

Adversarial Search and Game-Playing

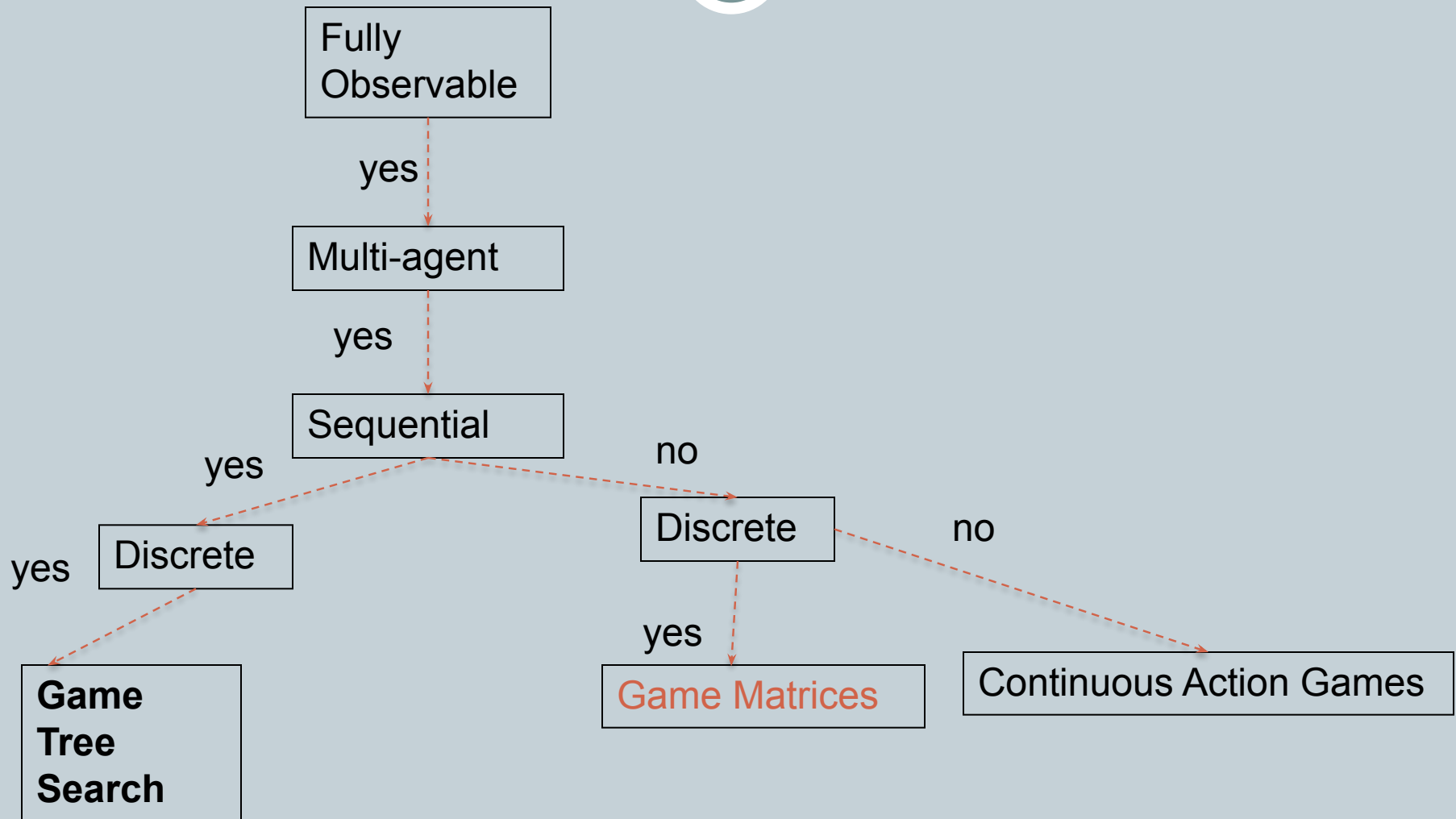
1

CHAPTER 6

Stuart Russell and Peter Norvig, Artificial Intelligence: A
Modern Approach, Global Edition 3/E

Environment Type Discussed In this Lecture

2



Adversarial Search

3

- Examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.
- A good example is in board games.

Typical AI assumptions

4

- Two agents whose actions alternate
- Utility values for each agent are the opposite of the other
 - creates the adversarial situation
- Fully observable environments

Search vs. Games

5

- Search – no adversary

- Solution is (heuristic) method for finding goal
- Heuristic techniques can find *optimal* solution
- Evaluation function: estimate of cost from start to goal through given node
- Examples: path planning, scheduling activities

- Games – adversary

- Solution is **strategy** (strategy specifies move for every possible opponent reply) “unpredictable opponent”
- **Optimality depends on opponent.**
- Time limits force an *approximate* solution
- Evaluation function: evaluate “goodness” of game position
- Examples: chess, checkers, Othello, backgammon

Types of Games

6

	deterministic	Chance moves
Perfect information	Chess, checkers, go, othello	Backgammon, monopoly
Imperfect information (Initial Chance Moves)	Bridge, Skat, blind tic tac to	Poker, scrabble, blackjack

<https://playtictactoe.org/>

<https://cardgames.io/backgammon/>

Game Setup

7

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
 - Winner gets award, loser gets penalty.
- Games as search:
 - Initial state: e.g. board configuration of chess
 - Successor function: list of (move,state) pairs specifying legal moves.
 - Terminal test: Is the game finished?
 - Utility function: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe or chess

Size of search trees

8

- b = branching factor
- d = number of moves by both players
- Search tree is $O(b^d)$
- Chess
 - $b \sim 35$
 - $d \sim 100$
 - search tree is $\sim 10^{154}$ (!!)
 - completely impractical to search this
- Game-playing emphasizes being able to make optimal decisions in a finite amount of time

Partial Game Tree for Tic-Tac-Toe

9

MAX (X)

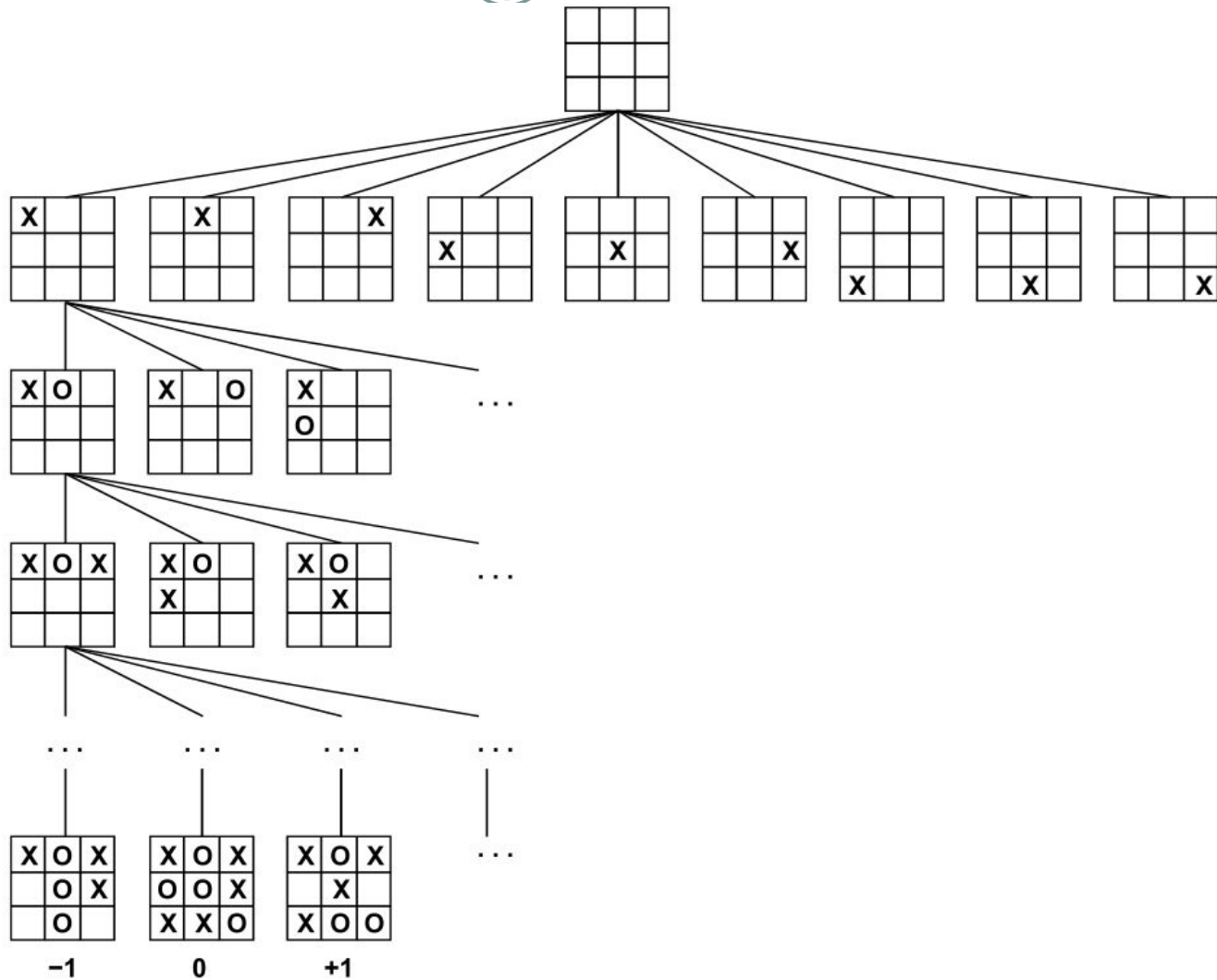
MIN (O)

MAX (X)

MIN (O)

