The MINIMAX Algorithm

Lecture Notes 3

Nadjib Lazaar
University of Paris-Saclay
lazaar@lisn.fr

The MINIMAX algorithm is a decision-making method used in two-player zero-sum games, where each player seeks to maximize their own gains while minimizing those of their opponent. It relies on a recursive exploration of the game tree to determine the best action at each step, assuming both players play optimally.

1 History and Background of the Minimax Algorithm

The Minimax algorithm has its roots in game theory, a field of applied mathematics. It was formalized in the 1920s and 1930s by the mathematician **John von Neumann**.

1.1 Origins and Early Work

The Minimax algorithm was introduced by **John von Neumann** in the 1920s. He laid the groundwork for game theory in his pioneering work, which established the foundations of the Minimax algorithm. In 1928, von Neumann proposed that two-player zero-sum games (where one player's gain is the other's loss) could be modeled and solved using optimal strategies.

1.2 Game Theory

Game theory was formally established with the publication of the book *Theory of Games and Economic Behavior* in 1944, co-authored by **John von Neumann** and **Oskar Morgenstern**. This work defined the framework of game theory as a standalone mathematical discipline, addressing strategic decision-making in competitive environments. Von Neumann demonstrated that zero-sum games could be solved with an optimal strategy, which corresponds to the Minimax algorithm.

1.3 Formalization of Minimax

The principle of Minimax is based on the idea that each player selects a strategy that minimizes the potential gain of the opponent while maximizing their own gain. This decision-making approach proved crucial for solving two-player games where each participant aims to optimize their chances while anticipating the opponent's moves.

1.4 Early Computer Programs

The Minimax algorithm was employed to solve two-player games such as chess and checkers. In 1951, the computer program *Turochamp*, developed by **Alan Turing** and **D. G. Champernowne**, was one of the first to use a rudimentary version of the Minimax algorithm to play chess. This marked an initial step in using AI to simulate complex strategy games.

1.5 Optimizations and Improvements

Over time, it became evident that the combinatorial explosion in large games made the Minimax algorithm impractical without enhancements. In the 1950s, **Allen Newell**, **Herbert A. Simon**, and **John McCarthy** introduced optimization techniques such as *alpha-beta pruning*, which reduces the number of nodes explored in the decision tree while guaranteeing the same result. This optimization made the algorithm more efficient, enabling it to handle larger decision trees.

1.6 Modern Applications

Today, the Minimax algorithm is still widely used in zero-sum games like chess and checkers. However, its principles have also inspired similar approaches in other fields of AI, including robotics, military strategy, and finance. Techniques derived from Minimax continue to be taught in AI and game theory courses.

1.7 Conclusion

The Minimax algorithm represents one of the earliest applications of game theory in computer science. From its initial applications in strategy games, it has evolved into a cornerstone of autonomous decision-making systems. Its development and subsequent improvements have driven significant advances in AI, and its legacy remains deeply embedded in modern technologies.

2 The Algorithm

The algorithm alternates between two types of steps:

- **MAX Steps**: Represent the decisions of the main player (Player 1), who seeks to maximize their gain.

Algorithm 1: MINIMAX Algorithm

Input: s: current state; d: max depth;
$player_1$: Indicates whether it is the maximizing player's turn.
Output: `bestValue`: The utility value for the current state.

```
1  begin
2      if isLeaf(s) ∨ d = 0 then
3          return EVAL(s)
4      if player_1 then
5          bestValue ← −∞
6          foreach s′ ∈ child(s) do
7              bestValue ← max(bestValue, MINIMAX(s′, d − 1, false))
8          return bestValue
9      else
10         bestValue ← +∞
11         foreach s′ ∈ child(s) do
12             value ← MINIMAX(s′, d − 1, true)
13             bestValue ← min(bestValue, MINIMAX(s′, d − 1, true))
14         return bestValue
15 Function EVAL(s):
16     if isLeaf(s) then return score(s)  // Return the exact score
17     value ← 0
18     foreach f ∈ feature(s) do
19         value ← value + cost(f)
20     return value
```

- **MIN Steps**: Represent the decisions of the opponent (Player 2), who seeks to minimize the main player's gain.

Each node in the game tree is evaluated using a utility function, and the node values are propagated to the root according to the following rules:

- A MAX node takes the maximum value among its children.
- A MIN node takes the minimum value among its children.

