Lecture Note on Adversarial Search in Artificial Intelligence

Md Rifat Ahmmad Rashid, Associate Professor, EWU

Introduction

Adversarial search is a fundamental concept in artificial intelligence, particularly in the domain of game playing. It involves making decisions in a competitive setting where an opponent is also trying to achieve their own objectives, often in direct opposition to yours. This lecture note will cover the basics of adversarial search, including the minimax algorithm and alpha-beta pruning, which are essential techniques for implementing AI agents that can play games effectively. It is particularly relevant for two-player games with perfect information, such as chess, tic-tac-toe, and checkers, and has applications in video games and strategic AI systems.

What is Adversarial Search?

Adversarial search is a type of search problem where the goal is to find the best possible move in a game or competitive scenario, considering that the opponent is also playing optimally to counter your moves. It is defined as a method applied to situations where you are planning while another actor prepares against you, and your plans could be affected by the opponent's actions. In the realm of AI, adversarial search is primarily used in games, with quick examples including chess, checkers, and tic-tac-toe, where moves depend on the opponent's countermoves.

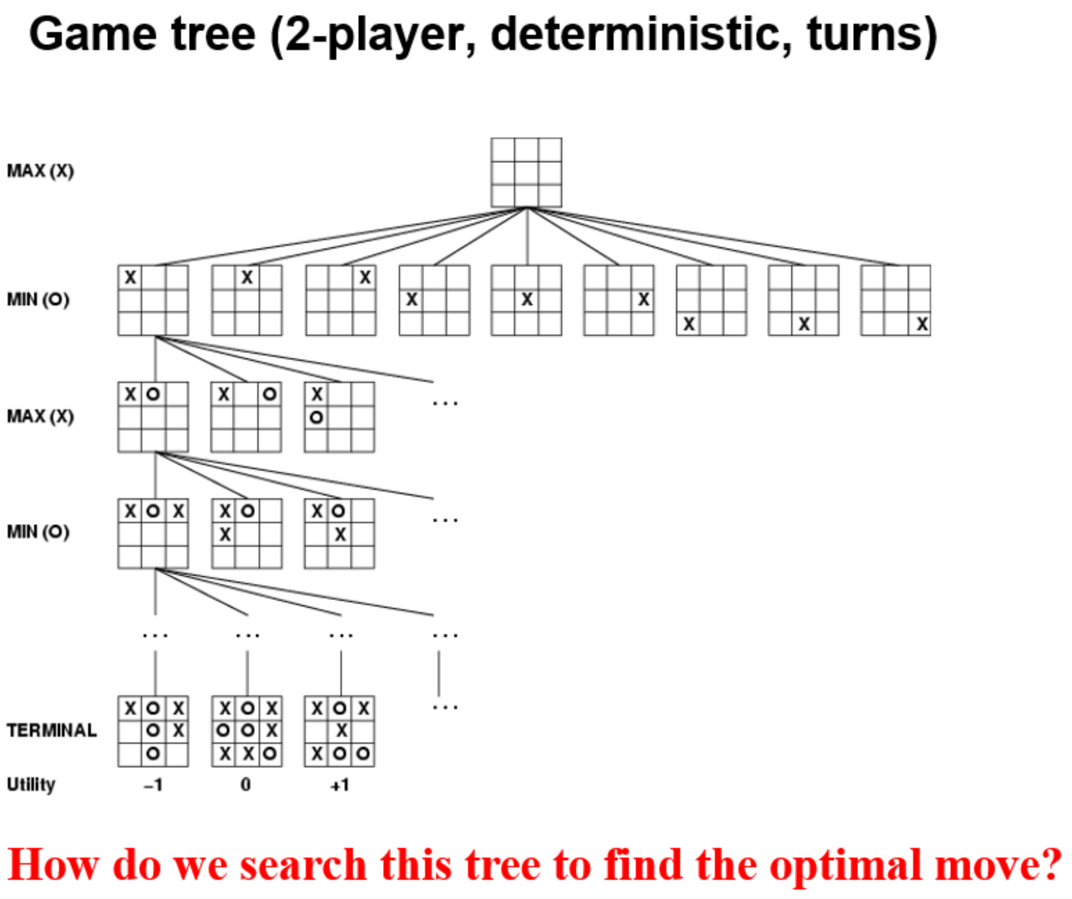
Search versus Games

Search — no adversary

- **Solution** is a (heuristic) method for finding the goal.
- Heuristics and CSP (Constraint Satisfaction Problem) techniques can find an optimal solution.
- **Evaluation function:** An estimate of cost from the start to the goal through a given node.
- **Examples:** path planning, scheduling activities.

