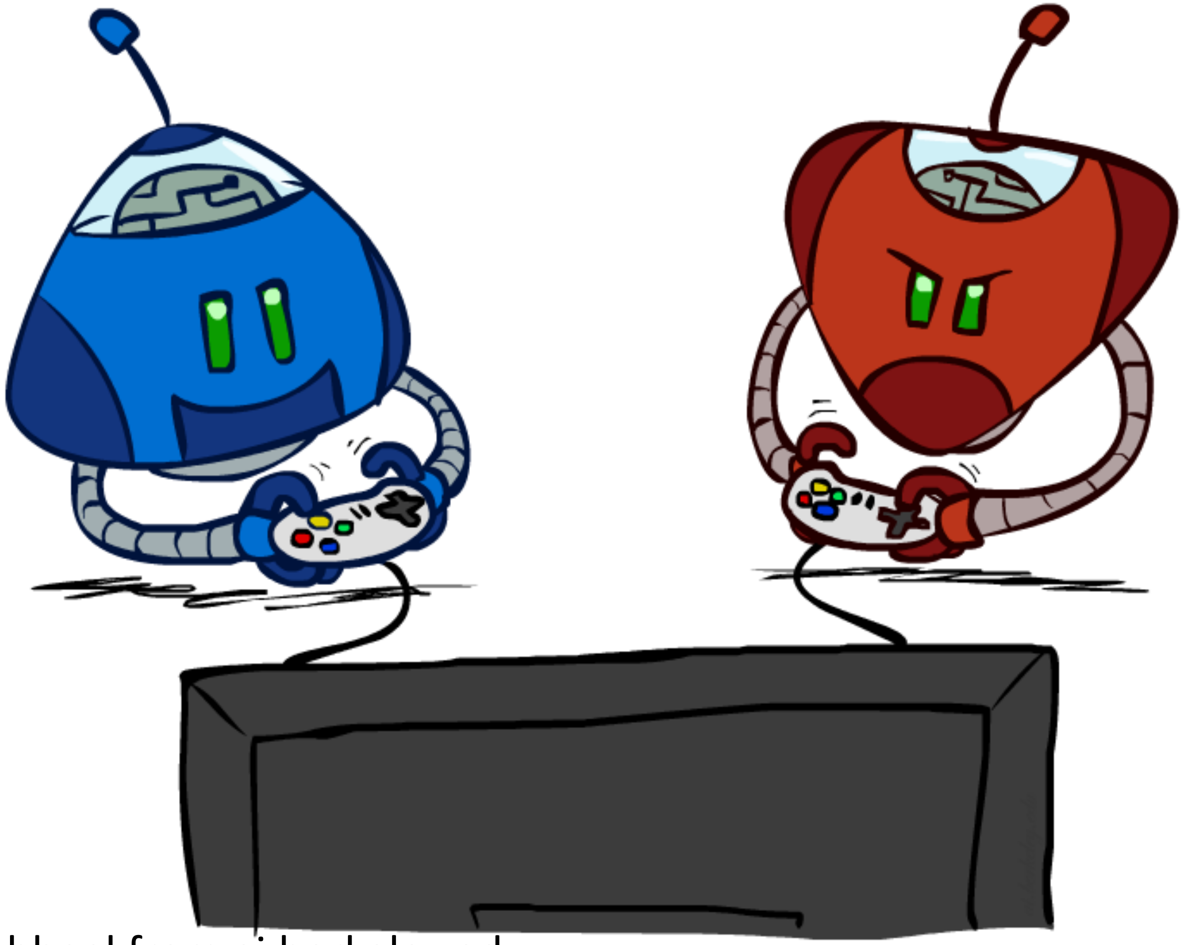
CSEP 573: Artificial Intelligence

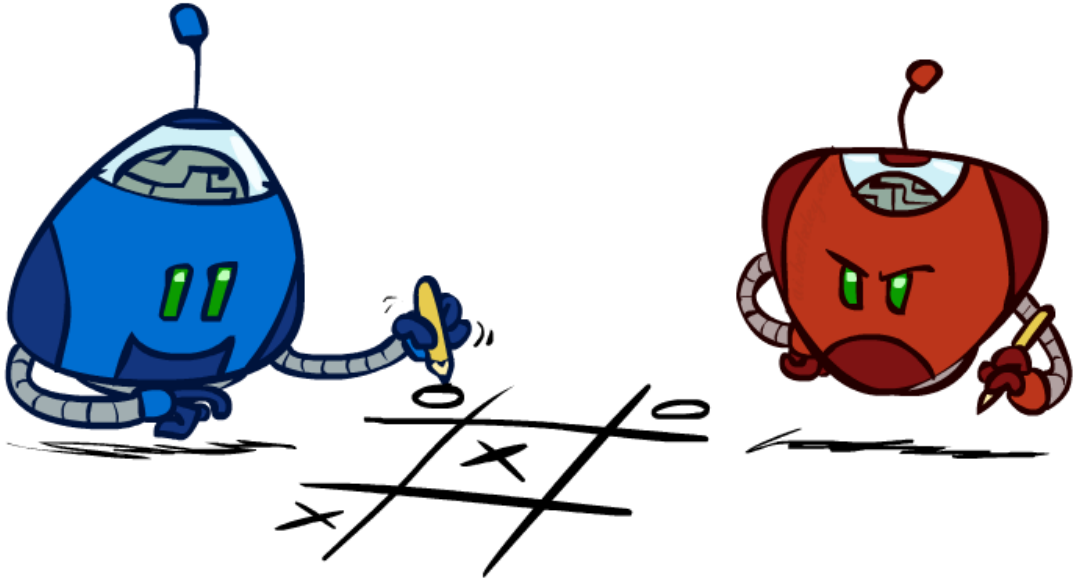
Adversarial Search



slides adapted from
Stuart Russel, Dan Klein, Pieter Abbeel from ai.berkeley.edu
And Hanna Hajishirzi, Jared Moore, Dan Weld

Outline

- History / Overview
- Minimax for Zero-Sum Games
- α-β Pruning
- Games with chance elements



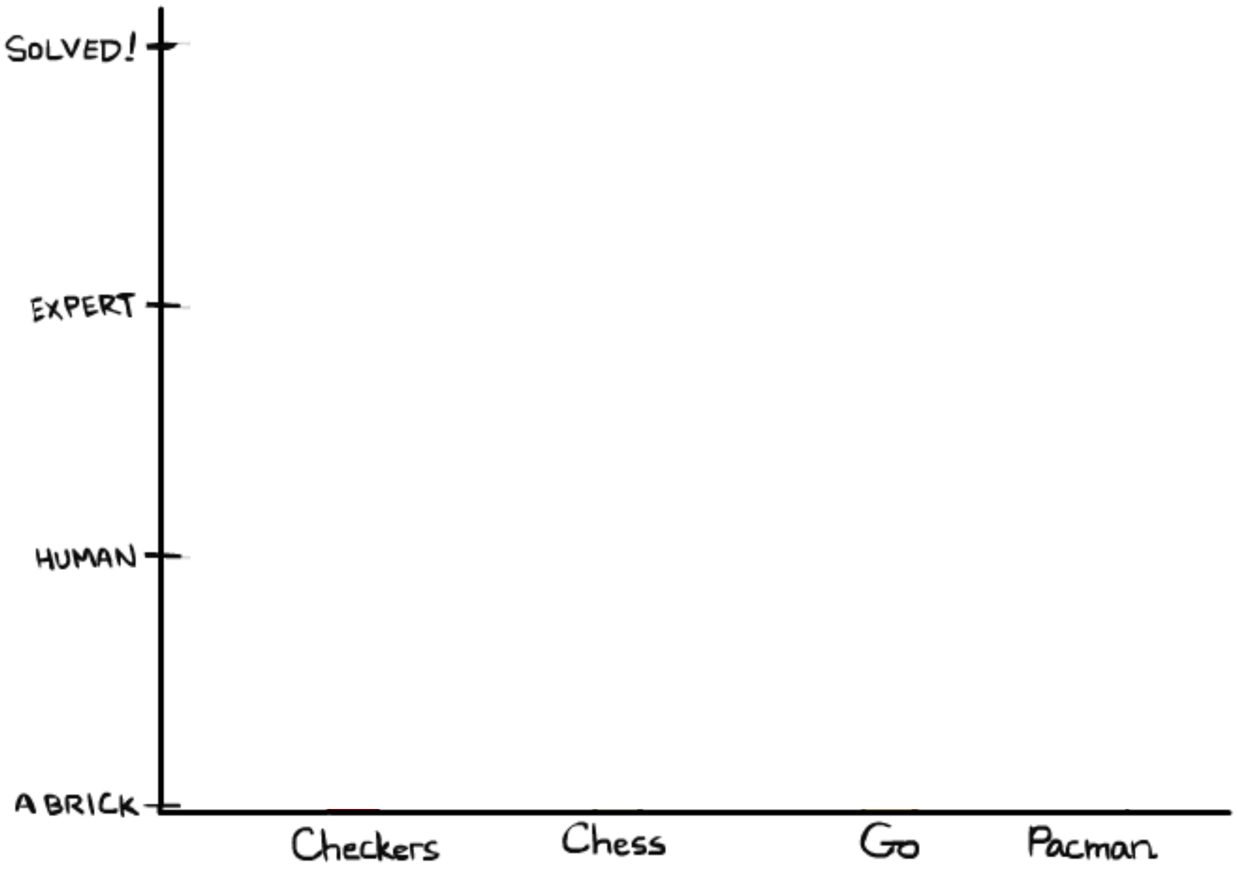
A brief history

- **Checkers:**
  - 1950: First computer player.
  - 1959: Samuel's self-taught program.
  - 1994: First computer world champion: Chinook defeats Tinsley
  - 2007: Checkers solved! Endgame database of 39 trillion states

- **Chess:**
  - 1945-1960: Zuse, Wiener, Shannon, Turing, Newell & Simon, McCarthy.
  - 1960s onward: gradual improvement under "standard model"
  - 1997: Deep Blue defeats human champion Gary Kasparov
  - 2021: Stockfish rating 3551 (vs 2870 for Magnus Carlsen).

- **Go:**
  - 1968: Zobrist's program plays legal Go, barely (b>300!)
  - 1968-2005: various ad hoc approaches tried, novice level
  - 2005-2014: Monte Carlo tree search -> strong amateur
  - 2016-2017: AlphaGo defeats human world champions

- **Pacman**



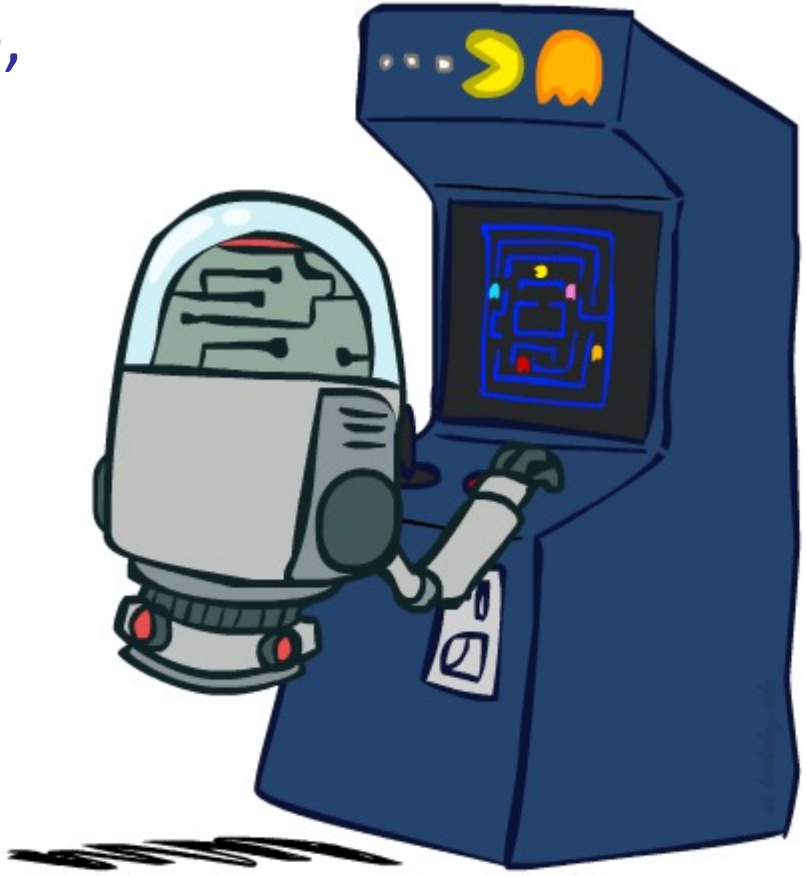
Types of Games

- Game = task environment with > 1 agent
- Axes:
  - Deterministic or stochastic?
  - Perfect information (fully observable)?
  - One, two, or more players?
  - Turn-taking or simultaneous?
  - Zero sum?

