# Chapter 1: Adversarial Search and Alpha-Beta Pruning

Vlad

April 17, 2025

## 1 Introduction: Game Playing as Search

Many classic board games, such as Chess, Checkers, Go, and Reversi (also known as Othello), can be modeled as problems of search. These are typically two-player, zero-sum games of perfect information. "Perfect information" means that both players know the complete state of the game at all times (no hidden cards or dice rolls influencing the outcome beyond the players' choices). "Zero-sum" implies that one player's gain is exactly the other player's loss; there are no cooperative outcomes.

In such games, players alternate turns, making moves that transition the game from one state to another. The goal is to find a sequence of moves that leads to a winning terminal state. Since the opponent is also trying to win (and thus force the first player into a losing state), this involves reasoning about the opponent's potential counter-moves. This adversarial nature distinguishes game-playing search from standard path-finding search problems.

We can represent the game as a *game tree*, where nodes correspond to game states and edges represent possible moves. The root of the tree is the current game state. The children of a node are the states reachable by making one valid move. Terminal nodes represent the end of the game (win, lose, or draw).

Our objective is to develop an algorithm that can choose the "best" possible move from the current state, assuming the opponent also plays optimally. The foundational algorithm for this is Minimax. However, the size of game trees grows exponentially with the number of moves (the ply or depth), making exhaustive search infeasible for most non-trivial games. Alpha-Beta pruning is a crucial optimization that significantly reduces the search space without affecting the final outcome determined by Minimax.

In this chapter, we will first explore the Minimax algorithm, then delve into the mechanics and justification of Alpha-Beta pruning. We will analyze a concrete implementation of an Alpha-Beta bot for the game of Reversi, discuss the critical role of heuristic evaluation functions, and finally touch upon common extensions to enhance the performance of adversarial search algorithms. Reversi serves as an excellent testbed due to its relatively simple rules but complex strategic depth, making it suitable for illustrating these concepts as we progress towards more advanced techniques later in this book.

## 2 The Minimax Algorithm

The Minimax algorithm provides a formal way to determine the optimal move in a zero-sum game, assuming both players play perfectly. It explores the game tree recursively, calculating the utility of each state for the player whose turn it is.

### 2.1 Core Concepts

**Definition 1** (Game State)**.** *A representation of the board configuration and whose turn it is.*

**Definition 2** (Terminal State)**.** *A game state where the game has ended (e.g., win, loss, draw).*

**Definition 3** (Utility Function). *A function $U(s)$ that assigns a numerical value to a terminal state $s$. By convention, higher values are better for MAX and lower values better for MIN; in a zero-sum game, $U_{\text{MAX}}(s) = -U_{\text{MIN}}(s)$.*

**Definition 4** (Moves Function). *A function* $\text{Moves}(s, \text{player})$ *that returns the set of valid moves available to* `player` *in state $s$.*

**Definition 5** (Result Function). *A function* $\text{Result}(s, \text{move})$ *that returns the game state resulting from applying* `move` *in state $s$.*

## 2.2 The Minimax Value

The Minimax value of a state $s$, denoted $\text{Minimax}(s)$, is defined by:

$$\text{Minimax}(s) = \begin{cases} U(s), & \text{if } s \text{ is terminal,} \\ \max\limits_{m \in \text{Moves}(s,\text{MAX})} \text{Minimax}(\text{Result}(s,m)), & \text{if it is MAX's turn,} \\ \min\limits_{m \in \text{Moves}(s,\text{MIN})} \text{Minimax}(\text{Result}(s,m)), & \text{if it is MIN's turn.} \end{cases}$$

To choose the best move from the current state $s_0$, MAX selects

$$m^* = \arg \max_{m \in \text{Moves}(s_0,\text{MAX})} \text{Minimax}\big(\text{Result}(s_0, m)\big).$$

## 2.3 Algorithm Pseudocode

**Algorithm 1** Minimax Search

```
function MINIMAX-DECISION($state$)
    $best\_m \leftarrow \arg\max_{m \in \text{Moves}(state,\text{MAX})} \text{Min-Value}(\text{Result}(state, m))$
    return $best\_m$
end function
function MAX-VALUE($state$)
    if Terminal-Test($state$) then return Utility($state$)
    end if
    $v \leftarrow -\infty$
    for $m$ in Moves($state$, MAX) do
        $v \leftarrow \max\big(v, \text{Min} - \text{Value}(\text{Result}(state, m))\big)$
    end forreturn $v$
end function
function MIN-VALUE($state$)
    if Terminal-Test($state$) then return Utility($state$)
    end if
    $v \leftarrow +\infty$
    for $m$ in Moves($state$, MIN) do
        $v \leftarrow \min\big(v, \text{Max} - \text{Value}(\text{Result}(state, m))\big)$
    end forreturn $v$
end function
```

### 2.4 Limitations: Computational Complexity

The Minimax algorithm performs a complete depth-first exploration of the game tree down to the terminal states. If the maximum depth of the tree is $d$ and the average number of legal moves from any state (the branching factor) is $b$, the time complexity of Minimax is $O(b^d)$. The space complexity, due to the depth-first nature, is $O(bd)$ if we store states, or potentially less if states can be generated/retracted efficiently.

