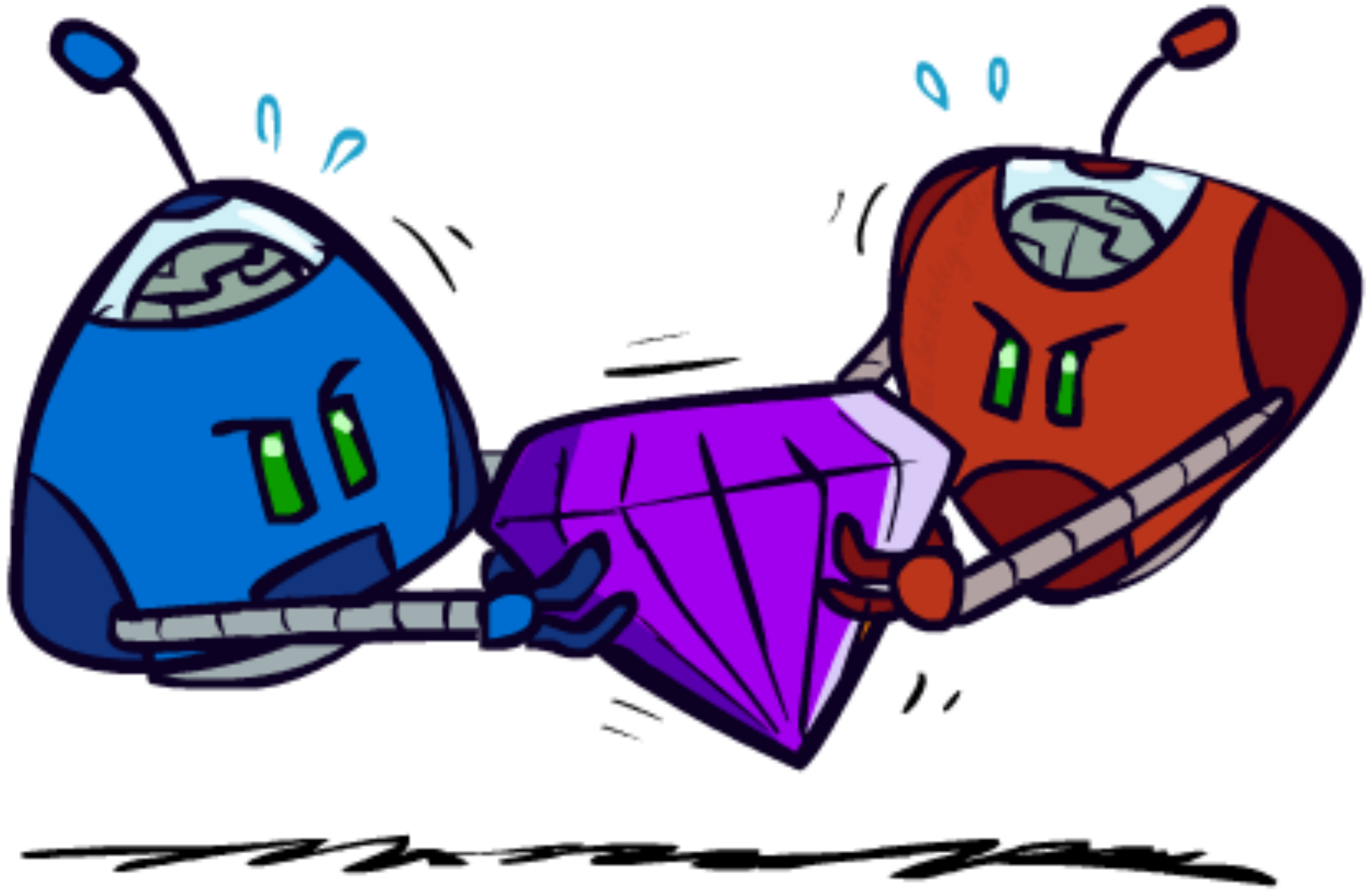
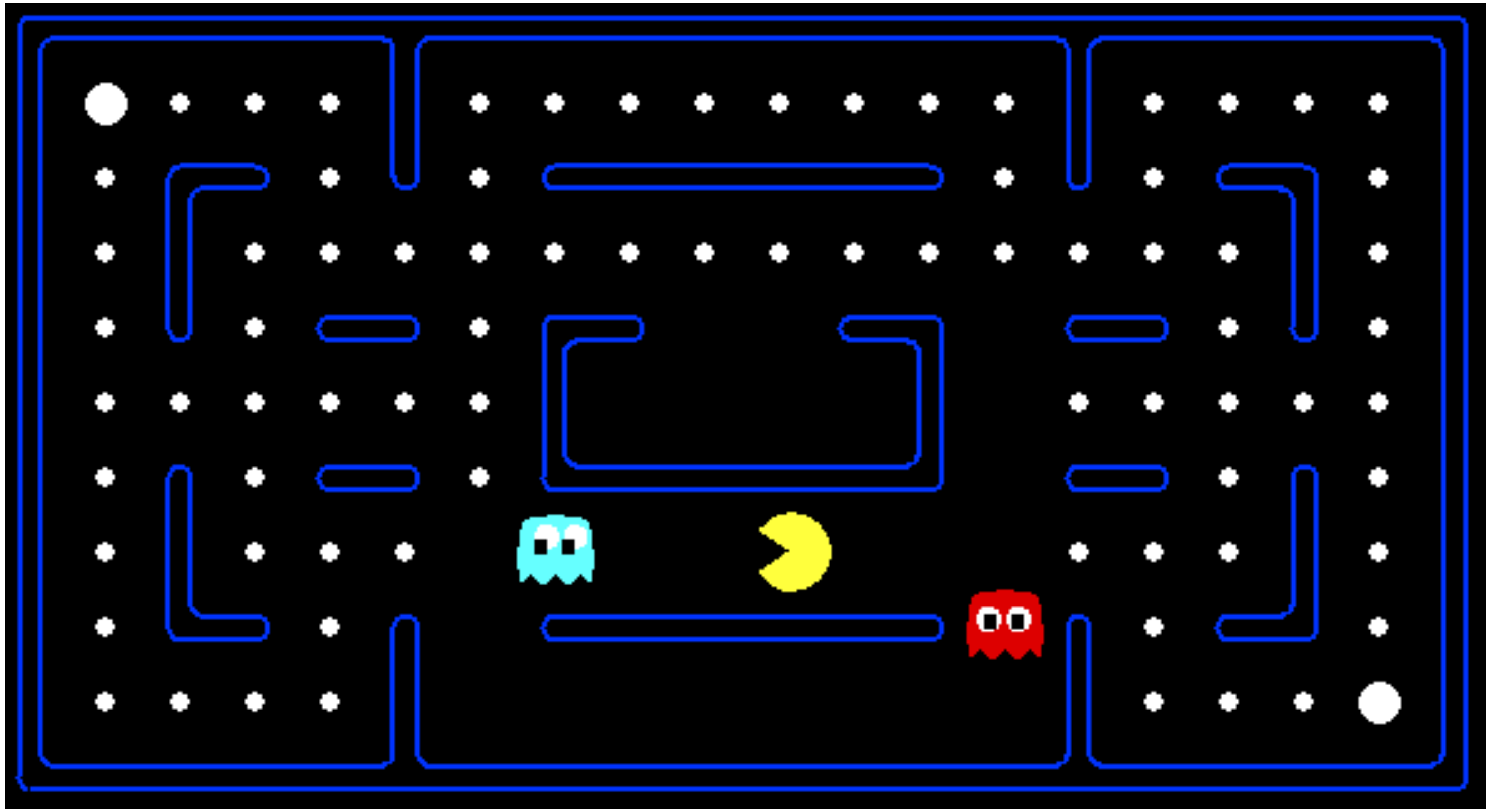
PURDUE CS47100
SEPT 9, 2019
PROF. JENNIFER NEVILLE

