AIPT - Four In A Row

Chiara Benini

June 2025

1. Introduction

This project explores practical applications of search algorithms through the implementation of an algorithm for the game of N in a Row. The goal was to compare two versions of the MinMax algorithm: a standard implementation and one enhanced with alpha-beta pruning. The comparison focuses on their runtime performance across different board sizes, values of N, and search depths. The project involved building a tree structure for game states, implementing both algorithms, and measuring the computational cost of each methods.

2. Minmax and Alpha-Beta

The **MinMax algorithm** is a decision rule used in two-player turn-based games. It evaluates game states by simulating all possible moves, storing them inside a tree structure and assuming both players play optimally; one trying to maximize the score and the other to minimize it. However, this exhaustive search can become computationally expensive as the number of possible moves grows, up to being almost impossible for higher levels of depth.

Alpha-beta pruning is an optimization that reduces the number of nodes the MinMax algorithm evaluates. It does this by "**pruning**" branches that cannot influence the final decision, based on two values: alpha (the best already explored option for the maximizer) and beta (the best for the minimizer). When a node is found to be worse than a previously explored option, it's skipped entirely. This technique can significantly improve efficiency without affecting the final outcome of the search.

3. Explanation of implementation

3.1 Tree Structure

To implement these algorithms, we need something to store the board states and to record which moves have been made. I found that making this as a separate part of the code made it easier to handle and to pass it on different algorithms without having to rely on overly complicated code. So, the tree structure is used to represent possible moves in the game. The information stored in the tree includes:

- The current board state.
- The player who made the last move.
- The move that led to this state.
- The depth (how many moves ahead the tree explores).
- A list of child nodes (possible next moves).

The `build_tree` function recursively generates child nodes by simulating all valid moves from the current board state. Each child node represents a possible move, and the tree expands until a stopping condition is met (either the game ends or the maximum depth is reached → if root.depth == 0 or winner != 0: return). The tree function is then called inside different algorithms, inside the `make_move()` function, to set up the board and manipulate it while the code runs → `build_tree(root_node)`.

3.2 MinMax Algorithm

MinMax is a decision-making algorithm that assumes both players play optimally. It evaluates all possible moves up to a certain depth and chooses the best one. It's set up inside the `make_move()` function in the MinMaxPlayer class, here we construct the game tree and apply the algorithm to the node structure. The function then evaluates terminal nodes by assingning them scores for win, loss or draw and once the node reaches the maximum search depth we use the heuristic function to estimiate its value `(self.heuristic.evaluate_board(self.player_id, node.board)`. By definition the minmax alternates between maziminzing and minimizing the player's score, depending who's turn it is and after evaluating the whole tree it picks the best possible move.

The **drawback** in this case is that by evaluating all possible moves, it will inevitably be computationally expensive, especially for higher depths level. This is why alpha beta pruning is necessary.

3.3 Alpha-Beta Pruning

The **AlphaBetaPlayer** class, like the MinMax one, implements an agent that plays a game of connect-N using the optimized version of the previous 'vanilla' MinMax algorithm, pruning branches that would not affect the final choice before they can become too computationally expensive. It works similarly to the MinMax, initializing a tree structure and applying the algorithm logic to evaluate all its nodes, maximaxing the player score and minimizing the opponent's.

If a move is found that is worse than the current best (for either player), the algorithm **stops** evaluating further moves in that branch (since they won't change the outcome) → `if beta <= alpha:  break`

If applied correctly it gives the same result as MinMax, but way faster and it dramatically reduces the number of nodes evaluated, making it faster, which is especially relevant for deeper searches.

3.4 Monte Carlo Algorithm

Monte Carlo is a different algorithm altogether which uses a probabilistic approach that simulates random games to estimate the best move.

1. Random Playouts:
   (a) For each possible move, it simulates many random games (playouts) where moves are chosen randomly until the game ends.
   (b) Starts from a given move (root_col).
   (c) Players alternate making random moves (`pick = random.choice(valid)`).
   (d) The game continues until a terminal state (win/loss/draw) is reached.
   (e) The result (winner) is returned to update win counts for the root move.
2. Best Move Selection:
   - The move with the highest win rate is chosen.

