UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Artificial Intelligence

Adversarial Search

LESSON 7

prof. Antonino Staiano

M.Sc. In ''Machine Learning e Big Data'' - University Parthenope of Naples

Adversarial Search

- The algorithms discussed so far need to find an answer to a question
- In adversarial search, the algorithm faces an opponent that tries to achieve the opposite goal
- Often, adversarial search is encountered in games

Types of Games

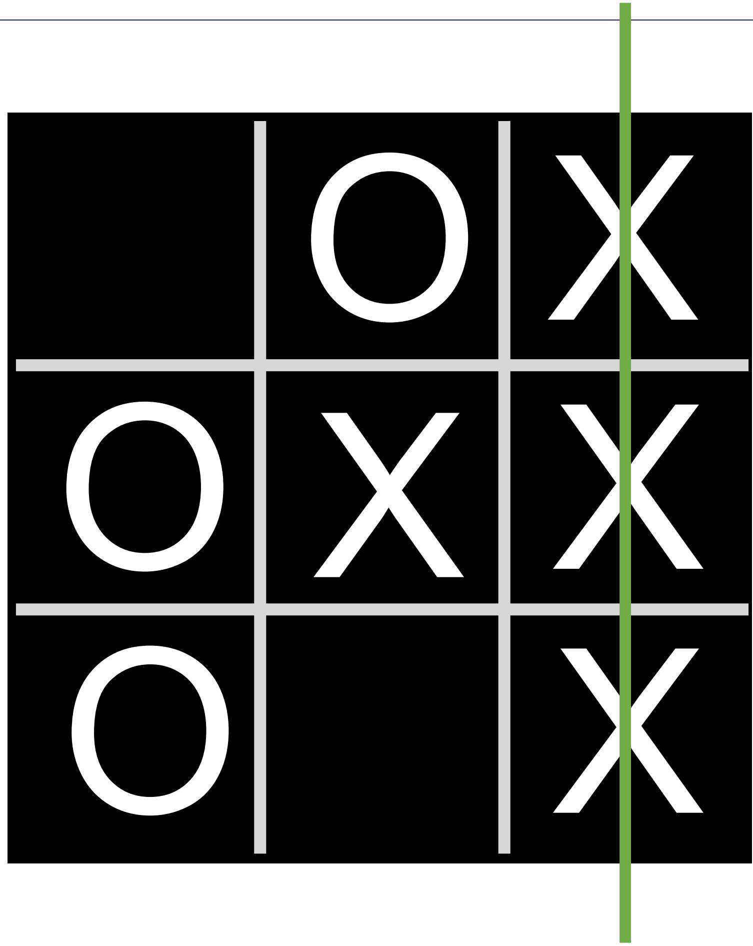
	deterministic	chance
perfect information	chess, checkers, go, othello	Backgammon, monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble

Perfect Information Zero-Sum Games

- The games most studied within AI (such as chess and Go) are
  - deterministic, two-player turn-taking, perfect information, zero-sum games
- Perfect information
  - Synonym for fully observable
- Zero-sum
  - means that what is good for one player is just as bad for the other
    - there is no "win-win" outcome
- Terminology
  - Move -> action
  - Position -> state

Tic-Tac-Toe

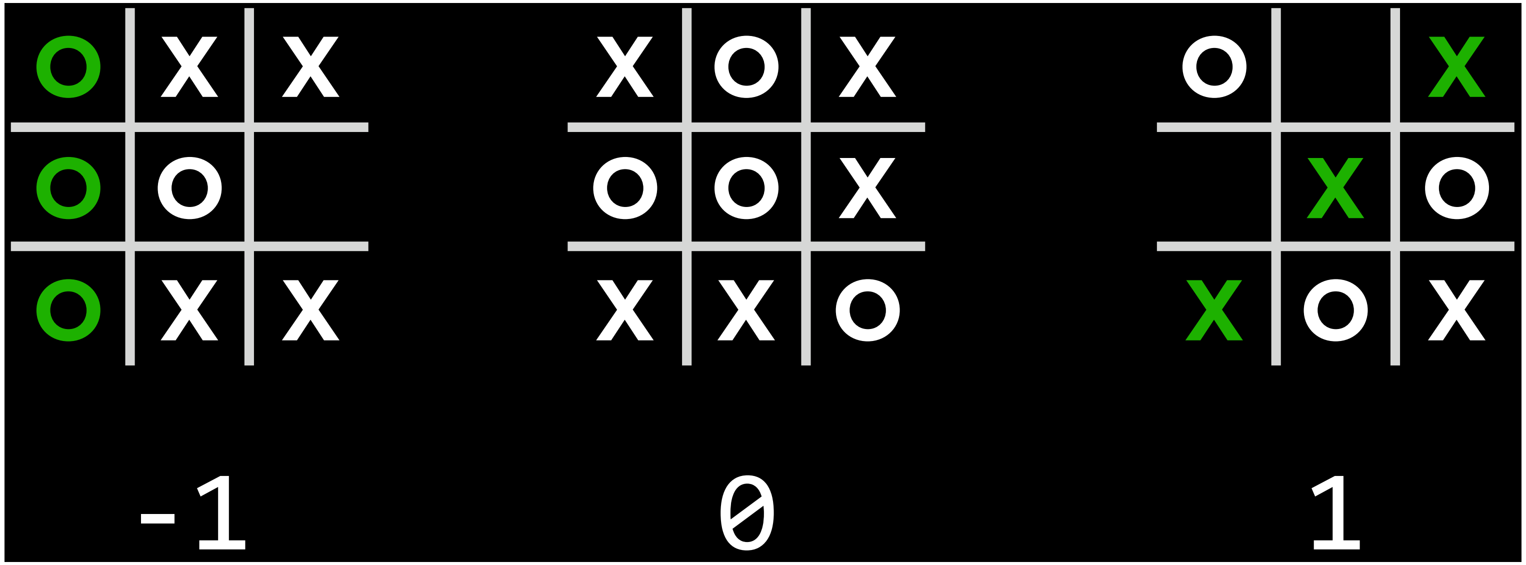
- Two players
  - O
  - X



Minimax

- 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
    - The minimizing side tries to get the lowest score
    - The maximizing side tries to get the highest score

