PURDUE CS47100

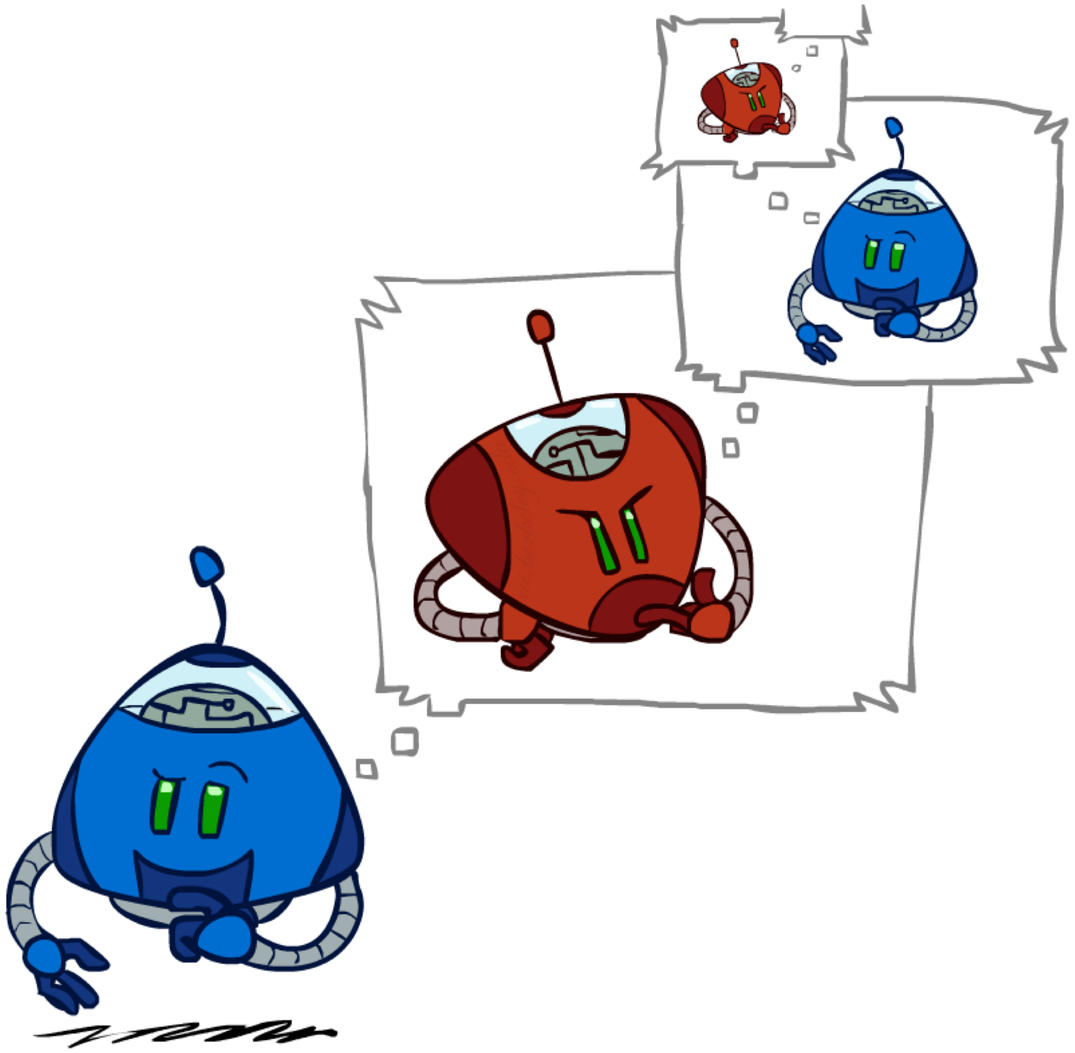
INTRODUCTION TO AI

RECAP: LOCAL SEARCH & ADVERSARIAL SEARCH

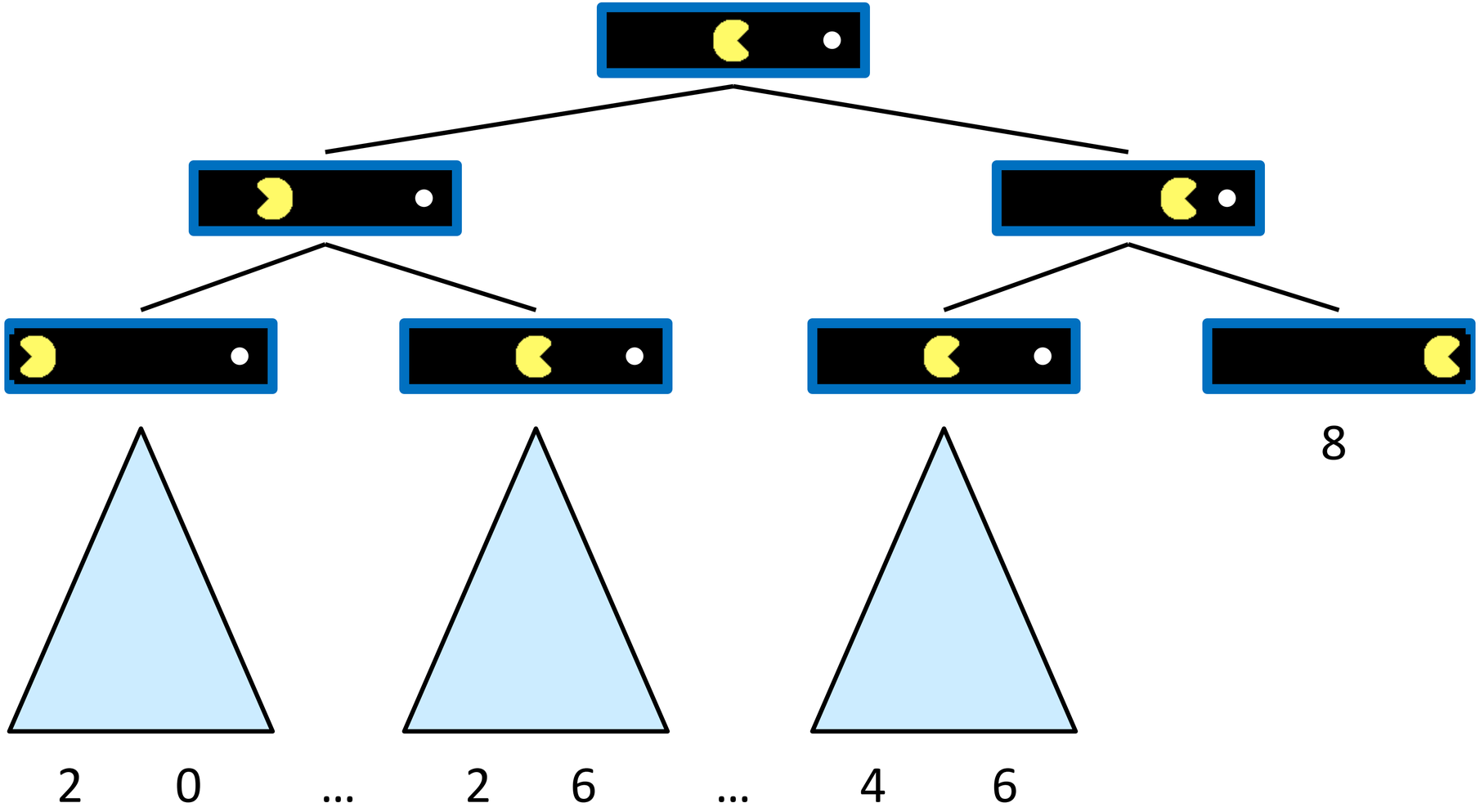
- Local search
  - Hill climbing, simulated annealing, beam search, genetic algorithm