The **advantages** are that it doesn't require a heuristic (works even with no domain knowledge), and that it's good for games with high branching factors where exhaustive search is impractical. On the other hand, it requires many simulations for accuracy and its **less precise** than MinMax/AlphaBeta since it only examines a random samples of gameplays.

My implementation represents a simplified version of the Monte Carlo algorithm since the complete one [2] would require more steps (selection, expansion, rollout and backpropagation). Instead I just simulate many random playouts and choose the move with the highest win rate. This collection of wins might look similar to backpropagation, but no tree is updated.

3.5 App.py

The game flow is controlled from **app.py** where the paramenters for the game are set including the value for N and the board size. Here the players also get initialized using the **get_players()** function which interacts with the user, prompting them into selecting the player's type (Human, MinMax, AlphaBeta or MonteCarlo). If necessary the user is also asked to set the depth value they want to implement for that specific game. nce the players are chosen, `start_game()` handles the main game loop. It alternates between the two players, asking each for their next move by calling their `make_move()` method. Each player determines the best move based on their strategy and the current board state. After a valid move is made, the program checks for a winner using the optimized and compiled `winning()` function, which efficiently checks vertical, horizontal, and diagonal lines for a victory or a draw.

When the game ends, the final board state is printed, the result (win/draw) is announced, and the number of evaluated board states for each player is displayed. This modular setup makes it easy to add or modify player strategies and supports fair and interactive matches between various AI methods or human participants.

4. Complexity Study

4.1 Overview

In terms of time complexity, we first have the **MinMax algorithm**, with $O(b^d)$, where b is the branching factor (moves per turn) and d is the search depth, which becomes almost impossible to compute with higher depths depending on the game structure. If you apply **Alpha-Beta pruning** we can have a best-case scenario of $O(b^{d/2})$**,** or again $O(b^d)$ if no pruning occurs. For **Monte Carlo Tree Search** we have $O(n \cdot k)$, where n is the number of playouts and k is the average game length, which is way better than the previous ones, but of course the trade-off is that it's an inherently stochastic method.

4.2 Methods

To study how the algorithms behave in my code specifically, I made another section dedicated solely to measure runtime for the various functions across different board sizes, Ns, and depths.

To do this I made a separate .py file where I set up a function that **repeats the move selection process multiple times** (`num_trials=5`) for a given player (either `MinMaxPlayer` or `AlphaBetaPlayer`) on an empty board and that returns the average time and average number of evaluated board states. I then run this fucntion with different board sizes, different N values and with depths from 2 to 6.

One of the values I'm monitoring is called '**Speedup**' and it's a measure of how much faster the Alpha-Beta pruning version of the algorithm is compared to the plain MinMax version. I calculated it by doing a simple ratio of MinMax Time/AlphaBeta Time: if the result is 1, then the algorithms take the same amount of time, while if its > 1 it means that AB was faster. Speedup = 2 means that AB was twice as fast as MinMax.

```
start = time.time()
player.make_move(board)
end = time.time()
total_time += (end - start)
total_evals += player.get_eval_count()
```

Results (time, evaluations, speedup) are saved to `benchmark_results.csv` and plotted with pandas and matplotlib, which are very useful for handling databases.

4.3 Results

These are all the results organized in a table for better visualization:

Board Size	N	Depth	MinMax Time (s)	AlphaBeta Time (s)	Speedup	MinMax Evals	AlphaBeta Evals
(6x6)	3	2	0.151212	0.000409	370.03	36	11
(6x6)	3	3	0.003253	0.002441	1.33	216	41
(6x6)	3	4	0.021983	0.012572	1.75	1296	76
(6x6)	3	5	0.185741	0.120728	1.54	7776	848
(6x6)	3	6	1.060991	0.682864	1.55	46656	436
(6x6)	4	2	0.000783	0.000488	1.6	36	11
(6x6)	4	3	0.004564	0.002241	2.04	216	41
(6x6)	4	4	0.050907	0.018321	2.78	1296	76
(6x6)	4	5	0.330344	0.198214	1.67	7776	848
(6x6)	4	6	1.061218	0.677747	1.57	46656	1553
(7x6)	3	2	0.000828	0.000536	1.54	49	13
(7x6)	3	3	0.004984	0.003329	1.5	343	55
(7x6)	3	4	0.047632	0.025496	1.87	2401	103
(7x6)	3	5	0.341675	0.2515	1.36	16807	1580
(7x6)	3	6	2.568954	1.773439	1.45	117649	691
(7x6)	4	2	0.001064	0.000648	1.64	49	13
(7x6)	4	3	0.00549	0.003186	1.72	343	55
(7x6)	4	4	0.052846	0.067926	0.78	2401	103
(7x6)	4	5	0.363929	0.260193	1.4	16807	1580
(7x6)	4	6	2.666038	1.792118	1.49	117649	2954
(8x7)	3	2	0.001069	0.000686	1.56	64	15
(8x7)	3	3	0.007689	0.004807	1.6	512	71
(8x7)	3	4	0.084557	0.065045	1.3	4096	134
(8x7)	3	5	0.70817	0.462502	1.53	32768	2719
(8x7)	3	6	6.248792	3.904074	1.6	262144	1030
(8x7)	4	2	0.001809	0.001204	1.5	64	15
(8x7)	4	3	0.012781	0.00594	2.15	512	71
(8x7)	4	4	0.173308	0.078457	2.21	4096	134
(8x7)	4	5	0.762255	0.468802	1.63	32768	2719
(8x7)	4	6	6.326735	3.763607	1.68	262144	5155

From these results we can see that in almost all configurations, **AlphaBeta is faster than MinMax**. The Speedup is consistently above 1 for all but one data point, confirming that AlphaBeta is more efficient. At depth 2 on a (6x6), N=3 board, speedup is ∼370x, which is an **outlier**. This is likely because MinMax is doing unnecessary evaluations or maybe its due to some error in the code that I did not catch. In addition to the outlier, the speedup values range from 1.3 to 2.8 across depths and board sizes, so it is generically pretty stable and proves that AlphaBeta is **consistently** faster. As the board size increases from (6x6) to (8x7), MinMax runtime increases significantly, as expected. AlphaBeta also becomes slower, but the runtime increase is much less dramatic and the speedup stays consistent in proportion.

Example: At depth 6, (8x7) MinMax takes ∼6.3s vs AlphaBeta ∼3.7s (Speedup ≈ 1.68). For the same depth on (6x6), the speedup is ∼1.57, suggesting that AlphaBeta's pruning remains effective even as the problem size increases.

4.4 Visualization

For simplicity and to have more informative plots I picked a single board size and N value to make plots with. I chose the same ones that were used in the code, which are (7x6) with N=4.

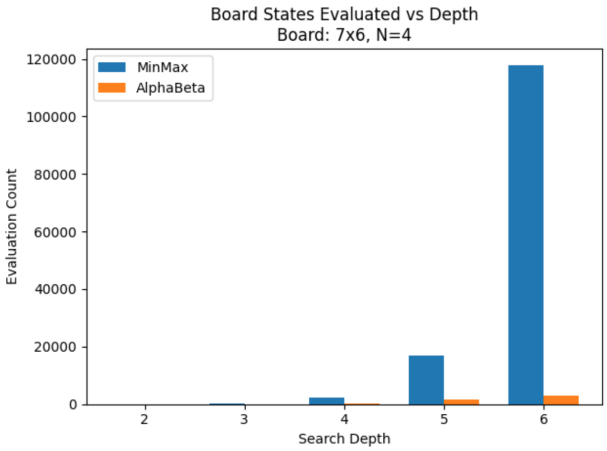

Figure 1: Node evaluation comparison between MinMax and AlphaBeta at varying depths (7x6 board, N=4)

Figure 1 demonstrates the core advantage of alpha-beta pruning:

- **Exponential Growth**: MinMax evaluations (orange bars) show the expected $O(b^d)$ explosion, with evaluations increasing from 49 at depth 2 to 117,649 at depth 6.
- **Pruning Effectiveness**: AlphaBeta (blue bars) maintains significantly lower counts, particularly at depth 6 where it evaluates only 2,954 nodes versus MinMax's 117,649 - a 40x reduction that directly reflects the $O(b^{d/2})$ best-case behavior.
- **Implementation Validation**: The divergence at higher depths confirms our pruning implementation is working as intended - ineffective pruning would show parallel growth curves.

