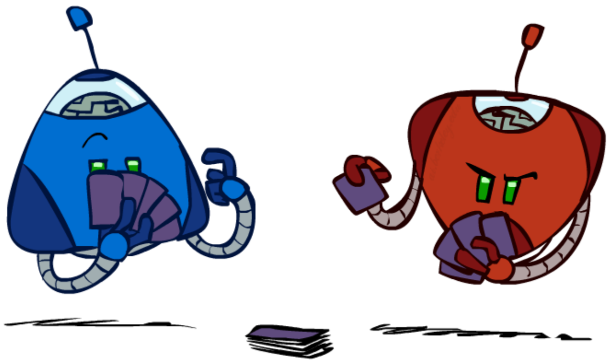
复旦大学大数据学院
School of Data Science, Fudan University
魏忠钰

Adversarial Search

April 4th, 2018

Types of Games

- Many different kinds of games!



- Axes:
  - **Deterministic** or stochastic?
  - One, **two**, or more players?
  - **Zero sum**?
  - **Perfect information** (can you see the state)?

- Want algorithms for calculating a strategy (policy) which recommends a move from each state

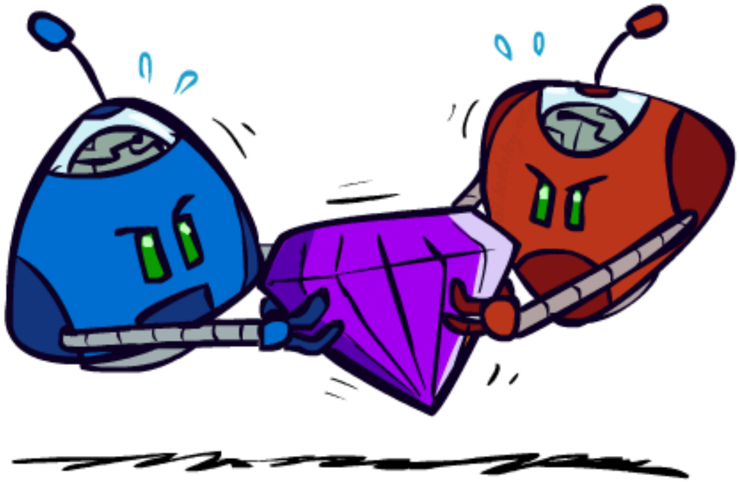
Deterministic Games

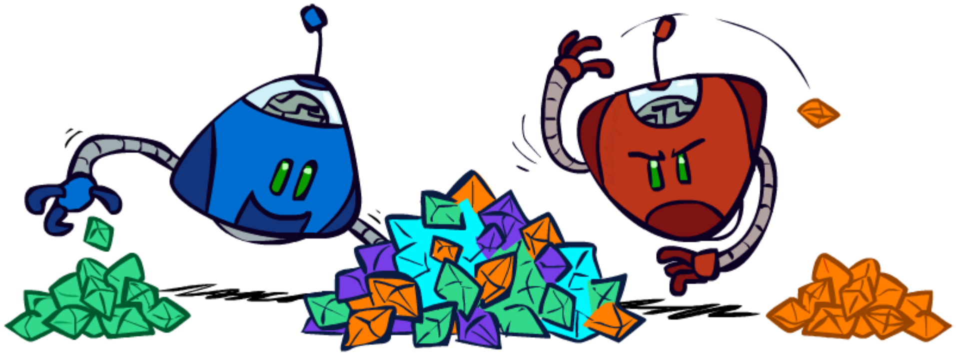
- Many possible formalizations, one is:
  - S: states (start at s_0)
  - Player (s): the player has the move in this state
  - Actions (s): A set of legal moves in a state
  - Results (s, a): A transition model, return the results of a move
  - Terminal Test (s): {true, false} if s is the terminal state
  - Terminal Utilities (s, p): A utility function gives the final numeric value of a game

- Solution for a player is a policy: $S \rightarrow A$ (a set of actions)

- What is the policy for depth-first-search?