- Adversarial search
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: S x A $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: S x P $\rightarrow$ R
  - Solution for a player is a **policy**: S $\rightarrow$ A

ADVERSARIAL SEARCH



SINGLE-AGENT TREES

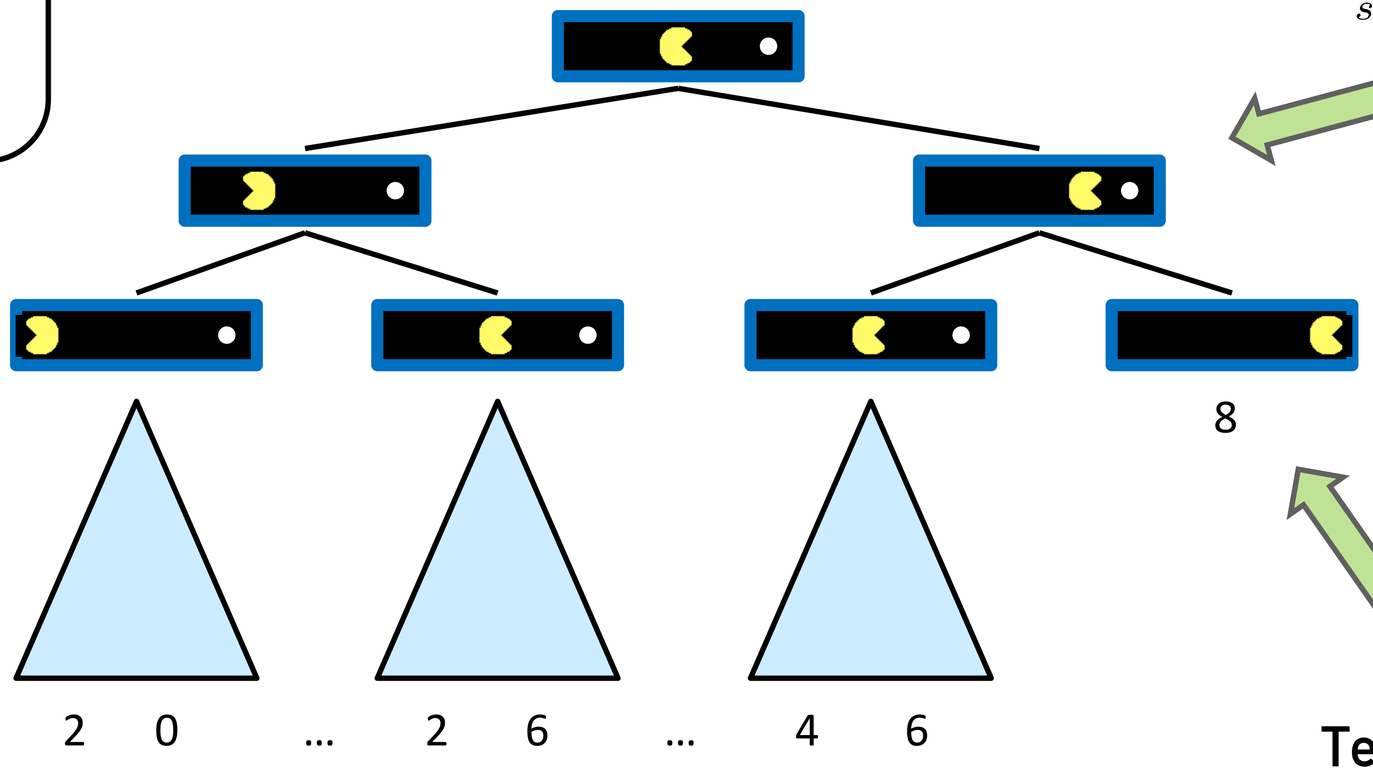


VALUE OF A STATE

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

Non-Terminal States:

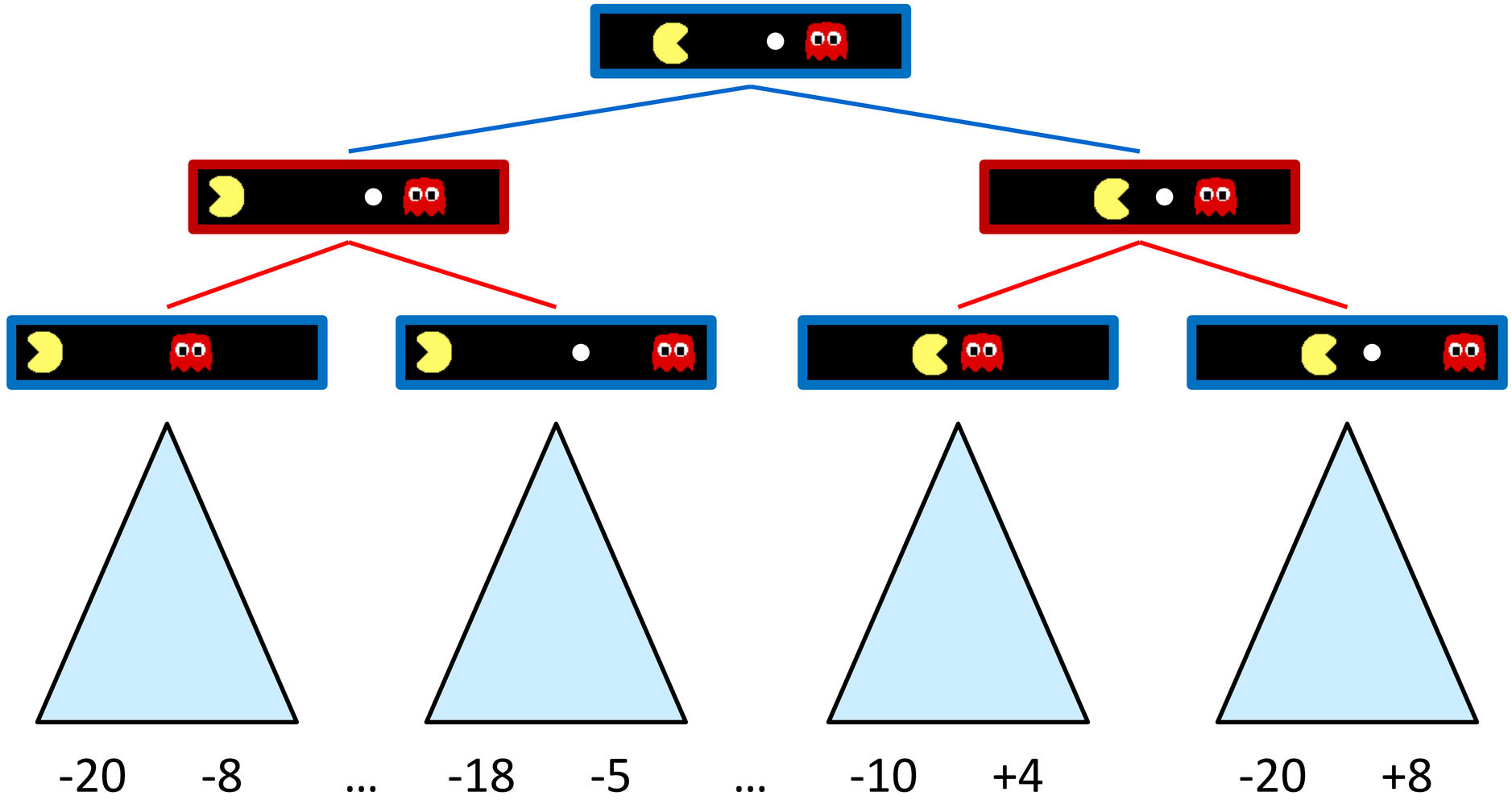
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