For games like Reversi (board size 8x8, potentially up to 60 moves deep) or Chess, $b$ and $d$ are large enough that $b^d$ becomes astronomically large, rendering a full Minimax search to terminal states computationally intractable. This necessitates optimizations or approximations.

## 3 Alpha-Beta Pruning

Alpha-Beta pruning is an optimization technique for the Minimax algorithm. It reduces the number of nodes evaluated in the game tree by eliminating branches that cannot possibly influence the final decision. Crucially, Alpha-Beta pruning guarantees to compute the *same* Minimax value as the full Minimax search, but often much faster.

### 3.1 The Core Idea: Pruning Useless Branches

Imagine MAX is evaluating its possible moves. Suppose MAX explores one move $m_1$ and finds that it guarantees a score of at least $V_1$ (because MIN's best response still leads to a state with value $V_1$). Now, MAX starts exploring another move $m_2$. During the exploration of $m_2$, MIN gets to play. If, at some point, MIN finds a response $m_{2,min}$ that leads to a state with value $V_2$, and $V_2 < V_1$, then MAX knows that choosing $m_2$ is already worse than choosing $m_1$. Why? Because MIN will force the outcome to be at most $V_2$ if MAX chooses $m_2$. Since MAX already has an option ($m_1$) that guarantees at least $V_1$ (where $V_1 > V_2$), MAX has no reason to further explore other responses MIN might have after $m_2$. The subtree under $m_2$ can be pruned below the point where $V_2$ was discovered.

Symmetrically, if MIN is evaluating its responses to a move by MAX, and finds one response $m_{min,1}$ leading to a value $V_1$, and then starts exploring another response $m_{min,2}$ under which MAX finds a move leading to value $V_2 > V_1$, MIN knows that MAX will achieve at least $V_2$ if MIN chooses $m_{min,2}$. Since MIN wants to minimize the score and already has an option ($m_{min,1}$) guaranteeing a score of at most $V_1$ (where $V_1 < V_2$), MIN has no reason to explore $m_{min,2}$ further. That branch can be pruned.

### 3.2 Alpha and Beta Values

To implement this pruning, we maintain two values during the search:

**Definition 6** (Alpha ($\alpha$))**.** *The best value (highest score) found so far for MAX along the path from the root to the current node. Initially $-\infty$.*

**Definition 7** (Beta ($\beta$))**.** *The best value (lowest score) found so far for MIN along the path from the root to the current node. Initially $+\infty$.*

The search proceeds recursively, passing $\alpha$ and $\beta$ down the tree.

- **At a MAX node:** The node tries to increase $\alpha$. If its current value $v$ ever becomes greater than or equal to $\beta$, it means MIN (at an ancestor node) already has a way to achieve a score of $\beta$ or lower. Since MAX is currently exploring a path that guarantees a score of at least $v \geq \beta$, MIN will never allow the game to reach this MAX node by choosing the corresponding move at the ancestor MIN node. Therefore, the remaining children of this MAX node need not be explored. We can prune the search and return the current value $v$. Otherwise, $\alpha$ is updated with $\max(\alpha, v)$.

- **At a MIN node:** The node tries to decrease $\beta$. If its current value $v$ ever becomes less than or equal to $\alpha$, it means MAX (at an ancestor node) already has a way to achieve a score of $\alpha$ or higher. Since MIN is currently exploring a path that guarantees a score of at most $v \leq \alpha$, MAX will never allow the game to reach this MIN node by choosing the corresponding move at the ancestor MAX node. Therefore, the remaining children of this MIN node need not be explored. We can prune the search and return the current value $v$. Otherwise, $\beta$ is updated with $\min(\beta, v)$.

The condition $\alpha \geq \beta$ is the core of the pruning mechanism. When this condition is met, the current subtree search can be terminated.

### 3.3 Algorithm Pseudocode

The Alpha-Beta algorithm modifies the Minimax functions to include $\alpha$ and $\beta$ parameters.

### 3.4 Effectiveness of Pruning

The efficiency of Alpha-Beta pruning heavily depends on the order in which moves are examined.

