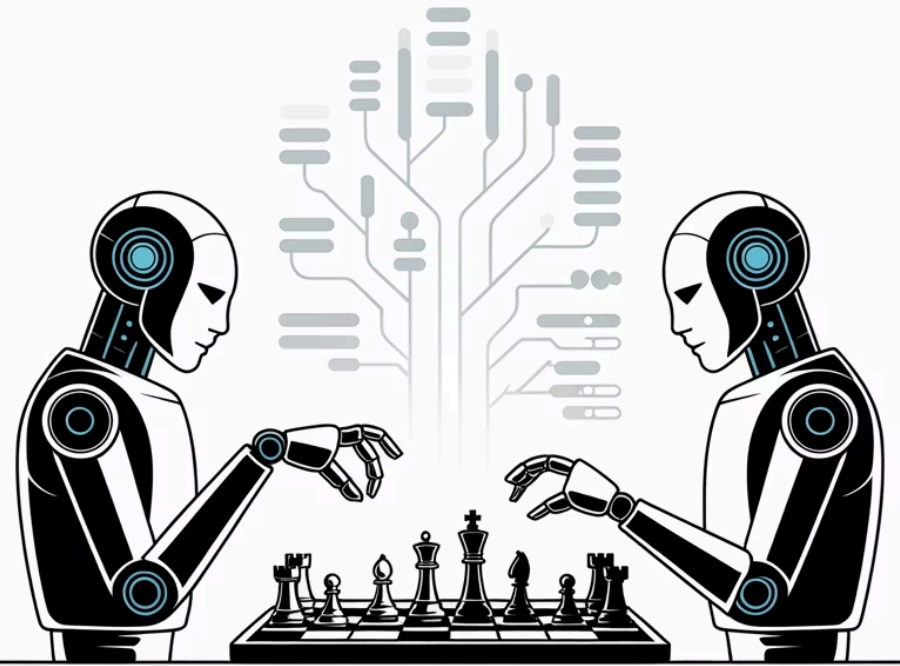




Adversarial Search in Competitive Environments

AI systems today face increasingly complex competitive scenarios where they must make optimal decisions against intelligent opponents. This presentation explores how adversarial search algorithms enable machines to navigate competitive environments—from classic board games to real-world applications in finance, security, and beyond.

Real-World Applications of Adversarial Search



Finance

Algorithmic trading systems anticipate competitor moves to maximize profits in securities markets. Adversarial models predict market reactions and optimize trade timing.



Cybersecurity

Defense systems model attacker strategies to identify vulnerabilities. Security teams simulate attack paths to strengthen defenses before real exploits occur.



Autonomous Vehicles

Self-driving cars predict other drivers' potential actions to navigate safely in complex traffic scenarios and unexpected road conditions.



Sports Analytics

Teams use adversarial models to simulate opponent strategies, optimize player positioning, and develop counter-strategies for competitive advantage.

What is a Competitive Environment?

A competitive environment in AI is characterized by:

- Multiple agents with conflicting goals and objectives
- Zero-sum or non-zero-sum reward structures
- Sequential or simultaneous decision making
- Actions by one agent directly affect outcomes for others
- Need to anticipate opponent strategy to optimize decisions



Chess exemplifies a classic competitive environment: two players with directly opposing goals, where each move changes the game state for both players. Every decision must consider not just immediate benefits but the opponent's potential responses.

