CS7IS2: Artificial Intelligence

Games and Adversarial Search

Adversarial search

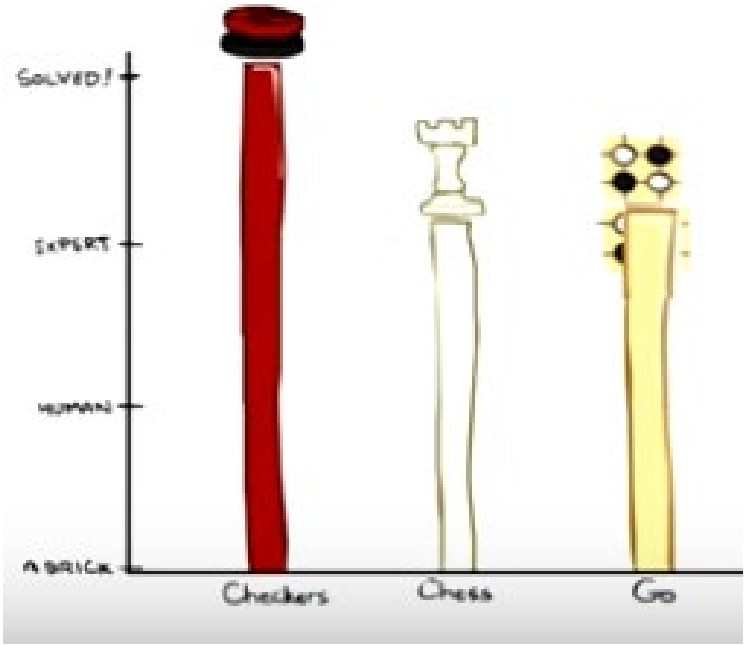
- Adversarial games - chess, business, trading, wars ...
- You change state, but then don't control next state.
- Opponent will change state in a way:
  - unpredictable
  - hostile to you
- Solution is a strategy specifying a move for every possible opponent reply
- Time limits => unlikely to find goal, must approximate

History of game playing

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- Go: 2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.



AI games

- AAAI-20 AI History Panel: Advancing AI by Playing Games
- https://www.youtube.com/watch?v=In14zA1XAVk
- Moderator: Amy Greenwald (Brown University)
  Panelists: Murray Campbell (IBM), Michael Bowling (University of Alberta), and Hiroaki Kitano (Sony), Garry Kasparov, and David Silver (Deepmind and University College London)

- Gary Kasparov –ted talk – don't fear the machines, work with them
- https://www.youtube.com/watch?v=NP8xt8o4_5Q

Types of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tic-tac-toe	bridge, poker, scrabble nuclear war

“Classical” (economic) game theory includes cooperation, chance, imperfect knowledge, simultaneous moves and tends to represent real-life decision making situations

Deterministic games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA → S
  - Terminal Test: S → {t,f}
  - Terminal Utilities: SxP → R

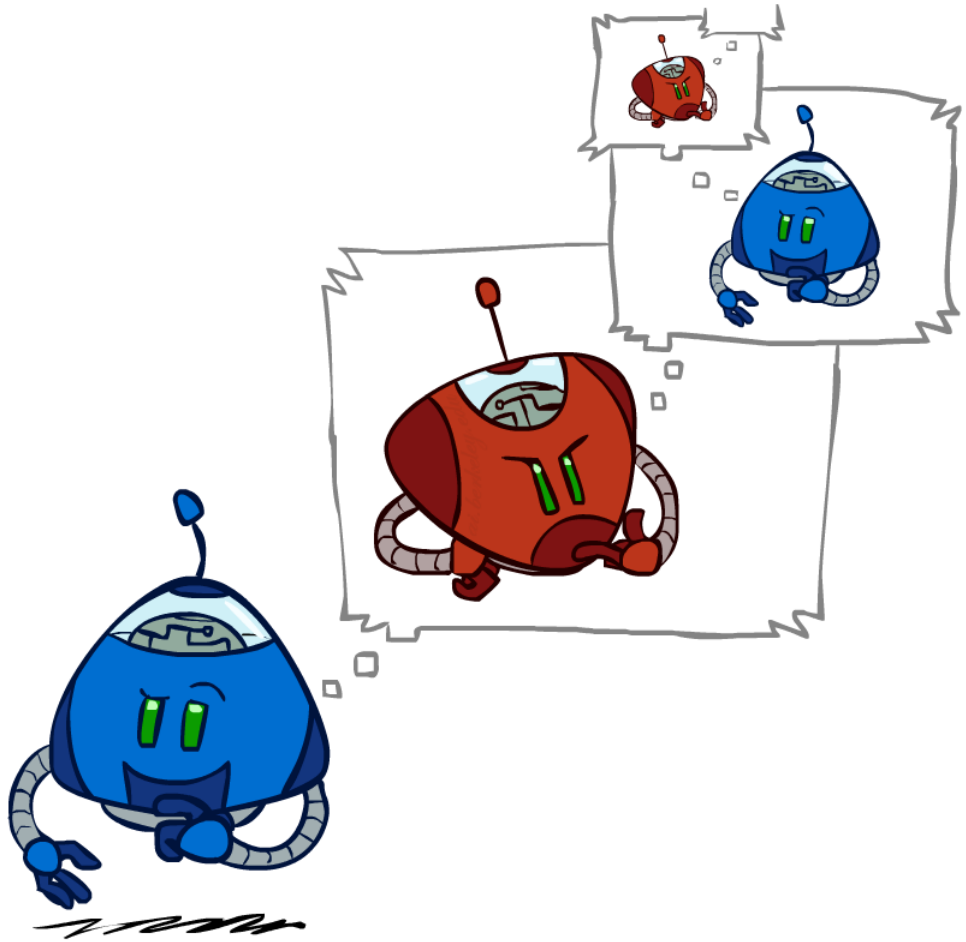
- Solution for a player is a policy: S → A