Minimax for Tic-Tac-Toe

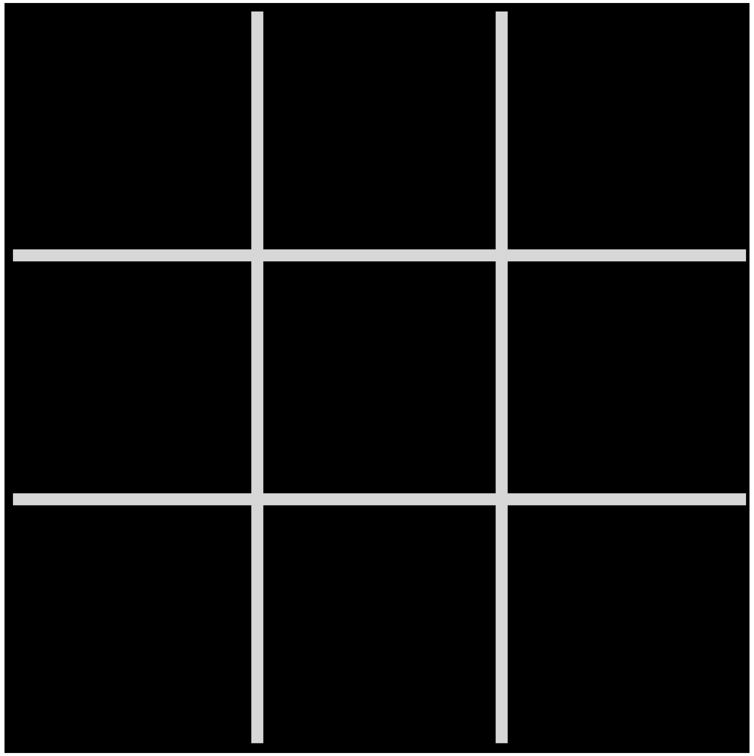


- Max(X) aims to maximize the score
- Min(O) aims to minimize the score

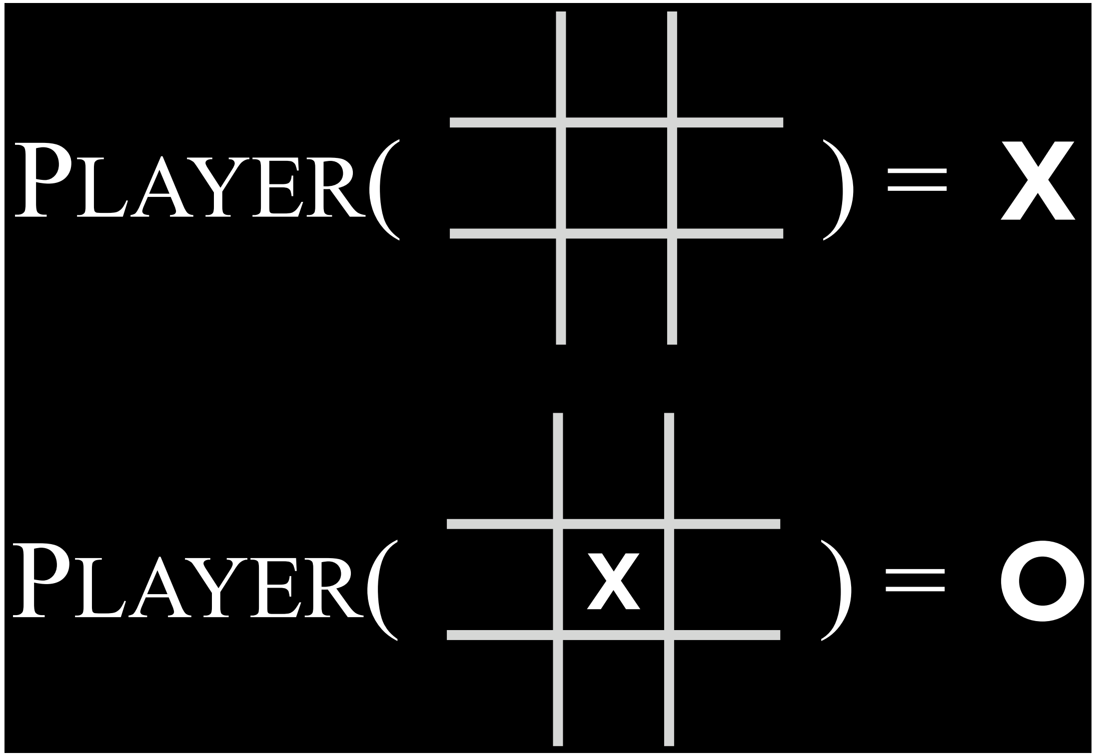
The Game

- S_0: initial state
- PLAYER(s): returns which player (X or O) to move in state s
- ACTIONS(s): returns legal moves in state s
  - What spots are free on the board
- RESULT(s,a): returns state after action a taken in state s
  - The board that resulted from performing the action a on the state s
- TERMINAL(s): checks if state s is a terminal state
  - If someone won or there is a tie
    - Returns True if the game has ended, False otherwise
- UTILITY(s): final numerical value for terminal state s
  - That is, -1, 0 or 1

