Linear Clobber Solver

Taylor Folkersen
Department of Computing Science, University of Alberta, Canada

Abstract. Linear Clobber is Clobber restricted to a $1 \times n$ board. This paper presents an improved Linear Clobber solver, based on an existing minimax solver.

Keywords. linear clobber, 1D clobber, iterative deepening, minimax

1. Introduction

Clobber is a partizan, 2 player, perfect information game, invented by Albert, Grossman, Nowakowski, and Wolfe in 2001, and has been shown to be NP hard [1]. Clobber is played on a rectangular board containing empty spaces, and black and white stones. For example, the 1×6 game (●○● − ○○) has 2 black stones, 3 white stones, and 1 empty tile. On their turn, a player's only legal moves are those which move one of their own stones to "clobber" an adjacent stone of the opposing player. For example, white could move from the above game to (○ − ● − ○○). A player who cannot move loses, so black would subsequently lose from here.

Positions in Clobber are all-small, meaning their values are infinitesimals. Linear Clobber is Clobber restricted to $1 \times n$ boards, and is the focus of this paper.

Albert et al. conjecture that games of the form $(●○)^n$ (i.e. $(●○)^3 =$ (●○●○●○)) are N positions (first player wins) for $n \neq 3$, and have verified this for n up to 19 [1]. The previous version of my solver has verified this for n up to 30. This paper pushes n to 33. Additionally, Albert et al. conjecture that games of the form $(●●○)^n$ are equal to $\lfloor \frac{n+1}{2} \rfloor . \uparrow$ and have verified this for n up to 17. These conjectures have yet to be proven or disproven, therefore, having a solver capable of evaluating these games for more values of n would aid in that goal.

2. Previous Work

There are some existing Clobber solvers, though most are focused towards solving general Clobber rather than linear Clobber.

Griebel et al. present an alpha-beta-based solver for general Clobber, which uses an endgame database containing exact CGT (combinatorial game theory) values for games [4]. Their search terminates when all subgames have at most 8 stones, by summing together the CGT values of all subgames.

Claessen combines MCTS (Monte Carlo tree search) with a database of endgame positions, to create a general Clobber player [3].

Hernandez uses a database of game value bounds, and left and right stops in a solver for general Clobber, and reports a significant reduction in the number of nodes visited during search [5].

Like the mentioned solvers, my solver uses an endgame database, though unlike Griebel [4], my database does not contain exact CGT values. Like Hernandez [5], my database contains game value bounds, though does not contain stops. Similarly to MCTS used by Claessen [3], I use an heuristic evaluation function for tree nodes that are considered "deep".

There is a lack of linear Clobber solvers, and while there are existing general Clobber solvers capable of solving linear positions, there is still a need for a strong linear Clobber solver more specialized for the linear case.

This project improves an existing linear solver from a previous course project by Taylor Folkersen, Fatemeh Tavokoli, and Zahra Bashir. Much of the codebase has been rewritten with the goals of improving performance through new algorithmic features, in addition to verifiability of correctness, and overall code quality.

3. Database

The solver uses an endgame database, which has been improved from the previous solver, and is able to efficiently represent many more endgame positions than before. The database creation process has also been sped up significantly. This section details the new database.

3.1. Database entries

The previous solver's database stores an entry for every board containing no empty tiles, with lengths 2 through 16 inclusive, for a total of $\sum_{n=2}^{16} 2^n = 131,068$ games represented.

In the new database, every board up to a length of 16 inclusive is represented, including those with empty tiles, for a total of $3^{16} = 43,046,721$ games represented. A naive implementation might store an entry for each game, though by ignoring redundant entries, only 866,924 entries need to be stored in practice. Only approximately 2.01% of all these games need to be stored. This amounts to 39 MB on disk. A database entry is 46 bytes, and has the following information:

- Outcome class (L, R, P, N)
- List of dominated moves for both players
- Lower/upper bounds of game value in terms of the games $\uparrow$, $\downarrow$ and $\star$
- A "complexity" metric which estimates the cost of solving this game
- A link to an equivalent, less complex game in the database (or a self link)
- A "shape" and game number, which can be used to reconstruct the game

Bounds in the database are stored as integers, and examples are shown below.

