Solving Linear NoGo with Combinatorial Game Theory

Haoyu Du[0009-0006-7710-8513] and Martin Müller[0000-0002-5639-5318]

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

Abstract. NoGo is a version of Go where stones are never removed from the board, once played. Strong computer players have been created for NoGo. However, the game properties and optimal play strategies are not well studied. We introduce CGTSolver, a search algorithm that applies concepts from combinatorial game theory (CGT) in order to solve Linear NoGo. We develop several decomposition strategies and simplification rules for this game. Our results show that CGTSolver is much more efficient than previous solvers, and as the board size increases, the performance gap widens. With this new approach we solved all NoGo boards up to 1×39—twelve boards more than in previous work.

Keywords: NoGo · Combinatorial Games · Minimax Search

1 Introduction

NoGo, or *Anti-Atari Go* [1], is a variant of the well-known game of Go. NoGo shares the basic mechanisms and concepts of Go, but with different rules of play:

- Capturing and suicide are both forbidden.
- Passing is forbidden.
- When a player cannot make a move, the game ends and the player loses.

With just a simple twist to the rules, NoGo becomes a very different game strategically. Linear NoGo or $1 \times n$ NoGo[1] refers to NoGo games played on a one-dimensional board, a $1 \times n$ strip consisting of n *points* which are initially empty. In Section 2, we review NoGo rules and previous work on this game. In Section 3, we briefly introduce our notation and the relevant combinatorial game theory (CGT) concepts. Starting in Section 4, we analyze and solve Linear NoGo by search. We represent a board as a sum of independent subgames in the sense of CGT [15]. We develop rules for board simplification, decomposition, and reduction. These rules allow us to replace a position by simpler sum games with equal outcome. In Section 5, we discuss the structure of reduced Linear NoGo positions, and construct a database of all such games with up to 15 empty points. We design a static evaluation function based on sum game properties in Section 6, and in Section 7 we implement the new NoGo solver *CGTSolver* based

[1] We use the terms Linear NoGo and $1 \times n$ NoGo interchangeably

on Negamax search with many CGT-related improvements. In the experiments in Section 8, we analyze the performance of CGTSolver and solve boards up to 1×39, greatly surpassing the previous state of the art.

2 The NoGo Game and Previous Work

NoGo is a relatively young game with few human players. It is a two-player perfect information game played on a graph, usually a grid. We study Linear NoGo on a one-dimensional strip. Each point on the board except the two endpoints has two neighbors. Following standard Go notation, a maximal set of adjacent stones of the same color is called a *block*. In Linear NoGo, each block has either one or two adjacent empty points, which are called its *liberties*. In contrast to Go, it is illegal to *capture*—to remove the last liberty of an opponent's block. It is also illegal to play a *suicide* move, which removes the last liberty of a player's own block.

NoGo is less studied than popular board games, and few useful heuristics are known for playing it. Previous research mainly focused on creating strong computer agents for game playing [3,6,13]. She [13] solved NoGo on the 5×5 board; Cazenave [2] provided an incomplete table of winners for boards up to 25 points. The most complete computer-proved NoGo results so far, by Du et al., include results for all starting moves on all rectangular boards up to 27 points [4]. They proved by a symmetry argument that the first player, Black, wins all $1 \times n$ NoGo games for odd $n > 1$.

Previous studies that combine search algorithms and CGT to solve games other than NoGo include [9,8,16,17]. Shan [12] used CGT methods to calculate the mean and temperature of many NoGo positions, and proved fundamental theorems about board partitioning into independent subgames.

The best previous computer program to solve NoGo is SBHSolver [4], a Negamax-based solver using a transposition table implemented with Sorted Bucket Hash (SBH), a memory-efficient perfect hashing scheme. SBHSolver also uses the History Heuristic [11] and Enhanced Transposition Cutoff [10]. This program does not use reduced NoGo positions or any CGT concepts. We use it as a baseline for evaluating CGTSolver.

