An Analysis of Adversarial Search Techniques in (m, n, k)-games

Owain Thorp and George Bateman

Department of Computer Science
Imperial College London

November 28, 2025

1 Code Overview

1.1 `game.py`

This file contains all of the source code for the `Game` class, which contains the core logic for the (m, n, k)-game. It is worth noting that we use a *bitboard* for keeping track of `min`/`max` board states. We use this more advanced technique for a significant drop in latency, which allowed us to probe higher-order games with less compute.

In our code, the board width is defined as $w = n + 1$. This creates a virtual "buffer column" to prevent horizontal win-checks from wrapping around to the next row. For a 3×3 board, $w = 4$. To provide more intuition, we provide some examples below.

Empty Board

Because $w = 4$, the indices of the board are not strictly sequential. Indices $3, 7$, and 11 are skipped (they act as the buffer). The valid indices for the playable cells are shown below:

0	1	2
4	5	6
8	9	10

Figure 1: Bit index mapping for a 3×3 board with buffer $w = 4$.

Having indexed the board, we now explain how to map board indices to bit representations.

Populated Board

Let us continue with the example of a 3×3 board. Suppose **Max** (X) has moved twice and **Min** (O) has moved once.

X		**X**
	O	

Figure 2: Game state with two Max pieces and one Min piece.

To encode this information as a (binary) integer, we make use of the **bitshift** operation, denoted $\ll$. If we have some binary string s, then $s \ll n$ is the bit string of s followed by n zeros. For example, if $s = 10101$ and $n = 4$:

$$s \ll n = \ldots 000101010000$$

In general, to encode some index i, we use the operation $1 \ll i$. For boards with multiple indices, we simply perform addition. In the above board, this corresponds to:

1. **Max's Board (`self.max_board`):**
   Max is at (0,0) and (0,2). The indices are 0 and 2.

$$\text{Value} = 2^0 + 2^2 = 1 + 4 = 5 \quad (\text{Binary: } \texttt{...000000101})$$

2. **Min's Board (`self.min_board`):**
   Min is at row 1, col 1. The index is $1 \times 4 + 1 = 5$.

$$\text{Value} = 2^5 = 32 \quad (\text{Binary: } \texttt{...000100000})$$

To check if a square is occupied (as seen in `is_valid_move`), the code performs a bitwise OR of both boards:

$$\texttt{occupied} = 5 \mid 32 = 37 \quad (\text{Binary: } \texttt{...000100101})$$

This integer uniquely represents the entire configuration of the board in memory, as the encoding function is a bijection.

Win Detection

We check for wins using highly efficient bitwise shifts rather than iterating through coordinates. To check for a winning line of length k, we iteratively shift the board and compute the bitwise AND.

For a vertical win check on a 3×3 board ($k = 3, w = 4$) with pieces at indices 0, 4, and 8:

$$\text{Board } B = 2^0 + 2^4 + 2^8 = 1 + 16 + 256 = 273 \quad (\texttt{...100010001})$$

The algorithm checks direction $d = 4$ (Vertical):

1. **Shift 1:** $B_1 = B \;\&\; (B \gg 4)$

$$B \gg 4 = 17 \quad (\texttt{...000010001})$$

$$B_1 = 273 \;\&\; 17 = 17 \quad (\texttt{...000010001})$$

   The AND operation returns a non-zero value, indicating overlapping 1's. This means we have a vertical sequence of at least 2. We repeat again to test whether this sequence is also of length 3.

2. **Shift 2:** $B_2 = B_1 \;\&\; (B_1 \gg 4)$

$$B_1 \gg 4 = 1 \quad (\texttt{...000000001})$$

$$B_2 = 17 \;\&\; 1 = 1 \quad (\texttt{...000000001})$$

Since the AND operation returns a non-zero value again, a win is confirmed. In the code, the process is repeated for all four directions: Horizontal (1), Vertical (w), Diagonal ($w + 1$), and Anti-Diagonal ($w - 1$).

Memoisation

Another computational optimisation used is *memoisation*. In the standard non-pruning Minimax, we cache the function values of visited states in order to drastically reduce the number of repeat calls. However, this direct caching strategy cannot be naively applied to α-β pruning because the algorithm often returns a lower or upper bound dependent on the specific search window (α, β) rather than the exact value of the node. More complex caching strategies can alleviate this issue, but we did not consider them for this assignment.