ADVERSARIAL GAME TREES



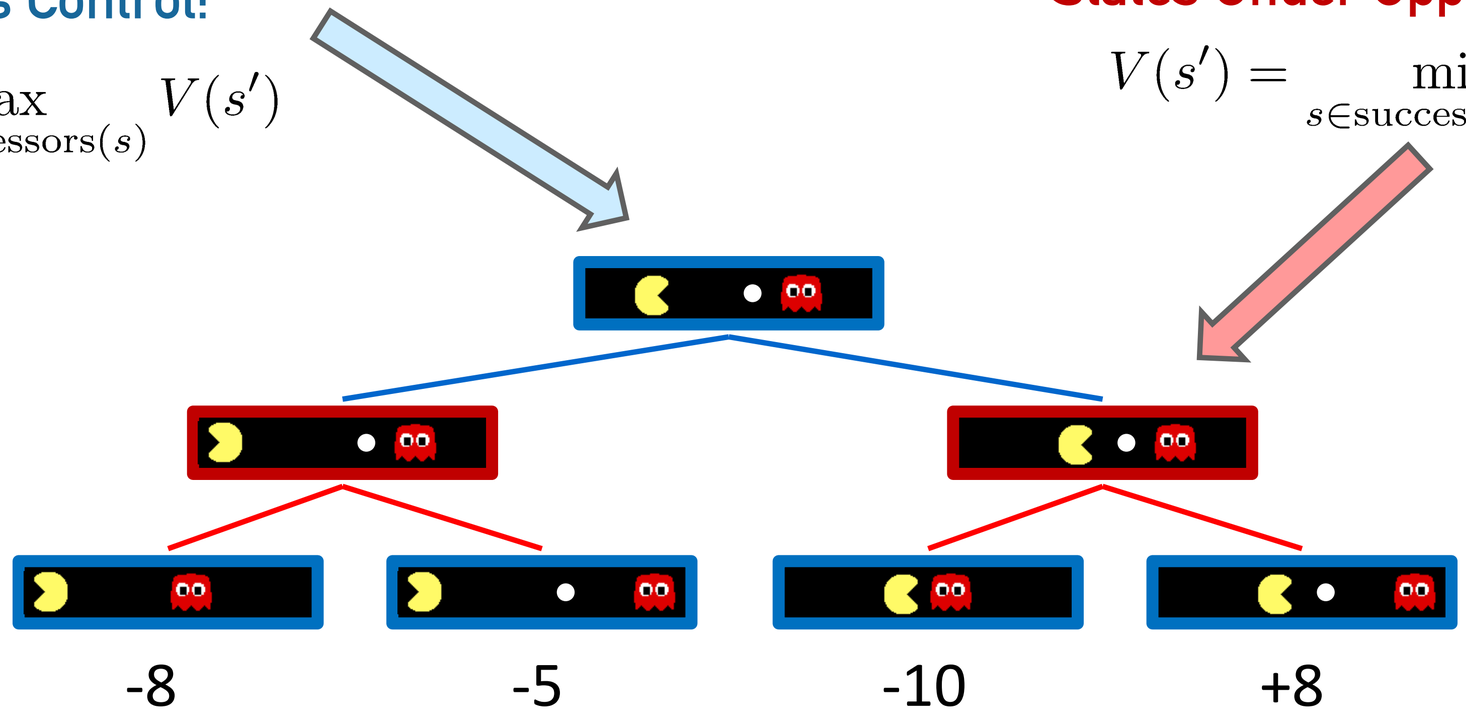
MINIMAX VALUES

States Under Agent's Control:

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

States Under Opponent's Control:

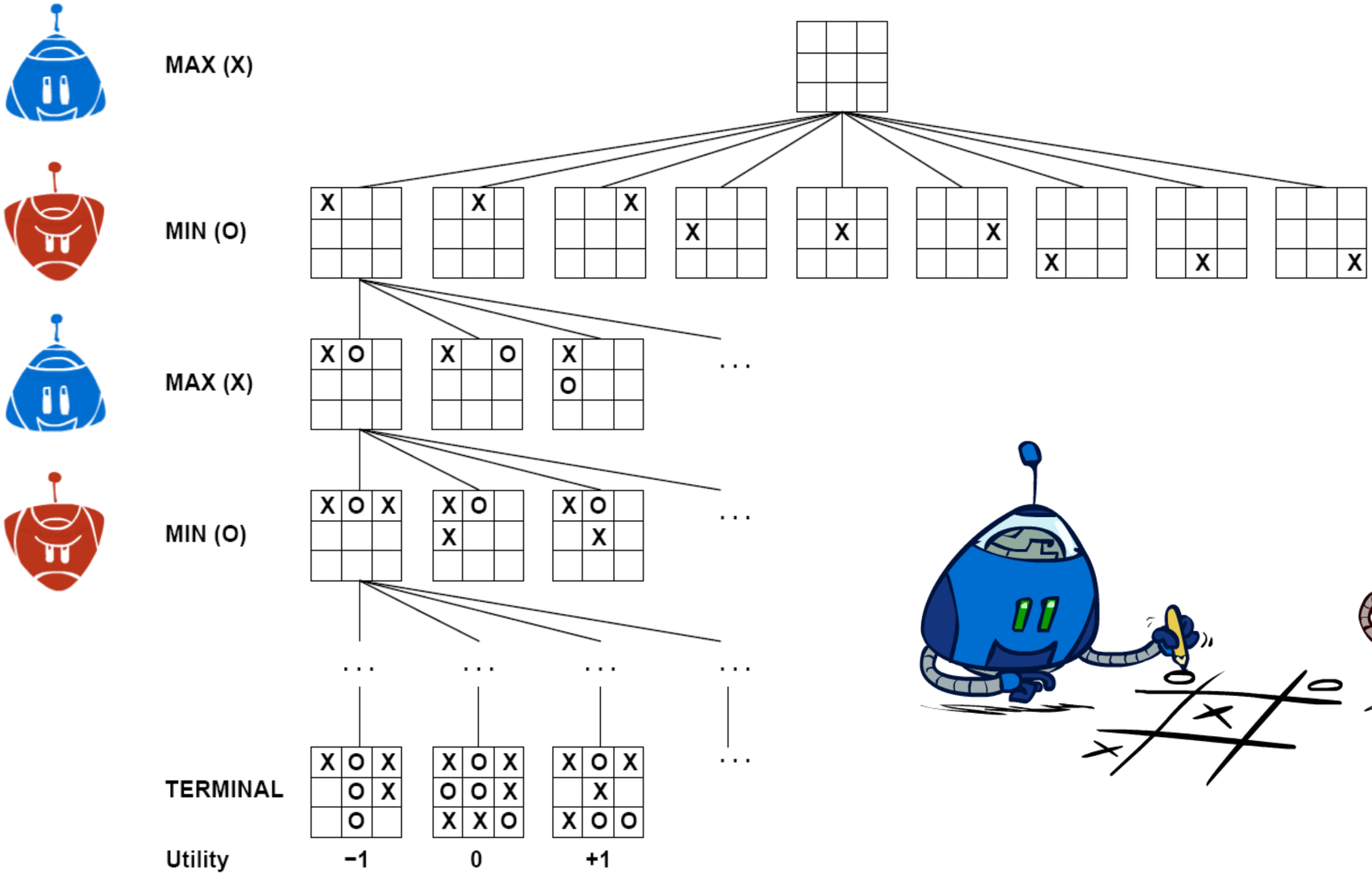
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