3 Notation and CGT Concepts

We write NoGo positions, and templates for them, as a string with the colors of each point. We use the characters x, o, . to indicate Black, White, and empty points respectively. x$^+$ and o$^+$ indicate a string of one or more stones of the same color. <L> and <R> denote arbitrary substrings, of size 0 or longer, of a legal NoGo position, while <L>$^+$ and <R>$^+$ denote substrings of size at least 1. Strings written next to each other are concatenated. For example, the template <L>x.o.x<R> matches positions such as x.o.x, x.o.x., and o.xx.o.x..o, but not the illegal ox.o.x, where the leftmost o has no liberty.

3.1 Notation for Move Restrictions

We use the following notation from [12] to express move restrictions on single points due to NoGo rules. A point on a NoGo board is called a *0-Go* if no one can play there, a *1-Go* if only one player can play, and a *2-Go* if both players can play. The set of 1-Go points can be partitioned into disjoint sets *B-Go* and *W-Go*, where only Black (White) can play.

3.2 Outcome Classes and Search Results

As usual, we identify Black with Left and positive values, and White with Right and negative values. The four possible outcome classes for a game G in CGT [15] correspond directly to the four combined results of two boolean minimax searches of G with alternating play: In outcome class $\mathscr{L}$, **L**eft wins both going first and going second; in $\mathscr{R}$, **R**ight wins in both searches; in $\mathscr{N}$, the **n**ext player (toPlay) wins both times; and in $\mathscr{P}$, the **p**revious (second) player wins both.

3.3 CGT Concepts

We provide a short review of the most relevant CGT concepts used in this paper. For a thorough introduction, we refer to the literature [15]. A game G is defined recursively as a pair $G = \{G^L|G^R\}$ of left and right options, which are in turn sets of games.

Sum of Games In a sum of subgames $G_1 + G_2 + \cdots + G_k$, a player must move in exactly one of the components G_i to an option G_i^L or G_i^R, leaving all other subgames unchanged.

Inverse For a game $G = \{G^L|G^R\}$, its inverse $-G$ is defined recursively by $-G = \{-G^R|-G^L\}$. In NoGo, the inverse of a game can be obtained by swapping the color of all stones from black to white and vice versa.

Equality Two games G and H are called *equal* if for all games X, the outcome class of $G+X$ is equal to the outcome class of $H+X$. All games of second-player win (in outcome class $\mathscr{P}$) are equal to 0. If $G = H$, then the *difference game* $G - H = 0$ is a second-player win. This fact can be used to simplify games by search.

4 Simplifying and Reducing $1 \times n$ NoGo Games

Theorem 1 (Block Simplification). *In a $1 \times n$ NoGo game, a block of stones of the same color can be replaced by a single stone of that color. The resulting game is equal to the original game. For arbitrary* <L> *and* <R>*,*
<L>x$^+$<R> $=$ <L>x<R> *and* <L>o$^+$<R> $=$ <L>o<R> .

This is clear from the rules, since representing a block by a single stone does not affect its liberties, or the set of legal move sequences of both players. As an example, .xxx..ooxx.. can be simplified to .x..ox.. by reducing larger blocks to single stones.

Theorem 2 (xo-Split). *A $1 \times n$ NoGo game can be split into two independent subgames at the boundary between two blocks of opposite colors:* <L>xo<R> = <L>x + o<R> *and* <L>ox<R> = <L>o + x<R> *.*

Proof. Separating a game G in this way into $G_1 + G_2$ does not affect the liberties of any block, or any future block when the game is continued. A move played on the left side of the boundary in the original game does not affect the liberties, and therefore the set of legal moves, in any continuation on the right, and vice versa. Since the set of legal move sequences does not change for both players, the outcomes $o(G + X) = o(G_1 + G_2 + X)$ are equal for any X, so the games are equal.

For example, .x..ox.. = .x..o + x.. is an xo-split.

Definition 1 (Reduced $1 \times n$ NoGo position). *A $1 \times n$ NoGo position is called* reduced *if neither block simplification nor xo-split can be applied.*

Any sum of $1 \times n$ NoGo positions can be simplified into an equal sum of reduced positions.

5 A Pre-computed Database of Reduced NoGo Positions

We study the set of all reduced positions with a given number n of empty points.