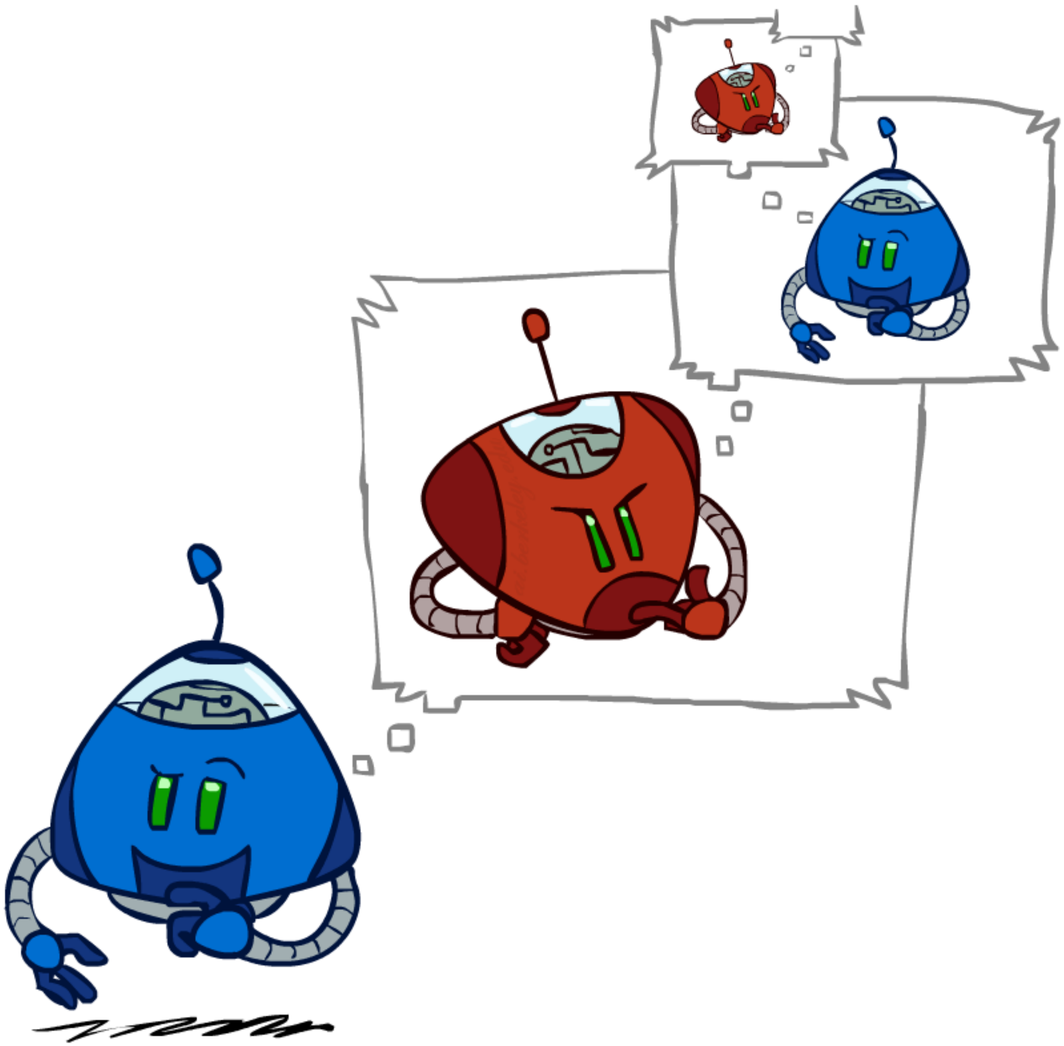
Zero-Sum Games



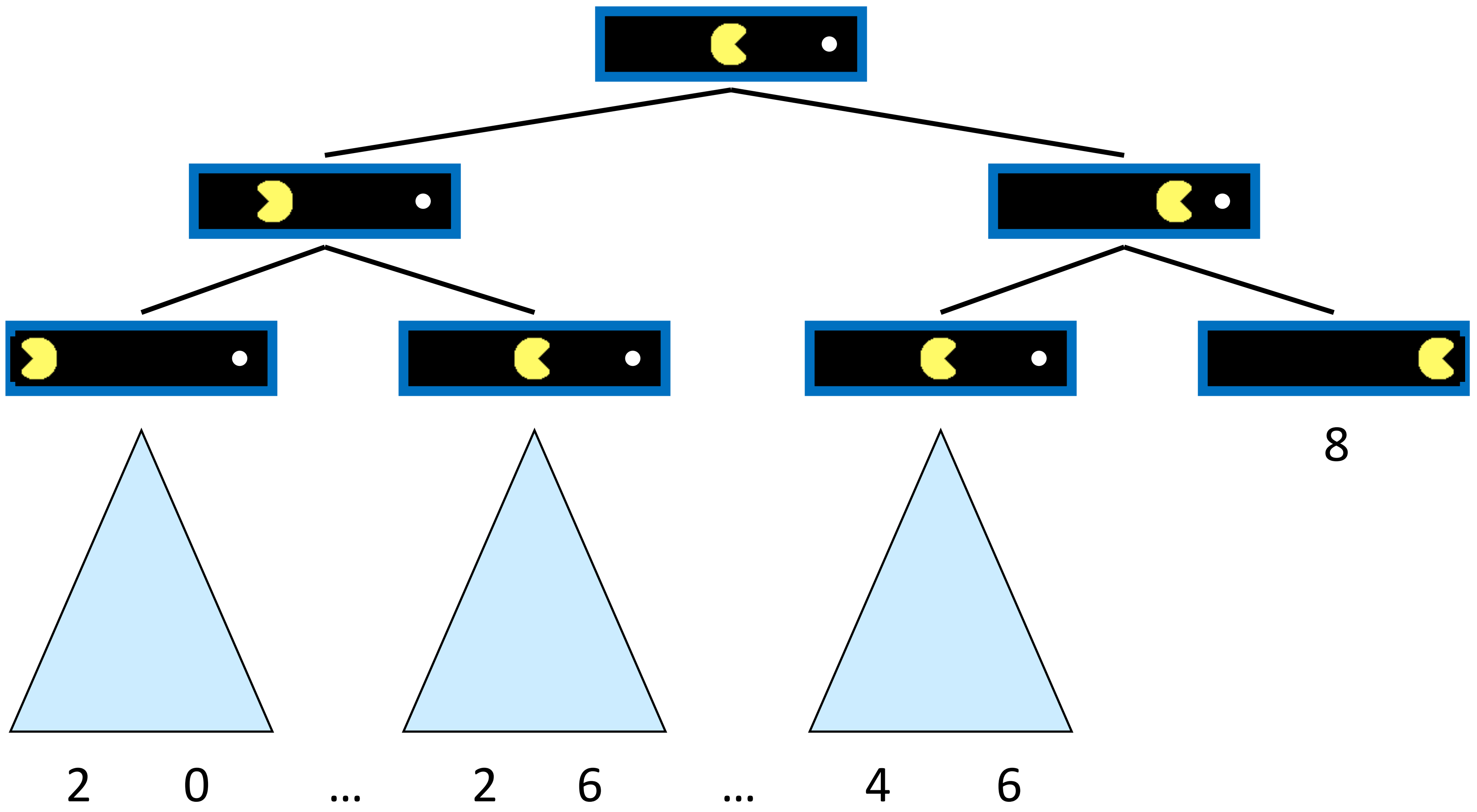


- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible

Adversarial Search



Single-Agent Trees

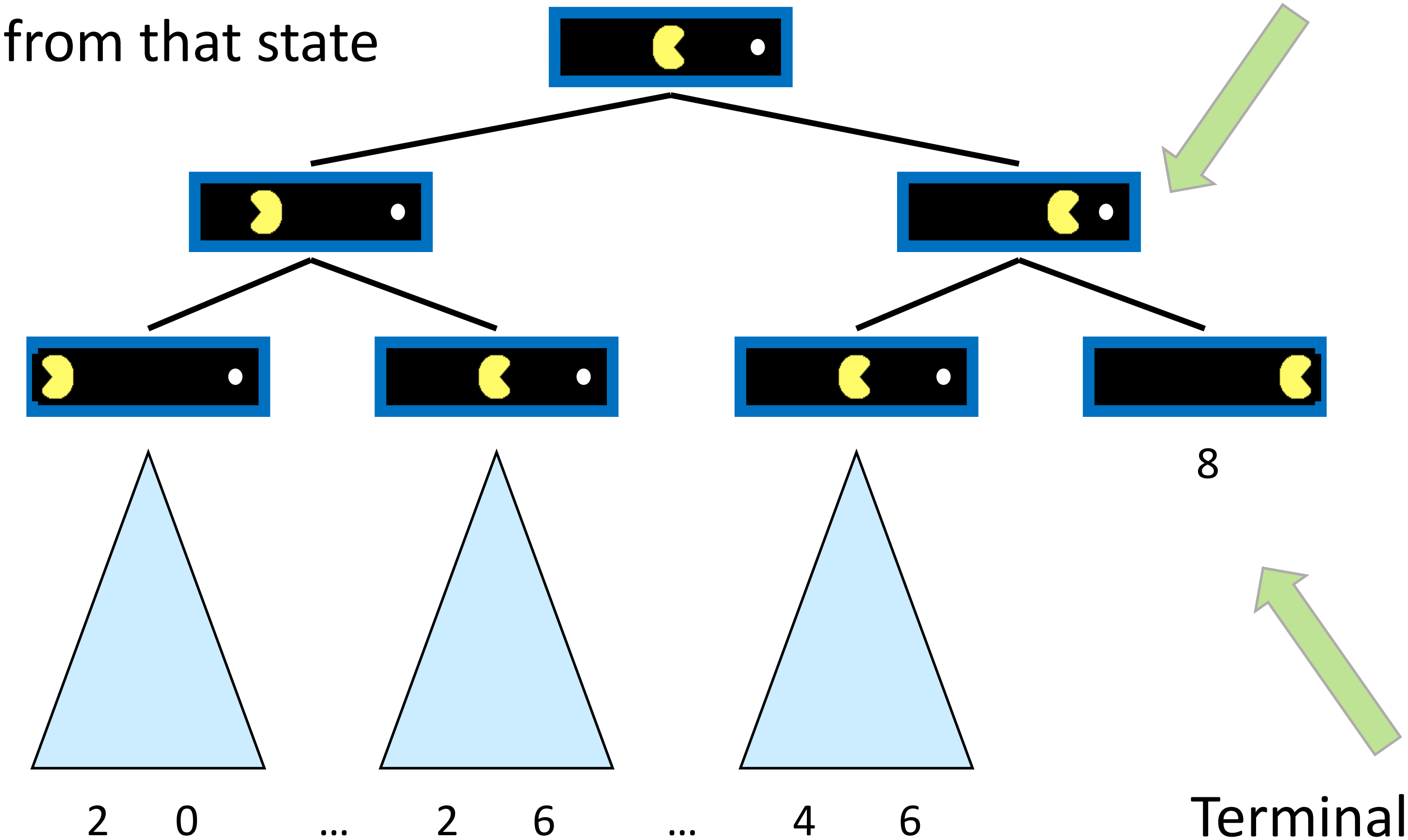


Value of a State

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

Non-Terminal States:

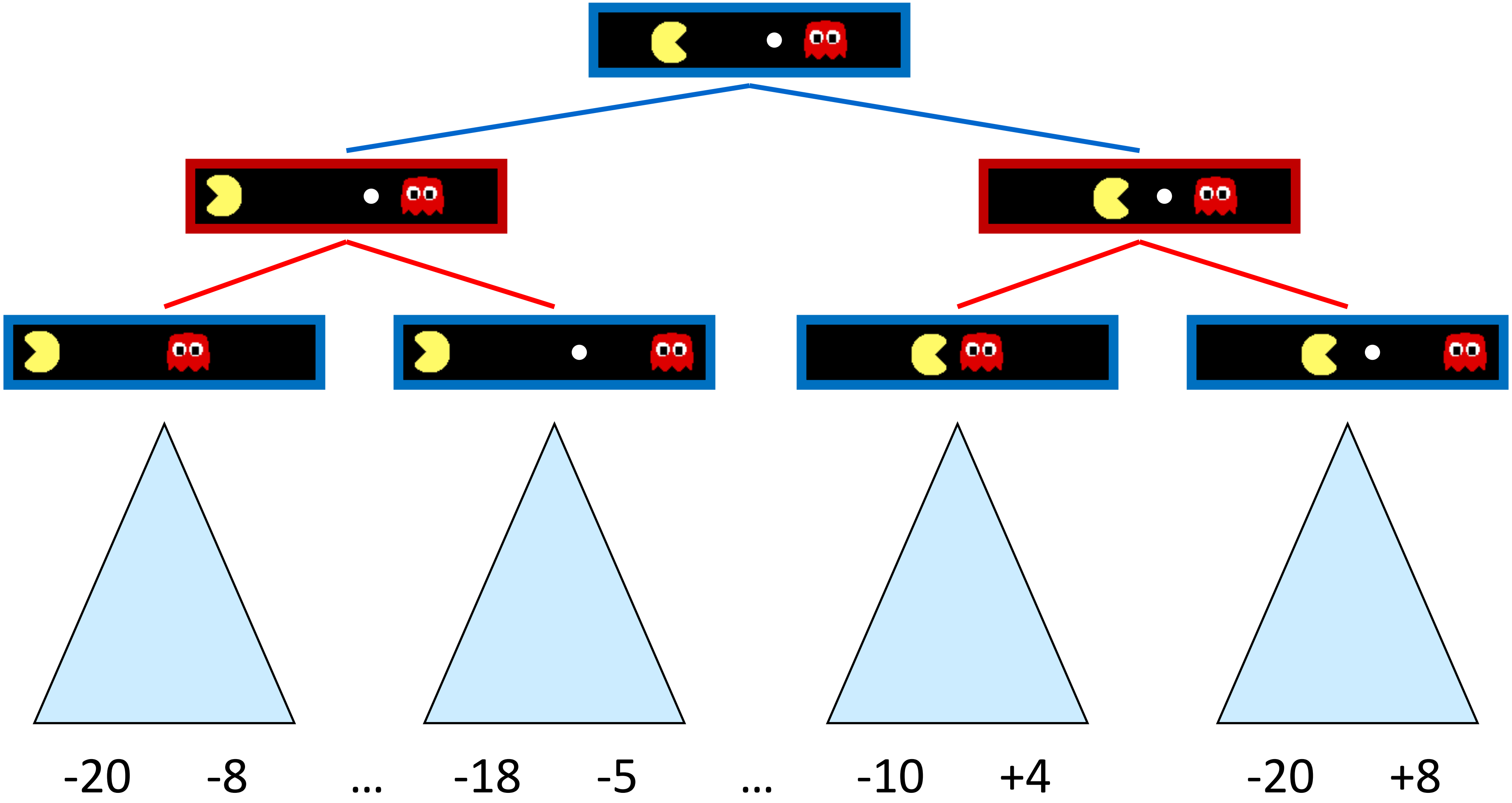
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial Game Trees



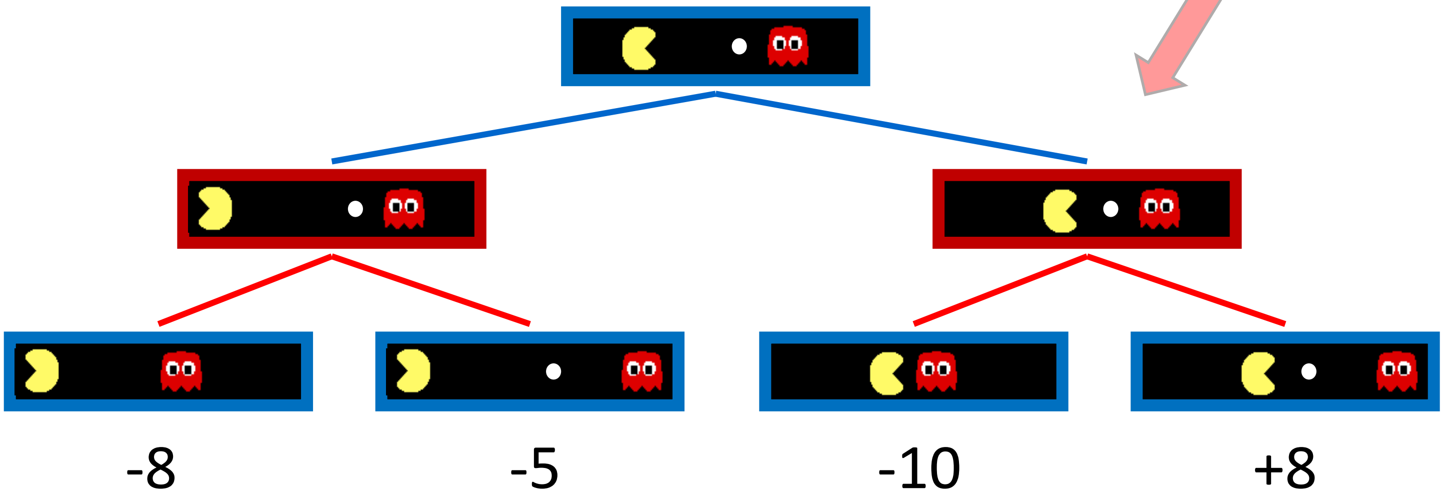
Minimax Values

States Under Agent’s Control:

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

States Under Opponent’s Control:

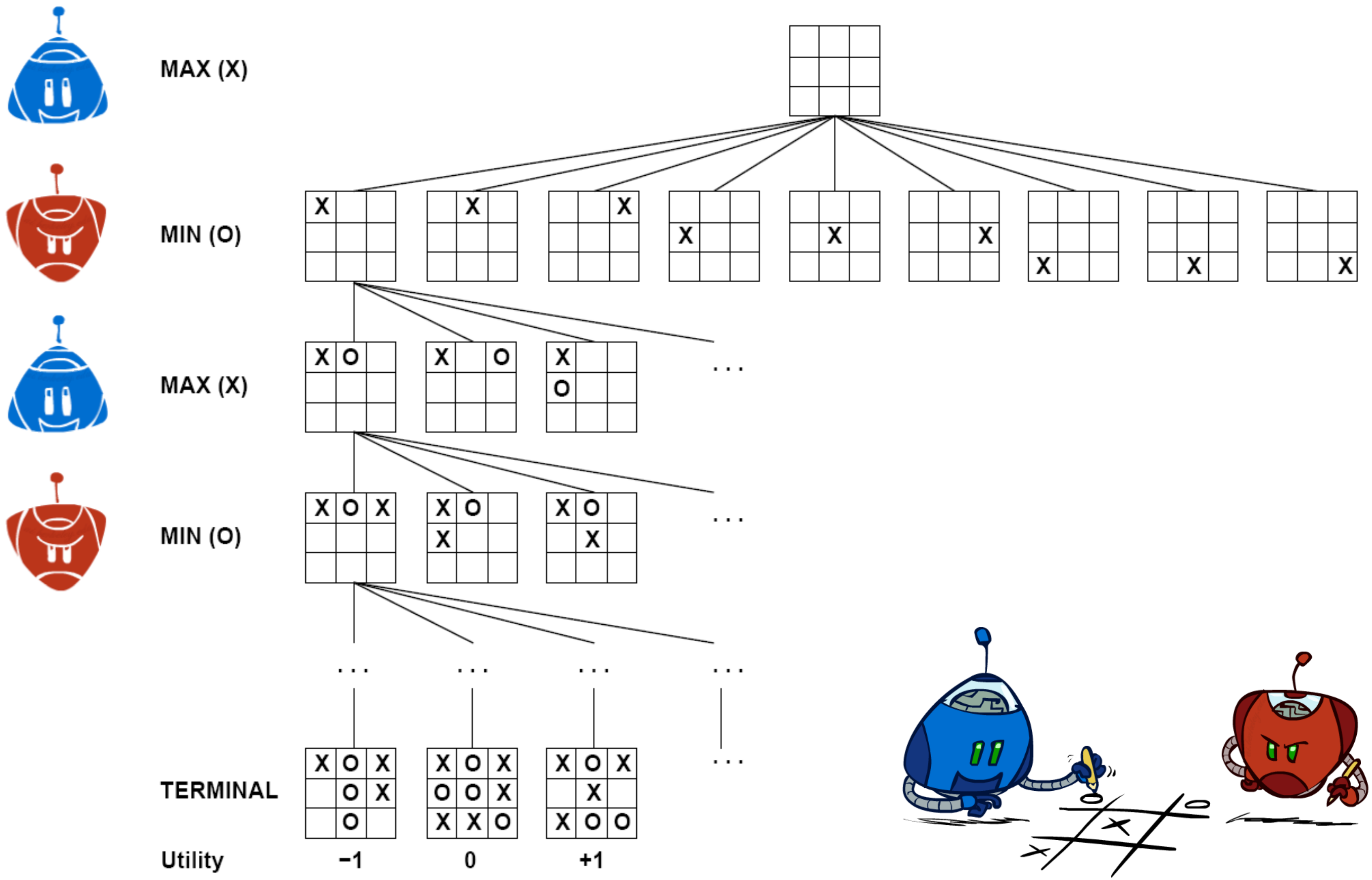
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

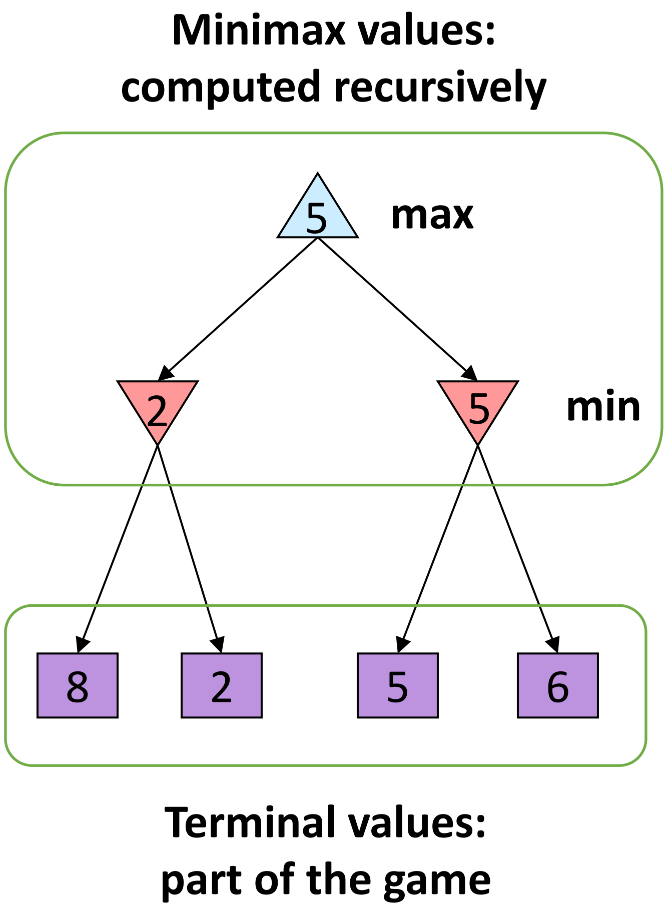
Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess
  - 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
```

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

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

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

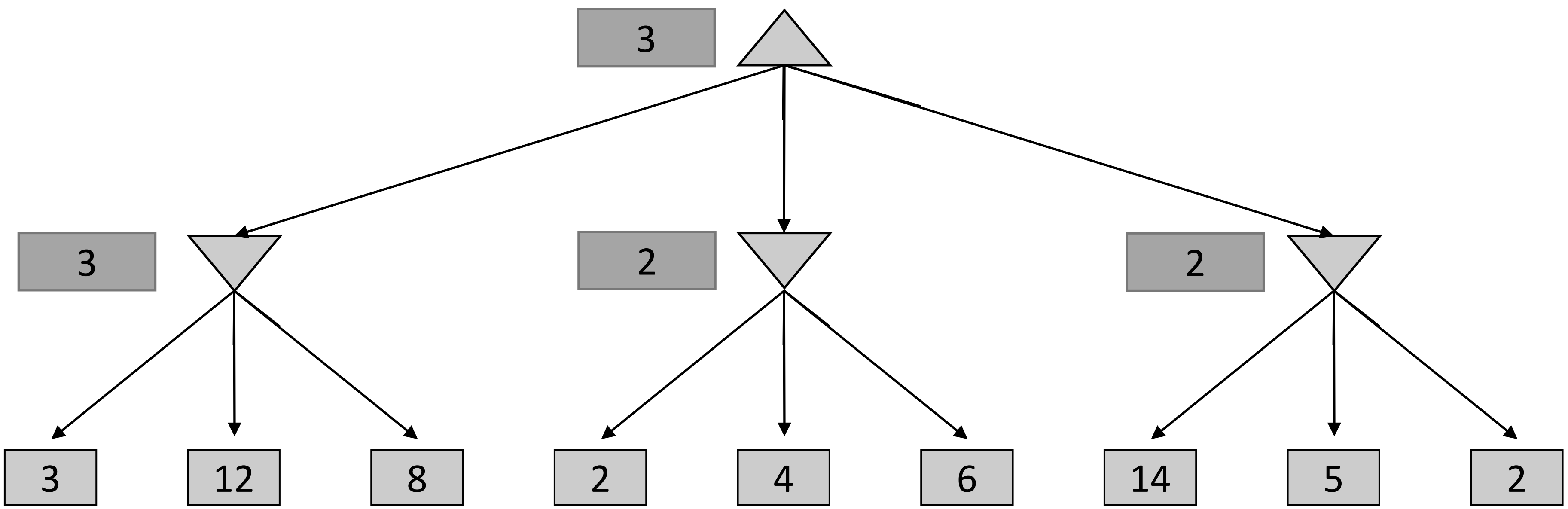
Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the 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
```