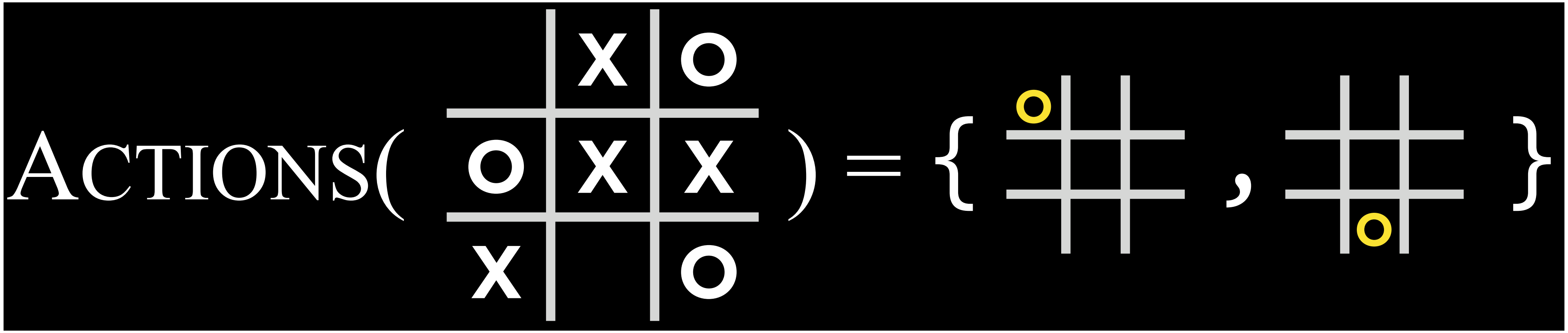
Initial State



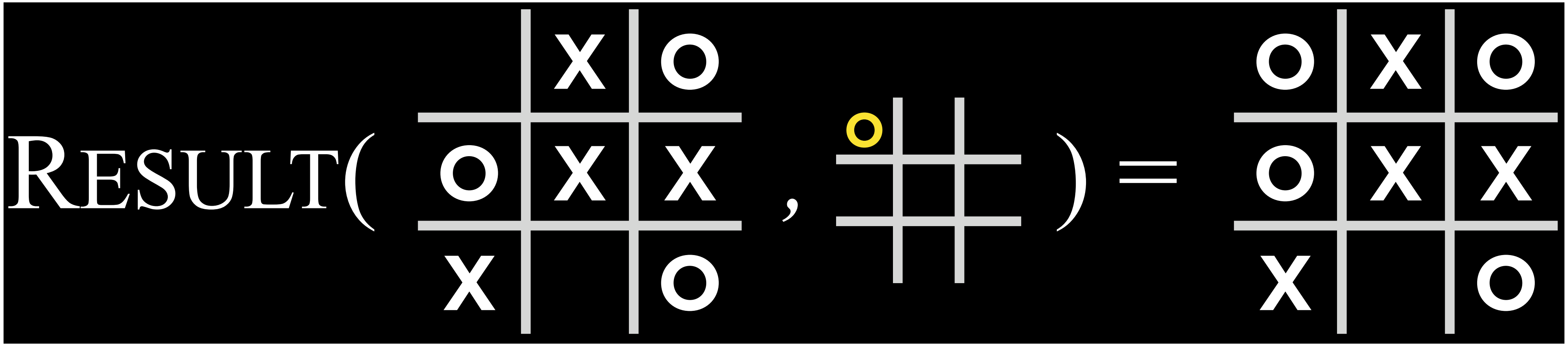
PLAYER(s)



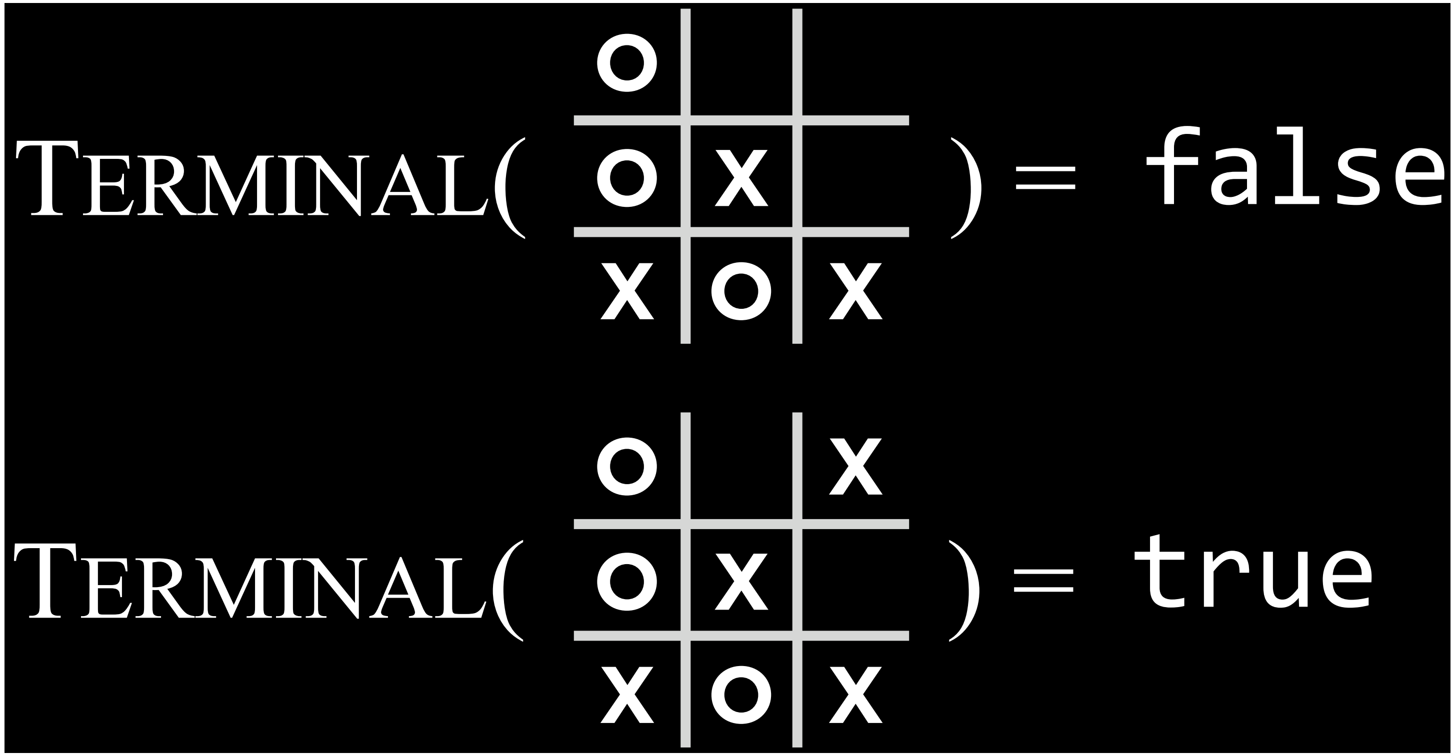
ACTION(s)



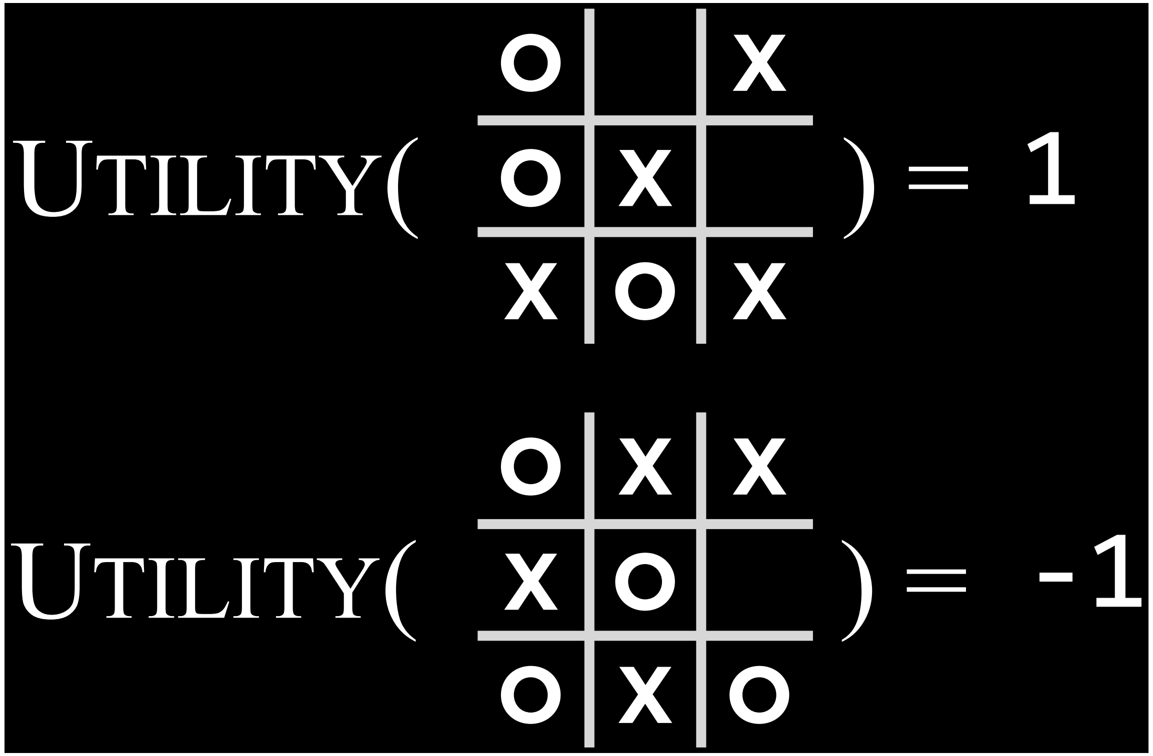
RESULTS(s,a)