Zero-Sum games

- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible

Adversarial search

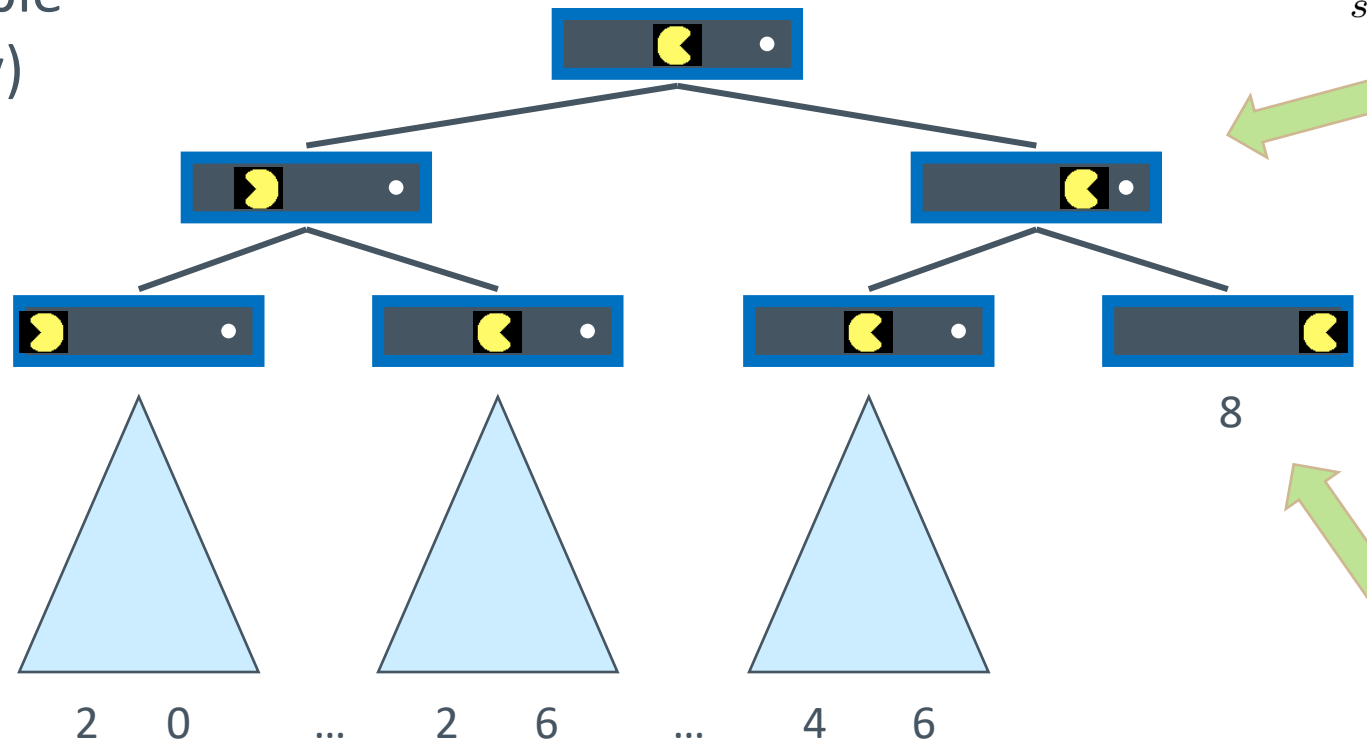


Single-agent trees

Value of a state:
The best achievable outcome (utility) from that state

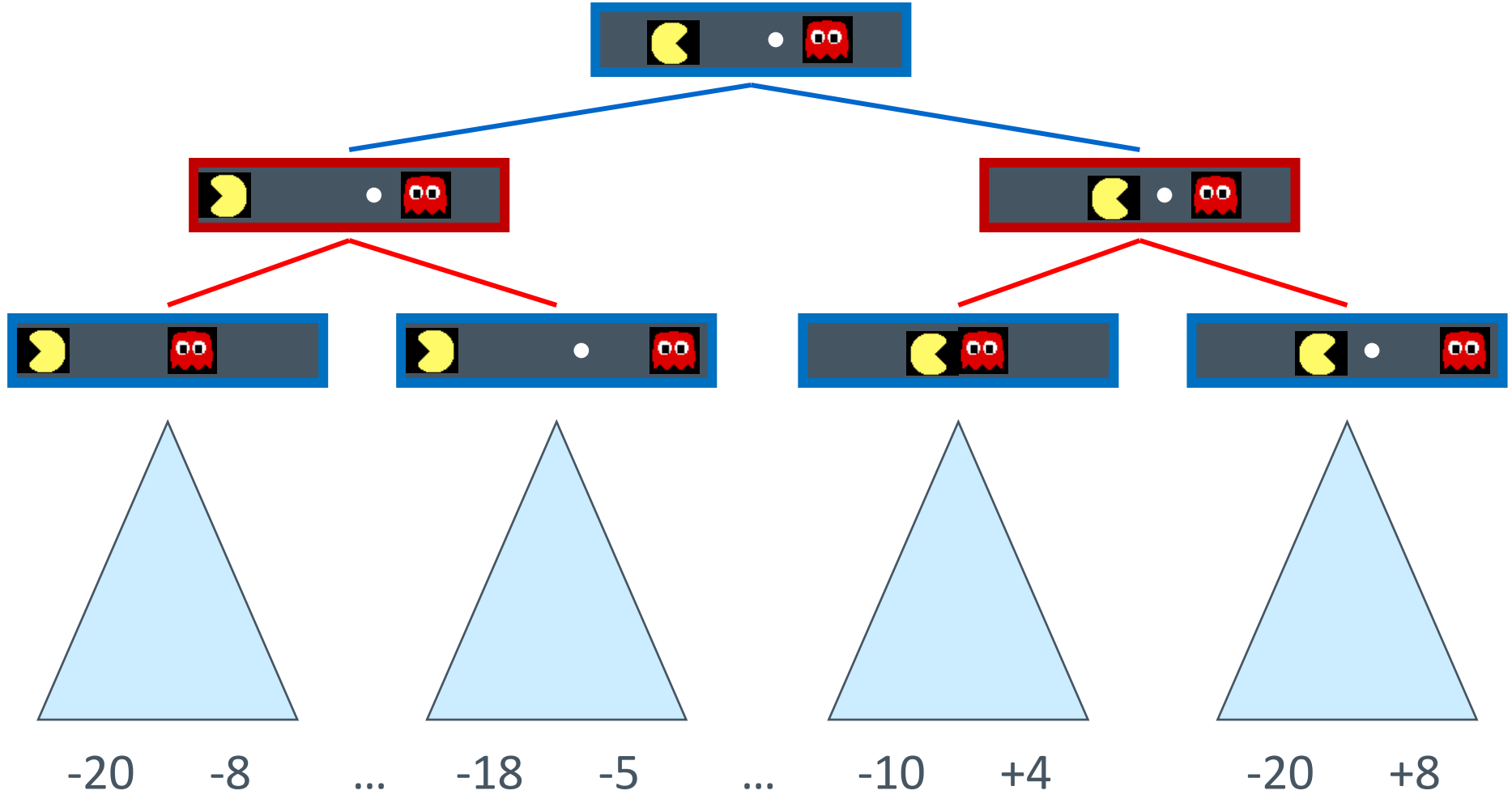
Non-Terminal States:

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



$$V(s) = \text{known}$$

Adversarial game trees



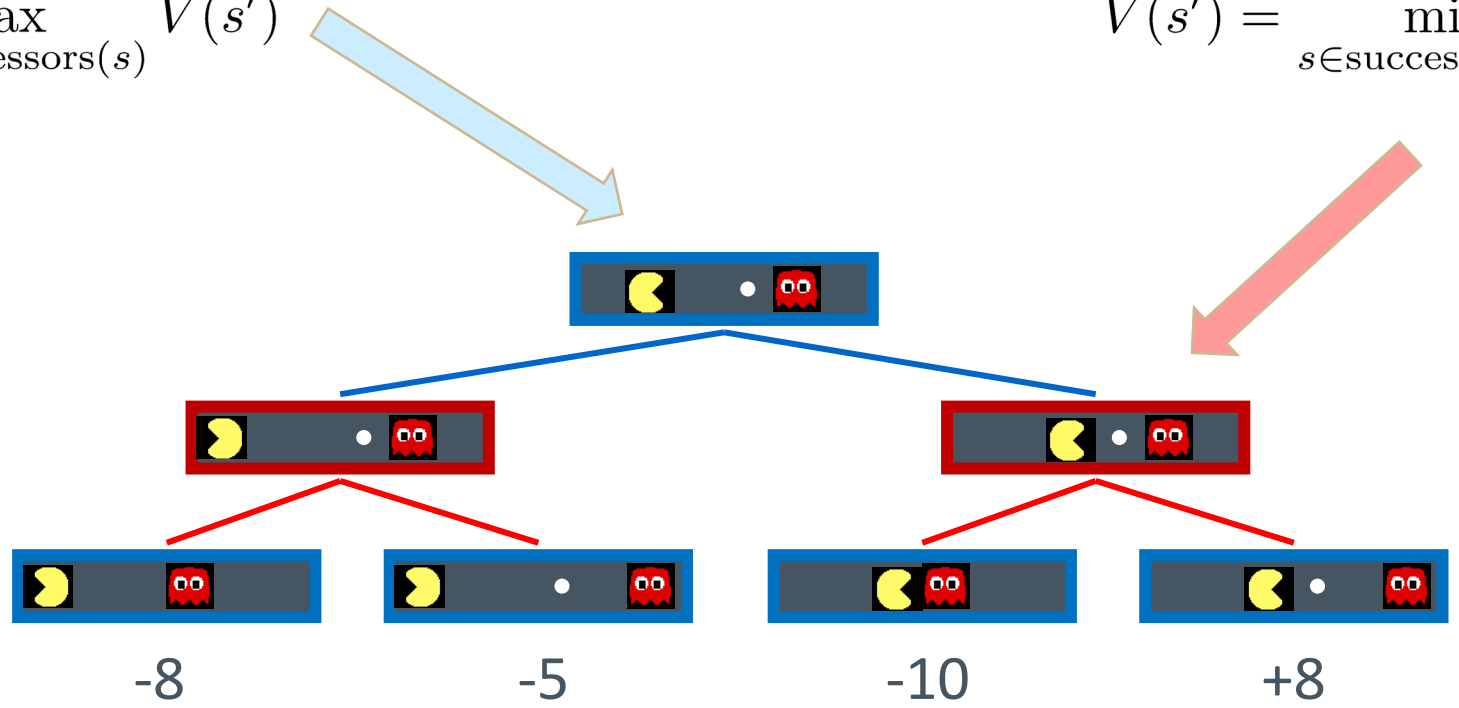
Minimax values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

