

---

# Lecture 9

## Games and Adversarial Search

**CS 180 – Intelligent Systems**

Dr. Victor Chen

Spring 2021

# Games and adversarial search

---



# Fully observable vs. partially observable

---

Fully observable: The values of all the state variables are known to the agent

# Deterministic vs. stochastic

---

Deterministic: The next state is completely determined by the current state and the action being taken.

Otherwise, stochastic (outcomes are controlled by chance)

# Types of game environments

**Fully observable:** The values of all the state variables are known to the agent

**Deterministic:** The next state is completely determined by the current state and the action being taken.

	Deterministic	Stochastic
Perfect information (fully observable)	Chess, checkers, go	Monopoly
Imperfect information (partially observable)	Crossword puzzle	Scrabble, poker, bridge

## CROSSWORD PUZZLE

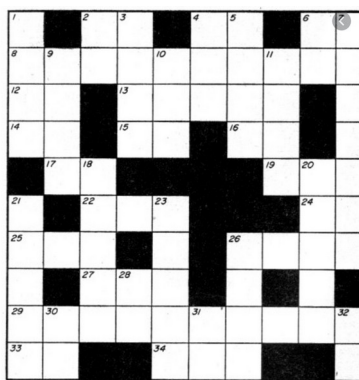
By Arthur L. Branch

**ACROSS**

- 1 Type of voice broadcasting: Abbr.
- 2 Exclamation.
- 3 Special type of a.c. generator.
- 4 Chemical symbol for lithium.
- 5 Electromagnetic wave used for communication.
- 6 In: Span.
- 7 Egypt: Abbr.
- 8 Registered nurse: Abbr.
- 9 Suffix denoting one who does.
- 10 Devour.
- 11 Sound detecting device.
- 12 Southern state: Abbr.
- 13 Liquid insulator.
- 14 Part of a transformer.
- 15 Assistance.
- 16 Tubes that operate at firing voltages.
- 17 Compass point: Abbr.
- 18 Five-and-a-half yards.

**DOWN**

- 19 Emitter of magnetic lines of force.
- 20 Input current to rectifier.
- 21 Deep mud.
- 22 Ancient.
- 23 Parts of the head.
- 24 To commit.
- 25 Piezoelectric material.
- 26 Wave propagated by one cycle of a.c. voltage.
- 27 Relation of current to voltage in an inductive circuit.
- 28 Quality of sound.
- 29 Electromagnetic switch.



Crossword puzzle



Monopoly



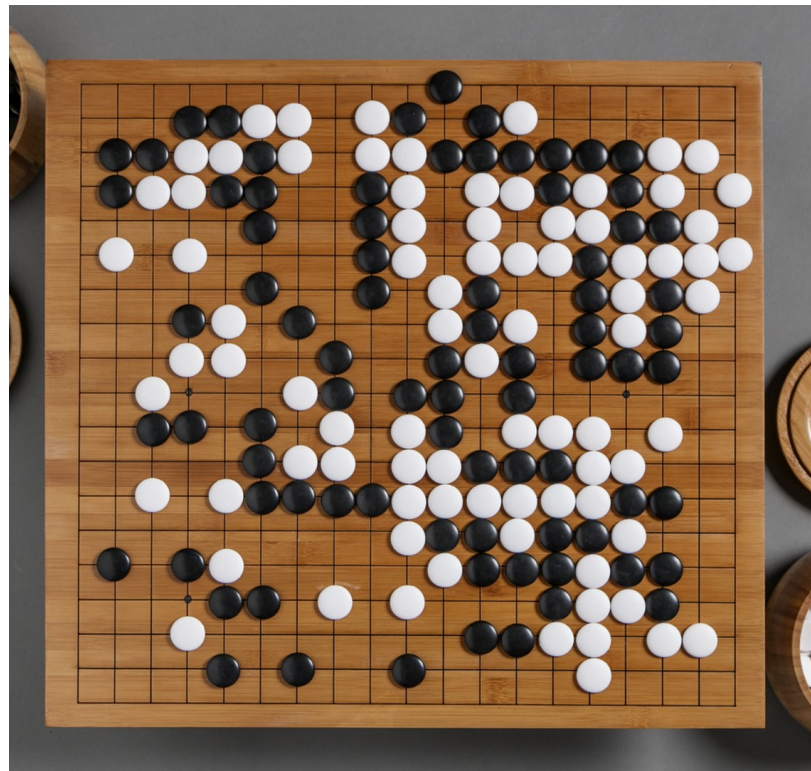
Bridge

# Our focus

---

In CSC180, we mainly focus on deterministic, fully observable, zero-sum games.

- Go, chess, checkers





# Zero-sum games

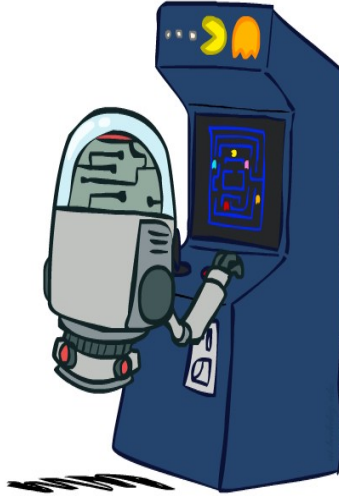
---

- The **sum** of the **amounts** some players win equals **the losses of the others**.
- Go, chess, checkers, poker and gambling are **zero-sum games**



# Build a game-playing AI agent...

---



Our goal: to look for such a *policy* that given a state, tells us what is best action/move in that state to maximize reward?



# Take Tic-tac-toe as example

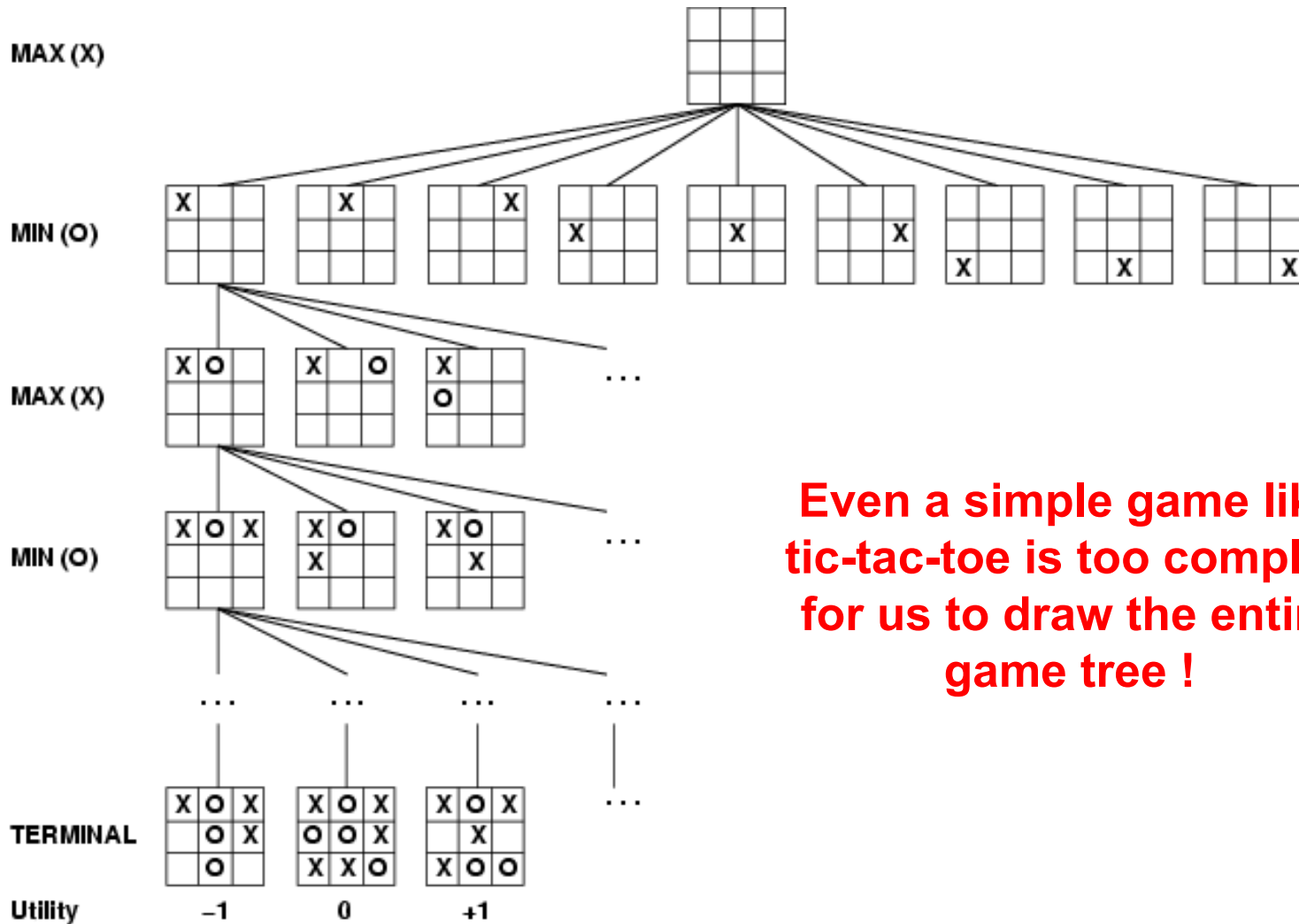
---

Tic-tac-toe: Two players take turns marking the spaces in a 3×3 grid. The player who places three same marks in a horizontal, vertical, or diagonal row wins the game