TERMINAL(s)

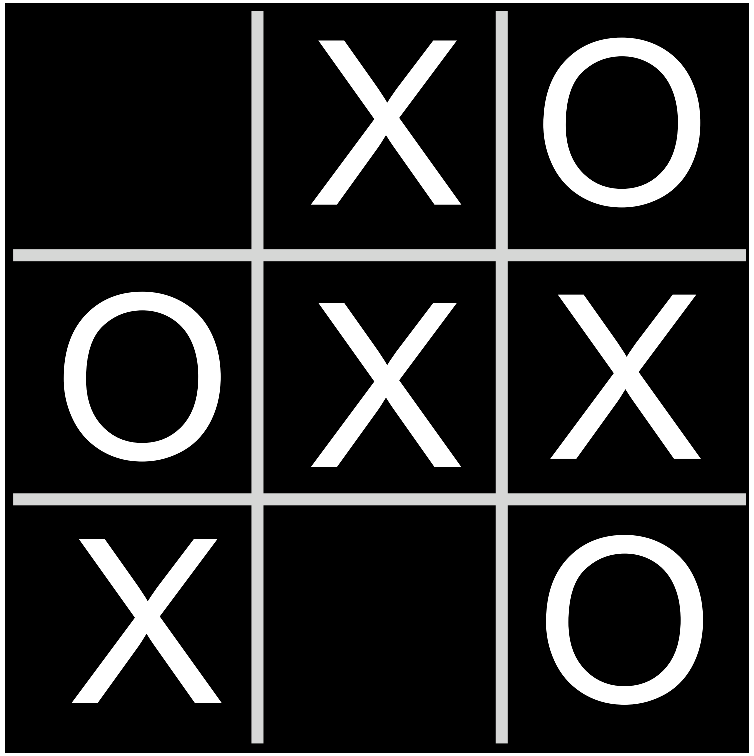


UTILITY(s)