INTRODUCTION TO AI

ADVERSARIAL GAMES

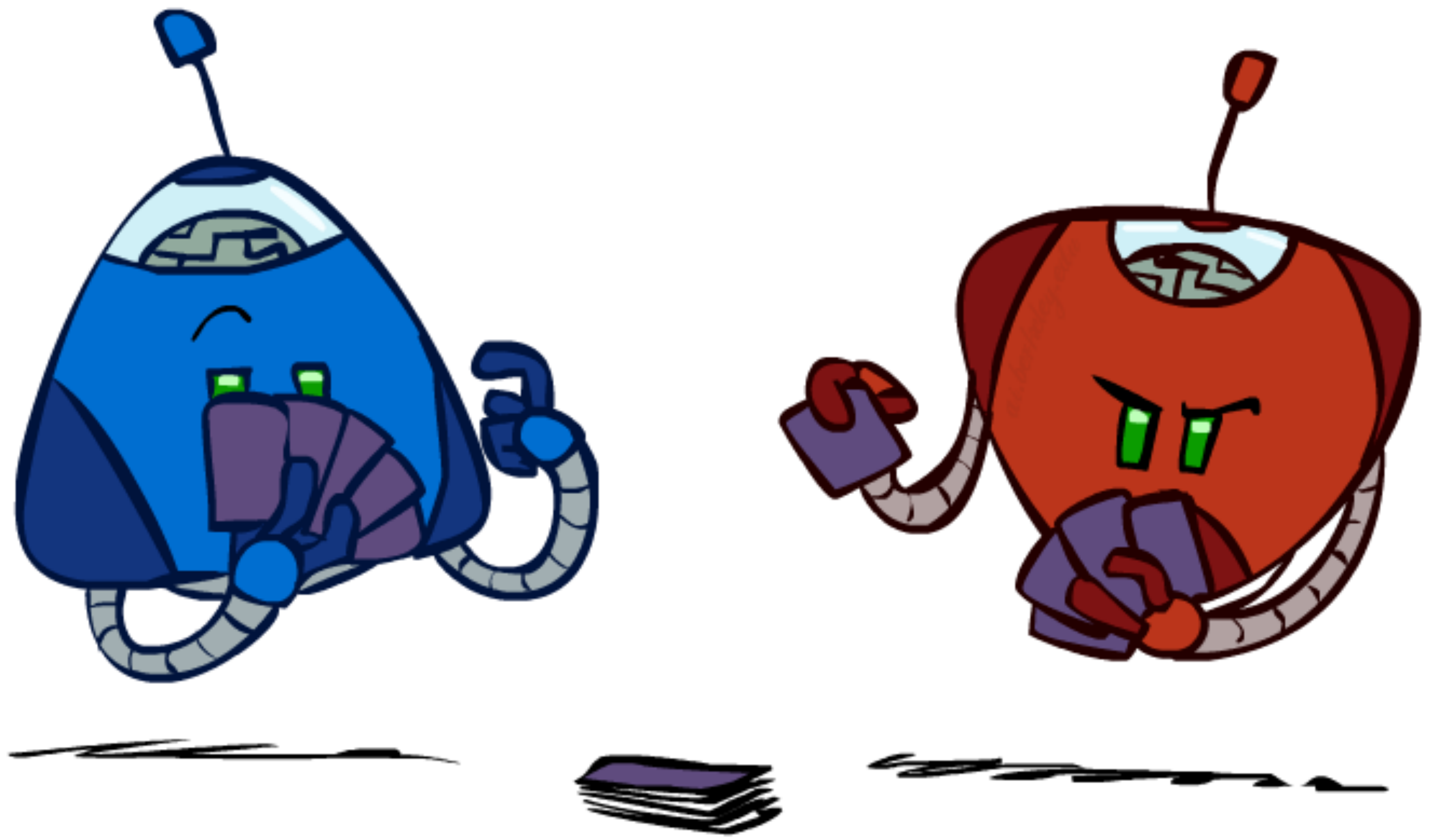


BEHAVIOR FROM COMPUTATION



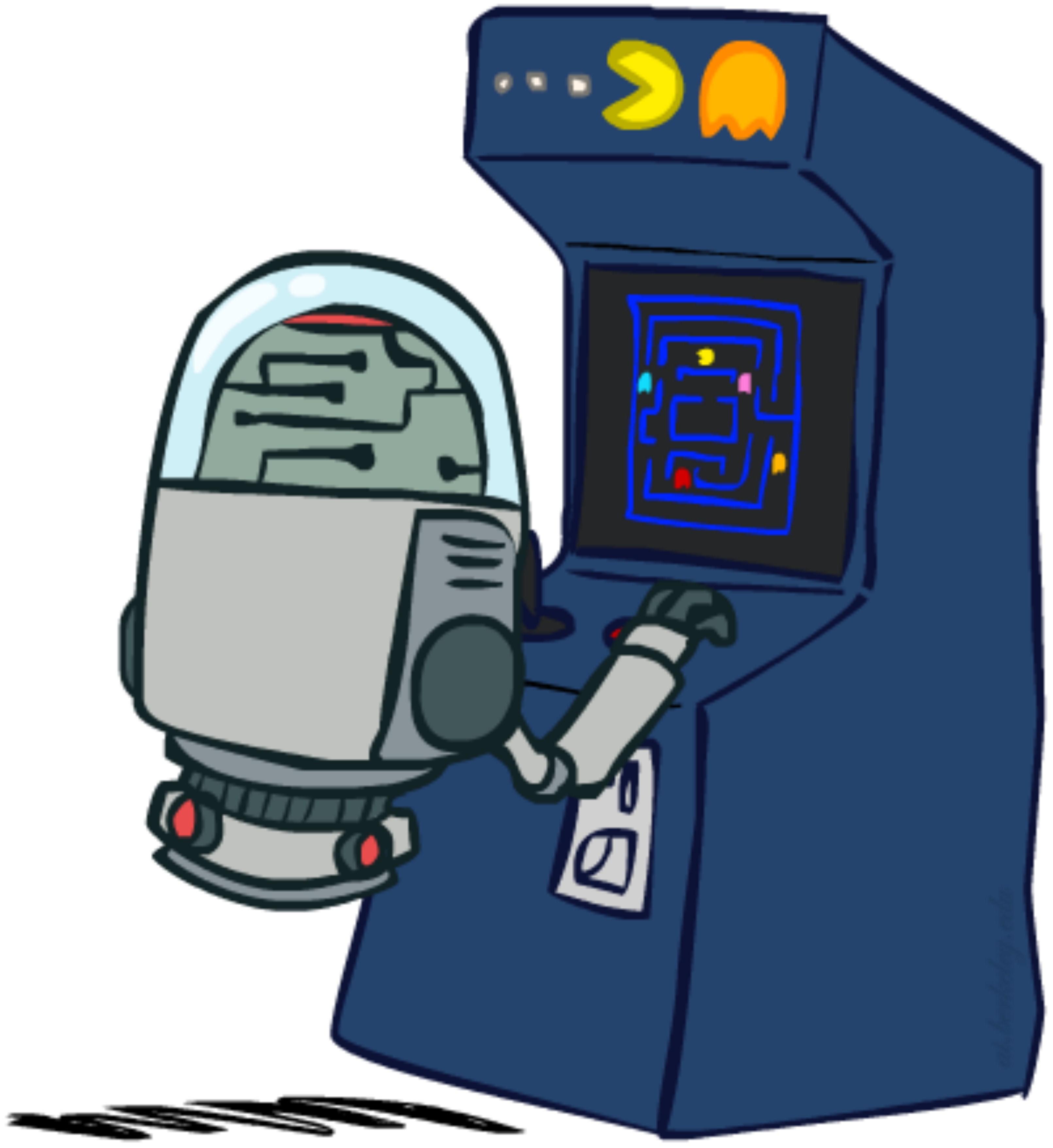
TYPES OF GAMES

- Many different kinds of games
- Axes:
  - Deterministic vs. stochastic
  - One, two, or more players
  - Zero sum
  - Perfect information (can you see the state)
- Algorithms need to calculate a strategy **(policy)** which recommends a move from each state

