MCGS: A Minimax-Based Combinatorial Game Solver

Taylor Folkersen(✉)[0009−0004−8147−903X], Haoyu Du[0009−0006−7710−8513], and Martin Müller[0000−0002−5639−5318]

University of Alberta, Edmonton, Canada
{folkerse,du2,mmueller}@ualberta.ca

Abstract. In combinatorial games, the most fundamental question is: who wins? We develop a Minimax-based Combinatorial Game Solver (MCGS), which can efficiently answer this question for "short" game positions. MCGS is specialised for solving positions that consist of a sum of independent subgames. Given a first player, it can find the winner of a sum of games by search. The algorithms and data structures in MCGS take advantage of subgame structure. In contrast to previous approaches, MCGS avoids computing the canonical forms of combinatorial game theory, which can very quickly become a major bottleneck for answering win/loss questions. Search improvements based on minimax search techniques and on principles of combinatorial games greatly increase the efficiency of MCGS.
After reviewing the background and some motivational examples, we introduce the methods used in MCGS, and show first computational results for popular combinatorial games such as Clobber and NoGo. MCGS strives for a balance, providing a game-independent general framework while supporting game-specific optimisations.

Keywords: Combinatorial games · Minimax solver · Sum game

1 Introduction

Many of the most popular board games, such as chess, Go, and checkers, are two player, perfect information games. Combinatorial Game Theory (CGT) is a mathematical theory for describing and solving such games. The area was greatly developed through the books *On Numbers and Games* [4] and *Winning Ways* [2]. In CGT, games are studied as abstract mathematical structures, often with the *normal play* convention that the last player able to make a move wins.

A key concept of CGT are *sums of games*, short *sumgames*, that consist of two or more independent *subgames*. The idea is to play many games together, and make a move in exactly one of them. A well-known example are Go endgames [3], which often break down into a sum of independent local endgame areas. Using the theory, games with sum structure can often be solved very efficiently - much faster than with traditional full-board minimax search algorithms [10].

We describe MCGS, a Minimax-based Combinatorial Game Solver. Our goal is to develop and implement general algorithms to solve sumgame positions,

taking advantage of sum structure and other algorithmic benefits of CGT to make the search efficient.

In Section 2 we review the background for our work - the underlying concepts of CGT and minimax search, and closely related previous work. We assume that the reader has a basic familiarity with Combinatorial Game Theory concepts, such as those in the first chapter of *Winning Ways* [2]. Section 3 introduces the MCGS program, its minimax search engine, and its support for solving sumgames. This is followed by a discussion of some technical challenges, and how they were addressed in MCGS. Section 4 shows experimental results and evaluates the contributions from different solver components. Section 5 concludes with a discussion of the current limitations of MCGS, and future work to address them.

2 Basic Concepts of Combinatorial Games

We focus on so-called *short* (finite and loop-free) combinatorial games [12]. A short game G between two players Left and Right is defined recursively in terms of its Left and Right options, $G = \{G^{\mathcal{L}}|G^{\mathcal{R}}\}$, where $G^{\mathcal{L}}$ and $G^{\mathcal{R}}$ are sets of games. The game $G = \{|\}$ where neither player has a move is called 0.

In the *sum* of two independent *subgames* G and H, a player can choose to move to an option in either G or H, while leaving the other game unchanged: $G + H = \{G + H^{\mathcal{L}}, G^{\mathcal{L}} + H \,|\, G + H^{\mathcal{R}}, G^{\mathcal{R}} + H\}$.

The *inverse* $-G$ of a game G is obtained by switching the roles of Left and Right throughout, recursively: $-G = \{-G^{\mathcal{R}}| - G^{\mathcal{L}}\}$. The *difference game* $G - H$ is defined as $G + (-H)$.

In CGT, games are studied independently of which player goes first in any position. In contrast, the main search of MCGS solves a game G under alternating play, for a given player (Left or Right) to move first.

