# Solving NoGo on Small Rectangular Boards

Haoyu Du, Ting Han Wei, and Martin Müller[0000−0002−5639−5318]

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

**Abstract.** The game of NoGo is similar to Go in terms of rules, but requires very different strategies. While strong heuristic computer players have been created for NoGo, solving and optimal play have been less studied. We introduce Sorted Bucket Hash (SBH), a new approach to building transposition tables for game solvers, and apply it to solve NoGo on small boards. Using boolean negamax with standard heuristics and SBH, our program SBHSolver has now solved NoGo on 50 different rectangular boards including $3 \times 9$, the largest solved NoGo game to date. It re-solved $5 \times 5$ NoGo much more efficiently than She's work in 2013 and Cazenave's work in 2020. The SBH data structure can also efficiently extract a proof tree for the game. We provide analyses of NoGo proof trees and games, and discuss human-understandable strategies from this perspective.

**Keywords:** Game solving · Transposition tables · NoGo · Sorted Bucket Hash.

## 1 Introduction

NoGo (or *Anti-Atari Go* [5]) is a lesser-known variant of the widely studied game of Go. The rules of NoGo are:

1. Black and White take turns to play, with Black going first. At each turn a stone of the player's colour is placed onto an empty point on the board. Passing is forbidden.
2. Connected stones of the same colour form a *block*. Adjacent empty points of blocks are called *liberties*.
3. All blocks must always have at least one liberty. For Go players, this means that both suicide and capturing are forbidden.
4. The game ends when a player has no legal move to play. This player is deemed the loser.

The simple twist to the rules makes NoGo very different to play. Unlike Go, where *ko* fights can greatly extend the length of a game, the number of moves in NoGo is strictly less than the size of the board, since each block needs an empty point for a liberty. The game state is completely determined by the current board. However, NoGo still has a state space that grows exponentially with the number of points on the board, and no easy winning strategies are known.

The three main contributions in this paper are:

1. The data structure of Sorted Bucket Hash (SBH) for space-efficient transposition tables based on perfect hashing. SBH is designed for use in weakly solving games.
2. Efficient solutions of NoGo on all 50 rectangular boards of size up to 27 points, with orders of magnitude fewer nodes than the $5 \times 5$ NoGo solution by She [9].
3. New results on NoGo, such as a winning strategy for Black on $5 \times 5$ in at most 21 moves, two general results for $1 \times n$ NoGo, and a statistical analysis of two human-understandable heuristics from the solver's perspective.

## 2 Related Work

NoGo is a relatively young game with few human players. Previous research mostly focused on creating strong computer agents for competitions such as the Computer Olympiad, TAAI (Taiwanese Association for Artifical Intelligence), and TCGA (Taiwan Computer Game Association) tournaments. Early programs include BobNoGo [4], an open-source program based on MCTS that includes an exact solver.

NoGo was proven a win for the first player (Black) on the $3 \times 3$ board and a loss on $4 \times 4$ in 2011 [6]. All three distinct opening moves on $3 \times 3$ boards win. $5 \times 5$ NoGo was solved in 2013 by Pohsuan She [9]. Black can win with all six distinct opening moves, as shown in Fig. 1. Cazenave determined the winner of NoGo played on boards of size up to 25 points by using alpha-beta search with Monte Carlo Move Ordering and a transposition table of size 1048575 entries [1].

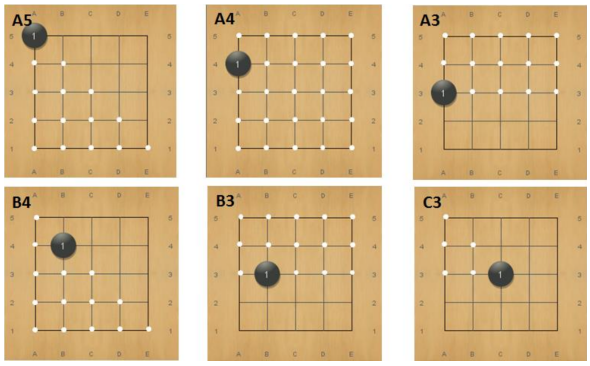



Fig. 1: The six distinct openings for $5 \times 5$ NoGo, from [9]. White dots represent distinct possible replies from White.