- **Best Case:** If the algorithm always explores the best move first (the one that leads to the true Minimax value), Alpha-Beta achieves significant pruning. For a game tree of depth $d$ and branching factor $b$, the number of nodes examined approaches $O(b^{d/2})$. This is effectively doubling the searchable depth compared to plain Minimax for the same computational effort.
- **Worst Case:** If the algorithm explores moves in the worst possible order (e.g., always exploring the weakest moves first), no pruning occurs, and the complexity remains $O(b^d)$, the same as Minimax.
- **Average Case:** In practice, with reasonably good move ordering heuristics, Alpha-Beta often performs much closer to the best case than the worst case.

This highlights the importance of *move ordering* heuristics, which we will discuss later.

## 4 Implementation in Reversi: The AlphaBetaBot

```python
"""
Alpha-Beta pruning bot for Reversi.
"""
import math

# Assuming 'bots.base.Bot' and 'game.opponent' are defined in the repository
# E.g., Bot is an abstract class with a 'move' method
# game contains the Reversi game logic (apply_move, get_valid_moves, etc.)
# opponent(color) returns the other player's color

class AlphaBetaBot(Bot):
    """Bot that uses alpha-beta pruning to choose moves."""
    def __init__(self, max_depth=3):
        self.max_depth = max_depth # Depth limit for the search

    def move(self, game, color):
        """Compute best move using alpha-beta search."""
```

**Algorithm 2** Alpha-Beta Search

```
function AlphaBeta-Search($state$)
    $v$, move $\leftarrow$ Max-Value($state$, $-\infty$, $+\infty$)
    return move                                   ▷ Return the best move found at the root
end function
function Max-Value($state$, $\alpha$, $\beta$)
    if Terminal-Test($state$) then return Utility($state$), null
    end if
    $v \leftarrow -\infty$
    $best\_move \leftarrow$ null
    for $m$ in Moves($state$, MAX) do
        $v_2, \_ \leftarrow$ Min-Value(Result($state$, $m$), $\alpha$, $\beta$)     ▷ Ignore move from lower levels
        if $v_2 > v$ then
            $v \leftarrow v_2$
            $best\_move \leftarrow m$
            $\alpha \leftarrow \max(\alpha, v)$
        end if
        if $v \geq \beta$ then                    ▷ Beta cutoff
            return $v$, $best\_move$
        end if
    end for
    return $v$, $best\_move$
end function
function Min-Value($state$, $\alpha$, $\beta$)
    if Terminal-Test($state$) then return Utility($state$), null
    end if
    $v \leftarrow +\infty$
    $best\_move \leftarrow$ null
    for $m$ in Moves($state$, MIN) do
        $v_2, \_ \leftarrow$ Max-Value(Result($state$, $m$), $\alpha$, $\beta$)     ▷ Ignore move from lower levels
        if $v_2 < v$ then
            $v \leftarrow v_2$
            $best\_move \leftarrow m$
            $\beta \leftarrow \min(\beta, v)$
        end if
        if $v \leq \alpha$ then                   ▷ Alpha cutoff
            return $v$, $best\_move$
        end if
    end for
    return $v$, $best\_move$
end function
```

```

    # Heuristic Evaluation Function (H): Estimates state value
    def evaluate(g):
        score = g.get_score() # e.g., {'BLACK': 30, 'WHITE': 34}
        # Simple heuristic: difference in piece count
        return score[color] - score[opponent(color)]

    # The core recursive Alpha-Beta function
    def ab_search(g, depth, maximizing_color, current_color, alpha,
        beta):
        # --- Base Cases ---
        # 1. Depth limit reached
        # 2. Game is over
        if depth == 0 or g.is_game_over():
            return evaluate(g), None # Return heuristic value, no
                move needed here

        moves = g.get_valid_moves(current_color)

        # --- Handling Pass Turn ---
        if not moves:
            g2 = g.clone()
            g2.current_player = opponent(current_color)
            if not g2.get_valid_moves(opponent(current_color)):
                return evaluate(g2), None
            return ab_search(g2, depth - 1, maximizing_color,
                opponent(current_color), alpha, beta)[0], None

        best_move = None

        # --- MAX Player Logic ---
        if current_color == maximizing_color:
            value = -math.inf
            for move in moves:
                g2 = g.clone()
                g2.apply_move(current_color, move[0], move[1])
                score, _ = ab_search(g2, depth - 1,
                    maximizing_color, opponent(current_color), alpha
                    , beta)

                if score > value:
                    value = score
                    best_move = move
                alpha = max(alpha, value)
                if alpha >= beta:
                    break
            return value, best_move

        # --- MIN Player Logic ---
        else:
            value = math.inf
            for move in moves:
                g2 = g.clone()
                g2.apply_move(current_color, move[0], move[1])
                score, _ = ab_search(g2, depth - 1,
                    maximizing_color, opponent(current_color), alpha
                    , beta)

                if score < value:
                    value = score
                    best_move = move
                beta = min(beta, value)
                if beta <= alpha:
                    break
            return value, best_move

    _, move = ab_search(game.clone(), self.max_depth, color, color,
        -math.inf, math.inf)
    return move
```