Winners, Outcomes and Equality of Games When playing a short game G, either player can play first, and with best play by both, the first player either wins or loses. Games are partitioned into four *outcome classes* accordingly, which can also be characterised by the relation of games G to 0 [12]:

- $\mathcal{L}$: **L**eft wins G both going first and going second, $G > 0$
- $\mathcal{R}$: **R**ight wins G in both cases, $G < 0$
- $\mathcal{N}$: The **n**ext (first) player wins both times, $G \gtrless 0$ (G is incomparable to 0)
- $\mathcal{P}$: The **p**revious (second) player wins both times, $G = 0$

Outcomes are used to define *equality* of games: $G = H$ iff outcome$(G + X)$ = outcome$(H + X)$ for all short games X. Also, $G = H$ iff $G - H = 0$, which provides a search-based way to test equality.

2.1 Minimax Search for Combinatorial Games

Two boolean minimax searches, one for each player going first, determine the outcome class of G. A single search can restrict a game to two outcome classes and provide a bound on the value of G, which is sufficient in some applications: If Right goes first and Left wins, then $G \geq 0$ ($G > 0$ or $G = 0$). Similarly, if Left goes first and Right wins, then $G \leq 0$ ($G < 0$ or $G = 0$).

Besides computing the outcome class, minimax search can also answer other yes/no questions about a game. For example, $G \geq H$ holds for two games G and H iff $G - H \geq 0$, which is true iff Left wins $G - H$ with Right going first. This enables pruning of dominated moves by local search: If a (sub)game has two Left options L_1 and L_2, and $L_1 \geq L_2$ is proven by a search, then the dominated option L_2 can be pruned.

2.2 Algorithms for Solving Short Games

Much work has been done on solving specific games by many variants of minimax search, typically with game-specific enhancements. In contrast, we propose a general-purpose application to solve sumgames by search. MCGS focuses on such general algorithms for all short games. More efficient approaches do exist for specific game classes such as *impartial* games, where both players always have the same options. The current MCGS does provide basic support for computing the nim value of impartial games.

Siegel's CGSuite [11] is a well-known general-purpose combinatorial games package, and provides a large number of tools for working with many types of games in many settings. For short games, CGSuite is built around computing the canonical form.

2.3 The Canonical Form of Games

The *canonical form* $C(G)$ of a short game G is a fundamental concept in CGT. $C(G)$ contains exactly enough information to determine all the outcomes of $G + X$ for any short game X. A canonical form can be obtained by repeatedly removing dominated options and reversing reversible moves [12]. For complex games G, the canonical form must often contain vast amounts of information, which causes a massive computational bottleneck in a win/loss solver.

As an example, for the empty 1×16 NoGo board, CGSuite takes almost 8 minutes on a state of the art machine to compute the massive canonical form with 1201194 *stops*, a measure for the size of the result. The 1×17 NoGo computation did not complete within 2 hours and 30 minutes. In contrast, the minimax search of MCGS can solve the win/loss outcomes of such games quickly, in a combined total of 1.6 seconds for both players on the 1×16 board, and in 5.3 seconds on the 1×17 board. This is a from-scratch computation, with both databases and transposition tables switched off. Similar observations for multiple games were among the main motivations for developing MCGS.

2.4 Predecessors of MCGS - Solvers for Specific Games

One aim of MCGS is to generalise and apply the lessons learned from building several successful game-specific solvers. For linear Clobber (Clobber played on a one-dimensional strip), Folkersen et al. developed an efficient solver that far advanced the state of the art [7,8,14]. Their SEGClobber solver uses many techniques inspired by CGT. Recent work on linear NoGo by Du et al. [5,6] found the outcome class for all empty boards up to 1×39. For the larger boards, CGT-based techniques [5] reduce the search effort by about two orders of magnitude over an optimised traditional full-board search [6].

3 Components of MCGS

MCGS version 1.3 is a game-independent search framework for solving sums of short games, with extra support for games played on one-dimensional strips or two-dimensional grids. MCGS 1.3 implements the games of Clobber, NoGo, Kayles, and Elephants and Rhinos, as well as the *basic* CGT games integer, dyadic rational, nimber, up-star, and switch. The minimax search engine computes the win/loss result of short games. An additional search engine specialised for impartial games determines their nim values. A wrapper can create an impartial version of any partizan game implemented in MCGS. Search enhancements include game-independent hashing and transposition tables for both single games and sumgames, and a database generator for creating tables of outcome classes and other game properties. MCGS features extensive documentation, a test framework with thousands of tests, and a simple extendible file format to describe game positions and expected search results. The open source program[1] is freely available under an MIT license.