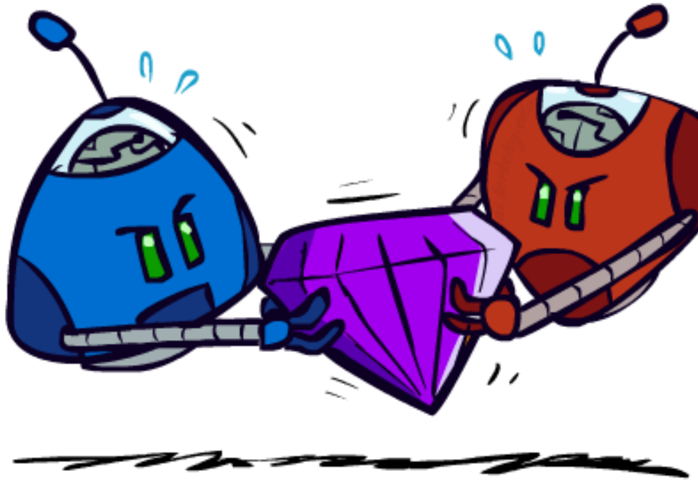
- Want algorithms for calculating a ***contingent plan*** (a.k.a. strategy or policy) which recommends a move for every possible eventuality

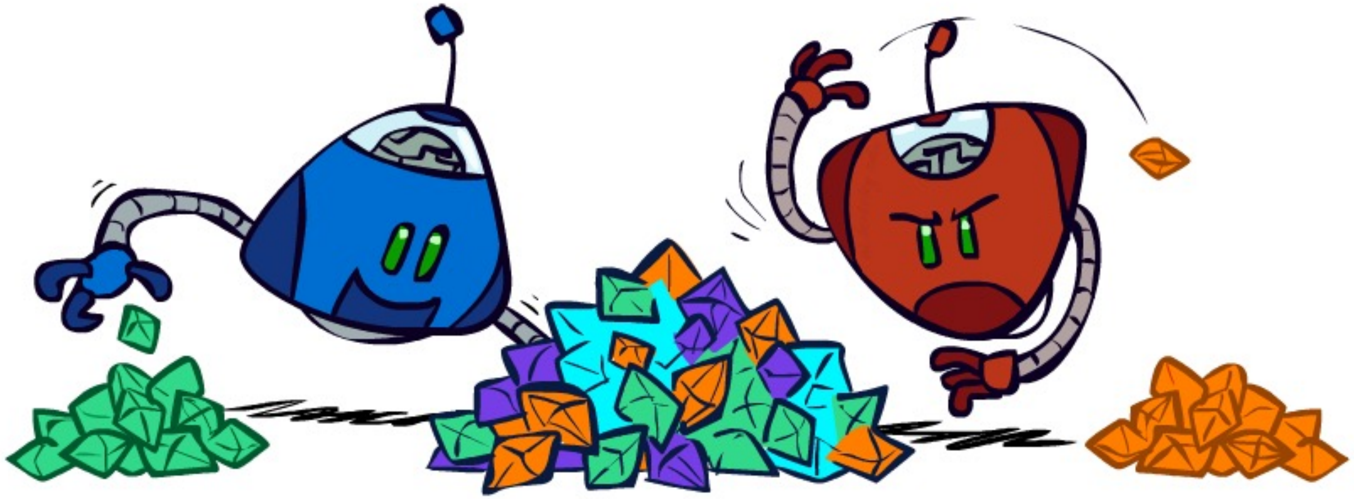
“Standard” Games

- Standard games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
  - Initial state: s_0
  - Players: Player(s) indicates whose move it is
  - Actions: Actions(s) for player on move
  - Transition model: Result(s,a)
  - Terminal (goal) test: Terminal-Test(s)
  - Terminal values: Utility(s,p) for player p
    - Or just Utility(s) for player making the decision at root



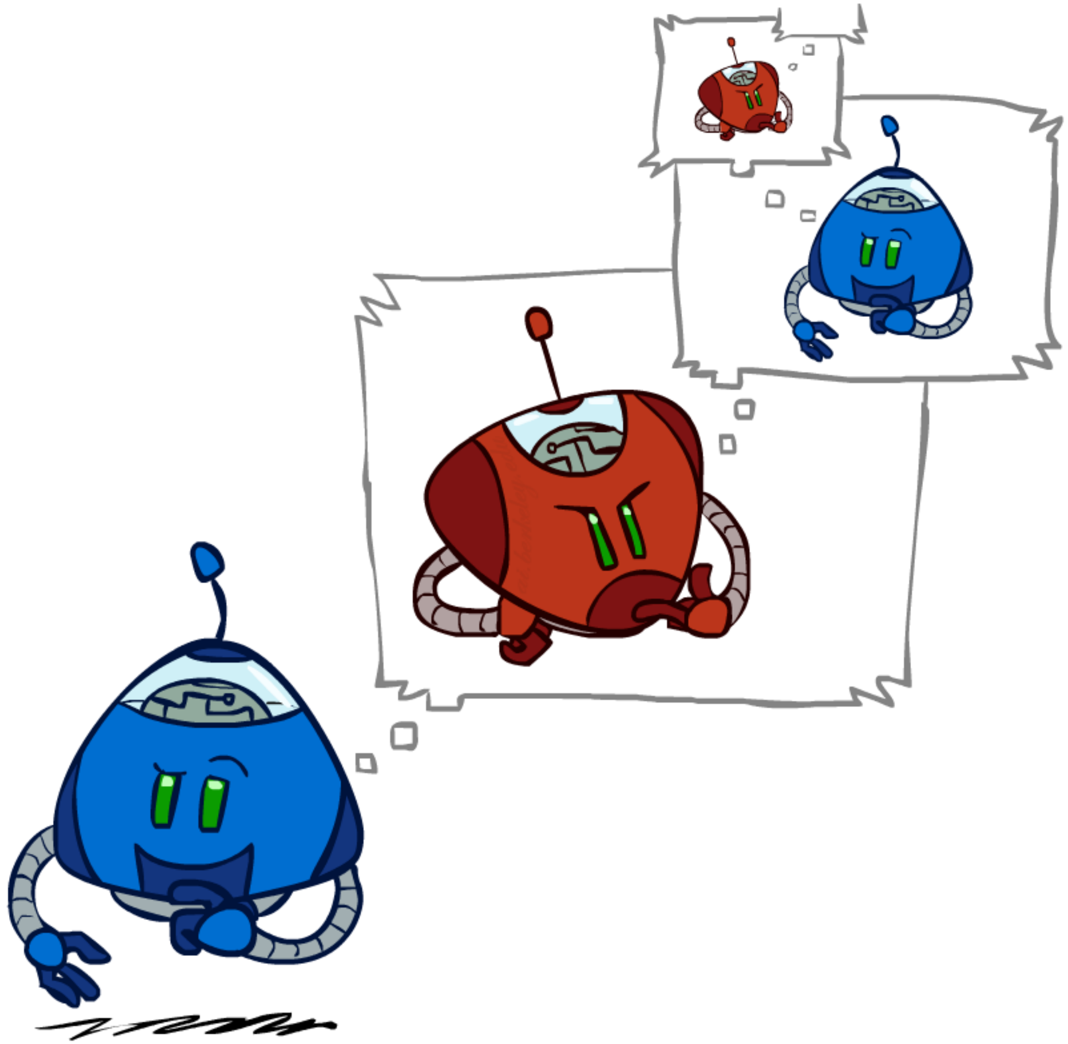
Zero-Sum Games



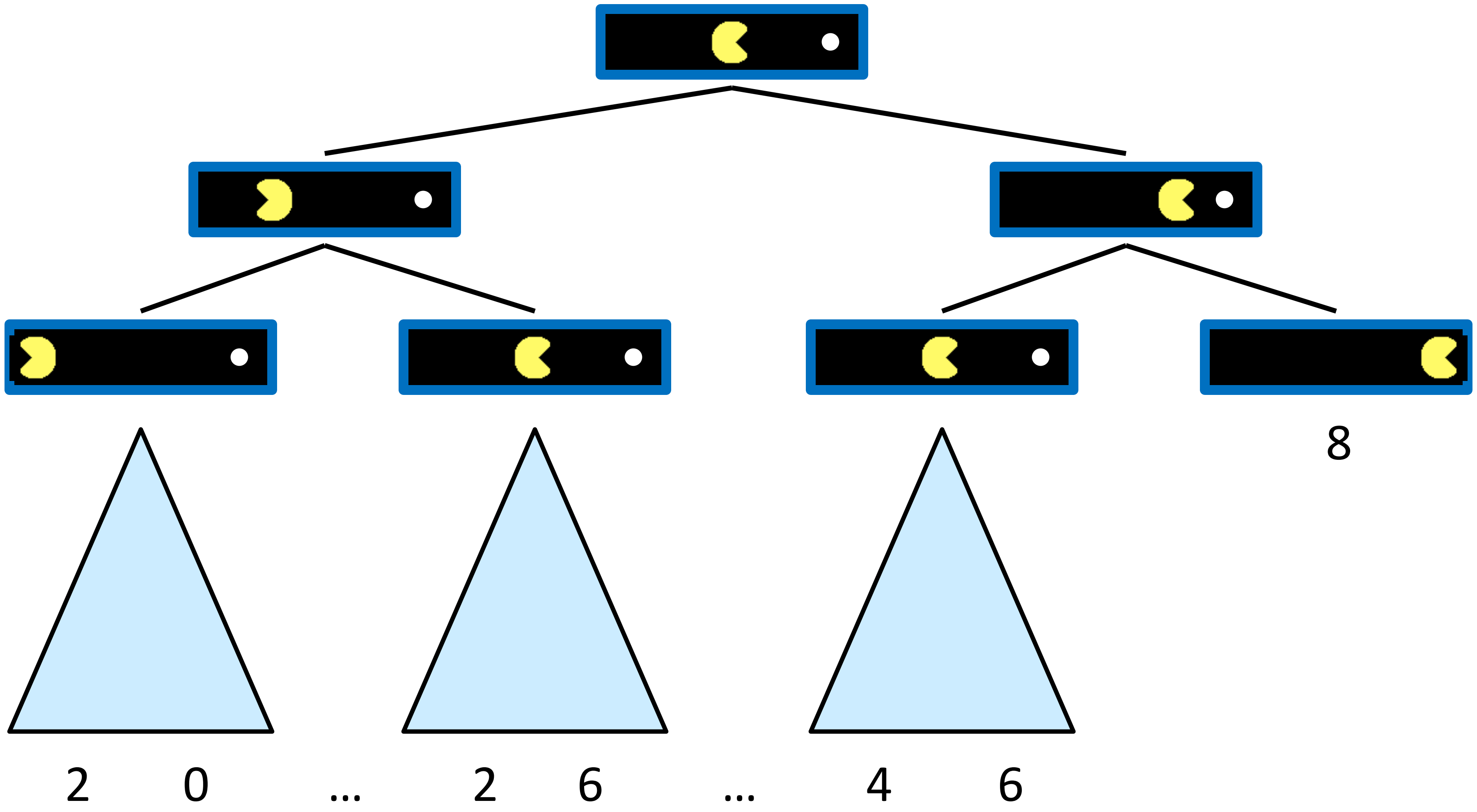


- Zero-Sum Games
  - Agents have ***opposite*** utilities
  - Pure competition: what is better for one player is worse for the other
- General Games
  - Agents have ***independent*** utilities
  - Cooperation, indifference, competition, shifting alliances, and more are all possible