TERMINAL

Utility



Minimax strategy: Look ahead and reason backwards

10

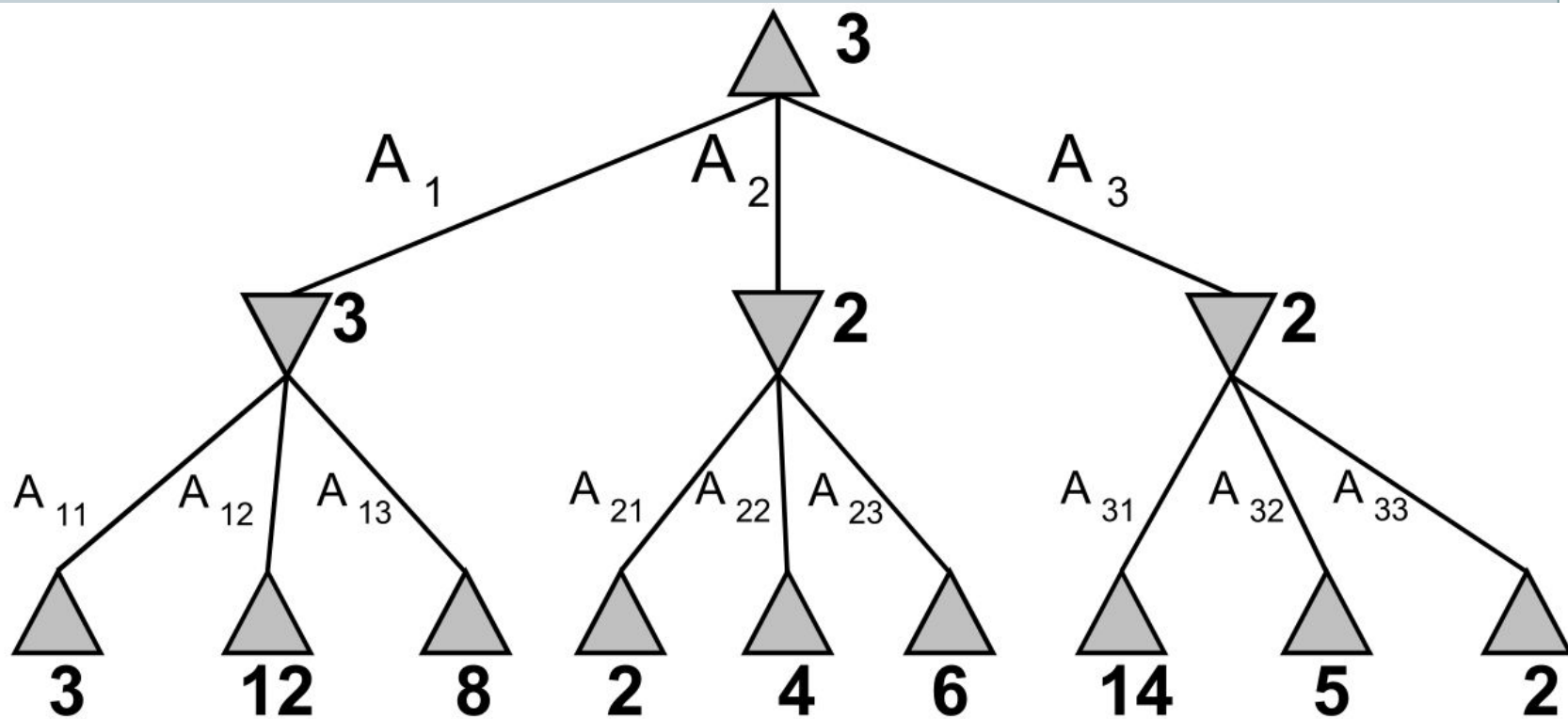
- Find the optimal *strategy* for MAX assuming an infallible MIN opponent
 - Need to compute this all the down the tree
- Assumption: Both players play optimally!
- Given a game tree, the optimal strategy can be determined by using the **minimax value of each node**.