# Game tree of tic-tac-toe

# A game of tic-tac-toe between two players, “max” and “min”



**Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree !**

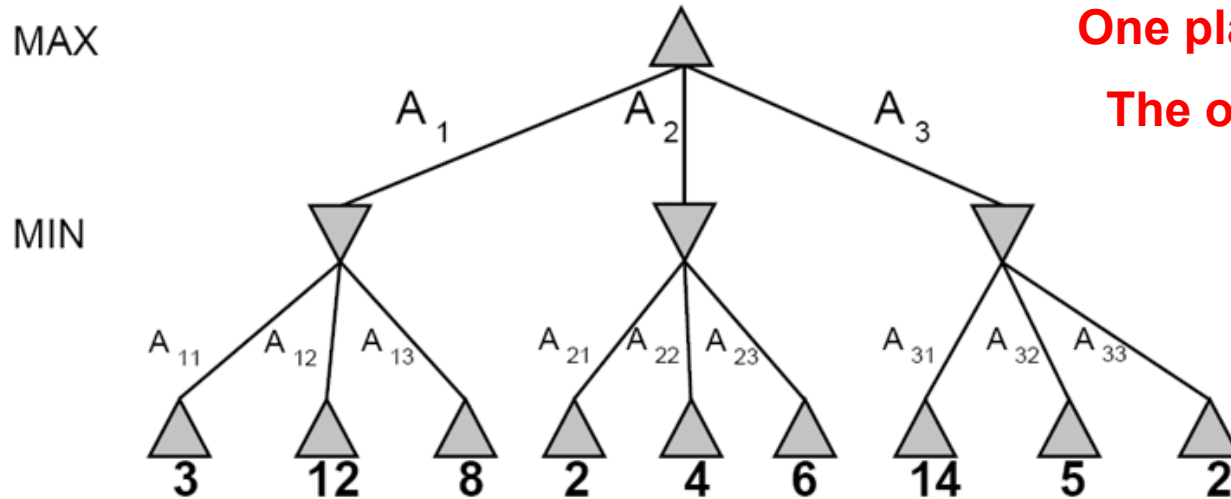
# A abstract game tree for a trivial game

---

**This is a zero-sum game**

**One player (MAX) maximizes result**

**The other (MIN) minimizes result**



**A game where two players (MAX and MIN) take turns.**



are MAX nodes, in which it is MAX's turn to move



are MIN nodes, in which it is MIN's turn to move

**Numbers on the leaf nodes (terminal states) are the reward MAX wins!**

# Smart search to win? Minimax search

---

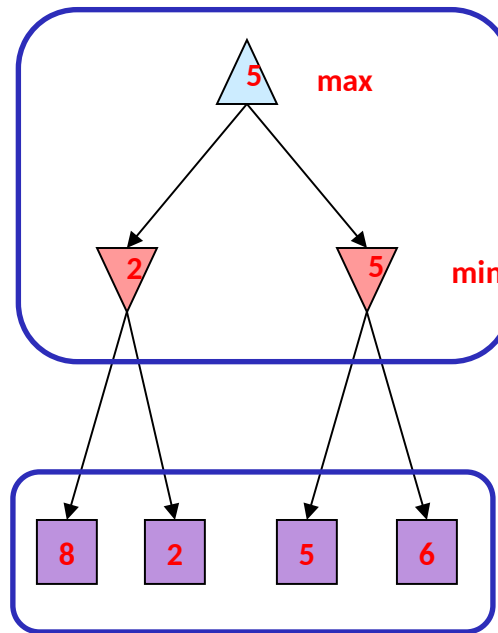
## Minimax search:

- Compute each node's minimax value:
  - Minimax value: The highest reward value MAX can achieve if MAX plays against an optimal adversary
- Best move for MAX in a state is its successor state with the highest minimax value

# Adversarial Search (Minimax search)

---

**Minimax values:  
computed recursively**



**Reward (terminal) values**

# Derive Minimax values

---

**def minimax\_value(state):**

if the state is a terminal state: return the state's reward

if *player* = **MAX**: return **max-value(state)**

if *player* = **MIN**: return **min-value(state)**

**def max-value(state):**

initialize  $v = -\infty$

for each successor of state:

$v = \max(v, \text{value}(\text{successor}))$

return  $v$

**def min-value(state):**

initialize  $v = +\infty$

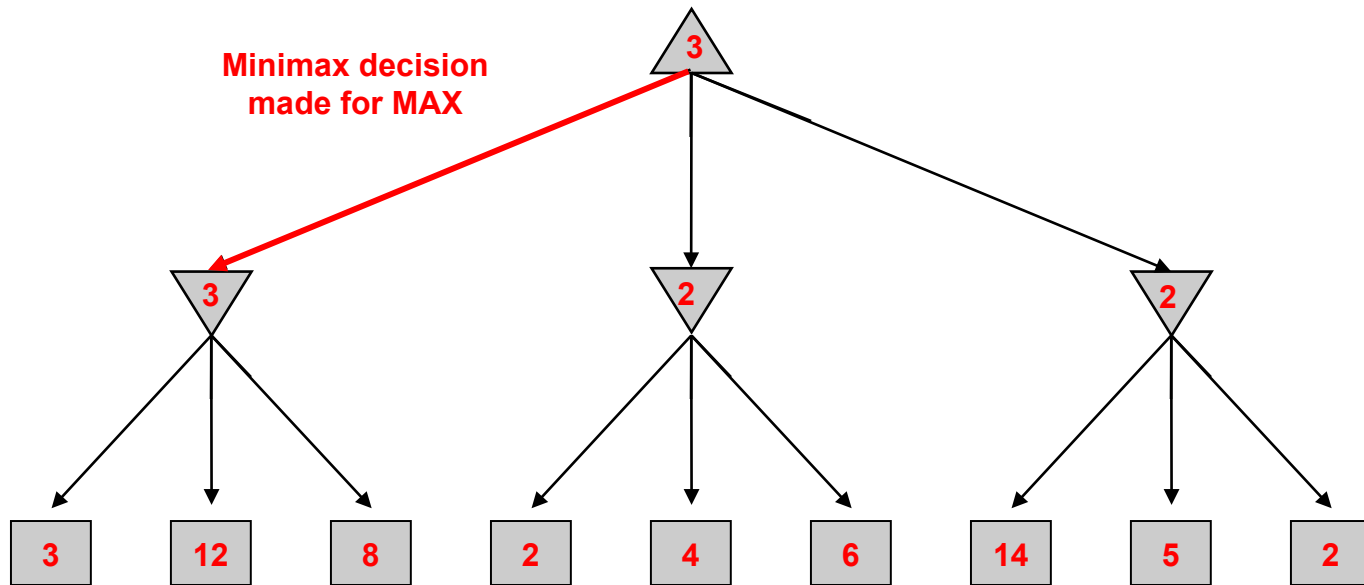
for each successor of state:

$v = \min(v, \text{value}(\text{successor}))$

return  $v$

# Minimax Example

---

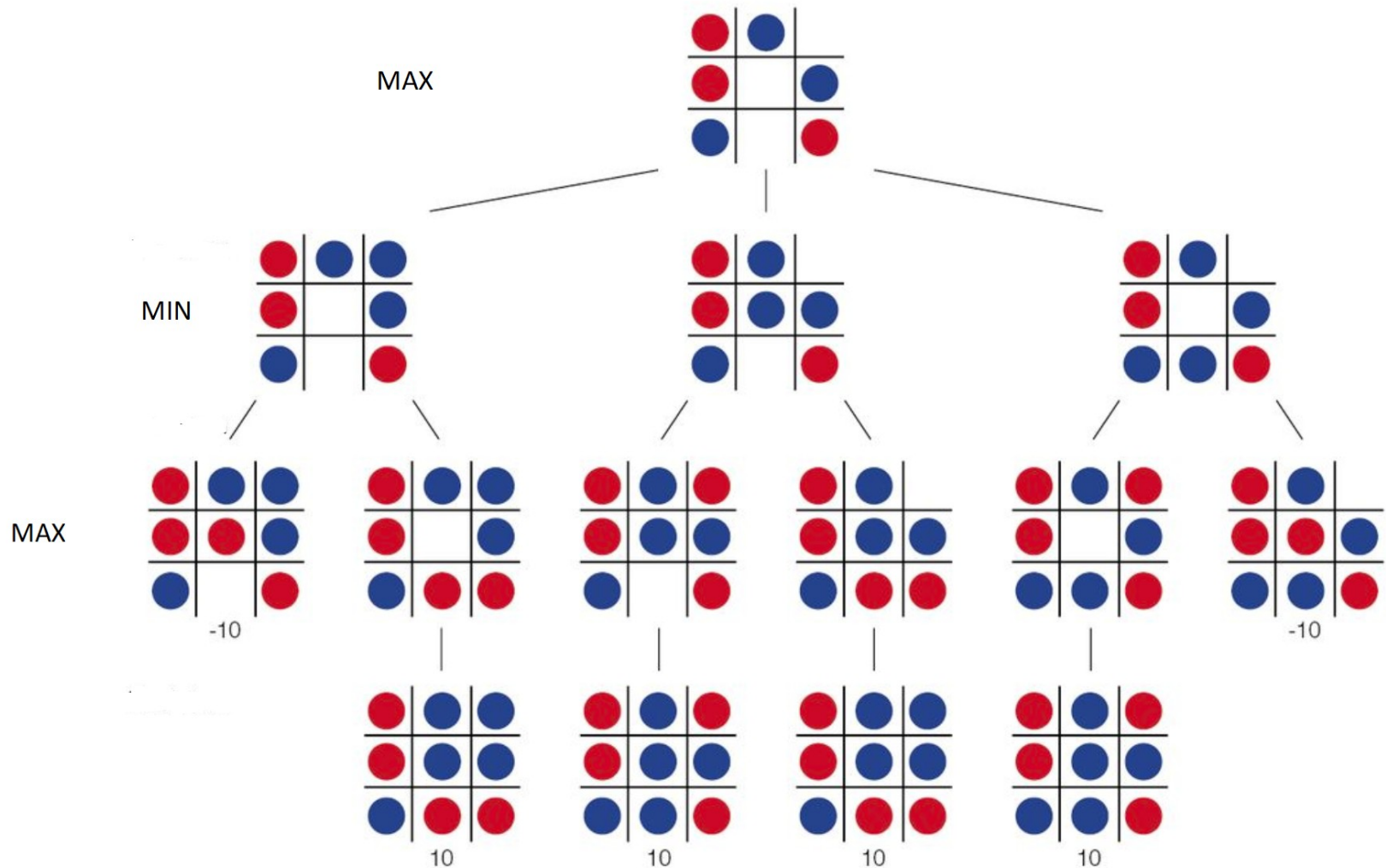


Each layer of the search tree is called a *ply*.



