Unit 3 (Ref to syllabus and Pls read theory from Prachi and Joshi Book Also)

Adversarial Search in Artificial Intelligence

Adversarial search is a key concept in artificial intelligence (AI), especially in competitive scenarios where two or more agents interact, often with opposing goals. It is commonly used in game-playing AI, where the goal is to anticipate the opponent's moves and respond strategically.

For instance, in games like chess or tic-tac-toe, adversarial search enables AI agents to evaluate possible outcomes of their actions and counter their opponent's strategies. This decision-making approach ensures that the AI remains competitive and effective in complex environments.

What is an Adversarial Search?

Adversarial search is a technique in AI that **deals with decision-making problems where multiple agents compete against each other**.

These agents often have opposing goals, making the task a zero-sum game—one agent's gain is another's loss.

It is a cornerstone of game theory and forms the foundation for AI systems in competitive environments.

In adversarial search, the AI agent evaluates all possible moves and counter-moves by the opponent to identify the best strategy.

This method ensures that the AI can anticipate the opponent's actions and adjust its approach accordingly.

Significance in AI:

Adversarial search is widely used in:

- **Game-playing AI**: For example, chess engines that predict the opponent's next moves.
- **Strategy-based systems**: Applications like financial modeling or military simulations.
- **Negotiation AI**: Where two parties aim to maximize their respective outcomes.

Role of Adversarial Search in AI

Adversarial search plays a critical role in enabling AI systems to make intelligent decisions in competitive environments. By anticipating the actions of opponents, the AI can strategize effectively, ensuring optimal outcomes even in complex and dynamic scenarios.

Key Contributions of Adversarial Search:

1. **Strategic Decision-Making**
2. **Handling Uncertainty**:
3. **Application in Competitive Systems**:
    - **Business negotiations**: Predicting competitor strategies.
    - **Autonomous vehicles**: Handling interactions with other agents on the road.
    - **AI-driven financial models**: Anticipating market changes.

Adversarial Search Algorithms

Adversarial search algorithms are designed to help AI agents evaluate and select the optimal moves in competitive scenarios. These algorithms simulate potential future actions and counteractions to identify the best strategy. Two of the most widely used algorithms are the **Minimax Algorithm** and **Alpha-Beta Pruning**.

Minimax Algorithm

The Minimax Algorithm is a foundational adversarial search algorithm used in turn-based games like chess or tic-tac-toe. It aims to minimize the possible loss for a worst-case scenario while maximizing potential gains. In other words:

- **The Maximizer**: Tries to maximize the score.
- **The Minimizer**: Tries to minimize the score.

Key Terminologies in the Minimax Algorithm:

1. ***Minimax Tree***:

- A decision tree representing all possible moves of both players.
- Each level alternates between the maximizer and minimizer.

2. ***MAX (Maximizer)***:

- The player aiming to maximize the score.
- Focuses on selecting the move with the highest value.

3. ***MIN (Minimizer)***:

- The player aiming to minimize the score.
- Focuses on reducing the opponent's score.

4. ***Initial State***:

- The starting point of the game or scenario.

5. ***Terminal State***:

- The end of the game, where the outcome (win, lose, or draw) is evaluated.

6. ***Heuristic Evaluation Function***:

- A function used to estimate the value of non-terminal states when the search cannot explore the entire tree due to computational limitations.

How Minimax Algorithm Works:

1. The algorithm generates a game tree, representing all possible moves up to a certain depth.
2. The terminal states are assigned a score based on the outcome.

3. The algorithm works recursively, calculating scores backward from the terminal states to the root node, considering the best possible move for both players.

Example:

In tic-tac-toe:

- The maximizer (AI) aims to win by placing three symbols in a row.
- The minimizer (opponent) tries to block the AI's moves and create their own winning opportunity.

Understanding the Minimax Algorithm: Decision-Making in Game Trees

The Minimax Algorithm is a decision-making process that navigates game trees to find the best move for a given player in a competitive environment. By considering all possible moves and counter-moves, it ensures that the AI agent can make informed decisions based on the assumption that the opponent is playing optimally.

Maximizer and Minimizer

In the context of the Minimax Algorithm:

- **Maximizer**: The player (usually AI) who aims to maximize the score, choosing moves with the highest evaluation value.
- **Minimizer**: The opponent who aims to minimize the score, selecting moves that reduce the maximizer's advantage.

Example of Decision-Making

Scenario: Consider a simple game tree where the maximizer and minimizer alternate turns.