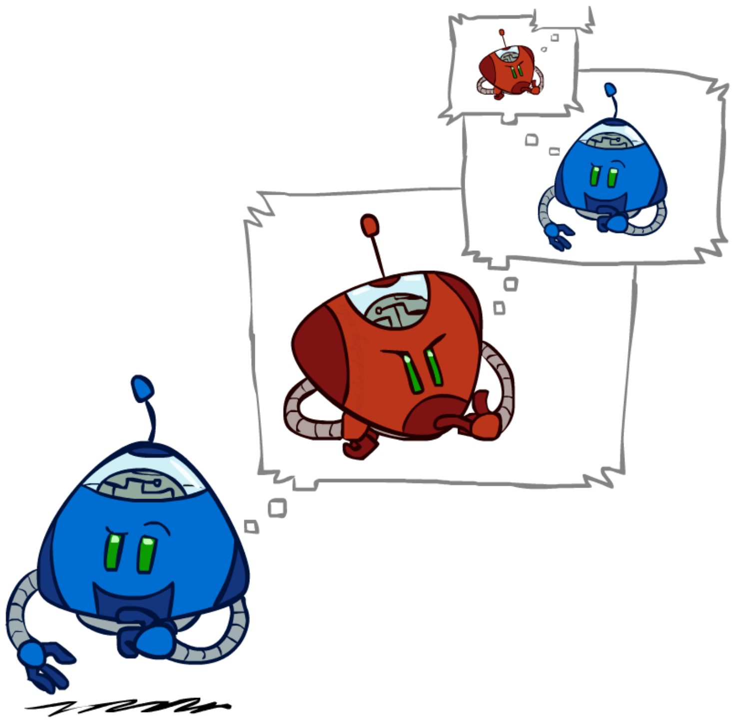
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v,
          value(successor))
    return v
```

Minimax Example



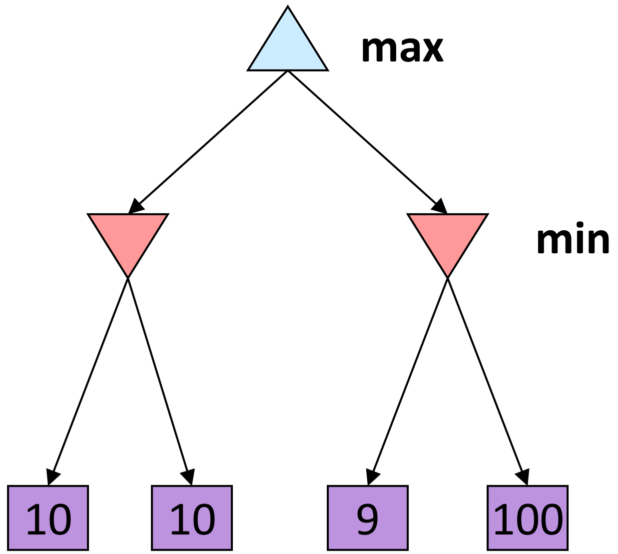
Minimax Efficiency

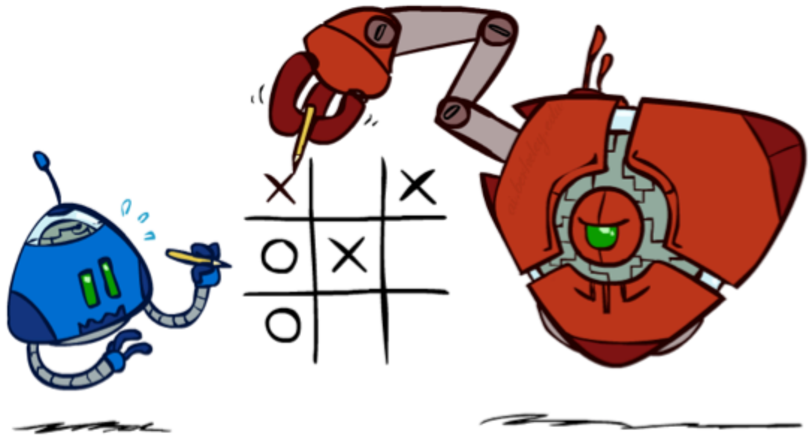
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$



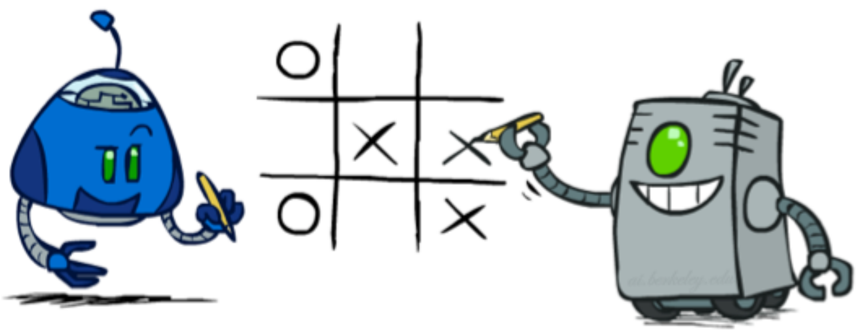
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

Minimax Properties





Optimal against a perfect player. Otherwise?



Resource Limits

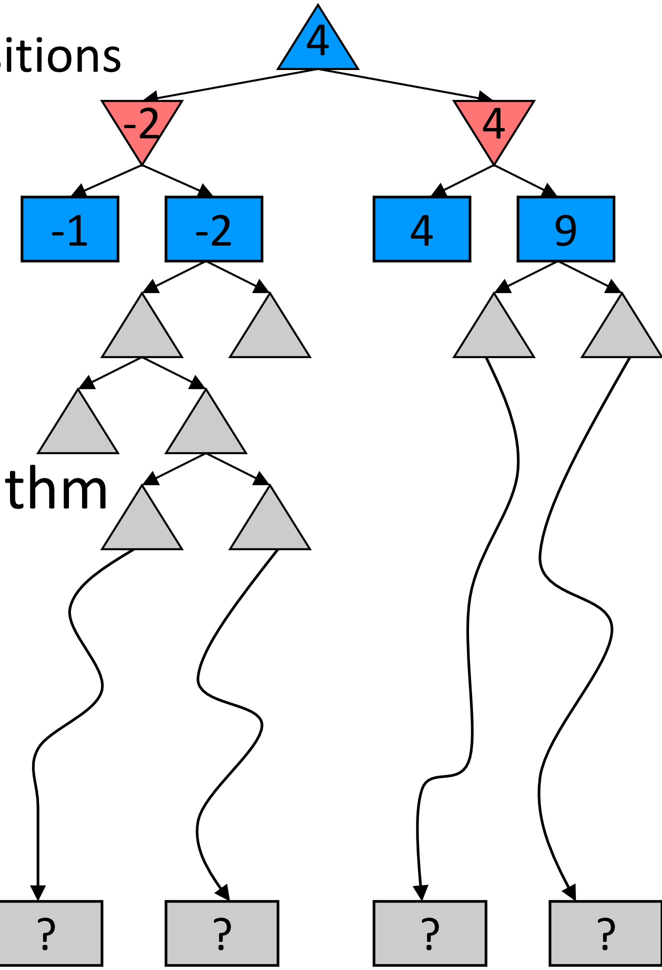
- Problem: In realistic games, cannot search to leaves!

- Solution: Depth-limited search
  - Search only to a limited depth in the tree
  - Need an evaluation function for non-terminal positions

- Guarantee of optimal play is gone

- More steps forward makes a BIG difference

- Use iterative deepening for an anytime algorithm



Issues about Evaluation Functions

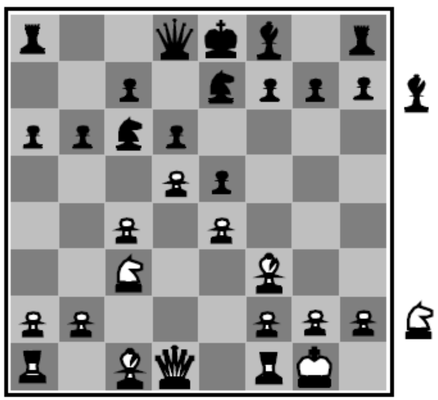