3.1 Search Engine

The goal of MCGS is to efficiently answer the question: Who wins game G, if player p goes first? The main search engine of MCGS is based on traditional minimax search, with extensive support for solving sumgames. The monolithic state of traditional board games such as chess, checkers and Go typically consists of a single game board. In contrast, a combinatorial game is represented as a *sumgame*, a set of independent subgames. Search of a sumgame can take advantage of this split into subgames, which opens many opportunities for simplification and optimisation, resulting in cumulative efficiency gains that often yield a result orders of magnitude faster than is possible with full-board search.

Hashing and Transposition Table The search engine supports a transposition table based on a two-part hashing scheme. A *local hash code* encodes a single subgame, and is similar to a traditional hash code for a monolithic game state.

[1] MCGS can be downloaded at: https://github.com/ualberta-mueller-group/MCGS.

The main difference is support for differentiating between games that might have the same board representation but are different games, such as Clobber and NoGo.

In contrast, a *global hash code* encodes a whole sumgame, which consists of a set of subgames. The main challenge is to recognise when two sumgames consist of the same set of subgames. Our solution is fully general and supports different game types within the same sum, such as a mixed sum containing NoGo and Clobber positions, integers, and switches.

Implementation of Local Hashing Local hash codes for each subgame are computed using *Zobrist hashing* [15]. A *random table* contains one fixed random 64 bit integer for each possible *(location, value)* pair, where *value* is the state of a *location* on a game board. The local hash of a subgame G is the XOR of the random table values for all (location, value) pairs in G, and a code for the game type such as Clobber or NoGo, from a separate random table. In order to increase the number of hash hits, each game can define a game-specific *normalisation* operator to convert equivalent representations of a game to one common representation. For example, normalisation can mirror or rotate the board, compress larger *blocks* of stones into single stones in linear NoGo, and remove redundant empty spaces in linear Clobber.

Most games use only a small set of possible values to index into the random tables. For example, the values of Clobber and NoGo locations are empty, black, and white. MCGS random tables contain random numbers for all 256 distinct 8 bit values. These are further extended to all 128 bit values by splitting them into 8 bit chunks, then computing and XORing bit-rotated codes for these chunks.

Sumgame Normalisation and Global Hash Since the same random table is used for the local hashes of all subgames, global hashing for a sumgame cannot simply XOR ($\oplus$) the local hashes together: this would cause hash collisions, as for example given a sum $S = G + G$ and a local hash function $h(G)$, the codes $h(G) \oplus h(G) = 0$ would cancel. To compute a global hash of a sumgame, we use a two step process. First, the (normalised) subgames are sorted according to their game types and lexicographic order, $S = G_0 + \ldots + G_n$. This defines a normal form for the sum. Next, a global hash code is computed from the corresponding list $[h(G_0), \ldots, h(G_n)]$ of local hashes. This process uses a separate global hash random table for the following (location, value) pairs as follows: For each subgame G_i, its (location, value) pair is $(i, h(G_i))$, with the index in the sorted list as location, and the 64 bit local hash as value. The global hash code for S is computed from an XOR of these codes, and an encoding of the current player. This two-level re-coding scheme avoids hash collisions even in cases when a sumgame consists of thousands of copies of the same subgame, or when the representations of subgames differ only by game type.

Precomputed Subgame Databases Databases augment the search engine by providing pre-computed information about subgames. This information can lead to sumgame simplification and earlier search termination.

A *database game generator* can be implemented for a game type. It provides a way to iterate over subgames in order of increasing size, up to some configurable maximum. The database generation algorithm uses these game generators to find and store outcome classes of games. Considering games in increasing size ensures that all smaller subgames already have database entries, which greatly increases the speed of finding outcome classes. The database is queried by the pair (game type, local hash).

This solution is efficient while still remaining simple and general, as adding database support for a game type only requires implementing a database game generator. A basic generator for strip and grid games is available.

Local and Global Move Generators Every MCGS game type defines a local *move generator*, which incrementally produces the valid moves for a given player in the subgame. A global move generator is used to generate moves in a sumgame. The global move generator has two optimisations. First, if a sum contains two or more identical subgames, then it skips moves on all but one of the copies. Second, it generates the negative incentive moves from rationals and integers last, after all other moves have been exhausted.