Board represented by bound	Canonical form	Database representation
(○○○○●)	$3\downarrow$	-4
(○○○●)	$2\downarrow\star$	-3
(○○●)	$\downarrow$	-2
(○●)	$\star$	-1
	zero	0
(●○)	$\star$	1
(●●○)	$\uparrow$	2
(●●●○)	$2\uparrow\star$	3
(●●●●○)	$3\uparrow$	4

3.2. Database access

This section details how to omit redundant database entries, and how to search the database.

I define a *subgame* as a contiguous string of stones. For example, the board (●●○ – ●○ – ○●○) contains 3 subgames: (●●○), (●○), and (○●○).

I denote by the *shape* of a game, a sequence of numbers indicating the length of each subgame. For example, the shape of (●○ – ● – ○○●) is (2, 1, 3). By observing that this game is equivalent to (○○● – ●○), which has the shape (3, 2), we can omit a database entry for the first board, and instead refer to the second board's database entry. In fact, only games with shapes consisting of non increasing lengths, and containing no 1s, need to be stored. I refer to a game whose shape has these properties as *normalized.* A game can be normalized by sorting its subgames in order of non-increasing length, and deleting subgames with only one stone.

The *number* of a game is the number found by treating the black and white stones as binary digits, and ignoring spaces. If white denotes 0, and black denotes 1, then (●○○ – ●●) has the number 10011_2.

To query a game from the database, the game's shape and number are used. The database contains a shape index, which given a normalized shape, provides a pointer to the database section storing entries for that shape. The shape index is searched by first converting the queried shape to a 64 bit integer, by using bit shifts and bitwise OR operations, and then binary searching the shape index. There are 120 shapes in the shape index.

After finding a database section corresponding to a game's shape, the number of the game indicates which entry to go to, relative to the section. For example, (○○●) has number 001_2, so it is the second entry in the section for shape (3). Below are more examples of shapes and numbers.

Game	Shape	Normalized Game	Normalized Shape	Normalized number
(●○ – ●●○)	(2, 3)	(●●○ – ●○)	(3, 2)	11010_2
(●○● – ●)	(3, 1)	(●○●)	(3)	101_2
(○●● – ● – ●●○○)	(3, 1, 4)	(●●○○ – ○●●)	(4, 3)	1100011_2

3.3. Database Generation

Before run time, the database needs to be generated. Database generation uses the solver, and the solver uses the database as it is still being filled in, therefore, the order in which entries are created has a big performance impact. Entries corresponding to smaller boards are considered first, and for a fixed board length, boards containing more subgames are considered first. For example, when generating entries corresponding to boards of length 11, boards with shape (2, 2, 2, 2) are considered before boards with shape (7, 3), and boards with shape (11) are considered last (note the empty tile between each subgame). This way, the result of a search done in the database generation process can often be looked up in the database after one move, if it is not already in the transposition table (which is not reset after solving a game).

Additionally, links to equivalent, less complex games are found and inserted into database entries. The database uses two types of passes over an entry: the type 1 pass fills in all fields mentioned previously in section 3.1, adding a self-link to the entry; a type 2 pass searches for a simpler equivalent game, and if found, a link to its entry replaces the self-link.

The database generation first does a type 1 pass over all games of length 12 or less, and inserts these games into a list, accessed via a map from the 3-tuple (lower bound, upper bound, outcome class) to a list of games matching these values. The games in this map/list structure are the only games which can be linked by other database entries. Next, a type 2 pass is made over games from lengths 1 through 12. For the remaining games of lengths 13 through 16, a type 1 pass is done for all games of a length, and then a type 2 pass is done over all games of that same length.