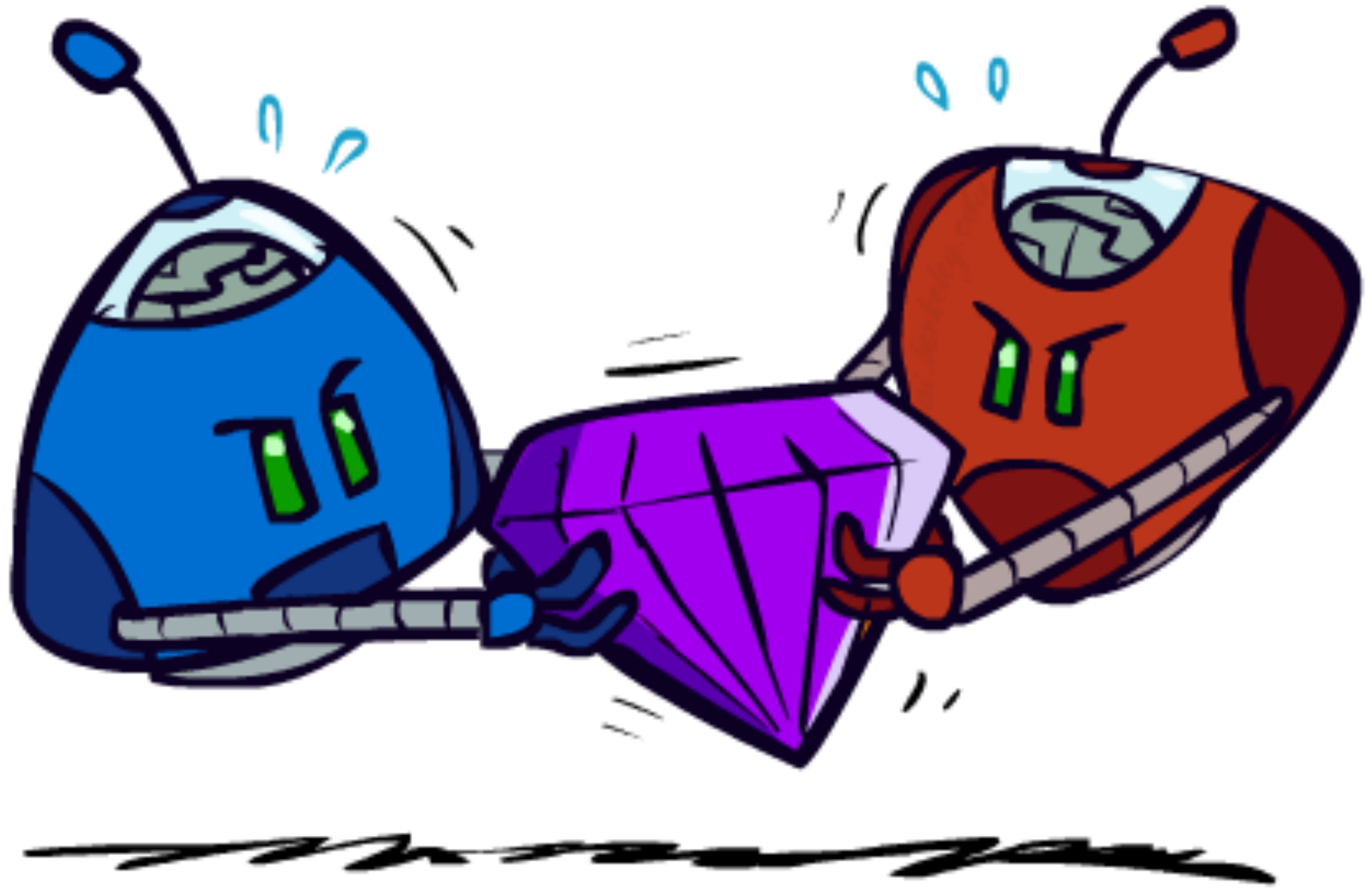


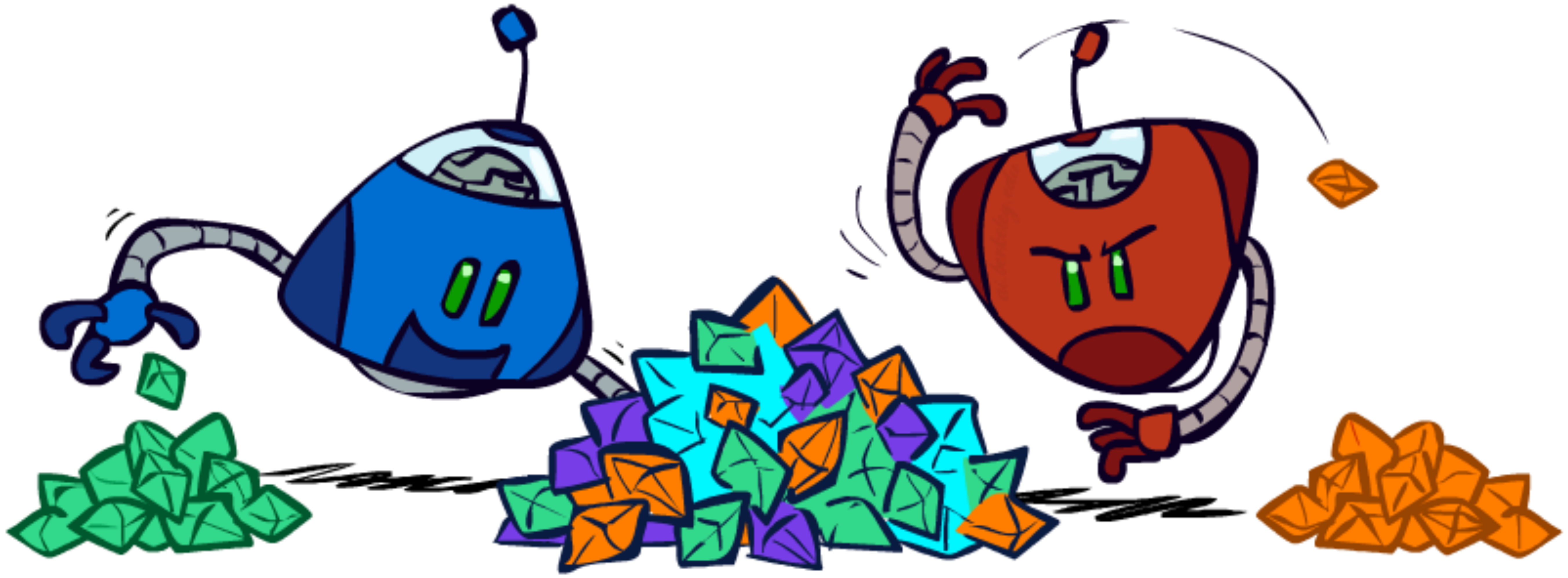
DETERMINISTIC GAMES

- Problem formulation:
  - 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



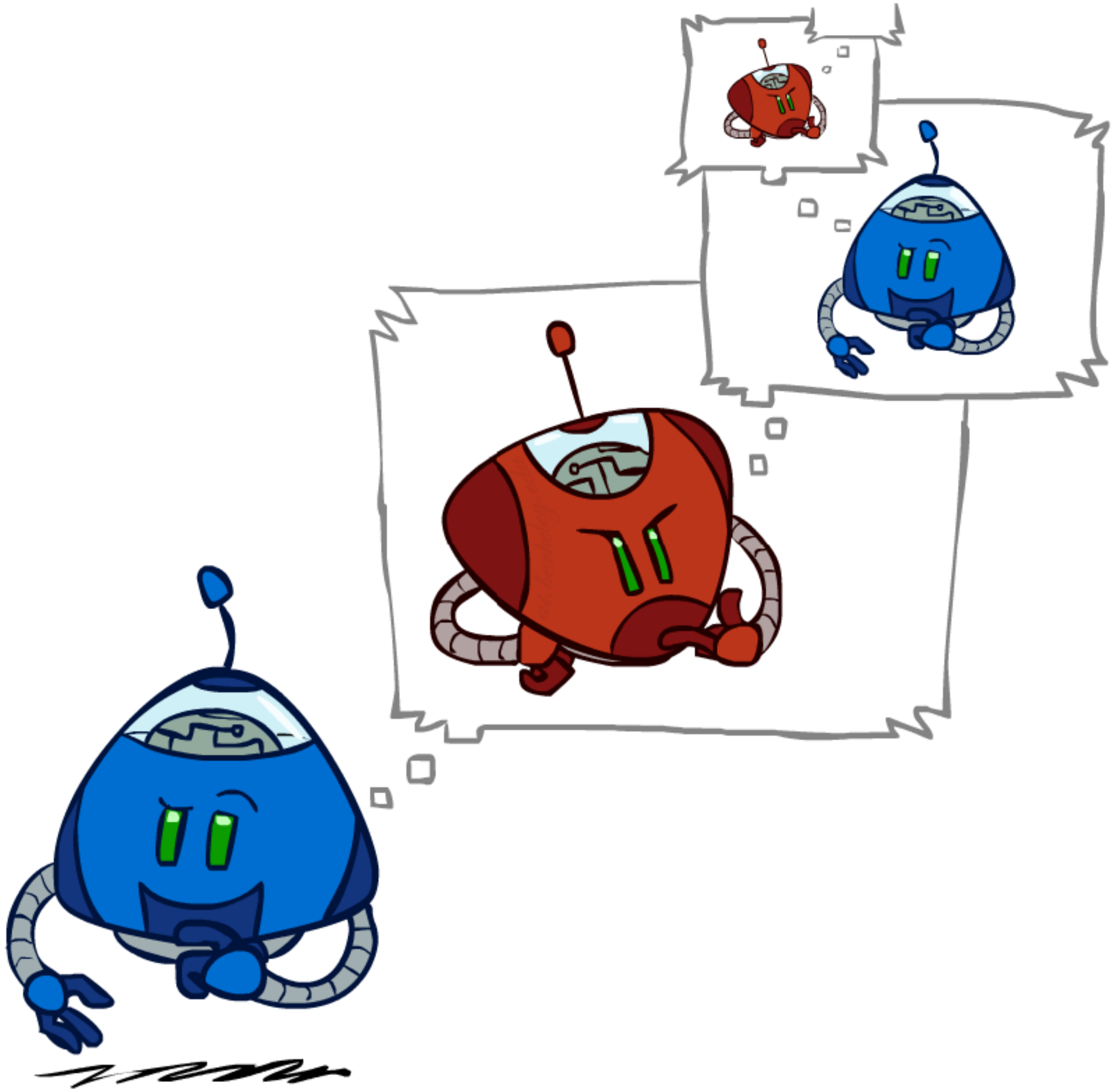
ZERO-SUM GAMES



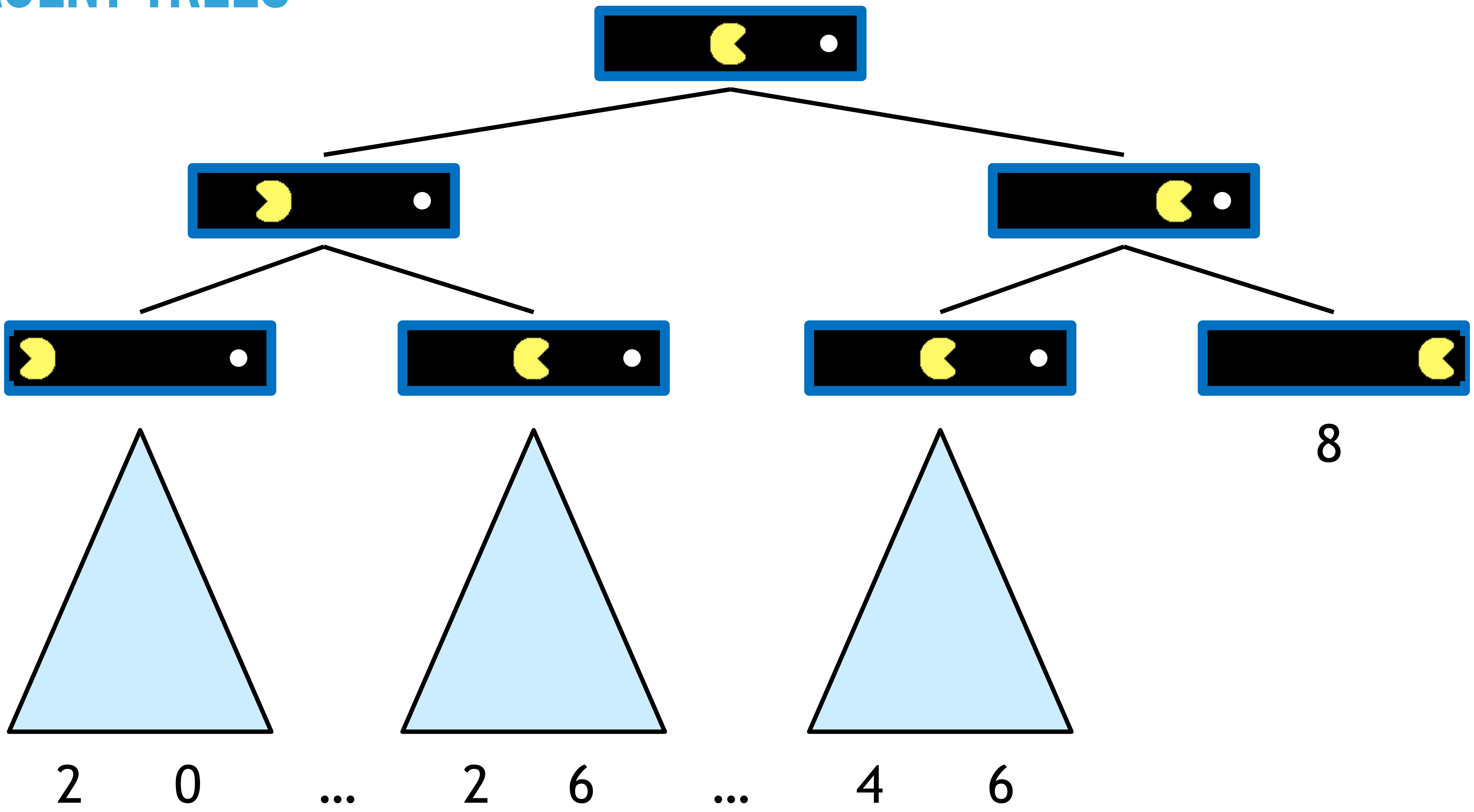


- **Zero-Sum Games**
  - Agents have opposite *utilities* (values on outcomes)
  - Can then think of outcome as a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- **General Games**
  - Agents have independent *utilities* (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

ADVERSARIAL SEARCH

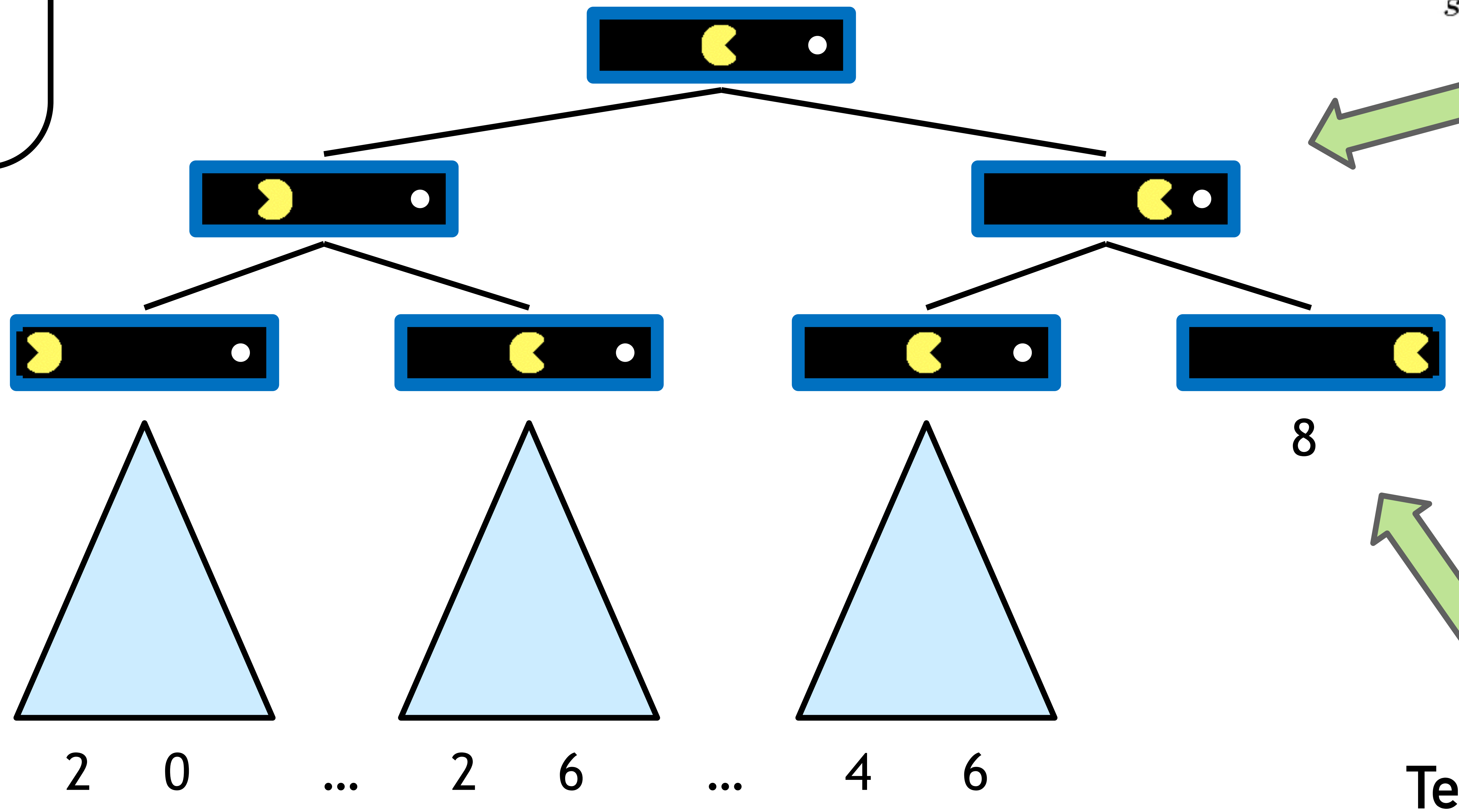


SINGLE-AGENT TREES



VALUE OF A STATE

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



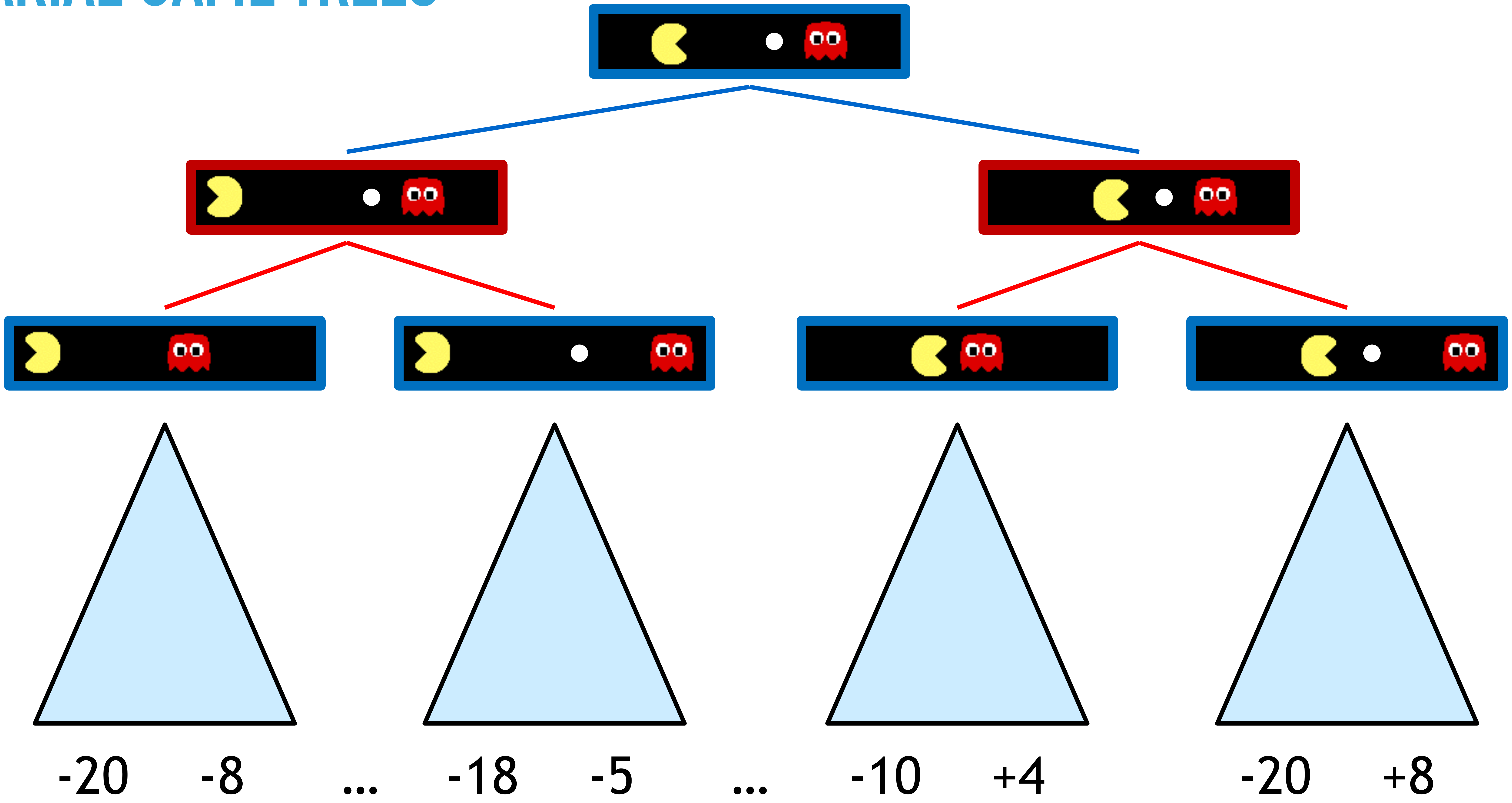
Non-Terminal States:

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

Terminal States:

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

ADVERSARIAL GAME TREES



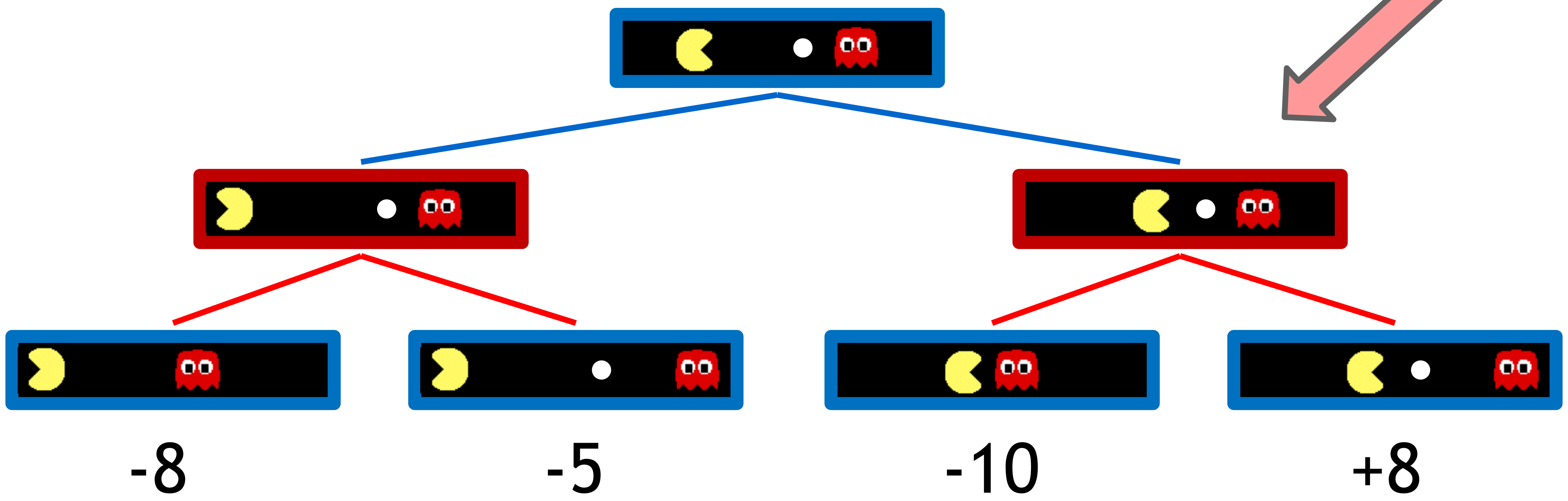
MINIMAX VALUES

States Under Agent's Control:

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

States Under Opponent's Control:

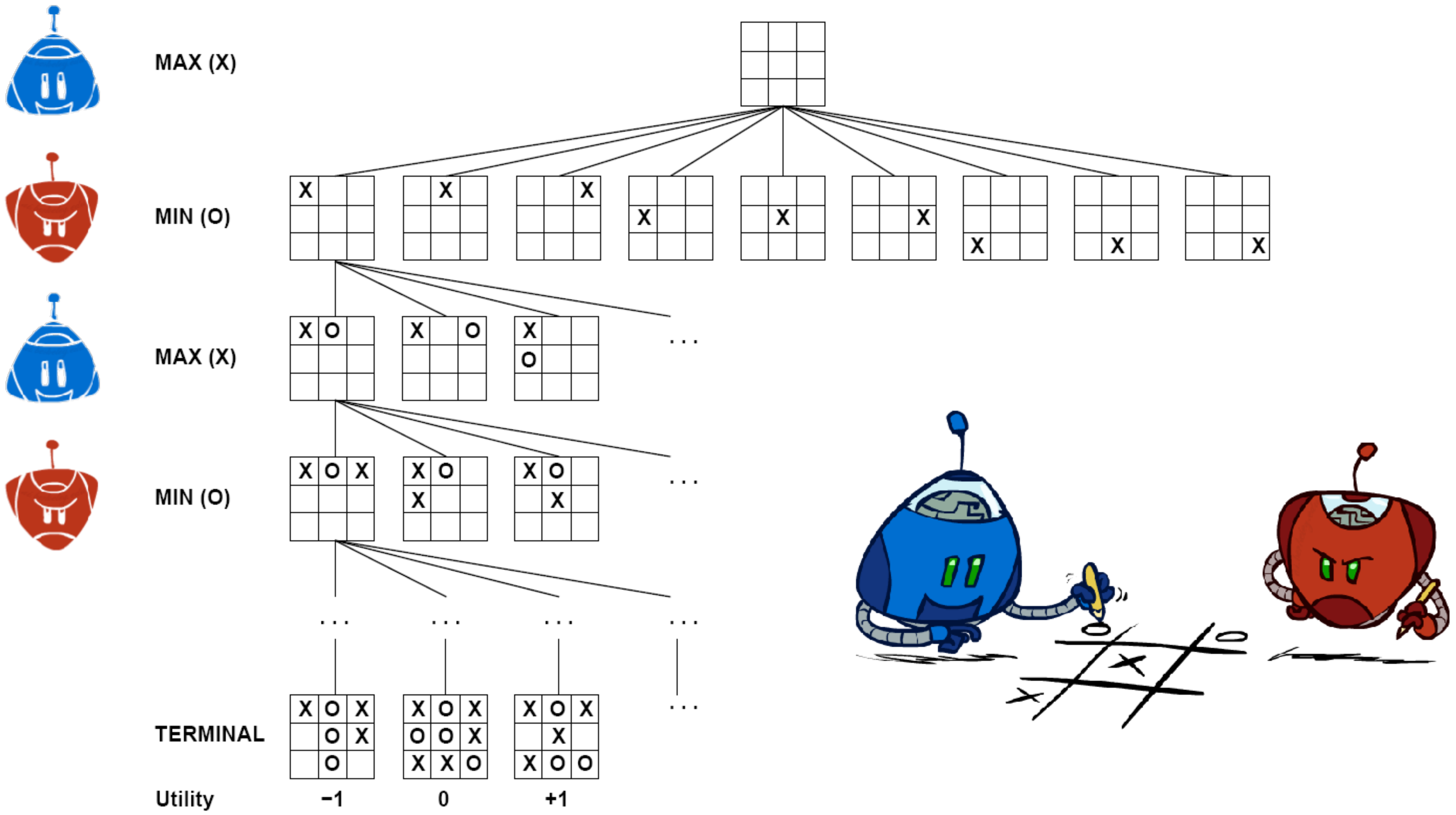
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