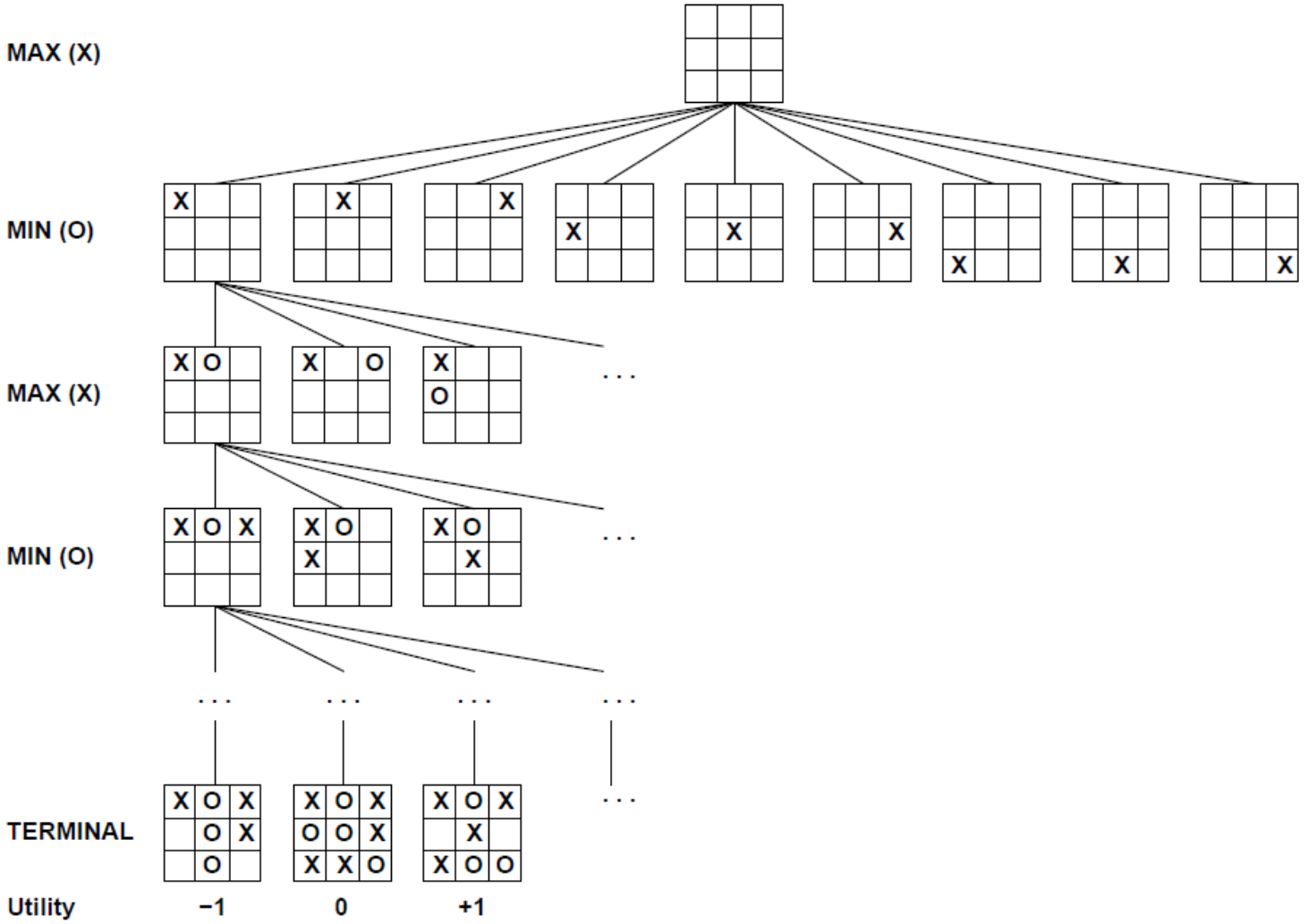
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

Game tree (tic-tac-toe: 2-player, deterministic, turns)

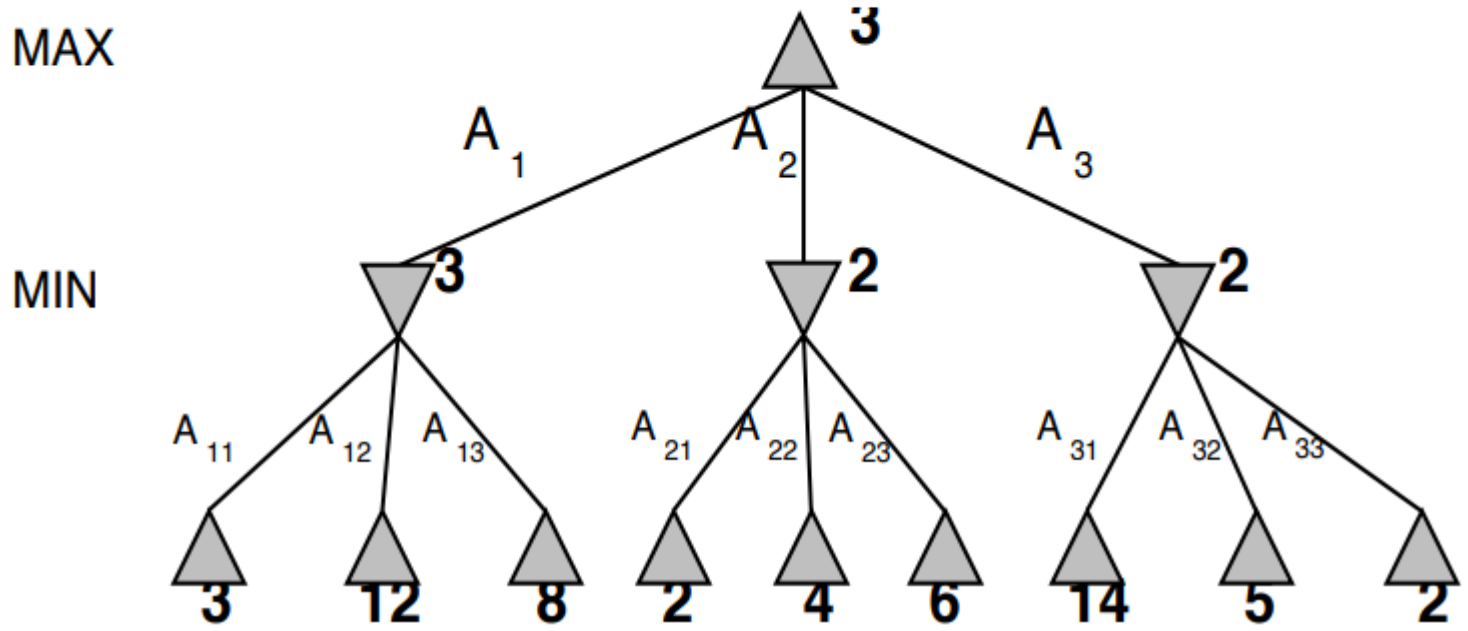


Minimax algorithm

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest minimax value = best achievable payoff against best play