Theorem 3. *For all $n > 0$, there are 3^{n+1} distinct reduced positions with n empty points.*

Proof. By definition, a reduced position does not contain adjacent stones of the same color, or adjacent stones of opposite color. So any neighbors of a stone must be empty points. We can build any position with n empty points by starting with those empty points, then optionally adding a single stone at either end and between two empty points. There are $n+1$ insertion locations with three choices each (Black stone, White stone, and no stone), a total of 3^{n+1} distinct positions.

Consider all reduced positions with $n = 3$. For the empty board ..., the four possible stone insertion points marked by underscores are _._._._, and at each _ we can insert x, o, or nothing. Positions ... and x..o.x are two examples.

5.1 The Database

Our pre-computed database contains all reduced $1 \times n$ NoGo positions with up to 15 empty points, and information about their value. With this database, CGTSolver can solve games more efficiently. It can query a game directly in the database, and it can use information about subgames of a sum. Each database entry stores a reduced position, its outcome class, and for parts of the database, a pointer to the simplest equal game. The database is organized in layers, by the number of empty points n in each position. It is built incrementally from $n = 1$ to $n = 15$. The data stored for positions with smaller n is used to speed up the computation of later positions.

The analysis above provides an efficient way of constructing all reduced boards with a fixed number of empty points. To order all reduced positions, we first sort them by the number of empty points, then by a base 3 encoding of the inserted stones. Let $n = n(G)$ be the number of empty points in game G, and $code(G)$ be a $n + 1$ digit base 3 number. Each potential insertion point has code 0 if no stone is inserted, 1 for a black stone, and 2 for a white stone. In the examples above, `code(...) = 0000` and `code(x..o.x) = 1021`. The ordering $\mathrm{ord}(G)$ is the dictionary ordering of the pairs $(n(G), code(G))$.

5.2 Simplest Equal Game

Given a $1 \times n$ NoGo position, its *simplest equal game* is defined to be the smallest game in the ordering defined above that is equal to this game. During search, we replace each subgame in a sum with its simplest equal game, if known. This reduces the breadth and the depth of the search.

As an example, the game $A =$ `..o.x.o..x` is equal to the game $B =$ `..`, $A = B$, and B is the simplest game equal to A. This is proven when constructing the database by verifying that $A - B = 0$, a second player win, while $A - X \neq 0$ for all games X that are simpler than B. The consequence is that during search, any subgame A can be replaced by the much simpler B.

In general, to add the simplest equal game of game G to the database, we search if $G - H = 0$ for simpler games H with $\mathrm{ord}(H) < \mathrm{ord}(G)$. We try candidate games H in increasing order in the database.

5.3 Database Statistics

Our database contains all $\sum_{i=1}^{15} 3^{i+1} = 64,570,077$ reduced positions for $n \in \{1, \ldots, 15\}$, and their outcome class. Due to computational limitations, we fully compute simplest equals only for positions up to $n = 8$. The most frequent values of positions with $n \leq 8$ are shown in Fig. 1 and Table 1. For each pair of games G and their inverse $-G$, only one is shown. "idx" in Table 1 corresponds to the "CGT game index" in Fig. 1. To limit the precomputation cost, for the larger database entries with $8 < n \leq 13$ we only computed simplest equals from among the 18 games in Table 1 and their inverses.

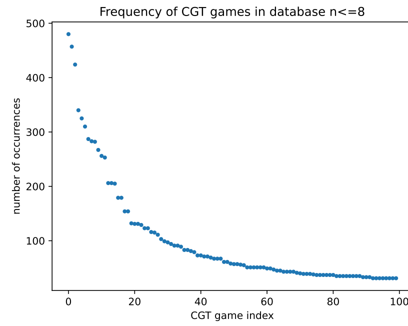


Fig. 1: The number of occurrences of the most frequent simplest equal games in the database.

idx	board	canonical form	occurrences
0	`.x.x.o.`	$1*$	481
1	`..`	$*$	457
2	`...`	± 1	424
3	`...x.o.`	$\pm(1*)$	341
4	`..x.x.`	$\{2\|1\}$	326
5	`.x.`	1	310
6	`.x.x.x.`	3	287
7	`.x.x.x.o.`	$2*$	284
8	`.`	0	282
9	`..x.`	$\{1\|0\}$	268
10	`.x.x.`	2	256

Table 1: The 11 most frequent simplest equal games in the $n \leq 8$ database.

