

IMPLEMENTATION OF MINIMAX AND ALPHA-BETA ALGORITHMS IN CONNECT N

By:

Juan Navarro — s1097545

Luca Laber — s1089333

Table of Contents

INTRODUCTION.....	2
METHOD.....	2
Algorithm Explanations & Implementation.....	2
Minimax Algorithm Explanation:.....	2
Alpha-Beta Pruning Explanation:.....	3
COMPLEXITY.....	5
RESULT.....	5
DISCUSSION.....	5
CONCLUSION.....	5

INTRODUCTION

The goal of this assignment was to implement and compare two specific search algorithms, **Minimax** and **Alpha-Beta pruning**, in a game of **N in a row**. Both algorithms are used for decision-making in most two-player games where players take turns to optimize their chances of winning. The task involved using both of these algorithms to evaluate possible game states and find the optimal move based on the current board configuration.

Minimax explores all potential future moves and selects the one that maximizes the player's chances of winning while minimizing the opponent's options. The downside, Minimax can take a toll on your RAM as the number of possible moves increases (requires computer memory), particularly with larger board sizes and depth. This is where Alpha-Beta pruning comes in to optimize the Minimax algorithm by skipping parts of the game tree that cannot influence the outcome (pruning), resulting in fewer nodes being evaluated.

In this project, we implemented both **Minimax** and **Alpha-Beta pruning** and compared their performance in terms of:

- **Computational complexity**, which is measured by the number of nodes evaluated.
- **Memory Usage**, which is measured by tracking how much memory was used each move.

To provide insight into how the algorithms work we did some experiments. The experiments were conducted by varying key parameters like **board size** (width and height) and **search depth**. This comparison provided insights into the trade-offs between a brute-force approach (Minimax) and an optimized one (Alpha-Beta pruning), especially when applied to deeper searches and larger game boards.

METHOD

Algorithm Explanations & Implementation

Below, there is an explanation for both algorithms used in the project: Minimax and Alpha-Beta pruning. As stated before, these algorithms were crucial for decision-making in the Connect-N game, determining optimal moves by evaluating potential game states.

Minimax Algorithm Explanation:

Minimax is a decision-making algorithm used in game theory for two-player games. It explores all possible moves to determine the optimal strategy for a player.

Minimax Algorithm Implementation:

The **Minimax** algorithm recursively explores all valid future game states up to a given depth and uses a heuristic to evaluate the board state. The goal is to determine the best move for the maximizing player (current player) while minimizing the opponent's chances of winning, and the minimizing player (minimizes the maximizing players options).

Minimax Pseudocode:

```
function minimax(node, depth, maximizingPlayer):
    if depth == 0 or node is terminal:
        return heuristic value of node

    if maximizingPlayer: # Maximize player
        maxEval = -infinity
        for each child of node:
            eval = minimax(child, depth - 1, False)
            maxEval = max(maxEval, eval)
        return maxEval
    else: # Minimize player
        minEval = +infinity
        for each child of node:
            eval = minimax(child, depth - 1, True)
            minEval = min(minEval, eval)
        return minEval
```

As said before the Minimax algorithm alternates between **maximizing** and **minimizing** players, recursively evaluating future game states until the specified depth limit or a terminal game state (win/loss/draw) is reached. As it goes down each level, the algorithm will choose the best move for the current player based on the heuristic evaluation of future board states. It utilizes recursion to be able to simulate and evaluate the future states of the game.

Alpha-Beta Pruning Explanation:

Alpha-Beta pruning is an optimization for the Minimax algorithm. It reduces the number of nodes evaluated by "pruning" branches of the game tree that are guaranteed not to influence the final decision.

Alpha-Beta Pruning Implementation

Alpha-Beta pruning is an optimization technique applied to the Minimax algorithm to reduce the number of nodes that need to be evaluated. It does this by "pruning" branches of the game tree that cannot affect the outcome, allowing the algorithm to skip unnecessary evaluations.

Alpha-Beta pruning maintains two values during its execution:

- **Alpha (α):** The best score that the maximizing player can guarantee.
- **Beta (β):** The best score that the minimizing player can guarantee.

As the algorithm passes through each node of the tree, it updates these values. If it finds a move where beta is smaller than alpha, it cuts the branch because any further inspection is unnecessary (next movements will not be better, meaning a previous branch was better).