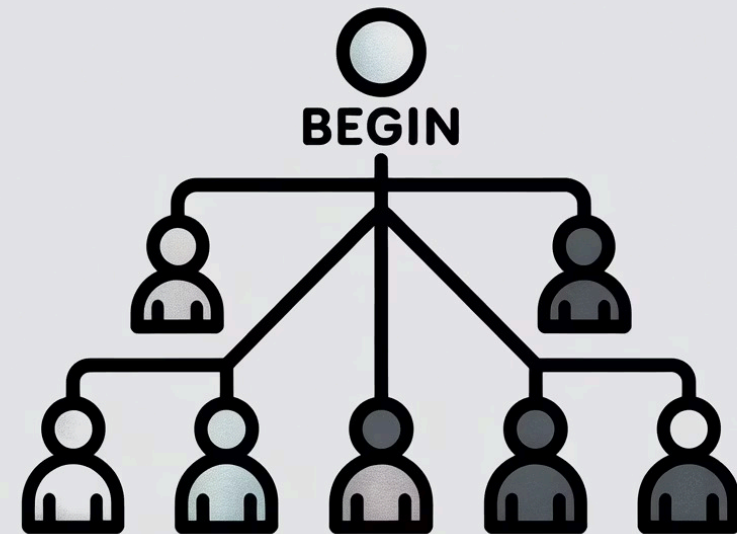
Game Trees: The Search Space

Understanding Game Trees

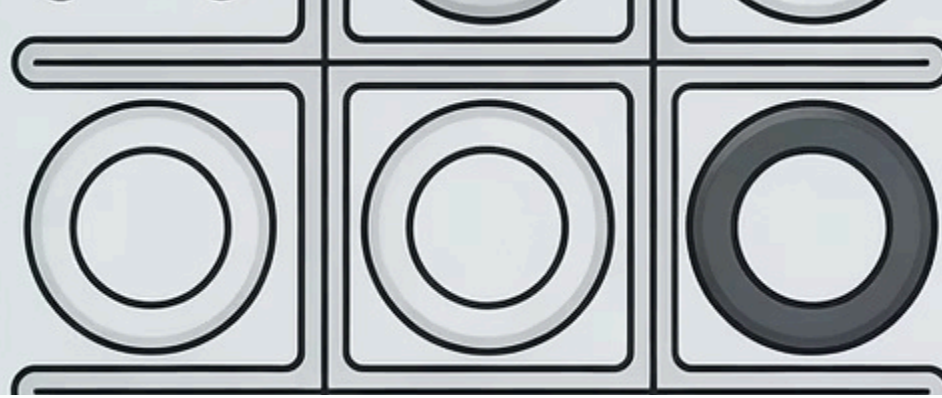
Game trees represent the complete search space of a competitive game:

- **Nodes:** Represent specific game states (e.g., board positions)
- **Edges:** Legal moves that transition between states
- **Alternating levels:** Represent turns by different players
- **Leaf nodes:** Terminal states with defined utility values

The depth of a game tree corresponds to the number of sequential turns, while the branching factor indicates the average number of legal moves available at each turn.



For even modestly complex games, complete game trees can be enormous. Chess has approximately 10^{120} possible states—far more than atoms in the observable universe!



Visualization: Example Game Tree (Tic-Tac-Toe)

Even in a simple game like Tic-Tac-Toe, the game tree expands rapidly:

- First move: 9 possible positions
- Second move: 8 remaining positions per first move (72 total states)
- Complete game tree: 255,168 possible game states
- With symmetry considerations: ~26,830 unique states

Imagine how quickly this grows for more complex games! Chess has an average branching factor of 35, and games typically last 80+ half-moves, making exhaustive search impossible.

Types of Games for Adversarial Search

Deterministic Games

No random elements; game progression depends solely on player decisions

Examples: Chess, Go, Checkers, Tic-Tac-Toe

Game state transitions are completely predictable based on player actions

Perfect Information Games

All players can observe the complete game state at all times

Examples: Chess, Go, Connect Four

No hidden information; every player knows exactly what every other player knows

Non-Deterministic Games

Include random elements (dice, cards) that affect game progression

Examples: Backgammon, Monopoly, Risk

Require probability assessments and expectiminimax algorithms

Imperfect Information Games

Players have private information not visible to opponents

Examples: Poker, Stratego, Battleship

Require modeling opponent knowledge and beliefs

Classical adversarial search algorithms like minimax are most directly applicable to deterministic, perfect-information games, though they can be adapted for other scenarios.