# You are MAX in blue (tic-tac-toe)...

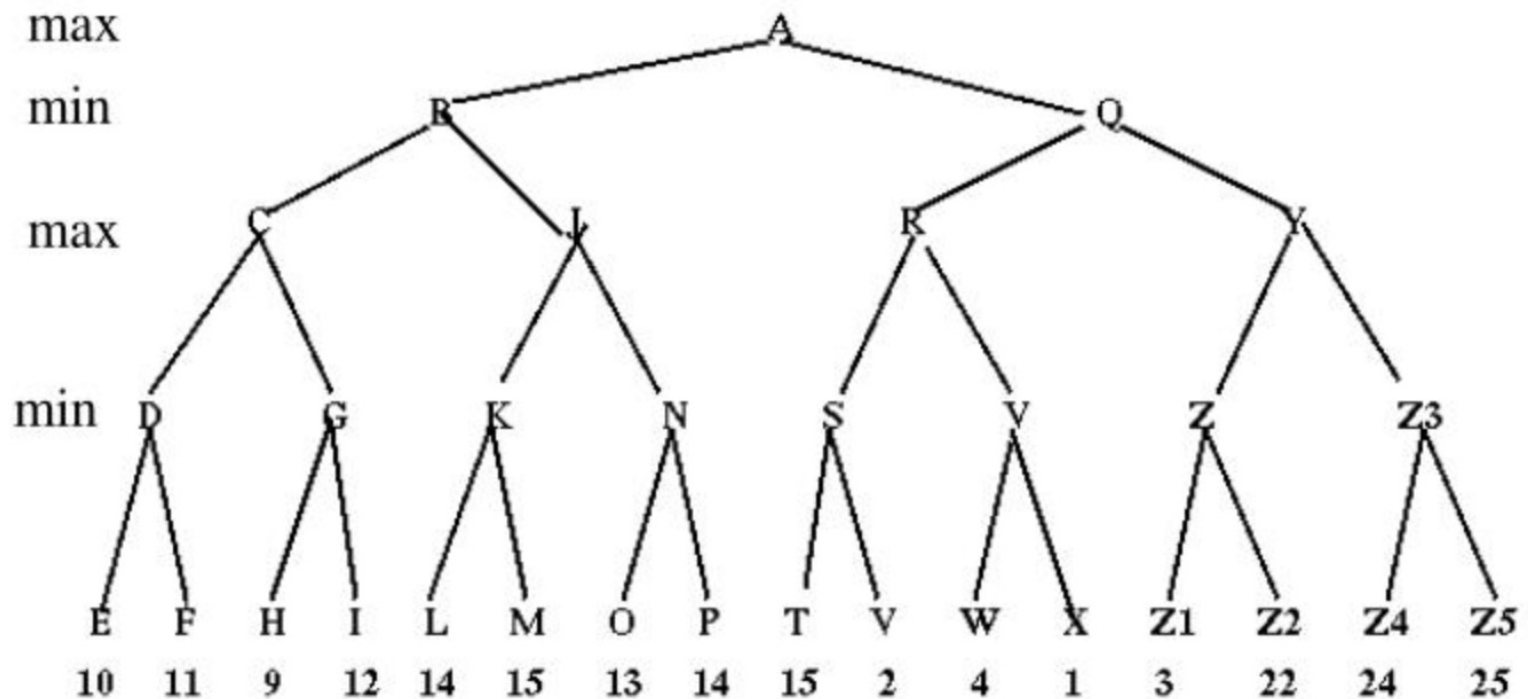
---



# Practice

---

You play as MAX and you are at A now. Which state you should move to next?



# Optimality of minimax

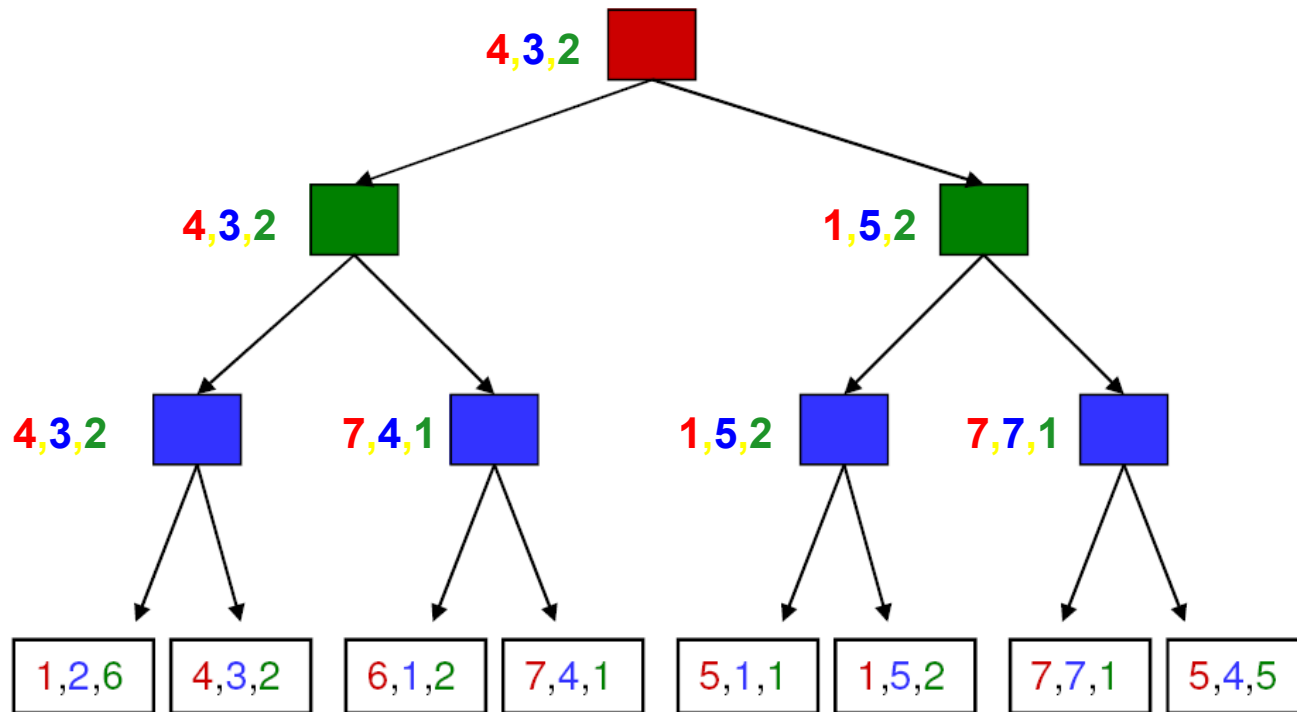
---

The minimax search is the optimal policy against an optimal adversary

If your adversary is suboptimal, your reward (utility) value can only be higher than if you were playing against an optimal adversary!

# More than two players, non-zero-sum?

---



Minimax values become **vector rather than scalar value**

Each player maximizes her own reward at her node (assuming no alliances)

# Minimax efficiency: Naïve DFS does not work

---

How efficient is minimax search?

- Perform a **complete DFS** of the game tree. If tree depth is  $m$  and branching factor is  $b$ 
  - Time:  $O(b^m)$
  - Space:  $O(bm)$

Example: For chess,  $b \approx 35$ ,  $m \approx 100$



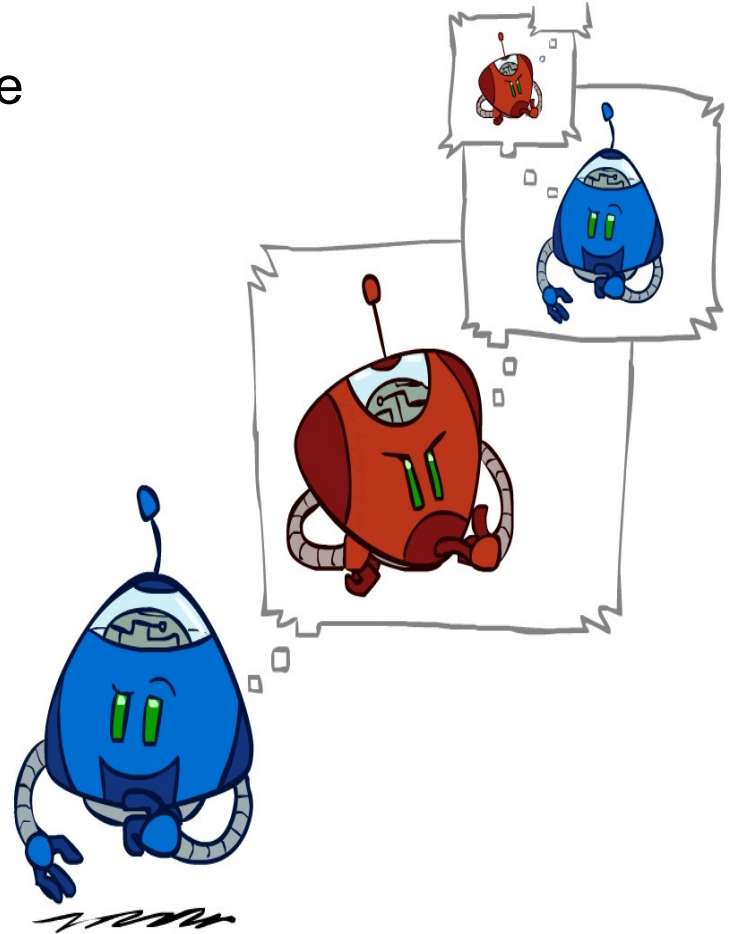
	Board Size	Branching Factor ( $b$ )
Go	$19 \times 19 = 361$	$\approx 250$
Chess	$8 \times 8 = 64$	$\approx 35$