6 State Representation and Static Evaluation of Sum Games

Much of the power of CGT comes from its representation of a game as the sum of independent subgames, and its rules for simplifying games and determining their outcome class. In CGTSolver, each node in the search tree stores a sum of one or more subgames, plus whose turn it is.

6.1 Static Evaluation

If the outcome classes of all remaining subgames are known from the database, we can sometimes determine the search result of the sum statically [16], and save further search. This is trivially true when a sum consists of a single such subgame only. In larger sums, Black going first wins games $G + H$ with $o(G) = \mathscr{L}$ and $o(H) = \mathscr{N}$, where G itself can be a sum, and similarly white going first wins when $o(G) = \mathscr{R}$.

7 Implementation Details of CGTSolver

In this section, we discuss some implementation details of CGTSolver. The core search engine is Boolean Negamax. In addition to static evaluation, CGTSolver uses CGT-based techniques to simplify and split large games into independent subgames. Pseudocode is provided in Algorithm 1.

7.1 Updating Sum Games

In each recursive negamax call, a move is played in one subgame, followed by simplification and split, which changes the sum. To undo a move, the sum is restored to its previous state. The search depth of a NoGo position with n empty

Algorithm 1 NEGAMAX for sum games

Require: *games* - a collection of games to evaluate
player - the color of current player
Return: true if current player wins; **false** otherwise

```
result ← TranspositionTable(games, player)
if result is valid then return result
result ← StaticEvaluation(games, player)
if result is valid then return result
moves ← GetLegalMoves(games)
for move in moves do
    Play(games, move, player)
    Simplify(games)
    Split(games)
    DataBaseLookUp(games)
    win ← not Negamax()
    undo(games, move, player)
    if win then return true
return false
```

points can be up to $n - 1$. CGTSolver uses a *change stack* to track the game history. The stack stores both previous (inactive) and current (active) subgames, in the order of their creation. The stack initially contains only a single active subgame for the starting position. Each move is played on one active subgame G, resulting in zero or more new subgames H_i. G is deactivated, and all H_i are pushed on top of the stack as active games. To undo this move, all H_i are popped from the stack, and G is re-activated.

To play a move in subgame G, the move is first played on a copy of G. The resulting local board is simplified, splits are done, and subgames found in the database are possibly replaced by their simplest equals. Subgames of value 0 are removed. The zero or more new subgames H_i are added to the current sum S. If for a new subgame H_i, its inverse $-H_i$ is also in S, both are deactivated, and this is recorded on the change stack as well.

7.2 Play-In-The-Middle Heuristic

The play-in-the-middle (PITM) heuristic tries moves closer to the middle of a subgame first. It helps to quickly break down a large game into smaller subgames, increasing the chance of database hits.

For global move ordering, active subgames in the stack are processed in order from largest to smallest in length, using PITM in each subgame.

7.3 Global Transposition Table

A transposition table stores the results of previously searched nodes. The table greatly improves a game solver's performance by searching nodes that can be

reached via different move sequences only once. The transposition table in CGT-Solver is implemented as a hash table. To efficiently hash a game position that is a sum of subgames, similar to the approach of Folkersen [5] we use a hashing scheme based on sorting to achieve a normal form of a sum, followed by Zobrist Hashing [18].

The board of each subgame is considered as a string, with standard string ordering. The string is replaced by its reverse if it comes first in the order. For the normal form, the strings of all subgames in a sum are sorted in increasing order, and then concatenated as follows: A sum game S is represented by a string of four characters: `x`, `o`, `.`, and a subgame separator symbol `|`. Sorted subgame strings are concatenated, with `|` separators in between. The resulting unique string representation of S is then hashed. Given a maximum game length (including separators) of N, a Zobrist hash code is prepared for each location $i \in \{0, \ldots, N-1\}$ and each of the four characters in four random number tables $\mathcal{R}^B$, $\mathcal{R}^W$, $\mathcal{R}^E$, $\mathcal{R}^|$ with the codes for black, white, empty, and separator. The string representation s of S is hashed to $h(s) \doteq \bigoplus_i \mathcal{R}_i^{s_i}$, where s_i is the i-th character in s, $\mathcal{R}_i$ is the i-th entry in a random number table, and $\bigoplus$ is the XOR operator.