MinMax

11

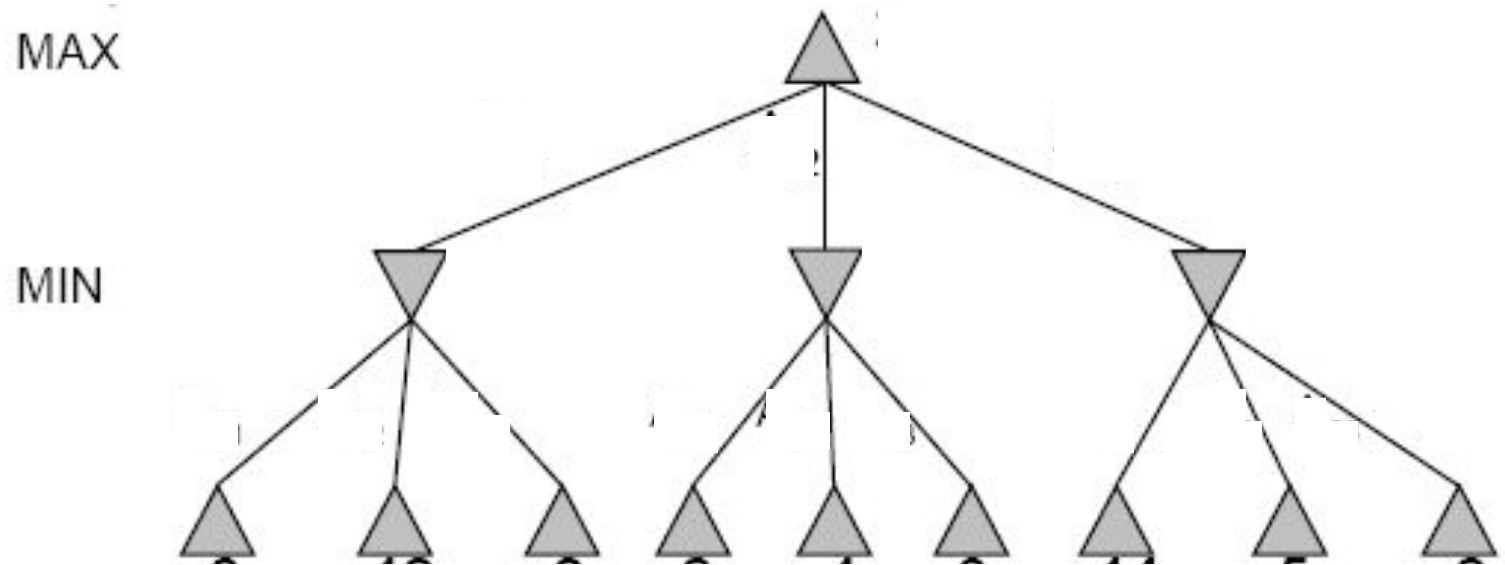
MAX

MIN



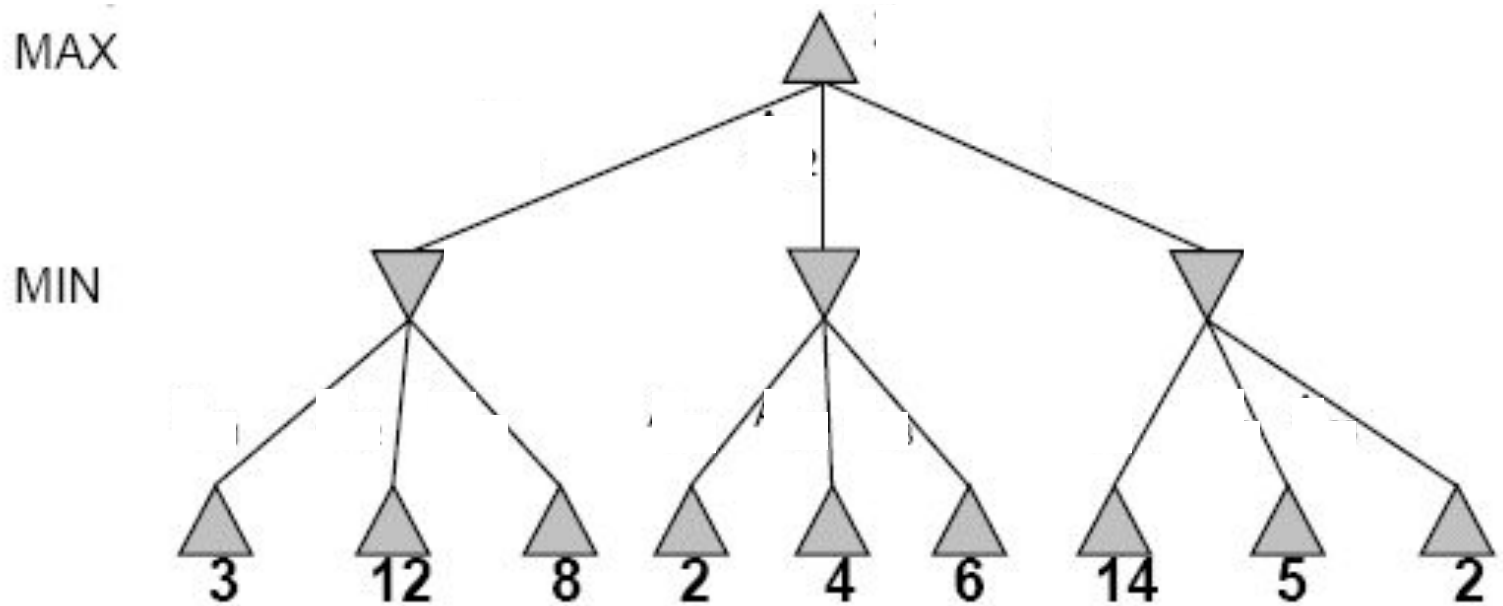
Two-Ply Game Tree

12



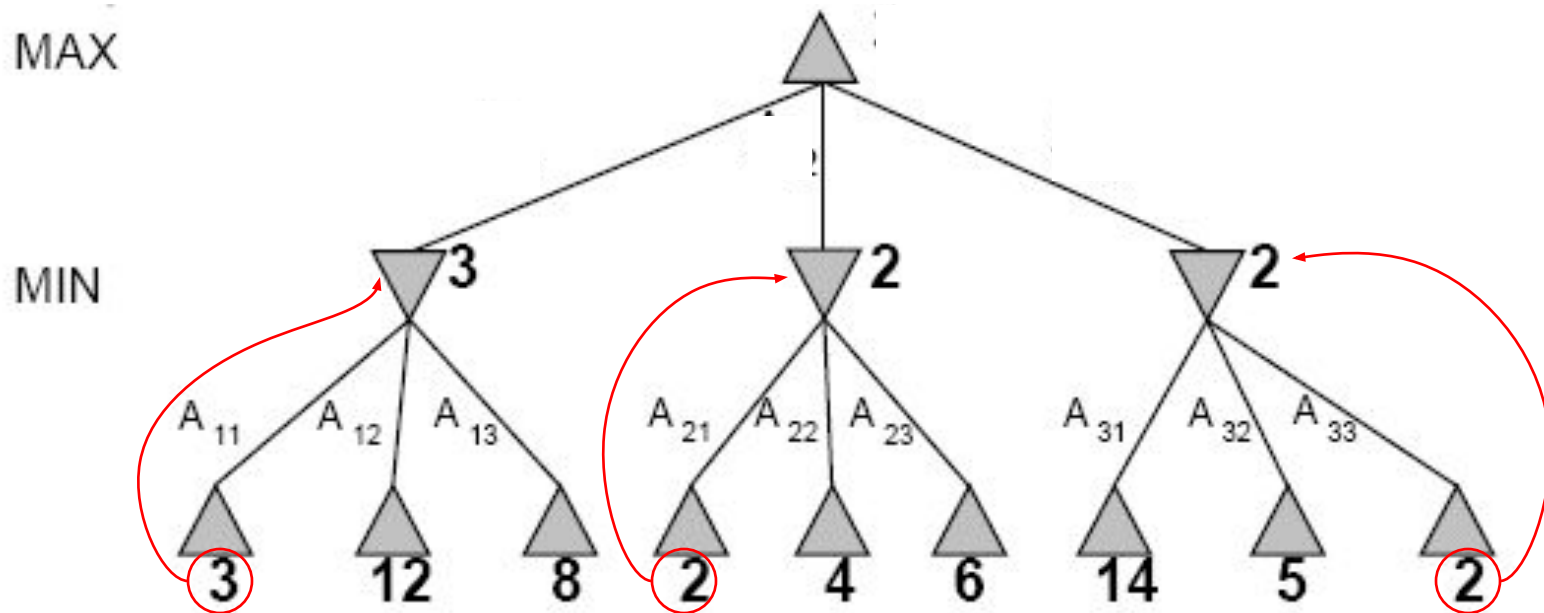
Two-Ply Game Tree

13



Two-Ply Game Tree

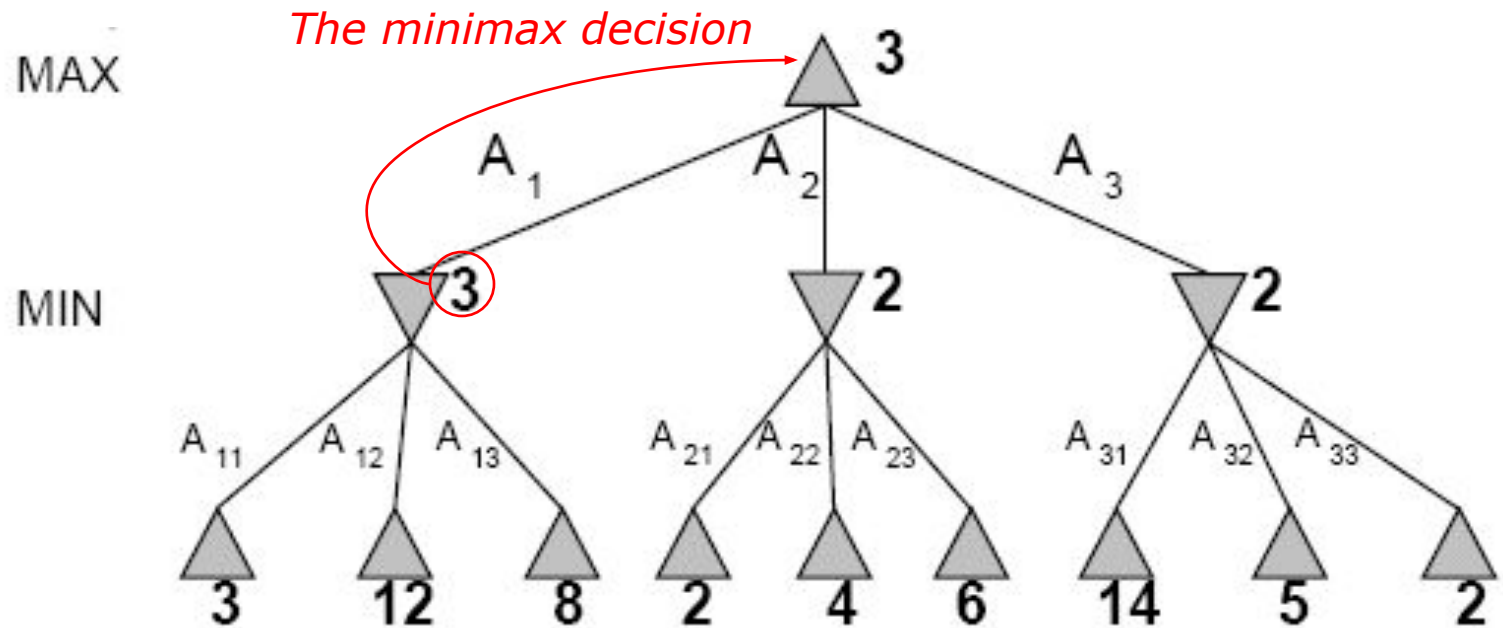
14



Two-Ply Game Tree

15

Minimax maximizes the utility for the worst-case outcome for max



Minmax Algorithm

(16)

function MINIMAX-DECISION(*state*) **returns** *an action*

inputs: *state*, current state in game

return the *a* in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(*a*, *state*))

function MAX-VALUE(*state*) **returns** *a utility value*

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

for *a, s* in SUCCESSORS(*state*) **do** $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$

return *v*

function MIN-VALUE(*state*) **returns** *a utility value*

if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow \infty$

for *a, s* in SUCCESSORS(*state*) **do** $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$

return *v*

Minmax Algorithm Steps

17

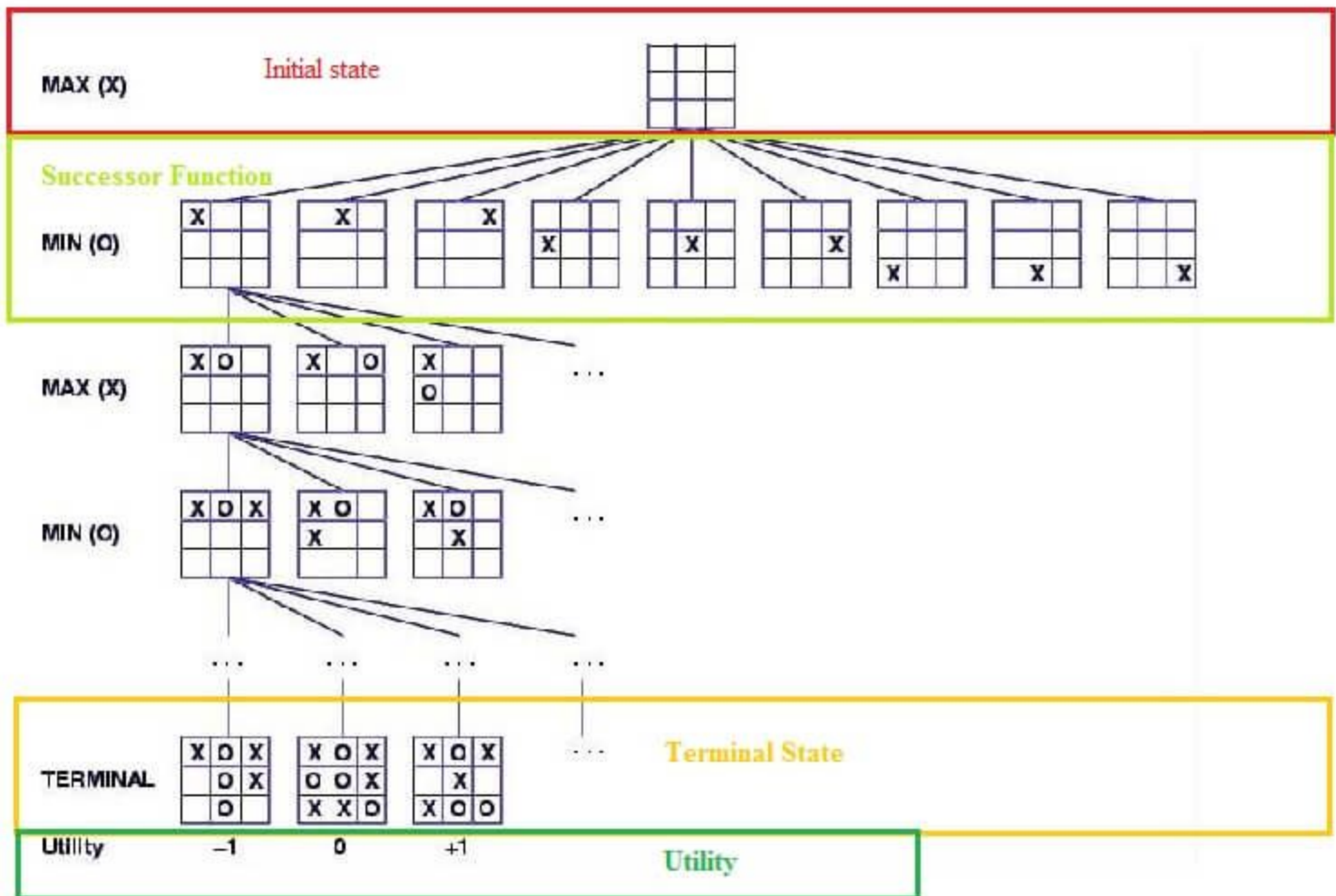
Step 1: First, generate the entire game tree starting with the current position of the game all the way upto the terminal states.

Let us understand the defined **terminology** in terms

1. The **initial state** is the first layer that defines that the board is blank it's MAX's turn to play.
2. **Successor function** lists all the possible successor moves. It is defined for all the layers in the tree.
3. **Terminal State** is the last layer of the tree that shows the final state, i.e whether the player MAX wins, loses, or ties with the opponent.
4. **Utilities** in this case for the terminal states are 1, 0, and -1 as discussed earlier, and they can be used to determine the utilities of the other nodes as well.

step 1: terminologies for tic tac to

18



Minmax Algorithm Steps

19

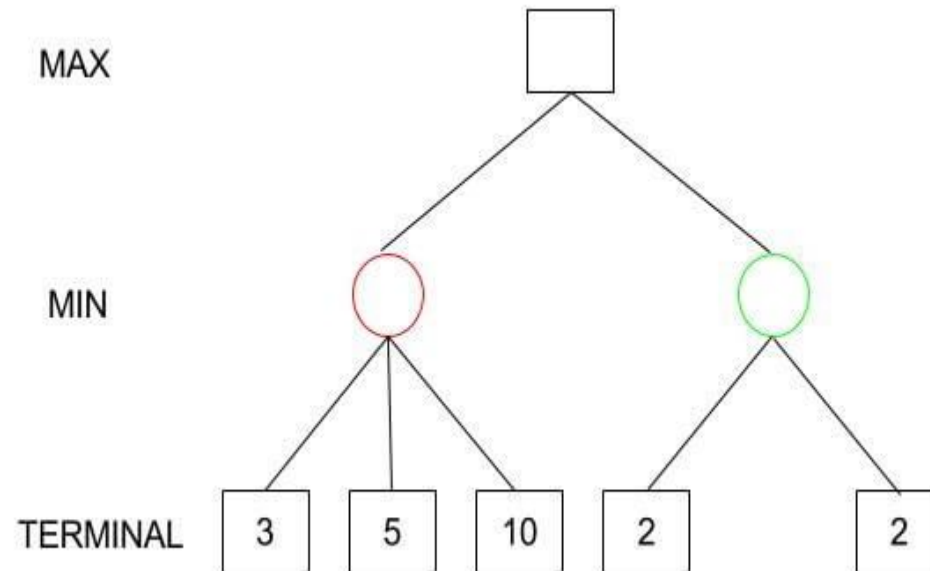
Step 2: Apply the utility function to get the utility values for all the terminal states.

Step 3: Determine the utilities of the higher nodes with the help of the utilities of the terminal nodes.