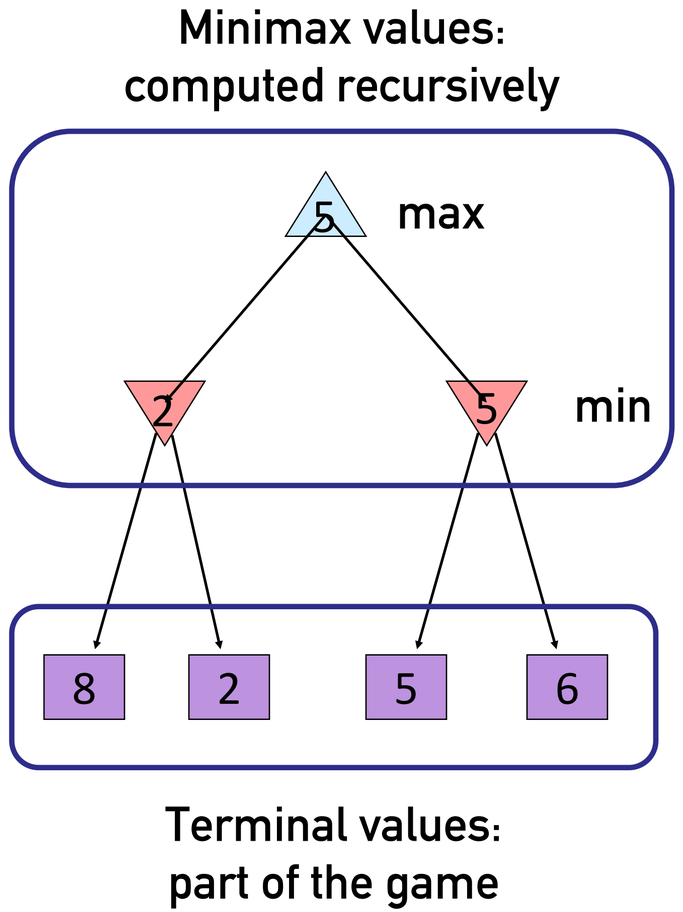
TIC-TAC-TOE GAME TREE



ADVERSARIAL SEARCH (MINIMAX)

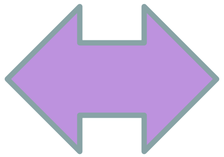
- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- 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 IMPLEMENTATION

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



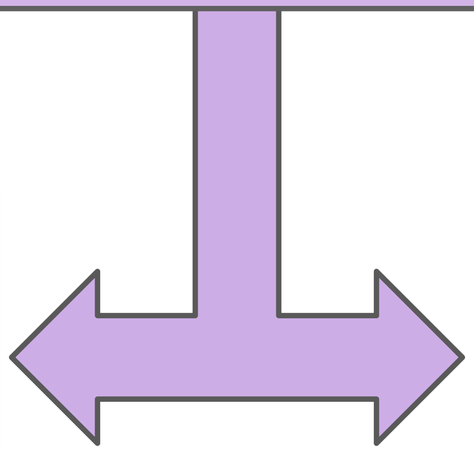
```
def min-value(state):
  initialize v = +∞
  for each successor of state:
    v = min(v, max-value(successor))
  return v
```

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

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

MINIMAX IMPLEMENTATION (DISPATCH)

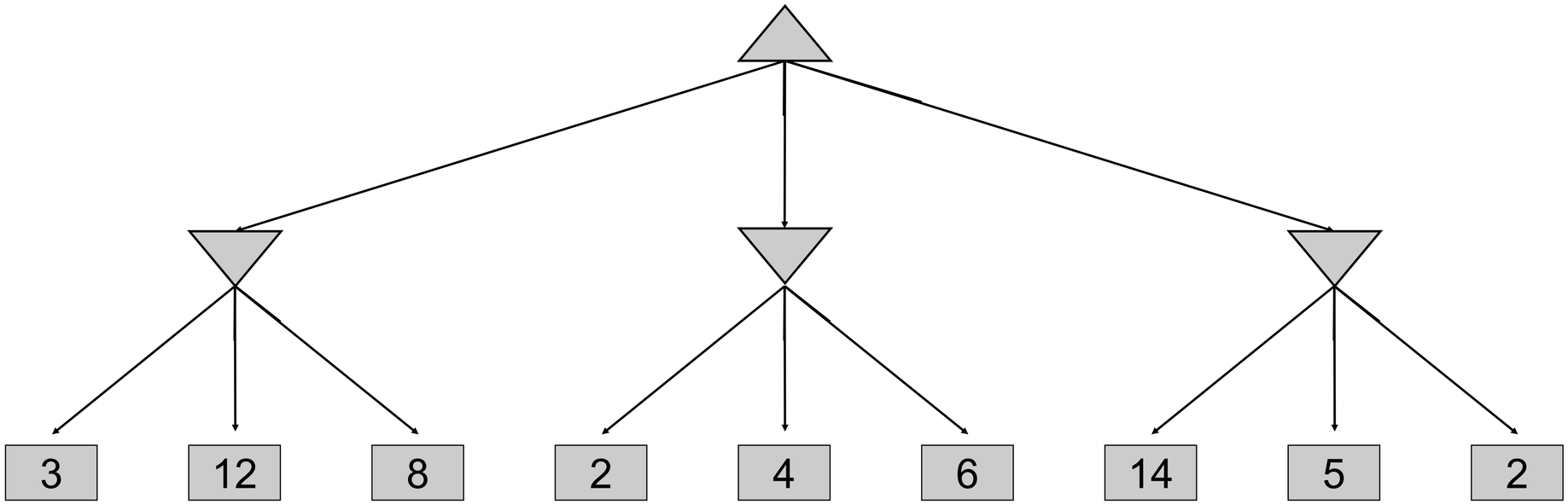
```
def value(state):
  if the state is a terminal state: return the state's
  utility
  if the state's agent is MAX: return max-value(state)
  if the state's agent is MIN: return min-value(state)
```



