




Piyush Wairale

For GATE DA full test series & study materials. Visit piyushwairale.com

GATE in Data Science & Artificial Intelligence

**GATE DA 2024
TEST SERIES**

By:
Piyush Wairale
MTech (IIT Madras)
Instructor at IIT Madras BS in Data Science Degree



Connect with me:

Youtube: [Piyush Wairale IITM](#)

Linkedin: [Piyush Wairale](#)

Facebook: [Piyush Wairale](#)

Instagram: [Piyush Wairale](#)

Please refrain from distributing this copyrighted study material to others. Sharing it with someone could potentially result in them securing a seat at IIT instead of you. 😊



3. Adversarial Search:

Whereas, previously, we have discussed algorithms that need to find an answer to a question, in adversarial search the algorithm faces an opponent that tries to achieve the opposite goal. Often, AI that uses adversarial search is encountered in games, such as tic tac toe

In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.

1. **Adversarial Search:** Competitive environments in AI involve agents with conflicting goals, leading to adversarial search problems often modeled as games.
2. **Game Theory Foundation:** Mathematical game theory is the basis for analyzing multi agent interactions, including deterministic, two-player, zero-sum games of perfect information.
3. **Chess as an Example:** Chess serves as a classic example of a deterministic, fully observable, turn-taking game where one player's win corresponds to the other's loss.
4. **Abstract and Analytical:** Games provide an abstract and analytically appealing subject for AI research due to their well-defined state representations and limited action choices.
5. **Limited Focus on Physical Games:** While AI research often focuses on abstract games, physical games like croquet and ice hockey, with complex descriptions and imprecise rules, receive less attention in the AI community.

Types of Adversarial Search:

1. Minimax search
2. Alpha-beta pruning

3.1 Minimax search :

1. **Minimax Decision Computation:** The minimax algorithm computes the minimax decision from the current state in a game.
2. **Recursive Approach:** It uses a recursive computation to determine the minimax values of each successor state, following the defining equations.
3. **Leaf Node Evaluation:** The recursion goes down to the leaves of the game tree, where the UTILITY function is used to evaluate their values.
4. **Backing Up Values:** Minimax values are then backed up through the tree as the recursion unwinds, considering minimum and maximum values.
5. **Time and Space Complexity:** The algorithm has a time complexity of $O(b^m)$, where b is the number of legal moves, and m is the maximum depth of the tree, making it



impractical for real games but useful for theoretical analysis and as a foundation for practical algorithms.

Minimax search is a decision-making algorithm commonly used in two-player games like chess or tic-tac-toe, where one player aims to maximize their chances of winning, while the other player tries to minimize their opponent's chances. It operates based on the assumption that both players are playing optimally.

Explanation using Tic-Tac-Toe example:

(Kindly refer to lectures for detailed explanation of this)