1. Initial Decision-Making:

The maximizer begins by exploring all possible moves. For each move, it assumes that the minimizer will respond optimally to minimize the maximizer's gain.

Example: The maximizer evaluates moves A, B, and C, each leading to different branches in the game tree.

2. Minimizer's Choice:

After the maximizer selects a move, the minimizer evaluates all possible responses. It chooses the move with the lowest value to counter the maximizer's strategy.

Example: If move A leads to outcomes 3, 5, and 7, the minimizer will choose the path with value 3 (lowest score).

3. Traversal to the Right:

The algorithm systematically explores the right subtree of the game tree, ensuring no possibility is overlooked.

Example: After evaluating the left branch, the algorithm moves to evaluate outcomes from move B, repeating the process for each branch.

4. Recursive Process:

At each level, the algorithm recursively evaluates future moves until it reaches a terminal state (end of the game or maximum depth). The evaluations are then propagated backward through the tree.

Example: If the outcomes of a branch are 4, 6, and 2, the minimax score is determined by the optimal responses of both players.

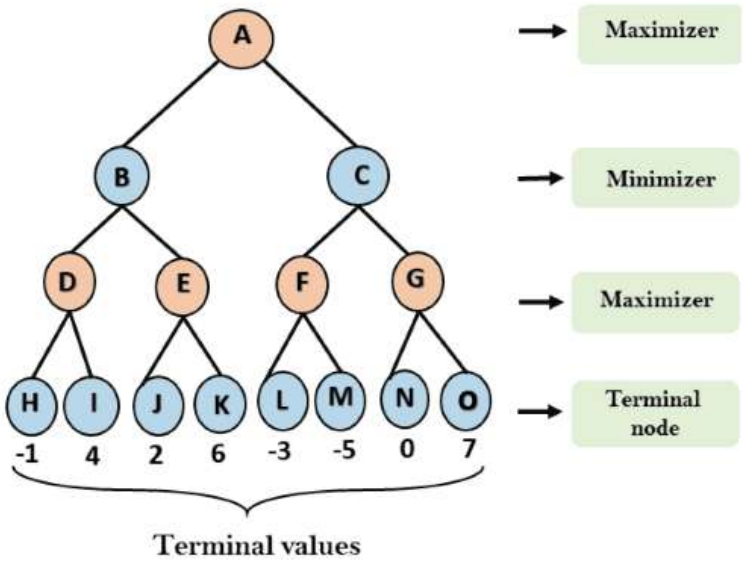
5. Comparison and Outcome:

The algorithm compares the minimax scores of all branches and selects the one with the best outcome for the maximizer.

Example: If the scores for moves A, B, and C are 3, 5, and 4 respectively, the maximizer will choose move B (highest score).

Problem

Step 1: The method constructs the whole game-tree and applies the utility function to obtain utility values for the terminal states in the first step. Let's assume A is the tree's initial state in the diagram below. Assume that the maximizer takes the first turn with a worst-case initial value of -infinity, and the minimizer takes the second turn with a worst-case initial value of +infinity.

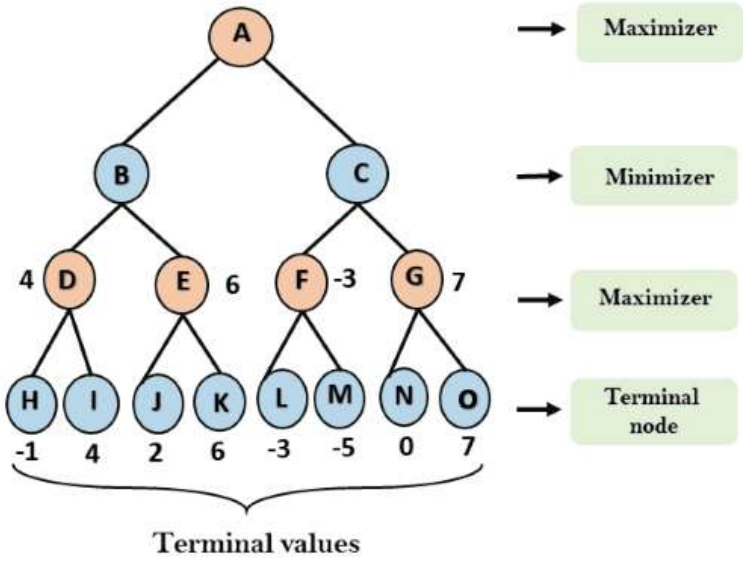


Example: Working of Min-Max Algorithm

step-1

Step 2: Next, we'll locate the Maximizer's utilities value, which is -, and compare each value in the terminal state to the Maximizer's initial value to determine the upper nodes' values. It will select the best option from all of them.