8 Results and Performance Analysis

We solve $1 \times n$ NoGo boards for $n = 8$ to $n = 39$. In each case, we pre-play B1, the second point from the end, as a strong first move and prove a win for Black. This confirms previous results up to $n = 27$, and agrees with the general conjecture in [4]. We extend these previous results by computing win/loss for all opening moves up to $n = 33$. New results are shown in Fig. 3.

We compare SBHSolver and CGTSolver in terms of node count and wall-clock time in Fig. 2a and 2b. For larger boards, CGTSolver uses over two orders of magnitude fewer nodes than SBHSolver, and is also faster by a similar ratio[2]. The CGT-based optimizations in CGTSolver do not add much overhead to the main search algorithm.

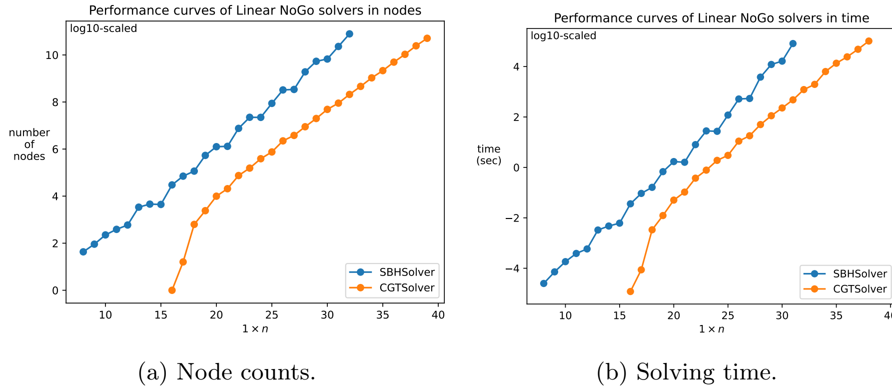
8.1 Ablation Study

The ablation study shown in Fig. 4 evaluates the effect of omitting one of the CGTSolver components: the PITM heuristic, the replacement scheme of simplest equal games, the static evaluation to catch early wins, and the transposition table. The transposition table most strongly improves the performance, but unexpectedly, the benefit of static evaluation diminishes with increasing board size.

8.2 In-Search Statistics

Fig. 5 presents detailed search statistics for solving 1×30 NoGo with B1 pre-played. The shallowest terminal nodes appeared early at a depth of 4. The CGT

[2] Wall-clock time measured on a single core AMD EPYC 7313.



(a) Node counts. (b) Solving time.

Fig. 2: Performance comparison of SBHSolver and CGTSolver for solving $1 \times n$ NoGo with the B1 opening. The graphs are log10-scaled on the y-axis.

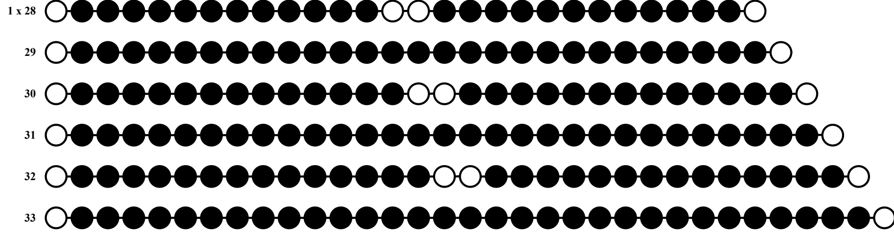


Fig. 3: The opening moves on empty $1 \times n$ NoGo with $28 \leq n \leq 33$. A black (white) stone indicates a winning (losing) opening move.

approach greatly reduces the search depth. The maximum number of subgames in a position was 6. The most common number of subgames was 2. Having more subgames of smaller size increases the likelihood of detecting wins and losses early. A potential improvement would be targeted move ordering heuristics that aim to produce more xo-splits.