3.2 Game-specific Functions in MCGS

MCGS provides an abstract `game` class, and a game-independent `sumgame` class. A new game is implemented in MCGS as a subclass of `game` with several game-specific methods. The abstract subclasses `strip` and `grid` simplify the implementation for games on 1-dimensional (linear) and 2-dimensional (rectangular) boards. Mandatory game-specific methods are `play`, `undo_move`, a move generator, hashing support, a simple `print` function, and computing the `inverse` by swapping the players' roles.

Solving efficiency for new games can be increased by adding optional methods to `split` a game into subgames after a move, and to `normalize` a subgame. In addition to increasing the number of transposition table hits, this allows building more compact databases which only include normalised single subgames, which are the only subgames accessed by the search.

Another optional efficiency feature is to update local hashes incrementally in functions which change the game's state: `play`, `undo_move`, `normalize`, and `undo_normalize`.

3.3 Tracking Changes During Search

MCGS uses stacks to track changes during search in both `game` and `sumgame`, and to implement efficient `play` and `undo_move` functions. Data stored includes the move history, and details of each split into subgames and simplification step. This enables fast undo of changes, to restore the previous state. A `sumgame` keeps a list of current and previous subgames. A flag indicates whether a subgame is currently active. When a subgame is split into new games after a move, the old

subgame is deactivated and the new, active subgames (if any) are appended. At `undo_move`, these changes are rolled back.

3.4 Sumgame Simplification and Optimisation

Simplification steps occur both on the game (Section 3.2) and sumgame level, both with and without a database. When a move is played, any resulting subgames are `normalized`, including those resulting from a `split`. A simplification pass combines basic game types: switches $\{a|b\}$ are converted to rationals (and possibly stars) when $a \leq b$, and are otherwise normalised to be of the form $m + \{c| - c\}$, by extracting the mean m. Integers and rationals are summed to a single number. Multiples of ups are also merged, pairs of stars cancel out, and nimbers are combined using nim addition.

The database enables more powerful optimisations. Subgames with outcome class $\mathcal{P}$ are deactivated. Knowing the outcome classes of all remaining subgames allows the early termination of search when all outcome classes are wins for the same player, and when exactly one outcome class is $\mathcal{N}$, and all others are wins for the current player.

4 Test Sets and Experiments

We describe the test sets used in our experiments, and give scaling results on linear Clobber, linear NoGo, and Elephants and Rhinos. These tests include ablation studies on the effectiveness of transposition tables, subgame splitting and normalisation, and databases. Further experiments study scaling by number of subgames, and Clobber on $2 \times n$ grids.

4.1 Test Sets

We created a total of five test sets: the sets `clobber-linear`, `nogo-linear` and `elephants`, test scaling on random linear Clobber, linear NoGo, and Elephants and Rhinos boards. Sets are further grouped into buckets, by number of moves for the first player in linear Clobber and linear NoGo, and by total number of stones in Elephants and Rhinos. `elephants` board lengths are scaled by number of stones. The set `clobber-subgame` contains random boards with 24 stones, pre-split into subgames by inserting 0 to 10 empty spaces in random locations. This set is grouped by number of subgames. Finally, `clobber-2xn` contains random Clobber positions on $2 \times n$ grids, grouped by n.

Games are generated with parameters such as board size, random first player, number of empty spaces, and distribution of stones. Each bucket contains at most 2000 games. Generally, test set instances are selected to be interesting, yet small enough so most can run to completion even with the weakest solver version in our ablation studies.

The full setup of experiments including test sets is in the Github repository [2].

4.2 Setup of Experiments

Experiments are run on a server with Intel Xeon E5-2665 0 @ 2.40GHz CPUs. The transposition table has 2^{28} entries, taking about 1.3 GB. 26 independent tests are run concurrently, with a 60 second timeout. For repeatability, we measure *node counts* rather than process time, as memory bandwidth is likely a bottleneck when running so many tests concurrently. A *node* is one invocation of the recursive minimax search function.

The database used in some of the experiments contains only normalised games which cannot be split into more subgames. It includes linear Clobber, linear NoGo, and Elephants and Rhinos games of lengths up to (and including) 15. It also includes Clobber games that fit within a 2×5 rectangle. For move generators, pruning of moves in duplicate subgames is disabled for all tests.