# Minimax Efficiency

---

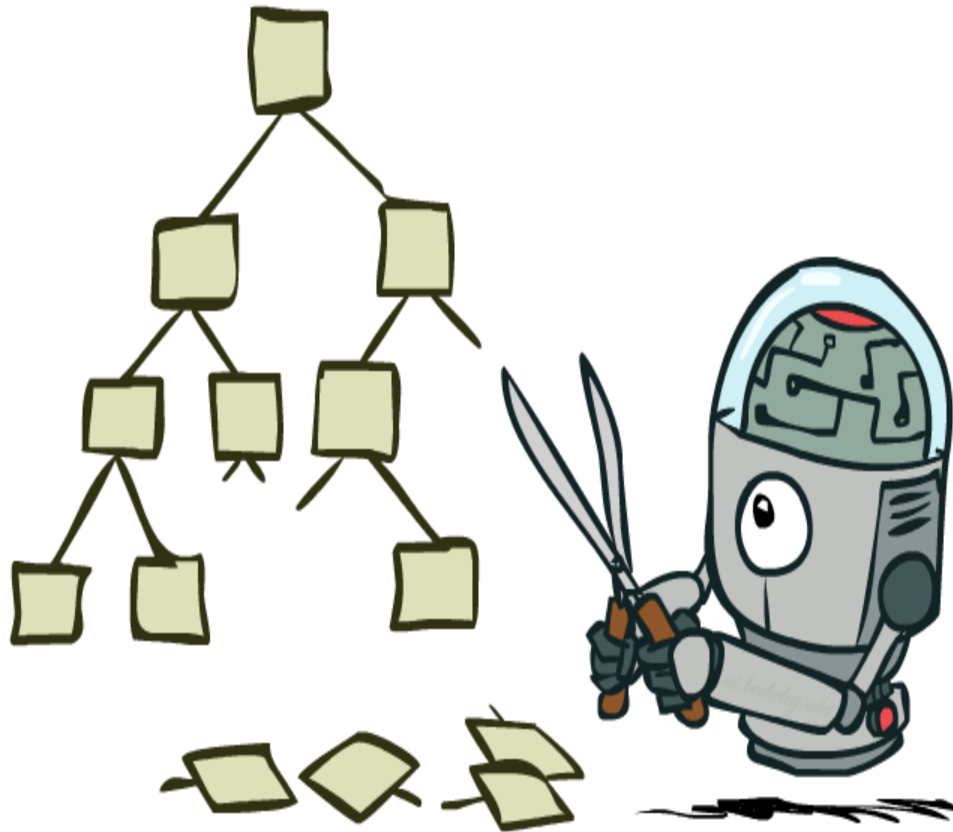
How to improve it?

- Do we need to explore/search the whole tree?
- No. We use tree pruning



# Game Tree Pruning

---

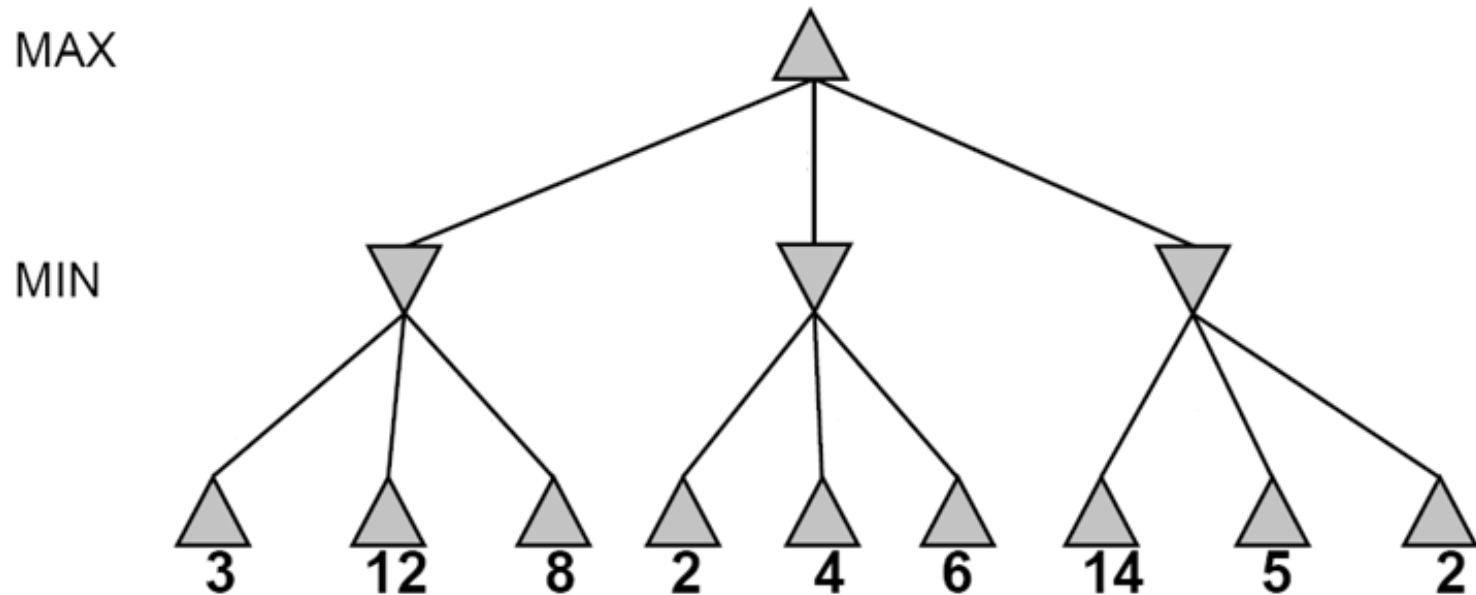




# Alpha-beta pruning

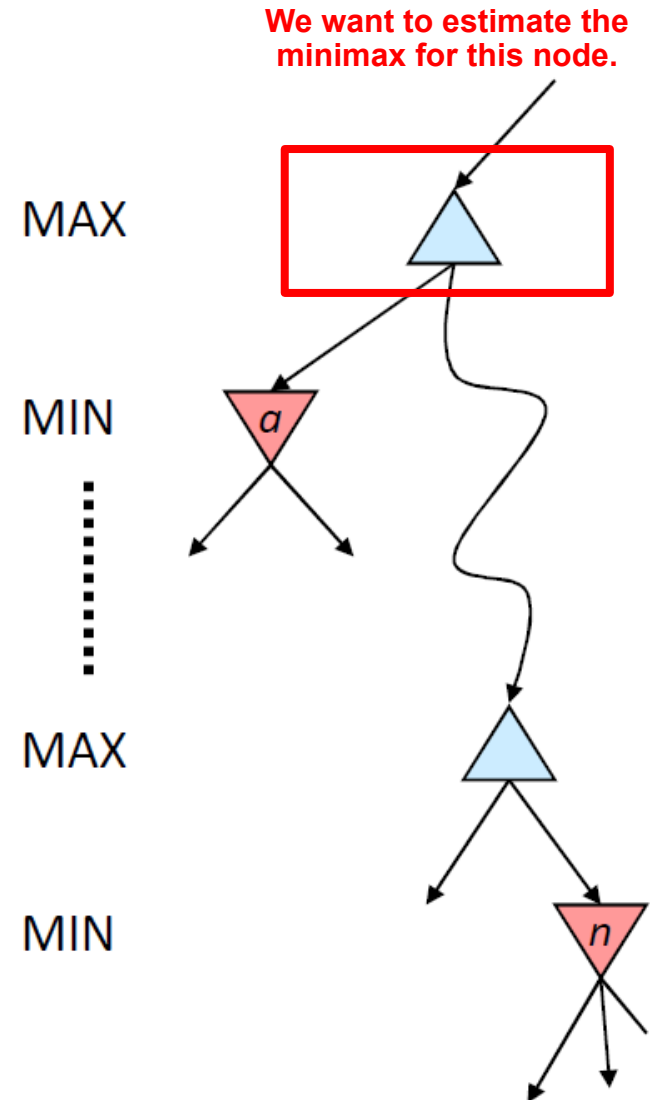
---

Compute the **minimax value of a node** without expanding every nodes under each child of the current node



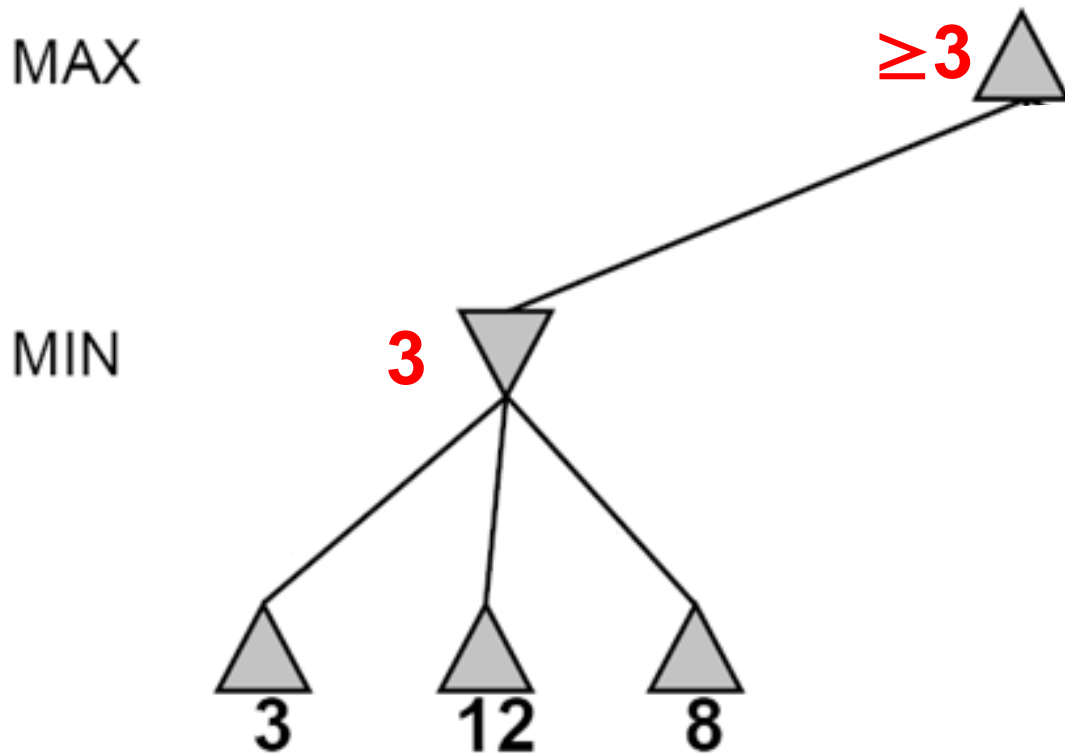
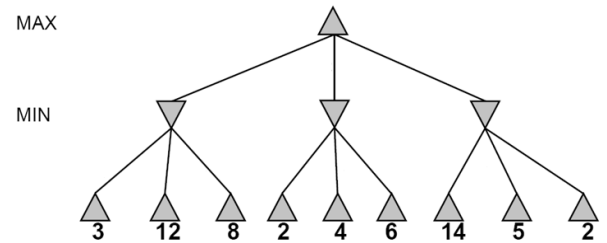
# Alpha-beta pruning for MAX player