Calculate the utility for the left node(red) of the layer above the terminal. Since it is the move of the player MIN, we will choose the minimum of all the utilities. For this case, we have to evaluate **MIN{3, 5, 10}**, which we know is certainly 3. So the utility for the red node is 3.

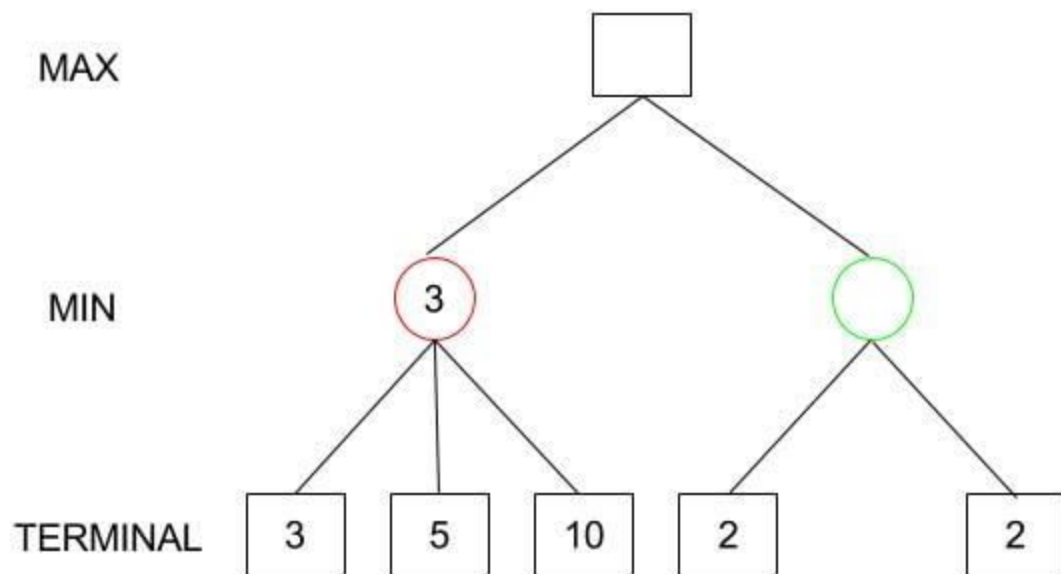
Steps 2, 3

20



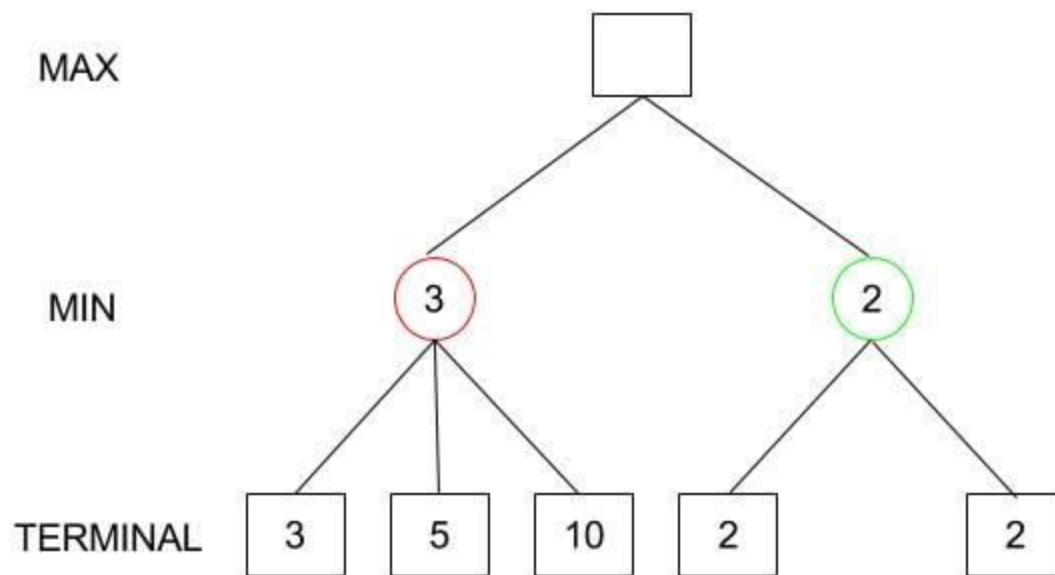
Step 2, 3

21



Step 2, 3

22



Minmax Algorithm Steps

23

Step 4: Calculate the utility values with the help of leaves considering one layer at a time **until the root** of the tree.

Step 5: Eventually, all the backed-up values reach to the root of the tree, i.e., the topmost point. At that point, MAX has to choose the highest value.

Minmax Algorithm Steps

24

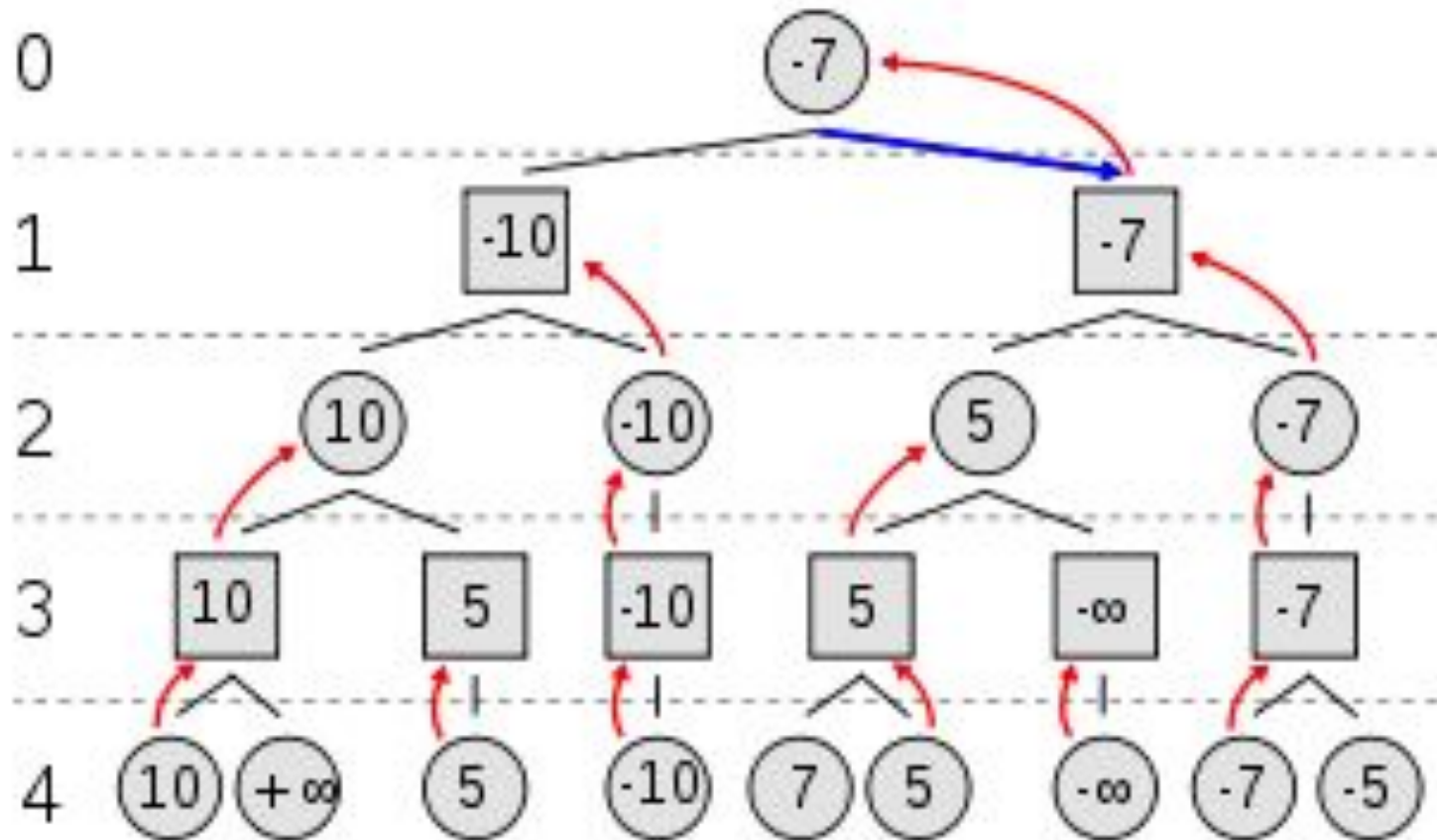
- In our example, we only have 3 layers so we immediately reached to the root but in actual games, there will be many more layers and nodes. So we have to evaluate $\text{MAX}\{3,2\}$ which is 3.
- Therefore, the best opening move for MAX is the left node(or the red one). This move is called the minimax decision as it maximizes the utility following the assumption that the opponent is also playing optimally to minimize it.
- To summarize: Minimax Decision =
 $\text{MAX}\{\text{MIN}\{3,5,10\}, \text{MIN}\{2,2\}\} = \text{MAX}\{3,2\} = 3$

Example of Algorithm Execution

25

$\max(\min(\max(\min(-10, +\infty)), \max(\min$

MAX to move



Properties of minmax



26

- Complete? Yes, if the tree is finite
- Optimal? Yes
- Time complexity? $O(b^m)$
- Space Complexity $O(bm)$

Minimax Algorithm

27

- Complete depth-first exploration of the game tree
- Assumptions:
 - Max depth = d , b legal moves at each point
 - E.g., Chess: $d \sim 100$, $b \sim 35$
 - do we need to explore every path?

Criterion	Minimax
Time 	$O(b^d)$
Space 	$O(bd)$

Practical problem with minimax search

28

- Number of game states is exponential in the number of moves.
- Solution: Do not examine every node
=> pruning
Remove branches that do not influence final decision
- Revisit example ...