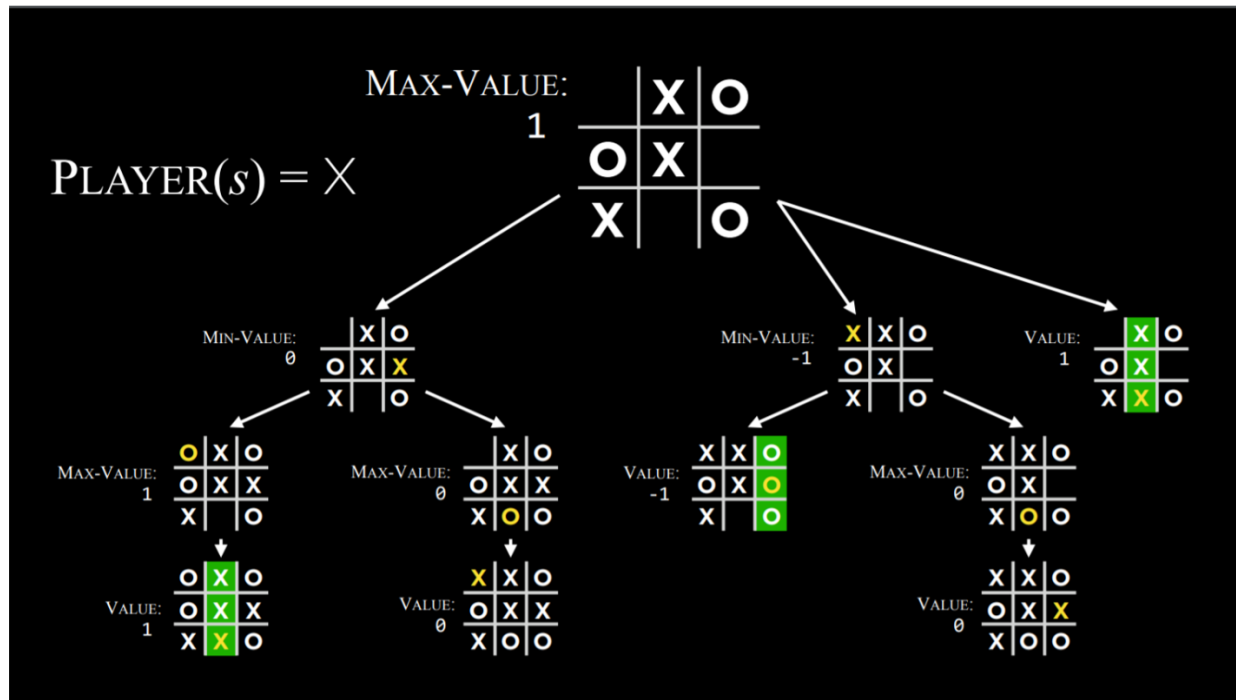
A type of algorithm in adversarial search, Minimax represents winning conditions as (-1) for one side and $(+1)$ for the other side. Further actions will be driven by these conditions, with the minimizing side trying to get the lowest score, and the maximizer trying to get the highest score.

Representing a Tic-Tac-Toe AI:

- S_0 : Initial state (in our case, an empty 3X3 board)
- $Players(s)$: a function that, given a state s , returns which player's turn it is (X or O).
- $Actions(s)$: a function that, given a state s , return all the legal moves in this state (what spots are free on the board).
- $Result(s, a)$: a function that, given a state s and action a , returns a new state. This is the board that resulted from performing the action a on state s (making a move in the game).
- $Terminal(s)$: a function that, given a state s , checks whether this is the last step in the game, i.e. if someone won or there is a tie. Returns *True* if the game has ended, *False* otherwise.
- $Utility(s)$: a function that, given a terminal state s , returns the utility value of the state: -1 , 0 , or 1 .

How the algorithm works:

Recursively, the algorithm simulates all possible games that can take place beginning at the current state and until a terminal state is reached. Each terminal state is valued as either (-1) , 0 , or $(+1)$.



Minimax Algorithm in Tic Tac Toe

Knowing based on the state whose turn it is, the algorithm can know whether the current player, when playing optimally, will pick the action that leads to a state with a lower or a higher value. This way, alternating between minimizing and maximizing, the algorithm creates values for the state that would result from each possible action.

To give a more concrete example, we can imagine that the maximizing player asks at every turn: “if I take this action, a new state will result. If the minimizing player plays optimally, what action can that player take to bring to the lowest value?” However, to answer this question, the maximizing player has to ask: “To know what the minimizing player will do, I need to simulate the same process in the minimizer’s mind: the minimizing player will try to ask: ‘if I take this action, what action can the maximizing player take to bring to the highest value?’”

This is a recursive process, and it could be hard to wrap your head around it; looking at the pseudo code below can help. Eventually, through this recursive reasoning process, the maximizing player generates values for each state that could result from all the possible actions at the current state. After having these values, the maximizing player chooses the highest one.



General Algorithm of Minimax Search:

1. Begin with the current game state, typically representing the board position.
2. Generate all possible moves or actions for the player whose turn it is, creating a set of child states.
3. Evaluate the utility or value of each child state using a heuristic function or game-specific rules.
4. If the maximum depth of the search tree is reached or a terminal state (win, loss, or draw) is encountered, assign a utility value to that state.
5. Propagate these utility values back up the tree to the parent node using the minimax principle:
 - If it's the player's turn to maximize, choose the child state with the highest utility value.
 - If it's the opponent's turn to minimize, choose the child state with the lowest utility value.
6. Continue this process recursively, alternating between maximizing and minimizing, until a best move is determined.