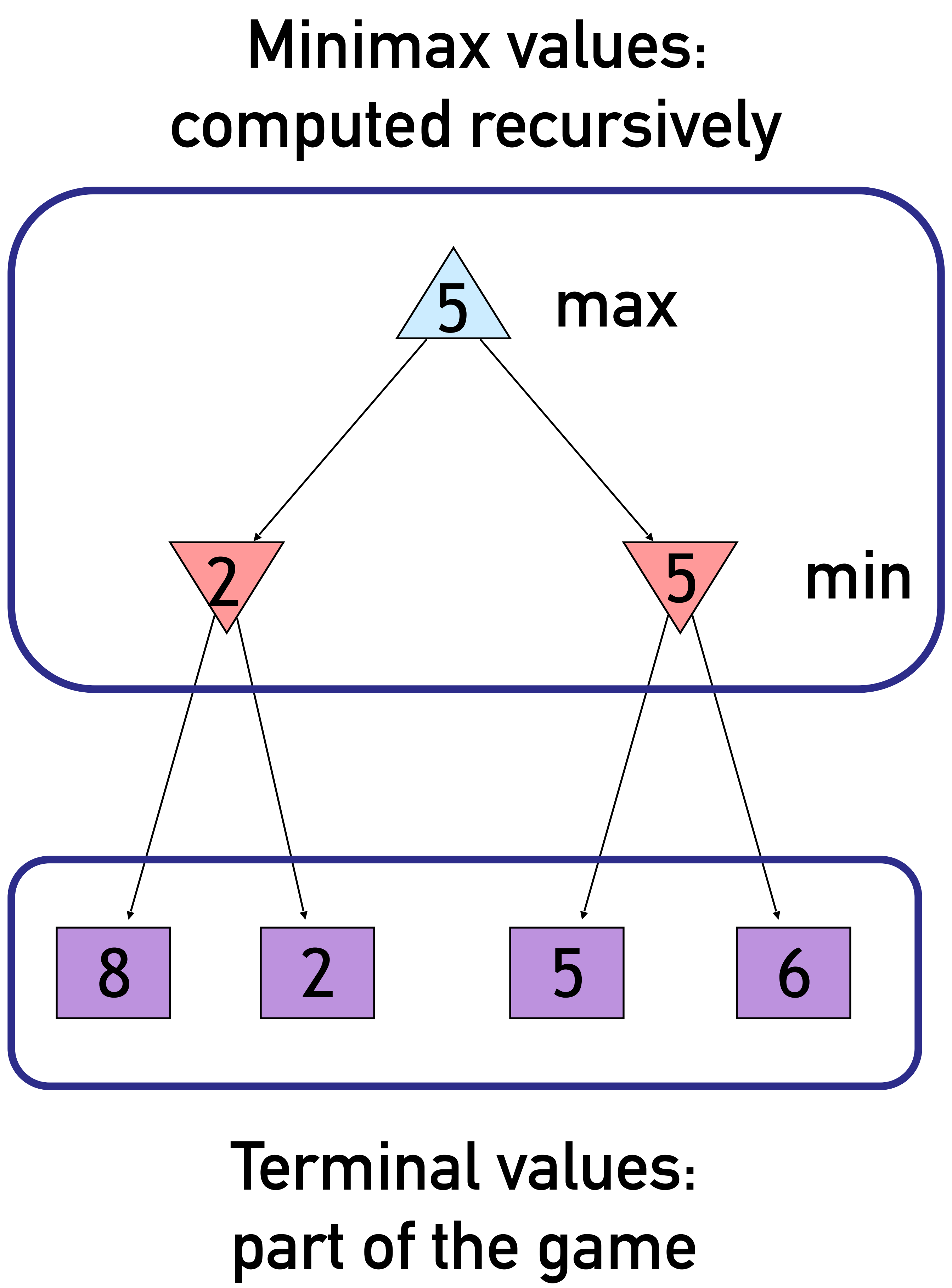
$$V(s) = \text{known}$$

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
```

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

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

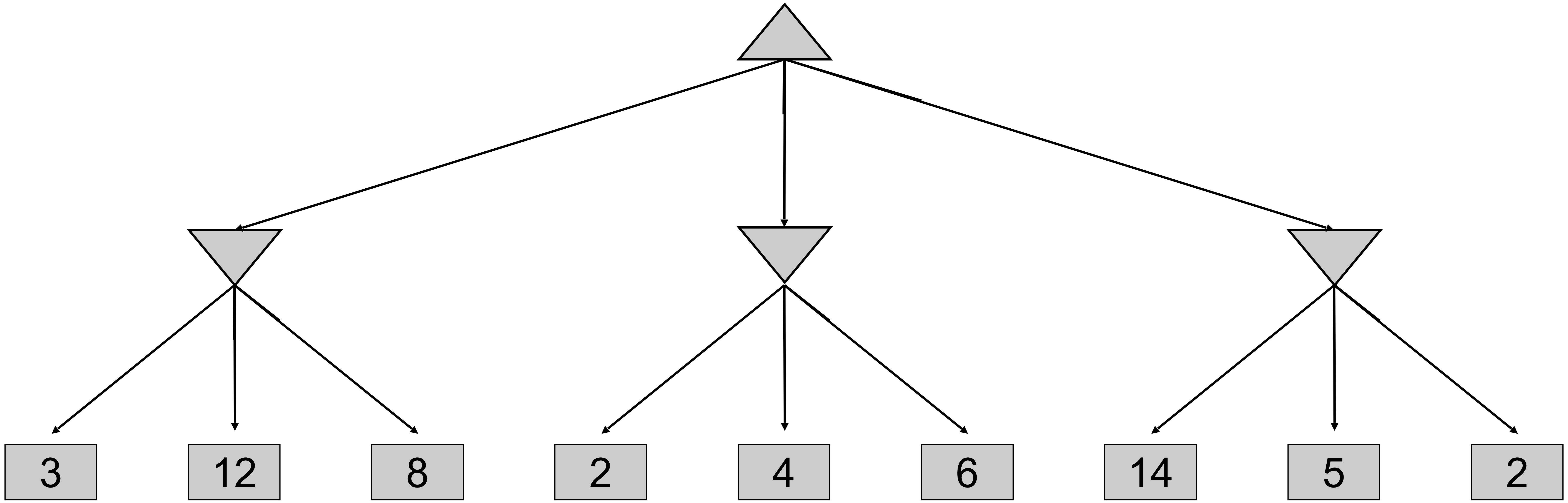
MINIMAX IMPLEMENTATION (DISPATCH)

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

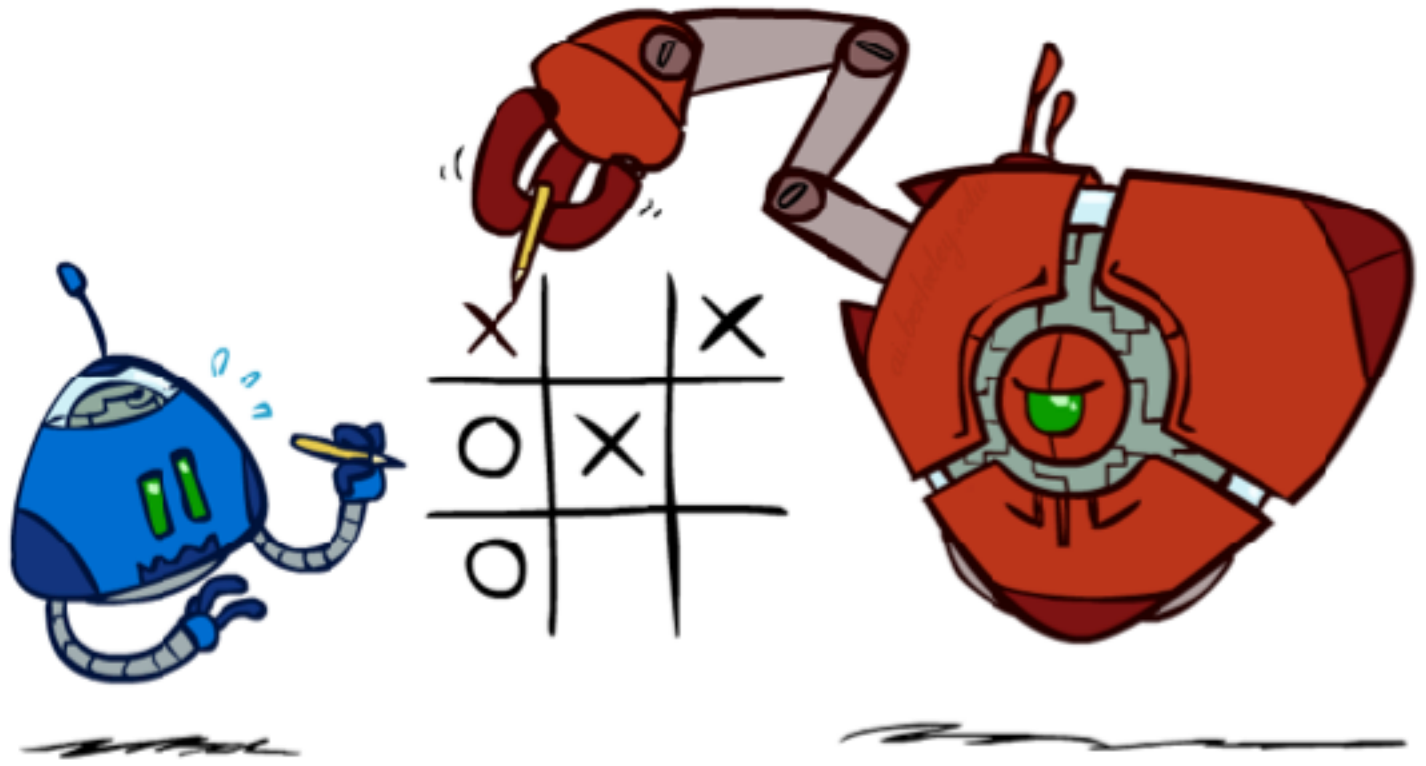
```
def max-value(state):
  initialize v = -∞
  for each successor of state:
    v = max(v, value(successor))
  return v
```

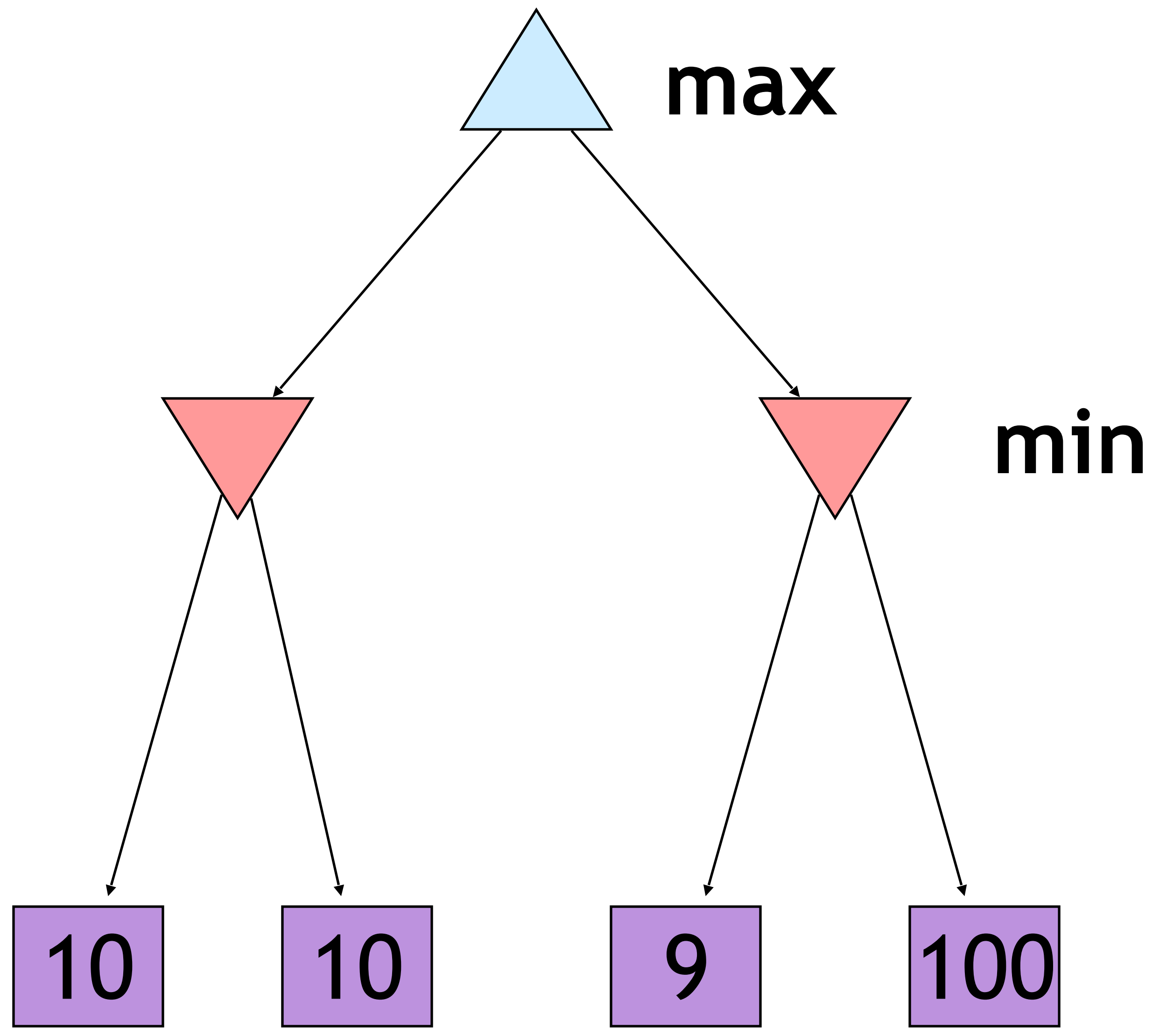
```
def min-value(state):
  initialize v = +∞
  for each successor of state:
    v = min(v, value(successor))
  return v
```

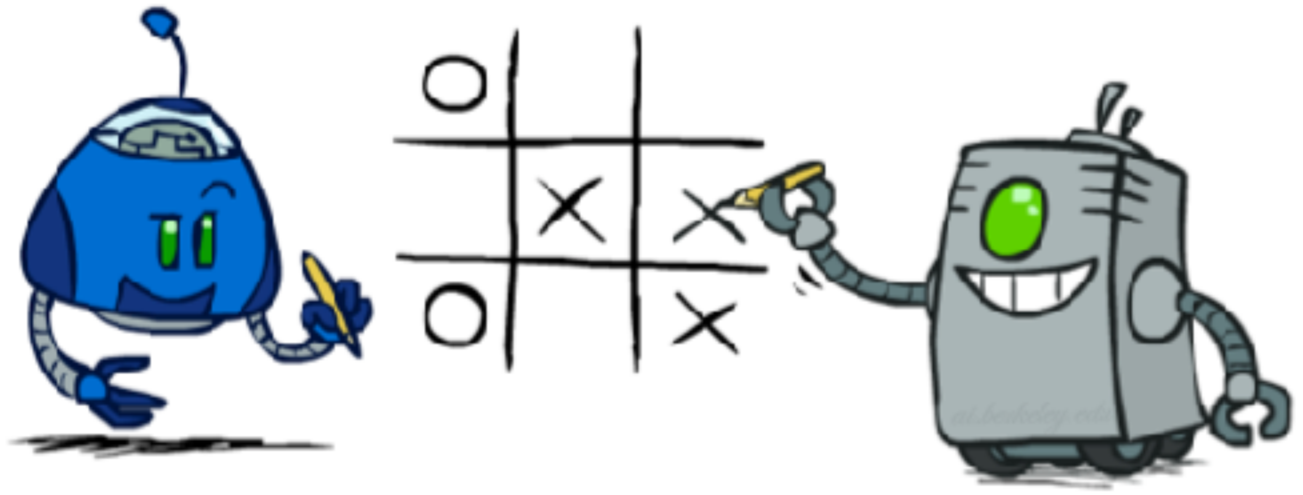
MINIMAX EXAMPLE



MINIMAX PROPERTIES



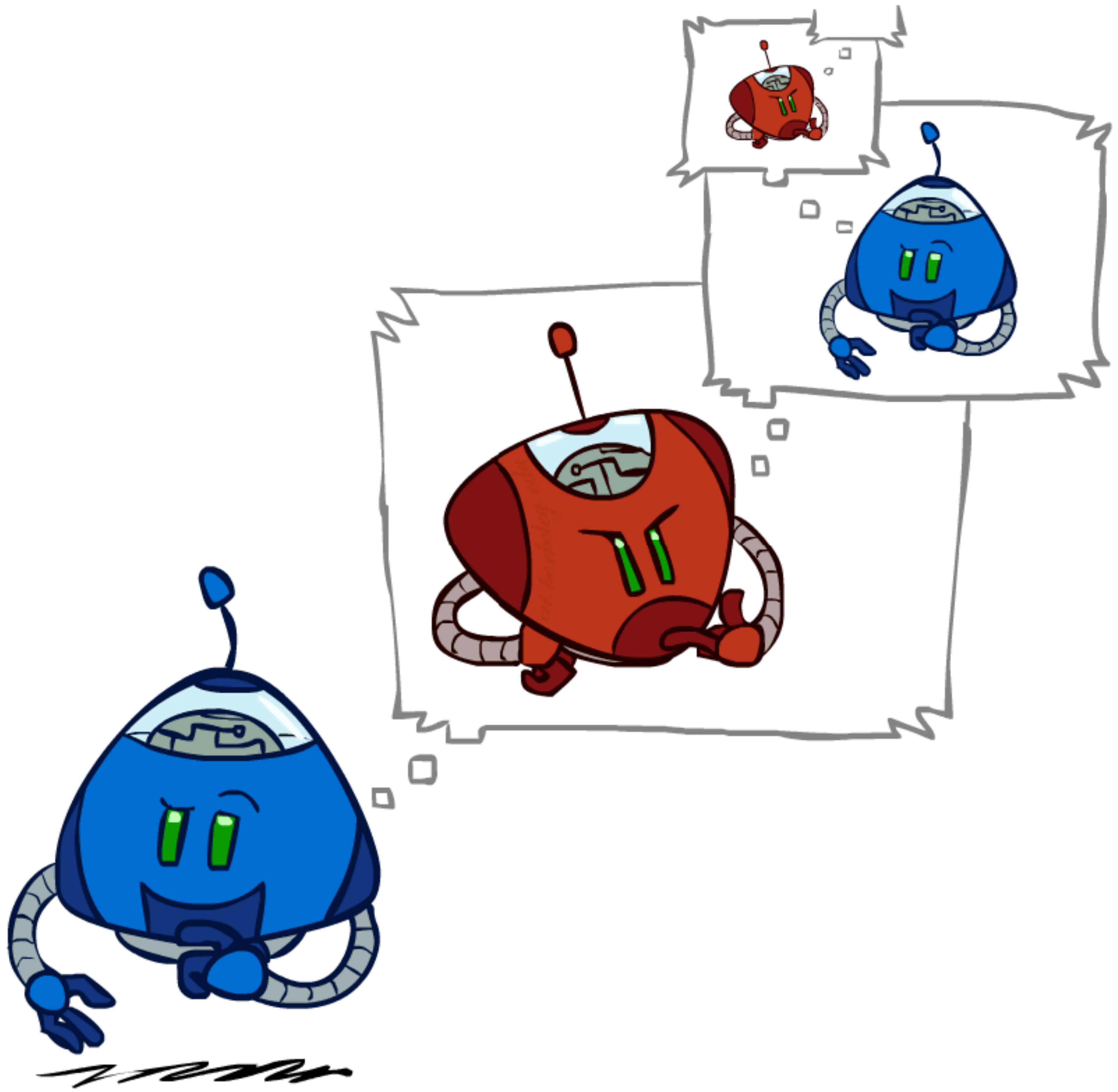