In doing a type 2 pass, if the complexity metric for the game linked by G_1's database entry (initially a self-link), is larger than the complexity metric of some G_2 found in the aforementioned map/list structure, then the solver checks that $G_1 = G_2$ by solving the difference game $G_1 - G_2$, and if so, G_1's database entry will contain a link to G_2's entry. This is repeated until G_1's entry links to an equivalent game with minimum complexity.

In the previous database, only games with lengths 12 or less have beneficial links or dominated move information. Using the previous solver to generate the previous database takes 31 minutes and 35 seconds. The new database with the new solver takes 38 minutes and 52 seconds to generate, but many more games are represented, and games with lengths greater than 12 have dominance information and beneficial links, and links may generally be of higher quality as they may be links to games not represented previously. Using the new solver and database to generate a database lacking the same information for games longer than 12 takes only takes 3 minutes and 4 seconds. Note that the two versions of the solvers can only use their respective version of the database.

4. Board simplification

4.1. Simplification Algorithm

When visiting a game during the search process, the solver first simplifies the board before doing any other action. The steps of simplification are as follows:

Given a game G to simplify, the algorithm produces a new game G' which is possibly simpler. First, G' is initialized as an empty game. Next, Each subgame of G is looked up in the database. For each subgame, one of three cases may happen:

- A database entry is found, and links to a simpler equivalent game; this linked game replaces the original subgame. Add the linked game to G'.
- A database entry is found, but links to itself; the subgame is set aside to later be combined with other subgames whose combination is searched again.
- No database entry is found because the subgame is too big, and this subgame cannot be replaced. The original subgame is used. Add this subgame to G'.

Each pair of subgames set aside previously are combined by summing, and for each pair-wise sum, one of two cases may happen:

- The combined game has a database entry linking a simpler equivalent game; this linked game replaces the two games. Both subgames are removed from the list of set aside subgames, and pairs containing them are not considered further. Add linked game to G'.
- The combined game has no database entry, OR has a database entry which links to itself; this pair is skipped.

There may be subgames remaining in the set aside list; add these to G'. This substitution process removes subgames known as P positions by the database, but not those which are too big for the database. Therefore, subgames determined to be negatives of each other by inspecting only their string representations (i.e. (●●○) and (○○●)) are removed from G'. Next, sort subgames of G' in order of decreasing length, and decreasing game number in the case of ties (this makes it more likely for games to be found in the transposition table). This concludes the simplification algorithm.

The pair-wise summing of subgames with no beneficial links, to find a beneficial link, is a new feature, enabled by the new database format. The sorting of subgames of G' was tweaked to put larger subgames first.

4.2. Complexity metric

The substitution algorithm presented is improved by the use of a new complexity metric. A game's complexity metric estimates the cost of solving the game using search. In previous versions of the solver, different metrics were tried:

1. The number of moves available to either player
2. The length of the board

3. The number of undominated moves available to either player

Metric 1 yielded the best results. Metric 2 worked, but was empirically worse than metric 1. Metric 3 did not work, as it made search not terminate, because it introduces cycles into the search. To see why, consider the board (○●○○○●●○○●). White's first available move goes to (−○○○○●●○○●), then, black's third available move goes to (−○○○○●●○●−). The first and third games here are equivalent, but the first game contains 2 undominated moves, while the third game contains 4. Neither black nor white plays a dominated move to reach the third board, so the substitution algorithm presented earlier will bring search back to the first board, and search will never terminate.

Metrics based on undominated move counts intuitively seem to be better estimates of the cost of search, as these are the only moves played, so I introduce a new metric for a game: the number of undominated moves immediately available to either player, plus the metric of each immediate child node following only undominated moves. This recursively-defined metric gives the number of undominated moves within the search tree for a game, not accounting for starting player at any node. This metric is strictly decreasing as undominated moves get played, so search will terminate.

5. Search Algorithm

The search algorithm is based on minimax, and uses an iterative deepening approach. Additionally, for each iteration of max depth, only a limited number of leaf nodes are visited, to avoid searching the entire game tree to the current maximum depth. A plain English description of the algorithm is given below.