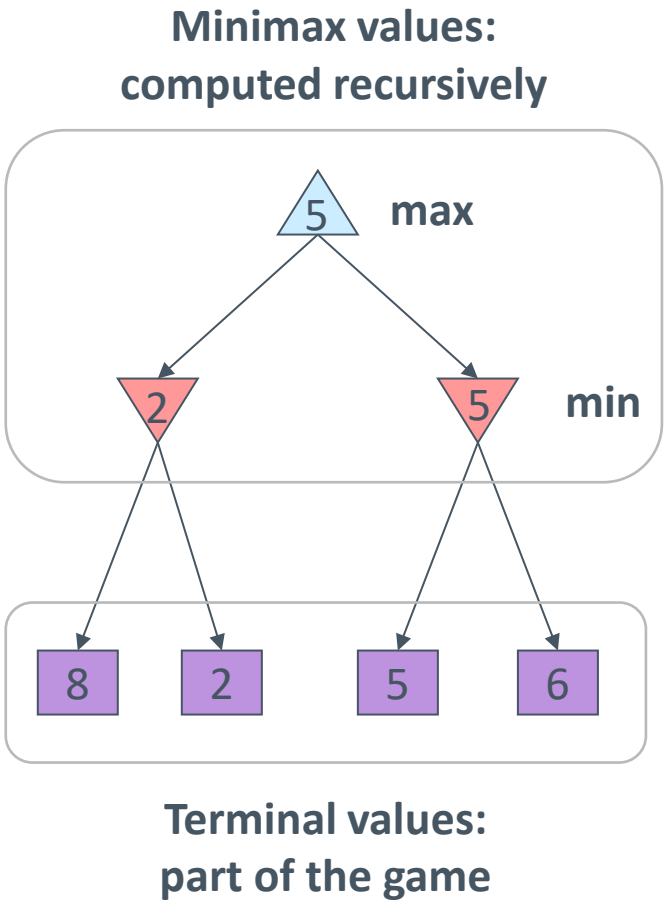
E.g., 2-ply game



Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



Minimax Algorithm

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** $v \leftarrow$ MAX(v, MIN-VALUE(s))
  return v

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

---

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** $v \leftarrow$ MIN(v, MAX-VALUE(s))
  return v

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

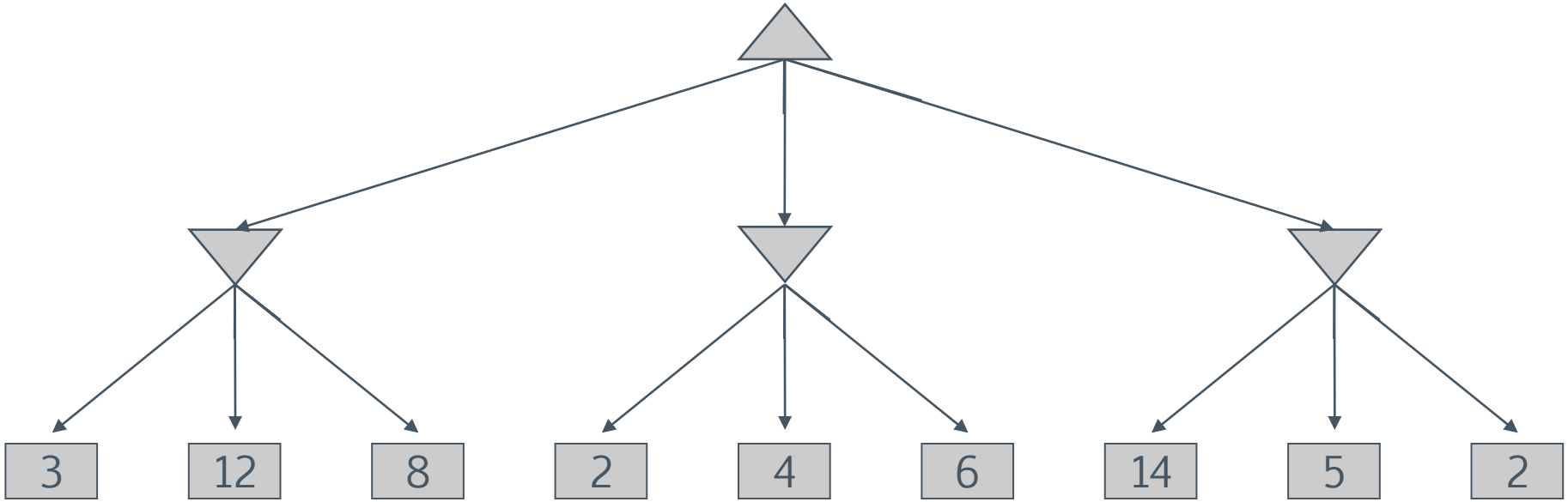
Minimax algorithm

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

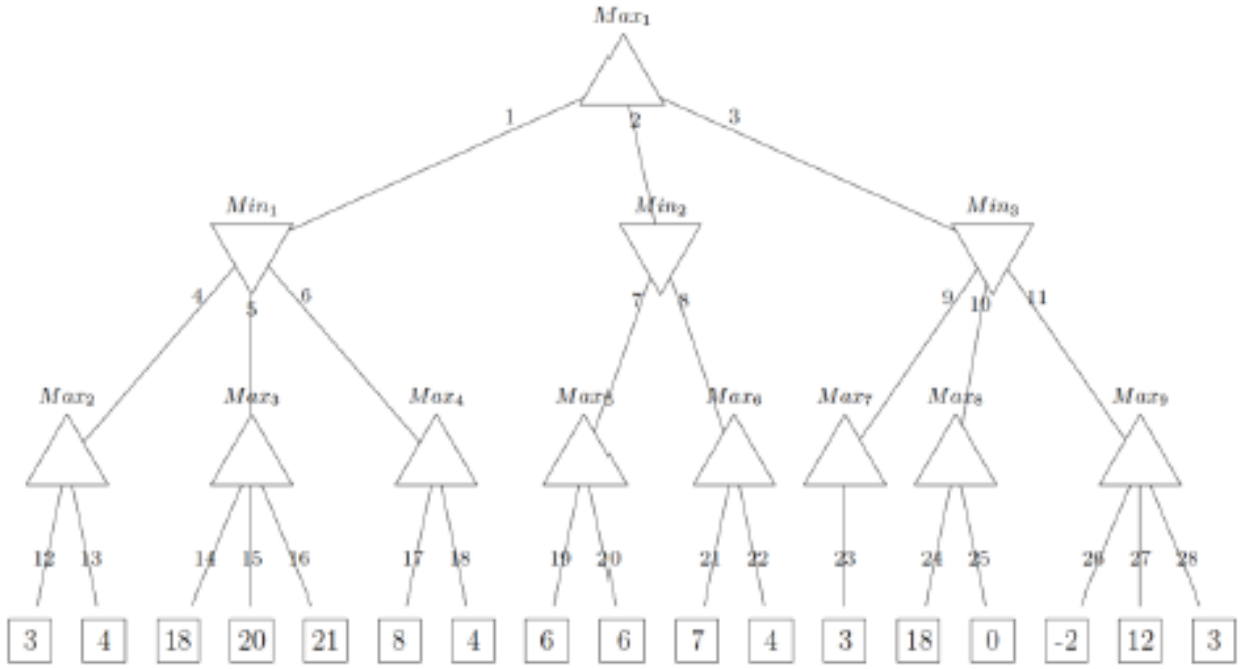
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Minimax algorithm – sample problem



Minimax Exercise

In the game below, it is MAX (△) turn to move and the game does not involve randomness. The terminal node shows the utility values for MAX. Write the values of the MIN (Min_1,...,Min_3) and MAX (Max_1,...,Max_9) nodes. Write the path that corresponds to the best move for MAX.



Minimax properties

Complete?

- Yes, if tree is finite (chess has specific rules for this)

Optimal?

- Yes, against an optimal opponent. Otherwise?

Time complexity?