Optimal against a perfect player. Otherwise?

MINIMAX EFFICIENCY

- Efficient 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
  - But, do we need to explore the whole tree?



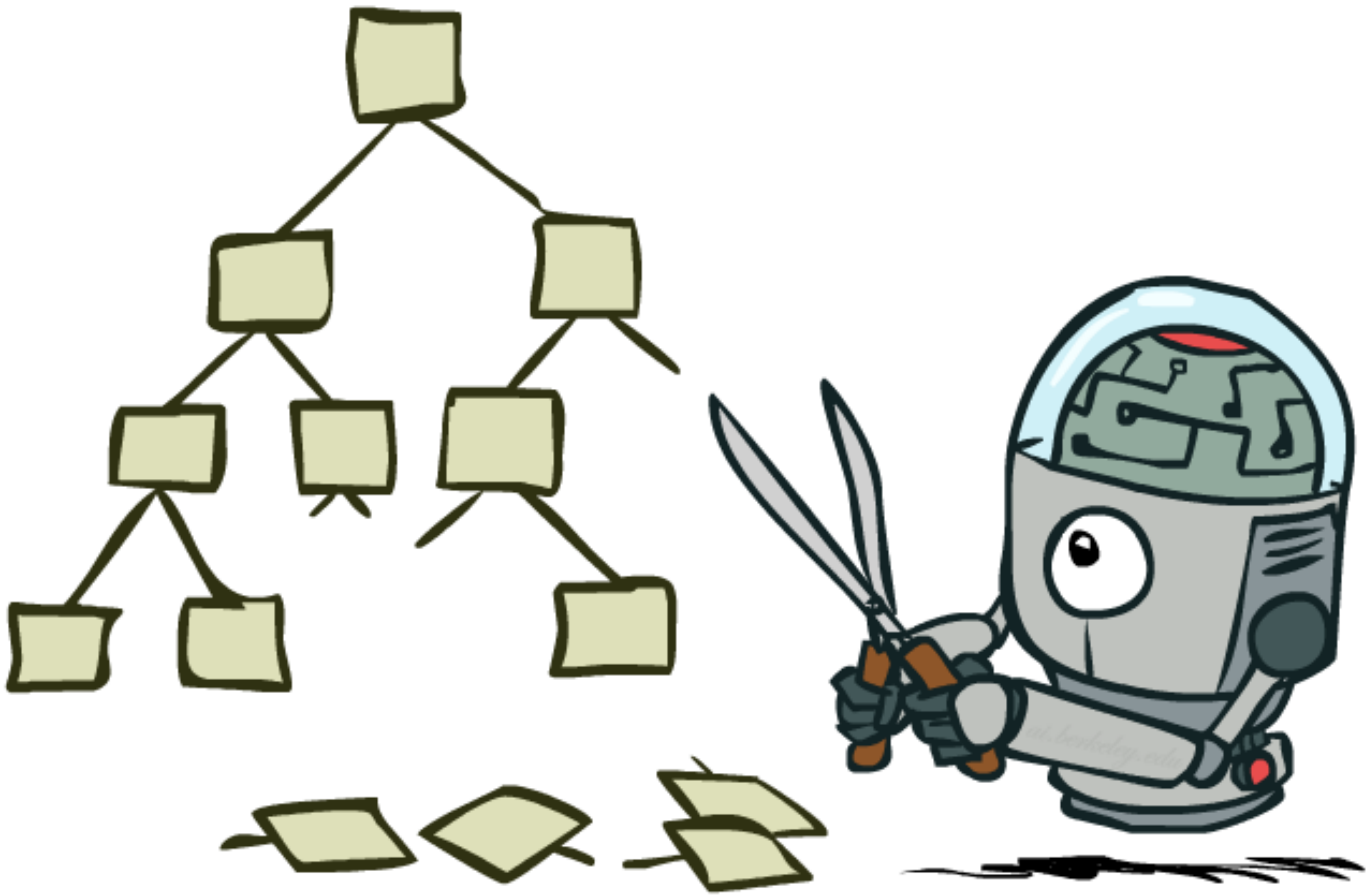
GAME TREE SIZES

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

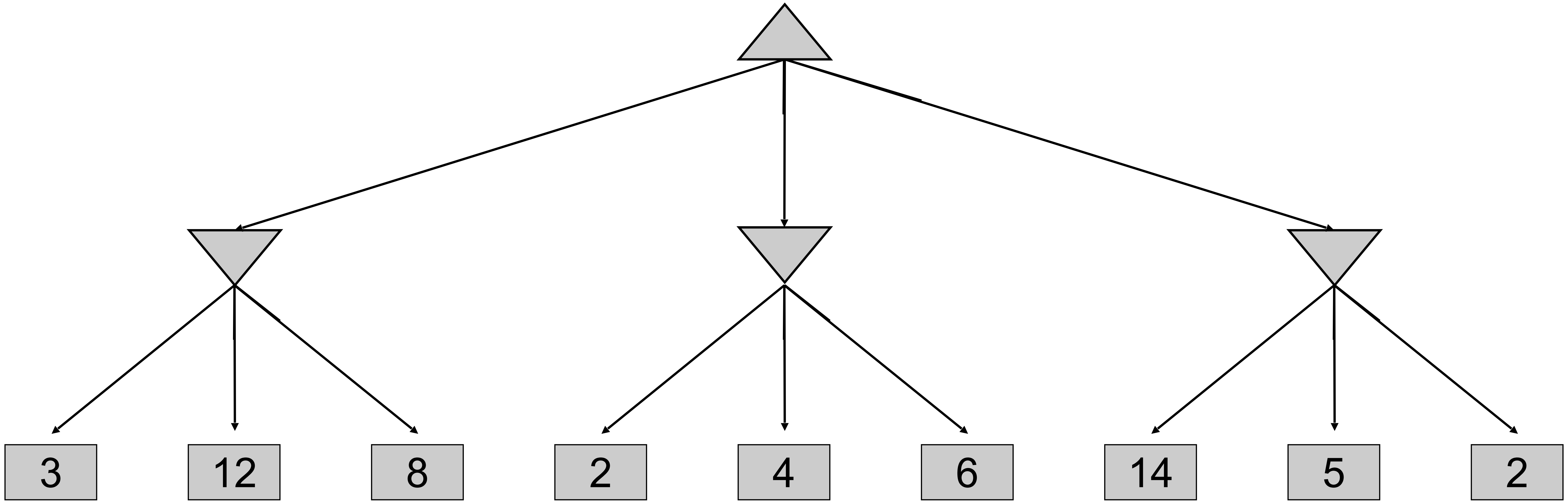
Assume that a computer can evaluate 1 million board configurations per second.

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.

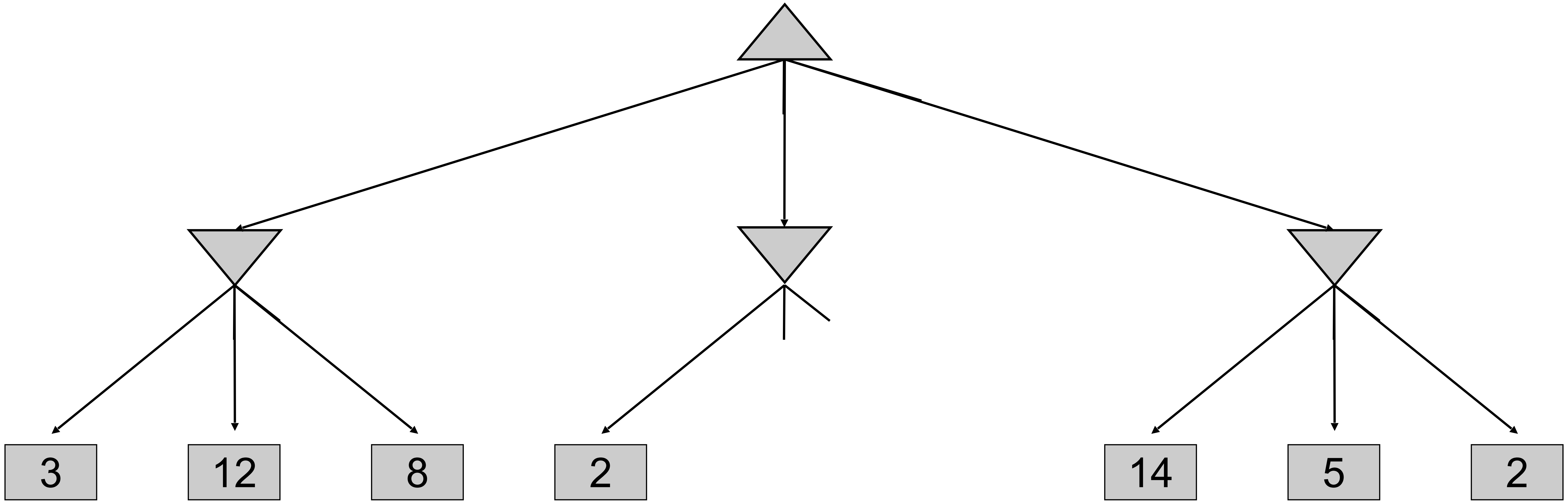
GAME TREE PRUNING



MINIMAX EXAMPLE

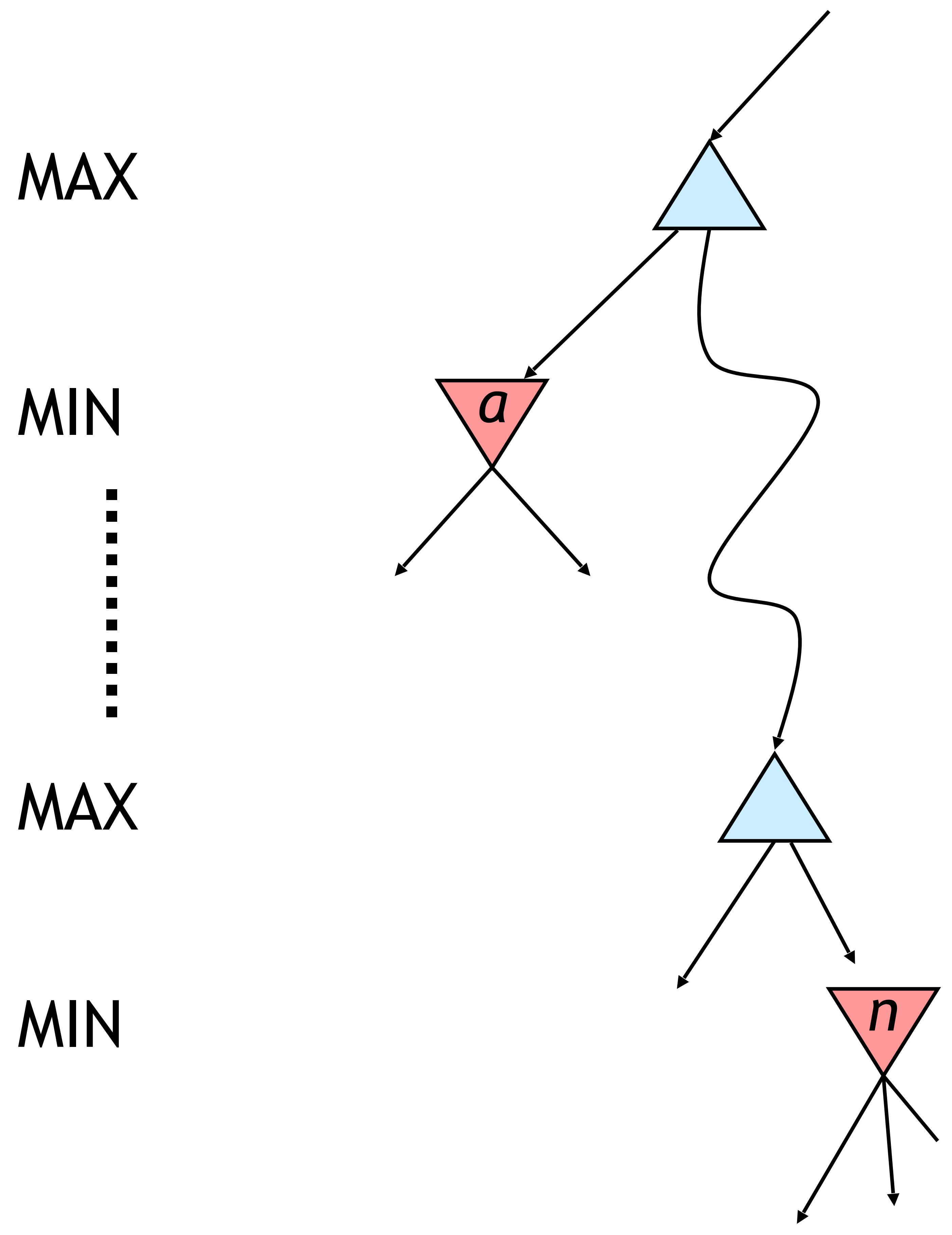


MINIMAX PRUNING



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 *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

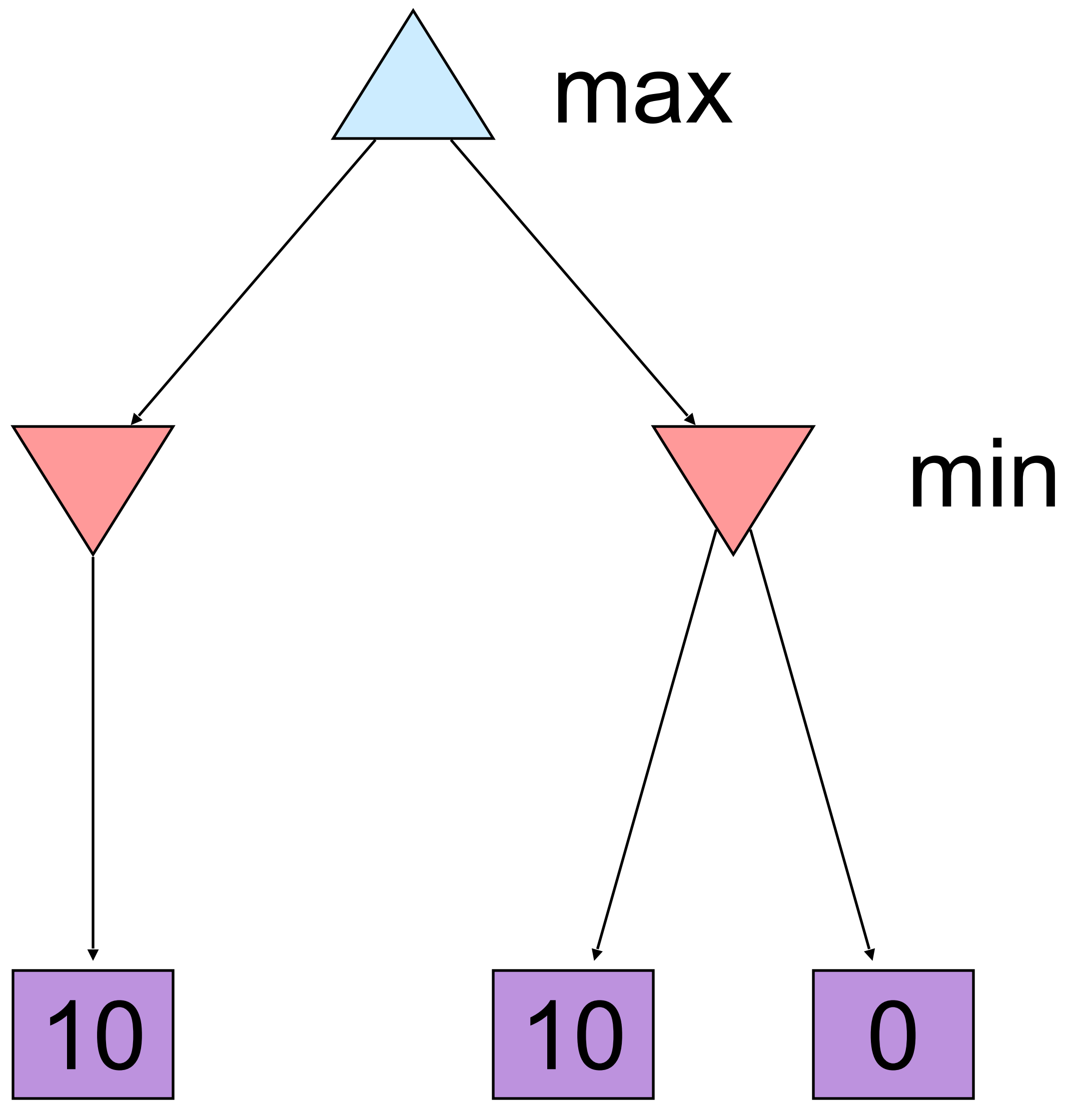
```
α: MAX's best option on path to root
β: MIN's best option on path to root
```