8.3 Comparison with CGSuite

Siegel's CGSuite [14] is a very powerful and versatile system that implements many CGT algorithms. This system has been widely used for research in this area. However, it is not an efficient tool for solving $1 \times n$ NoGo due to its dependence on canonical form computations, which can become very complex. We could compute results with CGSuite for boards up to $n = 16$ in several minutes on comparable hardware. The 1×16 board has a massive canonical form with 1201194 stops! Our attempt to compute $n = 17$ resulted in a Java heap space overflow. In contrast, CGTSolver running from scratch, without using its precomputed database, proves the Black win for $n = 16$ in less than 30 milliseconds.

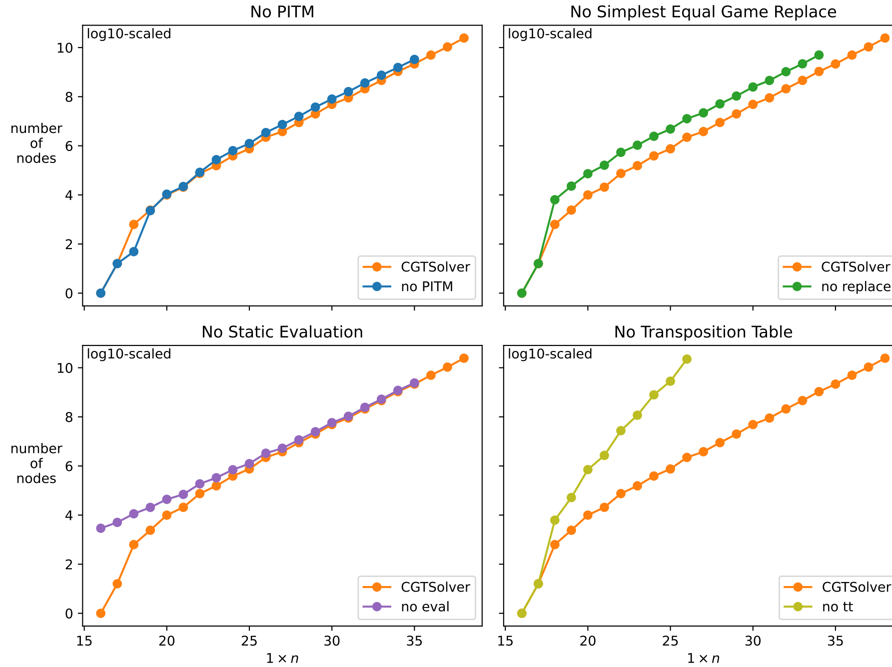


Fig. 4: Ablation on components of CGTSolver: the PITM heuristic, replacement with simplest equal games, static evaluation, and transposition table (TT).

9 Conclusions and Future Work

CGTSolver is a CGT-enhanced Negamax search algorithm that solves $1 \times n$ NoGo positions efficiently. We exploit several game properties in order to solve a position as a simpler sum of subgames, which greatly improves the search. Our experiments show that CGTSolver far surpasses the previous state of the art in terms of node counts, wall-clock time, and new results. We solved boards up to 1×39, including new results on 12 boards of size 28 and larger.

For future work, we would first like to extend our specific techniques to other NoGo boards. Block simplification works on any graph, not just on a one-dimensional strip, by contracting all edges between neighboring stones within the same block. For board decomposition, instead of xo-Split, which relies on the linear board structure, the techniques of Shan [12] can be used.

We believe that similar sum game solving techniques can be applied to a wide class of "short" combinatorial games. Search approaches based on decomposition work well when there are many subgames, as in linear Clobber [5] and Go endgames [7], and even on small Amazons boards, which have very few subgames [16].