The MINIMAX algorithm works recursively by exploring future game states. If the current state is a leaf of the tree (end of the game) or the maximum depth d is reached, the evaluation function EVAL returns a value representing the utility of that state. When it is the maximizing player's turn ($player_1 = true$), the algorithm initializes a variable `bestValue` to $-\infty$, then evaluates each possible move and updates this value with the maximum result from recursive calls. Conversely, when it is the minimizing player's turn ($player_1 = false$), `bestValue` is initialized to $+\infty$ and updated with the minimum result from

recursive calls. The goal is to anticipate the opponent's moves to optimize decisions. The evaluation function EVAL can either provide an exact score for terminal states or estimate a value based on specific characteristics of the state:

- **Accuracy for leaf nodes:** When a state s is a leaf, it means the game has ended at that state (win, loss, or draw). In this case, the function EVAL(s) returns an exact score based on the game's rules, as the outcome is definitively determined. For instance, in a game like chess, a win might be represented by a high score, a loss by a low score, and a draw by a neutral score. These scores directly reflect the true value of the state for the maximizing player.
- **Estimation for non-terminal nodes:** When s is not a leaf, the game is not yet finished, and the outcome still depends on future decisions. In this case, the function EVAL(s) must provide an estimate of the quality of the state, often based on specific characteristics (heuristics). For example, in chess, this estimate might consider the number and quality of pieces, control of the center, or the safety of the king. These heuristics provide a reasonable approximation but do not guarantee perfect accuracy since they do not account for all possible move sequences from that state.

3 Complexity

The MINIMAX algorithm takes a depth d as a parameter, which limits the search to a certain height in the game tree. In the worst case, this depth d corresponds to the full height of the tree, meaning that all possible game states are explored down to the leaves. The time complexity of the algorithm is then given by $O(n^d)$, where n is the **branching factor**, i.e., the average number of possible actions at each step of the game, and d is the **maximum depth** explored. Thus, the larger n or d are, the greater the total number of states to explore grows exponentially. This rapid growth makes the algorithm time-consuming for games where the branching factor or depth are large, which can make it impractical without optimizations such as alpha-beta pruning.

4 Limitations and Extensions

Although the MINIMAX algorithm guarantees an optimal strategy when players play perfectly, it suffers from combinatorial explosion due to the size of the search tree. To overcome these limitations and improve its efficiency, several optimizations and extensions can be implemented:

- **Alpha-Beta Pruning**: This method reduces the number of nodes explored by eliminating branches that cannot affect the final result. It maintains two values (α for the best guaranteed score for the maximizing

player, and β for the minimizing player) and interrupts the exploration of unnecessary branches when the limits are exceeded, while maintaining an optimal strategy.

- **Move Sorting**: The order in which moves are explored can influence the efficiency, particularly with alpha-beta pruning. Sorting promising moves first maximizes pruning opportunities and reduces the necessary work.
- **Memoization (Transposition Table)**: By using a transposition table to store the results of already evaluated states, we avoid recalculating the same values multiple times in the search tree. This is particularly useful in games where states can frequently repeat, such as chess.
- **Endgame Databases**: Certain winning or drawn positions in the endgame can be precomputed and stored in a database. When a state corresponds to a known position, the algorithm can immediately consult the database to determine the best move.
- **Good Move Sequences**: Identifying and memorizing patterns or effective move sequences for certain game configurations helps guide the algorithm towards fast and accurate solutions, thus limiting exhaustive exploration.
- **Heuristics**: When the search tree is too deep, it is common to limit the search to a fixed depth d and use a heuristic function to evaluate intermediate nodes. This function can incorporate specific game characteristics, such as piece quality in chess or control of strategic areas on a board.
- **Pattern-Based Pruning**: Recognizing and exploiting frequent patterns or specific game configurations for pruning or rapid evaluation helps reduce complexity without affecting search accuracy.

These optimizations and extensions, when used together, make the MINIMAX algorithm more efficient by reducing the number of nodes explored and guiding the search toward the most promising solutions, thereby mitigating the limitations of combinatorial explosion.

5 Illustration of the MINIMAX Algorithm and the EVAL Function: Example on Tic-Tac-Toe

To illustrate how the MINIMAX algorithm works, let's consider the example of the game of Tic-Tac-Toe. We assume that two players, X (Player 1) and O (Player 2), alternate their turns. The game ends when a player wins or when there are no more available spaces.

State s_i: Consider the following intermediate board:

X	O	
	X	O

In this turn, X must play. The goal is to maximize the value of the game state by choosing an optimal move. The EVAL function is used to evaluate the quality of each possible state.