For example:

In a game of tic-tac-toe, each player tries to maximize their chances of winning (maximize utility) while minimizing their opponent's chances (minimize utility).



```
function MINIMAX-DECISION(state) returns an action  
return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$ 
```

```
function MAX-VALUE(state) returns a utility value  
if TERMINAL-TEST(state) then return UTILITY(state)  
v  $\leftarrow -\infty$   
for each a in ACTIONS(state) do  
    v  $\leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$   
return v
```

```
function MIN-VALUE(state) returns a utility value  
if TERMINAL-TEST(state) then return UTILITY(state)  
v  $\leftarrow \infty$   
for each a in ACTIONS(state) do  
    v  $\leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$   
return v
```

An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\arg \max_a \in S f(a)$ computes the element *a* of set *S* that has the maximum value of *f(a)*.

Performance:

- 1. Completeness:** Minimax search guarantees finding the optimal solution in two-player, zero-sum games with perfect information, assuming both players play optimally.
- 2. Optimality:** It ensures an optimal strategy when used in conjunction with an appropriate heuristic function or evaluation method.
- 3. Time complexity:** The time complexity of minimax search is exponential in the depth of the search tree, $O(b^d)$, where *b* is the branching factor, and *d* is the depth.
- 4. Space complexity:** The space complexity is also $O(b^d)$ due to the need to store the entire search tree.

Advantages of Minimax Search:

- Guarantees an optimal strategy when applied to zero-sum games.
- Provides a clear decision-making process in games with perfect information.
- Can be extended with alpha-beta pruning to reduce the search space effectively.



Disadvantages of Minimax Search:

1. Intractable for games with deep or complex search trees due to exponential time and space requirements.
2. Assumes perfect play from both players, which may not reflect real-world scenarios.
3. Requires a well-defined heuristic function or evaluation method, which can be challenging to design for some games.
4. Not suitable for games with imperfect information or games involving chance elements.

3.2 Alpha-beta pruning:

A way to optimize *Minimax*, Alpha-Beta Pruning skips some of the recursive computations that are decidedly unfavorable. After establishing the value of one action, if there is initial evidence that the following action can bring the opponent to get to a better score than the already established action, there is no need to further investigate this action because it will decidedly be less favorable than the previously established one.

1. Minimax search can lead to examining an exponential number of game states in the tree, making it computationally expensive.
2. Alpha-beta pruning is a technique that helps reduce the number of nodes evaluated in a minimax tree by eliminating branches that cannot affect the final decision.
3. Alpha-beta pruning allows us to identify the minimax decision without evaluating all leaf nodes in the game tree.
4. This pruning technique simplifies the computation of the minimax decision and ignores the values of pruned branches that don't impact the final outcome.
5. Alpha-beta pruning is applicable to trees of any depth and can often eliminate entire subtrees, significantly reducing the search space.

Alpha-beta pruning is an optimization technique used in the minimax algorithm to reduce the number of nodes evaluated in the search tree. It efficiently narrows down the search space by eliminating branches that cannot affect the final decision, significantly improving the algorithm's performance.

How Alpha-Beta Pruning Works:

1. Alpha and Beta Values:

- a. In alpha-beta pruning, two values, alpha and beta, are maintained for each node in the search tree.
- b. Alpha represents the best (highest) score that the maximizing player (e.g., Player X) can guarantee from the current node or any of its ancestors.



- c. Beta represents the best (lowest) score that the minimizing player (e.g., Player O) can guarantee from the current node or any of its ancestors.
- d. Initially, alpha is set to negative infinity, and beta is set to positive infinity.

2. Pruning Condition:

- a. During the traversal of the search tree, if it is determined that a branch (subtree) can never lead to a better outcome than the current best-known outcome, it can be pruned or cut off from further evaluation.
- b. Pruning occurs when the following condition is met:
 - i. For maximizing nodes: If the current node's alpha value is greater than or equal to its beta value, pruning can be performed.
 - ii. For minimizing nodes: If the current node's beta value is less than or equal to its alpha value, pruning can be performed.