## 4.1 Analysis of the Implementation

1. **Class Structure:** The `AlphaBetaBot` class inherits from a base `Bot` class and takes `max_depth` as a parameter during initialization. This parameter controls how many moves ahead the bot searches.

2. `move` **Method:** This is the public interface of the bot. It clones the game state using `game.clone()` to avoid modifying the original game object passed to it. It then calls the internal `ab_search` function.

3. `evaluate` **Function (Heuristic):** This function is crucial because the search is depth-limited. When the maximum depth is reached (`depth == 0`) or the game ends, `ab_search` cannot recurse further. Instead, it calls `evaluate` to estimate the utility of the resulting game state for the `maximizing_color`. The provided implementation uses a simple heuristic: the difference between the number of pieces owned by the bot and by the opponent. This is often referred to as the "piece count" or "material" heuristic. While simple, it is typically a poor indicator of position strength in Reversi, especially during the mid-game. Better heuristics are discussed later.

4. `ab_search` **Function:** This is the core of the algorithm, implementing recursive Alpha-Beta logic:

   - **Parameters:**

     `g` Current game state.

     `depth` Remaining search depth.

     `maximizing_color` The bot's own color, fixed throughout the search.

     `current_color` Whose turn it is at this node.

     `alpha` Best score found so far for MAX.

     `beta` Best score found so far for MIN.
   - **Base Cases:** Checks whether the depth limit is reached or the game is over. If so, it returns the heuristic evaluation.
   - **Move Generation:** Retrieves the valid moves for the `current_color`.
   - **Handling Passes:** If no valid moves are available, the function simulates a pass by cloning the game, switching `current_player`, and making a recursive call to the opponent with `depth - 1`. It also handles the case where both players must pass (indicating the end of the game).
   - **MAX Node Logic (`current_color == maximizing_color`):** Initializes `value` to $-\infty$. For each move, it clones the game, applies the move, recursively calls `ab_search` for the MIN player, and updates `value` and `best_move` if a better score is found. It updates `alpha` and performs a beta cutoff if `alpha >= beta`.

- **MIN Node Logic (`else` case):** Symmetrical to the MAX case. Initializes `value` to $+\infty$, applies each move to a clone, recursively calls `ab_search` for the MAX player, updates `value` and `best_move` if a lower score is found, updates `beta`, and performs an alpha cutoff if `beta <= alpha`.
- **Return Value:** The function returns a tuple of the computed score (`value`) and the best move (`best_move`) found at that node. Only the best move is used by the top-level `move` method.

5. **State Cloning:** Using `g.clone()` is essential. Each recursive call explores a hypothetical future. Modifying the original game state directly would interfere with sibling branches in the search tree. Cloning ensures that each branch explores an independent copy of the game.

This implementation correctly captures the essence of depth-limited Alpha-Beta search with a pluggable evaluation function.

## 5 Heuristic Evaluation Functions for Reversi

As seen in the implementation, when the search cannot reach a terminal state (due to the depth limit), we need a way to estimate the value of non-terminal states. This is the role of the *heuristic evaluation function*. The quality of this function is paramount to the strength of the game-playing bot.

The simple piece difference heuristic (`score[color] - score[opponent(color)]`) used in the example code is often insufficient for strong Reversi play. Good heuristics should be:

1. **Fast to compute:** It is called at every leaf node of the search.
2. **Strongly correlated with win probability:** It should accurately reflect the long-term potential of the position.

Here are some common heuristic components used in Reversi bots, often combined using weights:

1. **Piece Count / Material:** The difference in the number of pieces. More useful towards the very end of the game, less so early on.
2. **Mobility:** The number of valid moves available to a player. Having more options (higher mobility) is generally advantageous as it restricts the opponent. Often calculated as:

$$H_{\text{mobility}} = |\text{Moves}(s, \text{MAX})| - |\text{Moves}(s, \text{MIN})|$$

3. **Corner Occupancy:** Corners (a1, a8, h1, h8) are strategically vital because they are stable (cannot be flipped once occupied).

$$H_{\text{corners}} = (\#\text{corners MAX}) - (\#\text{corners MIN})$$

4. **Edge Stability:** Pieces on edges can be stable or unstable. Stable edge pieces (e.g., next to an occupied corner) contribute to securing territory.
5. **Potential Mobility:** The number of empty squares adjacent to opponent pieces, representing potential future moves. Controlling this can be important for setting up future captures.