4.3 Results

Figure 1 shows the scaling performance of MCGS for Linear Clobber, Linear NoGo, and Elephants and Rhinos. The data is split into buckets along the x-axis, and each different ablation is drawn with a horizontal offset in order to improve visibility. The first ablation, shown as the leftmost data point in each bucket, disables all three optimisations: transposition table (TT), subgame splitting and normalisation (SN), and database (DB). The second ablation enables only the transposition table, and the third additionally enables splitting and normalisation. The fourth result in each group shows full MCGS, with all three optimisations enabled. The y-axis shows the natural logarithm of node counts. The dots are means of the log-transformed data, and the bars show ± 1 standard deviation.

Figure 2 on the top shows the effect of increasing the number of initial subgames in the `clobber-subgame` set. The diagram on the bottom shows scaling results for `clobber-2xn`, with an increasing number of columns n on the x-axis.

The transposition table and database both seem to generally increase performance by orders of magnitude. Splitting and normalisation appears to have varying efficacy for each game, which may be partially explained by differing rates of positions which can be split or normalised during search. Examining similar data with a different random seed and bucket sizes of 100, this rate is 100% for `clobber-linear`, 85% for `nogo-linear`, and 65% for `elephants`.

Interestingly, in the `clobber-subgame` data, the benefit of the database seems to diminish as the number of subgames increases. This is likely due to the limitations in the current database, which only stores outcome classes, and does

[2] MCGS versions 1.4 and later contain an `experiments` directory containing test sets, results, and instructions for running the experiments. The MCGS version used for the experiments in this paper was https://github.com/ualberta-mueller-group/MCGS/commit/4113a903b24744b901173e5d18f06484887415ba.