Games — adversary

- **Solution** is a strategy (specifies a move for every possible opponent reply).
- Time limits often force an approximate solution.
- **Evaluation function:** Evaluates the "goodness" of a game position.
- **Examples:** chess, checkers, Othello, backgammon.
- **Definition:** Adversarial search is search when there is an "enemy" or "opponent" changing the state of the problem every step in a direction you do not want, such as in chess, business, trading, or war (Adversarial Search).
- **Context:** It is well-suited for competitive environments, particularly two-player zero-sum games, where what is good for one player is the misfortune for the other, with no win-win outcome (Adversarial Search Algorithms in Artificial Intelligence (AI)).


Game tree (2-player, deterministic, turns)

How do we search this tree to find the optimal move?

Figure 1: Game Tree for 8 Puzzle Problem

Game Trees

In adversarial search, the game is represented as a game tree, a hierarchical structure where:

- Each node represents a game state.
- Each edge represents a move from one state to another.
- The root of the tree is the current state of the game, and the leaves are the terminal states where the game ends.
- Players alternate turns, with one player trying to maximize the outcome (the maximizing player) and the other trying to minimize it (the minimizing player).

Key components include:

- **Initial State:** The starting configuration before any moves, such as the starting positions in chess (Adversarial Search in Artificial Intelligence).
- **Player Turns:** Alternating between maximizing and minimizing players, creating a competitive dynamic.
- **Utility Function:** Evaluates the desirability of a game state for the maximizing player, typically at leaf nodes.

Static (Heuristic) Evaluation Functions

An Evaluation Function:

- Estimates how good the current board configuration is for a player.
- Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
- **Othello:** Number of white pieces − Number of black pieces.
- **Chess:** Value of all white pieces − Value of all black pieces.

Typical Values

- Ranges from $-\infty$ (loss) to $+\infty$ (win), or sometimes $[-1, +1]$.

Zero-Sum Property

If the board evaluation is X for one player, it is $-X$ for the opponent. This is characteristic of a "zero-sum game."

Evaluation Functions

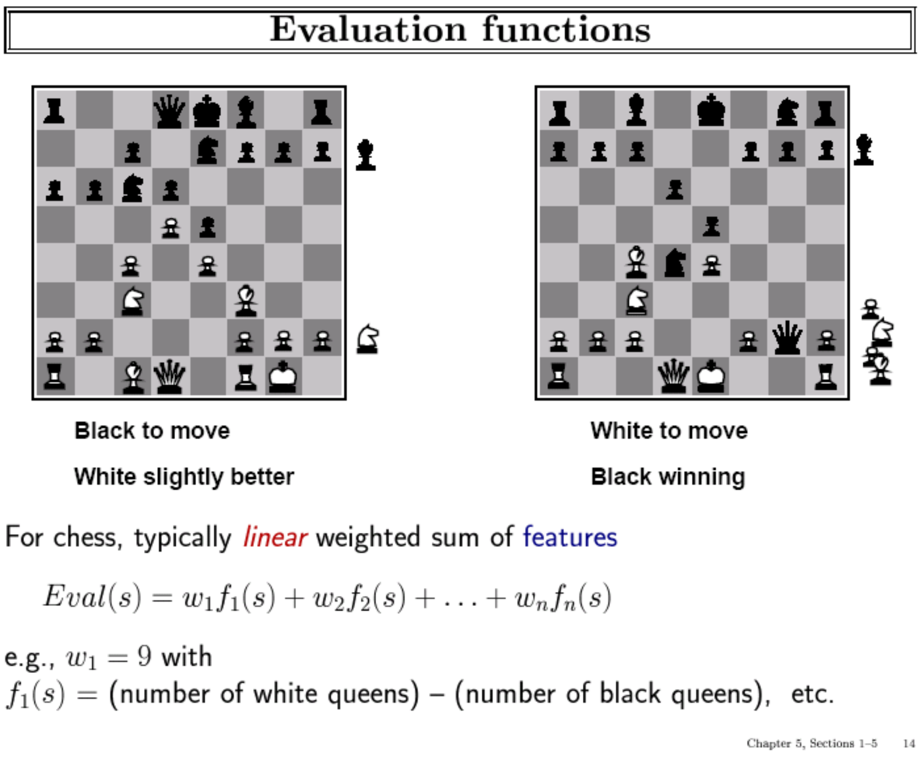
- **Black to move:** White slightly better
- **White to move:** Black winning

For chess, we typically use a *linear weighted sum* of features:

$$\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s).$$

Example:

- Let $w_1 = 9$.
- Let $f_1(s) = (\#\text{white queens}) - (\#\text{black queens})$.
- Other features (e.g., bishops, knights, rooks, king safety, etc.) have their own weights w_i.



