**LECTURE 9**

**ADVERSERIAL SEARCH**

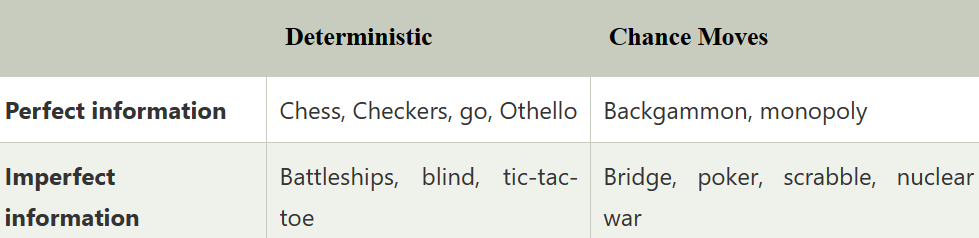
**Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.**

* The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
* So, Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games
* Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

**GAMES AS SEARCH PROBLEMS**

* States are easy to represent (unlike many other real world problems)
* Agents are restricted to a small number of actions
* Outcome of agent actions defined by precise rules
* Usually too hard to solve

**TYPES OF GAMES IN AI**



* Perfect information: A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
* Imperfect information: If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
* Deterministic games: Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
* Non-deterministic games: Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Zero-Sum Game

* Zero-sum games are adversarial search which involves pure competition.
* In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
* One player of the game try to maximize one single value, while other player tries to minimize it.
* Each move by one player in the game is called as ply.
* Chess and tic-tac-toe are examples of a Zero-sum game.

### **Zero-sum game: Embedded thinking**

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

* What to do.
* How to decide the move
* Needs to think about his opponent as well
* The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

**A game can be defined as a type of search in AI which can be formalized of the following elements:**

* **Initial state:** It specifies how the game is set up at the start.
* **Player(s):** It specifies which player has moved in the state space.
* **Action(s):** It returns the set of legal moves in state space.
* **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
* **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
* **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

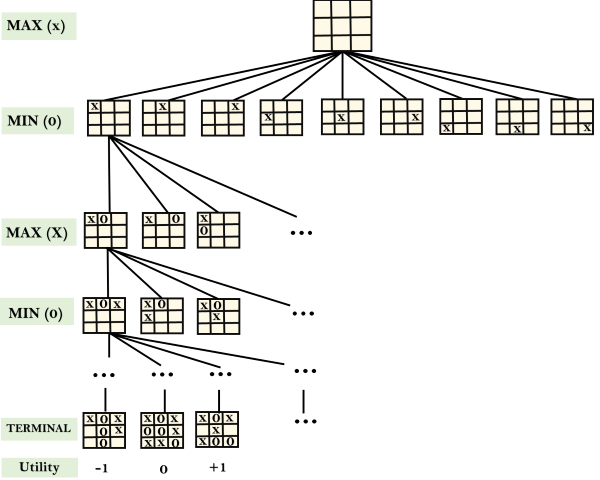
**GAME TREE**

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

**Example: Tic-Tac-Toe game tree:**

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

* There are two players MAX and MIN.
* Players have an alternate turn and start with MAX.
* MAX maximizes the result of the game tree
* MIN minimizes the result.



* From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
* Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
* Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
* So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
* In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

