incorporated these ideas into some solvers that have the Basic solver as their core. For instance, Combinatorial 1 is implemented based on the Basic Solver but utilizes CGT techniques. Below there is a summary of the different solvers and the naming convention that we use in our experiments.

- BasicSolver → ID search + heuristics + Transposition Table
- Combinatorial 1 → Solving the board using subgame outcomes saved in the endgame database
- Combinatorial 2 → Applying dominated moves (a major improvement reported in the project milestone) on Combinatorial 1 solver.
- Combinatorial 3 (Final Solver) → Adding game substitution improvement to Combinatorial 2 solver.
- Combinatorial 4 → Having all the improvements in the solver except for game substitution.
- Combinatorial 5 → Game bounds improvement added to Combinatorial 3 solver.
- Combinatorial 6 → Applying alpha-beta pruning on bounds used in Combinatorial 5 solver.

Combinatorial 3 solver happens to be our best solver which includes the dominated moves and game substitution features. To find our best solver, we considered the important features (as in table 3) and tweaked them to see which feature is improving and which is not improving according to different test cases mentioned in section 6.1.1.

	CGT based on outcomes classes only	Dominated moves	Substitution	Bounds	Alpha-beta on bounds

Basic Solver					
Combinatorial 1	✓				
Combinatorial 2	✓	✓			
Combinatorial 3 (Final Best Solver)	✓	✓	✓		
Combinatorial 4	✓	✓		✓	✓
Combinatorial 5	✓	✓	✓	✓	
Combinatorial 6	✓	✓	✓	✓	✓

Table 3:Description and features of different solvers. The best solver is shown in the green row.

For all these settings, we do have some flags in our code in the "options.h" file that can be set easily to enable or disable different improvements.

6.2.2 Experiments on all the solvers

Figure 4 and Figure 5 show the comparison results of different solvers based on runtime and number of visited nodes respectively with respect to the baseline solver and negamax-based full board search solver that uses a transposition table. Boards that these solvers are tested on are generated with consecutive W and B stones. In these figure 4 and figure 5, the progress made using different improvements is illustrated. According to the figures, the final solver has shown significant improvement (based on runtime and number of visited nodes) compared to the baseline and other solvers.

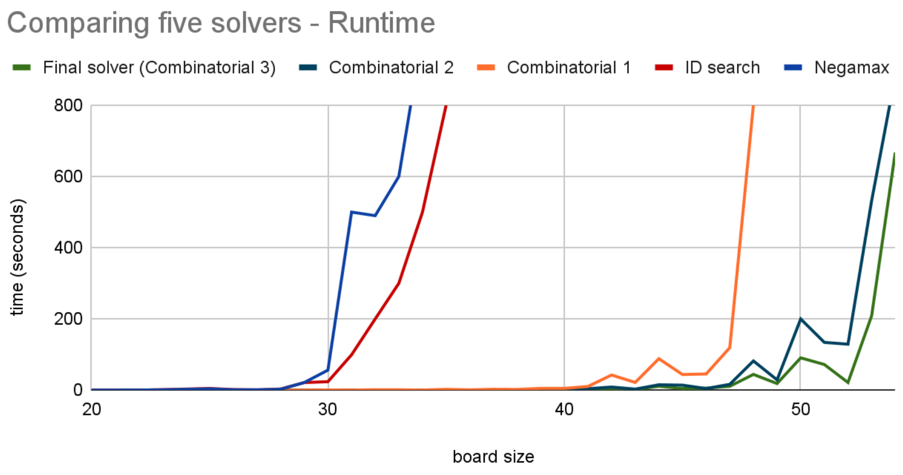

Fig 4: Runtime for worst-case scenario board having W and B consecutive stones.

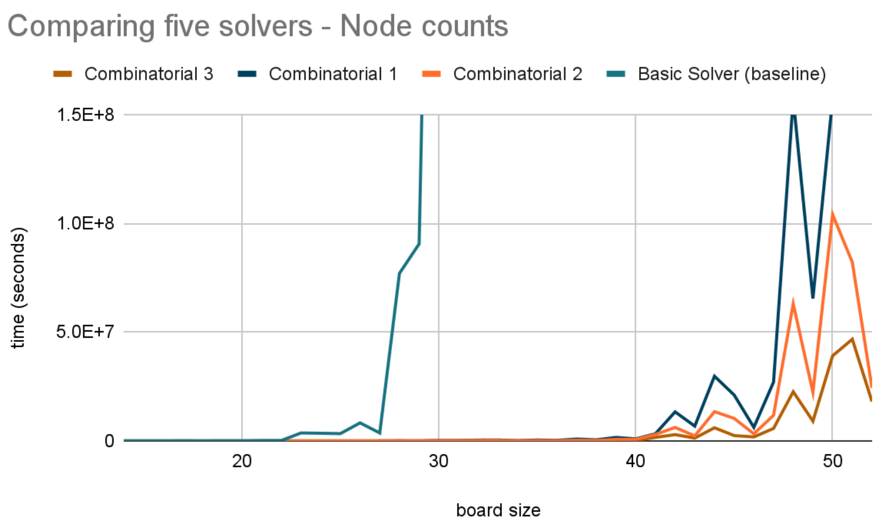

Fig 5: Node count results for worst-case scenario board having W and B consecutive stones.