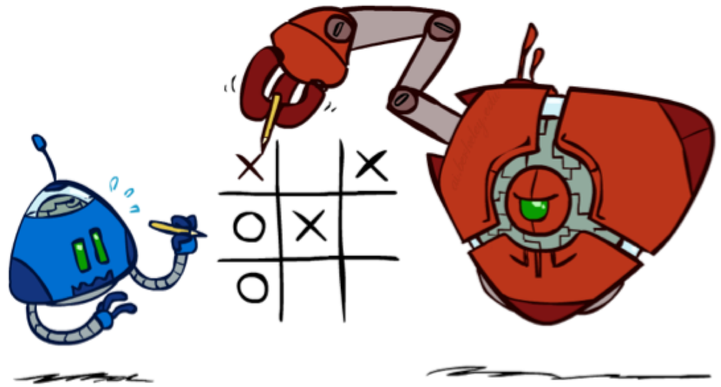
```
def max-value(state):
  initialize v = -∞
  for each successor of state:
    v = max(v, value(successor))
  return v
```

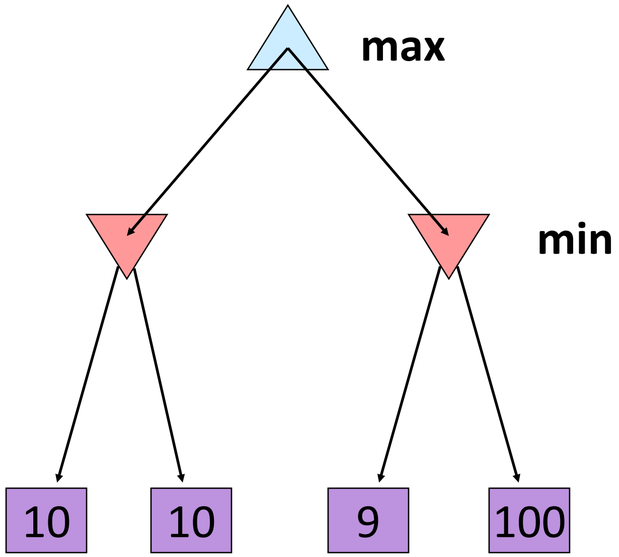
```
def min-value(state):
  initialize v = +∞
  for each successor of state:
    v = min(v, value(successor))
  return v
```

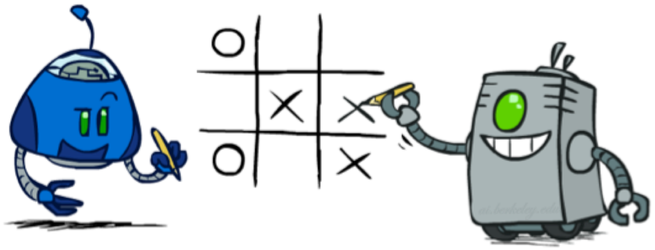
MINIMAX EXAMPLE



MINIMAX PROPERTIES





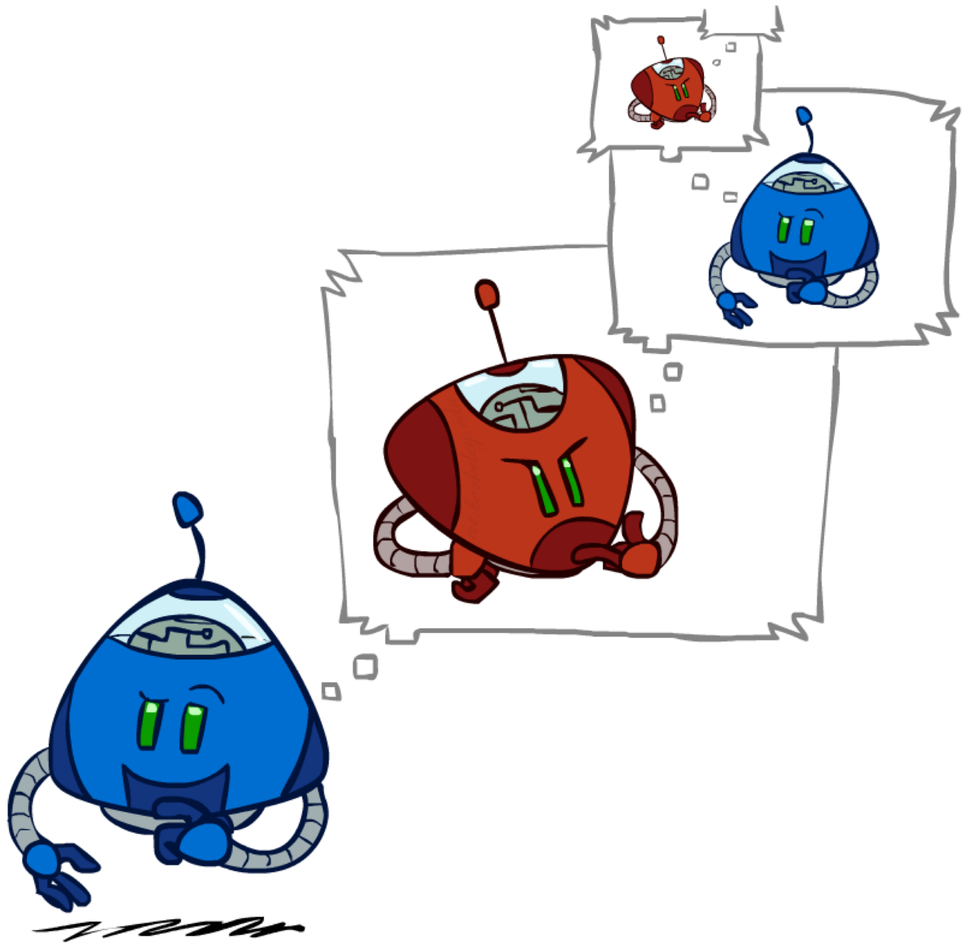


Optimal against a rational player. Otherwise?