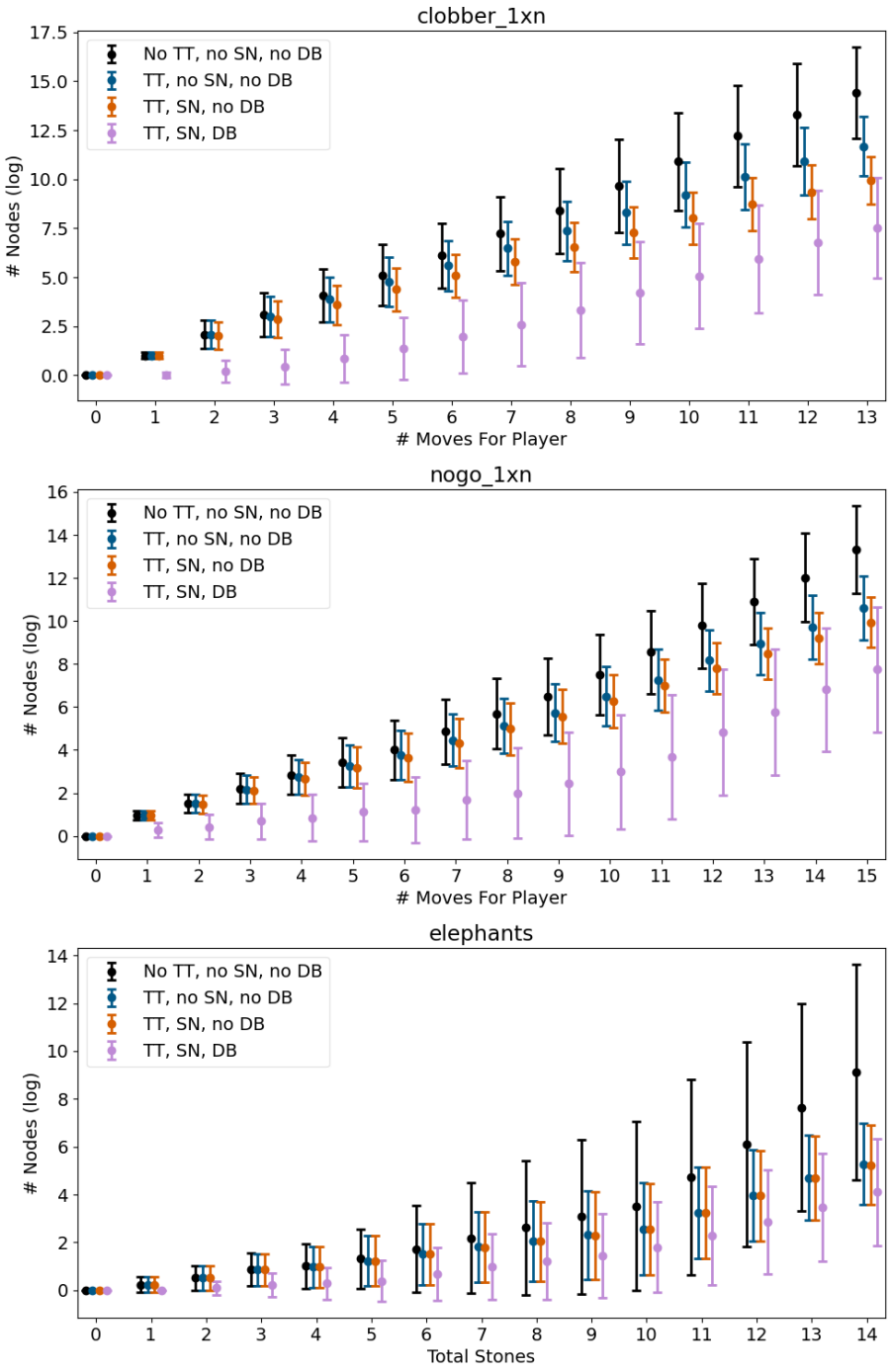

Fig. 1. Scaling of MCGS with the number of moves or stones, on random games of linear Clobber, linear NoGo, and Elephants and Rhinos.

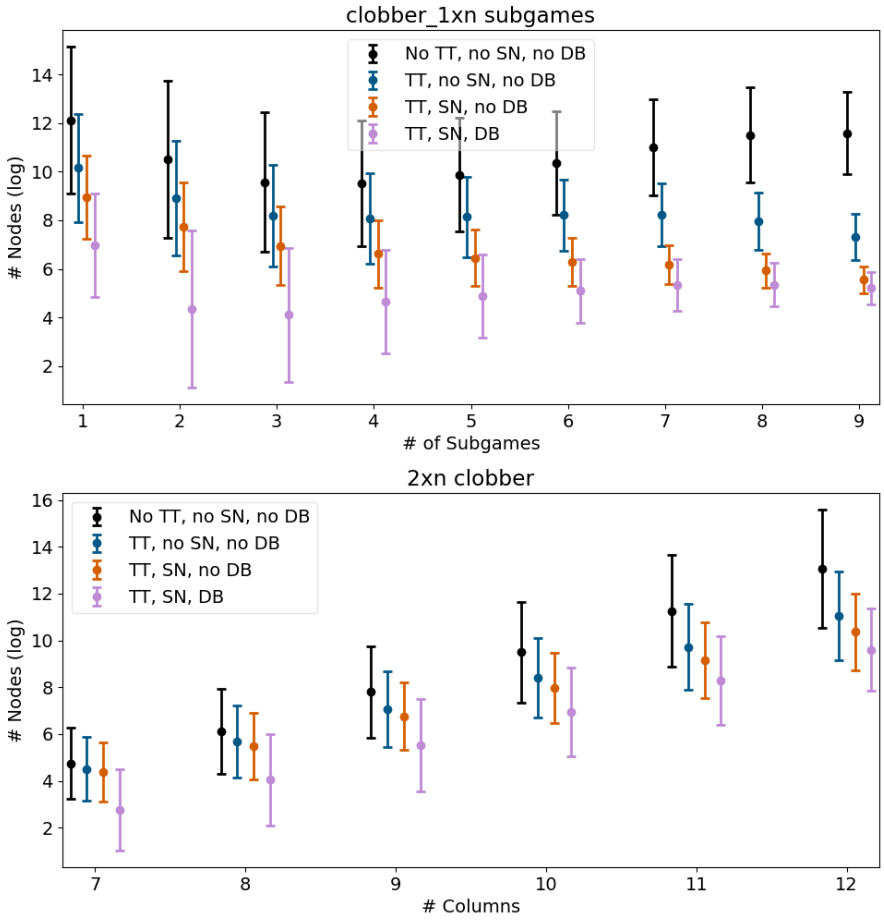

Fig. 2. Top: Increasing the number of subgames in linear Clobber. Bottom: Clobber on a $2 \times n$ grid, with n columns.

not use this information for move ordering. For example, if a position consists of subgames with known outcome classes which don't determine the result, such as several subgames with outcome $\mathcal{N}$, it may take many moves to reach a position where one of the outcome class rules of Section 3.4 can be used to immediately solve the position. As the number of subgames increases, it becomes increasingly unlikely that one of these rules can be used.