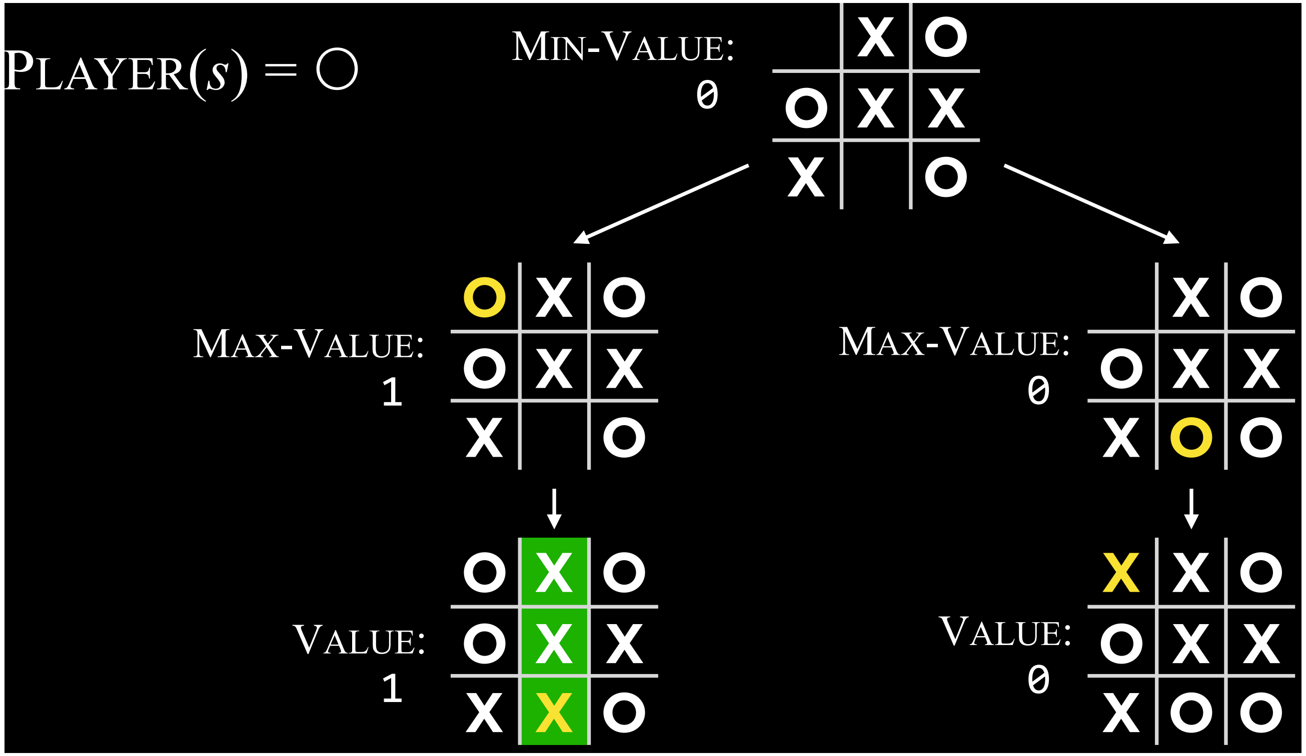


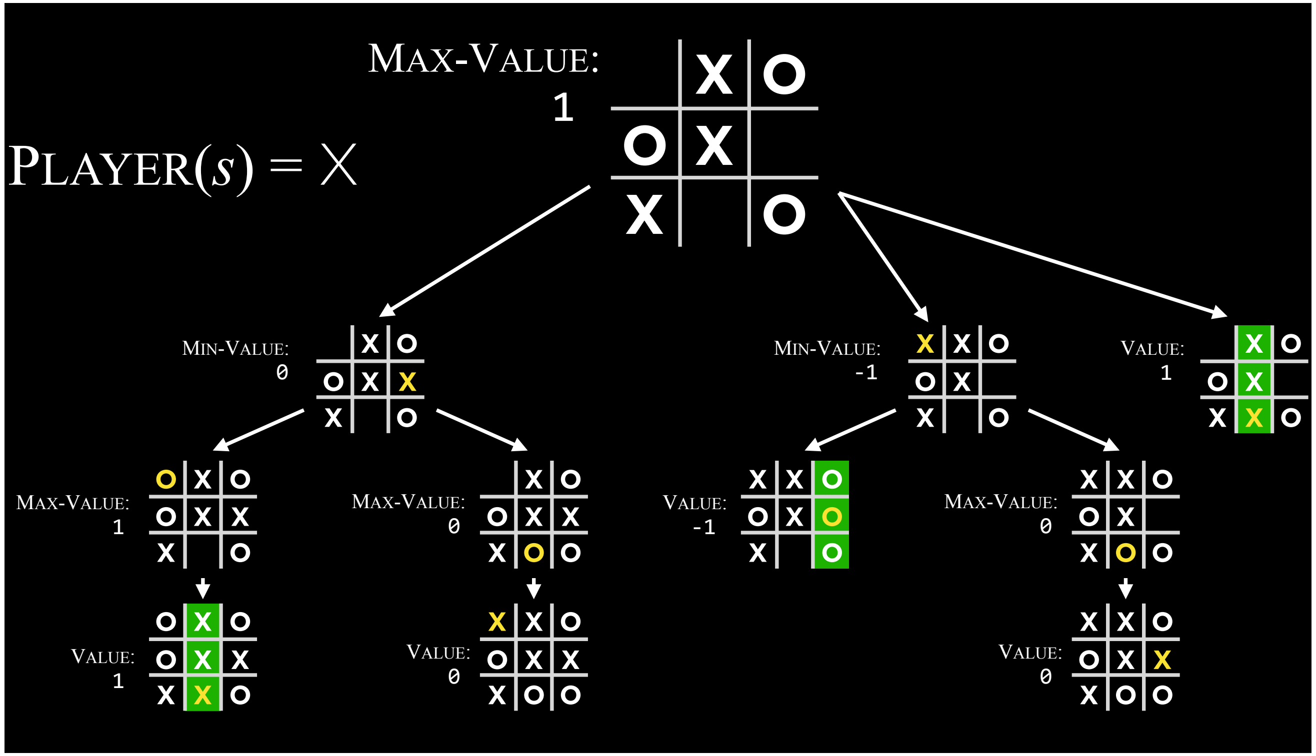
What Action should O take?

- Player(s) = O



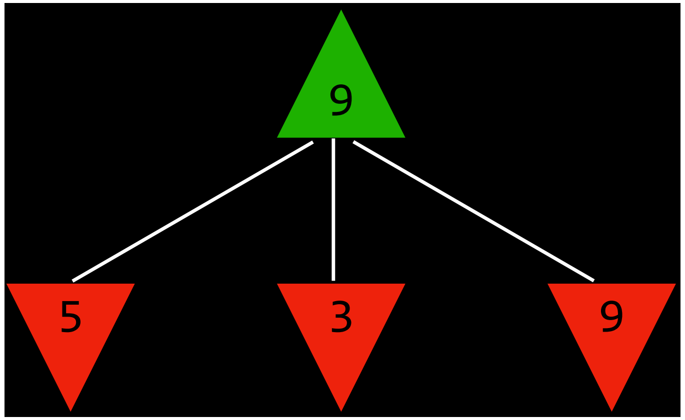
PLAYER(s) = O




MAX-VALUE:
1
PLAYER(s) = X
MIN-VALUE:
0
MIN-VALUE:
-1
VALUE:
1
MAX-VALUE:
1
MAX-VALUE:
0
VALUE:
-1
MAX-VALUE:
0
VALUE:
1
VALUE:
0
VALUE:
0

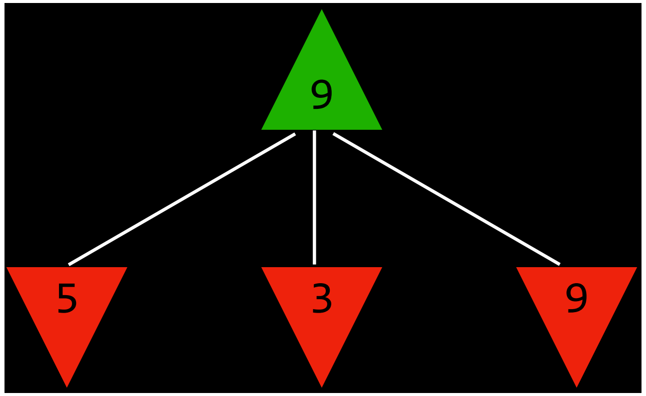
Generalizing the Game Tree

- We can simplify the diagram into a more abstract Minimax tree
  - each state is just representing some generic game that might be tic-tac-toe or some other game
  - Any of the green arrows that are pointing up, represents a maximizing state, where the player is the *max* player
    - the score should be as big as possible
  - Any of the red arrows pointing down are minimizing states, where the player is the *min* player
    - trying to make the score as small as possible



Generalizing the Game Tree

- Let's consider the maximizing player
  - He has three choices
    - one choice gives a score of 5
    - one choice gives a score of 3
    - one choice gives a score of 9
- Between those three choices, his best option is to choose 9
  - the score that maximizes his options out of all three options

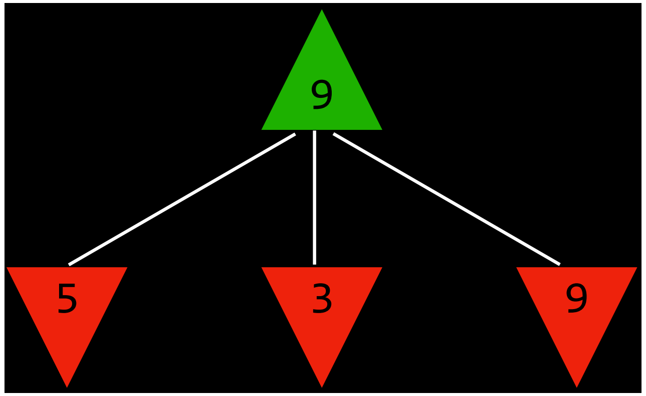


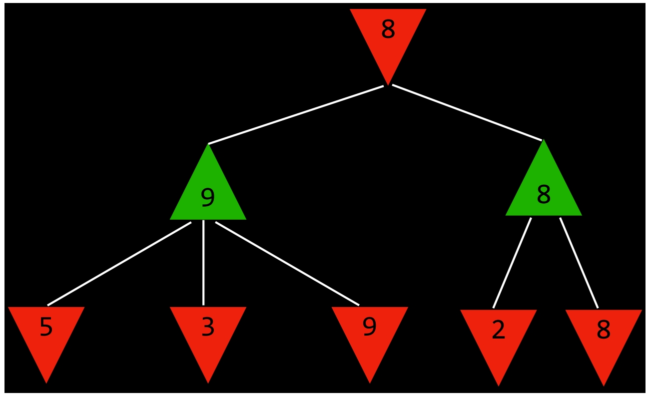
Observation:

For some games, a single move of a player is called a ply to distinguish it from a move where both players have taken an action

Generalizing the Game Tree

- Now, one could also ask a reasonable question
  - What might my opponent do two moves away from the end of the game?
  - The opponent is the minimizing player
    - He is trying to make the score as small as possible
    - Imagine what would have happened if they had to pick which choice to make





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 in Tic-Tac-Toe

- Knowing the state whose turn it is, the algorithm can know whether the current player, if playing optimally, will choose the action that leads to a state with a lower or higher value
- In this way, the algorithm alternates between minimizing and maximizing, generating values for the state that would result from each possible action
- This is a recursive process
  - Eventually, through this recursive reasoning process, the maximizing player generates values for each state that could result from all possible actions at the current state
  - After having these values, the maximizing player chooses the highest one
- The maximizer considers the possible values of future states

Minimax

- Given a state s:
  - MAX picks action a in ACTIONS(s) that produces highest value of MIN-VALUE(RESULT(s,a))
  - MIN picks action a in ACTIONS(s) that produces smallest value of MAX-VALUE(RESULT(s,a))
- Everyone makes their decision based on trying to estimate what the other person would do

Minimax

```
function MAX-VALUE(state):
    if TERMINAL(state):
        return UTILITY(state)
    v=-inf
    for action in ACTIONS(state):
        v=MAX(v,MIN-VALUE(RESULT(state,action)))
    return v
```

Minimax

```
function MIN-VALUE(state):
    if TERMINAL(state):
        return UTILITY(state)
    v=+inf
    for action in ACTIONS(state):
        v=MIN(v,MAX-VALUE(RESULT(state,action)))
    return v
```

Minimax properties

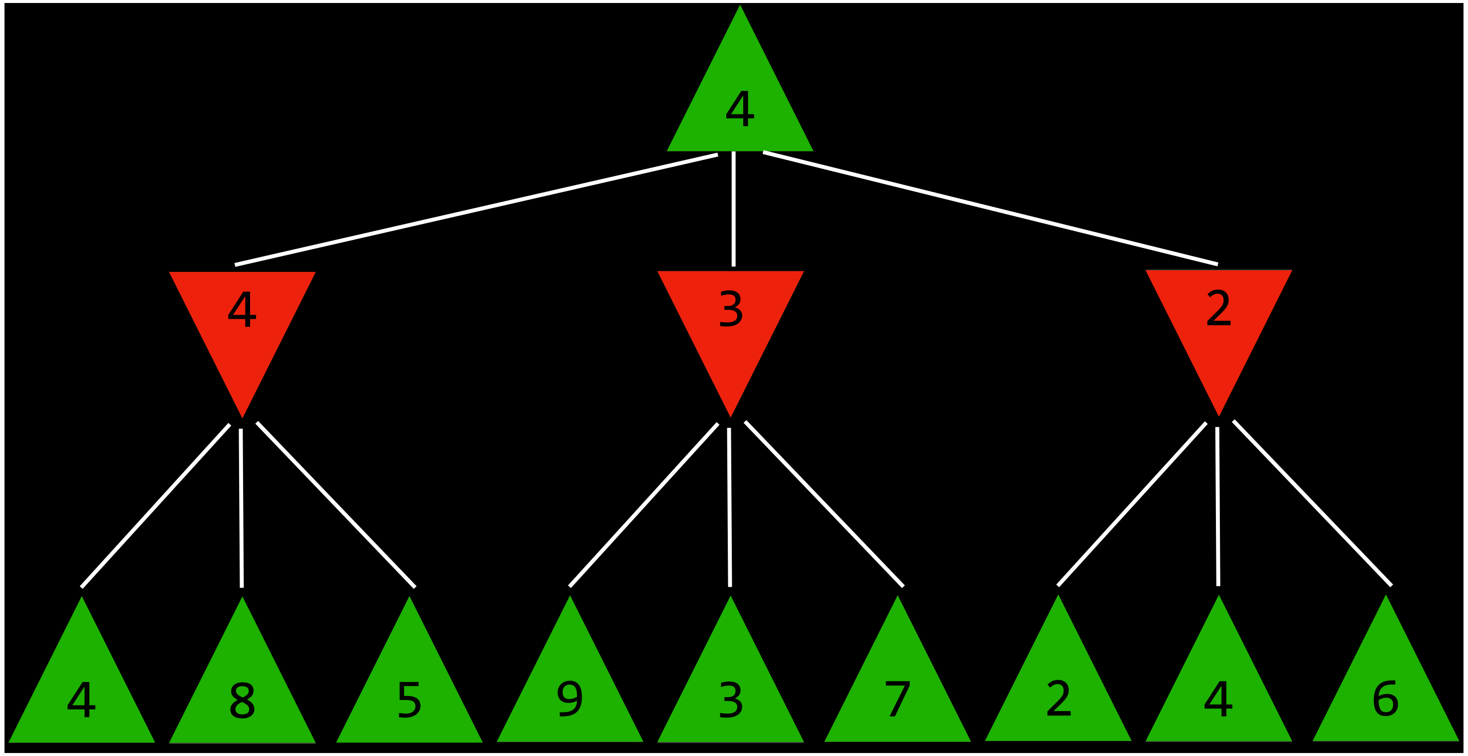
- Performs a complete depth-first exploration of the game tree
  - Time complexity -> $O(b^m)$
    - m maximum depth of the tree
    - b number of legal moves at each point
  - Space complexity -> O(bm)

Optimizations?