MINIMAX EFFICIENCY

- Efficiency of minimax search
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: O(bm)

- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible



GAME TREE SIZES

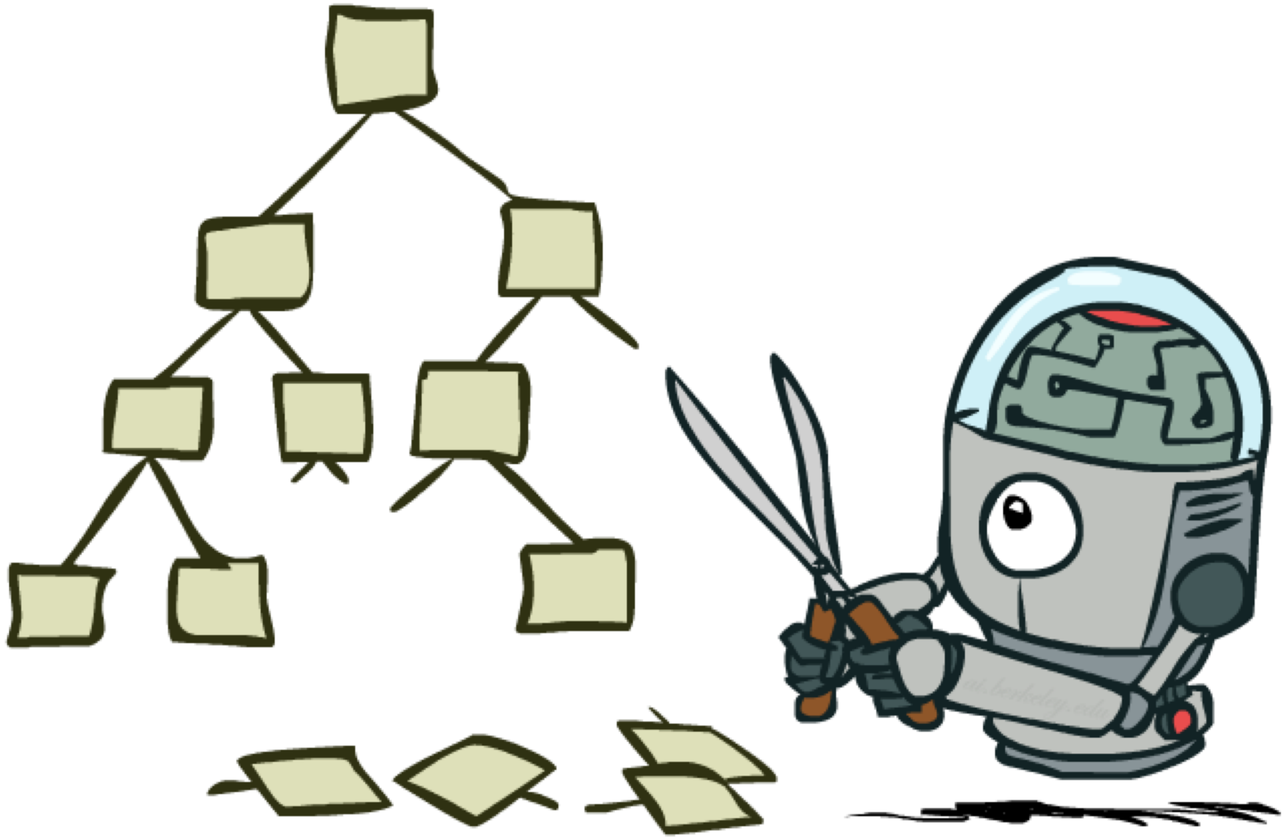
- Tic-tac-toe: 10^5
- Checkers: 10^{31}
- Chess: 10^{123}
- Backgammon: 10^{144}
- Go: 10^{360}

Assume that a computer can evaluate 1 million board configurations per second.

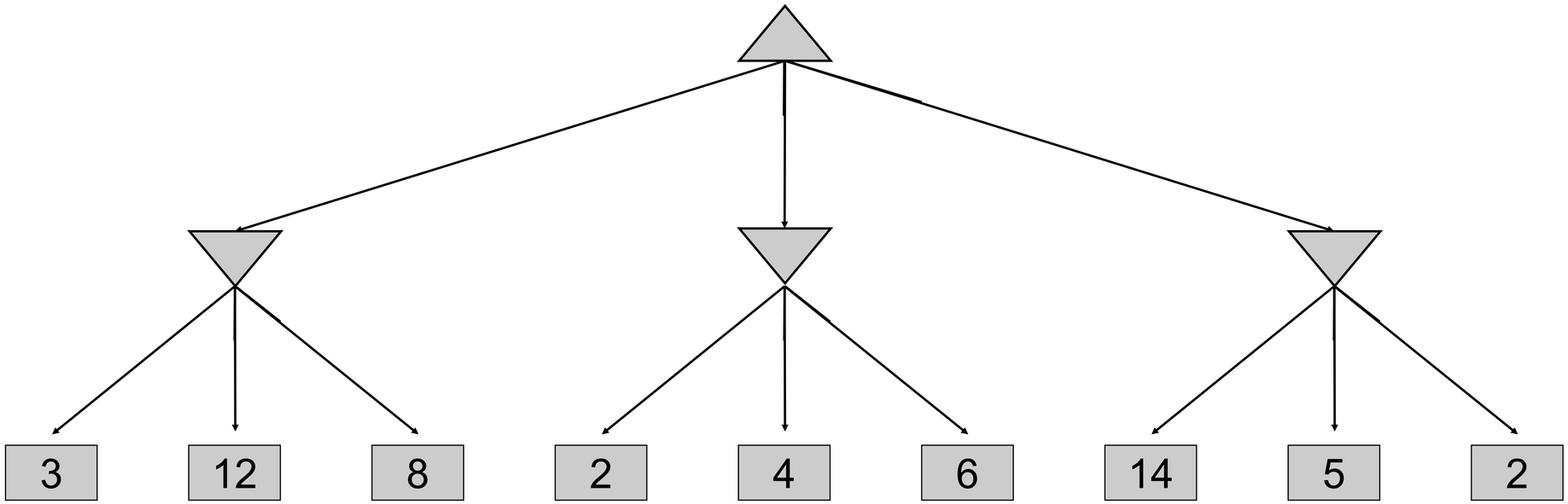
Then it would take **0.1 seconds to search** the entire tic-tac-toe game tree but it would still take **10^{18} years** to search the full checkers tree.