## 3 Sorted Bucket Hash and its Use in Solving Games

### 3.1 Boolean Negamax Search and Heuristics

While Sorted Bucket Hash is a general data structure, we explore it in the specific context of transposition tables for Boolean Negamax, which is a simpler special case of alpha-beta search [3] for two-valued outcomes. Our search also uses two standard heuristics, Enhanced Transposition Cutoff [7] and History Heuristic [8].

### 3.2 A Transposition Table with Perfect Hashing

A transposition table is useful to avoid redundant evaluation when an equivalent game position can be reached via different move sequences. In typical implementations, each game state is mapped to a $k$-bit hash code. An $m$-bit part of the code is used as an index into a hash table, and the remaining $n = k - m$ bits are used as a validation code which is stored with each hash entry.

**Tradeoffs for Designing Hash Tables** Designing an efficient hash table requires navigating trade-offs between speed, memory used, and the amount of information stored per position. Our solver is designed for the following scenario:

- Large state space, with many transpositions, so maximizing the size of the table is important.
- Storing all solved positions, in contrast to using a fixed memory table with a replacement scheme and re-search [2].
- The amount of data per position that needs to be stored is minimal. One bit for storing win/loss is enough (a few extra bits are useful as discussed below).
- Perfect hashing, as discussed below.

**Perfect vs Lossy Hashing** With perfect hashing, each game position is mapped to a unique hash code. For example, all the states of a NoGo board with $n$ points can be mapped to $3^n$ distinct codes. If storing a full hash code takes too much space, lossy hash functions such as 64-bit Zobrist Hashing [10] are used in practice. However, in our application 64-bit Zobrist codes used too much memory, and smaller codes led to too many hash collisions. This was the main motivation for developing SBH based on perfect hashing.

**Sorted Bucket Hash** Sorted Bucket Hash (SBH) is a new method for organizing a transposition table. Given a game state $s$, SBH uses a perfect hash function $h(s)$ that produces $k = m + n$-bit hash codes. The $m$-bit segment of a hash code represents the bucket index, and the $n$-bit segment serves as the validation code. A SBH hash table consists of $2^m$ buckets. Each bucket holds at most $2^n$ entries. Buckets are empty at the beginning and become populated as code-value pairs

are stored. A bucket entry consists of a 1-bit game value and an $n$-bit validation code. SBH keeps the entries in each bucket sorted by their validation codes. Binary search based on validation codes is used to find entries and insertion points in a bucket. The find operation of SBH takes $O(1)$ for the hashing part, and $O(n)$ for binary search among the at most $2^n$ validation codes inside a bucket. This compares favorably with the $O(2^n)$ linear search in chaining.

The values of $k$, $m$, and $n$ are problem-dependent and should be carefully selected by finding the balance between algorithm efficiency and the actual memory size. Below, we discuss our choices for solving NoGo.

**SBH Find and Store Operations** The operations find and store are implemented as follows: A given $k$-bit hash key is split into $m$-bit index and $n$-bit validation code. The index selects the bucket, and binary search of the validation code completes the find operation. Store involves a find of the correct location within the right bucket, followed by allocating a new one larger array and copying the old data over in two parts, with the new entry stored in between.

**Collecting Solutions in SBH** SBH provides an easy and efficient way to extract a winning strategy from the transposition table after a successful search.

This uses an extra "proof flag" and an encoded winning move stored in each hash entry (see details for NoGo below). Solution extraction marks all nodes that are part of the proof tree, starting with the root. Another Boolean Negamax "search" is guided by the results stored in the transposition table: at each OR node, the stored move is chosen to find a child node, while in an AND node all children are traversed. The proof flag is set for all the nodes encountered, and the set of marked nodes forms the solution.

To actually store the solution to disk, sequentially visit all buckets and write out all marked nodes. The size of the stored solution is typically much smaller than the original transposition table after the search. This stored solution is also sorted by full hash codes, making it easy to reload the solution into the SBH transposition table for game playing or analysis later. For example, storing the solution takes less than a minute for $5 \times 5$ NoGo on modest hardware.