Alpha-Beta Pruning Pseudocode:

```
function alpha_beta(node, depth,  $\alpha$ ,  $\beta$ , maximizingPlayer):
    if depth == 0 or node is terminal:
        return heuristic value of node

    if maximizingPlayer:
        maxEval = -infinity
        for each child of node:
            eval = alpha_beta(child, depth - 1,  $\alpha$ ,  $\beta$ , False)
            maxEval = max(maxEval, eval)
             $\alpha$  = max( $\alpha$ , eval)
            if  $\beta$  <=  $\alpha$ :
                break # Prune remaining branches return maxEval
    else:
        minEval = +infinity
        for each child of node:
            eval = alpha_beta(child, depth - 1,  $\alpha$ ,  $\beta$ , True)
            minEval = min(minEval, eval)
             $\beta$  = min( $\beta$ , eval)
            if  $\beta$  <=  $\alpha$ :
                break # Prune remaining branches
    return minEval
```

The code alternates between maximizing and minimizing players, updating alpha for the maximizer and beta for the minimizer. When beta becomes less than or equal to alpha, the branch is pruned, saving computation time. The algorithm continues recursively until a specified depth or terminal game state is reached, where it evaluates the board using a heuristic function and returns the best move.

COMPLEXITY

To analyze better how both algorithms function, and to prove that alpha-beta pruning is more efficient, we perform a complexity study where we compare the number of nodes evaluated for both algorithms (in the same board and with the same move), and how much memory each algorithm uses with a single move:

Test Case	Algorithm	Nodes Evaluated	Memory Usage (KB)
<i>Board 7x6, Depth 3, N=4</i>	Minimax	400	4.1
<i>Board 7x6, Depth 3, N=4</i>	Alpha-Beta Pruning	76	4.1
<i>Board 7x6, Depth 5, N=4</i>	Minimax	19608	154.4
<i>Board 7x6, Depth 5, N=4</i>	Alpha-Beta Pruning	2020	5.96
<i>Board 7x6, Depth 7, N=4</i>	Minimax	944118	156.93
<i>Board 7x6, Depth 7, N=4</i>	Alpha-Beta Pruning	27719	155.79
<i>Board 8x8, Depth 3, N=5</i>	Minimax	585	4.76
<i>Board 8x8, Depth 3, N=5</i>	Alpha-Beta Pruning	95	4.76
<i>Board 8x8, Depth 5, N=5</i>	Minimax	37449	155.69
<i>Board 8x8, Depth 5, N=5</i>	Alpha-Beta Pruning	2791	6.95

The complexity test results highlight the differences in performance between the Minimax and Alpha-Beta pruning algorithms. In all the cases, Alpha-Beta pruning evaluates fewer nodes than Minimax, with the gap increasing as the depth increases. For instance, on a **7x6 board with N=4 and depth 7**, Minimax evaluates **944,118 nodes**, while Alpha-Beta pruning evaluates **27,719 nodes**, showing a substantially smaller difference in the number of nodes evaluated.

On the other hand, memory usage remains similar for both algorithms at lower depths, but Alpha-Beta pruning becomes much more efficient at deeper levels. For example, at **depth 5**, Alpha-Beta pruning uses **5.96 KB**, while Minimax uses **154.4 KB**. These findings show how much more efficient Alpha-Beta pruning is in reducing both node evaluations and memory usage, hence making it the best option for deeper and more complex game situations.

In terms of runtime, the Minimax algorithm evaluates all nodes up to the given depth, with a runtime of $O(b^d)$ in the best, worst, and average cases. Alpha-Beta pruning, on the other hand, significantly improves performance in the best case, reducing the runtime to $O(b^{(d/2)})$ by pruning irrelevant branches. In the worst case, where pruning is ineffective, its runtime reverts to $O(b^d)$, similar to Minimax. On average, Alpha-Beta

pruning typically achieves a runtime of around $O(b^{(3d/4)})$, making it more efficient than Minimax in most scenarios.

DISCUSSION

The tests that were run above confirm that Alpha-Beta pruning is significantly more efficient than Minimax, especially as the search depth increases. Alpha-Beta pruning evaluates fewer nodes while still maintaining better memory usage. This makes it the better option for deeper searches, thus confirming the fact that Minimax's exhaustive exploration of game states becomes computationally expensive. Despite these advantages, the performance of Alpha-Beta pruning can change depending on the evaluation of the moves made. In cases where bad moves are evaluated early, pruning is less effective, and performance then becomes extremely similar to Minimax's.

CONCLUSION

In conclusion, after implementing both algorithms and performing the complexity study, it is safe to say that the Alpha-Beta pruning algorithms are overall better. It not only works just as efficiently as minimax for lower depths but also increases its performance exponentially (as well as reducing considerably the RAM usage) for higher depths.

While most of this process went smoothly, there were some issues throughout. In the code there were errors that prevented us from having an accurate algorithm. After some time we found a way to bypass this error as it seemed to be coming from the template provided. The algorithm works and does its job, yet it is pertinent to mention what is said above.

ACKNOWLEDGEMENTS

On behalf of Group 44, we want to thank those who contributed to our work. The teacher, TAs for their guidance, and others who supported us with feedback in the

process. Special thanks to the content provided in brightspace, which helped us understand the algorithms, without which we would not have understood.