1.2 `tests.py`

This file contains all of the logic for our code evaluation. The script automatically runs many games across the different search architectures. The results are printed to the terminal as well as being saved to a csv file.

2 Results and Discussion

For our experiments, we benchmarked wall-clock time on the standard Minimax search with and without memoisation, as well as α-β pruning on the search for the first move of `Max`. We explored a wide range of board sizes, but insisted on a time cutoff of 100s to take a measurement.

	MinMax Memo		**MinMax NoMemo**		**Pruning**	
Board	Time	Nodes	Time	Nodes	Time	Nodes
(2,2,2)	0.00005s	41	0.00003s	41	0.00003s	21
(2,3,2)	0.00041s	213	0.00012s	285	0.00004s	53
(3,2,2)	0.00040s	213	0.00015s	285	0.00004s	51
(3,3,2)	0.00144s	2110	0.00261s	7002	0.00010s	165
(3,3,3)	0.00909s	16168	0.33609s	549946	0.01108s	18297
(4,3,3)	0.18725s	391063	>100s	N/A	0.01969s	30936
(3,4,3)	0.19418s	391063	>100s	N/A	0.04748s	75266
(4,4,3)	14.44474s	23453345	>100s	N/A	0.59299s	947128
(4,4,4)	32.57527s	51562425	>100s	N/A	>100s	N/A
(5,4,4)	>100s	N/A	>100s	N/A	>100s	N/A
(5,5,4)	>100s	N/A	>100s	N/A	>100s	N/A

The experimental results clearly demonstrate the combinatorial explosion inherent in these search-based methods. The standard Minimax algorithm operates with a time complexity of $O(b^d)$, where b is the effective branching factor (approximating $m \times n$ at the root) and d is the search depth required to reach a terminal state. As observed in the transition from $(3, 3, 3)$ to $(4, 4, 4)$, a linear increase in board dimension results in an exponential increase in execution time. Regarding the number of visited states, the unoptimised Minimax required evaluating nearly $550,000$ nodes for a simple 3×3 game, confirming that the state space grows too rapidly for a complete solution without more advanced methods on larger boards.

Looking at the unpruned methods, it is clear how powerful memoisation is for lowering latency. Since the (m, n, k)-game is Markovian (not history dependent), *how* we reach a state is not important; the state itself contains all necessary information. Empirical analysis shows that on the $(3, 3, 3)$ board alone, there were over $500,000$ redundant calculations which were avoided by the memoised implementation. Though not measured for larger instances, this redundancy scales to higher boards, explaining why the standard method timed out so quickly.

While α-β pruning theoretically improves the complexity to $O(b^{d/2})$ in the optimal case, it still struggled for larger configurations. The "N/A" entries for larger boards (e.g., 5×5) confirm the limits of basic pruning. However, the difference in visited states is significant;

on the $(3, 3, 3)$ board, pruning reduced the search space from $\approx 550,000$ nodes to just $18,297$, a reduction of over 96%.

We observe some interesting behaviours when directly comparing memoised Minimax against Pruning. It is not always the case that pruning yields lower latency. For example, on the $(4, 4, 4)$ grid, we observe that pruning takes over 100s, whereas memoised Minimax takes only 32.5s.

Pruning seems to be more effective on asymmetric boards. This makes sense as asymmetric dimensions or configurations where the winning line length k is notably smaller than the board size (e.g., $(4, 4, 3)$) often lead to "shallower" winning states. In these scenarios, the algorithm discovers terminal states or strong heuristic values earlier in the tree, allowing it to prune vast subtrees.

However, in boards where there are no shallow win conditions, pruning is less effective and the raw strength of memoisation allows it to outperform other methods. Crucially, in all boards where pruning is not faster (such as the $(4, 4, 4)$ case), the end result is a draw. In a drawn game, α-β pruning is forced to search the entire tree to prove no win exists, rendering it unable to cut off branches. Memoisation excels here by recognising that distinct move orders lead to identical drawn states, preventing the re-evaluation of the same deep subtrees.