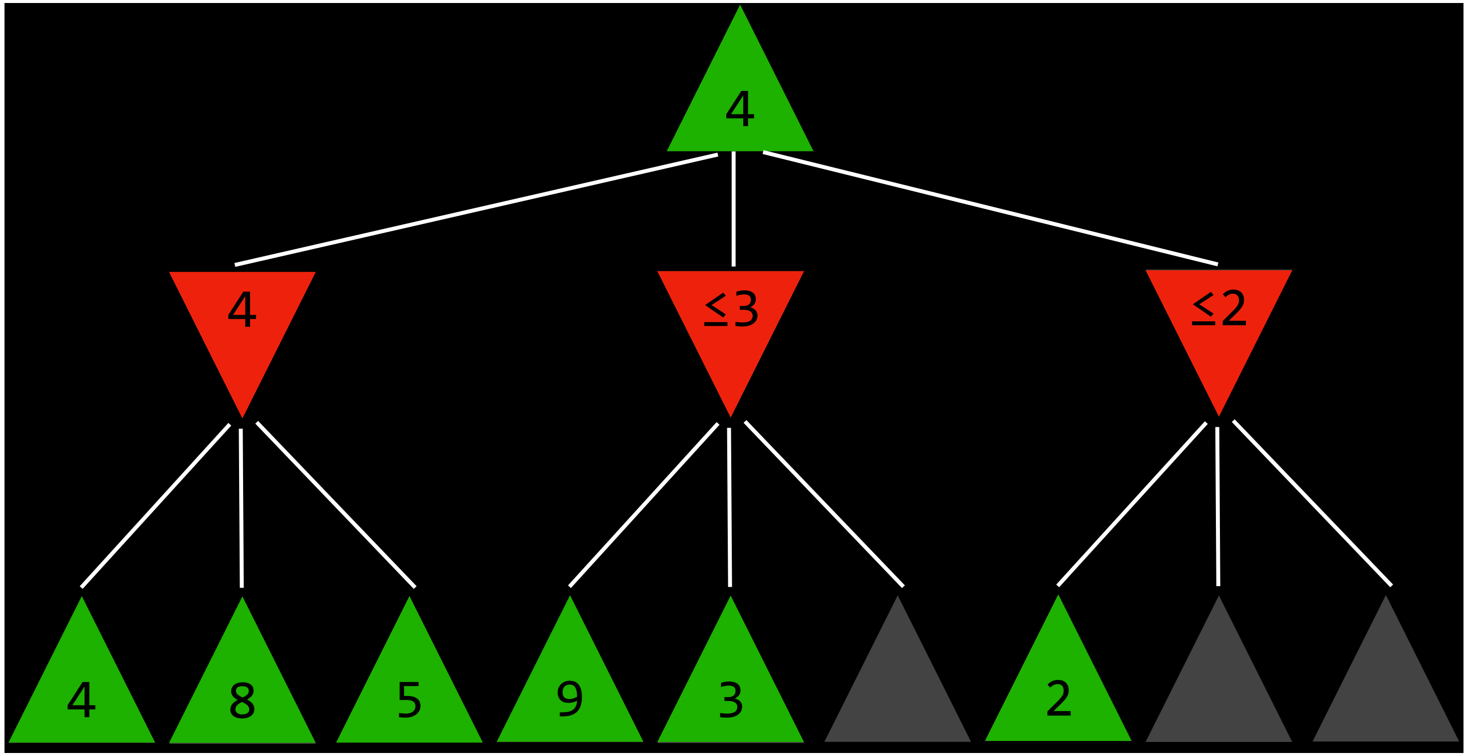
- The entire process could be long, especially as the game starts to get more complex, as we start to add more moves and more possible options
  - E.g., chess has a branching factor of about 35 and the average game has a depth of about 80 ply, not feasible to search 35^{80} states (about 10^{123})

- What sort of optimizations can we make here?
  - How can we do better to
    - use less space
    - take less time

What Minimax Does so far



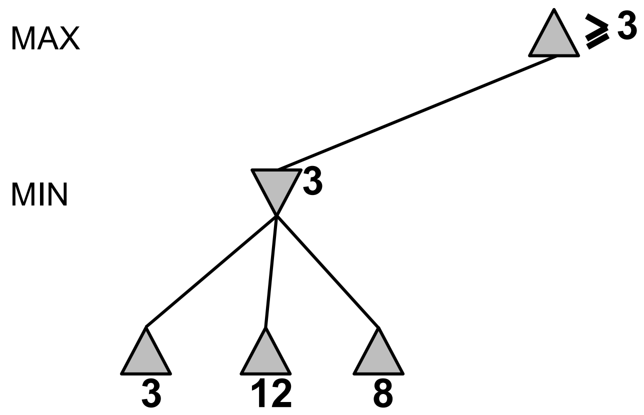
Pruning Useless Sub-Trees



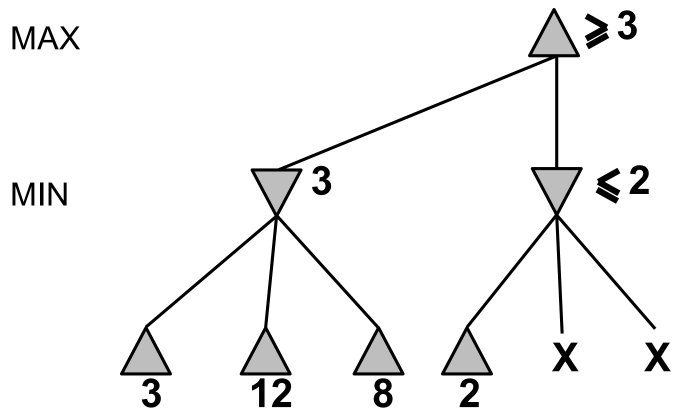
Alpha-Beta Pruning

- As a way to optimize Minimax, Alpha-Beta Pruning skips some of the recursive computations that are decidedly unfavorable
- If, after determining the value of an action, there are initial indications that the following action may cause the opponent to achieve a better result than the action already determined, there is no need to investigate this action further
  - because it will be decidedly less favorable than the previously determined action