3. Propagation:

- a. Alpha and beta values are propagated up the tree during the recursive traversal, adjusting them based on the values obtained from child nodes.
- b. For maximizing nodes, alpha is updated to the maximum of its current value and the maximum value obtained from child nodes.
- c. For minimizing nodes, beta is updated to the minimum of its current value and the minimum value obtained from child nodes.

4. Improved Efficiency:

- a. Alpha-beta pruning eliminates the need to evaluate certain branches, as it recognizes that the final decision cannot be influenced by those branches.
- b. This reduction in search space significantly improves the algorithm's efficiency, especially in deep or complex game trees.

Example:

A maximizing player knows that, at the next step, the minimizing player will try to achieve the lowest score.

Suppose the maximizing player has three possible actions, and the first one is valued at 4. Then the player starts generating the value for the next action. To do this, the player generates the values of the minimizer's actions if the current player makes this action, knowing that the minimizer will choose the lowest one.

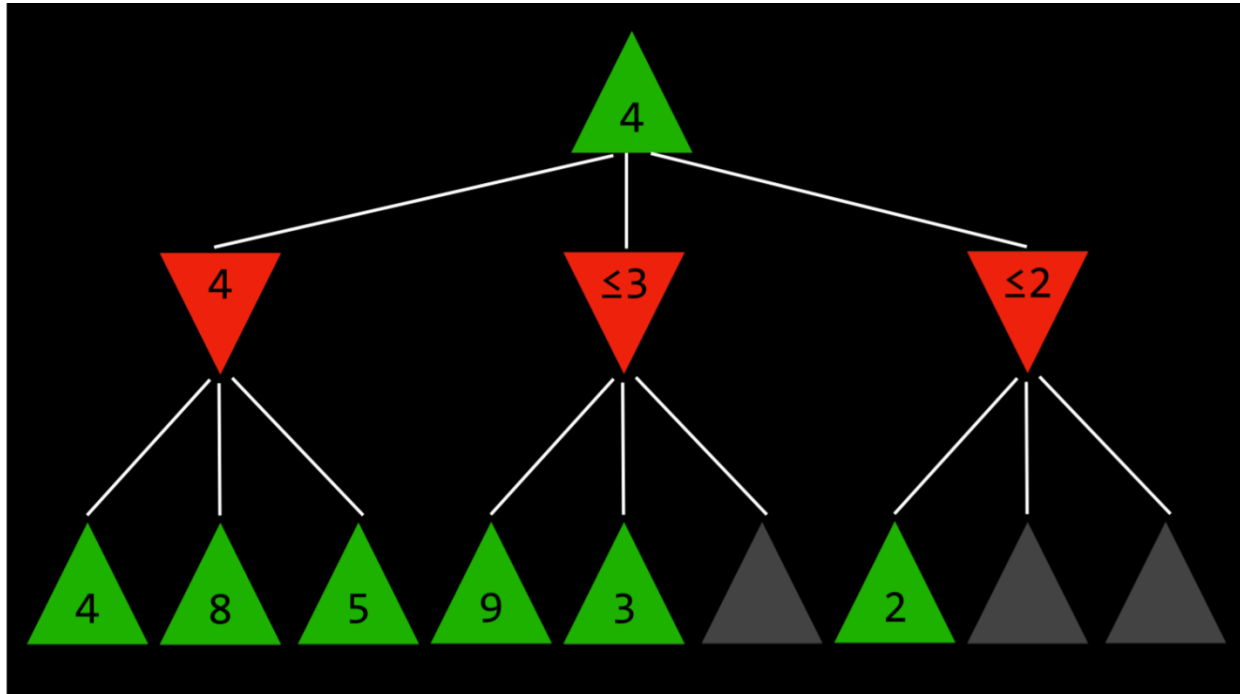
However, before finishing the computation for all the possible actions of the minimizer, the player sees that one of the options has a value of three. This means that there is no reason to keep on exploring the other possible actions for the minimizing player. The value of the not-yet-valued action doesn't matter, be it 10 or (-10).

