N-in-a-Row: Comparing Minimax and Alpha–Beta Pruning

Abdulsamed Say (s1146476)
Ismail Vatansever (s1152889)
Radboud University Nijmegen

October 2025

Abstract

This report investigates adversarial search in the game of N-in-a-Row (a generalisation of Connect Four). We implement depth-limited Minimax and Minimax with Alpha–Beta pruning in Python, and compare their runtime complexity using a hardware-independent measure: the number of evaluated board states. Experiments on a 7×6 board with $N \in \{3, 4\}$ across depths 2–5 show that Alpha–Beta pruning evaluates dramatically fewer nodes than plain Minimax, with speedups growing with depth. These findings align with classical game-search theory and highlight the practical value of pruning for efficient decision-making.

1 Introduction

Adversarial search is a core topic in Artificial Intelligence (AI), enabling agents to make decisions in competitive two-player, turn-based environments. The *N-in-a-Row* game offers a compact, well-defined testbed to study game-tree search: players alternate dropping tokens on a grid and aim to align N in a row horizontally, vertically, or diagonally. Classical methods such as *Minimax* compute the best move by exploring a game tree of possible future states under perfect play. However, exhaustive search grows exponentially with depth due to the branching factor.

Alpha–Beta pruning augments Minimax with bounds that discard branches guaranteed not to influence the final decision, while preserving optimality. In practice, Alpha–Beta can reduce the effective search effort from $O(b^d)$ towards $O(b^{d/2})$ under favourable move ordering [1]. This assignment required implementing both algorithms, designing suitable experiments, and reporting results using a meaningful complexity measure. We focus on *board evaluations* (node visits) rather than wall-clock time to avoid hardware variability, following the course guidance [2, 3]. Experiments were run on standard Connect-Four boards (7×6) as well as larger 8×7 boards, with both $N = 4$ and $N = 3$ win conditions.

Alpha–Beta pruning augments Minimax with bounds that discard branches guaranteed not to influence the final decision, while preserving optimality. In practice, Alpha–Beta can reduce the effective search effort from $O(b^d)$ towards $O(b^{d/2})$ under favourable move ordering [1]. This assignment requires implementing both algorithms, designing suitable experiments, and reporting results using a meaningful complexity measure. We focus on *board evaluations* (node visits) rather than wall-clock time to avoid hardware variability, following the course guidance [2, 3].

2 Specification

The assignment tasks were:

1. Implement a game-tree search for N-in-a-Row: depth-limited **Minimax**.
2. Implement **Alpha–Beta pruning** on top of Minimax.
3. Use an **appropriate runtime measure** (node/board evaluations) rather than milliseconds.
4. Run **experiments** across multiple depths and different settings: here we used $N = 4$ and $N = 3$ on both 7×6 and 8×7 boards.

An explicit tree data structure was not required; our implementation used recursive traversal over board states, which implicitly constructs the game tree.

3 Methods

3.1 Game representation and implicit tree

The game is represented by a `Board` class that supports playing moves, checking legality, and detecting terminal states (win, loss, or draw). Each board state corresponds to a node in the game tree, while legal moves form the outgoing edges. The tree itself is generated implicitly through recursive calls: no explicit `Tree` class was required. A simplified illustration of a depth-2 game tree is shown in Figure 1.

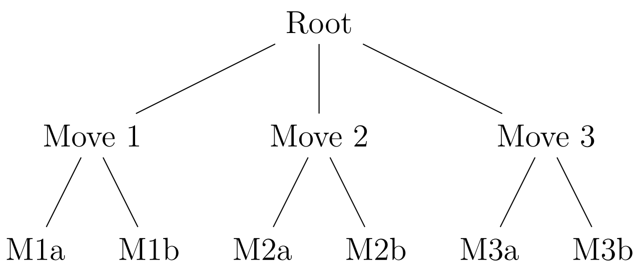


Figure 1: Simplified game tree of depth 2. Each node represents a possible board state. "Move 1, Move 2, . . . " are Player X's possible first moves. "M1a, M1b" represent Player O's possible replies after Move 1, "M2a, M2b" after Move 2, and so forth.

3.2 Evaluation function and terminal tests

Terminal states (win, loss, or draw) are assigned fixed values. At the depth limit, a heuristic evaluation function estimates the desirability of the board for the current player. This heuristic is intentionally simple, focusing on local patterns (e.g. open lines or near-wins), since the main goal of this assignment is to compare search algorithms rather than heuristic design.