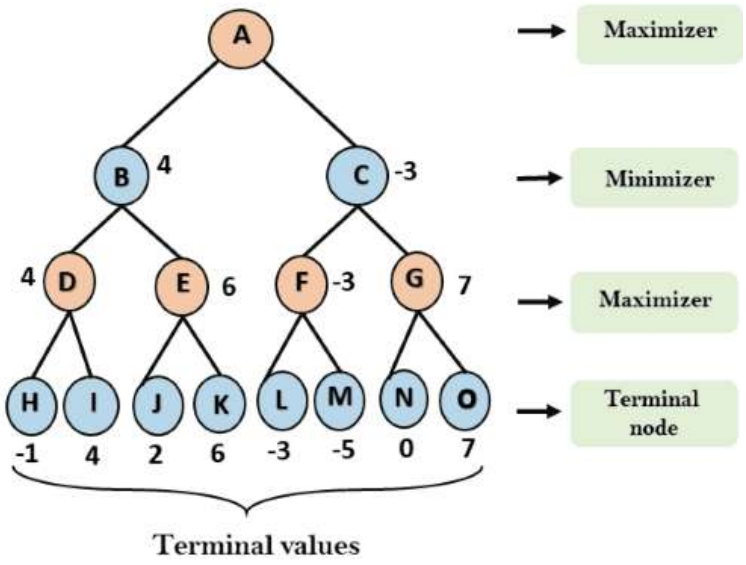
- For node D max(–1,– –∞) => max(–1,4)= 4
- For Node E max(2, –∞) => max(2, 6)= 6
- For Node F max(–3, –∞) => max(–3,–5) = –3
- For node G max(0, –∞) = max(0, 7) = 7



Step-2

Step 3: Now it's the minimizer's time, thus it'll compare all nodes' values with + and determine the 3rd layer node values.

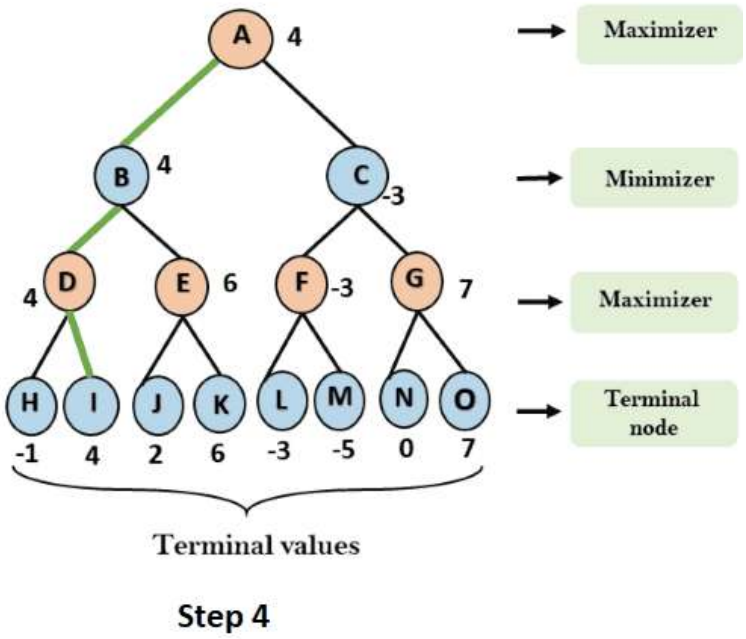
- For node B = min(4,6) = 4
- For node C = min (–3, 7) = –3



Step 3

In the next step, algorithm traverse the next successor of Node B which is node E, and the values of $\alpha = -\infty$, and $\beta = 3$ will also be passed.

Step 4: Now it's Maximizer's turn, and it'll choose the maximum value of all nodes and locate the root node's maximum value. There are only four layers in this game tree, so we can go to the root node right away, but there will be more layers in real games.
For node A max(4, −3)= 4



Step 4

That was the complete workflow of the minimax two player game.

Time and Space Complexity of Minimax Algorithm

1. Time Complexity:

The time complexity of the Minimax Algorithm is , where:

- : Branching factor (average number of possible moves at each step).
- : Depth of the game tree.

2. Explanation:

As the depth and branching factor increase, the number of nodes to evaluate grows exponentially, making the algorithm computationally expensive.

3. Space Complexity:

The space complexity is , as the algorithm uses a recursive stack to store states up to the depth .