- $O(b^m)$

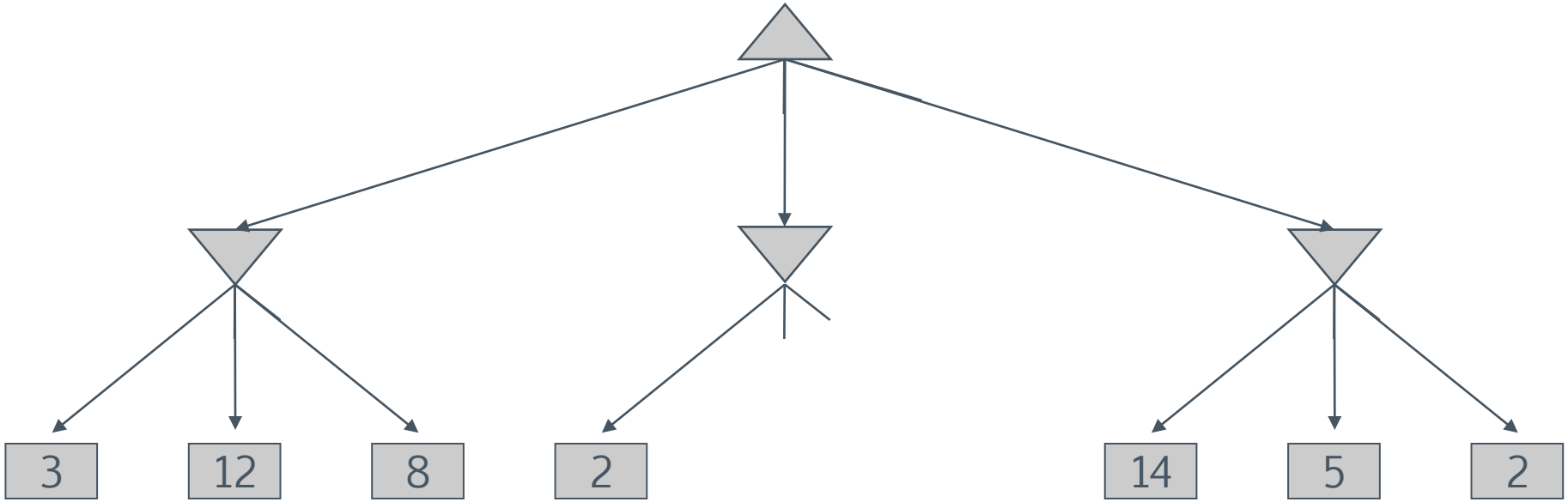
Space complexity?

- O(bm) (depth-first exploration)

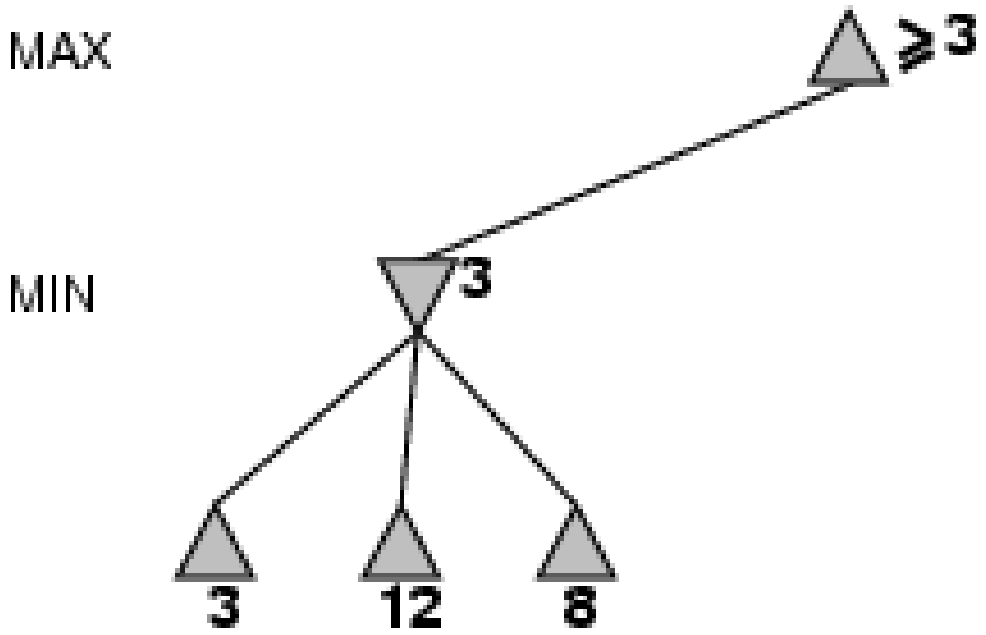
For chess, b≈35, m ≈ 100 for “reasonable" games => exact solution completely infeasible

But do we need to explore every path?