- Evaluation functions are always imperfect

- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters

- **It takes time to compute the evaluation function.**
  - An important example of the tradeoff between complexity of features and complexity of computation

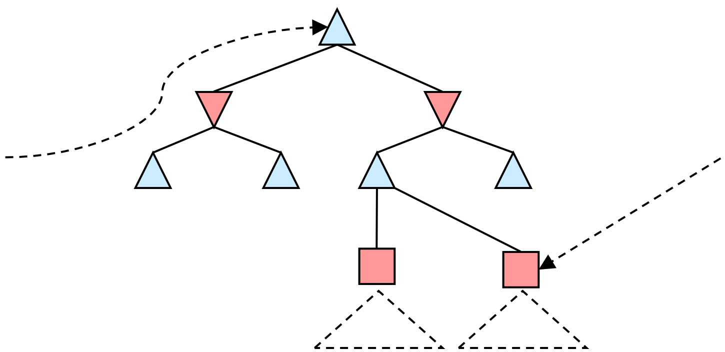
Evaluation Functions

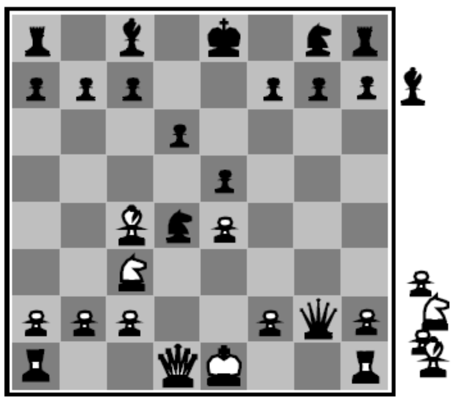
- Evaluation functions score non-terminals in depth-limited search



Black to move

White slightly better





White to move

Black winning

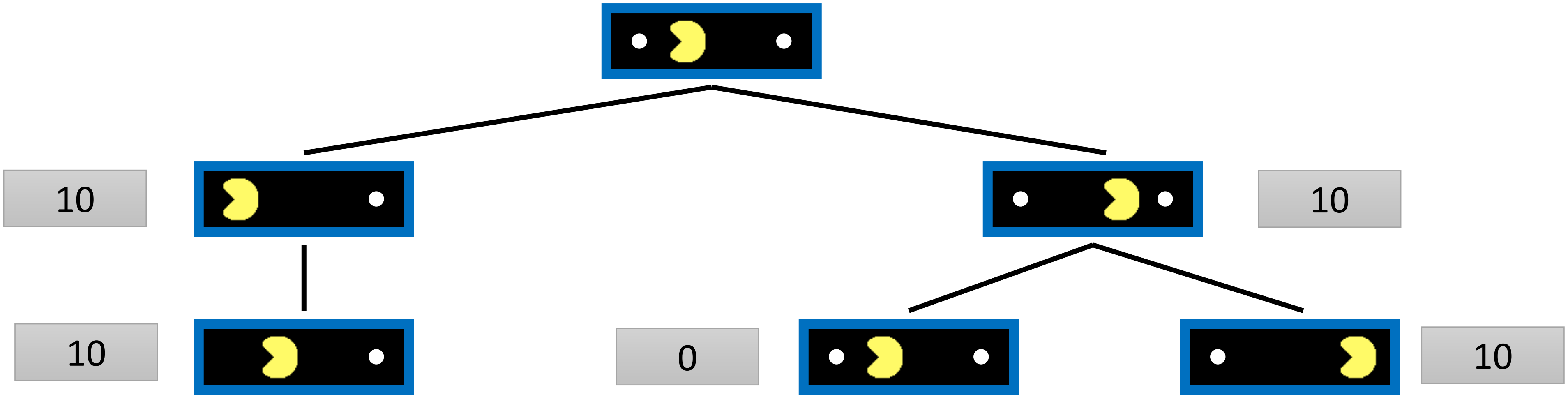
- Ideal function: returns the actual minimax value of the position

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

Eval(s) = material + mobility + king-safety + center-control

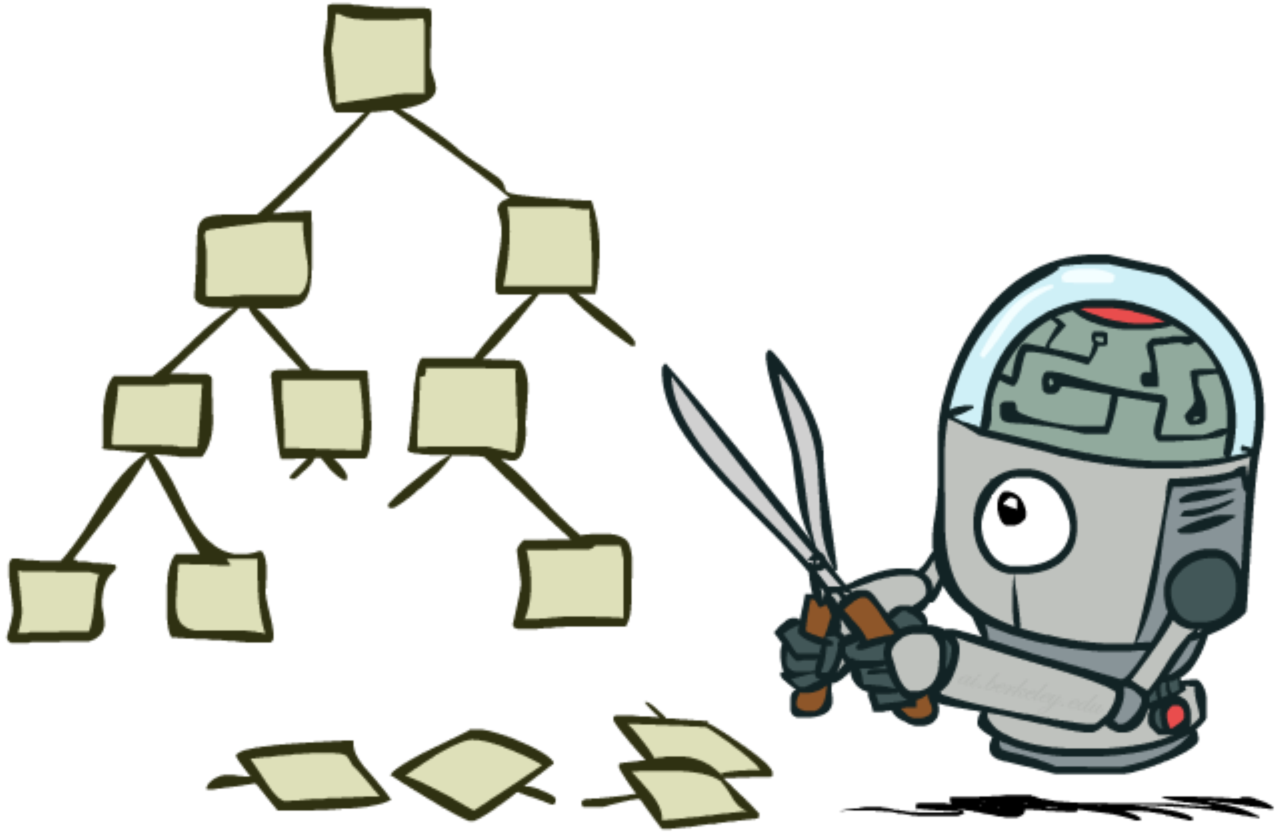
$$\begin{aligned}\text{material} &= 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + 3(B - B' + N - N') + 1(P - P') \\ \text{mobility} &= 0.1(\text{num-legal-moves} - \text{num-legal-moves}')\end{aligned}$$