6. **Weighted Piece Squares (Positional Strategy):** Assigns static weights to each square. For instance, corners have high positive weights, adjacent squares (e.g., a2, b1, b2) may be negatively weighted, and central squares receive moderate weight. The heuristic is computed as:
$$H_{\text{positional}} = \sum_{i \in \text{Squares}_{\text{MAX}}} W(i) - \sum_{j \in \text{Squares}_{\text{MIN}}} W(j)$$
where $W(k)$ is the weight of square $k$.

A common approach is to use a weighted sum of several such features:

$$H(s) = w_1 H_{\text{material}}(s) + w_2 H_{\text{mobility}}(s) + w_3 H_{\text{corners}}(s) + \dots$$

The weights $w_i$ might vary depending on the stage of the game (e.g., mobility is more important early, while piece count matters more late). These weights are often chosen based on expert knowledge or learned using machine learning.

Replacing the simple `evaluate` function in `AlphaBetaBot` with a more sophisticated heuristic incorporating features like mobility and corner control would significantly improve its playing strength.

## 6 Extensions and Improvements

The basic Alpha-Beta algorithm with a fixed depth limit and a simple heuristic can be significantly enhanced. Here are some common extensions:

1. **Iterative Deepening Search (IDS):** Instead of choosing a fixed depth `d`, IDS starts by searching to depth 1, then depth 2, and so on, until a time or resource limit is reached.
   - **Anytime Algorithm:** Returns a valid move even if interrupted early.
   - **Improved Move Ordering:** The best move at depth $k$ often guides move ordering at depth $k+1$, improving Alpha-Beta pruning efficiency and approaching the $O(b^{d/2})$ best case.
2. **Enhanced Move Ordering:** Efficient pruning depends on evaluating strong moves early. Enhancements include:
   - **Heuristic Move Ordering:** Use fast heuristics to prioritize promising moves (e.g., capturing corners).
   - **Killer Moves:** Track moves that previously caused cutoffs and try them first in sibling nodes.
   - **History Heuristic:** Track moves that often yield good outcomes and prioritize them globally.
3. **Quiescence Search:** Addresses the "horizon effect" by extending the search at unstable positions (e.g., pending captures) beyond the depth limit until a stable ("quiet") state is reached. This improves evaluation accuracy.
4. **Transposition Tables (Memoization):** Game states can recur via different move sequences (transpositions). Storing their evaluations in a hash table allows reuse and pruning. Typically, keys are Zobrist hashes of game states, and stored values include the computed Minimax value, best move, and search depth.

## 7 Conclusion

The Minimax algorithm is the foundation for optimal play in two-player, zero-sum, perfect-information games. Alpha-Beta pruning significantly optimizes Minimax by discarding branches that do not influence the decision. Its practical effectiveness relies on good heuristics and move ordering.

The provided `AlphaBetaBot` demonstrates a depth-limited Alpha-Beta search for Reversi, incorporating alpha-beta bounds, recursive evaluation, and a simple heuristic.

Enhancements like iterative deepening, stronger heuristics, better move ordering, quiescence search, and transposition tables can substantially increase playing strength.

While Alpha-Beta remains central in classical AI for games, modern approaches—especially for vast or imperfect-information games—often use Monte Carlo Tree Search (MCTS) and Deep Reinforcement Learning. Understanding Alpha-Beta provides essential groundwork for these more advanced strategies.

## 8 Next Steps

1. Trace the execution of the Alpha-Beta algorithm on a small sample tree (e.g., depth 3, branching factor 2) with assigned terminal utilities. Show $\alpha$ and $\beta$ values at each node and mark pruned branches.

2. Implement an alternative heuristic for `AlphaBetaBot`, e.g., combining piece count with mobility. Let bots using different heuristics play against each other and compare performance.

3. Modify `AlphaBetaBot` to incorporate Iterative Deepening. The `move` method should run increasing-depth searches (1, 2, 3, ...) within a time limit and return the best move from the deepest completed search. Analyze the impact on performance.

4. Explain why `g.clone()` is essential in `ab_search`. What problems arise if `g.apply_move` is used directly on the original game state?

5. Research Zobrist hashing and explain how it enables efficient implementation of a transposition table. What should be stored in each table entry?