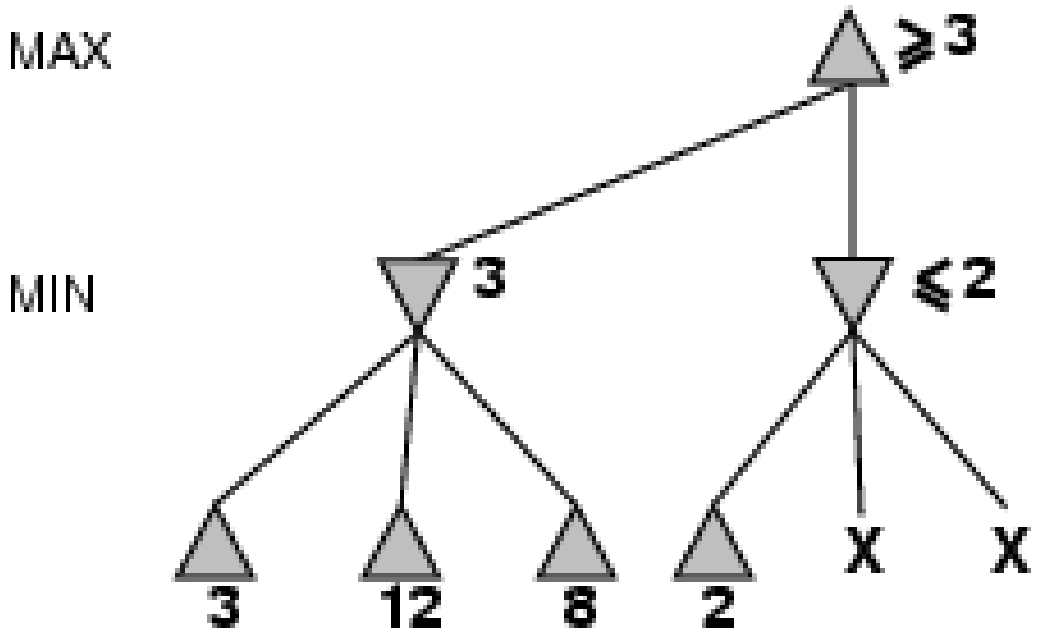
Minimax Pruning



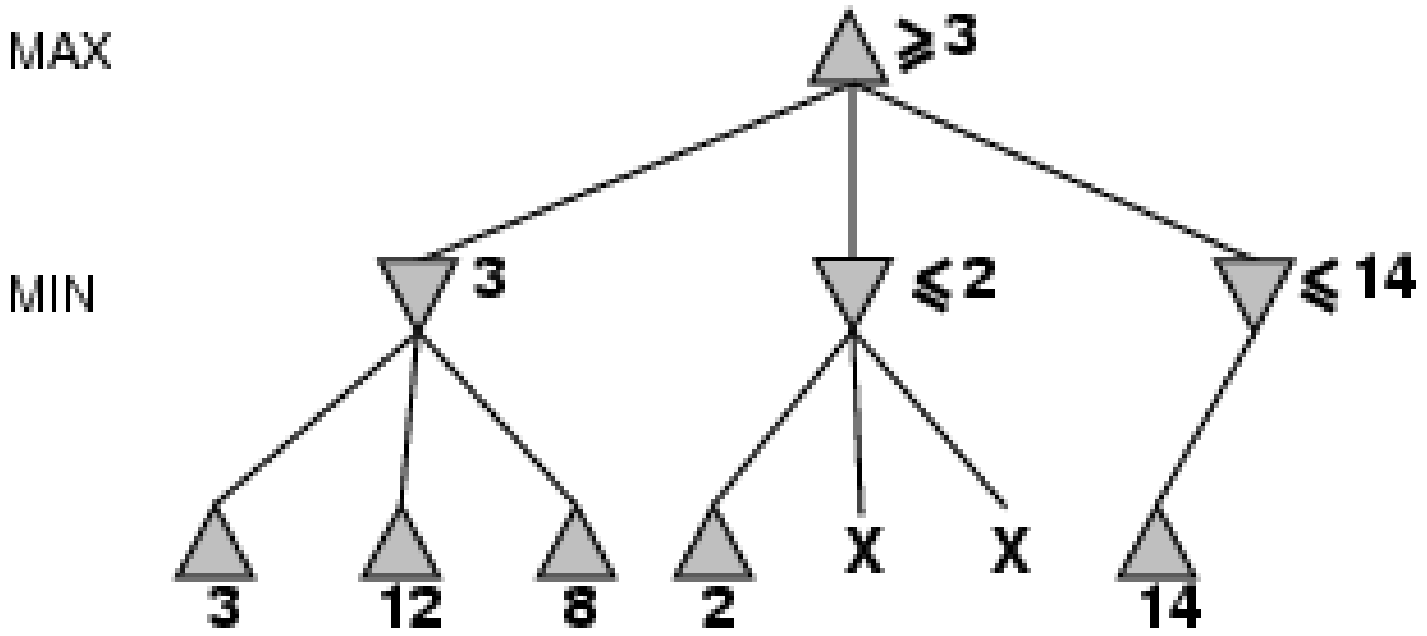
α-β pruning example



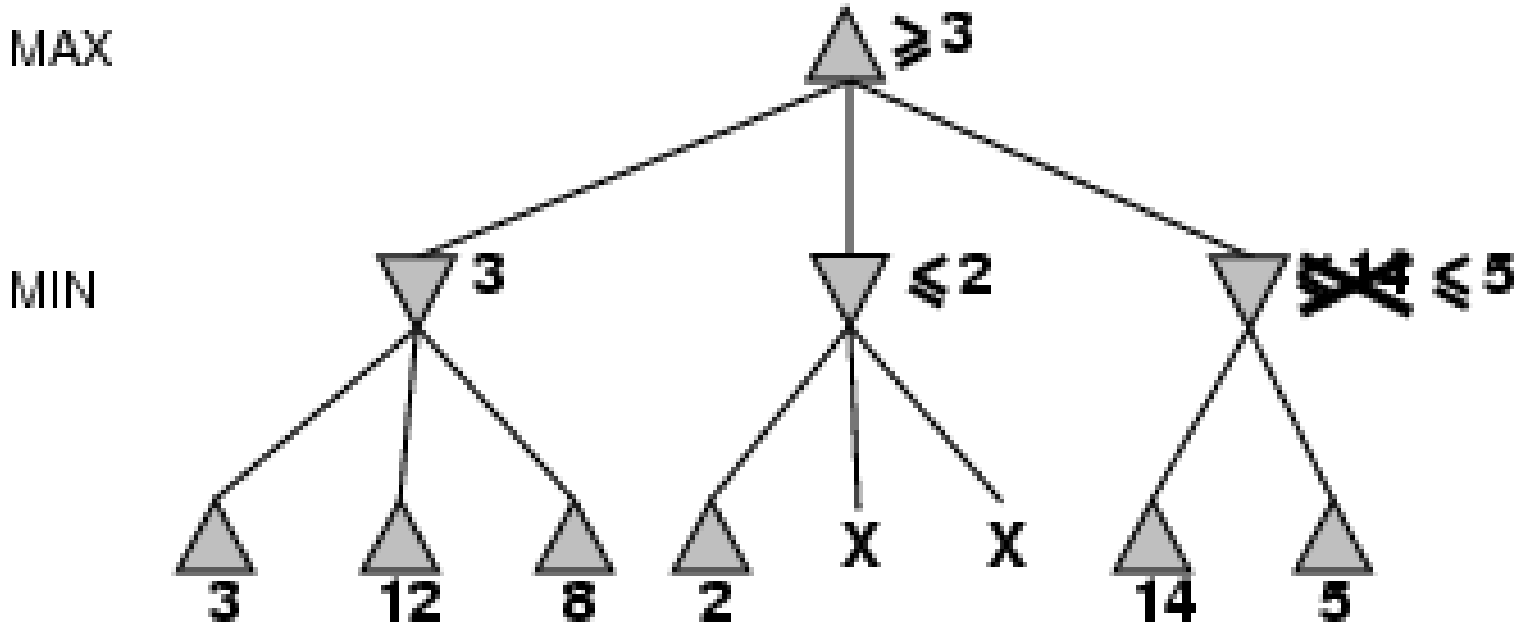
α-β pruning example



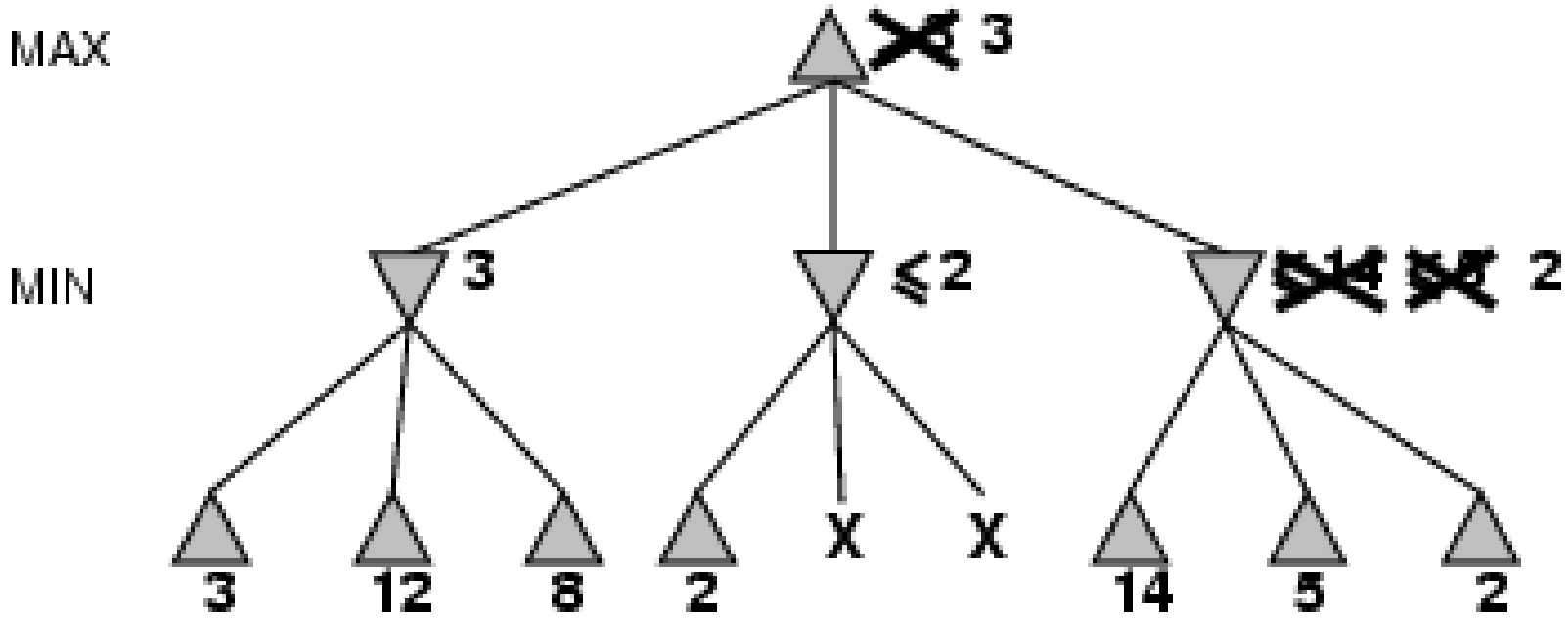
α-β pruning example



α-β pruning example

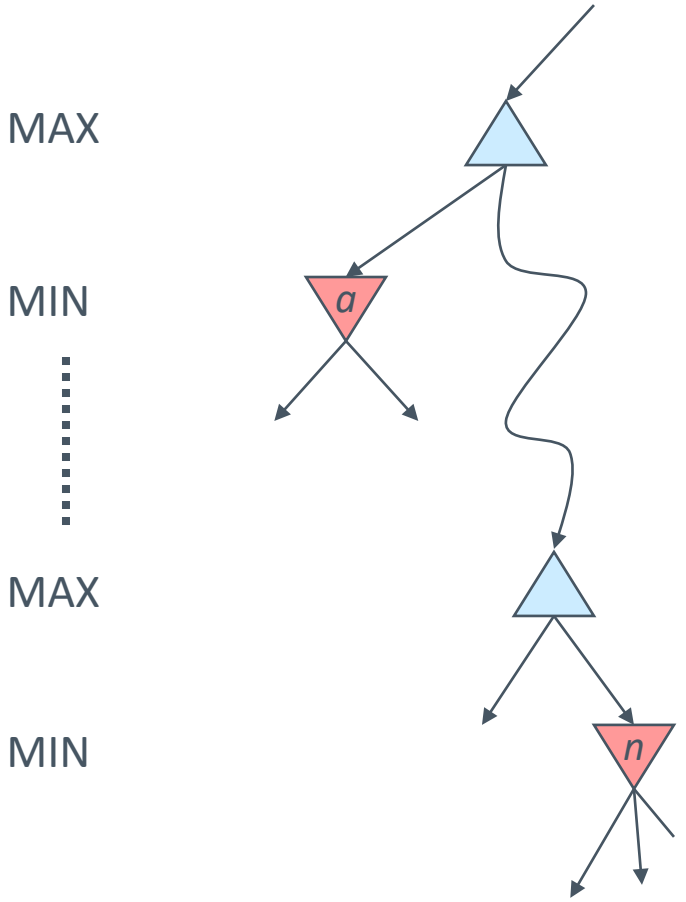


α-β pruning example



Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



- MAX version is symmetric

Alpha-Beta Implementation

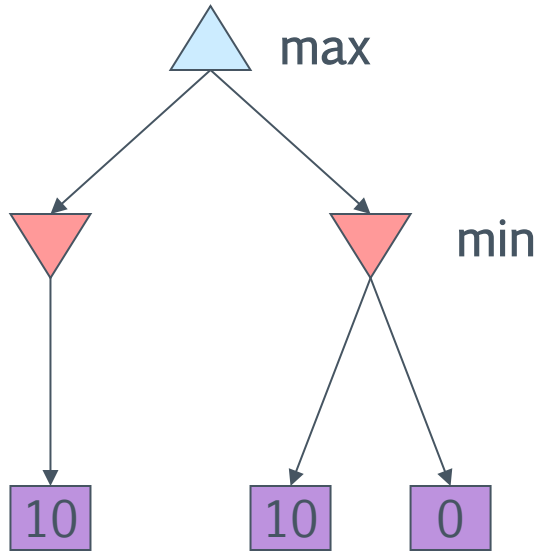
α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

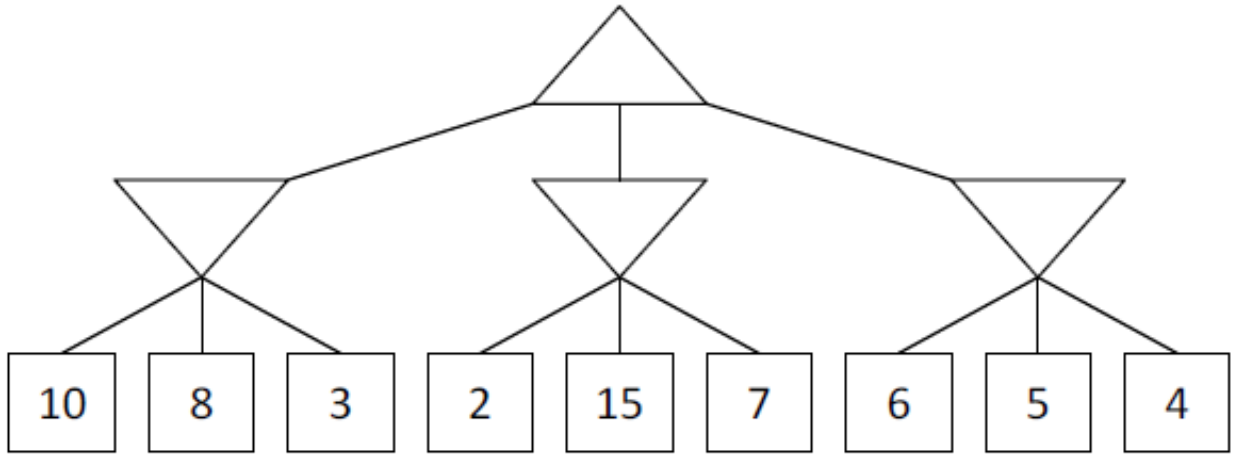
Alpha-Beta properties

- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)



Sample problem

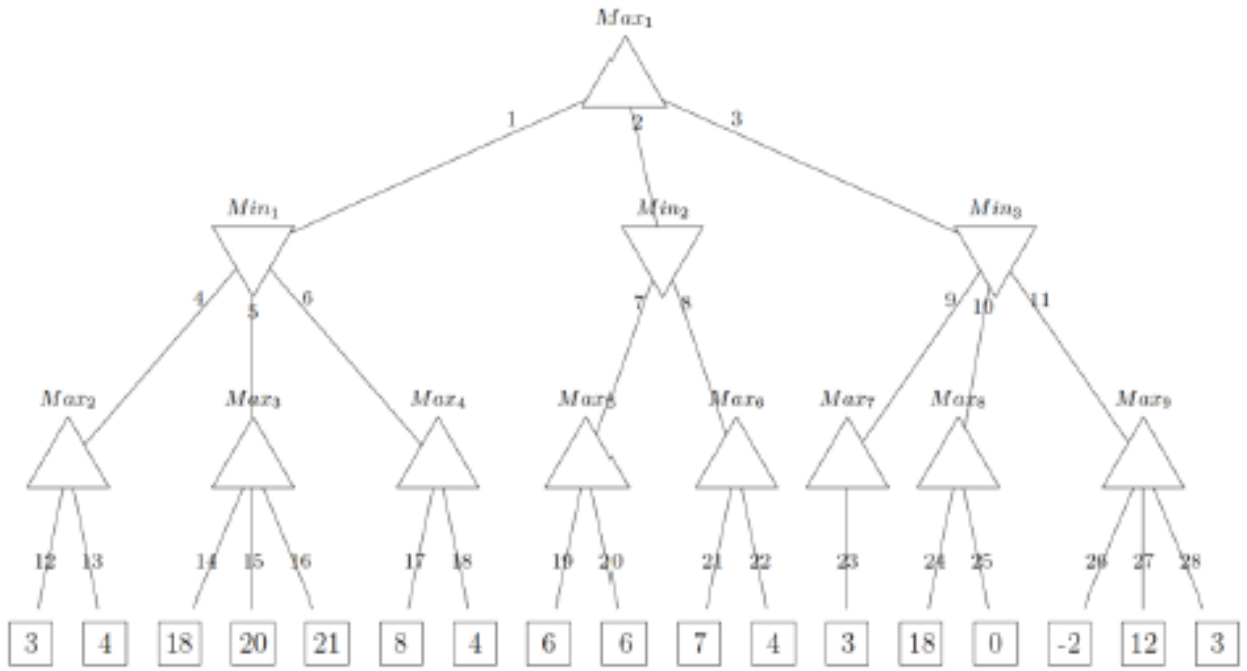
Consider the zero-sum game tree shown below. Triangles that point up, such as at the top node (root), represent choices for the maximizing player; triangles that point down represent choices for the minimizing player. Assuming both players act optimally, fill in the minimax value of each node.



Which nodes can be pruned from the game tree above through alpha-beta pruning? If no nodes can be pruned, explain why not. Assume the search goes from left to right; when choosing which child to visit first, choose the left-most unvisited child.