Example for Evaluation Function Design

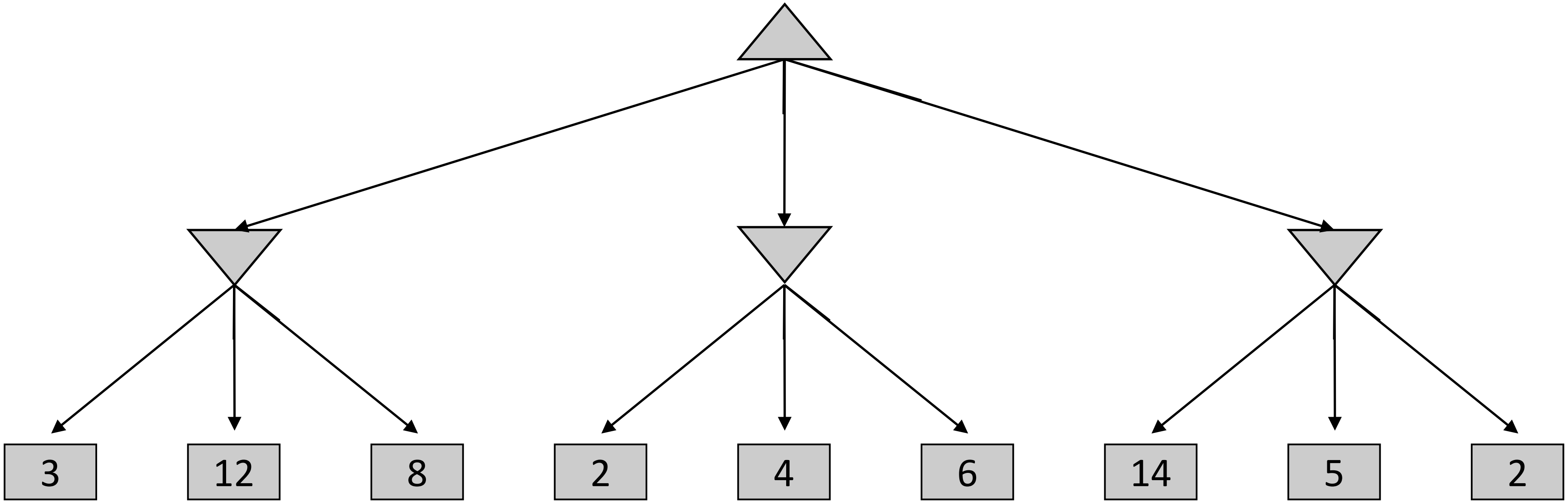


- A danger of re-planning agents!
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

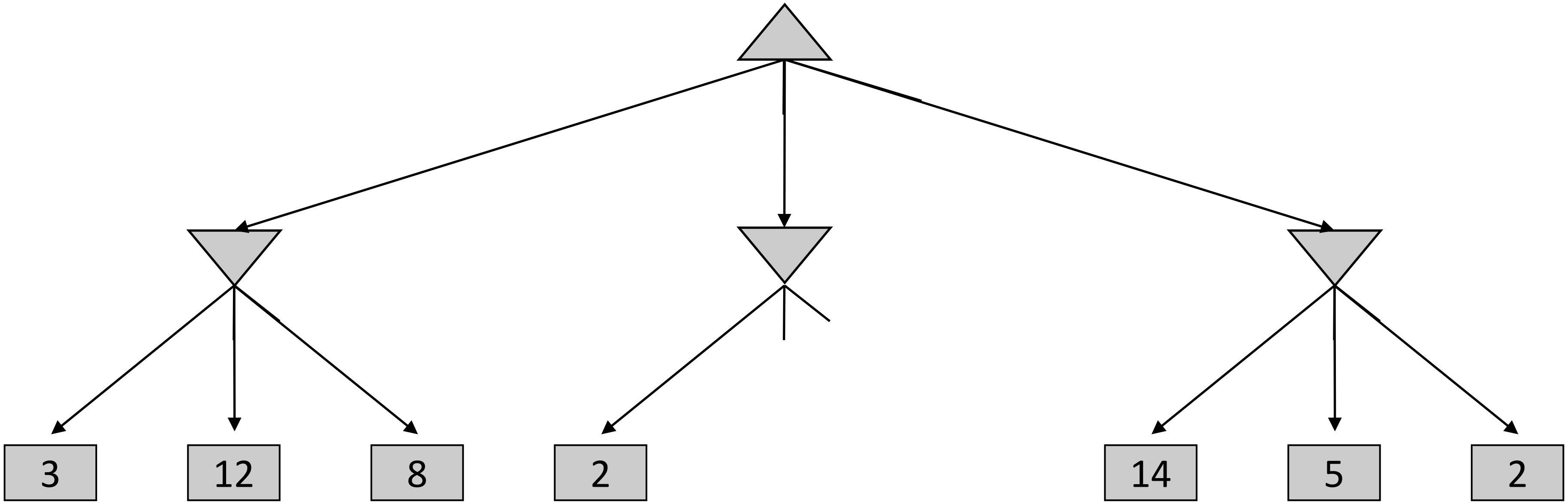
Game Tree Pruning



Minimax Example

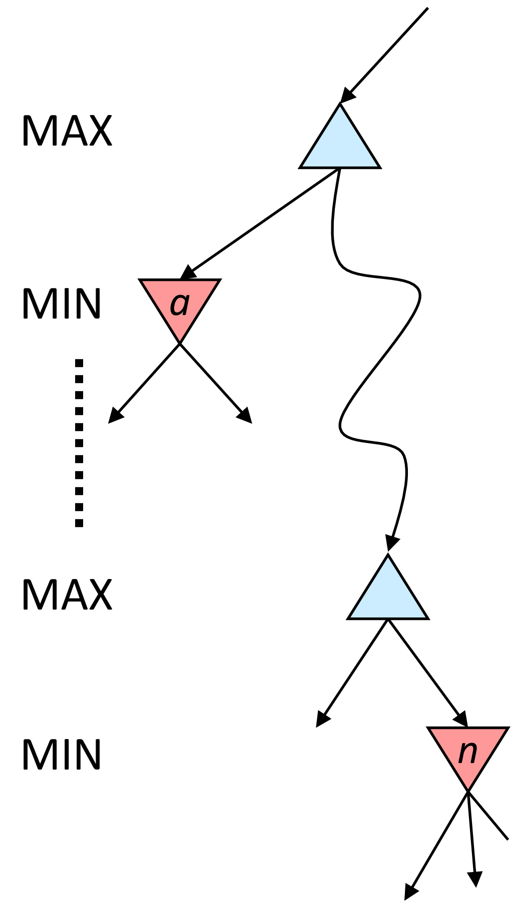


Minimax Pruning



Alpha-Beta Pruning

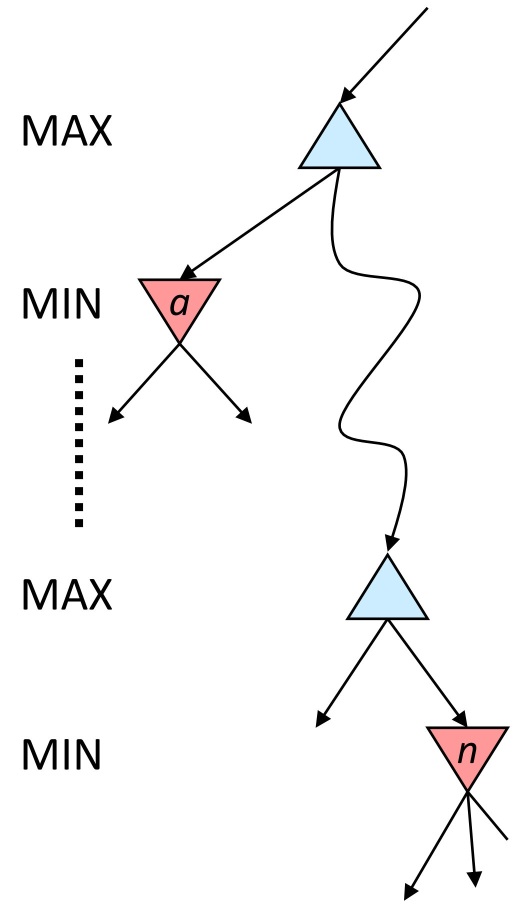
- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is decreasing
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current 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 Pruning

- General configuration (MAX 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 decreasing
  - 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

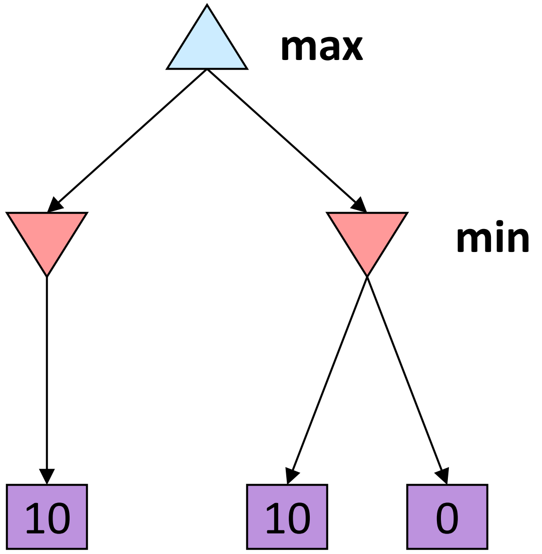
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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
```