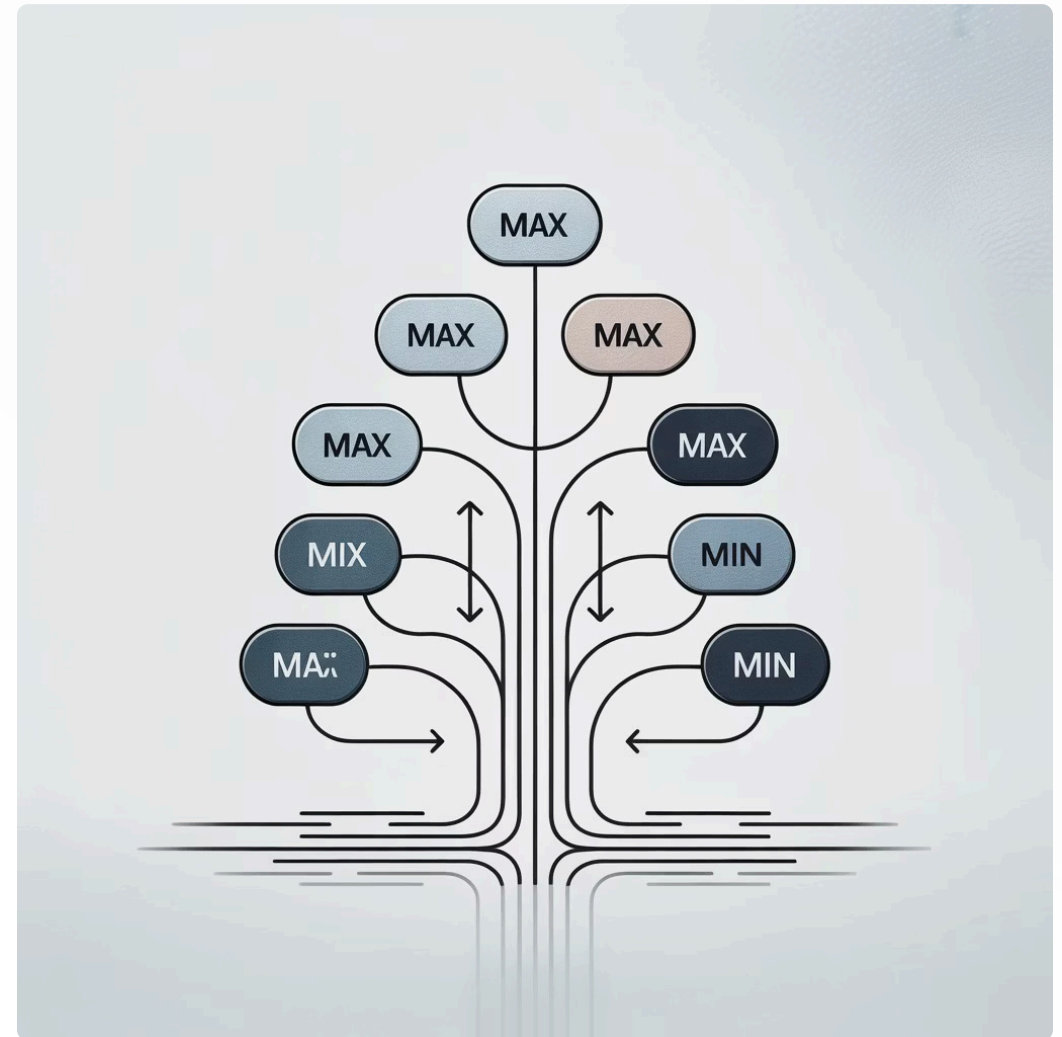
The Minimax Algorithm: Basic Idea

Fundamental Concept

Minimax models a two-player zero-sum game where:

- MAX player tries to maximize the score
- MIN player tries to minimize MAX's score
- Both players play optimally (worst-case reasoning)

The algorithm recursively evaluates all possible moves, assuming each player makes their best move at each step. It then selects the move that guarantees the best outcome regardless of opponent choices.



The key insight: MAX should choose the move that maximizes the minimum value its opponent can force (hence "minimax"). This guarantees a minimum score regardless of what the opponent does.

Minimax: Step-by-Step Example

Evaluate Leaf Nodes

Calculate utility values at terminal states using an evaluation function

In chess, this might consider material advantage, piece positions, king safety, etc.

Bottom-up Propagation

At MIN levels: Select minimum child value (opponent's best move)

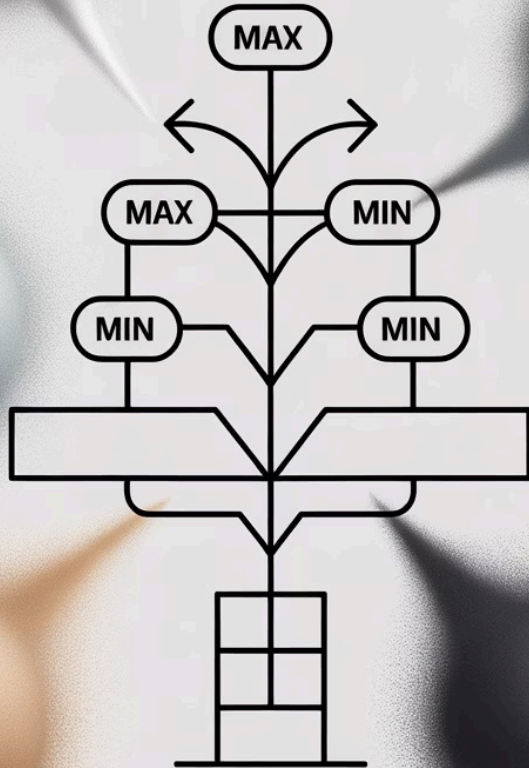
At MAX levels: Select maximum child value (our best move)

Root Decision

MAX selects the action leading to the child with highest value

This guarantees the best possible outcome assuming optimal play

A key limitation is that minimax must evaluate the entire game tree to guarantee optimality. For complex games, this is computationally infeasible, requiring strategies like depth-limited search with heuristic evaluation functions.