Minmax optimization

29

- Game trees are, in general, very time consuming to build, and it's only for simple games that it can be generated in a short time.
- If there are b legal moves, i.e., b nodes at each point and the maximum depth of the tree is m , the time complexity of the minimax algorithm is of the order b^m ($O(b^m)$).
- It is viable to find the actual minimax decision without even looking at every node of the game tree.
- **Hence, we eliminate nodes from the tree without analyzing, and this process is called pruning.**

Alpha-beta pruning

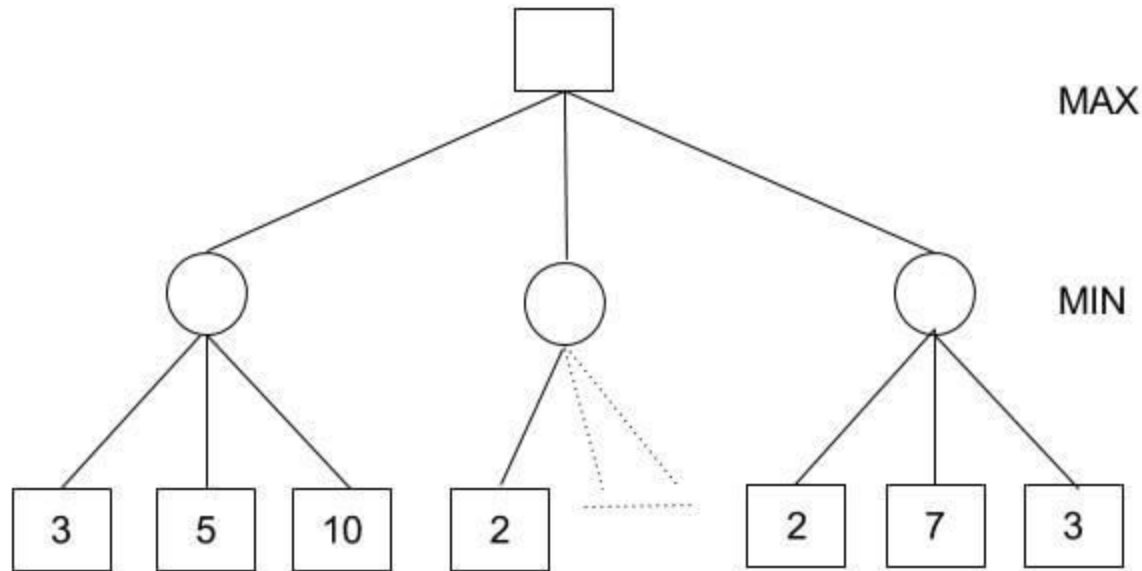
30

- If we apply alpha-beta pruning to a standard minimax algorithm, it returns the same move as the standard one, but it removes (**prunes**) all the nodes that are possibly not affecting the final decision.

Alpha-beta pruning

31

Consider the following tree



Minimax Decision = $\text{MAX}\{\text{MIN}\{3,5,10\}, \text{MIN}\{2,a,b\}, \text{MIN}\{2,7,3\}\} = \text{MAX}\{3,c,2\} = 3$

Alpha-beta pruning

32

$\text{MIN}\{2, a, b\}$ would certainly be less than or equal to 2, i.e., $c \leq 2$ and hence $\text{MAX}\{3, c, 2\}$ has to be 3.

We could have reached a conclusion without looking at those nodes. And this is where alpha-beta pruning comes into the picture.

A few definitions

33

Alpha: It is the best choice so far for the player MAX. We want to get the highest possible value here.

Beta: It is the best choice so far for MIN, and it has to be the lowest possible value.

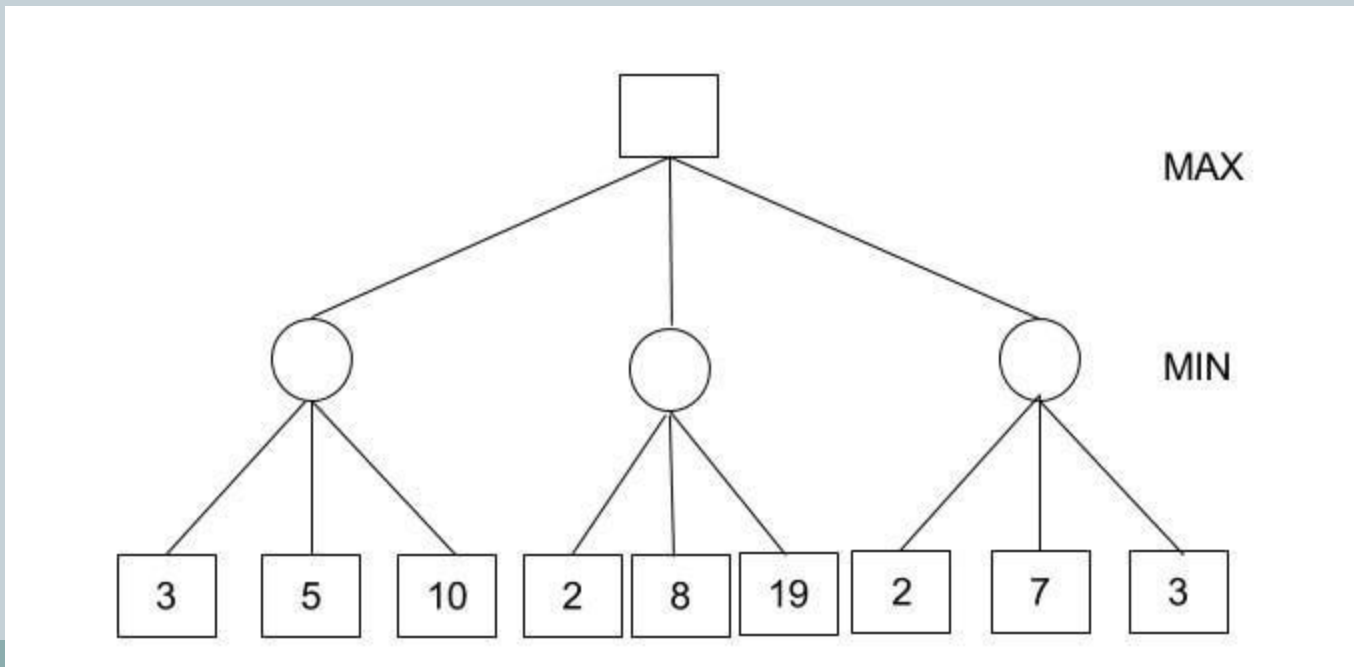
Note: Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and, similarly, beta can be updated only when it's MIN's chance.

Alpha-beta pruning algorithm steps

34

1 - Initialize $\alpha = -\text{infinity}$ and $\beta = \text{infinity}$ as the worst possible cases. The condition to prune a node is when α becomes greater than or equal to β .

$\alpha \geq \beta$



Alpha-beta pruning algorithm steps

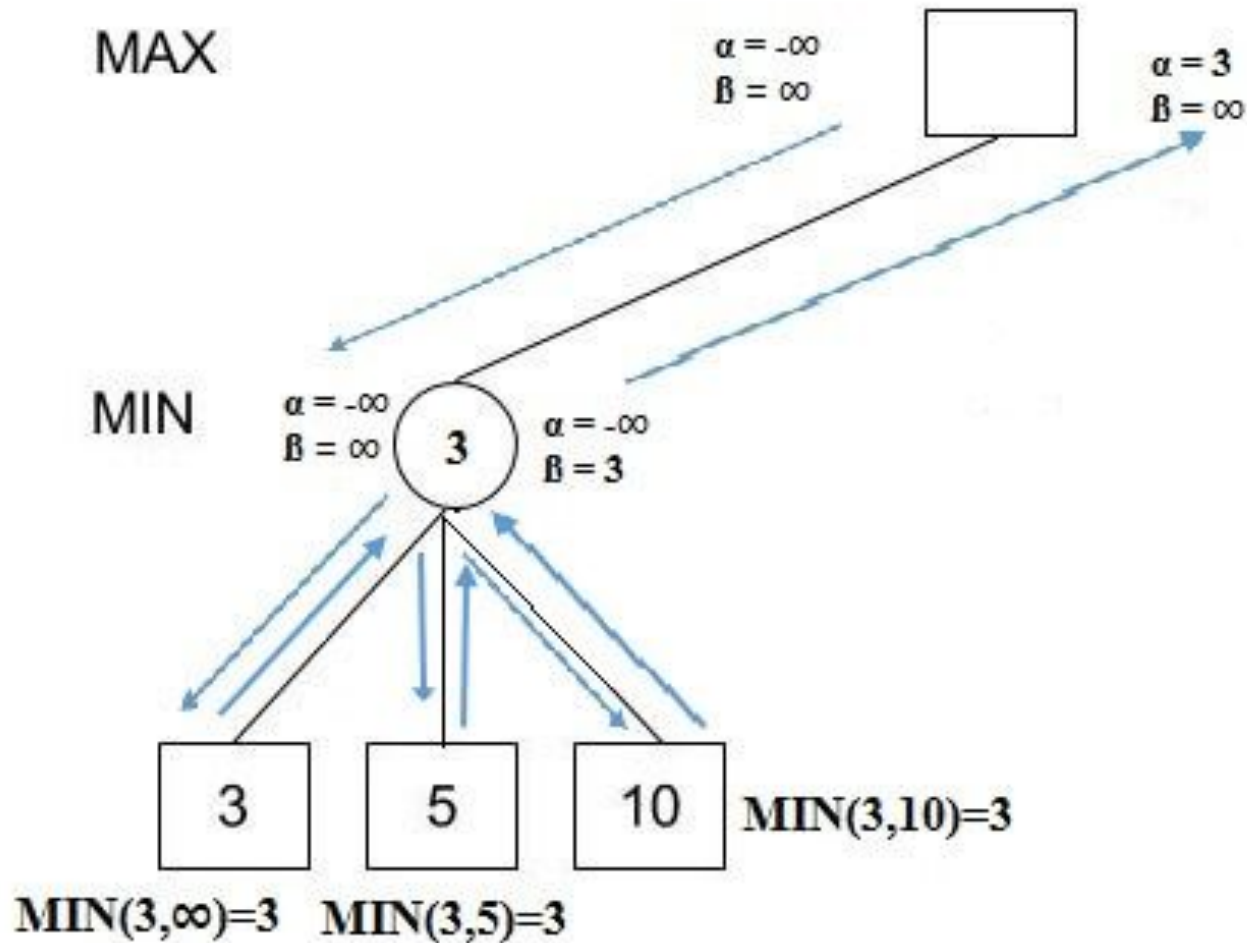
35

2- Start with assigning the initial values of alpha and beta to root and since alpha is less than beta we don't prune it.

3 - Carry these values of alpha and beta to the child node on the left. And now from the utility value of the terminal state, we will update the values of alpha and be, so we don't have to update the value of beta. Again, we don't prune because the condition remains the same. Similarly, the third child node also. And then backtracking to the root we set $\alpha=3$ because that is the minimum value that alpha can have.

Alpha-beta pruning algorithm steps

36



Alpha-beta pruning algorithm steps

37

4 - Now, $\alpha=3$ and $\beta=\text{infinity}$ at the root. So, we don't prune. Carrying this to the center node, and calculating $\text{MIN}\{2, \text{infinity}\}$, we get $\alpha=3$ and $\beta=2$.

5 - Prune the second and third child nodes because α is now greater than β .

6 - α at the root remains 3 because it is greater than 2. Carrying this to the rightmost child node, evaluate $\text{MIN}\{\text{infinity}, 2\}=2$. Update β to 2 and α remains 3.

Alpha-beta pruning algorithm steps

38

7 - Prune the second and third child nodes because alpha is now greater than beta.

8 - Hence, we get 3, 2, 2 at the left, center, and right MIN nodes, respectively. And calculating $\text{MAX}\{3, 2, 2\}$, we get 3. Therefore, without even looking at four leaves we could correctly find the minimax decision.