**SBH Implementation for NoGo** We discuss implementation details of SBH for our program SBHSolver in the case of $5 \times 5$ NoGo, where a comparison to previous work is possible. For hashing, a board is encoded as a 25-digit base 3 number, with point encoding empty = 0, black = 1, and white = 2. Since $2^{39} < 3^{25} < 2^{40}$ this requires using $k = 40$-bit hash codes. A small $m$ results in poor performance due to many positions being hashed into the same bucket, while a large $m$ potentially leads to more memory overhead. Based on our hardware (32GB of memory) and some experimentation, we used an $m = 30$-bit index and an $n = 40 - 30 = 10$-bit validation code, with a bucket size limit of $2^{10} = 1024$. An example of calculating the hash code, index, and validation code of a NoGo board is shown in Fig. 2.

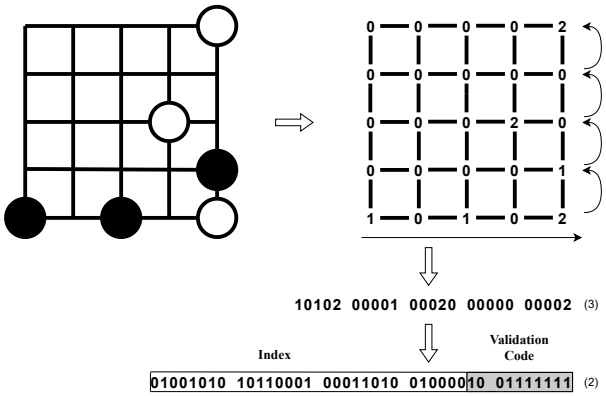



Fig. 2: The calculation of hash code, index, and validation code of a NoGo board.

$5 \times 5$ NoGo has over $8.47 \times 10^{11}$ possible states that need to be addressed, requiring 40 bits. A typical proof of one opening move visited about $2.42 \times 10^8 < 2^{28}$ distinct game positions, less than 0.03% of the represented space of $2^{40}$.

SBHSolver is implemented in C++. The hash table is an array of $2^{30}$ pointers as shown in Fig. 3. Each pointer points to a bucket, or is null if the bucket is empty. A bucket is a dynamically allocated sorted array with at most $2^{10}$ entries. Each entry occupies 2 bytes and consists of 1-bit proof flag, 5-bit winning move/ outcome, and 10-bit validation code. The 5-bit winning move represents either a winning legal move for the current player or an illegal move encoded as 11111. The illegal move 11111 implies a loss, while any legal move implies a winning state. The proof flag is used for collecting a solution. Bucket entries are sorted by validation code in ascending order. The size of a bucket grows by 1 with each insertion.

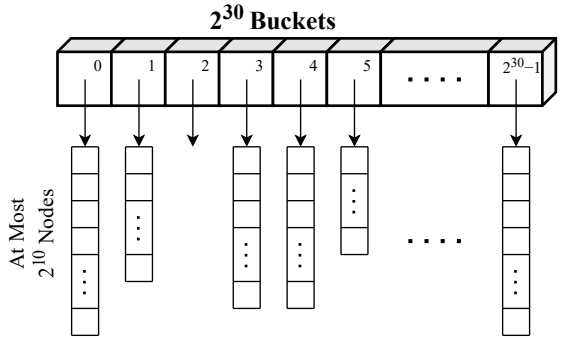



Fig. 3: Data structure for $5 \times 5$ NoGo using Sorted Bucket Hash with $m = 30$, $n = 10$. Each cube represents one of the $2^{30}$ buckets. Each bucket stores up to $2^{10}$ positions sorted by validation code. In this example, bucket 2 is empty.

## 4 NoGo Results and Solution Analysis

### 4.1 4 × 4 NoGo

In 4×4 NoGo, the second player (White) wins. For each of Black's three distinct initial moves, the symmetric reply shown in Fig. 4 is the only win for White.