Definition of the EVAL Function

The EVAL function is based on two elements:

1. *Features* extracted from the board, with values assigned based on their importance.
2. A scoring function applied to the terminal leaves of the game.

These two elements are detailed in the tables below:

Feature	**Value**
Winning rows/columns/diagonals for X or O	+1
Almost winning rows/columns/diagonals for X	+3
Almost winning rows/columns/diagonals for O	-3
Center controlled by X	+2
Center controlled by O	-2

Table 1: Features used to evaluate an intermediate state in Tic-Tac-Toe

Terminal State (Leaf)	**Score**
Immediate victory for X	$+\infty$
Immediate victory for O	$-\infty$
Draw (full board without a winner)	0

Table 2: Scores assigned to terminal leaves in Tic-Tac-Toe

Rows/columns/diagonals are considered *winnable* if they contain only X's, O's, or are empty. A row is *almost winning* when it contains exactly two X's or two O's and no opposing pieces.

Analysis of State s_i

From the given board, we identify the present *features*:

1. **Winnable Rows:**
   - Row 3 (_, _, _) is winnable for X or O: +1.

2. **Winnable Columns:**
   - Column 1 $(X, \_, \_)$ is winnable for X or O: $+1$.
   - Column 3 $(\_, O, \_)$ is winnable for X or O: $+1$.
3. **Winnable Diagonals:**
   - Diagonal 2 $(\_, X, \_)$ is winnable for X or O: $+1$.
4. **Almost Winning Diagonals for X:**
   - Diagonal 1 $(X, X, \_)$: $+3$.
5. **Center Controlled:**
   - Central square controlled by X: $+2$.

Total EVAL Score for State s_i:

$$\text{eval}(s) = +1(\text{row 3}) + 1(\text{column 1}) + 1(\text{column 3})$$

$$+1(\text{diagonal 2}) + 3(\text{almost winning diagonal 1}) + 2(\text{center controlled}) = +9$$

Minimax Execution

X (Player 1, maximizing) examines the possible moves. Let's assume the available moves are:

- s_1: X plays at (3,3):

X	O	
	X	O
		X

  $\text{eval}(s_1) = +\infty$ (immediate win for X).

- s_2: X plays at (3,2):

X	O	
	X	O
	X	

  $\text{eval}(s_2) = +8$.

- s_3: X plays at (3,1):

X	O	
	X	O
X		

  $\text{eval}(s_3) = +13$.

Game	Branching Factor	Depth	Estimated Size
Tic-Tac-Toe	3	9	26,830
Connect Four	7	42	10^{14}
Checkers	8	50	10^{20}
Othello (Reversi)	10	60	10^{58}
Chess	35	80	10^{120} (Shannon's number)
Go	250	300+	10^{170}

Table 3: Examples of game tree sizes for different games

Optimal Choice for X: The algorithm chooses s_1 because it immediately maximizes the score by guaranteeing a win ($\text{eval}(s_1) = +\infty$).

6 Alpha-Bêta Pruning

The Alpha-Bêta algorithm, or Alpha-Bêta pruning, was designed to improve the efficiency of the Minimax algorithm, widely used for decision-making in combinatorial games such as chess. Its foundational ideas were first introduced in the 1940s and 1950s.

Formal mentions of Alpha-Bêta pruning first appeared in the work of **John McCarthy** and other pioneers of AI. However, the true development and formalization of the algorithm are attributed to two independent groups: **Allen Newell** and **Herbert Simon** on one side, and researchers like **Arthur Samuel** on the other, during the 1950s and 1960s.

The key idea is to prune (i.e., avoid exploring) branches of the decision tree that cannot influence the final result. This optimization significantly reduces the number of nodes evaluated, allowing deeper exploration with the same computational resources.

Over the decades, the Alpha-Bêta algorithm has played a central role in the development of AI-based game systems, including famous programs like IBM's Deep Blue. Its importance lies in its simplicity and efficiency, making it a standard for many two-player games.

6.1 The Alpha-Bêta Algorithm

The Alpha-Bêta algorithm is an optimization of the Minimax algorithm that significantly reduces the number of nodes to explore in the game tree. In practice, it prunes branches that cannot affect the final decision, thereby improving the efficiency of the Minimax algorithm while guaranteeing the same results.

The algorithm relies on two values, α and β, which serve as bounds to determine whether a branch can be pruned:

- α represents the maximum value the maximizing player (Player 1) is willing to accept.