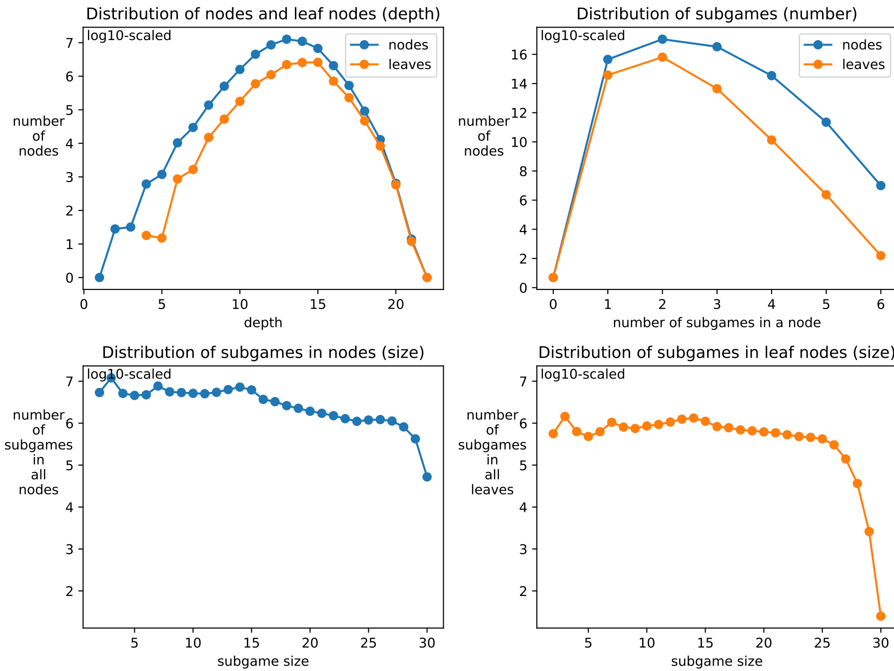


Fig. 5: Statistics in solving 1×30 NoGo with B1 opening: the number of nodes and leaf nodes across depths, the number of nodes and leaf nodes having different numbers of subgames, and the number of subgames of different sizes in all nodes and leaf nodes of the game DAG.

References

1. Anti Atari Go. https://senseis.xmp.net/?AntiAtariGo, accessed: 2024-02-25
2. Cazenave, T.: Monte carlo game solver. In: Monte Carlo Search, MCS 2020. Communications in Computer and Information Science, vol. 1379, pp. 56–70 (2021)
3. Chou, C.W., Teytaud, O., Yen, S.J.: Revisiting monte-carlo tree search on a normal form game: NoGo. In: Applications of Evolutionary Computation (2011)
4. Du, H., Wei, T.H., Müller, M.: Solving NoGo on small rectangular boards. In: Advances in Computer Games. pp. 39–49 (2024)
5. Folkersen, T.: Linear Clobber solver (2022), Capstone report, University of Alberta
6. Gao, Y., Wu, L.: Efficiently mastering the game of NoGo with deep reinforcement learning supported by domain knowledge. Electronics **10**(13) (2021)
7. Müller, M.: Decomposition search: A combinatorial games approach to game tree search, with applications to solving Go endgames. In: IJCAI. pp. 578–583 (1999)
8. Müller, M.: Global and local game tree search. Information Sc. **135**, 187–206 (2001)
9. Müller, M., Li, Z.: Locally informed global search for sums of combinatorial games. In: Computers and Games. LNCS, vol. 3846, pp. 273–284 (2006)
10. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Exploiting graph properties of game trees. In: AAAI/IAAI, Vol. 1 (1996)

11. Schaeffer, J.: The history heuristic. ICGA Journal **6**(3), 16–19 (1983)
12. Shan, Y.C.: Solving Games and Improving Search Performance with Embedded Combinatorial Game Knowledge. Ph.D. thesis, National Chiao Tung University (2013)
13. She, P.: The Design and Study of NoGo Program. Master's thesis, National Chiao Tung University (2013)
14. Siegel, A.: CGSuite. A computer algebra system for research in combinatorial game theory (2003–2024), https://www.cgsuite.org
15. Siegel, A.: Combinatorial Game Theory, vol. 146. American Math. Soc. (2013)
16. 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). https://doi.org/10.1109/TCIAIG.2014.2309077
17. Uiterwijk, J., Griebel, J.: Combining Combinatorial Game Theory with an α-β Solver for Clobber: Theory and Experiments. In: BNAIC 2016 (2017)
18. Zobrist, A.L.: A new hashing method with application for game playing. ICCA Journal **13**(2), 69–73 (1990)