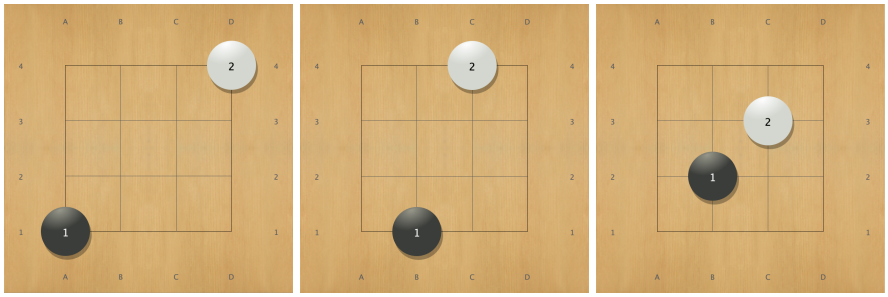


Fig. 4: The 3 distinct Black opening moves and White's unique winning replies.

Playing symmetrically does not win in all 4 × 4 positions where it applies, but it is an effective heuristic. Among 24,708 game positions where a white move can make the board symmetric, such a move wins 85.3% of the time.

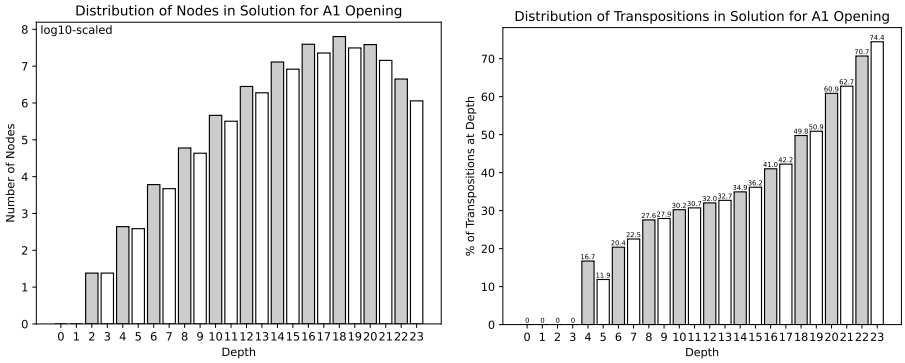
### 4.2 5 × 5 NoGo

SBHSolver proved that Black wins with all six distinct opening moves shown in Fig. 1, confirming the result from She [9]. As a new result, we proved that Black can force a win in at most 21 moves for all six openings, but cannot force a win in 19 moves or less. We showed these by introducing a threshold $T = 21$ (or $T = 19$), and modifying the evaluation such that games that are not won in $T$ moves are losses for Black.

As an example of the efficiency gains, to solve the A1 opening SBHSolver evaluated 2,968,264,746 distinct game positions. A subset of 242,002,061 positions forms the solution. Fig. 5a shows the distribution of these proven game positions by depth. The number of nodes is largest at depth 18, then decreases due to two factors: more games being decided, and having more transpositions.

There are fewer white-to-play positions than black-to-play positions one ply earlier, since for each white-to-play position we only identify one winning black move, and there are transpositions among the resulting black-to-play positions. The frequency of transpositions within the solution increases with depth, as shown in Fig. 5b.

SBHSolver is efficient compared to She's solution [9]: using SBH with boolean minimax and two standard heuristics, SBHSolver searches on average 188× fewer nodes in solving the six distinct openings. It is also 15× faster than Cazenave's solution [1] that evaluated 46,092,056,485 moves.




(a) Distribution of nodes by depth. (b) Frequency of transpositions by depth.

Fig. 5: Analysis of a solution to 5x5 NoGo with the A1 opening, by search depth.

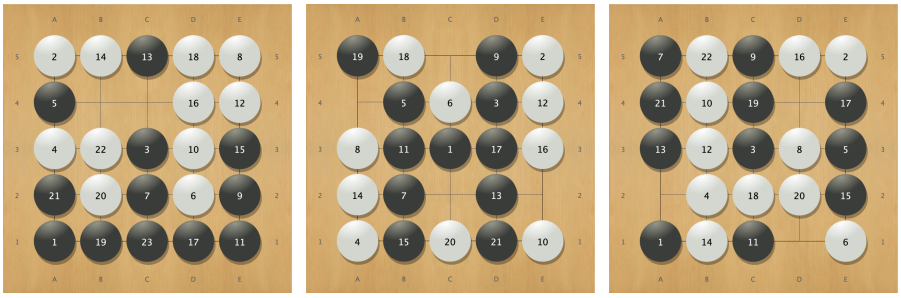
**Sample $5\times5$ Games** Our perfect SBHPlayer follows one of our winning strategies for Black. We tested it against the strong open source program BobNoGo [4]. Figs. 6a and 6b show two representative wins. The first game follows the original A1 proof, and the second game the 21-move solution of C3.