Alpha-beta pruning algorithm steps

39

function ALPHA-BETA-DECISION(*state*) **returns** an action
return the *a* in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(*a*, *state*))

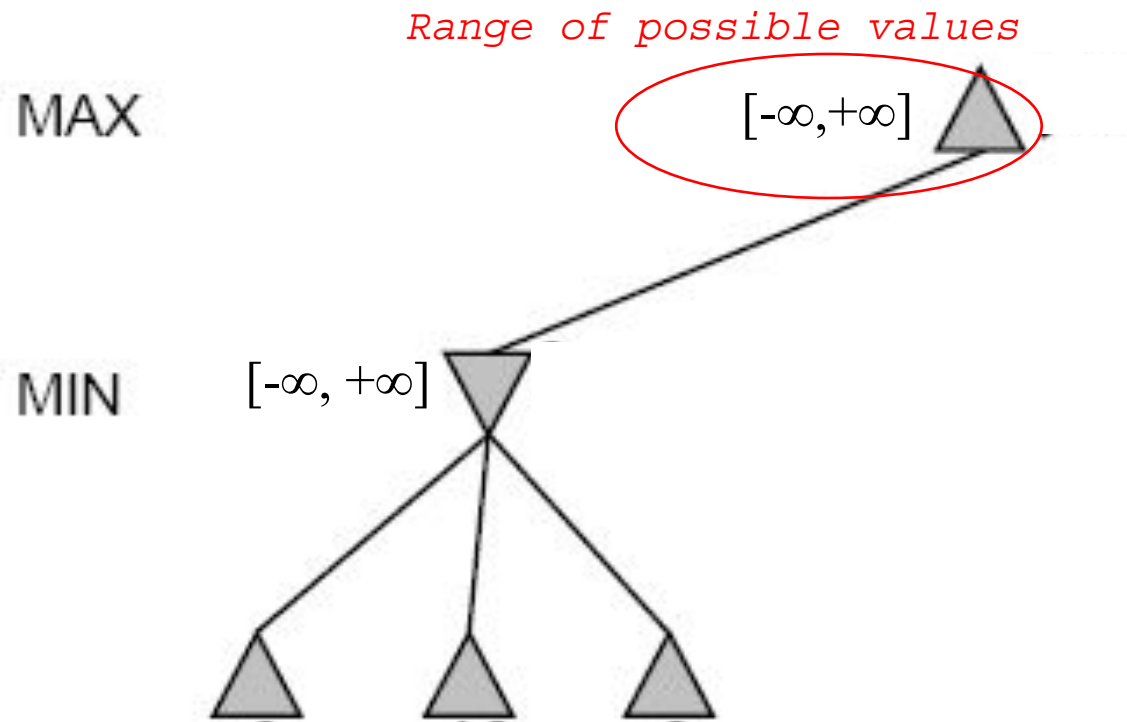
function MAX-VALUE(*state*, α , β) **returns** *a utility value*
inputs: *state*, current state in game
 α , the value of the best alternative for MAX along the path to *state*
 β , the value of the best alternative for MIN along the path to *state*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
 $v \leftarrow -\infty$
for *a*, *s* in SUCCESSORS(*state*) **do**
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$
 if $v \geq \beta$ **then return** *v*
 $\alpha \leftarrow \text{MAX}(\alpha, v)$
return *v*

function MIN-VALUE(*state*, α , β) **returns** *a utility value*
same as MAX-VALUE but with roles of α , β reversed

Alpha-Beta Example

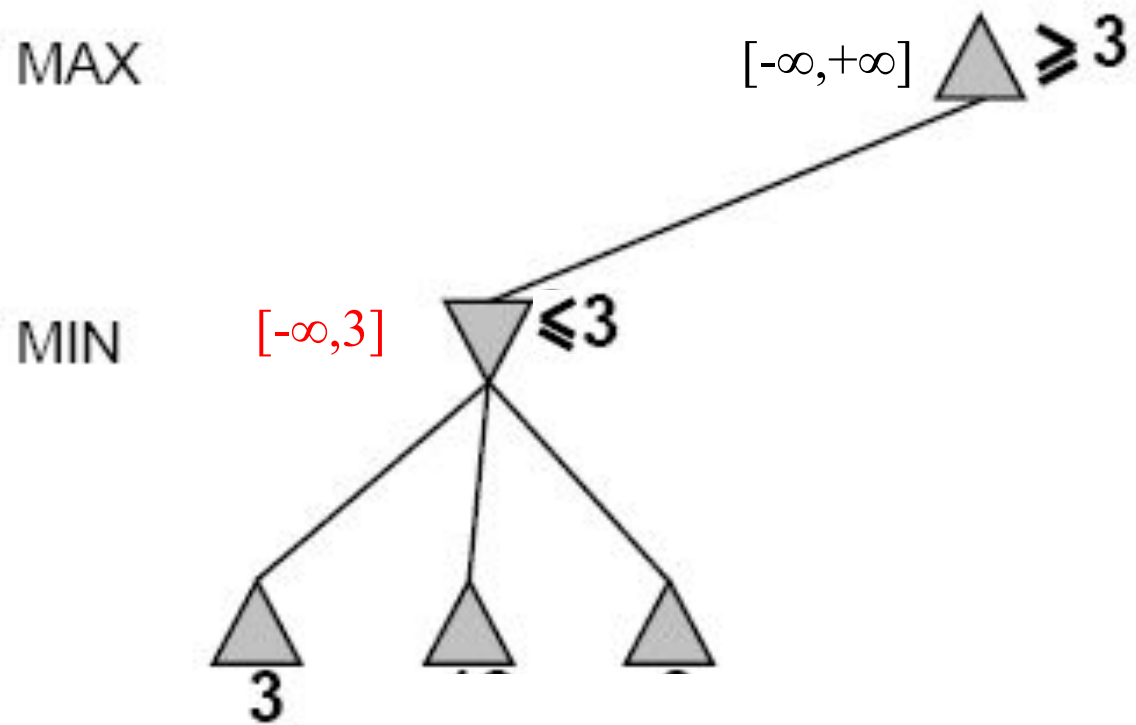
40

Do DF-search until first leaf



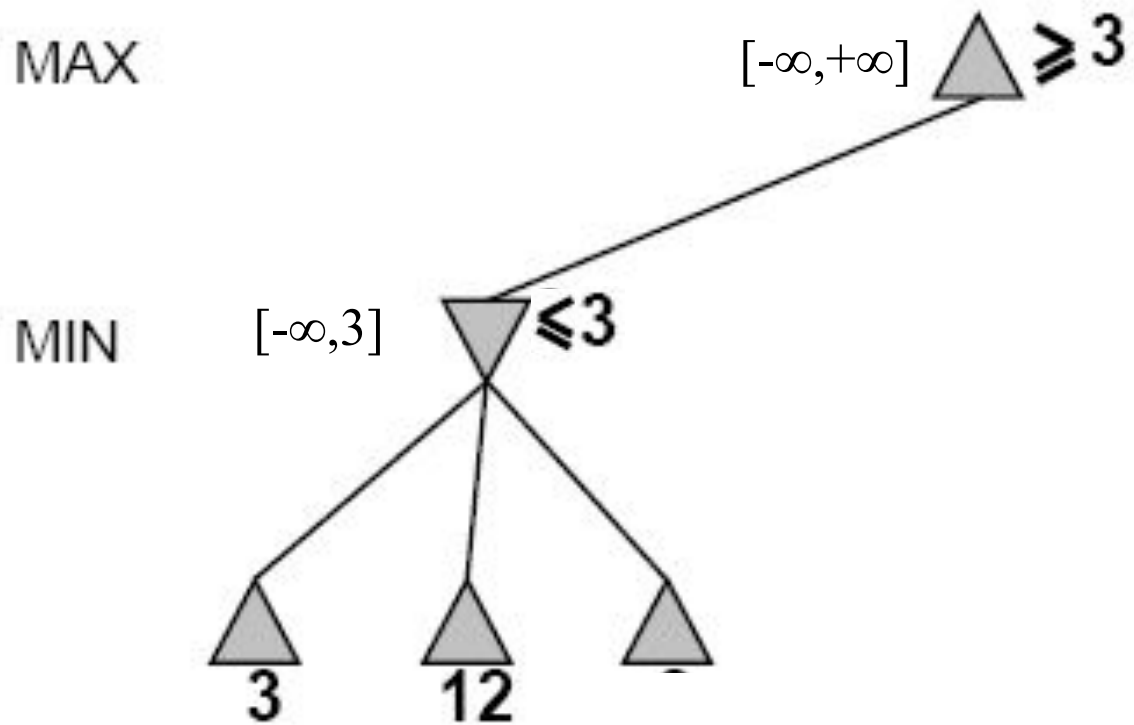
Alpha-Beta Example (continued)

41



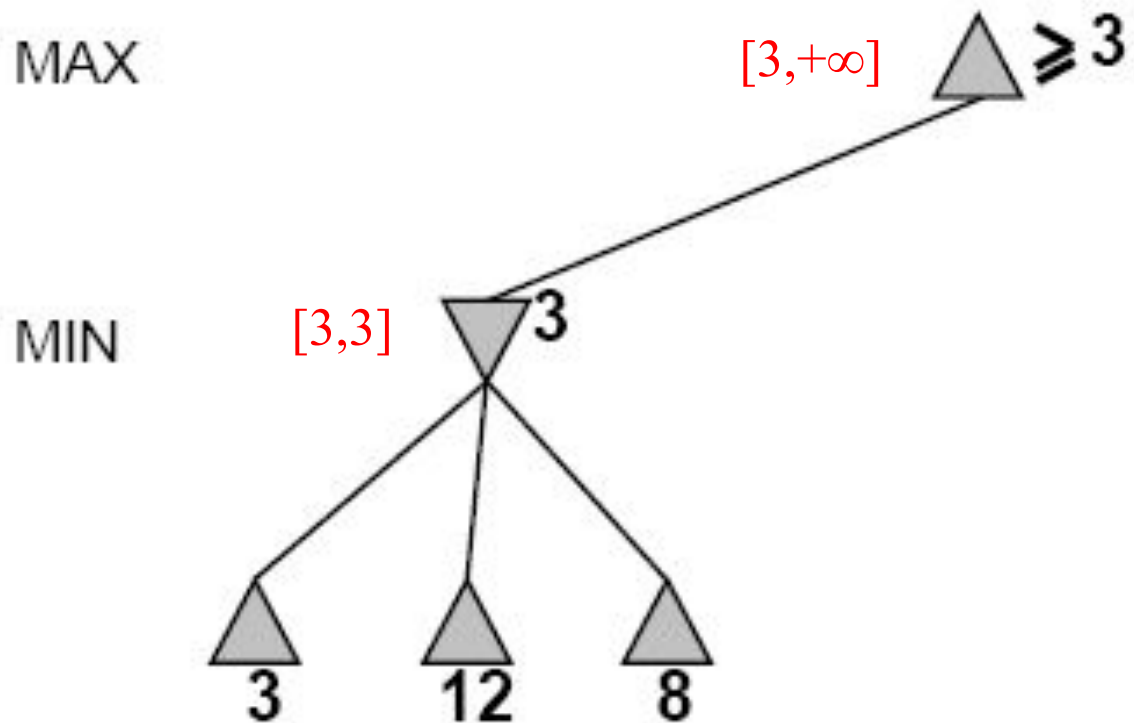
Alpha-Beta Example (continued)