Adversarial Search



Single-Agent Trees

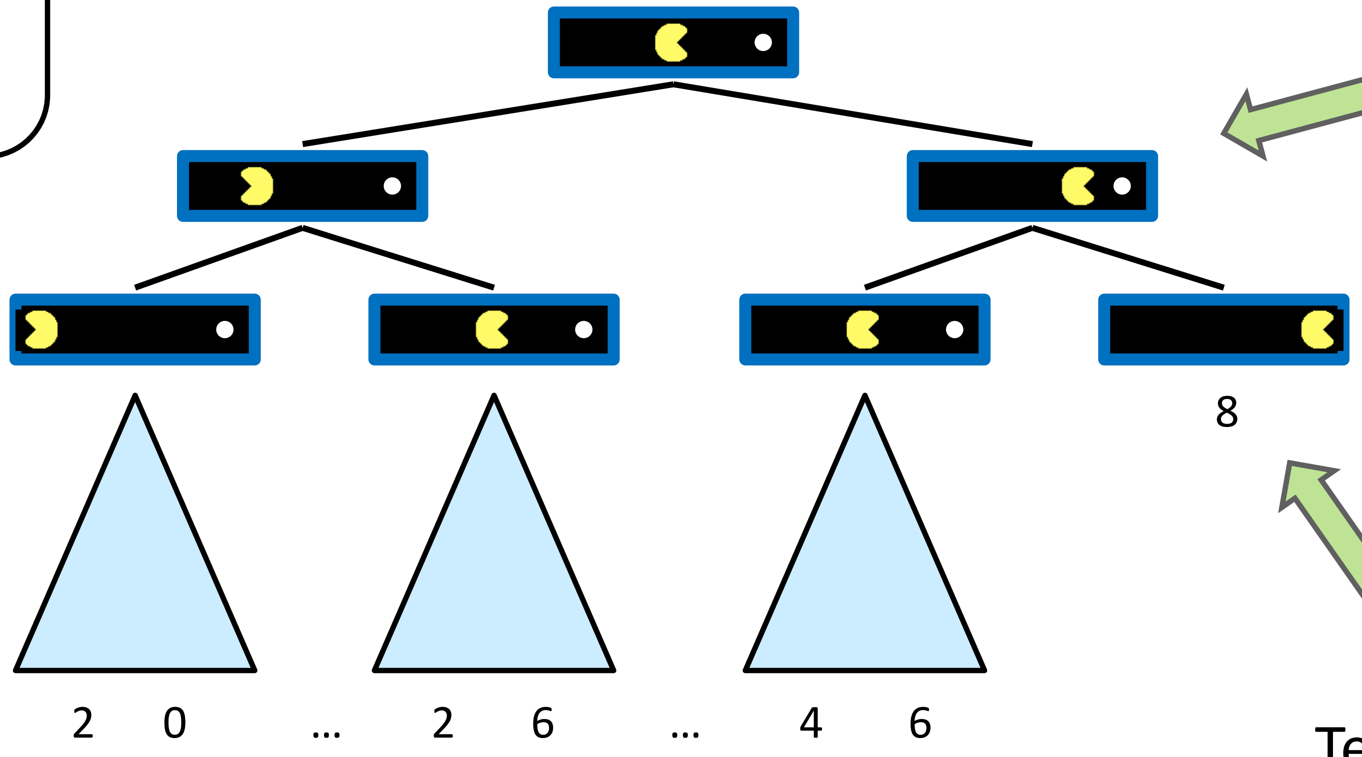


Utility (value) of a State

Utility of a state:
The best achievable outcome (value) from that state

Non-Terminal States:

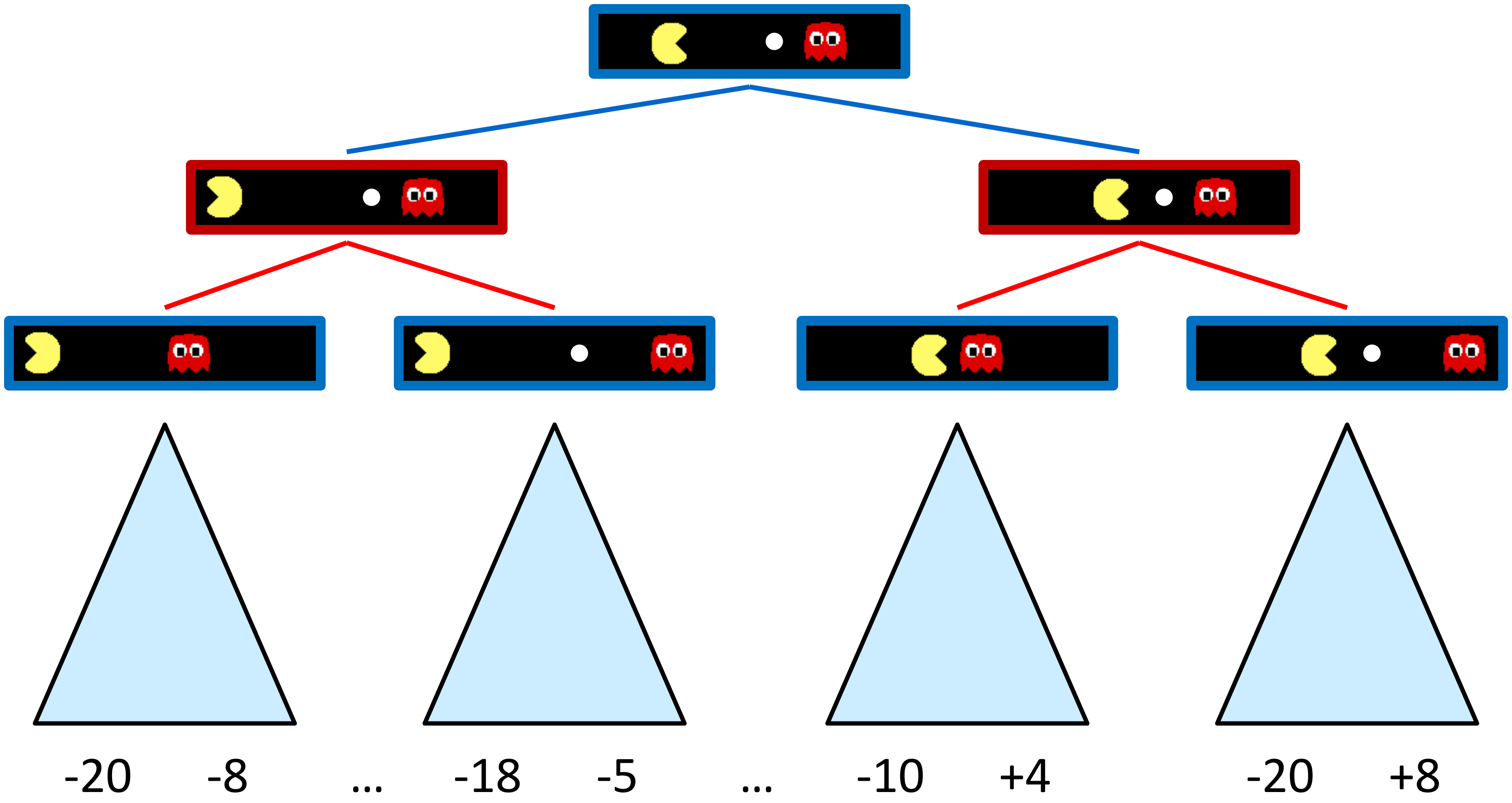
$$U(s) = \max_{s' \in \text{successors}(s)} U(s')$$



Terminal States:

$U(s)$ = known

Adversarial Game Trees



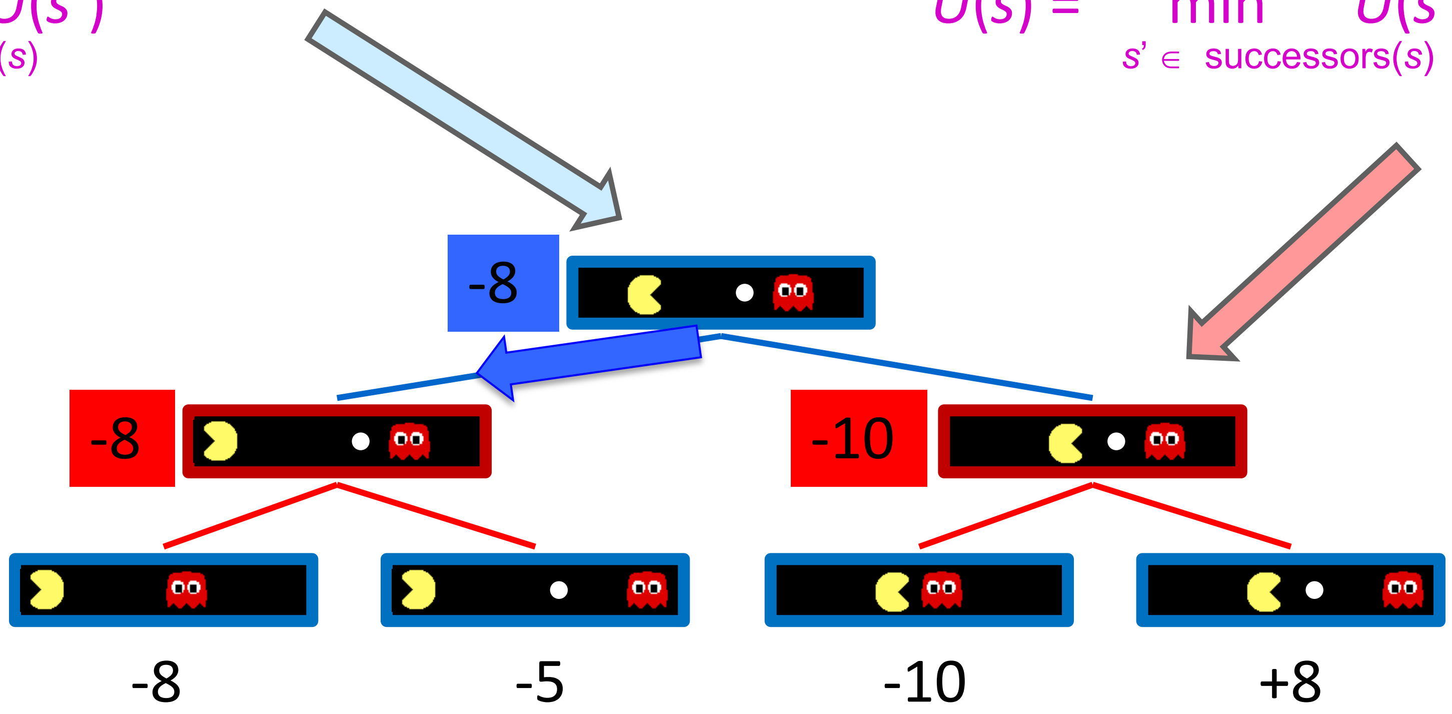
Minimax Values

MAX nodes: under Agent's control

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

MIN nodes: under Opponent's control

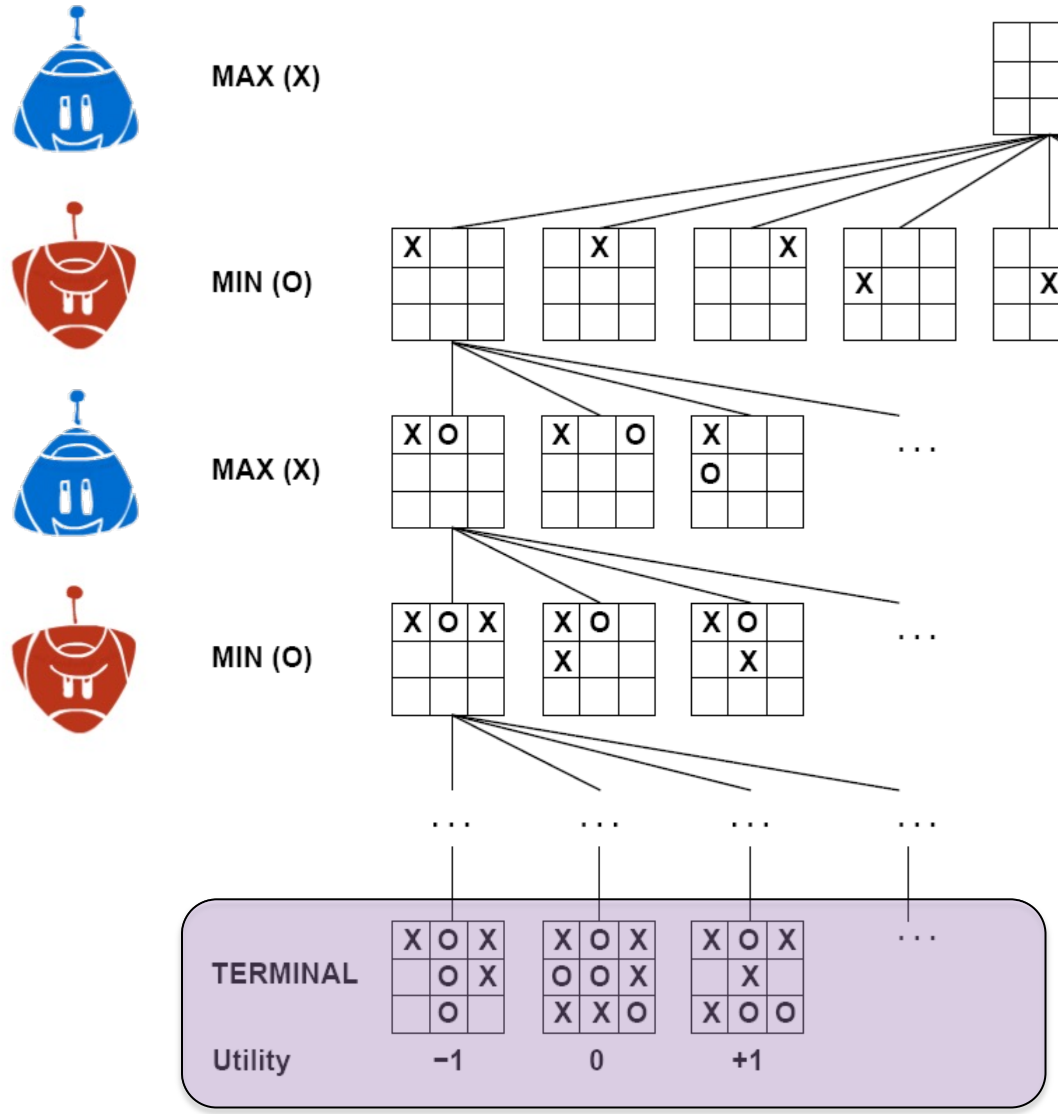
$$U(s) = \min_{s' \in \text{successors}(s)} U(s')$$

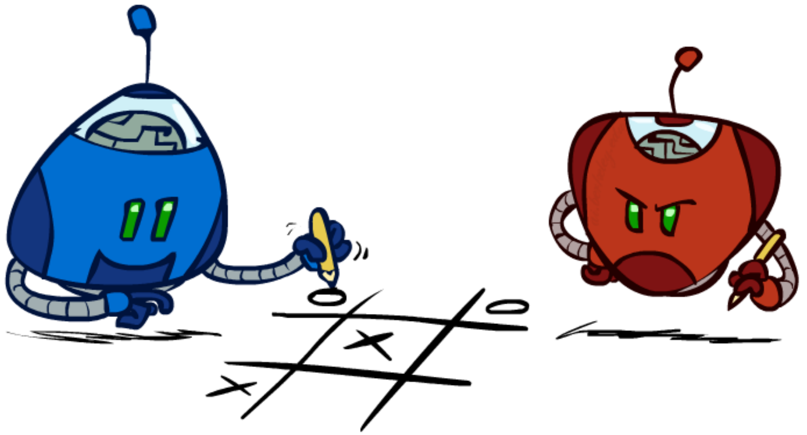


Terminal States:

$U(s)$ = known

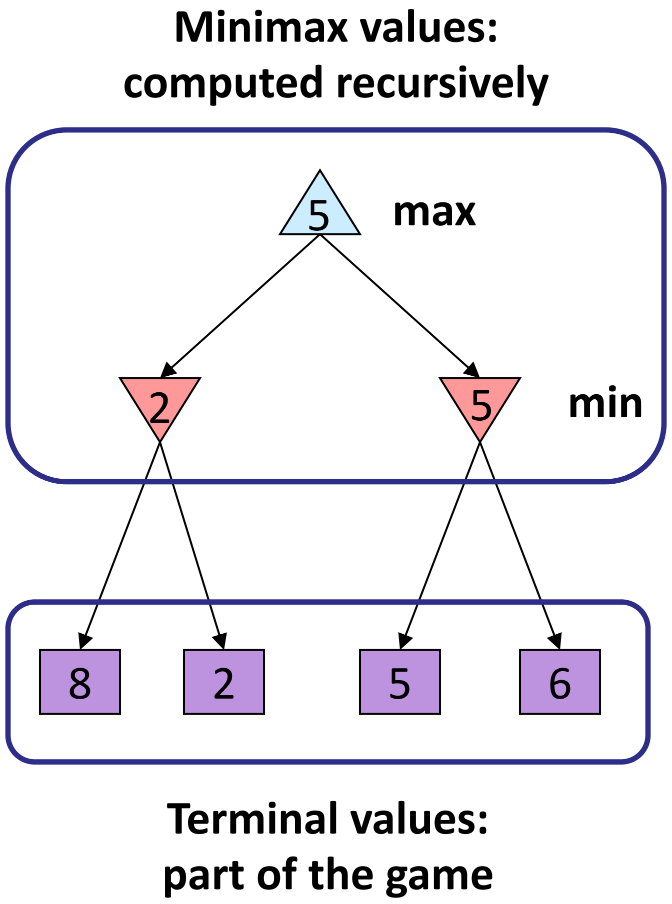
Tic-Tac-Toe Game Tree





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



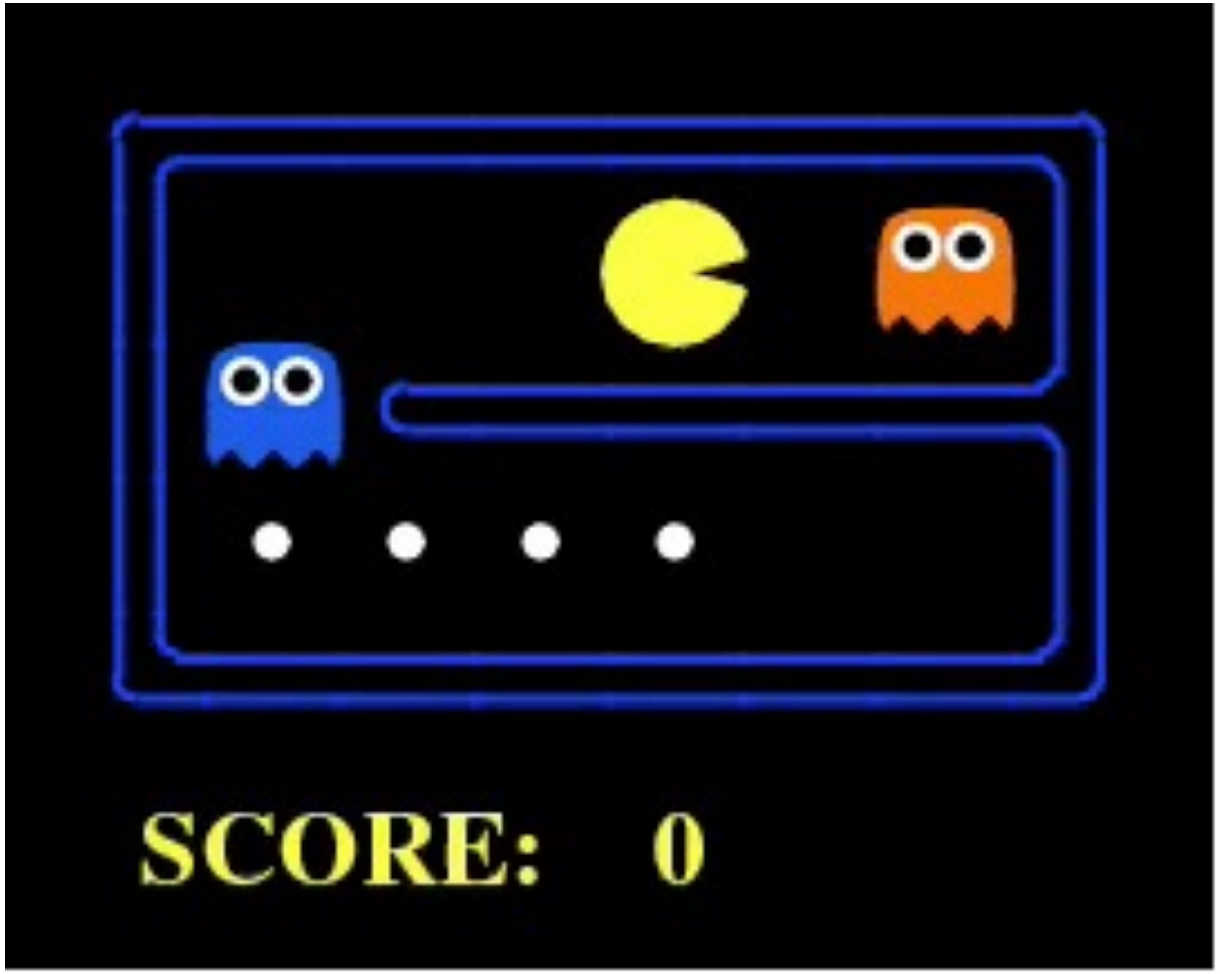
Implementation

```
function minimax-decision(s) returns an action
    return the action a in Actions(s) with the highest
     minimax_value(Result(s,a))
```