In the third game in Fig. 6c SBHPlayer wins against BobNoGo even as White, from the losing side. At move three, SBHPlayer knows that E2 is winning for Black. However, BobNoGo chose C3, and SBHPlayer quickly exploited this weak move and solved this variation by search, partially relying on the precomputed solution to avoid mistakes, and won in 22 moves.

**Comments on NoGo Strategy** Making eyes, an important element of Go strategy, is often recommended for NoGo as well, and is frequently seen from the MCTS-based BobNoGo player. Eyes represent a local advantage, "reserving" points for the player. However, only 45.2% of the terminal positions in the A1 solution contain any eyes for Black, so SBHPlayer very often wins without them. Instead, SBHPlayer seems to favor creating long blocks that separate the opponent's stones. This seems to be an effective strategy, at least in $5\times5$ NoGo. Both behaviors are seen in all the games in Fig. 6: SBHPlayer creates long blocks, but no eyes.

### 4.3 Solutions of Rectangular NoGo Boards

Figs. 7 and 8 summarize the results of SBHSolver on solving rectangular NoGo boards. It solved all such boards with up to 27 empty points. Black wins on most boards. Notable exceptions, which are White (second player) wins, are $1\times1$, $4\times n$ with $n\leq4$, and several $2\times n$ boards. The results on $3\times9$, $1\times n$ with $10<n\leq27$, and $2\times n$ with $10<n\leq13$ are new compared to [1].



(a) SBHPlayer (Black) (b) SBHPlayer (Black) (c) SBHPlayer (White)

Fig. 6: Three sample games against BobNoGo. SBHPlayer wins even as White.

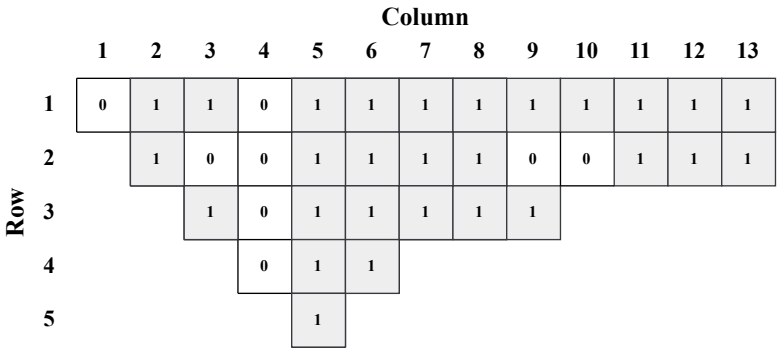



Fig. 7: Table of game theoretic values for rectangular NoGo boards. 1 means a win for Black from the empty board, 0 a loss. Symmetric results for *row* > *column* are omitted to save space. For more $1 \times n$ results see Fig. 8.

**$1 \times n$ NoGo** Fig. 8 summarizes the win/loss results of all first moves on $1 \times n$ NoGo boards for $n \leq 27$. A black stone indicates a winning move for Black, a white stone a loss. Black wins with most opening moves, especially on large boards. Symmetry arguments lead to two general results for arbitrary $n$:

**Theorem 1.** *In $1 \times n$ NoGo, with odd $n > 1$, the center point $(n+1)/2$ wins.*

*Proof.* The Black move at $(n+1)/2$ divides the board into two subgames. A winning strategy for Black is to mirror the previous White move. Whenever a white move is legal because of a liberty at some point $x$, the mirror black move is also legal because of a corresponding liberty at $n+1-x$. An example of this strategy is shown in Fig. 9.