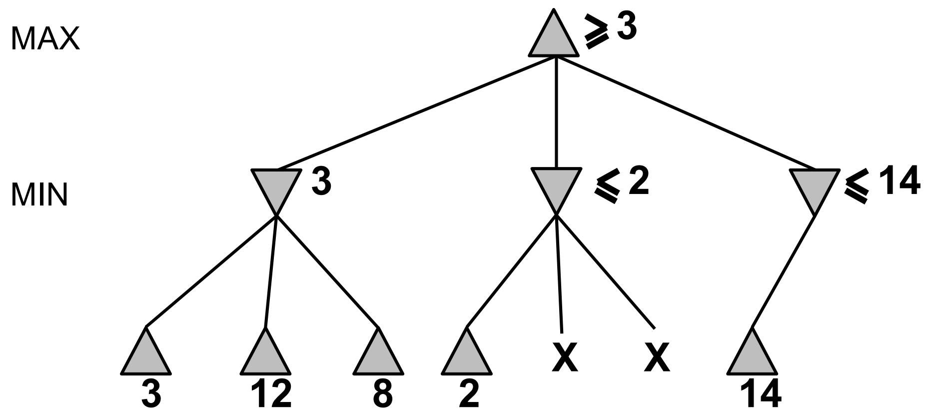
$\alpha - \beta$ Pruning Example



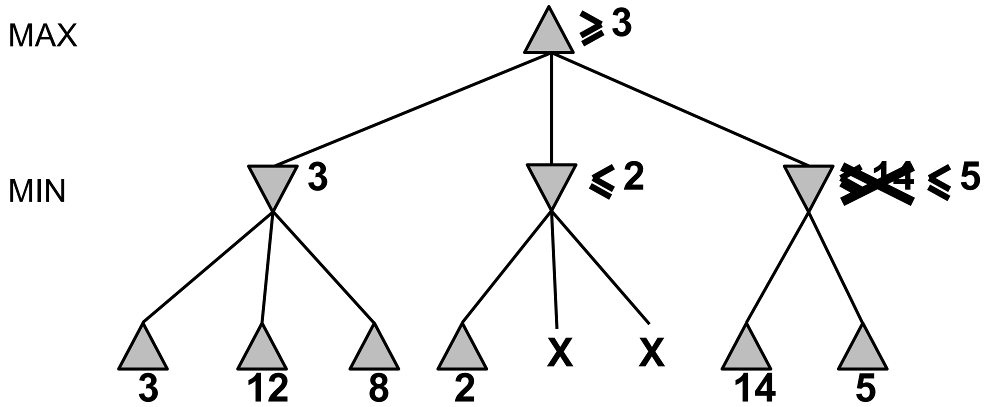
$\alpha - \beta$ Pruning Example



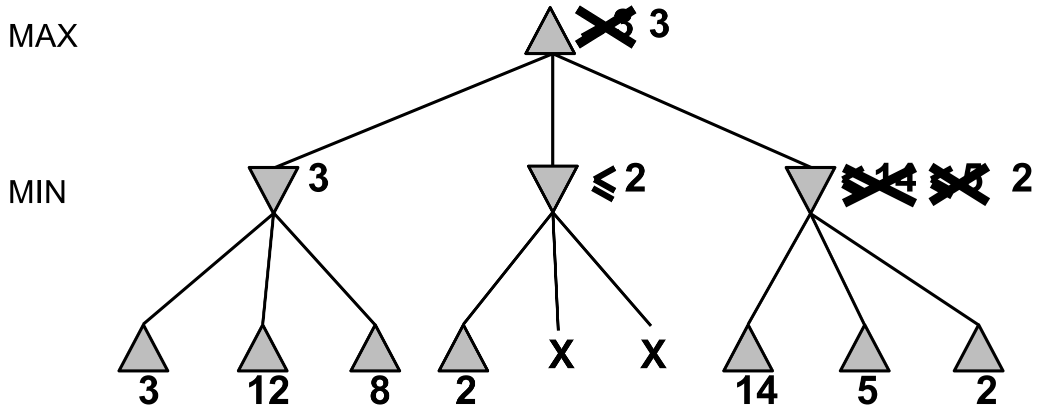
$\alpha - \beta$ Pruning Example



$\alpha - \beta$ Pruning Example

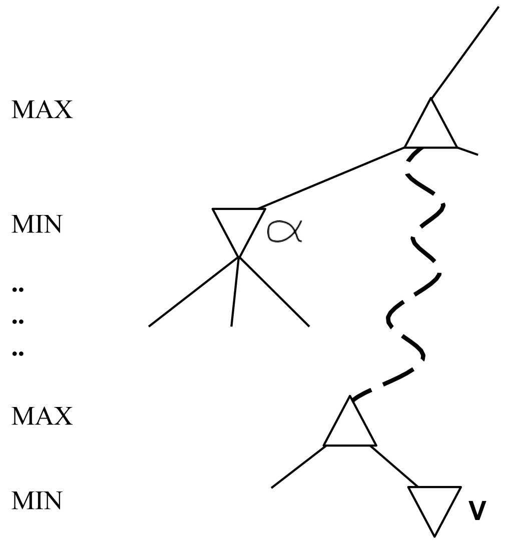


$\alpha - \beta$ Pruning Example



Minimax(root)	=	max(min(3,12,8), min(2, x, y), min(14,5,2))	
	=	max(3, min(2, x, y), 2)	
	=	max(3, z, 2)	where z = min(2, x, y) <= 2
	=	3	

Why is it Called $\alpha - \beta$?



- α is the best value (to max) found so far off the current path
- If V is worse than α, max will avoid it $\Rightarrow$ prune that branch
- Define β similarly for min

The $\alpha - \beta$ Algorithm

```
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 ← − ∞
   for a, s in Successors(state) do
      v ← Max(v, Min-Value(s, α, β))
      if v ≥ β then return v
      α ← Max(α, v)
   return v

function Min-Value(state, α, β) returns a utility value
   same as Max-Value but with roles of α, β reversed
```

Properties of $\alpha - \beta$

- Pruning does not affect the final result
- Good move ordering improves the effectiveness of pruning
- With a *perfect ordering*, time complexity = $O(b^{m/2})$
  - This means it doubles the solvable depth
  - For chess (about 35^{100}), unfortunately, 35^{50} is still impossible!
- A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Total Possible Games

- 255.168 total possible Tic-Tac-Toe games
- More complex game
  - 288.000.000.000 total possible chess games
    - after four moves each
  - 10^{29000} total possible chess games (lower bound)
- A big problem for Minimax
- So what?
  - Do not look through all the states (also called type A strategy, Shannon 1950)
    - Depth-limited Minimax

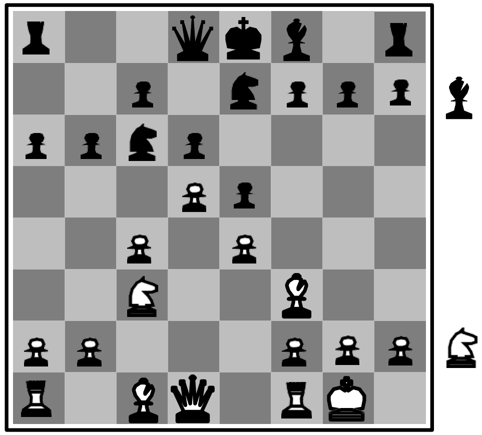
Depth-Limited Minimax

- Depth-limited Minimax only considers a predefined number of moves before stopping, without ever reaching a terminal state
  - However, this does not allow to obtain an exact value for each action, since the end of the hypothetical games has not yet 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 estimated values to states

Evaluation function

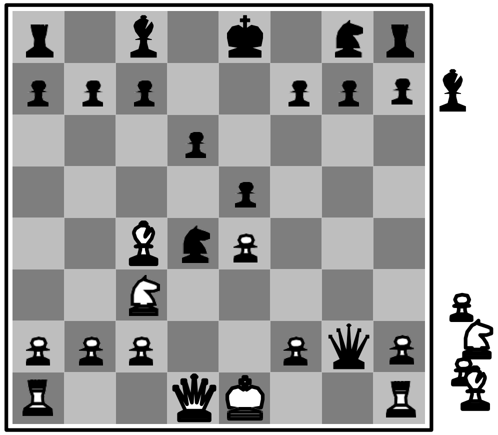
- Evaluation function
  - Function that estimates the expected utility of the game from a given state
- Example
  - In a game like chess, if you imagine that a game value of 1 means white wins, -1 means black wins, 0 means it's a draw
    - A score of 0.8 means white is very likely to win though certainly not guaranteed
    - Depending on how good that evaluation function is, ultimately constrains how good the AI is

Evaluation Functions



Black to move

White slightly better



White to move

Black winning

- For chess, typically linear weighted sum of features
  - $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
- For instance, $w_1 = 3$ with
  - $f_1(s)$ = (number of white pawns) – (number of black pawns), etc.