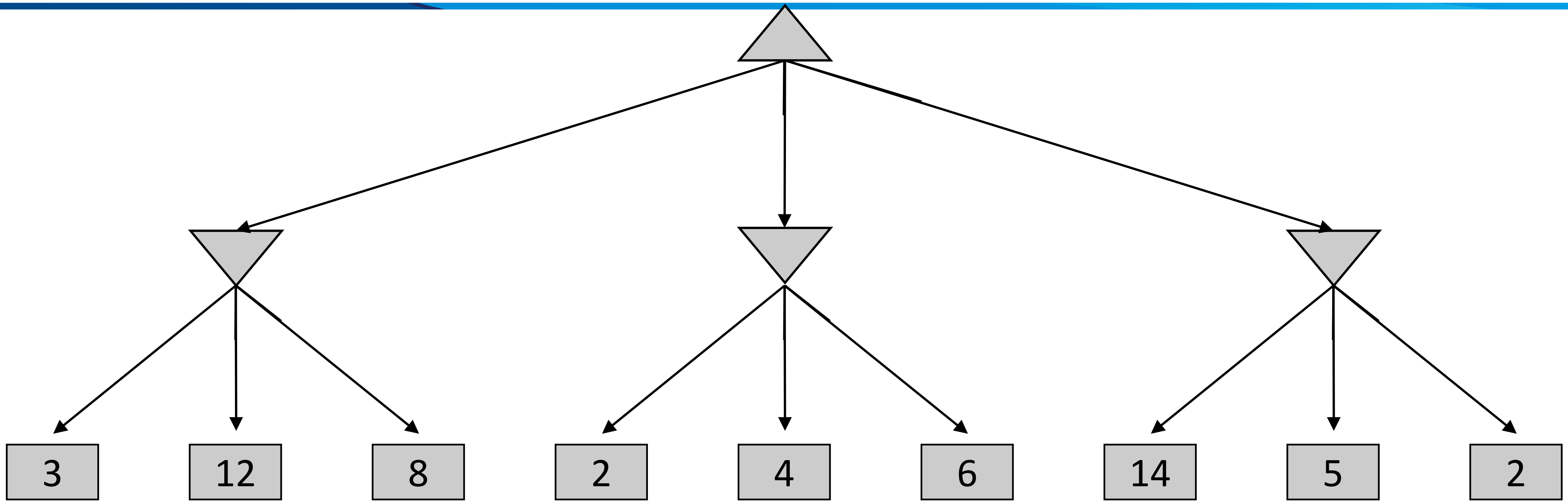
Alpha-Beta Pruning Properties

- This pruning has no effect on minimax value computed for the root!
- Minimax 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…



Minimax Example




```
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 Quiz 2