**Theorem 2.** *In $1 \times n$ NoGo, with even $n > 2$, the two middle points $n/2$ and $n/2+1$ lose.*

*Proof.* If Black plays at one middle point, a winning strategy for White is to play the other, then follow the mirroring strategy. The game splits into two

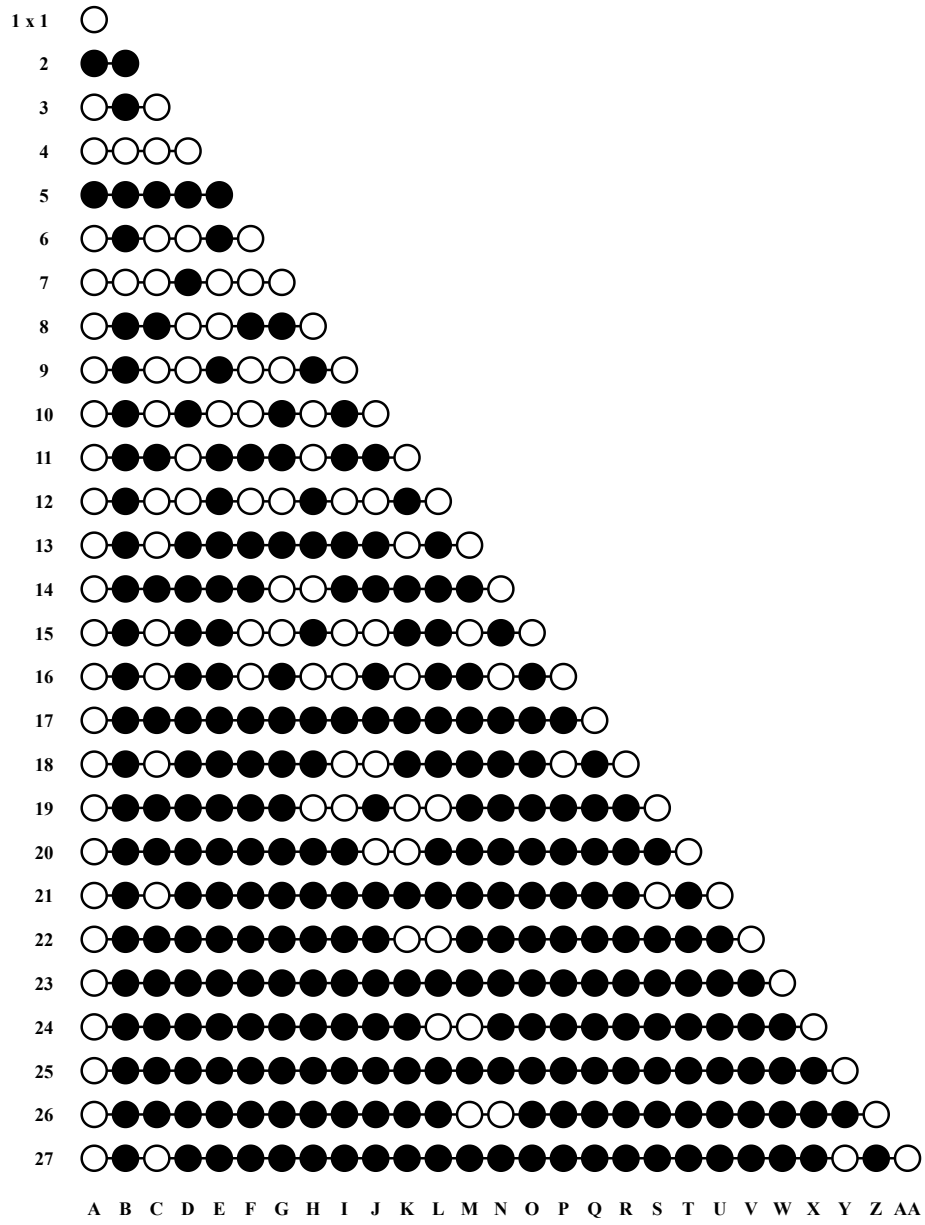



Fig. 8: Opening results for $1 \times n$ NoGo with $n \leq 27$. Row $i$ evaluates the opening moves of $1 \times i$ NoGo (black = winning move, white = losing). For example, in $1 \times 7$ NoGo, the only winning first move for Black is D1 in the middle.

independent subgames $G$ and $-G$ in terms of Combinatorial Game Theory, and $G + (-G) = 0$, a second player win. If Black plays at point $x$ (in either $G$ or $-G$), then $n + 1 - x$ is guaranteed to be a legal move for White in the inverse game. An example is shown in Fig. 10.