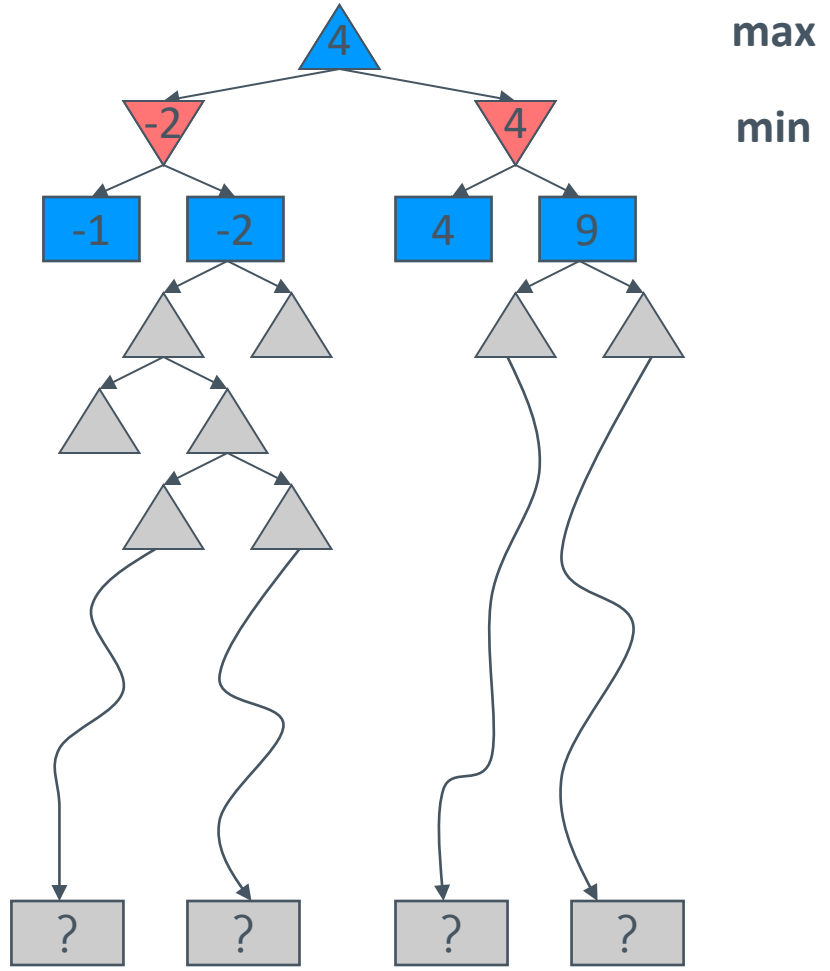
Alpha-beta pruning exercise

This is the same tree as above, it is MAX (△) turn to move and the game does not involve randomness. The terminal node shows the utility values for MAX. Write the path number (1,..., 28) of all the paths that will not be examined because they are pruned by **Alpha-Beta Pruning.**



Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



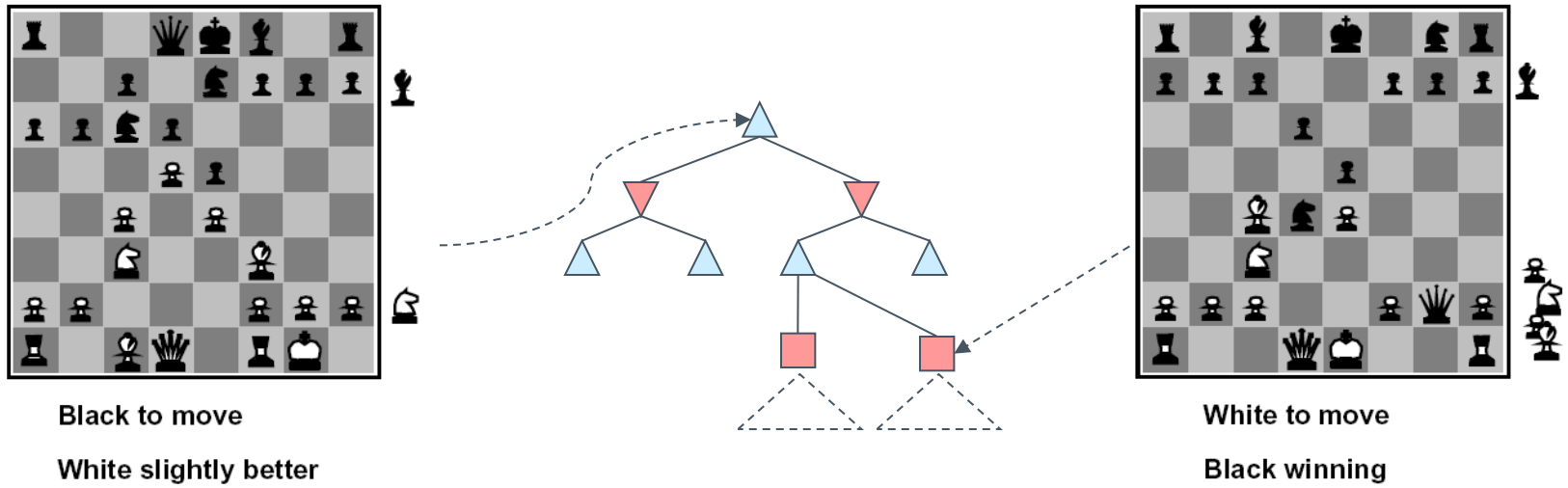
Resource Limits

Standard approach:

- Use `Cutoff-Test` instead of `Terminal-Test`
  e.g., depth-limited search
- Use `Eval` instead of `Utility`
  i.e., evaluation function that estimates desirability of position

Evaluation functions

› Evaluation functions score non-terminals in depth-limited search



› Ideal function: returns the actual minimax value of the position

› In practice: typically weighted linear sum of features:

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

› e.g. $f_1(s)$ = (num white queens – num black queens), etc.

Evaluation functions

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

Alpha-Beta and Evaluation functions

- Alpha-Beta: amount of pruning depends on expansion ordering
  - Evaluation function can provide guidance to expand most promising nodes first (which later makes it more likely there is already a good alternative on the path to the root)
    - (somewhat similar to role of A* heuristic, CSPs filtering)

- Alpha-Beta: (similar for roles of min-max swapped)
  - Value at a min-node will only keep going down
  - Once value of min-node lower than better option for max along path to root, can prune
  - Hence: IF evaluation function provides upper-bound on value at min-node, and upper-bound already lower than better option for max along path to root THEN can prune

Monte Carlo Tree Search (MCTS)

- Suitable for more complex games – uses randomness instead of imperfect evaluation functions
- Play out numerous trees with random moves to estimate utility

- Layman intro
  - https://towardsdatascience.com/game-ais-with-minimax-and-monte-carlo-tree-search-af2a177361b0
- Very good tutorial
  - https://netman.aiops.org/~peidan/ANM2018/3.MachineLearningBasics/LectureCoverage/9.MCTS_tutorial.pdf

Monte Carlo Tree Search (MCTS)

- **Selection**
  Selecting good child nodes, starting from the root node R, that represent states leading to better overall outcome (win).
- **Expansion**
  If *L* is a not a terminal node (i.e. it does not end the game), then create one or more child nodes and select one (*C*).
- **Simulation (rollout)**
  Run a simulated playout from *C* until a result is achieved.
- **Backpropagation**
  Update the current move sequence with the simulation result.

Monte Carlo Tree Search (MCTS)

- AlphaGo documentary
- https://www.youtube.com/watch?v=WXuK6gekU1Y

› Alpha Go comprises of:

- **Monte Carlo Tree Search:** AI chooses its next move using MCTS
- Residual **CNNs (Convolutional Neural Networks):** AI assesses new positions using these networks
- **Reinforcement learning:** Trains the AI by using the current best agent to play against itself

AlphaGo

› https://deepmind.google/technologies/alphago/

“We created AlphaGo, an AI system that combines deep neural networks with advanced search algorithms.

One neural network — known as the “policy network” — selects the next move to play. The other neural network — the “value network” — predicts the winner of the game.

Initially, we introduced AlphaGo to numerous amateur games of Go so the system could learn how humans play the game. Then we instructed AlphaGo to play against different versions of itself thousands of times, each time learning from its mistakes — a method known as reinforcement learning. Over time, AlphaGo improved and became a better player.”

AlphaZero

› https://deepmind.google/technologies/alphazero-and-muzero/

“Unlike AlphaGo, which learned to play Go by analyzing millions of moves from amateur games, AlphaZero’s neural network was only given the rules of each game.

It then learned each game by playing itself millions of times. Through a process of trial and error, called reinforcement learning, the system learned to select the most promising moves and boost its chances of winning.

AlphaZero mastered chess in just 9 hours. Shogi in 12 hours. And Go in 13 days. In each game, it learned to play with a unique and creative style.

In chess, for example, the model developed a highly dynamic and “unconventional” playing style, which has since been studied at the highest levels of the game.”

Grandmaster-Level Chess Without Search

https://arxiv.org/abs/2402.04494

“Unlike traditional chess engines that rely on complex heuristics, explicit search, or a combination of both, we train a 270M parameter transformer model with supervised learning on a dataset of 10 million chess games. We annotate each board in the dataset with action-values provided by the powerful Stockfish 16 engine, leading to roughly 15 billion data points. Our largest model reaches a Lichess blitz Elo of 2895 against humans, and successfully solves a series of challenging chess puzzles, without any domain-specific tweaks or explicit search algorithms. We also show that our model outperforms AlphaZero's policy and value networks (without MCTS) and GPT-3.5-turbo-instruct.”