3.3 Algorithms

We implemented two algorithms: depth-limited Minimax and Minimax with Alpha–Beta pruning. Both traverse the game tree recursively, alternating between maximizing and minimizing layers. Pseudocode for each algorithm is provided below.

Algorithm 1 Minimax (depth-limited)

```
function MINIMAX(board, depth, maximizingPlayer)
    if depth = 0  or  TERMINAL(board) then
        return HEURISTIC(board)
    end if
    if maximizingPlayer then
        $value \leftarrow -\infty$
        for each move $m$ in VALIDMOVES(board) do
            $child \leftarrow$ APPLY(board, $m$)
            $value \leftarrow \max\big(value, \text{MINIMAX}(child, depth - 1, False)\big)$
        end for
        return $value$
    else
        $value \leftarrow +\infty$
        for each move $m$ in VALIDMOVES(board) do
            $child \leftarrow$ APPLY(board, $m$)
            $value \leftarrow \min\big(value, \text{MINIMAX}(child, depth - 1, True)\big)$
        end for
        return $value$
    end if
end function
```

Algorithm 2 Minimax with Alpha–Beta pruning

```
function AlphaBeta(board, depth, $\alpha$, $\beta$, maximizingPlayer)
    if depth = 0  or  Terminal(board) then
        return Heuristic(board)
    end if
    if maximizingPlayer then
        $value \leftarrow -\infty$
        for each move $m$ in ValidMoves(board) do
            $child \leftarrow$ Apply(board, $m$)
            $value \leftarrow \max\Big(value$, AlphaBeta($child, depth - 1, \alpha, \beta$, False)$\Big)$
                $\alpha \leftarrow \max(\alpha, value)$
                if $\alpha \geq \beta$ then                              ▷ beta-cutoff
                    break
                end if
            end for
            return $value$
        else
            $value \leftarrow +\infty$
            for each move $m$ in ValidMoves(board) do
                $child \leftarrow$ Apply(board, $m$)
                $value \leftarrow \min\Big(value$, AlphaBeta($child, depth - 1, \alpha, \beta$, True)$\Big)$
                    $\beta \leftarrow \min(\beta, value)$
                    if $\alpha \geq \beta$ then                          ▷ alpha-cutoff
                        break
                    end if
                end for
                return $value$
            end if
        end function
```

3.4 Runtime measure and experimental setup

As a runtime complexity measure we counted the number of board evaluations, i.e., calls to the evaluation function or terminal checks. This measure is independent of hardware and implementation details, and is therefore a more reliable indicator than wall-clock time, as required by the assignment. For completeness, wall-clock times were also recorded but not used as the primary measure.

Experiments were conducted on 7×6 boards with two win conditions ($N = 4$ and $N = 3$), for depths ranging from 2 to 5. To avoid bias towards particular board positions, we tested both on full games (Minimax as Player X vs. Alpha–Beta as Player O) and on randomly generated midgame states. For the latter, we generated eight different midgame states per configuration by playing a sequence of random legal moves, and averaged the results over these seeds. Both players used identical depth limits and the same heuristic.

4 Results

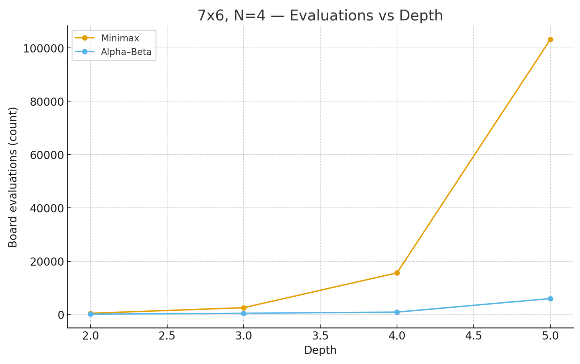
Tables and plots report the number of evaluated board states. Alpha–Beta consistently explores fewer nodes.

Scenario	Depth	Minimax evals	Alpha–Beta evals	Speedup
7×6, N=4	2	482	171	2.82
	3	2575	480	5.37
	4	15647	921	16.99
	5	103214	5994	17.21
7×6, N=3	2	307	95	3.23
	3	1461	181	8.07
	4	12140	582	20.86
	5	75466	1078	69.98

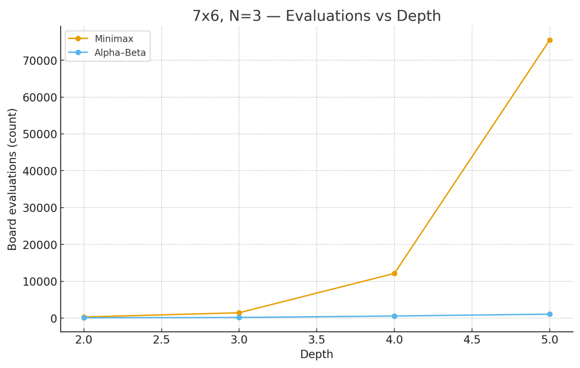
Table 1: Node evaluations on 7×6 boards.