42



Alpha-Beta Example (continued)

43

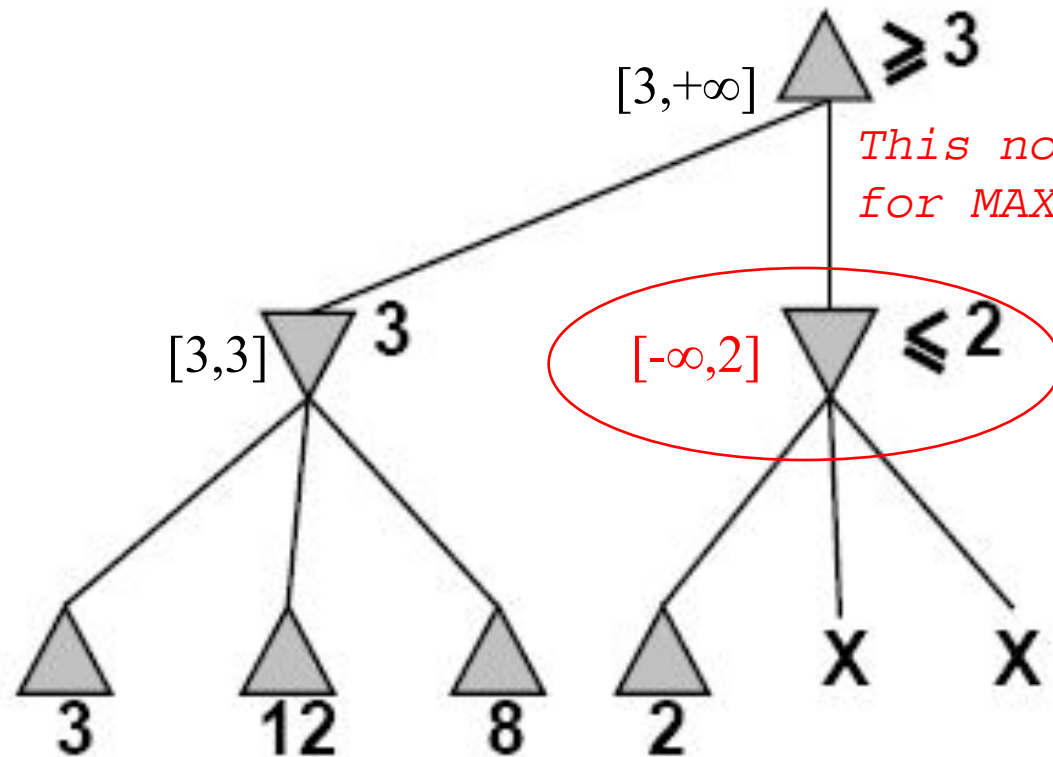


Alpha-Beta Example (continued)

44

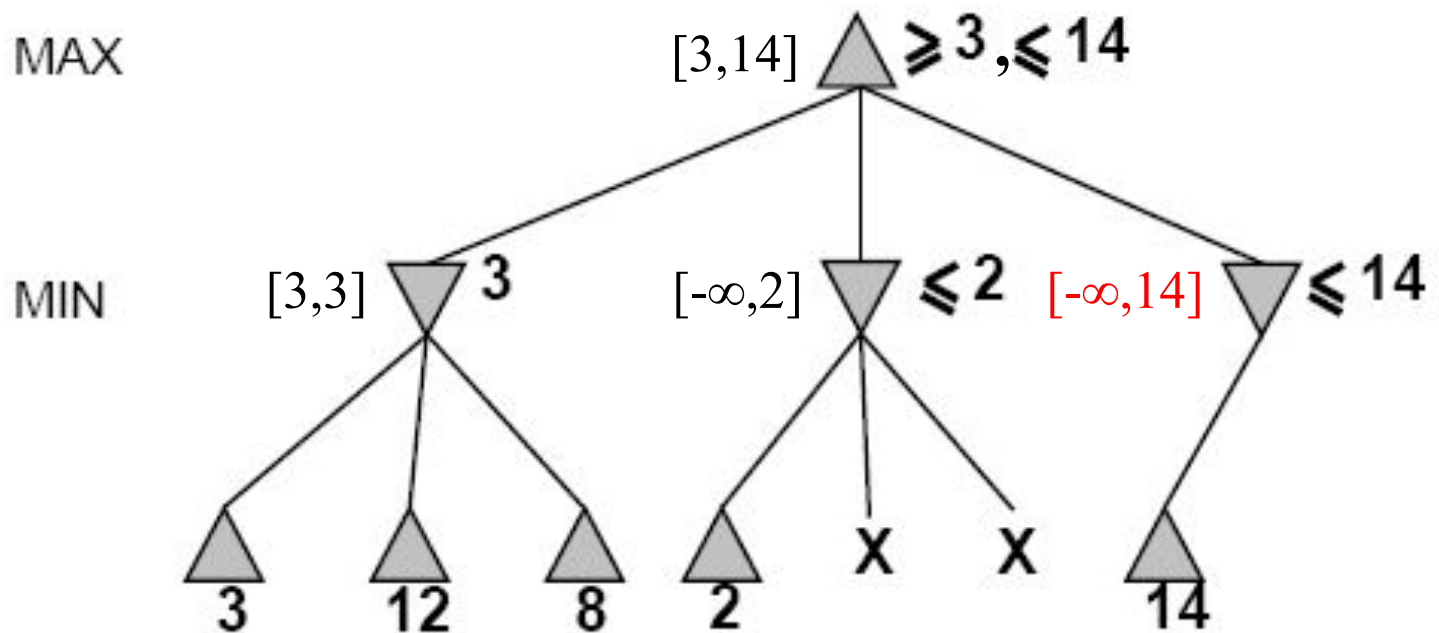
MAX

MIN



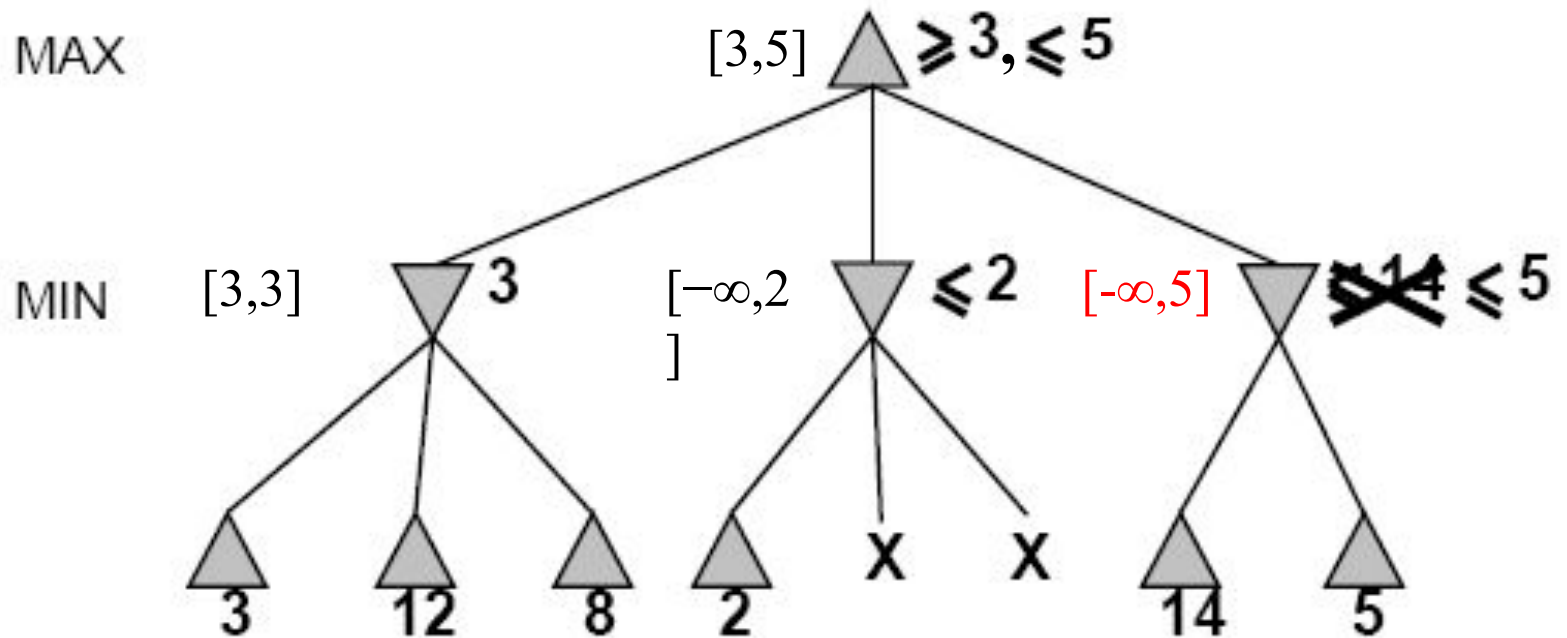
Alpha-Beta Example (continued)

45



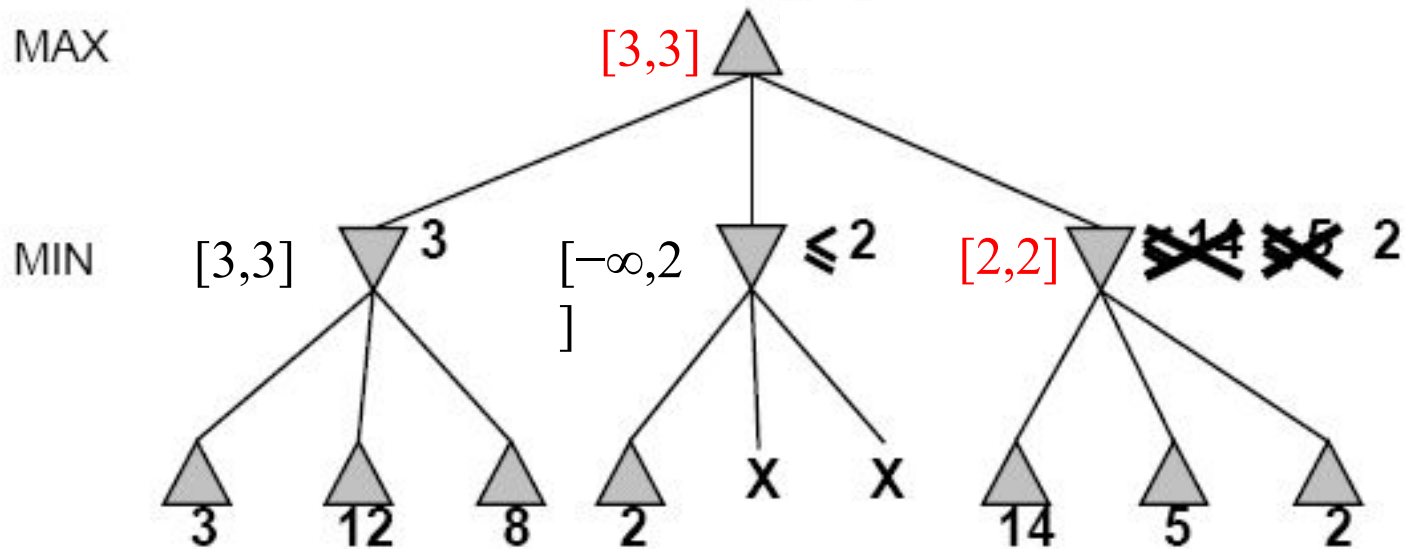
Alpha-Beta Example (continued)