Visualization: Minimax on a Simple Game Tree

In this visualization:

- Square nodes represent MAX's turn (trying to maximize value)
- Circular nodes represent MIN's turn (trying to minimize value)
- Leaf nodes show utility values (higher is better for MAX)
- Values propagate upward: MIN selects smallest child value, MAX selects largest
- Bold lines show the optimal path selected by the minimax algorithm

MAX chooses the leftmost action with value 3, as this guarantees a score of at least 3, regardless of MIN's response. This is better than risking the middle path, where MIN could force a value of 2.

Minimax Pseudocode

```
function minimax(node, depth, isMaximizingPlayer):
    if node is terminal or depth = 0:
        return evaluation(node)

    if isMaximizingPlayer:
        value =  $-\infty$ 
        for each child of node:
            value = max(value, minimax(child,
                                         depth-1, false))
        return value

    else: # Minimizing player
        value =  $+\infty$ 
        for each child of node:
            value = min(value, minimax(child,
                                         depth-1, true))
        return value
```

Algorithm Characteristics

- **Time Complexity:** $O(b^d)$ where b is branching factor and d is depth
- **Space Complexity:** $O(d)$ for the recursion stack
- **Completeness:** Yes, if tree is finite
- **Optimality:** Yes, for perfect play in zero-sum games

For practical games, we typically limit the search depth and use a heuristic evaluation function to estimate non-terminal state values. This introduces a trade-off between computation time and decision quality.

Alpha-Beta Pruning: Optimizing Minimax

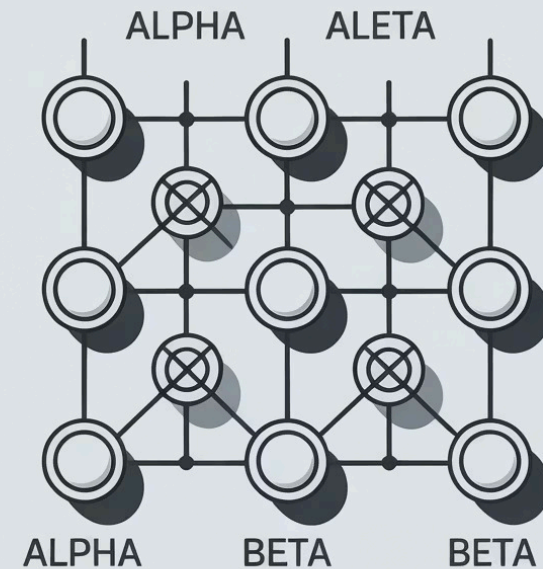
The Challenge

Standard minimax explores every node in the game tree—highly inefficient for complex games.

The Solution

Alpha-beta pruning eliminates branches that cannot influence the final decision:

- Alpha (α): Best value found so far for MAX
- Beta (β): Best value found so far for MIN
- Prune when $\alpha \geq \beta$ (current path cannot improve result)



With perfect move ordering, alpha-beta pruning can reduce the effective branching factor from b to $\sqrt{b}$, allowing search to approximately double in depth for the same computation.

Improvements via Alpha-Beta Pruning



Move Ordering

Examine most promising moves first to maximize pruning. Use iterative deepening, killer moves, and history tables to improve ordering.



Transposition Tables

Cache evaluated positions to avoid recalculating identical states reached through different move sequences.



Forward Pruning

Strategically ignore some moves that appear unpromising (e.g., null-move pruning, futility pruning).

```
function alphabeta(node, depth,  $\alpha$ ,  $\beta$ , isMaximizing):
  if node is terminal or depth = 0:
    return evaluation(node)

  if isMaximizing:
    for each child of node:
       $\alpha$  = max( $\alpha$ , alphabeta(child, depth-1,  $\alpha$ ,  $\beta$ , false))
      if  $\alpha \geq \beta$ : # Beta cutoff
        break # Prune remaining branches
    return  $\alpha$ 
  else:
    for each child of node:
       $\beta$  = min( $\beta$ , alphabeta(child, depth-1,  $\alpha$ ,  $\beta$ , true))
      if  $\beta \leq \alpha$ : # Alpha cutoff
        break # Prune remaining branches
    return  $\beta$ 
```