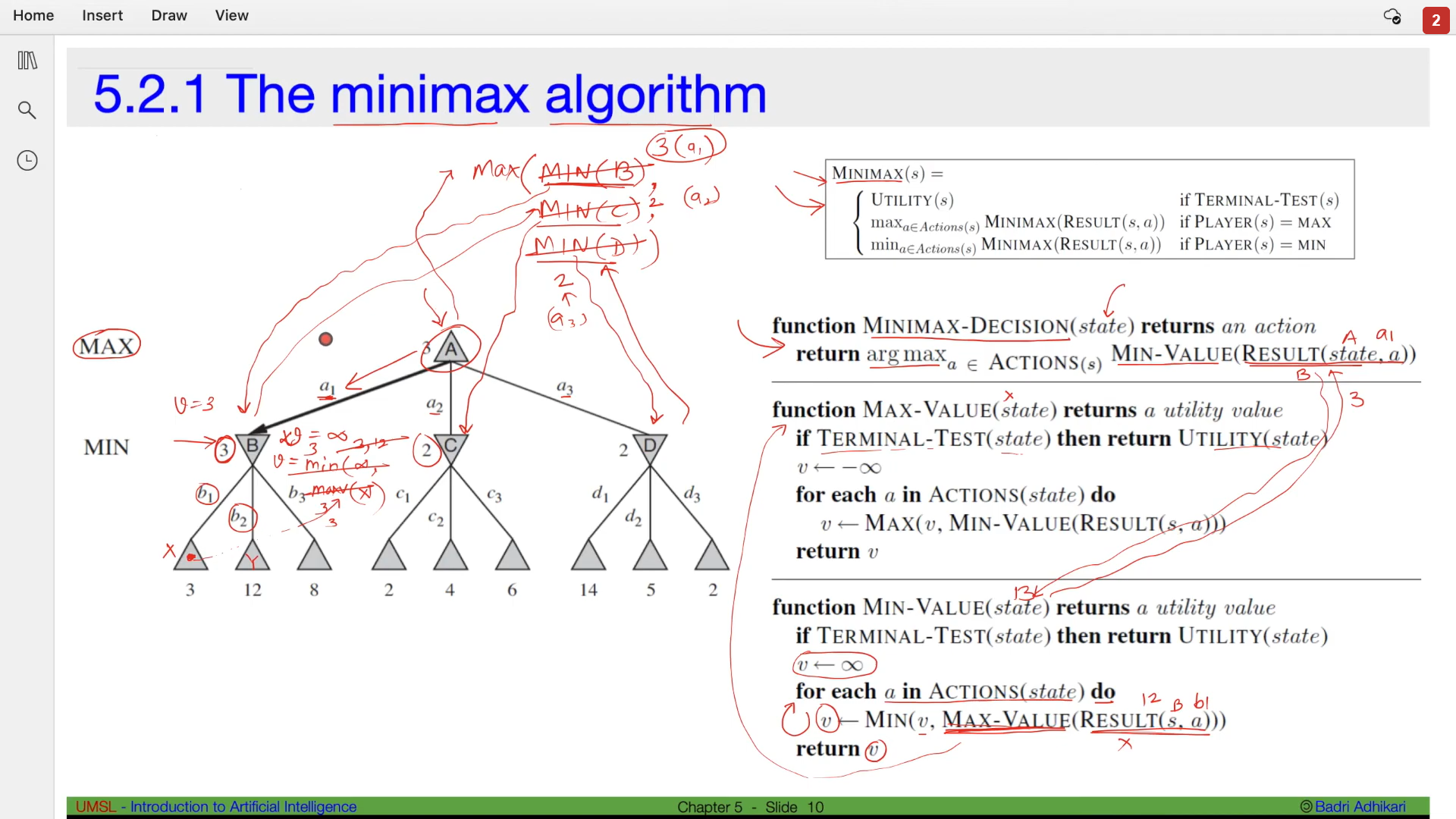
* It aims to find the optimal strategy for MAX to win the game.
* It follows the approach of Depth-first search.
* In the game tree, optimal leaf node could appear at any depth of the tree.
* Propagate the minimax values up to the tree until the terminal node discovered.

How many nodes in this game tree?

• 255, 168 leaf nodes

**MINIMAX ALGORITHM**

* Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
* Mini-Max algorithm uses recursion to search through the game-tree.
* Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
* In this algorithm two players play the game, one is called MAX and other is called MIN.
* Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
* Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
* The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
* The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.



**Working of Min-Max Algorithm:**

* The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
* In this example, there are two players one is called Maximizer and other is called Minimizer.
* Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
* This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
* At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

**Step-1:** In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



**Step 2:** Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

* 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 3:** In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

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



**Step 4:** Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

* For node A max(4, -3)= 4



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

**Properties of Mini-Max algorithm:**

* **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
* **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
* **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **O(bm)**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
* **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