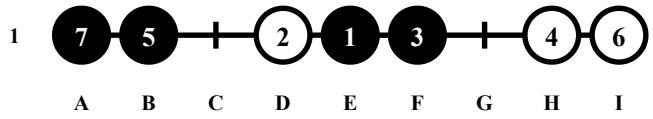



Fig. 9: In $1 \times 9$ NoGo, Black wins by playing at the middle point E1 and then mirroring White's moves.

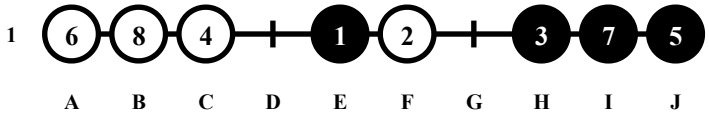



Fig. 10: In $1 \times 10$ NoGo, Black loses if playing at E1 or F1 as the opening move. White can win by mirroring Black's moves.

Black wins on all solved boards except $n = 1$ and $n = 4$. The data in Fig. 8 supports two conjectures: For $n > 5$, a move at location 1 always loses, and for $n > 7$, a move at location 2 always wins. A move at location 3 often loses on smaller boards, but these losses seem to become less frequent. What is the trend for larger $n$?

## 5 Summary and Future Work

The new hashing scheme of Sorted Bucket Hash (SBH) is designed for weakly solving games, and extracting their solution strategies efficiently. The effectiveness of SBH is validated by the SBHSolver program, which found the game-theoretical value and winning strategies for NoGo on all board sizes up to 27 points. An analysis of NoGo strategies shows that on the $4 \times 4$ board, White can often benefit from playing symmetrically. On the $5 \times 5$ board, in addition to making eyes as in Go, building long blocks that separate the opponent's stones is also a good strategy.

The SBH data structure as well as the SBHSolver implementation can be improved further:

1. With the current simple encoding, many game positions hash into the same buckets, slowing down find and store operations over time. Spreading out the perfect hash codes, such as with a linear congruence mapping, should give better distribution across buckets, potentially improving the performance.
2. Depending on game encoding details, large stretches of consecutive buckets may remain empty. A better data structure could compress long stretches of null pointers.
3. SBHSolver reallocates bucket memory at each insertion. An alternative strategy such as doubling the bucket size would reduce memory copies at the cost of larger storage requirements.
4. To solve even larger games, where the search does not fit into main memory, SBH should be combined with a two-tier (disk+memory) storage scheme.

## References


1. Cazenave, T.: Monte carlo game solver. In: Monte Carlo Search, MCS 2020. Communications in Computer and Information Science, vol. 1379, pp. 56–70 (2021)
2. Kishimoto, A.: Correct and Efficient Search Algorithms in the Presence of Repetitions. Ph.D. thesis, University of Alberta (2005)
3. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence **6**(4), 293–326 (1975)
4. Müller, M.: The BobNoGo program. `https://webdocs.cs.ualberta.ca/~mmueller/nogo/BobNoGo.html`, accessed: 2023-05-31
5. Müller, M.: NoGo history and competitions. `https://webdocs.cs.ualberta.ca/~mmueller/nogo/history.html`, accessed: 2023-05-31
6. Müller, M.: Solving NoGo on small board sizes. `https://webdocs.cs.ualberta.ca/~mmueller/nogo/solving.html`, accessed: 2023-05-31
7. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Exploiting graph properties of game trees. In: AAAI/IAAI, Vol. 1 (1996)
8. Schaeffer, J.: The history heuristic. ICGA Journal **6**(3), 16–19 (1983)
9. She, P.: The Design and Study of NoGo Program. Master's thesis, National Chiao Tung University (2013)
10. Zobrist, A.L.: A new hashing method with application for game playing. ICCA Journal **13**(2), 69–73 (1990)