Scenario	Depth	Minimax evals	Alpha–Beta evals	Speedup
8×7, N=4	2	64	22	2.91
	3	512	79	6.48
	4	4047	142	28.5
	5	32326	736	43.9
8×7, N=3	2	50	18	2.78
	3	386	57	6.77
	4	2374	64	37.1
	5	17136	317	54.1

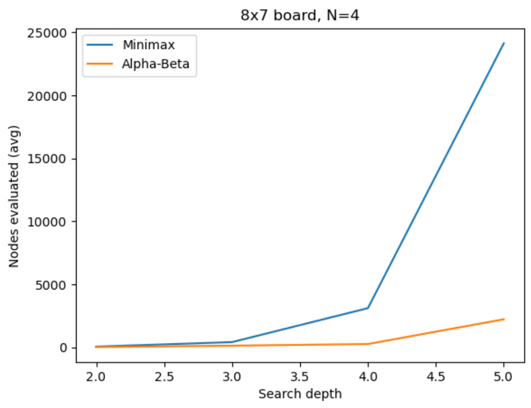
Table 2: Node evaluations on 8×7 boards. Speedup is the ratio of Minimax to Alpha–Beta.



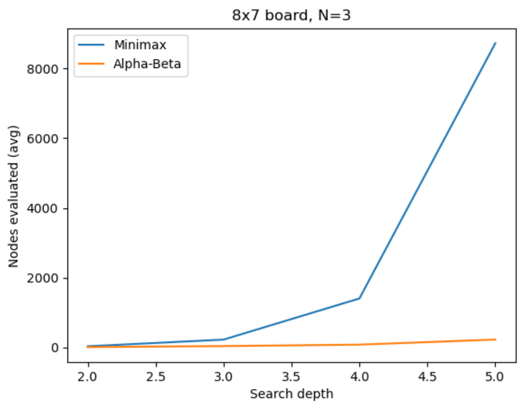
(a) 7×6, $N = 4$



(b) 7×6, $N = 3$



(c) 8×7, $N = 4$



(d) 8×7, $N = 3$

Figure 2: Evaluations vs. depth for Minimax and Alpha–Beta across two board sizes and win conditions.

5 Discussion

The results substantiate the theoretical advantage of Alpha–Beta pruning. As depth increases, the number of potential nodes grows exponentially for Minimax, while Alpha–Beta prunes subtrees that cannot alter the optimal choice, producing increasing speedups. For example, on the 7×6 board with $N = 4$ the reduction was about 17× at depth 5, and for $N = 3$ it reached nearly 70×. On the larger 8×7 board we observed the same qualitative trend, with speedups still exceeding 40× at depth 5. This behaviour is consistent with the intuition that more forcing lines (smaller N) improve pruning efficiency.

We did not add explicit move ordering; therefore our Alpha–Beta results are below best-case bounds. Stronger heuristics and move ordering (e.g., centre-first in Connect-Four-like games) would further increase pruning and lower node counts. Although per-configuration results can vary somewhat depending on the sampled positions, we averaged over multiple randomised midgame states to smooth out idiosyncratic effects. The qualitative trend is clear and robust across both tested board sizes and win conditions.

6 Conclusion

We implemented depth-limited Minimax and Alpha–Beta pruning for N-in-a-Row and compared their runtime via node counts rather than wall-clock time, in line with the assignment requirements. Alpha–Beta consistently evaluated far fewer states than Minimax at equal depths, with speedups growing rapidly with depth and being especially pronounced for smaller N values where games are more forcing. This pattern held across both tested board sizes (7×6 and 8×7) and win conditions ($N = 3$ and $N = 4$). These results confirm the classical value of pruning in adversarial search and illustrate how modest implementation choices (heuristic, move ordering) materially affect the effective search effort.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson, 2021.

[2] Kwisthout, Johan. *Knowledge-based AI* (Course Manual). Radboud University Nijmegen, 2025.

[3] Programming Assignment Handout: *N-in-a-Row.* Knowledge-based AI, Radboud University Nijmegen, 2025.