- Suppose we are working at node **n**
- Suppose  **$\alpha$**  is the current best-choice value for MAX
- As we loop over **n**'s children, the value at node **n** decreases
- If it drops below  **$\alpha$** , MAX will never choose **n**, so we can ignore n's remaining children
  - **$\alpha$**  is a better choice than **n**
- Analogously,  **$\beta$**  is the current best-choice value for MIN player

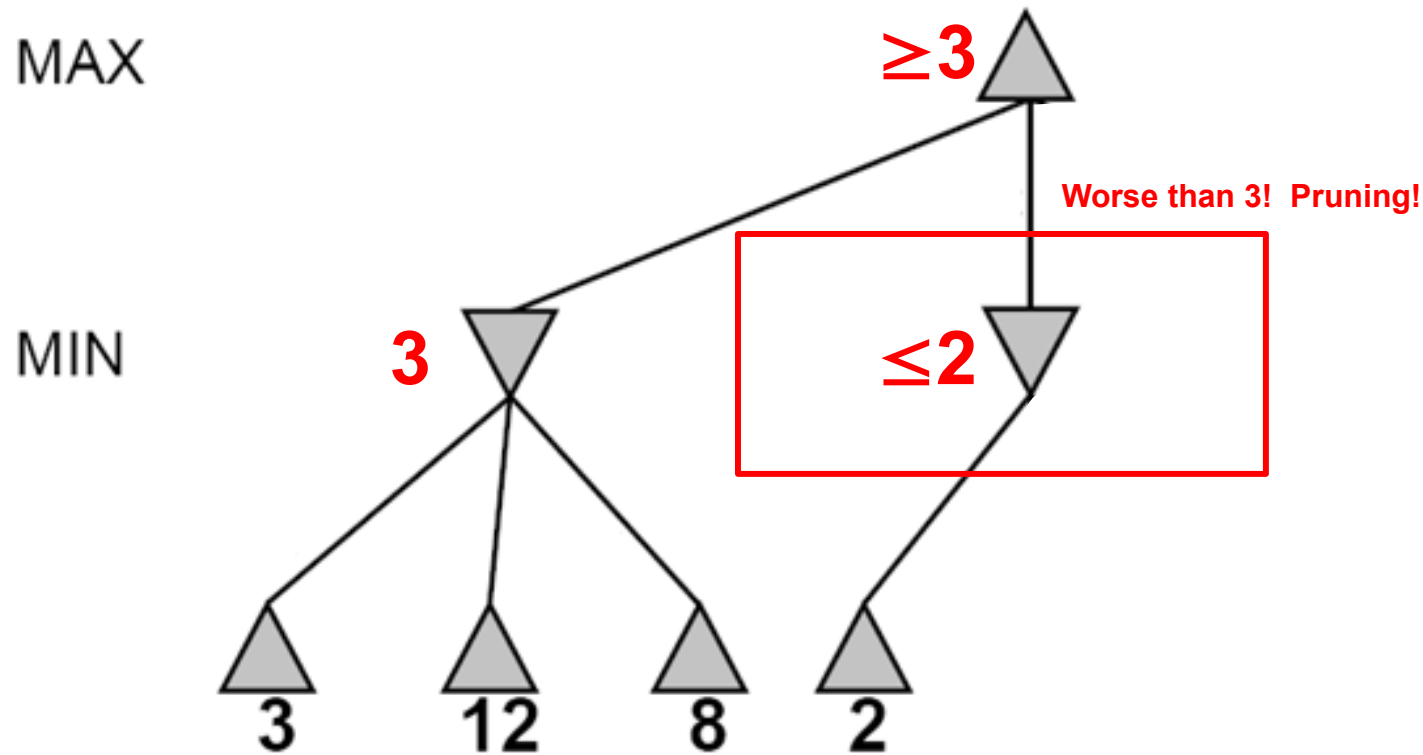
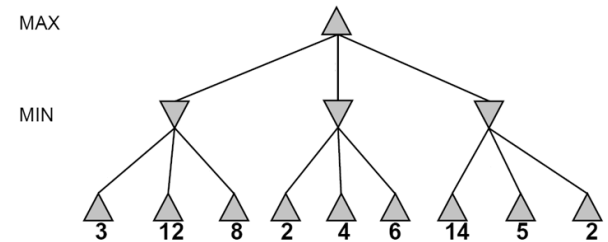


# Alpha-beta pruning

---

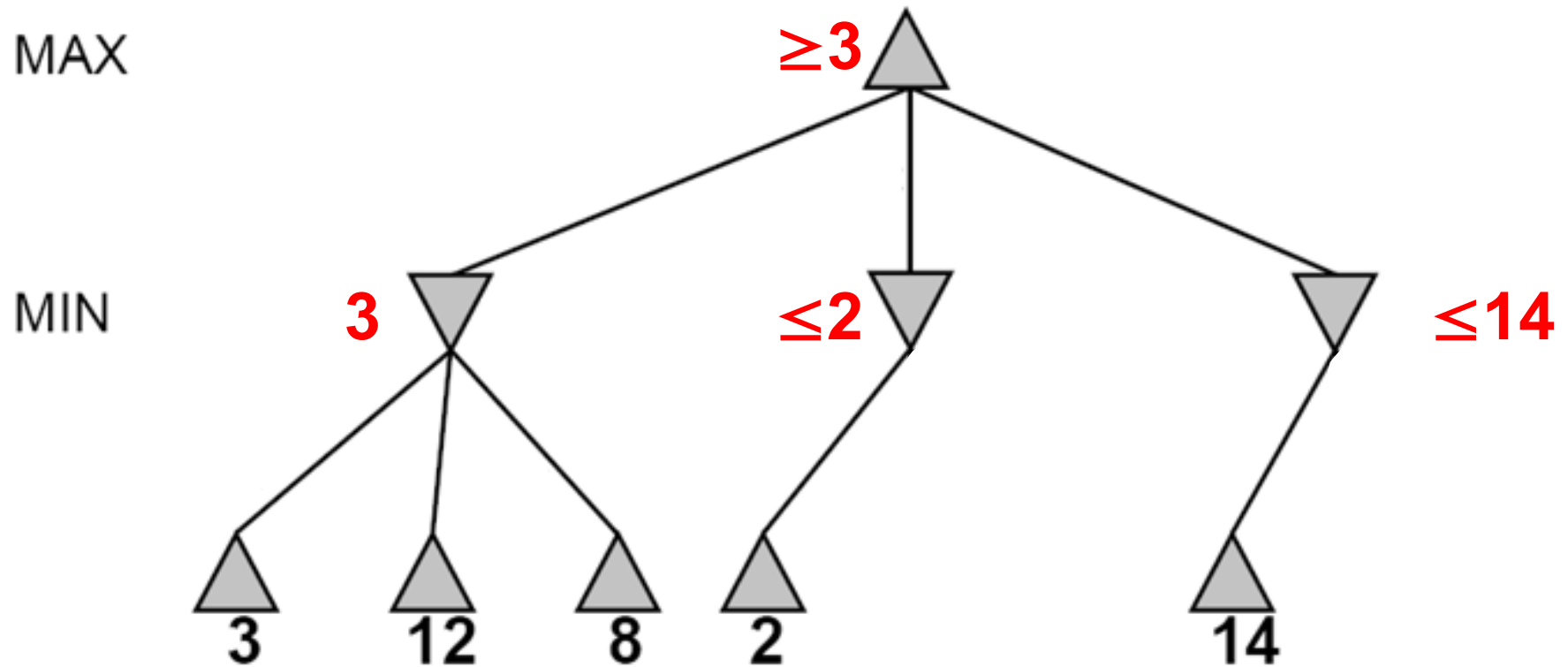
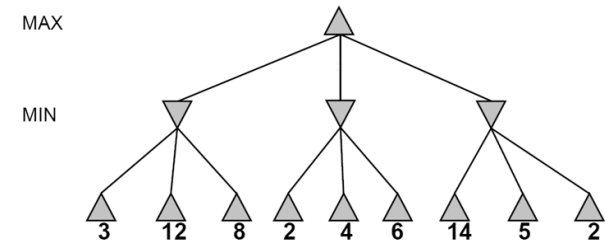