Evaluation functions

Black to move

White slightly better

White to move

Black winning

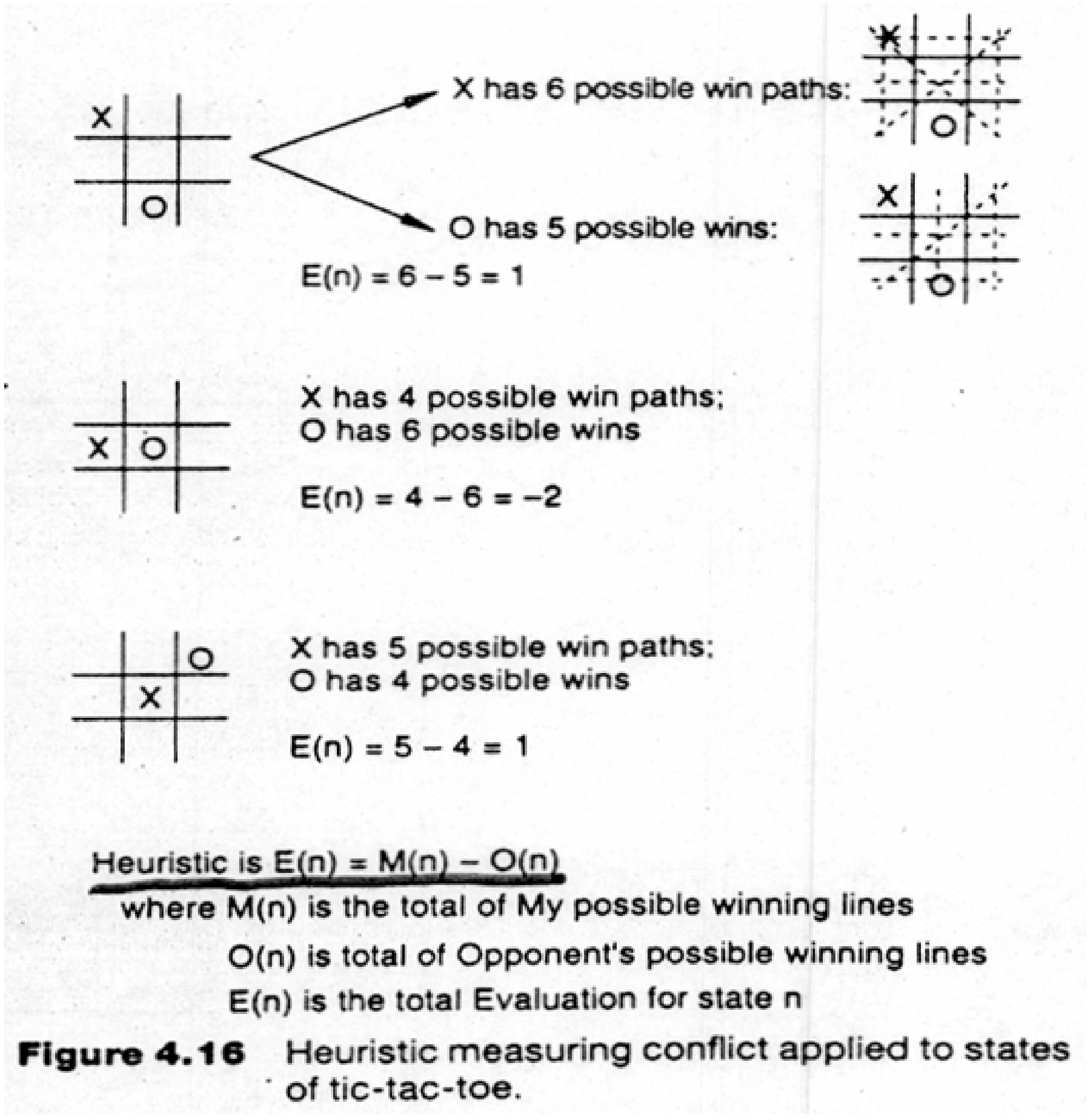
For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