```
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

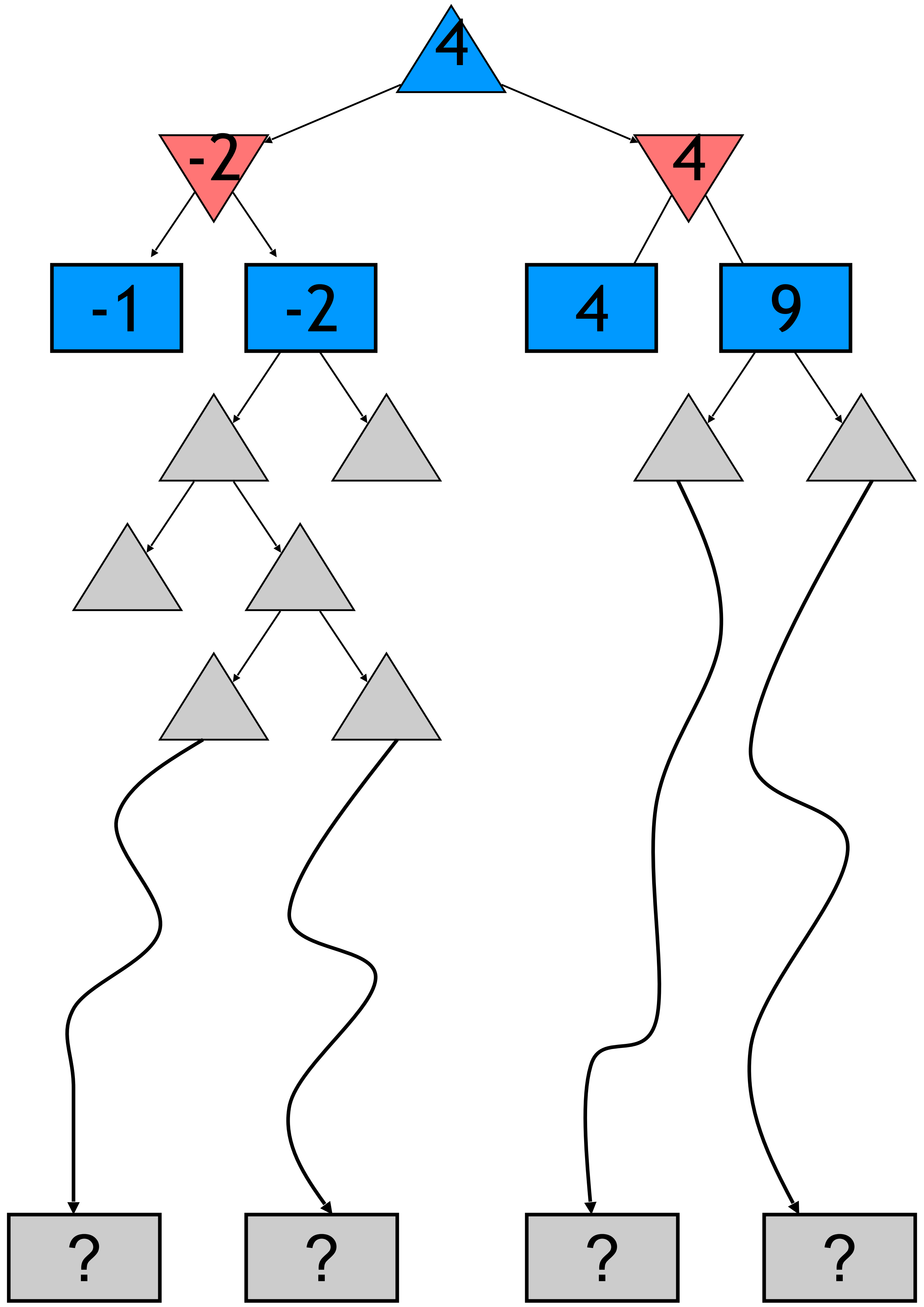
- This pruning has **no effect** on minimax value computed for the root
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- 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)