# Alpha-beta pruning



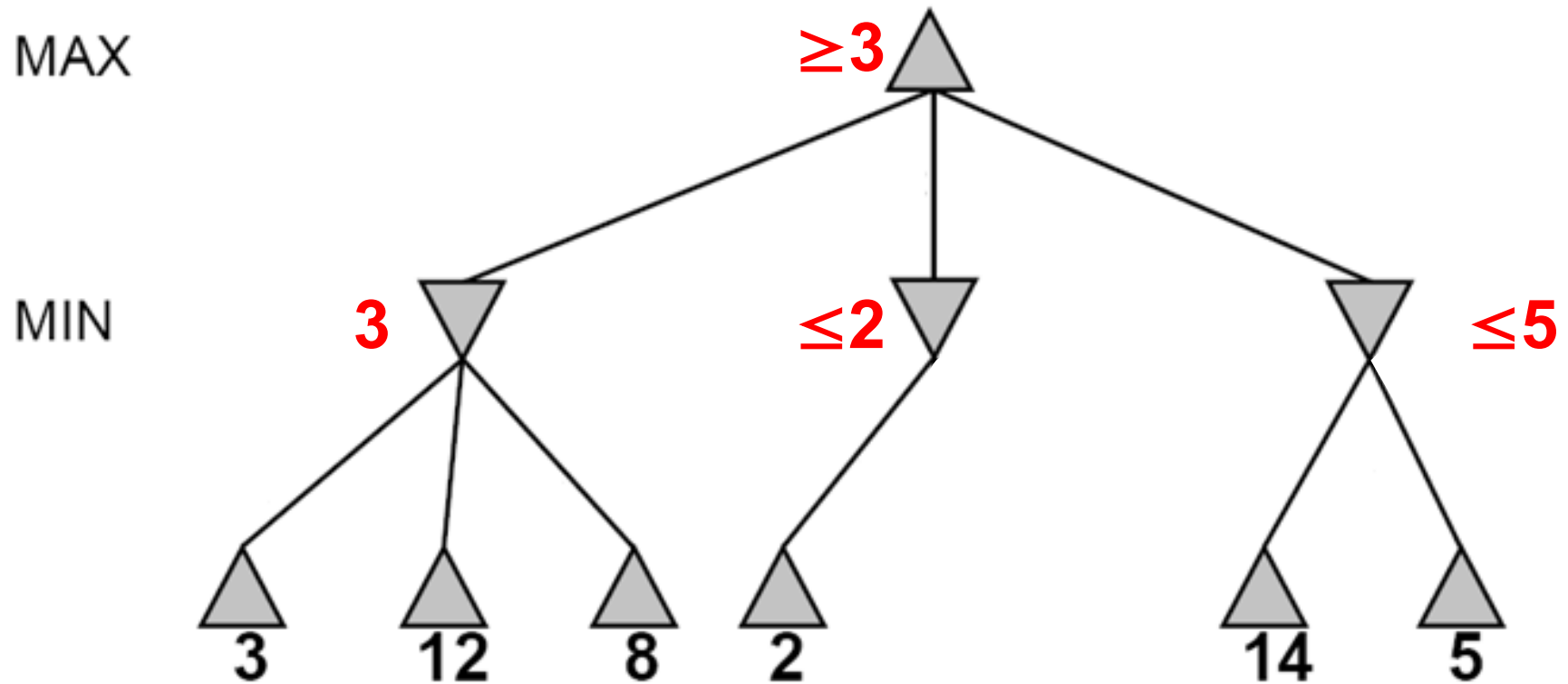
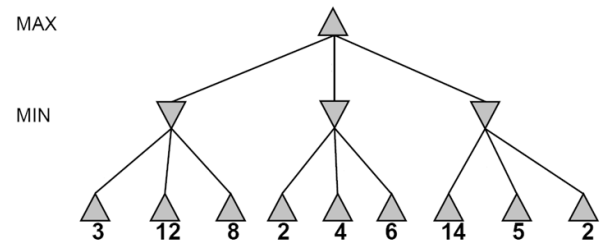
# Alpha-beta pruning

---

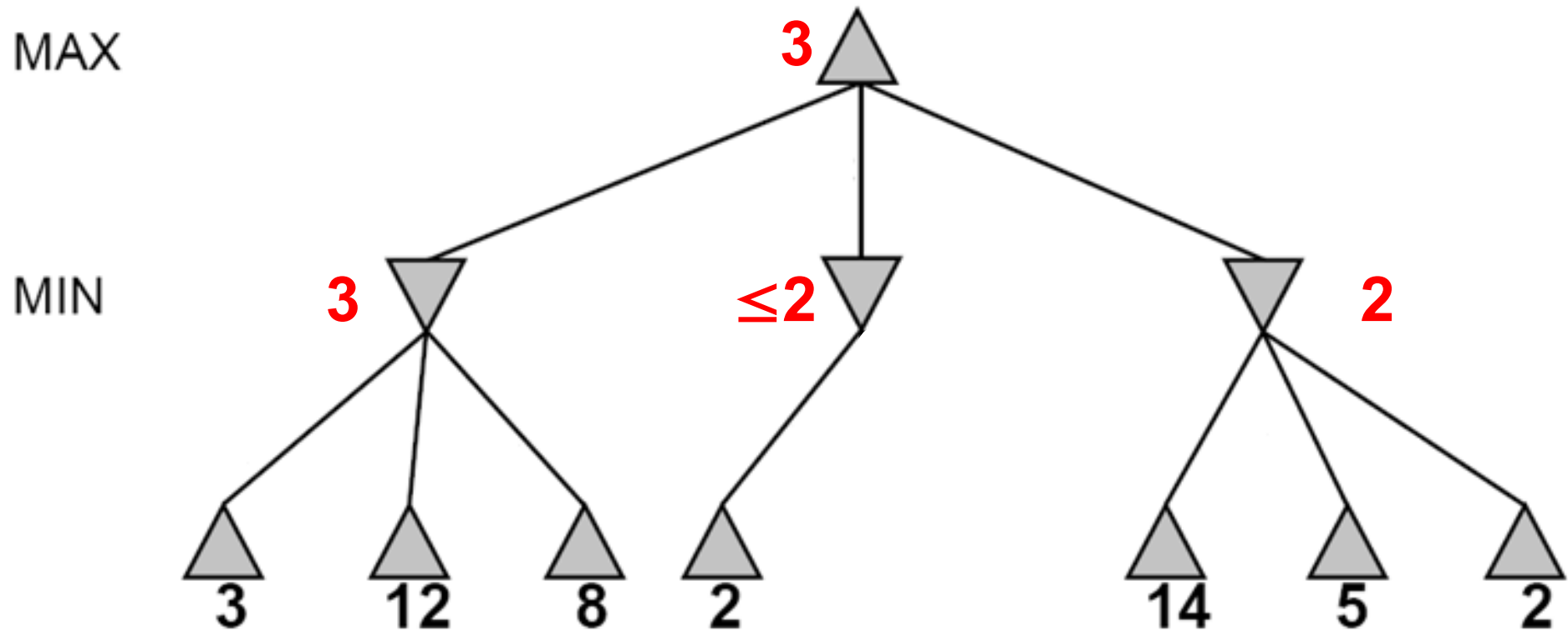
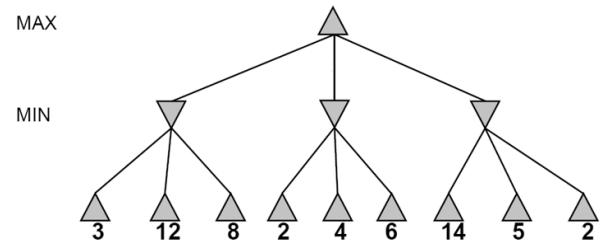


# Alpha-beta pruning

---



# Alpha-beta pruning





# Alpha-beta search

---

## Alpha-beta search

- updates the values of  $\alpha$ ,  $\beta$  as the search goes
- and prune **the nodes whose values are worse** than the current  $\alpha$  or  $\beta$  value (**current best choice**) for MAX or MIN, respectively.

In chess, the branching is about **35**. Alpha Beta search can reduce it to **about 6** and therefore enables a **10 ply search on a regular PC**.



# Alpha-beta demo

---

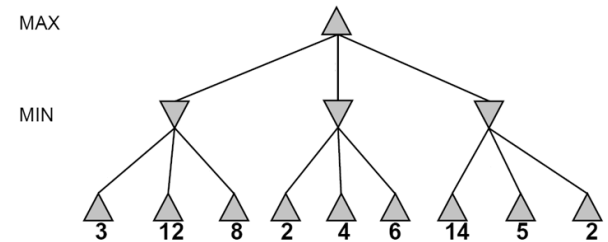
<http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html>

# Reflection on Alpha-Beta search

---

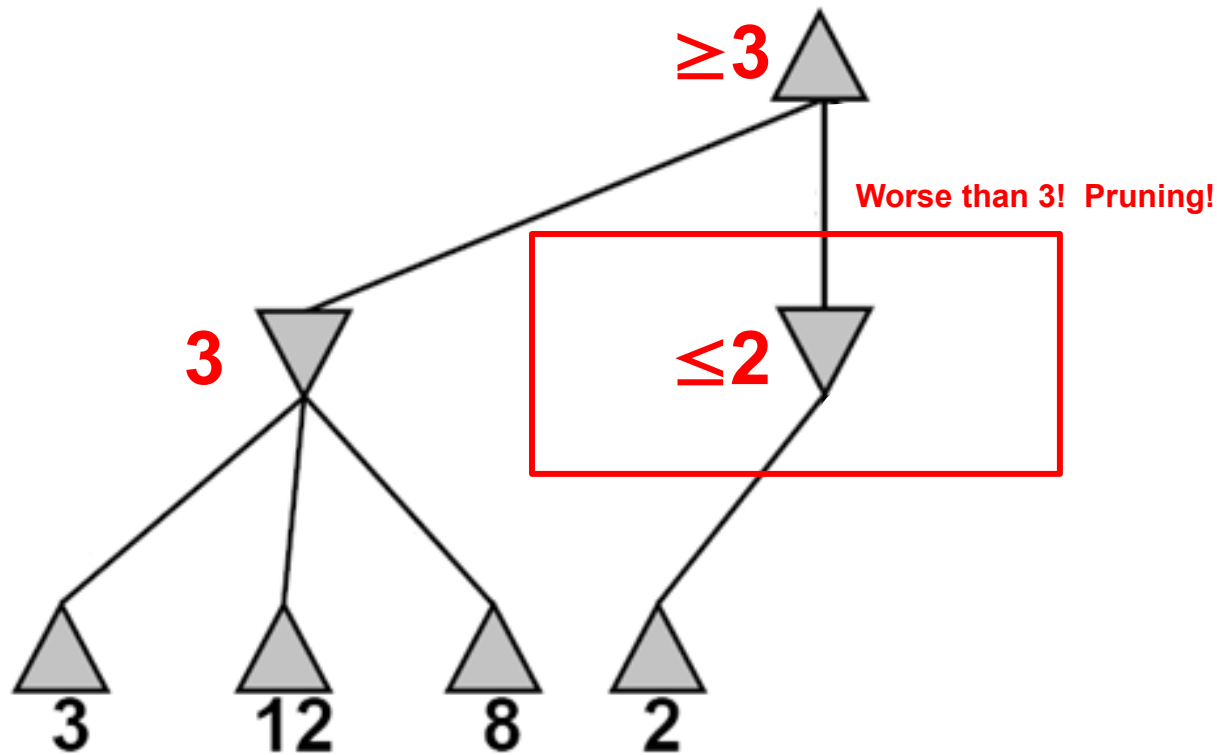
- Alpha-Beta pruning has **no effect on the minimax value for the tree root node**. However, the value at intermediate nodes may not be the exact minimax values. How to solve this?
- Answer: We **run alpha beta pruning for each child of the current node** and then **choose the action that moves to the successor with the maximum minimax value**.
- Good child ordering improves effectiveness of pruning.
  - With “perfect ordering”: Time complexity drops to  $O(bm/2)$ .