Optimization:

- Techniques like **Alpha-Beta Pruning** can significantly reduce the number of nodes evaluated, improving efficiency without compromising accuracy.

Alpha-Beta Pruning

Alpha-Beta Pruning is an optimization technique applied to the Minimax Algorithm. It significantly reduces the number of nodes evaluated in the game tree, thereby improving the algorithm's efficiency. By "pruning" branches that cannot influence the final decision, the algorithm saves computational resources without compromising the quality of the solution.

How Alpha-Beta Pruning Works

Alpha-Beta Pruning introduces two parameters, **alpha** and **beta**, to minimize unnecessary evaluations:

- **Alpha**: The best value the maximizer can guarantee so far.
- **Beta**: The best value the minimizer can guarantee so far.

During tree traversal:

1. If a branch cannot produce a better outcome than the current alpha or beta value, it is pruned (skipped).
2. This reduces the number of nodes evaluated, especially in scenarios with deep game trees.

Why Alpha-Beta Pruning is Effective Algorithm in Adversarial Search

- It eliminates branches where further exploration would not affect the final decision.
- Reduces the time complexity significantly, making adversarial search more practical for real-world applications.

Understanding Alpha-Beta Pruning in Game Trees

1. Pruning in Action:

During tree traversal, branches that cannot affect the decision are skipped, reducing the number of nodes evaluated.

Example:

In a tic-tac-toe game tree:

The maximizer explores a branch and finds that its score is already greater than the best score achievable by the minimizer in other branches. The remaining branches are pruned.

2. Time Complexity Improvement:

With Alpha-Beta Pruning, the effective time complexity is reduced to in the best-case scenario, where is the branching factor and is the depth of the tree.

Comparison with Minimax Algorithm

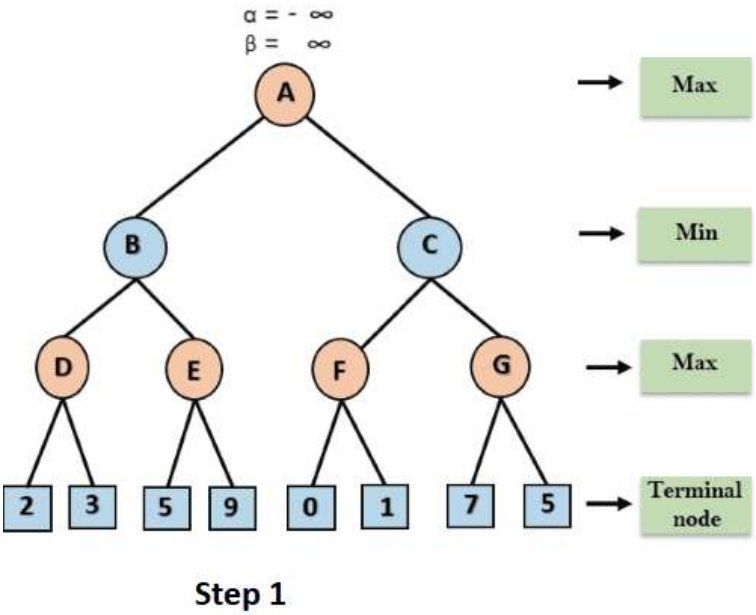
Feature	Minimax	Alpha-Beta Pruning
Node Evaluations	All Nodes	Fewer Nodes
Time Complexity	O(bd)	O(bd/2) (Best Case)

Efficiency	Lower	Higher

Working of Alpha-Beta Pruning:

To better understand how Alpha-beta pruning works, consider a two-player search tree.

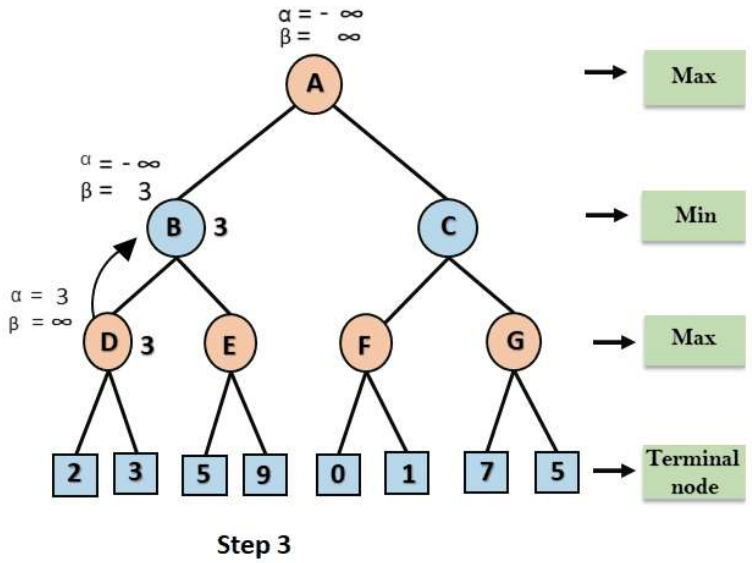
Step 1: The Max player will begin by moving from node A, where = – and = +, and passing these values of alpha and beta to node B, where again = – and = +, and Node B passing the same value to its offspring D.