When visiting a node (G, n) consisting of board G with player n to play, and at depth d in the game tree, do the following:

Compute G', the result of simplifying G as described in section 4.1. Search for G' in the database, and if found, this node is a leaf node, and the result is returned. Search for (G', n) in the transposition table and return its result if both found and SOLVED (not just heuristically evaluated).

Subgames of G' are looked up in the database, and static rules are used to try to solve the node based on subgame outcome classes. The node is solved if all subgame outcome classes are only one of L or R, or there is only one subgame with class N, and other subgames, if they exist, are positive for n (i.e. L for black, R for white). If the node is solved here, write the result into the transposition table and return it. This also constitutes a leaf node.

Next, for each subgame g of G' whose outcome class is positive for n (i.e. L for black), check if n can win on $G' - g$ by recursively applying this search procedure to node $(G' - g, n)$ at the same depth d. If n can win on $G' - g$, then n can win on G', and the node is solved. If the node is solved here, write the result to the transposition table and return. For example, if black is to play on (●●●○ − ●○), here search will find that black can win on (●○), solving the node.

If there are no moves to play, the node is solved. Write the result into the transposition table and return. This is a leaf node.

If d has reached the maximum depth, or the maximum number of leaf nodes has been exhausted, a heuristic value of this node is written to the transposition table, defined as the negative of the opponent's number of moves. This heuristic result is returned.

Dominated moves are then removed using the database for each subgame, and moves are arranged so that they are played in the following order:

1. Best move (if stored in the transposition table)
2. Moves in a subgame whose outcome class is negative for n (black plays on subgame with R outcome, white plays on subgame with L outcome)
3. Any remaining moves

Moves within each of these categories are randomized.

Each move is then played, and its resulting node is visited recursively according to this same procedure, with the opposite player to play and at depth $d + 1$. Here some moves may result in heuristic scores rather than solved outcomes. The node is solved if a winning move is played and the resulting node is solved, or if all moves resulted in losses for n, and the resulting nodes were all solved. If neither of these conditions are met, the node returns a heuristic value defined as the best heuristic value from visited child nodes. The result is stored in the transposition table, whether heuristic or solved, and a best move is stored in the table entry when a winning move is found, or if none is found, the move with the best heuristic score is stored instead.

This is the end of the search algorithm.

Because the solver returns a winning move to the user (if one exists), the above algorithm is only for search below the root node. Search at the root node is similar, but does not simplify the board, or search the database for an immediate result, or search on boards with deleted positive subgames for n; search must play a move at the root node.

In this project, a new static evaluation rule was added, and the iterative deepening parameters were tuned so that the maximum number of leaf nodes doubles with each increment to maximum depth, instead of being raised by a static amount. Using database lookups to bypass search entirely is enabled by the new database format. Move ordering was explored more, by ordering moves according to their outcome classes, but these move orders were empirically worse. I tried making the transposition table store more than 1 best move, but this yielded worse performance, suggesting that the heuristic node value function may be too primitive.

5.1. Transposition Table

The transposition table stores 2^{24} entries, or 2^{23} if the root node's board size is above a certain threshold, to keep the table size down. The previous version of the solver had a naive replacement policy, where a table entry could only be in one spot, and any new entry that collides with an existing one would overwrite it.

The improved transposition table groups entries into blocks of 4. Each block remembers the order in which its elements were accessed, and each element has a unique age from 1 to 4, with 1 indicating the entry as the most recently used within the block. When no entries are free within a block, older entries, and deeper entries, are selected for replacement. The entry with the highest $depth^2 \times age$ is replaced.

I use MurmurHash3 to determine where a game goes in the table [2].

6. Correctness

To be more confident in the correctness of the solver, much of the codebase was rewritten, as previously it was verbose C-like code. Asserts have been added for safety checks, and C++ templates are used instead of macros for computing offsets into database and table entries, making entry formats easier to modify and more safe. When changes are made, the solver is run on a set of 675 test inputs with known values, produced by a third party's solver. Also, the database can be generated again and compared byte for byte to the known correct database. Future efforts to validate correctness could include an additional external program which validates the search tree.