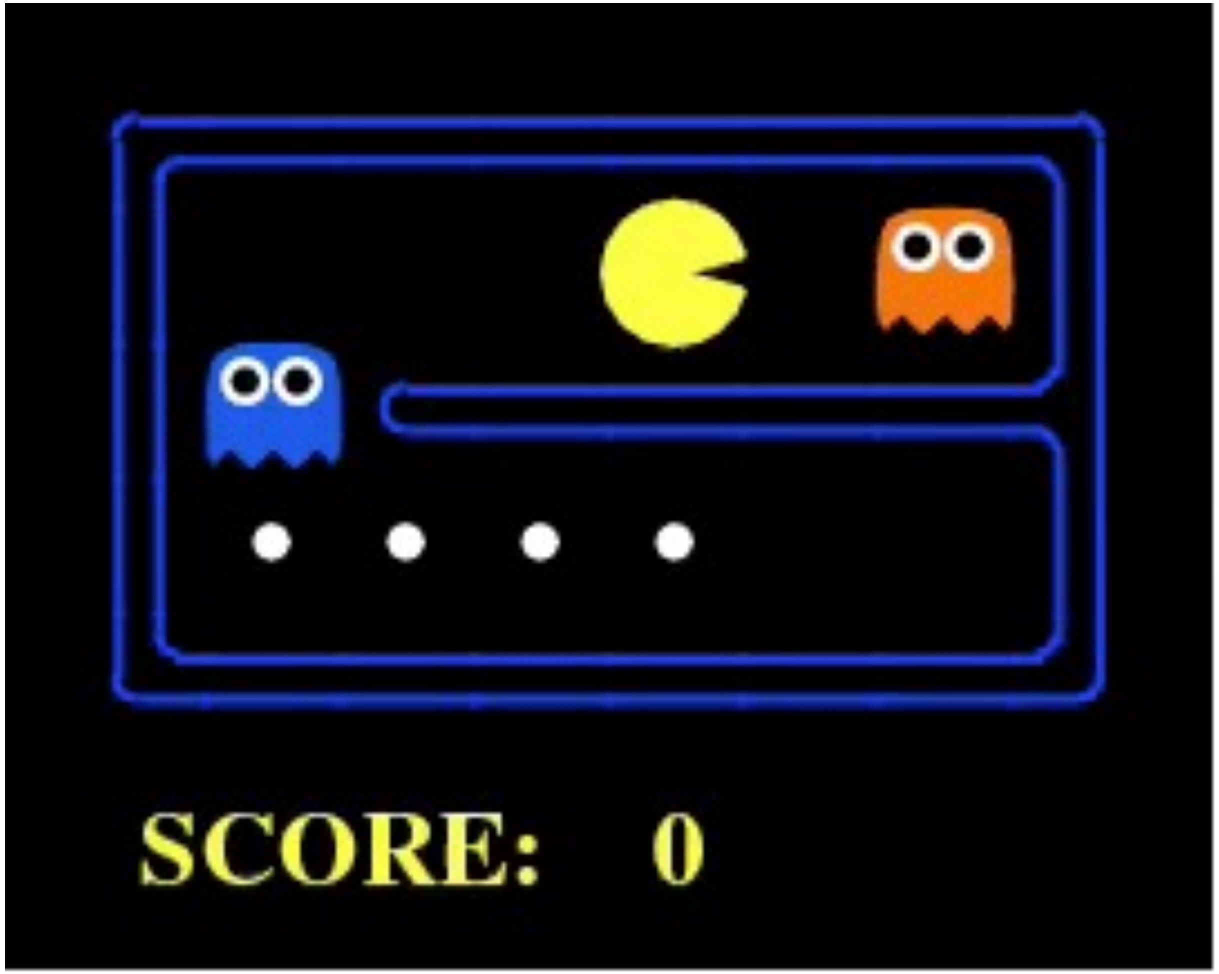
⇕

```
function minimax_value(s) returns a value
    if Terminal-Test(s) then return Utility(s)
    if Player(s) = MAX then return max_{a in Actions(s)} minimax_value(Result(s,a))
    if Player(s) = MIN then return min_{a in Actions(s)} minimax_value(Result(s,a))
```

Video of Demo Min vs. Exp (Min)



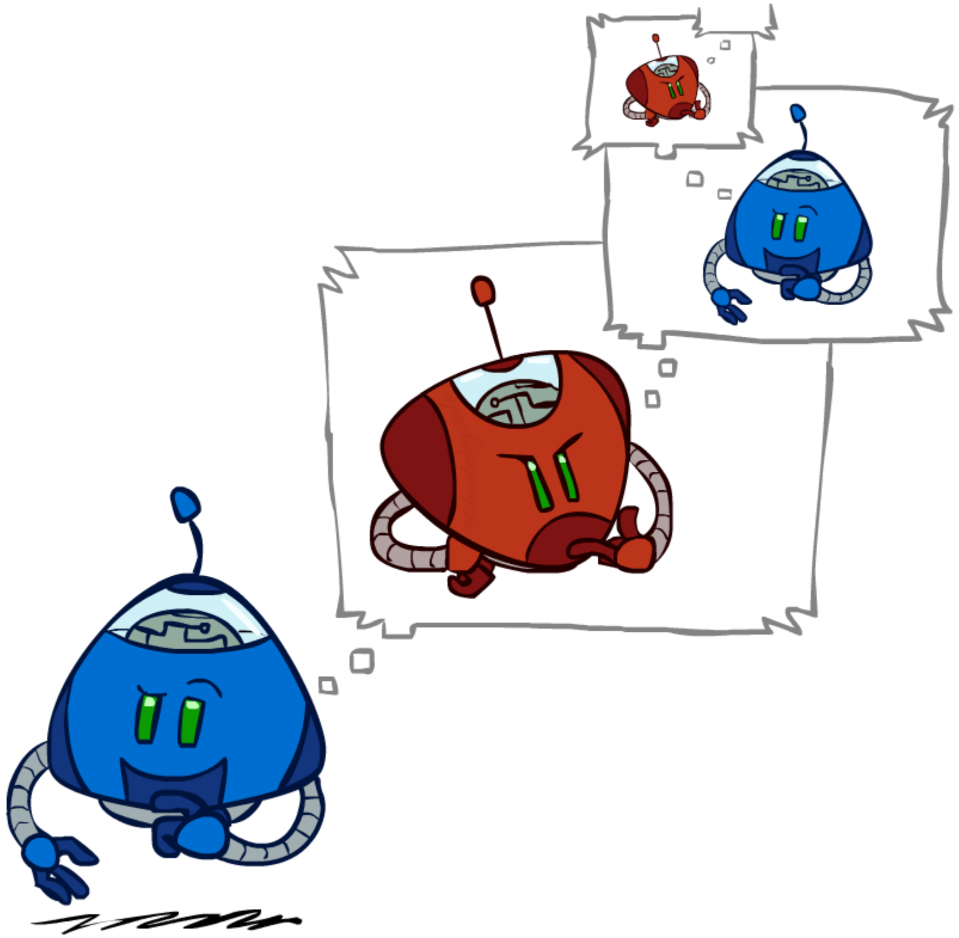
Video of Demo Min vs. Exp (Exp)



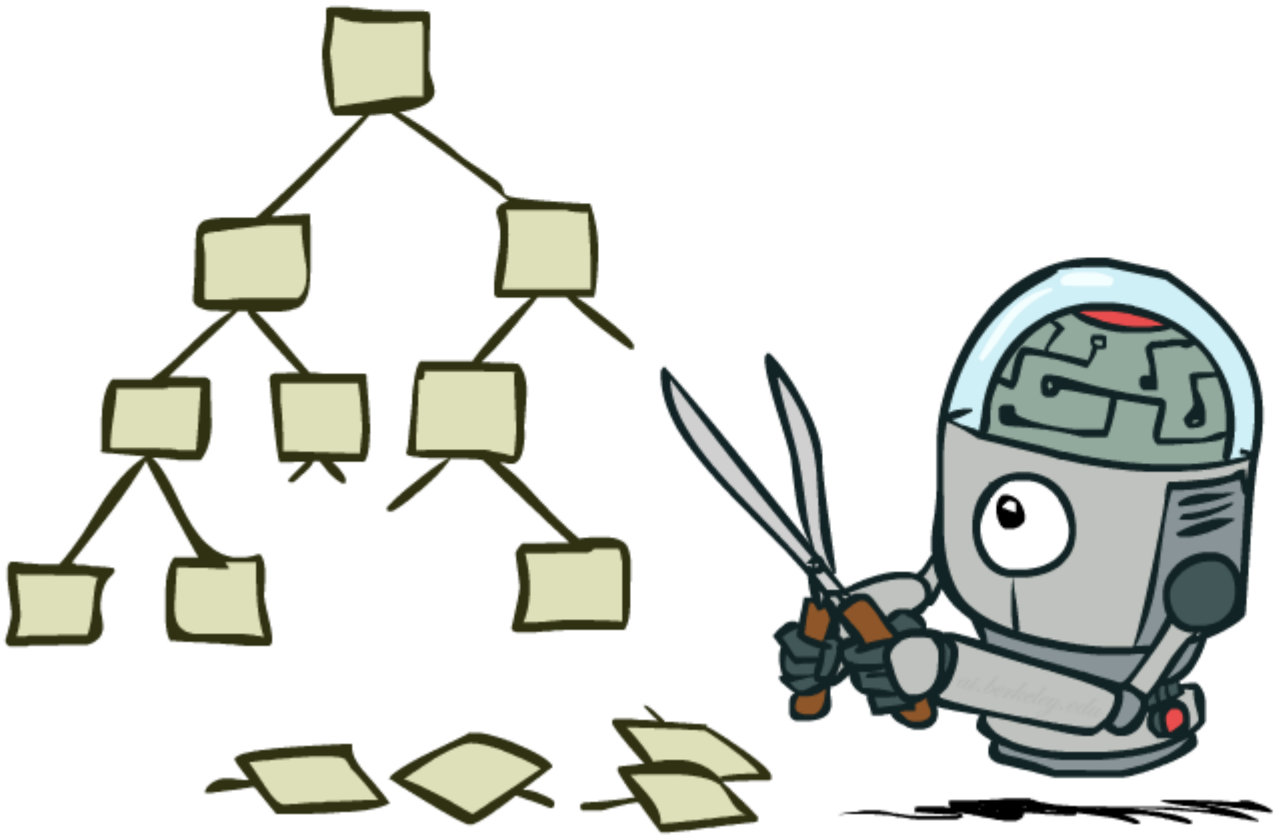
Minimax Efficiency

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

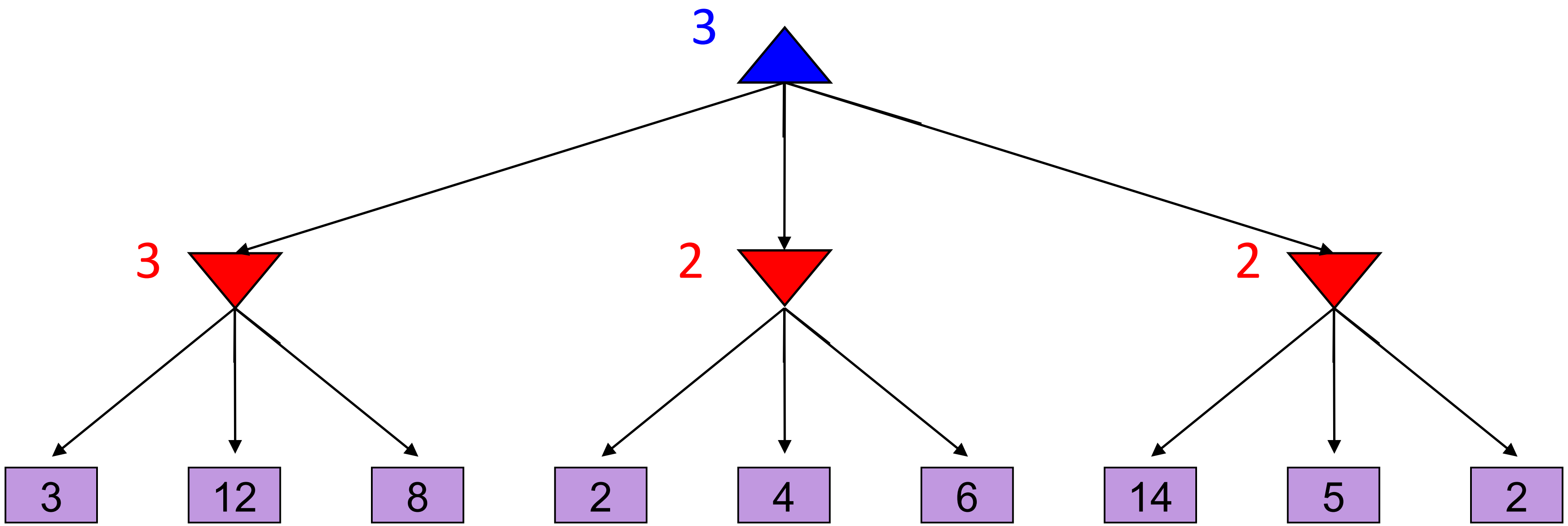
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - Humans can't do this either, so how do we play chess?