- β represents the minimum value the minimizing player (Player 2) is willing to accept.

When the algorithm explores a node, it updates these bounds, and if the value of a node exceeds α or β, it is no longer necessary to explore its descendants, as they cannot influence the final decision.

6.2 Principle of the Alpha-Bêta Algorithm

The Alpha-Bêta algorithm follows a process similar to Minimax but with dynamic bounds that enable pruning irrelevant branches of the game tree. The process is as follows:

- When exploring a MAX node, if the value of a child exceeds β, the exploration of this subtree can be stopped (pruning), as the MIN player (Player 2) would never allow this value to be reached.
- When exploring a MIN node, if the value of a child is less than α, the exploration of this subtree can be stopped, as the MAX player would never accept this value.

Explanation of the Alpha-Bêta Algorithm

The Alpha-Bêta algorithm uses two bounds, α and β, to represent the best possible values for the maximizing player ($player_1$) and the minimizing player, respectively.

- α: The best value found so far for the maximizing player.
- β: The best value found so far for the minimizing player.

The algorithm recursively explores the game tree following these steps:

1. **Termination Condition**: If the current state s is a terminal state (e.g., a win, loss, or draw) or if the maximum depth d is reached, the function returns the evaluation value of the current state, provided by the Eval function.
2. **Maximizing Player's Turn**:
   - The best value (`bestValue`) is initialized to $-\infty$.
   - For each successor state s', the algorithm calls Alpha-Bêta recursively.
   - If the returned value is greater than or equal to β, pruning is applied because the minimizing player will never allow this branch to be chosen.
   - Otherwise, α is updated if a better choice is found.

3. **Minimizing Player's Turn**:

   - The best value (`bestValue`) is initialized to $+\infty$.
   - For each successor state s', the algorithm calls ALPHA-BÊTA recursively.
   - If the returned value is less than or equal to α, pruning is applied because the maximizing player will never allow this branch to be chosen.
   - Otherwise, β is updated if a better choice is found.

Advantages of Pruning ALPHA-BÊTA pruning avoids exploring unnecessary branches of the tree, reducing the number of nodes evaluated:

- In the best case (with properly ordered successors), the algorithm can reduce the complexity from $O(b^d)$ to $O(b^{d/2})$, where b is the branching factor and d is the depth.
- This allows deeper exploration of the tree with the same computational resources.

Result At the end, the algorithm returns the optimal utility value for the current state s, enabling the player to make the best possible decision based on the specified depth and pruning rules.

6.3 Complexity

The ALPHA-BÊTA algorithm improves the time complexity of the MINIMAX algorithm by pruning unnecessary branches of the game tree. The time complexity in the best case (optimal pruning) is $O(n^{\frac{d}{2}})$, where n is the branching factor and d is the maximum depth. In the worst case (no pruning), the complexity remains $O(n^d)$, which is equivalent to MINIMAX.

In practice, pruning significantly reduces computation time, especially for large game trees.

6.4 Limitations and Extensions

Despite its efficient pruning, the ALPHA-BÊTA algorithm remains sensitive to the combinatorial explosion in games with a large number of possibilities. However, it performs better than MINIMAX for large game trees.

Possible extensions of this algorithm include:

- **Adaptive Evaluation**: Using game-specific evaluation functions for large-scale games.
- **Parallel Optimization**: Leveraging parallel computing resources to explore different subtrees simultaneously.

Algorithm 2: ALPHA-BÊTA Algorithm

Input: s: current state; d: max depth maximale;
$player_1$: Indicates wheter it is the maximizing player's turn;
α: max value for player 1; β: min value for player 2.
Output: `bestValue`: The utility value for the current state.

```
1  begin
2      if isLeaf(s) ∨ d = 0 then
3          return EVAL(s)
4      if player_1 then
5          bestValue ← −∞;
6          foreach s' ∈ child(s) do
7              bestValue ← max(bestValue,
               ALPHA-BÊTA(s', d − 1, false, α, β));
8              if bestValue ≥ β then
9                  return bestValue;                    // prunning
10             if bestValue > α then
11                 α← bestValue;
12         return bestValue
13     else
14         bestValue ← +∞;
15         foreach s' ∈ child(s) do
16             bestValue ← min(bestValue,
               ALPHA-BÊTA(s', d − 1, true, α, β));
17             if bestValue ≤ α then
18                 return bestValue;                    // prunning
19             if bestValue < β then
20                 β← bestValue;
21         return bestValue
```