Many test cases timed out for the weakest solver with all three optimisations disabled: 266 `clobber-linear` games with 13 moves, 10 `nogo-linear` games with 15 moves, 395 `elephants` games with 14 stones, 1 `clobber-subgame` game with 2 subgames, and 48 `clobber-2xn` games with 12 columns.

To relate node counts to time, we run a small linear Clobber data set generated with the same parameters single-threaded. The average speeds for the four versions tested, using increasingly more components, were 4,189,266 N/s (nodes per second), 1,719,864 N/s, 727,696 N/s, and 510,238 N/s for full MCGS. While the speed per node decreases, the node counts are reduced, often by orders of magnitude.

5 Limitations, Conclusions and Future Work

5.1 Limitations of Current MCGS

The design of MCGS is specialised for win/loss computations for short games under normal play. This is in contrast to game-specific solutions on one side, and to the many types of games and computations supported by CGSuite on the other side. For example, MCGS cannot solve *loopy* games such as *ko* situations in Go. Compared to existing specialised solvers such as [5,7], there is some overhead due to the generality of MCGS. However, it is much easier to add a new game to MCGS rather than build a full-featured solver from scratch. A detailed analysis of the tradeoffs involved is left for future work.

5.2 Conclusions and Future Work

MCGS fills a gap between search engines for monolithic game states and CGT packages based on canonical form such as CGSuite. Our experiments show the usefulness of our generic optimisations that come pre-implemented in our framework, including hashing/transposition tables, subgame structure, and databases. We envision many more improvements for Future versions of MCGS:

- Databases containing sumgames (also see Section 3.1).
- Support for games given in text representation, such as {3|{2|1}}.
- *Thermographs* for computing temperatures for move ordering, and as bounds on subgame values [13].
- More efficient algorithms for impartial games [1,9]
- Many more games, such as Amazons and Domineering
- Game-independent and -dependent move ordering heuristics
- Improved pruning during search, as in specialised solvers.

- An extended database for pruning dominated moves, and substituting groups of subgames with equal but simpler groups of subgames [8]
- Incentive-based pruning as in Go endgames [10]
- Parallel search for solving large games

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. Beling, P., Rogalski, M.: On pruning search trees of impartial games. Artificial Intelligence **283**, article 103262 (2020)
2. Berlekamp, E., Conway, J., Guy, R.: Winning Ways. Academic Press, London (1982)
3. Berlekamp, E., Wolfe, D.: Mathematical Go: Chilling Gets the Last Point. A K Peters (1994)
4. Conway, J.: On Numbers and Games. Academic Press (1976)
5. Du, H., Müller, M.: Solving linear NoGo with combinatorial game theory. In: Computers and Games (CG 2024). LNCS, vol. 15550, pp. 54–65. Springer (2025)
6. Du, H., Wei, T., Müller, M.: Solving NoGo on small rectangular boards. In: Advances in Computer Games. LNCS, vol. 14528, pp. 39–49. Springer (2023)
7. Folkersen, T.: Linear Clobber solver (2022), Capstone report, University of Alberta
8. Folkersen, T., Bashir, Z., Tavakoli, F., Müller, M.: SEGClobber - a linear Clobber solver (2025), accepted for ACG 2025
9. Lemoine, J., Viennot, S.: Nimbers are inevitable. Theoretical Computer Science **462**, 70–79 (2012)
10. Müller, M.: Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames. In: IJCAI. pp. 578–583 (1999)
11. Siegel, A.: CGSuite. a computer algebra system for research in combinatorial game theory (2003–2025), https://www.cgsuite.org
12. Siegel, A.: Combinatorial Game Theory. American Mathematical Society (2013)
13. Song, J., Müller, M.: An enhanced solver for the game of Amazons. IEEE Transactions on Computational Intelligence and AI in Games **7**(1), 16–27 (2015)
14. Tavakoli, F., Folkersen, T., Bashir, Z.: Strong 1-dimensional Clobber (2022), CMPUT 655 project report, University of Alberta
15. Zobrist, A.: A new hashing method with application for game playing. Tech. Rep. 88, Univ. of Wisconsin (1970)