Game Tree Pruning

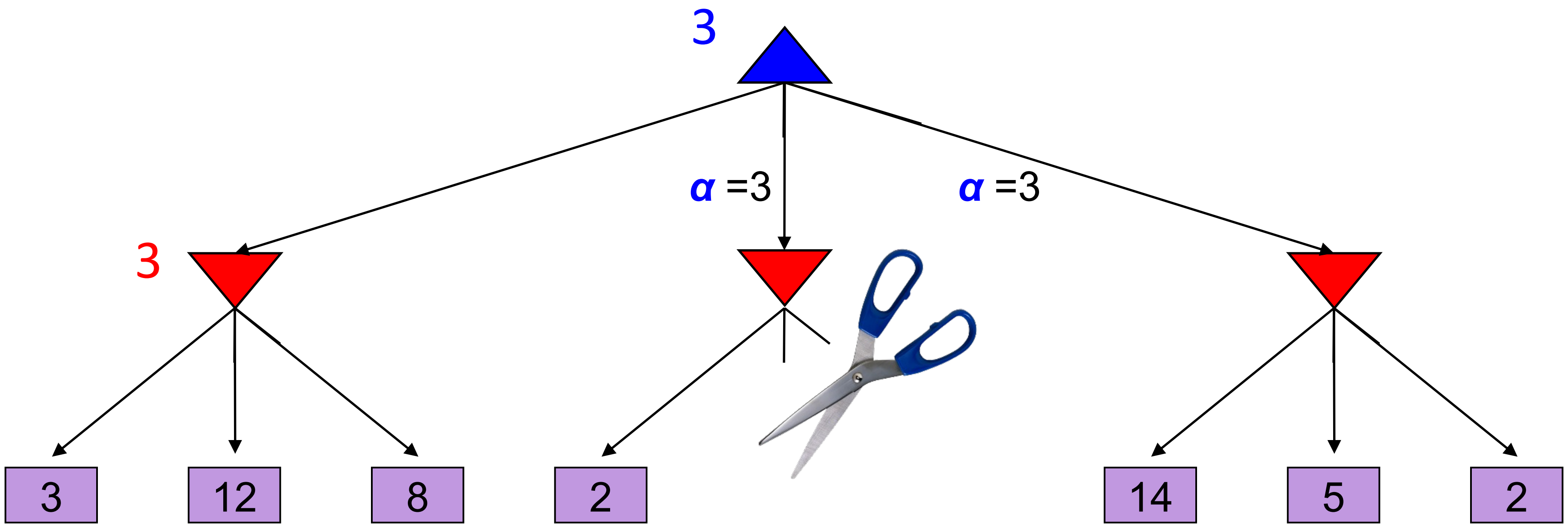


Minimax Example



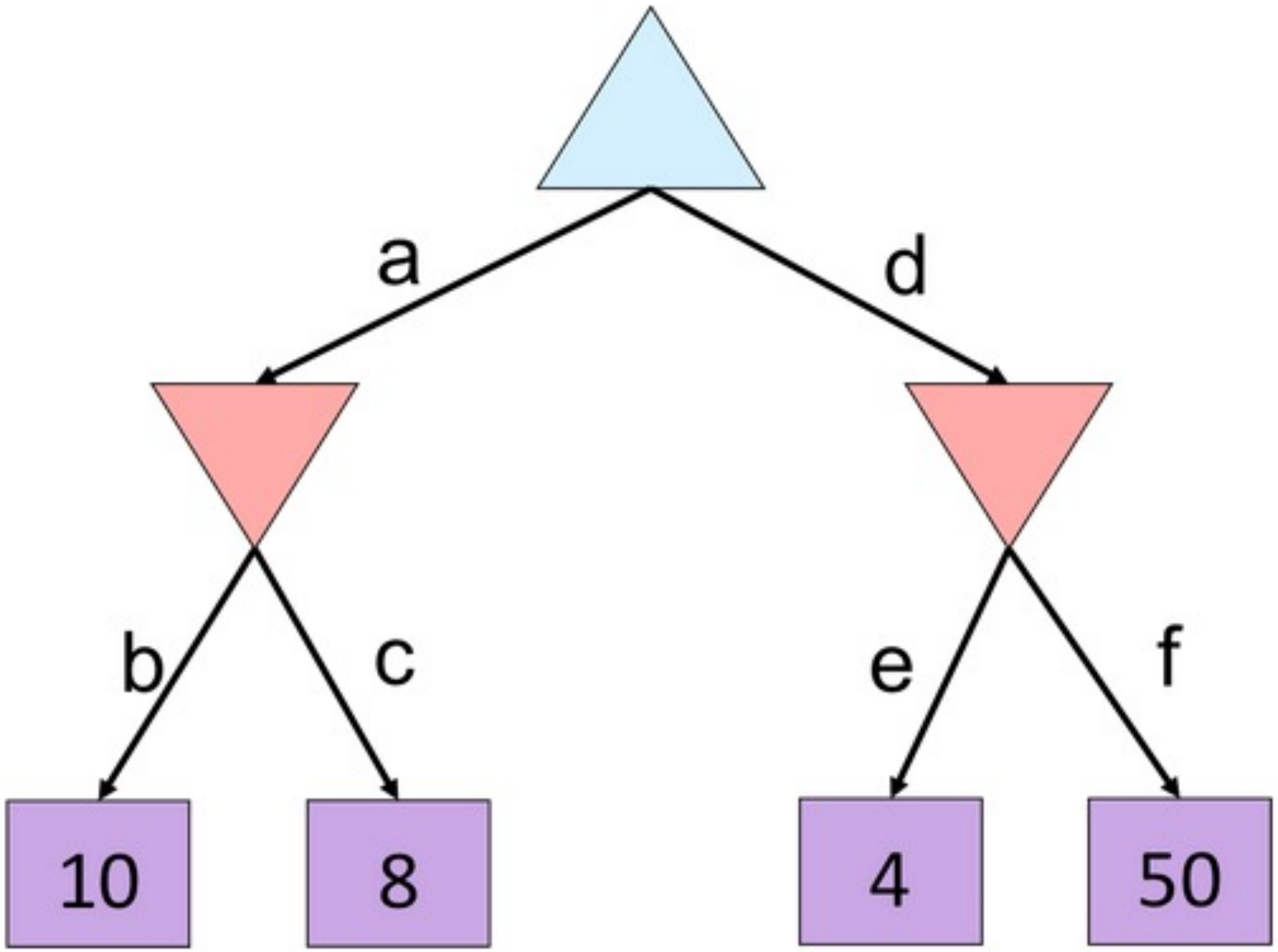
Alpha-Beta Example

α = best option so far from any MAX node on this path

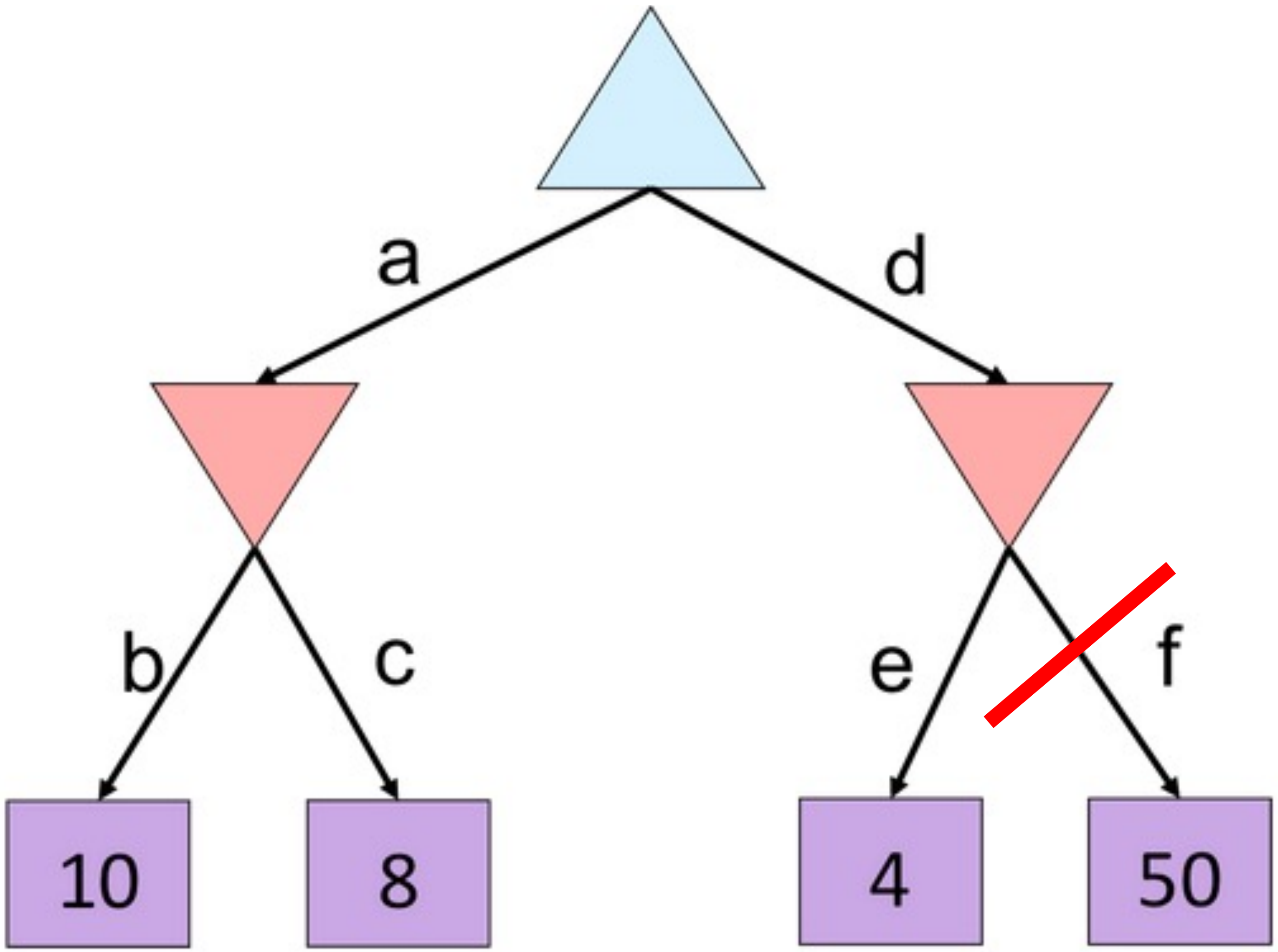


The order of generation matters: more pruning is possible if good moves come first