Chapter 5, Sections 1–5 14

Figure 2: Evaluation Function



X has 6 possible win paths:

O has 5 possible wins:

$E(n) = 6 - 5 = 1$

X has 4 possible win paths;
O has 6 possible wins

$E(n) = 4 - 6 = -2$

X has 5 possible win paths;
O has 4 possible wins

$E(n) = 5 - 4 = 1$

Heuristic is $E(n) = M(n) - O(n)$
where $M(n)$ is the total of My possible winning lines
$O(n)$ is total of Opponent's possible winning lines
$E(n)$ is the total Evaluation for state n

Figure 4.16 Heuristic measuring conflict applied to states of tic-tac-toe.

Figure 3: The static evaluation function heuristic

The Static Evaluation Function Heuristic

- **Example 1:**
  - X has 6 possible win paths.
  - O has 5 possible win paths.
  - $E(n) = 6 - 5 = 1$.
- **Example 2:**
  - X has 4 possible win paths.
  - O has 6 possible win paths.
  - $E(n) = 4 - 6 = -2$.
- **Example 3:**
  - X has 5 possible win paths.
  - O has 4 possible win paths.
  - $E(n) = 5 - 4 = 1$.

A common heuristic formula is:

$$E(n) = M(n) - O(n),$$

where

- $M(n)$ = total number of my possible winning lines,
- $O(n)$ = total number of opponent's possible winning lines,
- $E(n)$ = overall evaluation for state n.

Figure 4.16: Heuristic measuring conflict applied to states of tic-tac-toe.

The Minimax Algorithm

The minimax algorithm is a recursive algorithm used to evaluate the game tree and determine the best move for the maximizing player, assuming that the minimizing player will also play optimally. It is based on the principle that each player will make the best possible move given the current state, leading to a zero-sum game where one player's loss is the other's gain.

- **Process:**
  - At each maximizing node, the algorithm chooses the move that *maximizes* the value returned by its child nodes.
  - At each minimizing node, it chooses the move that *minimizes* the value returned by its child nodes.
  - The value of a leaf node is determined by a utility function that evaluates the desirability of that game state for the maximizing player.
- **Theoretical Basis:** It is based on von Neumann's minimax theorem, which states that in zero-sum games, there is always a set of strategies leading to both players gaining the same value, and this is the best possible value one can expect (Artificial Intelligence/Search/Adversarial search/Minimax Search).
- **Pseudocode:**

```
def minimax(node, depth, maximizingPlayer):
    if depth == 0 or node is terminal:
        return utility(node)
    if maximizingPlayer:
        maxEval = -infinity
        for child in node.children:
            eval = minimax(child, depth-1, False)
            maxEval = max(maxEval, eval)
        return maxEval
    else:
        minEval = +infinity
        for child in node.children:
            eval = minimax(child, depth-1, True)
            minEval = min(minEval, eval)
        return minEval
```

- **Example:** Consider a simple game where *Start (A, maximizing)* has two moves:
  - Move 1 $\rightarrow$ Intermediate 1 (B, minimizing) with end states 10 and 0 $\rightarrow \min(10, 0) = 0$
  - Move 2 $\rightarrow$ Intermediate 2 (B, minimizing) with end states 5 and 8 $\rightarrow \min(5, 8) = 5$
  - Therefore, *Start* chooses $\max(0, 5) = 5$, so Move 2 is selected.

Alpha-Beta Pruning

Alpha-beta pruning is an optimization technique for the minimax algorithm that reduces the number of nodes evaluated by pruning branches that won't influence the final decision. It passes two extra parameters, α and β, to the minimax function to achieve this.

Parameters:

- **Alpha:** The best value that the maximizing player currently can guarantee at that level or above.
- **Beta:** The best value that the minimizing player currently can guarantee at that level or below.

Process: It cuts off branches in the game tree which need not be searched because there already exists a better move available, significantly reducing computation time and allowing deeper searches.