46



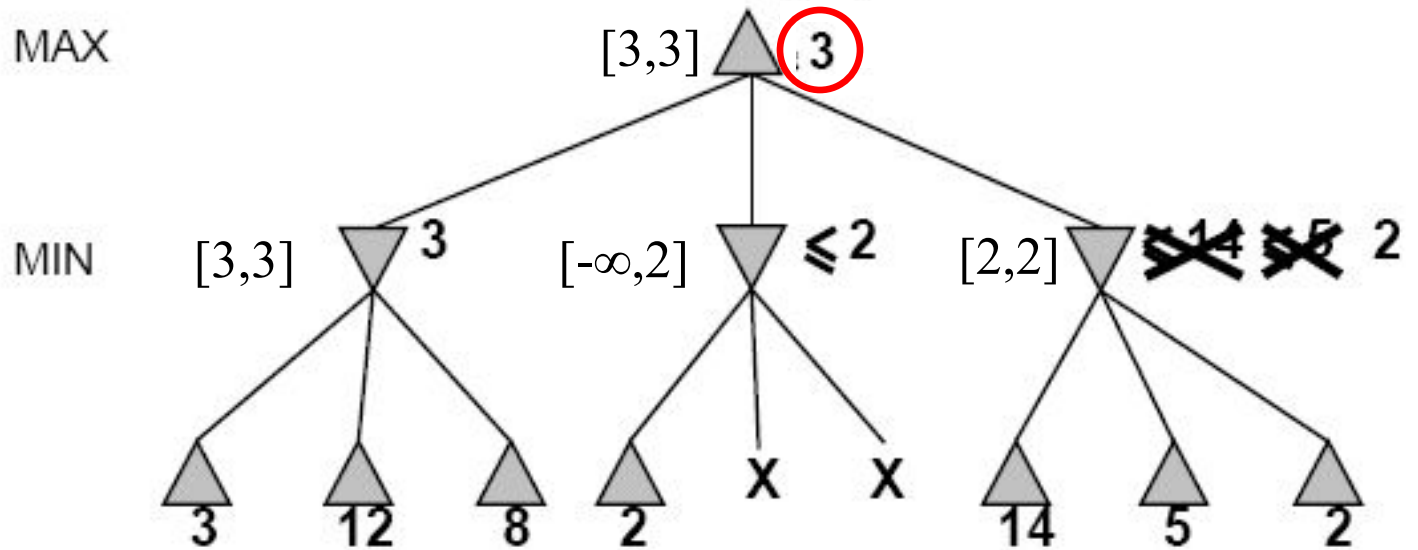
Alpha-Beta Example (continued)

47

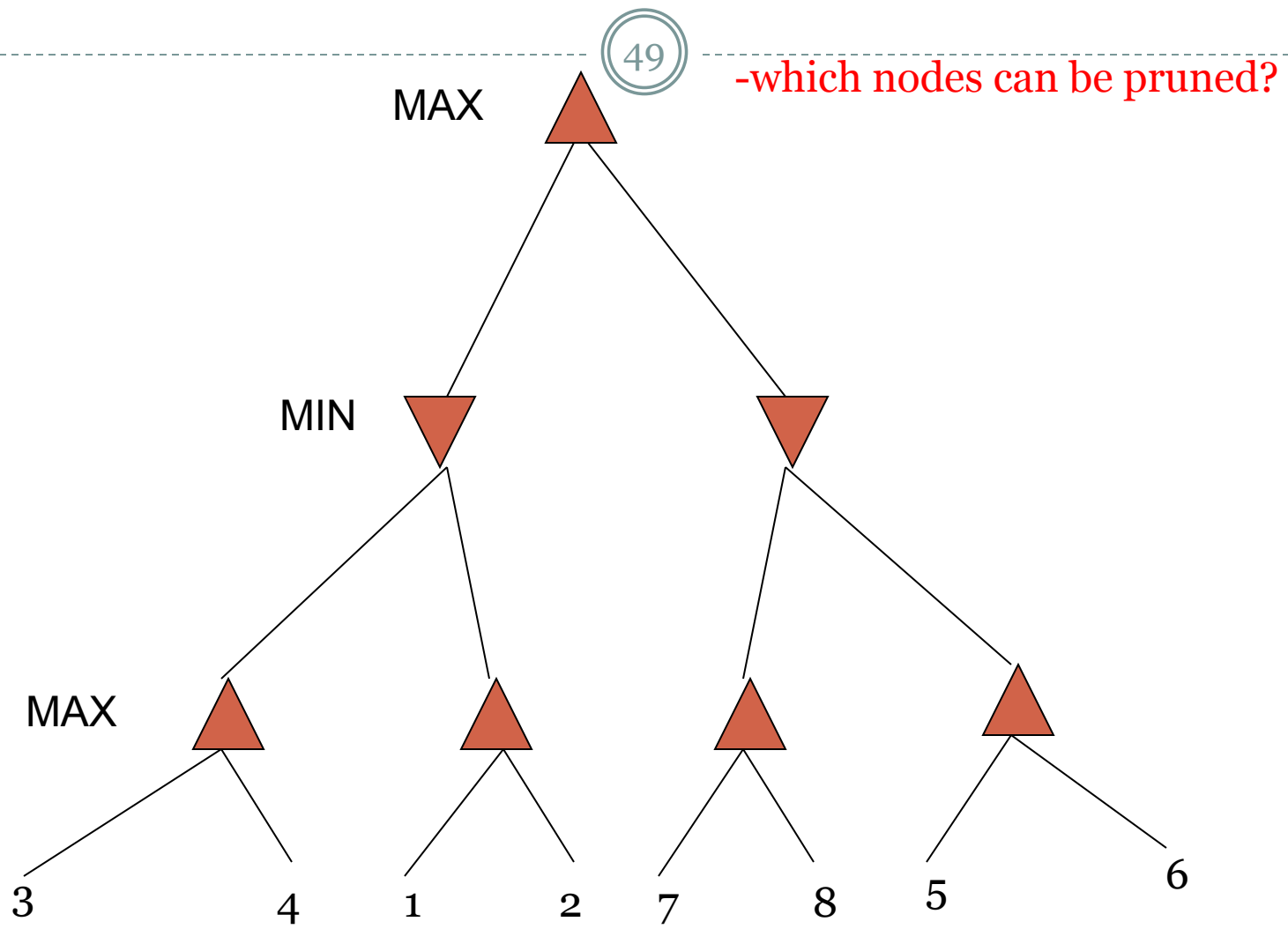


Alpha-Beta Example (continued)

48



Example



Final Comments about Alpha-Beta Pruning

50

- Pruning does not affect final results
- Entire subtrees can be pruned.
- Good move *ordering* improves effectiveness of pruning
- Repeated states are again possible.
 - Store them in memory = transposition table

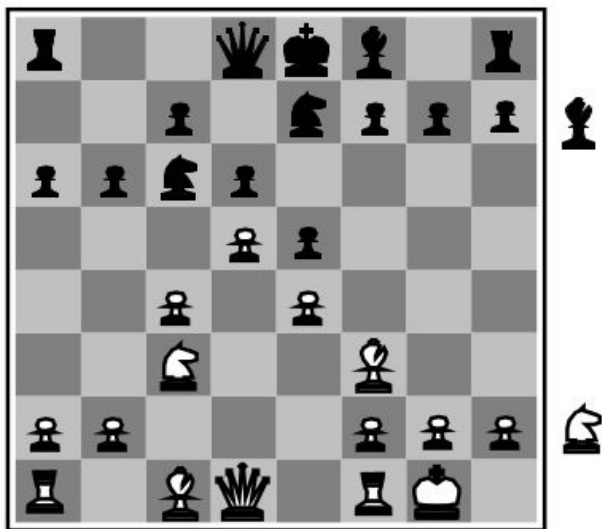
Static (Heuristic) Evaluation Functions

51

- An Evaluation Function:
 - estimates how good the current board configuration is for a player.
 - Typically, one figures how good it is for the player, and how good it is for the opponent, and subtracts the opponents score from the players
 - Othello: Number of white pieces - Number of black pieces
 - Chess: Value of all white pieces - Value of all black pieces
- Typical values from -infinity (loss) to +infinity (win) or $[-1, +1]$.
- If the board evaluation is X for a player, it's $-X$ for the opponent.
- Many clever ideas about how to use the evaluation function.
 - e.g. null move heuristic: let opponent move twice.
- Example:
 - Evaluating chess boards,
 - Checkers
 - Tic-tac-toe

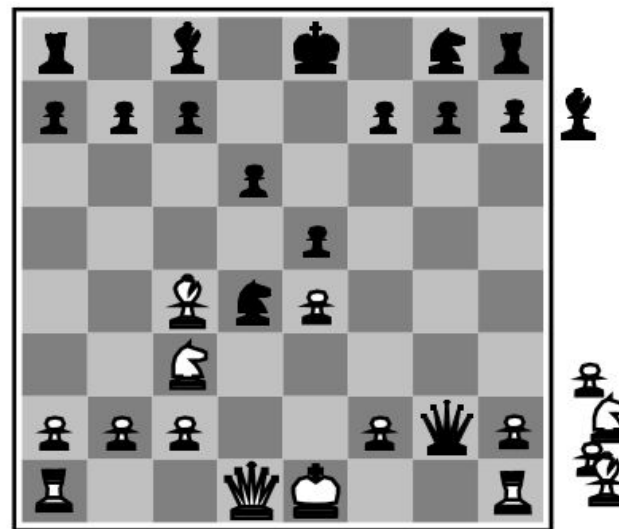
Evaluation functions

52



Black to move

White slightly better



White to move

Black winning

For chess, typically **linear** weighted sum of **features**

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

e.g., $w_1 = 9$ with

$f_1(s) = (\text{number of white queens}) - (\text{number of black queens}), \text{ etc.}$