If the value is 10, the minimizer will choose the lowest option, 3, which is already worse than the preestablished 4.

If the not-yet-valued action would turn out to be (-10), the minimizer will choose this option, (-10), which is even more unfavorable to the maximizer.



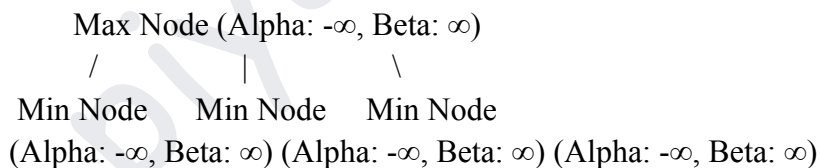
Therefore, computing additional possible actions for the minimizer at this point is irrelevant to the maximizer, because the maximizing player already has an unequivocally better choice whose value is 4.



Example :

An example of the alpha-beta pruning algorithm applied to a simplified game tree. In this example, we'll use a small game tree to demonstrate how alpha-beta pruning efficiently reduces the search space.

Game Tree:



- The game tree consists of three levels: one maximizing node (Max Node) at the top, followed by three minimizing nodes (Min Node) at the second level.

Initial State:

- We start at the root node, a maximizing node (Max Node), with an initial alpha value of negative infinity ($-\infty$) and an initial beta value of positive infinity (∞).

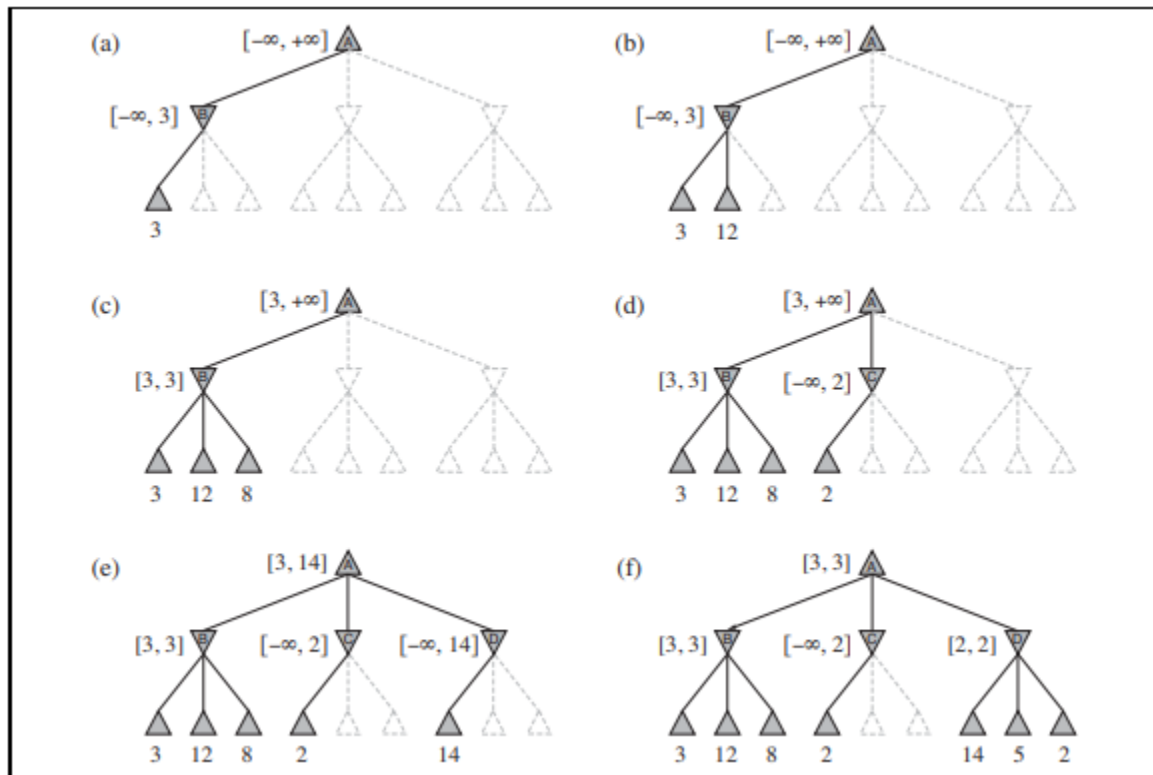
Step 1 - Max Node (Alpha: $-\infty$, Beta: ∞):



- The Max Node examines its first child, a Min Node on the left.
- The Min Node has an initial alpha value of negative infinity ($-\infty$) and an initial beta value of positive infinity (∞).