Pseudocode:

```
def alphaBeta(node, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or node is terminal:
        return utility(node)
    if maximizingPlayer:
        maxEval = -infinity
        for child in node.children:
            eval = alphaBeta(child, depth-1, alpha, beta, False)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, maxEval)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = +infinity
        for child in node.children:
            eval = alphaBeta(child, depth-1, alpha, beta, True)
            minEval = min(minEval, eval)
            beta = min(beta, minEval)
            if beta <= alpha:
                break
        return minEval
```

Example: Consider the game tree:

- **Max (root)**
- **Min1:** Max a (3), Max b (5) $\rightarrow \min(3, 5) = 3$
- **Min2:** Max c (2), Max d (9) $\rightarrow$ Initially evaluate Max c: 2, then = 2, $\beta = 2$; if $\alpha = 3$, prune Max d and return 2.
- **Min3:** Max e (1), Max f (4) $\rightarrow$ Evaluate Max e: 1, then = 1, $\beta = 1$; prune Max f and return 1.
- **Max chooses** $\max(3, 2, 1) = 3$, demonstrating that pruning reduces evaluations.

Other Techniques

Beyond minimax and alpha-beta pruning, other methods enhance adversarial search:

- **Monte Carlo Tree Search (MCTS):** Combines random sampling with tree search, useful for games with large branching factors, such as Go, as seen in AlphaGo (Adversarial Search Algorithms in Artificial Intelligence (AI)).
- **Heuristics and Evaluation Functions:** Used to estimate the value of non-terminal states when exhaustive search is impractical, relying on past precedence or guesswork (Adversarial Search — Comprehensive Guide to Adversarial Search).
- **Transposition Tables:** Store and reuse evaluations of game states reached through different paths, improving efficiency.

Applications

Adversarial search has significant real-world applications:

- **Chess Programs:** Deep Blue used minimax with alpha-beta pruning to beat Garry Kasparov in 1997, a milestone in AI (Alpha–beta pruning).
- **Go Programs:** AlphaGo combined MCTS with deep learning to defeat the world champion, showcasing advanced adversarial search (Adversarial Search in Artificial Intelligence).
- **Video Games:** AI opponents in games like StarCraft II use simplified versions for real-time decision-making.
- **Other Domains:** While primarily for games, it extends to planning problems with multiple agents and conflicting goals, such as in business or trading strategies (What Is Adversarial Search? — Definition by Techslang).

Limitations and Challenges

Adversarial search faces several challenges:

- **Computational Complexity:** The game tree grows exponentially with depth, with a time complexity of $O(b^d)$ without pruning, where b is the branching factor and d is the depth (Alpha Beta Pruning in AI).
- **Heuristics Dependency:** Requires good evaluation functions, which can be difficult to define in complex domains.
- **Assumptions:** Assumes perfect information and deterministic outcomes, which may not hold in real-world scenarios like poker with incomplete information.
- **Real-Time Constraints:** In fast-paced games, exhaustive search is often impractical, necessitating approximations.

Conclusion

Adversarial search, particularly through minimax and alpha-beta pruning, is fundamental for AI in game playing. Understanding these concepts provides a basis for developing more sophisticated AI agents capable of making optimal decisions in competitive environments. While challenges like computational complexity exist, advancements like MCTS and heuristics continue to expand its applicability.

Table: Comparison of Search Techniques

Technique	**Description**	**Use Case**	**Advantages**	**Limitations**
Minimax	Evaluates game tree assuming optimal opponent play	Chess, Tic-tac-toe	Guarantees optimal move	Exponential time complexity
Alpha-Beta Pruning	Optimizes minimax by pruning unnecessary branches	Chess, Go	Reduces computation, faster search	Requires ordered moves for best effect
Monte Carlo Tree Search	Combines random sampling with tree search	Go, Complex games	Handles large search spaces	Stochastic, less deterministic

Table 1: Comparison of Search Techniques

Key Citations

- Adversarial Search in Artificial Intelligence Scaler Topics
- Minimax Algorithm in Game Theory Set 4 Alpha-Beta Pruning GeeksforGeeks
- Alpha–beta pruning Wikipedia
- Adversarial Search Algorithms in Artificial Intelligence AI GeeksforGeeks
- Adversarial Search Comprehensive Guide to Adversarial Search Educba
- Alpha Beta Pruning in AI My Great Learning
- What Is Adversarial Search — Definition by Techslang
- Adversarial Search Artificial Intelligence CodePractice
- Artificial Intelligence Search Adversarial search Minimax Search Wikibooks
- Adversarial Search Computing DCU