6.2.3 Experiment on Randomly Generated Batches of boards

The results in table 3 are the average overall runtime of 5 different batches of 500 randomly generated boards ranging from 10 to 52 tiles. According to these results, the final solver had a significant improvement over the baseline (about 100x faster!).

Solver	Runtime (s)
Baseline (basic solver)	421.32
Combinatorial Search (combinatorial 1)	15.93
Dominated Moves (combinatorial 2)	7.18
Final Solver (combinatorial 3)	4.80

Table 3:Description and features of different solvers. The best solver is shown in the green row.

6.2.3 Ablation results in different improvements

Figure 6 illustrates the runtime results of the solvers mentioned in table 3 on consecutive W and B boards.

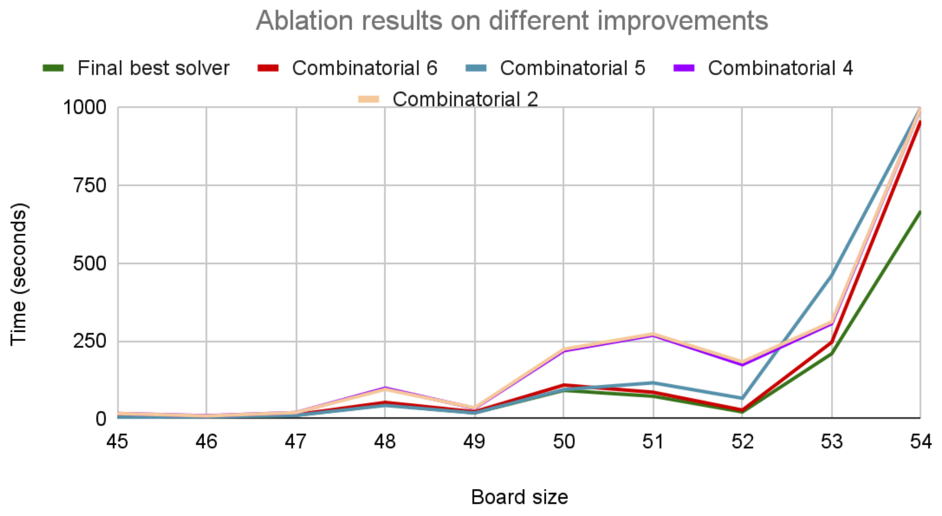

Figure 6: runtime results on consecutive W and B boards for 5 solvers from table 3.

Our best solver as shown in fig 6 is the Combinatorial 3 setting from table 3. Adding the bounds to the setting made it a bit slower (the reason is discussed in section 8) than the best solver but still, its performance is quite good compared to previous solvers. Applying alpha-beta on that often improves the runtime results. As you can see, combinatorial 2 and 4 solvers both do not have the substitution feature, which decreases their performance compared to the other three solvers considerably. We experimentally concluded that the substitution feature and removing dominated moves have the most considerable improvement on the solver's speed.

6.2.3 Conjecture results

1. Pattern 1: $(WB)^n$

We have verified this conjecture for all the n values starting from 1 to 30. This conjecture was previously verified up to n = 19 in [2], we pushed this limit by 11 more values of n. In table 4, the results for the n =27, n=28, n=29, n=30 are reported. For smaller values of n refer to Appendix A.

You can see that the runtime and node counts are reasonable and the outcome agrees to the conjecture hypothesis which shows that the result is the first player win.

n (length/2)	Player	result	Time (second)	Nodes
30	B	B	18952.1	1119579275

29	B	B	963.493	213669906
28	B	B	2363.51	1149041477
	W	W	748.658	366428812
27	B	B	476.425	236760698
	W	W	232.458	116167087

Table 4: Verification results of $(WB)^n$ for n values between 27 and 30.

2. Pattern 2: $(BBW)^n$
We have verified this conjecture for all the n values starting from 1 to 25. This conjecture was previously verified up to n =17 in [2], we pushed this limit by 8 more values of n. In table 5, the results for the n =24 and n=25 are reported. For smaller values of n refer to Appendix A.

n (length/3)	Player	result	Time	Nodes
25	B	B	166.566	76607439
	W	B	618.076	285267668
24	B	B	50.3102	23052361
	W	B	203.946	93229722

Table 5: Verification results of $(BBW)^n$ for n=24 and n=25.