Do we need to explore the whole tree?

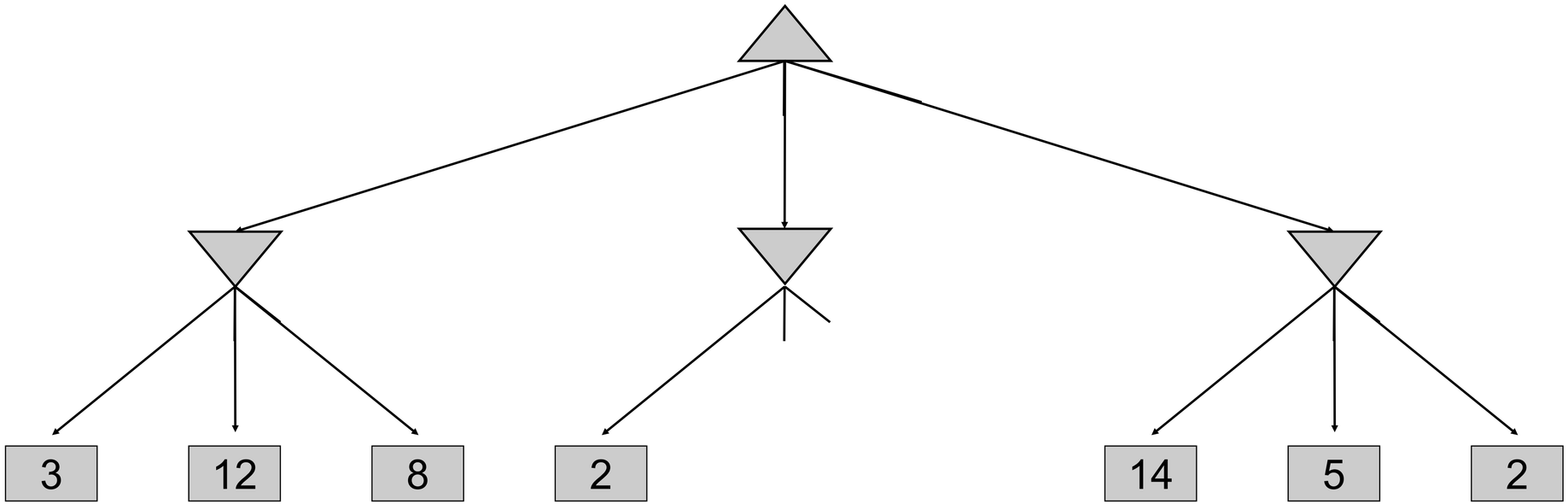
GAME TREE PRUNING



MINIMAX EXAMPLE

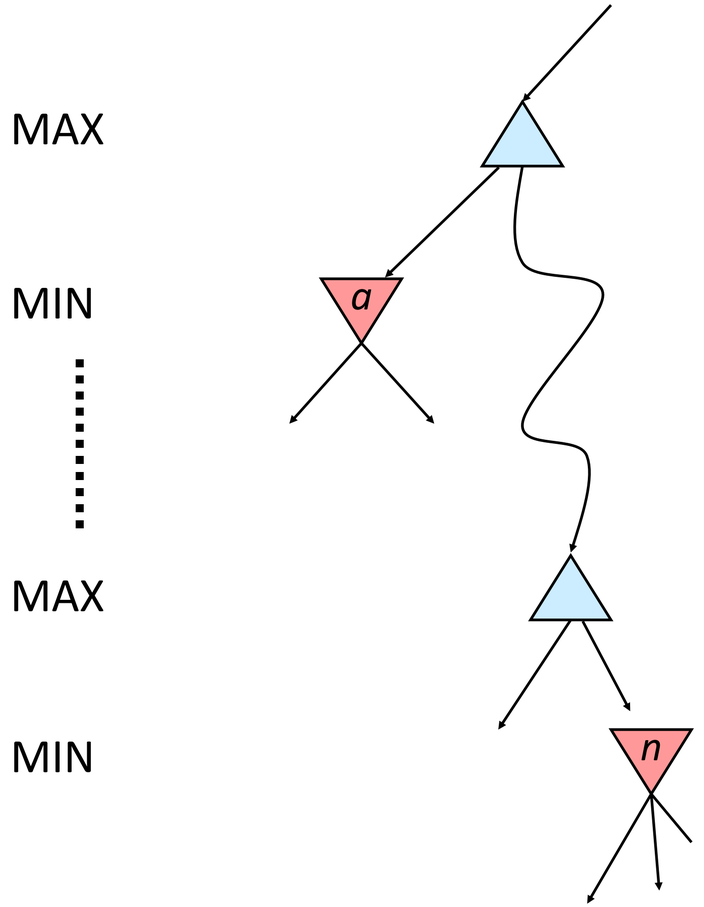


MINIMAX PRUNING



ALPHA-BETA PRUNING

- General configuration (MIN version)
  - When computing the MIN-VALUE at some node *n*
  - We loop over *n*'s children
  - *n*'s estimate of the children's min is dropping
  - Who cares about *n*'s value? MAX
  - Let α be the best value that MAX can get at any choice point along the current path from the root so far
  - If *n* becomes worse than α, 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 PRUNING