RESOURCE LIMITS

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

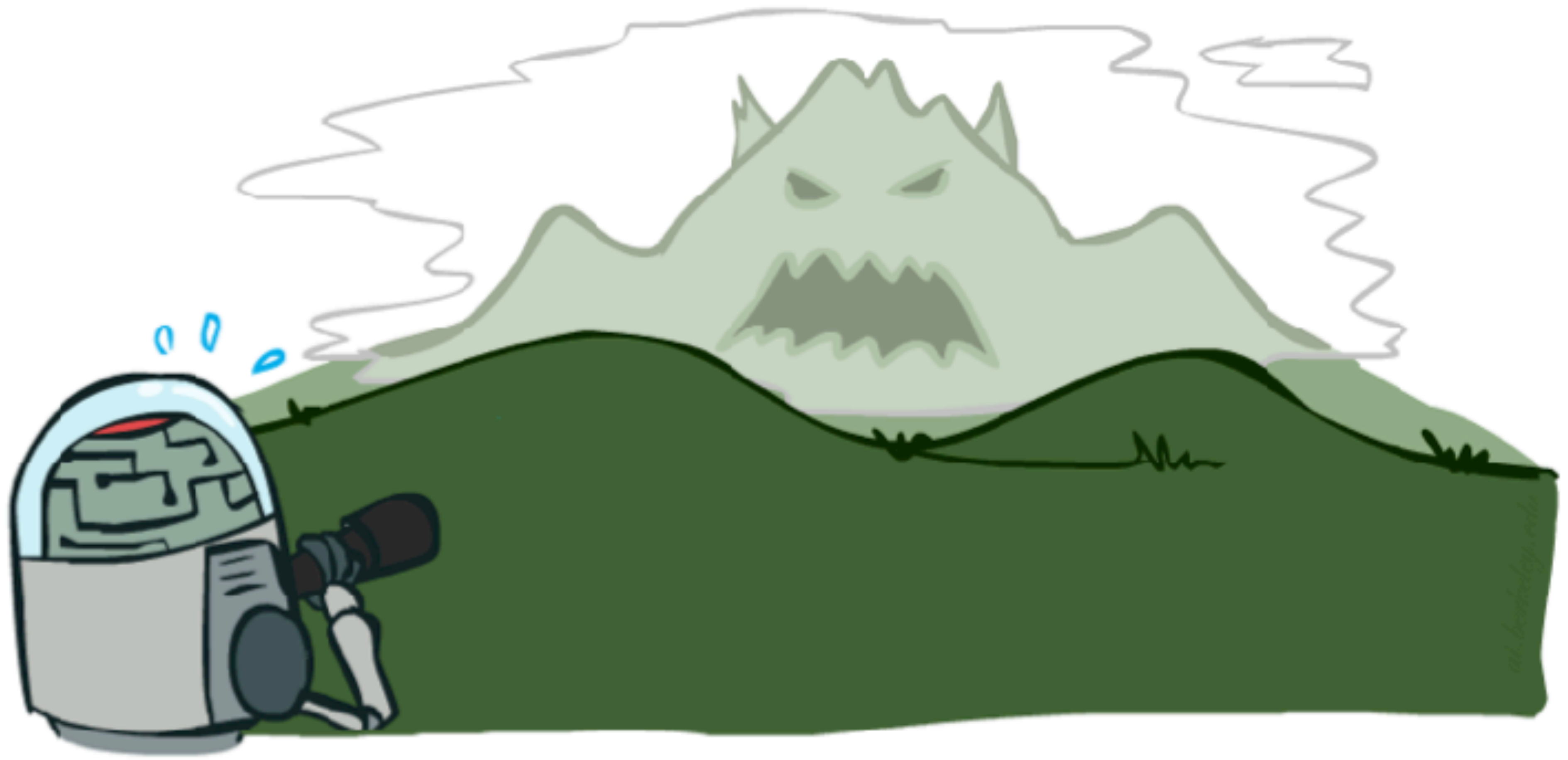


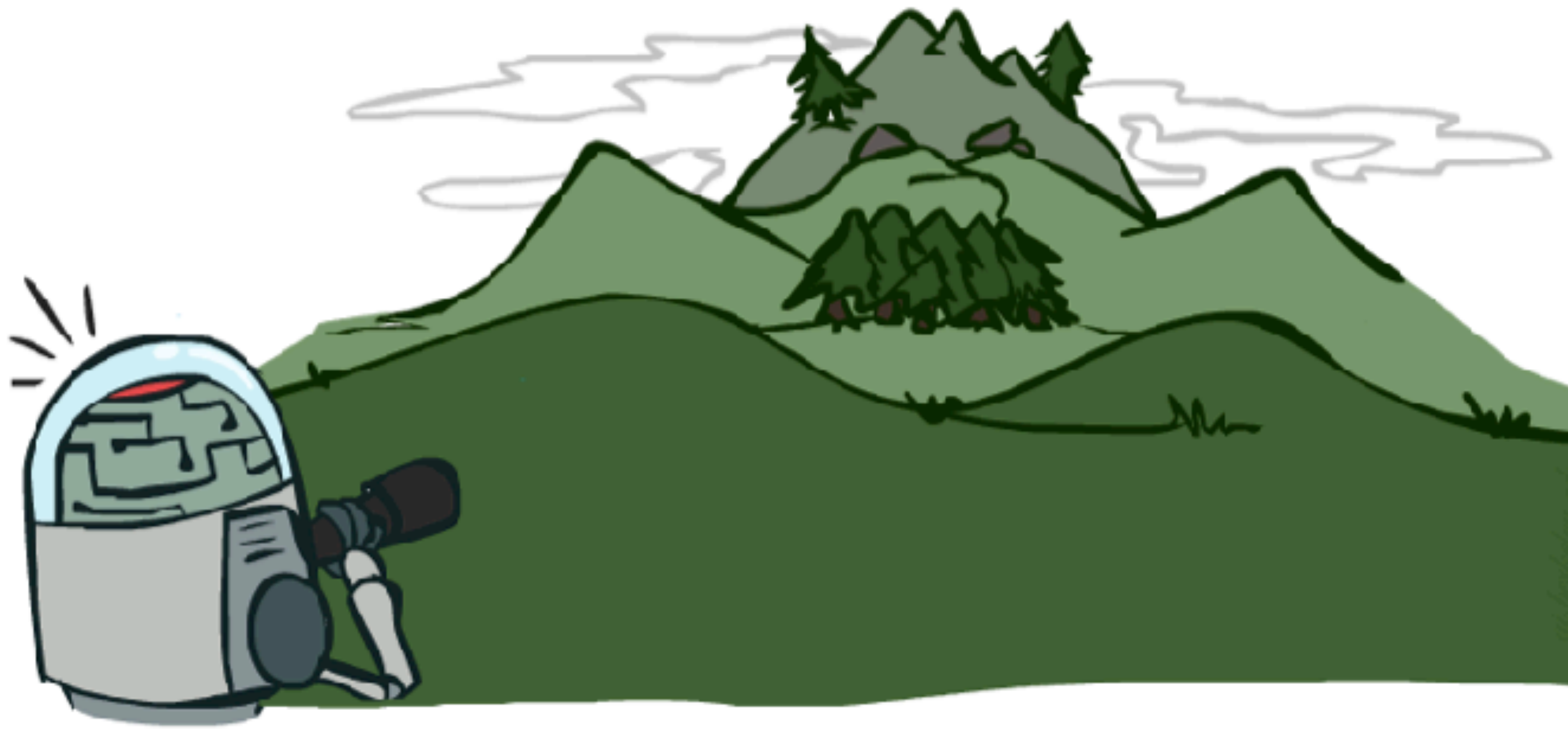
max

min

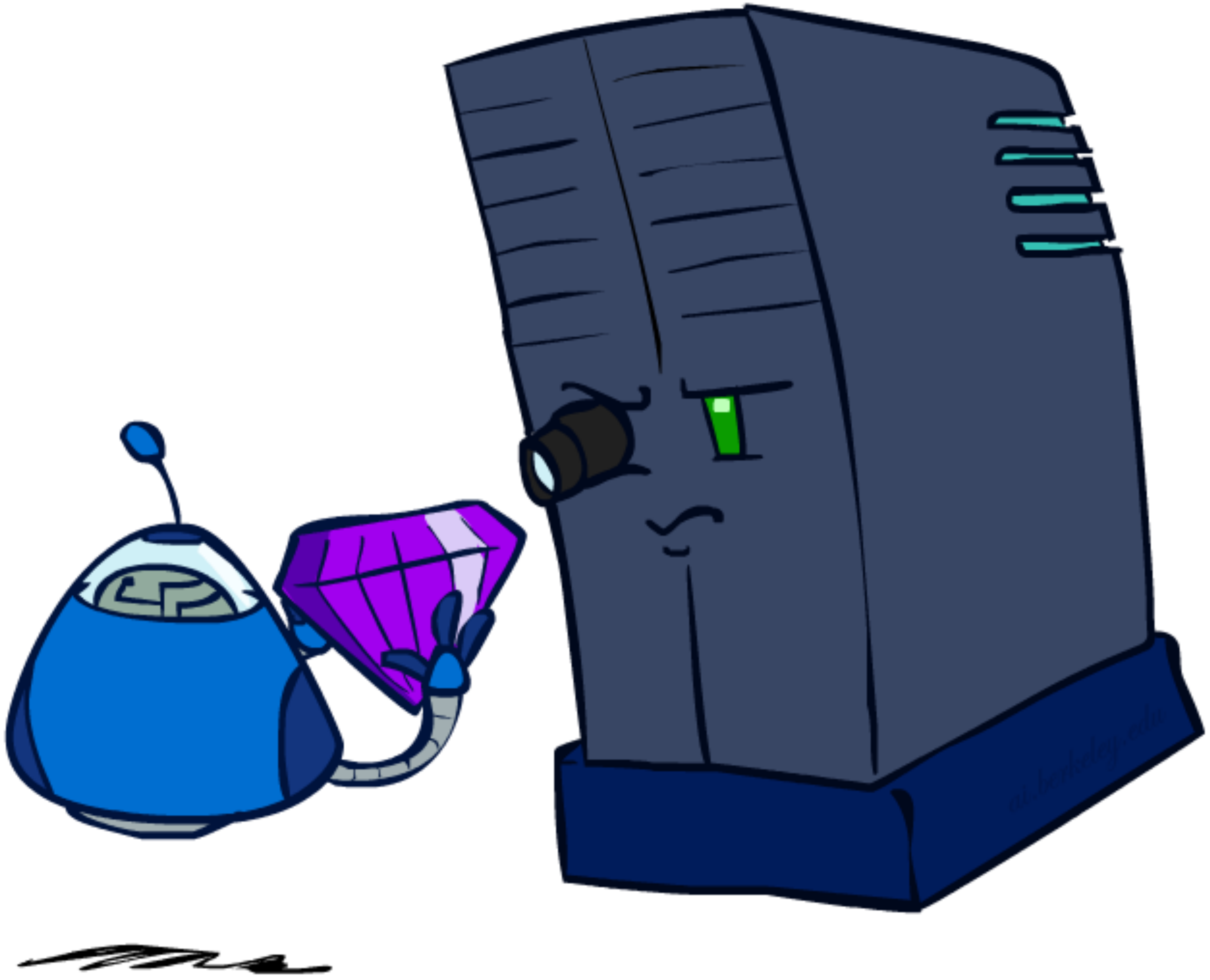
DEPTH MATTERS

- 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



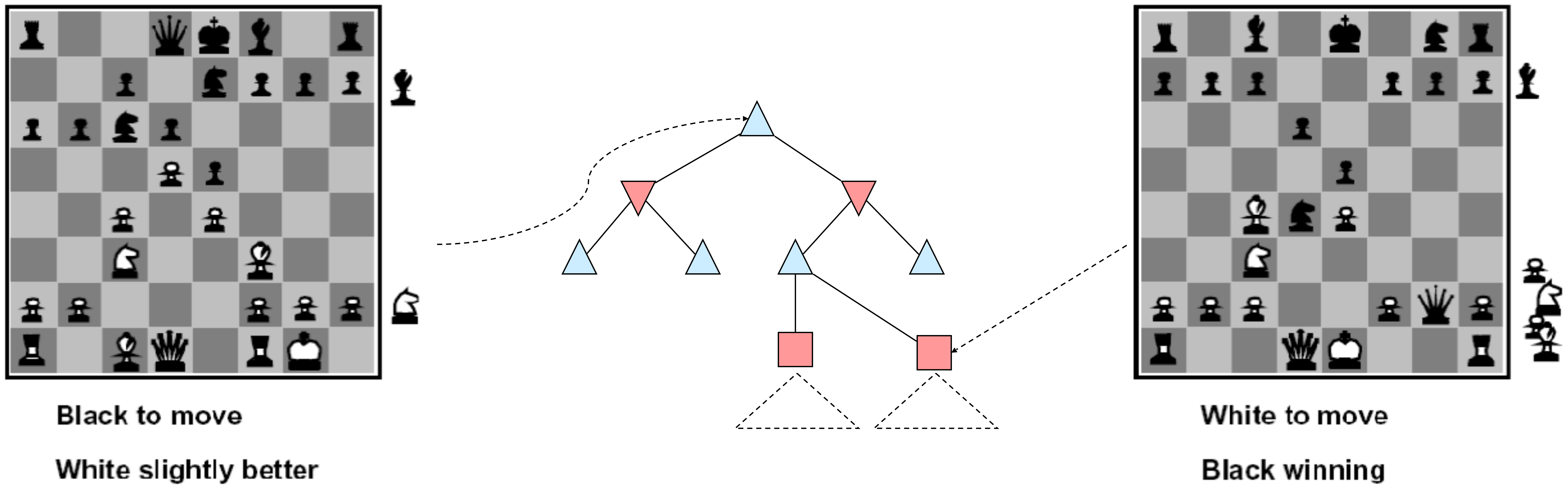


EVALUATION FUNCTIONS



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.