**Limitation of the minimax Algorithm:**

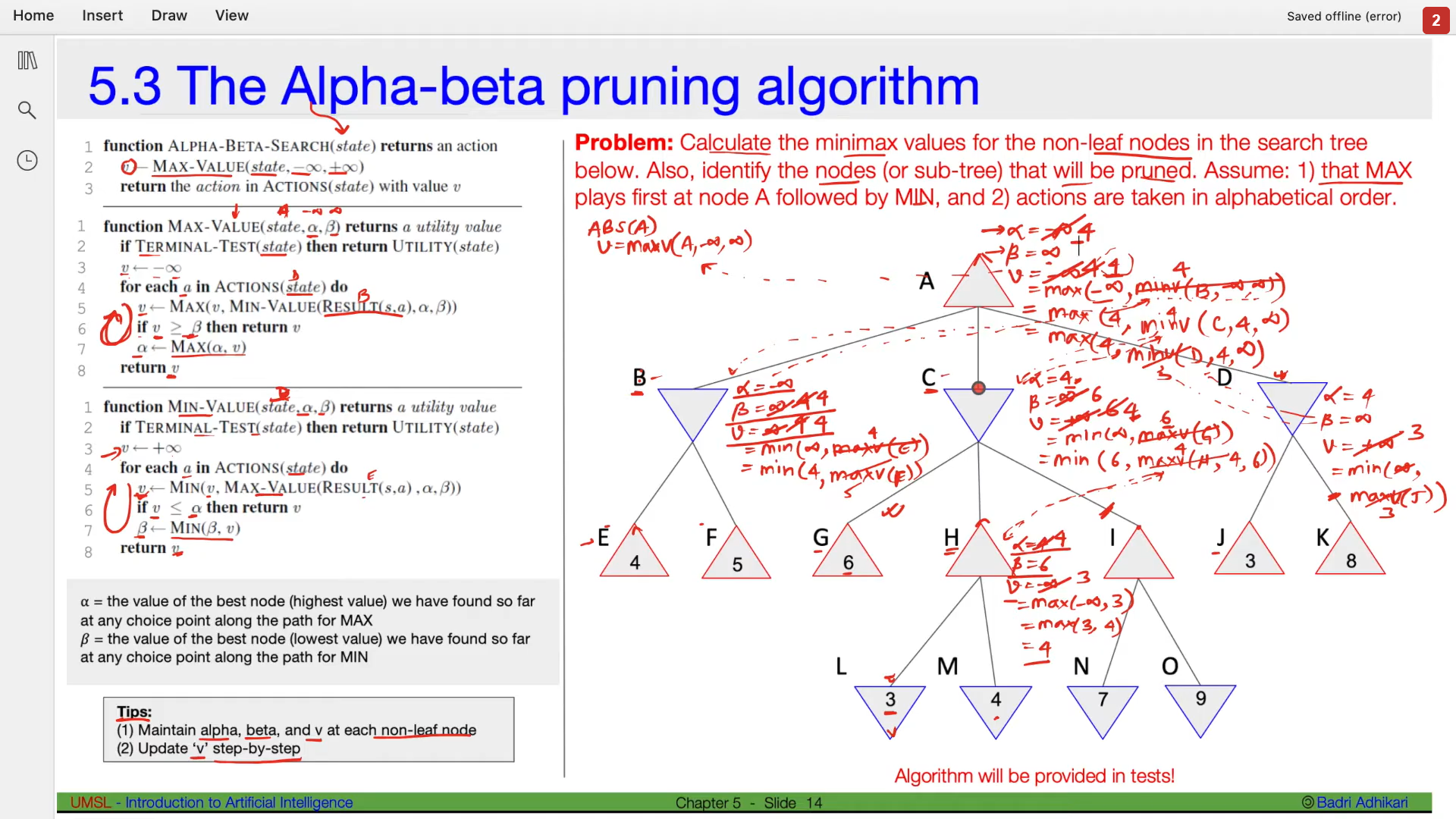
The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from alpha-beta pruning which we have discussed in the next topic.

**PSEUDOCODE**

1. function minimax(node, depth, maximizingPlayer) is
2. **if** depth ==0 or node is a terminal node then
3. **return** **static** evaluation of node
5. **if** MaximizingPlayer then      // for Maximizer Player
6. maxEva= -infinity
7. **for** each child of node **do**
8. eva= minimax(child, depth-1, **false**)
9. maxEva= max(maxEva,eva)        //gives Maximum of the values
10. **return** maxEva
12. **else**                         // for Minimizer player
13. minEva= +infinity
14. **for** each child of node **do**
15. eva= minimax(child, depth-1, **true**)
16. minEva= min(minEva, eva)         //gives minimum of the values
17. **return** minEva

**ALPHA BETA PRUNING**

* Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
* As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
* Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
* The two-parameter can be defined as:
  1. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  2. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.
* The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.



<https://youtu.be/n6mbgKICrVU>