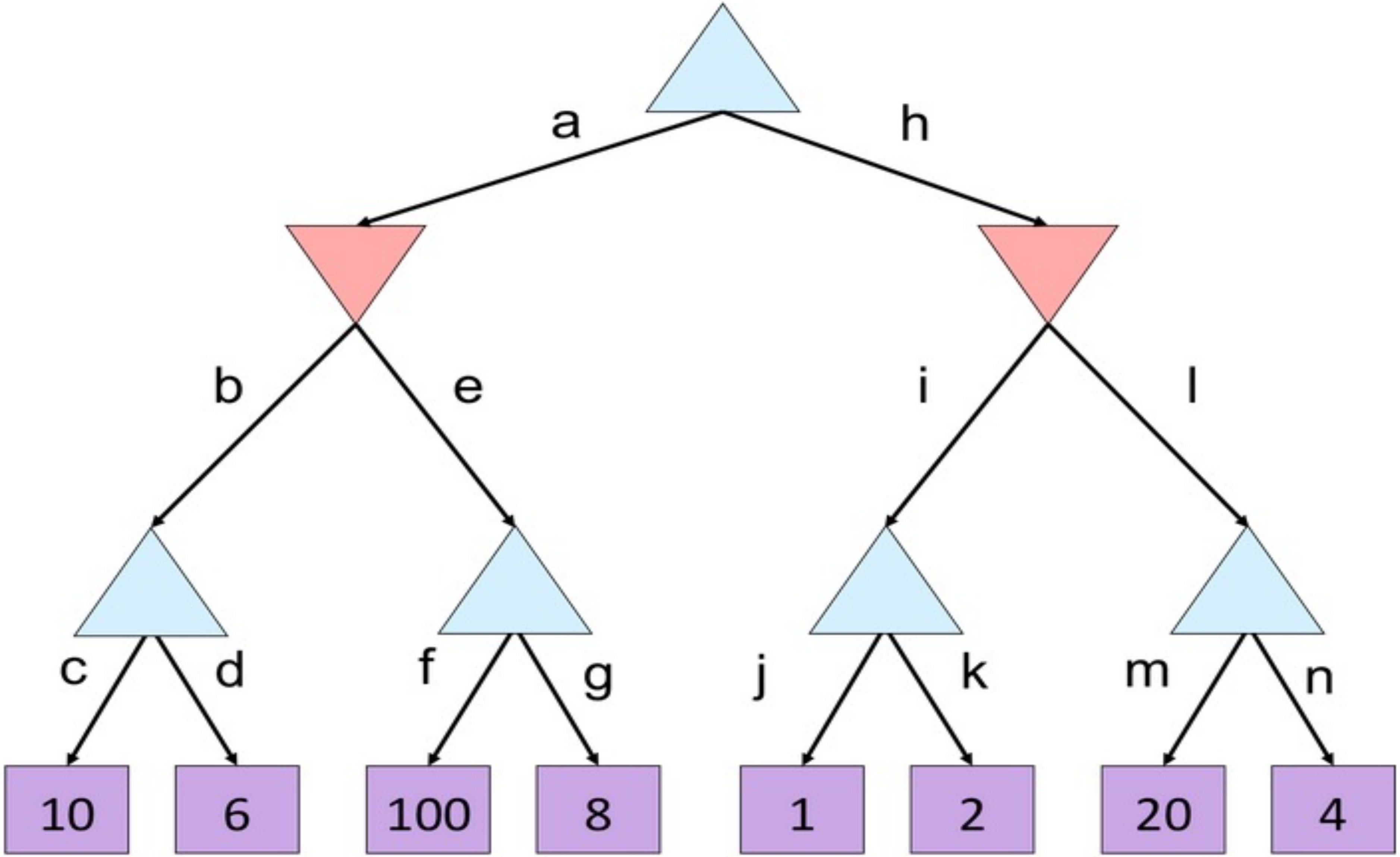
Alpha-Beta Quiz



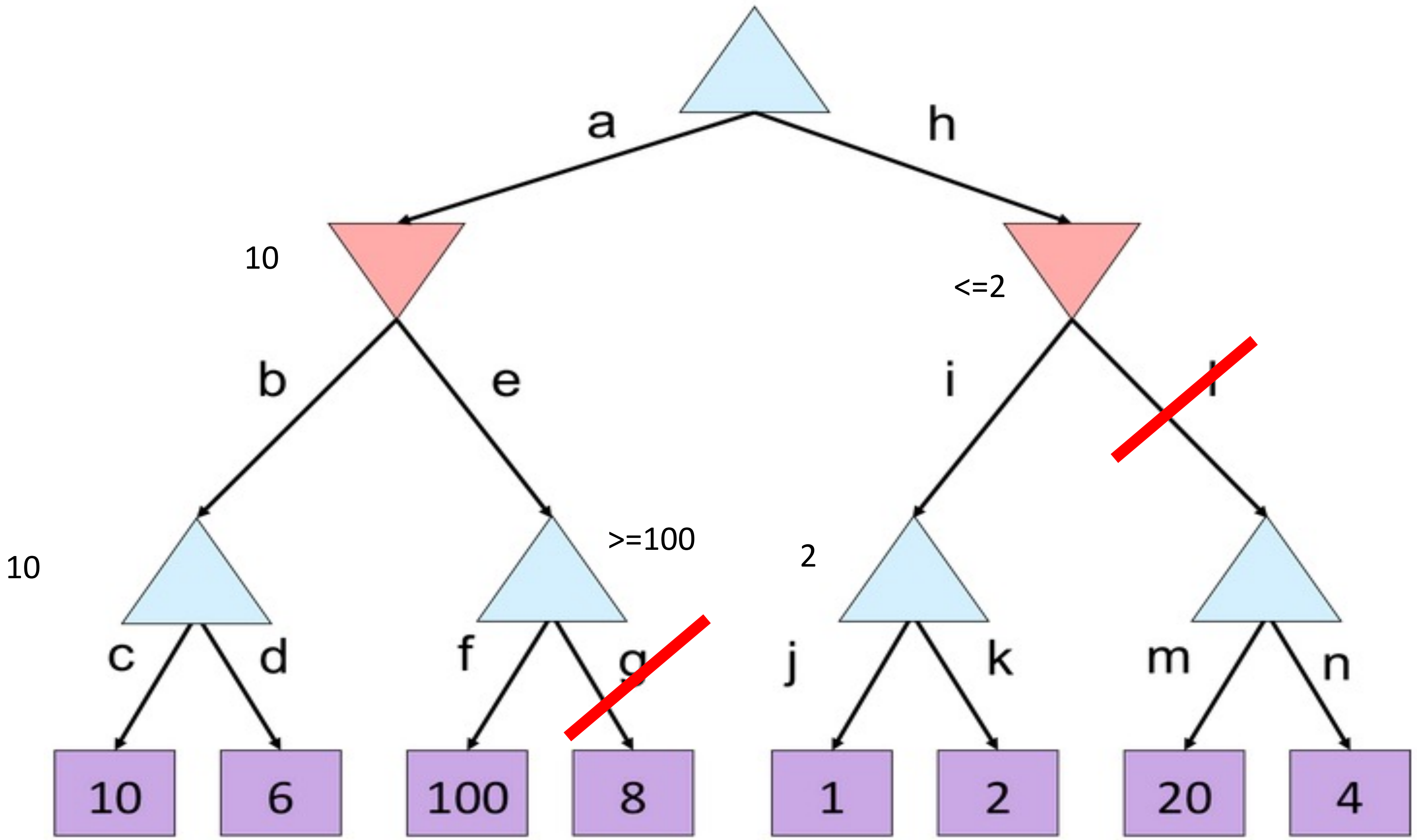
Alpha-Beta Quiz



Alpha-Beta Quiz 2

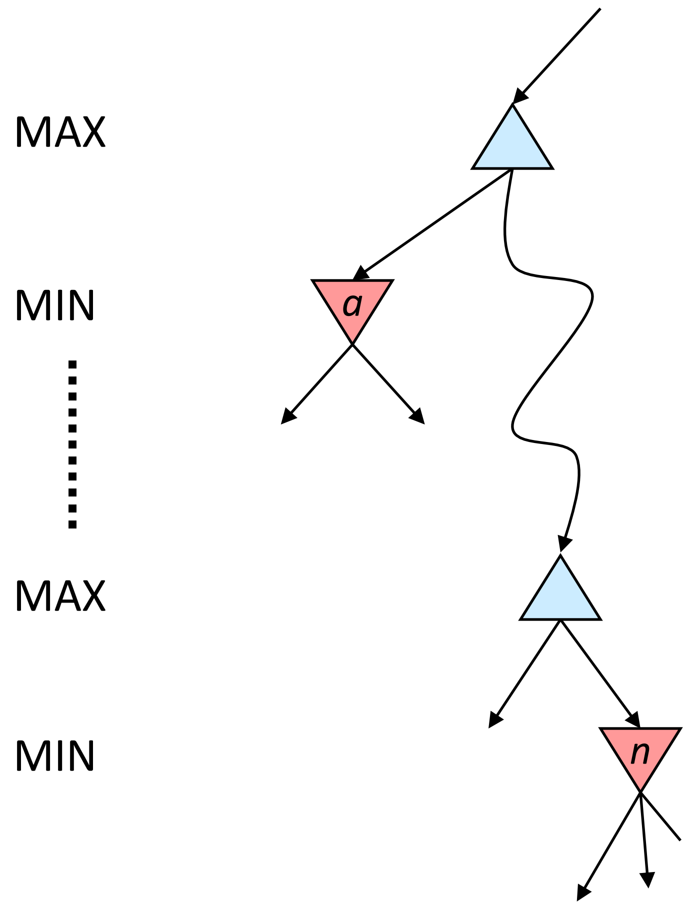


Alpha-Beta Quiz 2



Alpha-Beta Pruning

- General case (pruning children of MIN node)
  - 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 ***α*** be the best value that MAX can get so far at any choice point along the current path from the root
  - If *n* becomes worse than ***α***, MAX will avoid it, so we can prune *n*'s other children (it's already bad enough that it won't be played)
- Pruning children of MAX node is symmetric
  - Let **β** be the best value that MIN can get so far at any choice point along the current path from the root