Notably, we could verify them for more n values than what is mentioned, but the runtimes went beyond 700s for each board. Therefore, we stopped the solver before 700 seconds but we expect to get the results of some larger values of n in less than 1000 seconds.

7. Extended Summary

As we have extensively analyzed and explained the working ideas to make the solver stronger, this section is devoted to ideas that were tried but didn’t improve the solver significantly.

Starting with the game values, as in Fig 6 in Appendix, it can be concluded that only a few exact values are found for the boards up to 20 tiles. As a result, no considerable improvement was

made while including game values. The main reason for this could be that the clobber 1-d game values are all infinitesimals and their canonical forms are often exponential and not simple to calculate and use in the solver.
Therefore, patterns and game values are removed in the final solver as their improvement was not considerable.

Also, as mentioned before, we applied alpha-beta pruning on the game bounds, but as you can see in figure 5 in the results section, it did not improve the solver that much. One reasoning behind this is that alpha/beta cuts are interfering with the shallow searches done by the iterative deepening. These shallow searches allow useful information to be saved into the transposition table entries. Therefore, an important parameter to be tweaked is “when to allow cuts”, and “number of searches done at each depth”.

8. Future Works

This project opens several avenues for future research in all-small games. Due to time constraints, we were unable to examine all the ideas. Here is a list of future works that might be promising to try:

1. In the implementation of game bounds, we are only comparing games to boards in the form of BW, BWW, BWWW, BWWWW, …. Our comparing list must be more complete. We don’t have boards with values like $\downarrow *$ or $\uparrow^2$. In this regard, games bounds could be improved by having a wider range of games that cover all the combinations of game values including powers like $\uparrow^2$.
   So, the bound implementation has yet to be improved for the future.
2. We are using Iterative depending as our search algorithm which has shown quite improved results compared to Negamax. Even more improvements could probably be made by replacing ID with MCTS as it offers a better heuristic.
3. Computing local incentives within subgames could possibly improve move ordering.

9. Authorship Statement

This project is implemented by TheSolvers team. Thanks to Martin for providing some previous codes and his own c++ code and test cases. We did not directly use any of the provided codes or their ideas. The main help for us was Martin’s notes and ideas about 1-dimentional clobber in class. Also, we want to thank Owen Randall for sending us his report which guided us through the exact game value idea. Though we haven't used it in our final solver, it gave us some perspective on exact game values.

10. References

[1] J Fernando Hernandez, Using Left and Right Stop Information for Solving Clobber.
[2] Michael Albert et al., Introduction to Clobber.
[3] Claessen. "Combinatorial Game Theory in Clobber." Maastricht University. 2011.
[4] Kristopher De Asis, A CGT-informed clobber player.
[5] Michael Albert, Richard Nowakowski, and David Wolfe. 2016. Lessons in Play: An Introduction to Combinatorial Game Theory, Second Edition (2nd. ed.). A. K. Peters, Ltd., USA.
[6] A.N. Siegel. Combinatorial game suite: A computer algebra system for research in combinatorial game theory, 2003. Available from http://cgsuite.sourceforge.net/.
[7] Griebel, Janis & Uiterwijk, Jos. (2016). Combining Combinatorial Game Theory with an α-β Solver for Clobber. 10.1007/978-3-319-67468-1_6.

11. Appendix A.

Database statistics of game values are shown in Table 6.

*	48	↓	32
0	74	↑↑	2
↑	32	↓↓	2

Table 6: Number of exact game values within the ↑↑ and ↓↓ bound

In Table 7, more conjecture results for Pattern 1: $(WB)^{n}$ are reported. Other n values for pattern $(BBW)^{n}$ are reported in table 8.

n (length/2)	Player	result	Time (second)	Nodes
26	B	B	204.167	104144264
	W	W	79.6249	40213562
25	B	B	84.2962	41650831
	W	W	75.3637	37212672
24	B	B	14.8867	7614681
	W	W	19.1057	9794344
23	B	B	14.8425	7716949
	W	B	10.918	5687858
22	B	B	4.75737	2533320
	W	W	5.81727	3054000
21	B	W	2.47846	1366162
	B	W	2.75556	1445106

Table 7: Verification results of $(WB)^{n}$ for n values in range [21,36].

n (length/3)	Player	result	Time (second)	Nodes
23	B	B	17.1428	8052387
	W	B	72.6614	32766656
22	B	B	9.26085	4570900
	W	B	30.4634	14394300
21	B	B	3.94298	1894953
	W	B	11.542	5776293

20	B	B	2.67103	1349866
	W	B	5.97677	3164854
19	B	B	1.50087	761440
	B	B	3.12909	1606214
18	B	B	0.667979	323733
	B	B	1.48286	781487

Table 8: Verification results of $(BBW)^n$ for n values in range [23,18].