# Alpha-beta pruning with child ordering

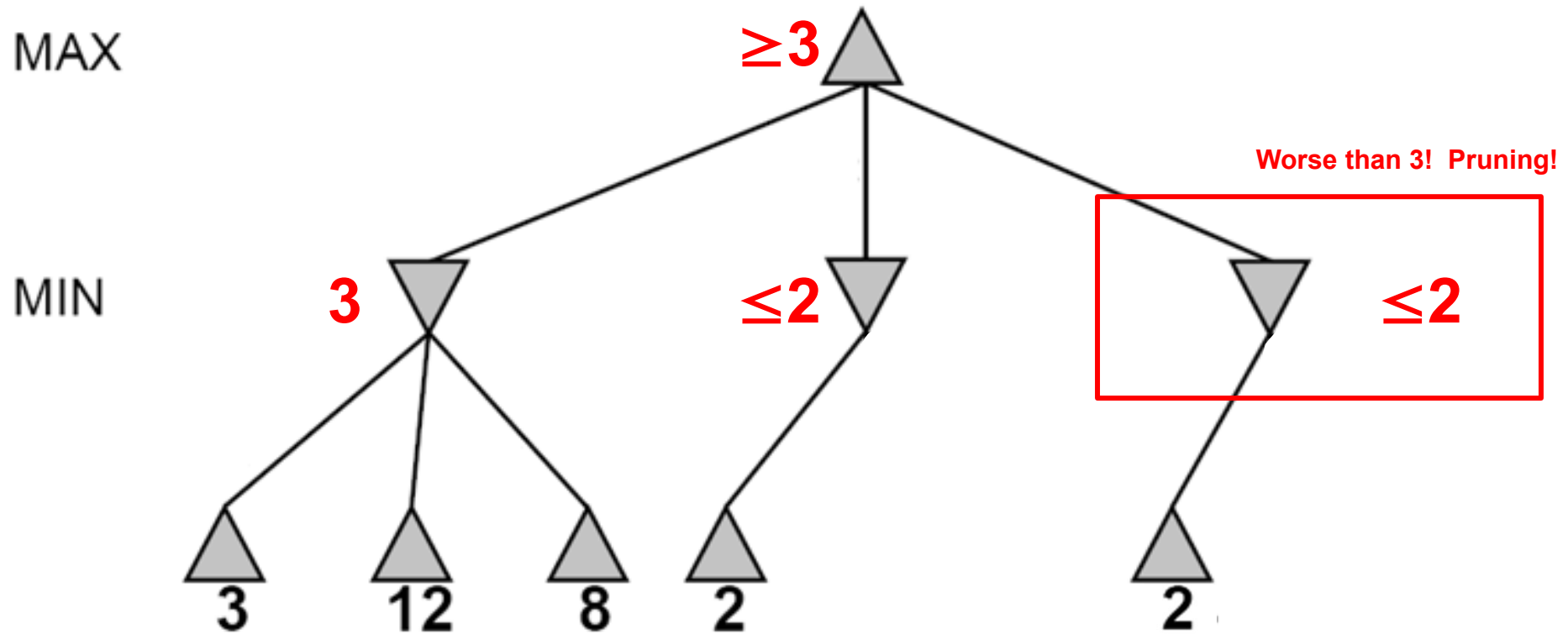
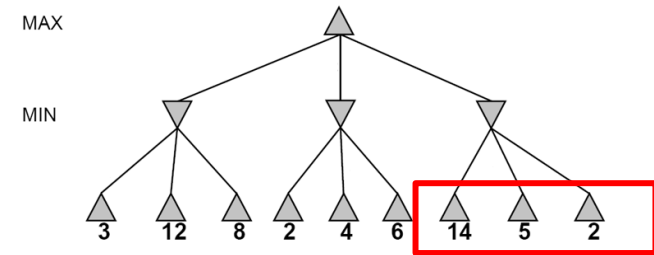


MAX

MIN



# Alpha-beta pruning with child ordering



# Alpha-Beta search on deep tree game

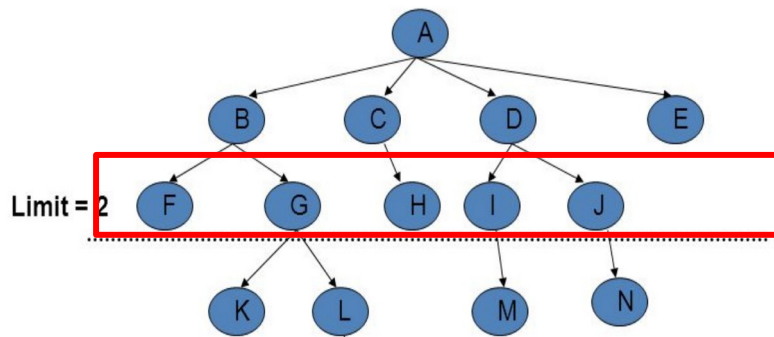
---

- Alpha beta pruning works only for small problems (shallow tree).
- How can we apply alpha beta pruning to solve large problems?
- Basic idea: We cut the tree at a certain depth to limit the search for a tradeoff between computation cost and decision optimality.

Approach: limited-depth alpha-beta search with heuristic function.

# Limited-depth search with heuristic function

- Step 1: We cut off the game tree at a certain depth,  $d$ , called a *d-ply search*.
- Step 2: For any non-terminal states at the depth  $d$ , we apply some heuristic function to estimate the success likelihood (e.g., minimax value, the probability to win) of those states.
- Step 3: Apply alpha-beta pruning in the *d-ply search*



**What would be a good heuristic to estimate the success likelihood of a non-terminal states for chess?**

# Heuristic function

---

- A heuristic function could be as simple as a weighted sum of some features.

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

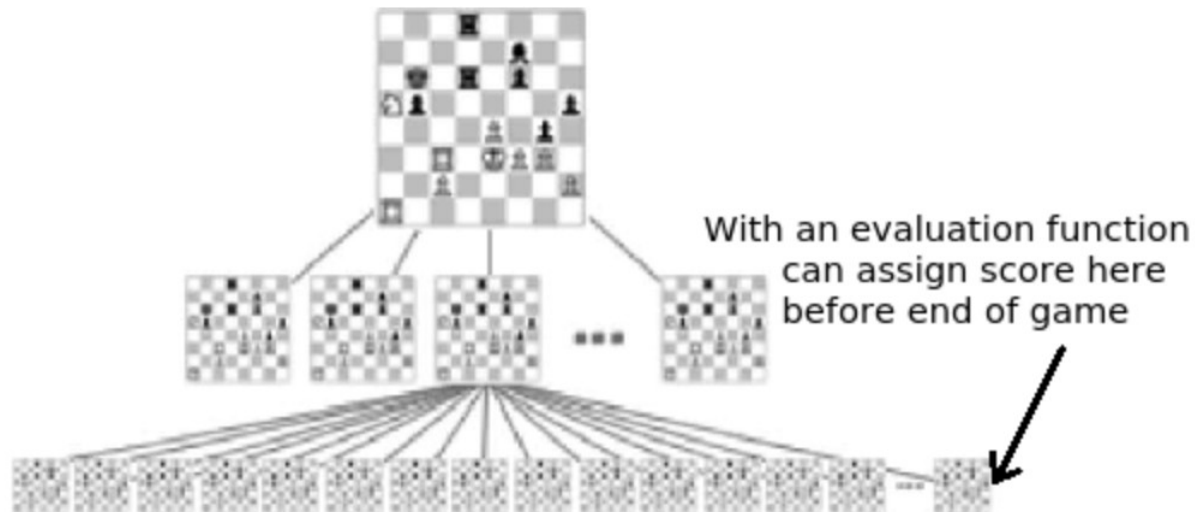
- Each feature:
  - $f_k(s)$  is the advantage in terms of a piece, e.g.,  $f_1(s) = (\text{number of white queens}) - (\text{number of black queens})$

Symbol					
Piece	pawn	knight	bishop	rook	queen
Value	1	3	3	5	9



# Limited-depth search with heuristic function

---



# Limited-depth search with heuristic function

---

- Deep Blue was a chess-playing agent developed by IBM.
  - Mainly based on **limited-depth alpha-beta search with heuristic function**.
  - Searches 200 million states per second, uses very sophisticated heuristic function, and performs search **up to 40 ply**.
- Deep Blue beat human world champion Gary Kasparov in a six-game match in 1997.



# Minimax alpha-beta pruning implementation

---

# Beyond minimax search

---

- Most game-playing AIs (Deep Blue) in the past are **based on game tree search** we learned in this lecture.
- However, today's game AIs (e.g., AlphaGo) **rely less on tree search and more on knowledge** (machine learning).