Step 1

Step 2: The value of will be determined as Max's turn at Node D. The value of is compared to 2, then 3, and the value of at node D will be max (2, 3) = 3, and the node value will also be 3.

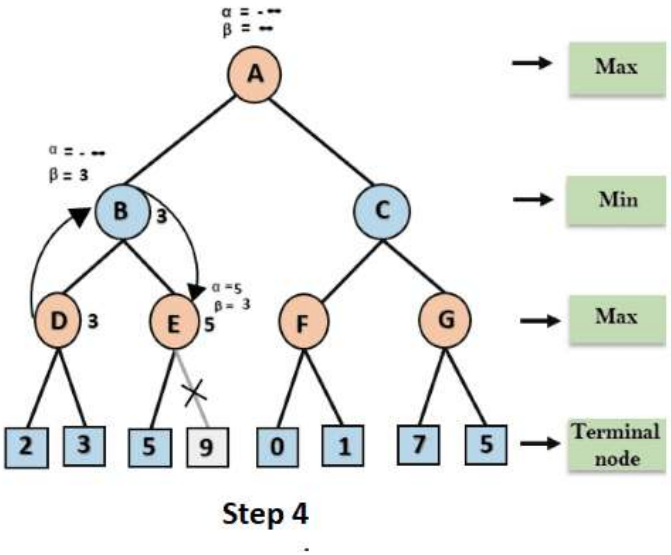
Step 3: The algorithm now returns to node B, where the value of will change as this is a turn of Min, now = +, and will compare with the value of the available subsequent nodes, i.e. min (, 3) = 3, so at node B now = –, and= 3.



Step 3

In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= −∞, and β= 3 will also be passed.

Step 4: Max will take its turn at node E, changing the value of alpha. The current value of alpha will be compared to 5, resulting in max (−, 5) = 5, and at node E= 5 and= 3, where >=, the right successor of E will be pruned, and the algorithm will not traverse it, and the value at node E will be 5.

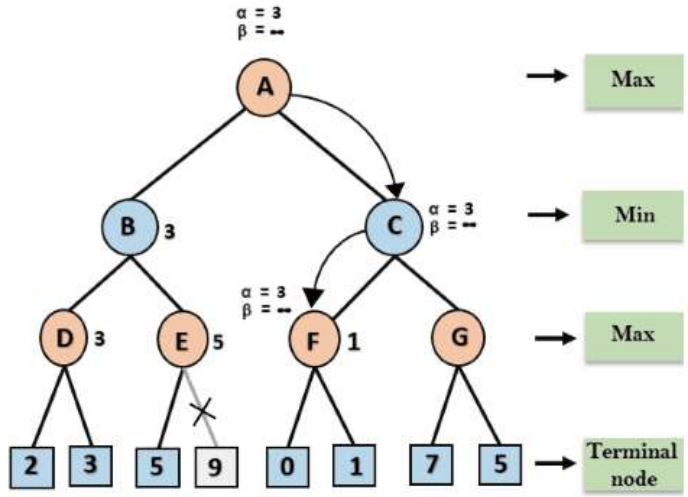


Step 4

Step 5: The method now goes backwards in the tree, from node B to node A. The value of alpha will be modified at node A, and the highest available value will be 3 as max (-, 3)= 3, and = +. These two values will now transfer to A's right successor, Node C.

=3 and = + will be passed on to node F at node C, and the same values will be passed on to node F.

Step 6: TAt node F, the value of will be compared with the left child, which is 0, and max(3,0)= 3, and then with the right child, which is 1, and max(3,1)= 3 will remain the same, but the node value of F will change to 1.



Step 6

Step 7: Node F returns the node value 1 to node C, at C = 3 and = +, the value of beta is modified, and it is compared to 1, resulting in min (, 1) = 1. Now, at C, =3 and = 1, and again, it meets the condition >=, the algorithm will prune the next child of C, which is G, and will not compute the complete sub-tree G.