Previously, games longer than 64 tiles were not guaranteed to give the correct result, due to 64 bit types being used to encode some move information. This has been fixed, and the solver should now work with arbitrarily large boards.

7. Performance

Below are running times of $(\bullet\circ)^n$ for various values of n. Times are in seconds. The first time is for black playing first, and the second time is for white playing first. "-" denotes the absence of a time. "New solver time" uses the full database as described in section 3.3, and "New solver time (less DB)" uses a database where games of length 13 or more contain only self links and no dominated move info. Note that this database is still larger than the one used for "previous solver". I also include the average time to solve 150 random games, and the same games are used for each column.

Game	Previous solver time	New solver time	New solver time (less DB)
$(●○)^{20}$	1.23, 1.20	0.16, 0.24	0.19, 0.42
$(●○)^{21}$	2.76, 3.17	0.40, 0.43	0.56, 0.57
$(●○)^{22}$	5.57, 5.56	0.62, 0.57	0.70, 1.15
$(●○)^{23}$	18.68, 11.92	2.17, 1.44	3.34, 1.82
$(●○)^{24}$	51.65, 18.04	1.83, 2.51	1.96, 4.36
$(●○)^{25}$	75.65, 60.15	2.88, 4.88	5.23, 7.43
$(●○)^{26}$	70.03, 274.98	6.21, 7.65	10.04, 16.11
$(●○)^{27}$	220.24, -	9.66, 16.43	13.32, 30.85
$(●○)^{28}$	851.57, -	17.71, 35.09	27.93, 64.65
$(●○)^{29}$	538.72, -	39.81, 73.29	72.94, 123.65
$(●○)^{30}$	-, -	80.68, 152.09	127.49, 751.7
$(●○)^{31}$	-, -	246.71, 523.94	576.68, -
$(●○)^{32}$	-, -	734.92, 3508.51	-, -
$(●○)^{33}$	-, -	5170.84, -	-, -
random game of length 50	5.73 (average over 150)	0.47 (average over 150)	1.13 (average over 150)

The improved solver is much faster than the previous one for large enough boards, sometimes 50 times faster as shown above. An exception is when the board is simple enough that the cost of loading the larger database from disk offsets the benefit of having it. Solving $(●○)^n$ for white first seems to consistently be slower than for black first, possibly because of move ordering.

8. Conclusion and Future Work

While the solver was improved in many ways, there are still many potential improvements to look into. Alpha beta pruning using the bounds of subgames was tried in a previous version of the solver, but did not work well, possibly because of the lack of granularity in the bounds used. Hernandez uses more detailed bounds, which consider not just sums of $\uparrow$, $\downarrow$, and $\star$, but also exponents of these [5]. Perhaps better bounds would help. Griebel et al. sum together exact values of subgames, if known, to end search early, which could be an improvement for my solver on large boards with many small subgames [4]. Better move ordering, heuristic node valuation and transposition table policies are likely possible.

This improved solver can hopefully be used in the future to help prove or disprove the conjectures about $(●○)^n$ and $(●●○)^n$ by quickly generating lists of winning moves.

References

[1] Michael H Albert, JP Grossman, Richard J Nowakowski, and David Wolfe. An introduction to clobber. *INTEGERS: The Electronic Journal of Combinatorial Number Theory*, 5(2), 2005.

[2] Austin Appleby. Murmurhash3.cpp. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp. Accessed 2022-11-23.

[3] J Claessen. *Combinatorial game theory in Clobber*. PhD thesis, Master's thesis, Maastricht University, 2011.

[4] Janis Griebel and JWHM Uiterwijk. Combining combinatorial game theory with an α-β solver for clobber. In *BNAIC*, pages 48–55, 2016.

[5] J Fernando Hernandez. Using left and right stop information for solving clobber.