# AlphaGo (an AI from Google)



---


nature.com > nature > articles > article

MENU ▾ **nature**  
International journal of science

Article | Published: 27 January 2016

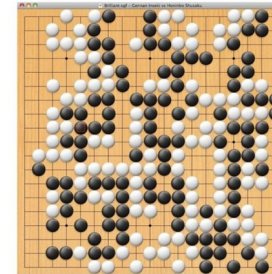
## Mastering the game of Go with deep neural networks and tree search

David Silver , Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel & Demis Hassabis 

Nature **529**, 484–489 (28 January 2016) | [Download Citation](#) 

### Abstract

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and

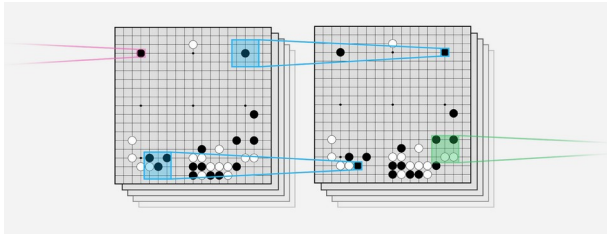


D. Silver et al., [Mastering the Game of Go with Deep Neural Networks and Tree Search](#), Nature 529, January 2016

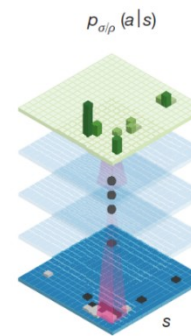
# AlphaGo

---

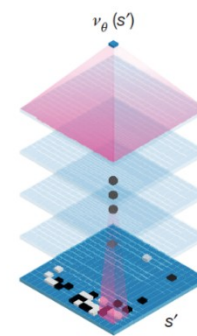
Treat the Go board as an image



Policy network

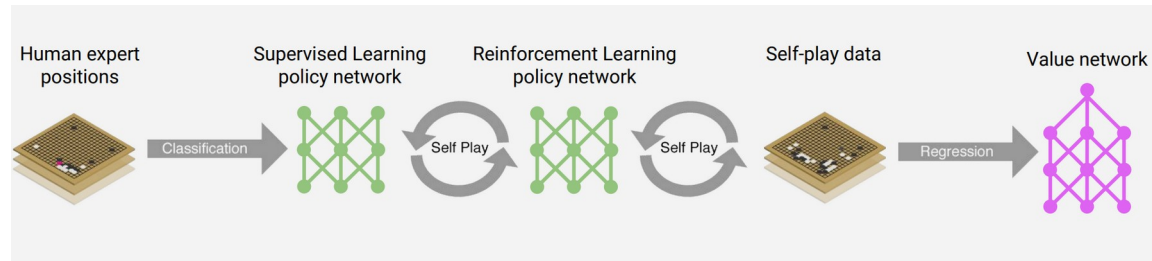


Value network



# AlphaGo

---

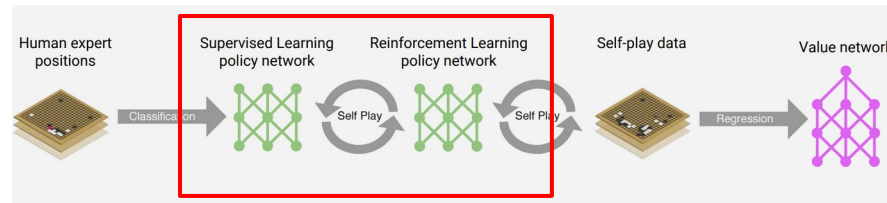


Policy network (CNN) was trained to predict human moves

Value network (CNN) was trained to predict reward of each state.

# AlphaGo uses two policy networks

---



## SL policy network (CNN)

- Given state  $s$ , predict probabilities of human moves
- Trained on 30M positions (states) from human experts, 57% accuracy on predicting human moves
- 12 layer-CNN
- 4 weeks on 50 GPUs using Google Cloud

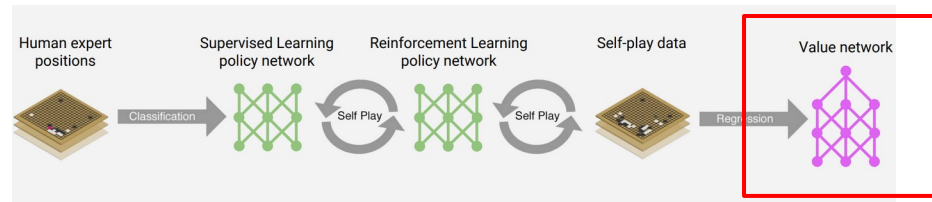
## RL policy network (CNN)

- Two CNN networks play against each other and update parameters.
- Machine plays against itself (self-play)
- 12 layer-CNN
- 1 week on 50 GPUs using Google Cloud



# AlphaGo uses one value network

---

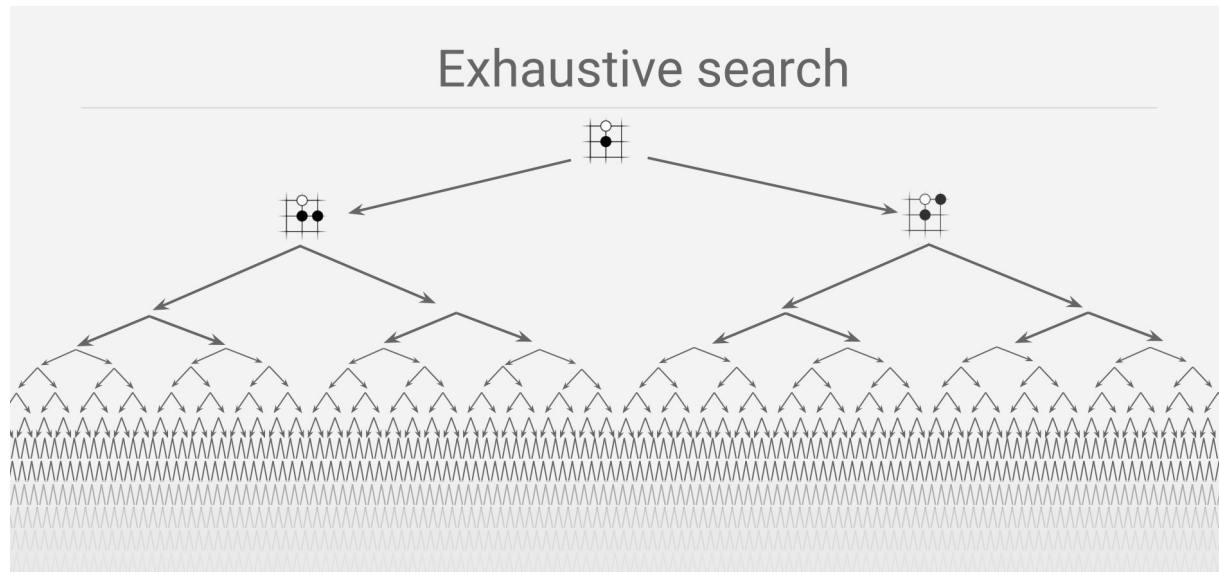


Value network (CNN)

- Given any state  $s$ , predict the reward of that state
- This is a regression network use stochastic gradient descent (SGD) to minimize mean squared error (MSR) between actual rewards (based on self-play) and predicted rewards
- 12 layer CNN
- Trained on 30M self-play positions (states)
- 1 week on 50 GPUs using Google Cloud

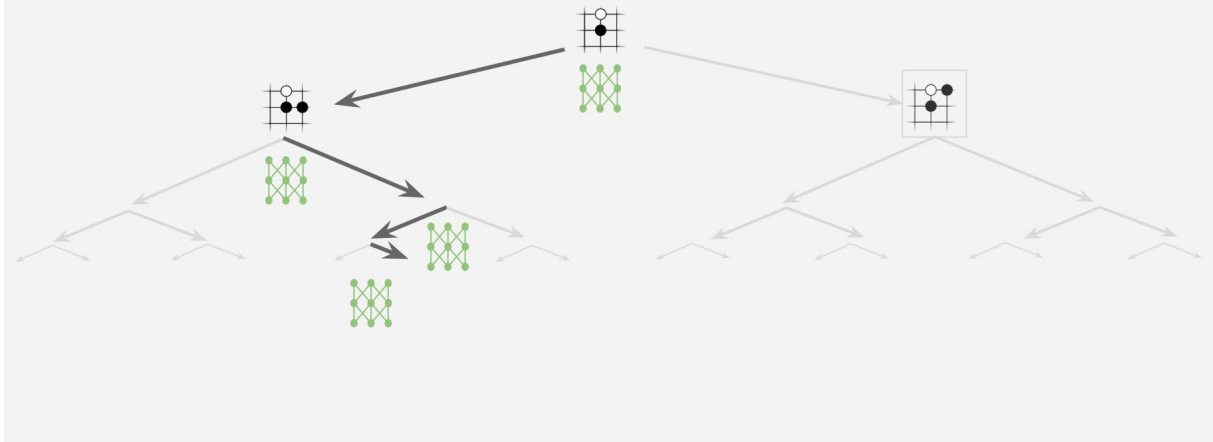
# Tree pruning

---



---

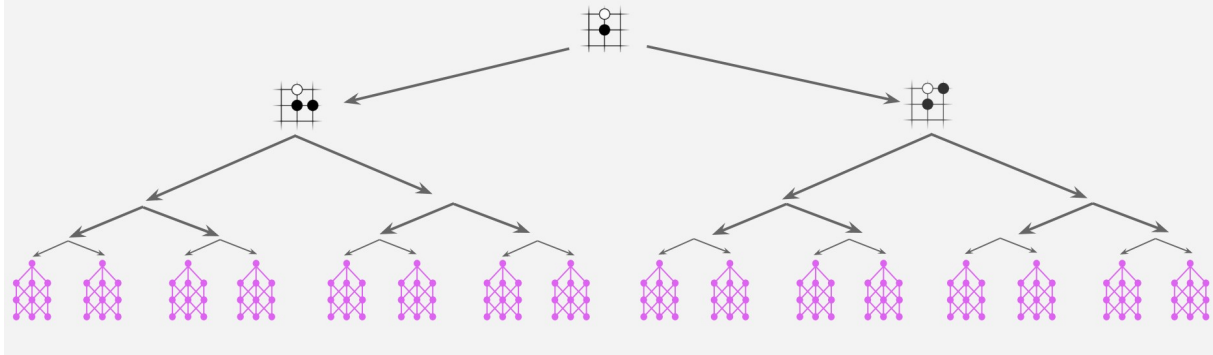
## Reducing breadth with policy network



Policy network (CNN) was trained to predict human moves

---

## Reducing depth with value network



Value network (CNN) was trained to predict reward of each state.