Step 2 - Min Node (Alpha: $-\infty$, Beta: ∞):

- The Min Node explores its own children.
 - During the search, it evaluates a child node and updates its alpha and beta values based on the evaluation.
- The algorithm starts at the root node, a maximizing node (Max Node), with initial alpha as negative infinity and beta as positive infinity.
- It evaluates the first child node, a minimizing node (Min Node), and updates its alpha and beta values based on the evaluation.



Stages in the calculation of the optimal decision for the game tree in Figure. At each point, we show the range of possible values for each node.

- (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3.
- (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.
- (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.



- (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha-beta pruning.
- (e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.
- (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Performance:

1. **Completeness:** Alpha-beta pruning does not affect the completeness of the minimax algorithm. It guarantees finding the optimal solution if both players play optimally.
2. **Optimality:** Alpha-beta pruning, when used with minimax, ensures optimal decision-making in two-player, zero-sum games with perfect information.
3. **Time Complexity:** The time complexity of alpha-beta pruning reduces the effective branching factor, significantly improving search efficiency. However, it still depends on the depth of the tree.
4. **Space Complexity:** The space complexity is the same as the time complexity, $O(b^d)$, where b is the branching factor, and d is the depth of the tree.

Advantages of Alpha-Beta Pruning:

- **Efficient Pruning:** Alpha-beta pruning dramatically reduces the number of nodes evaluated in the search tree, making it suitable for games with deep or complex trees.
- **Optimality:** When used within the minimax framework, alpha-beta pruning guarantees the same optimal solution as a full minimax search but with fewer nodes evaluated.
- **Practical Applicability:** Alpha-beta pruning is widely used in various board games, including chess, to make the search feasible within a reasonable time frame.



Disadvantages of Alpha-Beta Pruning:

- **Requires Perfect Information:** Alpha-beta pruning assumes perfect play from both players, which may not hold in real-world scenarios.
- **Complexity in Multiplayer Games:** Adapting alpha-beta pruning to multiplayer games with more than two players can be challenging.
- **Not Suitable for Non-Deterministic Games:** Alpha-beta pruning is most effective in deterministic games where chance does not play a role.
- **Heuristic Quality:** The quality of the heuristic evaluation function used in conjunction with alpha-beta pruning can affect its effectiveness and the quality of the decisions made.

3.3 Depth-Limited Minimax

(Just read and understand, no need to go in details)

There is a total of 255,168 possible Tic Tac Toe games, and 10^{29000} possible games in Chess. The minimax algorithm, as presented so far, requires generating all hypothetical games from a certain point to the terminal condition.

While computing all the Tic-Tac-Toe games doesn't pose a challenge for a modern computer, doing so with chess is currently impossible.

Depth-limited Minimax considers only a pre-defined number of moves before it stops, without ever getting to a terminal state. However, this doesn't allow for getting a precise value for each action, since the end of the hypothetical games has not been reached. To deal with this problem, *Depth-limited Minimax* relies on an evaluation function that estimates the expected utility of the game from a given state, or, in other words, assigns values to states.

For example, in a chess game, a utility function would take as input a current configuration of the board, try to assess its expected utility (based on what pieces each player has and their locations on the board), and then return a positive or a negative value that represents how favorable the board is for one player versus the other. These values can be used to decide on the right action, and the better the evaluation function, the better the Minimax algorithm that relies on it.

References:

- **FIRST COURSE IN ARTIFICIAL INTELLIGENCE.** Deepak Khemani.
- CS50: Introduction to AI course, Harvard University
- Artificial Intelligence: A Modern Approach Textbook by Peter Norvig and Stuart J. Russell
- Artificial Intelligence Book by Saroj Kaushik
- IIT Madras BS in Data Science Degree Lectures