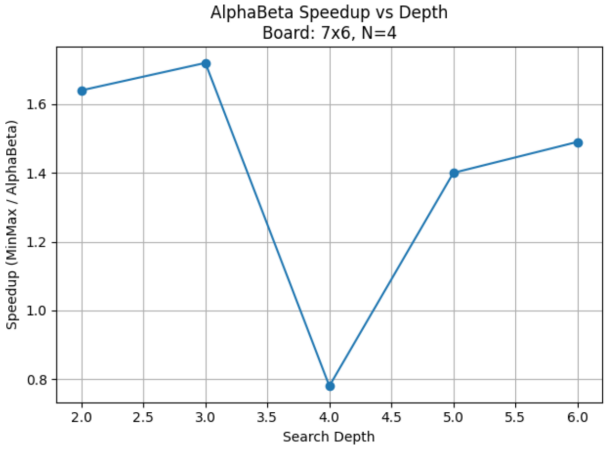

Figure 2: Speedup ratio of AlphaBeta versus MinMax across search depths

Figure 2 reveals practical performance characteristics:

- **Consistency**: Speedup remains bigger than 1 across all depths, averaging 1.5x, peaking at 1.72x (depth 3). This matches theoretical expectations
- **Anomaly at Depth 4**: there's one dip in the speedup ratio which is definitely an outlier and its probably due to some mistake in my code or some variance in the trials made by the program to measure the data

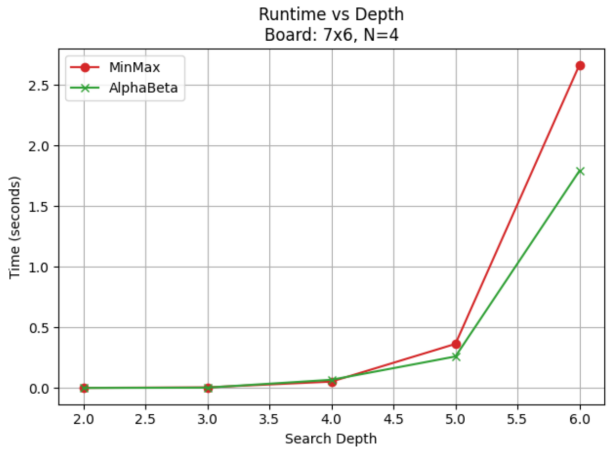

Figure 3: Runtime Comparison of AlphaBeta and MinMax over Depth

Figure 3 finally is the most straightforward of the plots, showing the correlation between dearch depth and runtime. The results are quite self-explanatory as we can see the exponential increase in time as we increase the depth value.

5. Conclusion

This project demonstrated the practical benefits of several search algorithms in the game of Connect-N, first by themselves (MinMax and Monte Calro) and later with optimization (AlphaBeta). Playing the game through the user interface and switching between the different options for the player settings it's easy to see how it works, the speed and the success rate for these algorithms. It was also interesting to use different combinations for the two players, for example MinMax+AlphaBeta aginst Monte Carlo, and see the theoretical performance be mirrored by the practical. Through the complexity study it was obvious that the performances vastly differed and that depth heavily influenced these values, making it extremely time consuming at higher values. We could also see that there were a few outliers in the results, though I don't know for sure why that was, it would be interesting to try and figure that out to see if it was due an error in the code or if its to be expected when using these algorithms.

References

[1] M. Muens, *Minimax and Monte Carlo Tree Search*, https://muens.io/minimax-and-mcts/, accessed June 2025.

[2] Shehan Rathnayake, *Implementation of Monte Carlo Tree Search Algorithm for Connect 4 Game*, Medium, 2019, Available at: `https://medium.com/@shehan_rathnayake/implementation-of-monte-carlo-tree-search-algorithm-for-connect-4-game-89c9ae23858f`