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, min-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, max-value(successor, α, β))
        if v ≤ α
            return v
        β = min(β, v)
    return v
```

Alpha-Beta Implementation

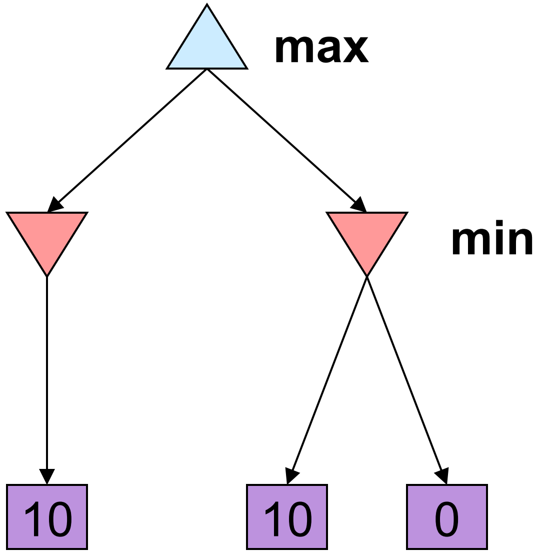
```
function minimax-decision(s) returns an action
    return the action a in Actions(s) with the highest
     max-value(Result(s,a), -∞, +∞)
```

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

Alpha-Beta Pruning Properties

- Theorem: This pruning has ***no effect*** on minimax value computed for the root!



- Good child ordering improves effectiveness of pruning
  - Iterative deepening helps with this

- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
    - Square root!
  - Doubles solvable depth!

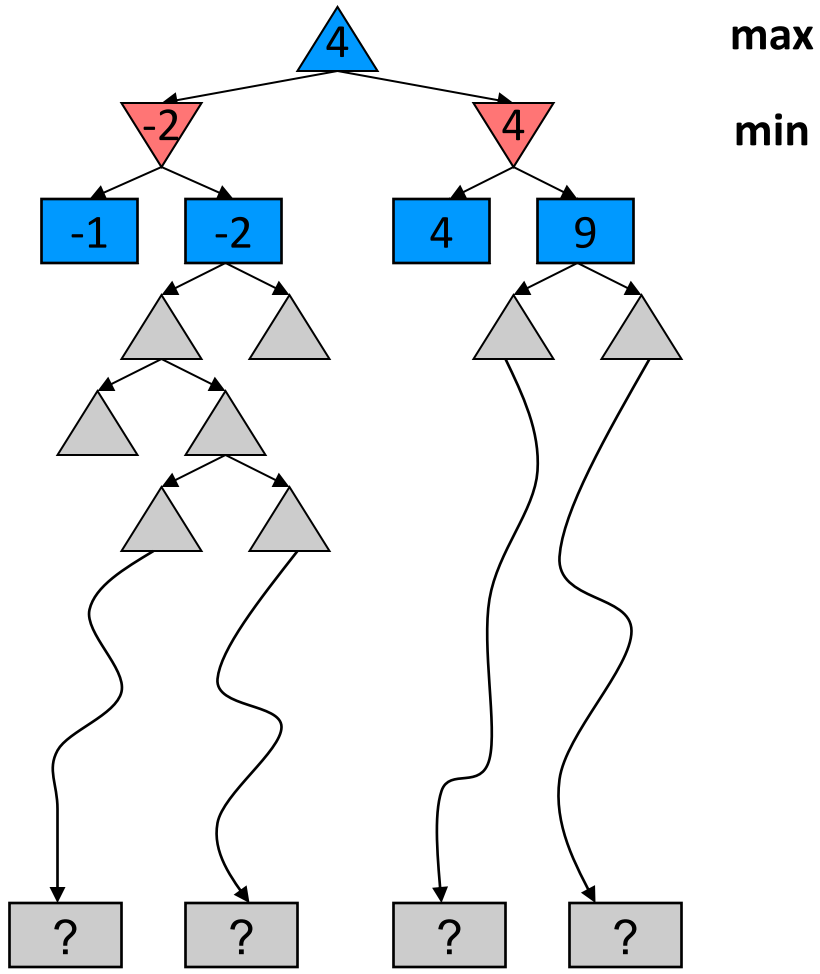
- This is a simple example of metareasoning (reasoning about reasoning)

- For chess: only 35^{50} instead of 35^{100}! Yay!

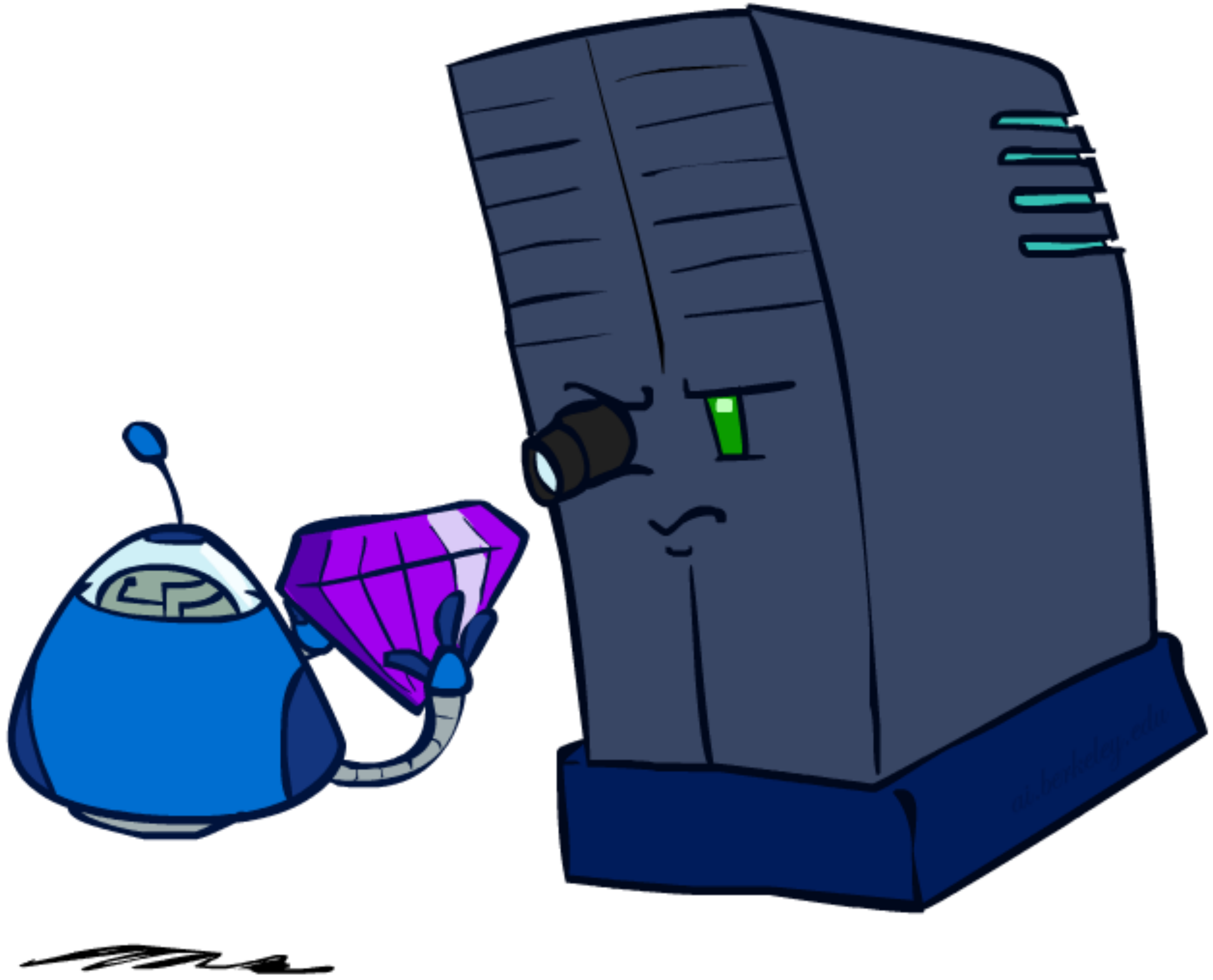
Resource Limits

Resource Limits

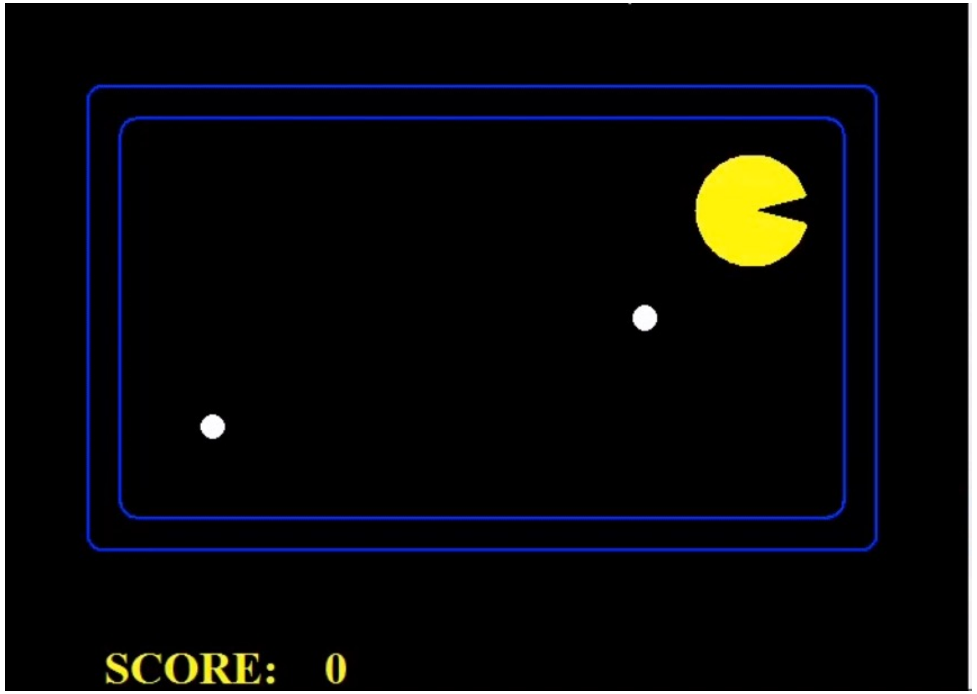
- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - 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



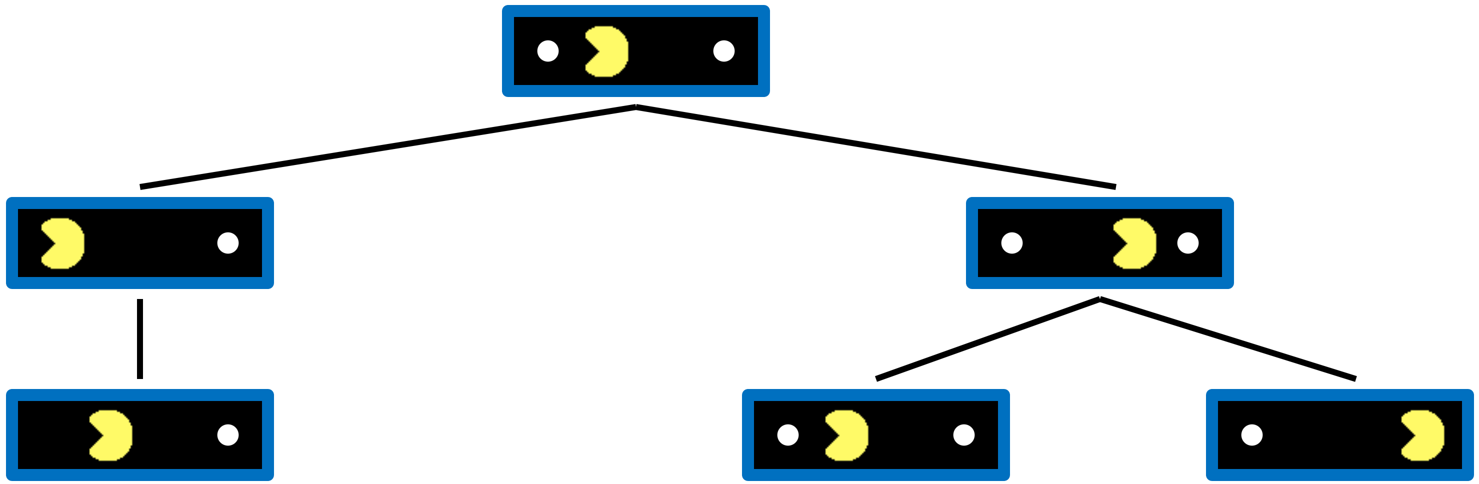
Evaluation Functions



Video of Demo Thrashing (d=2)



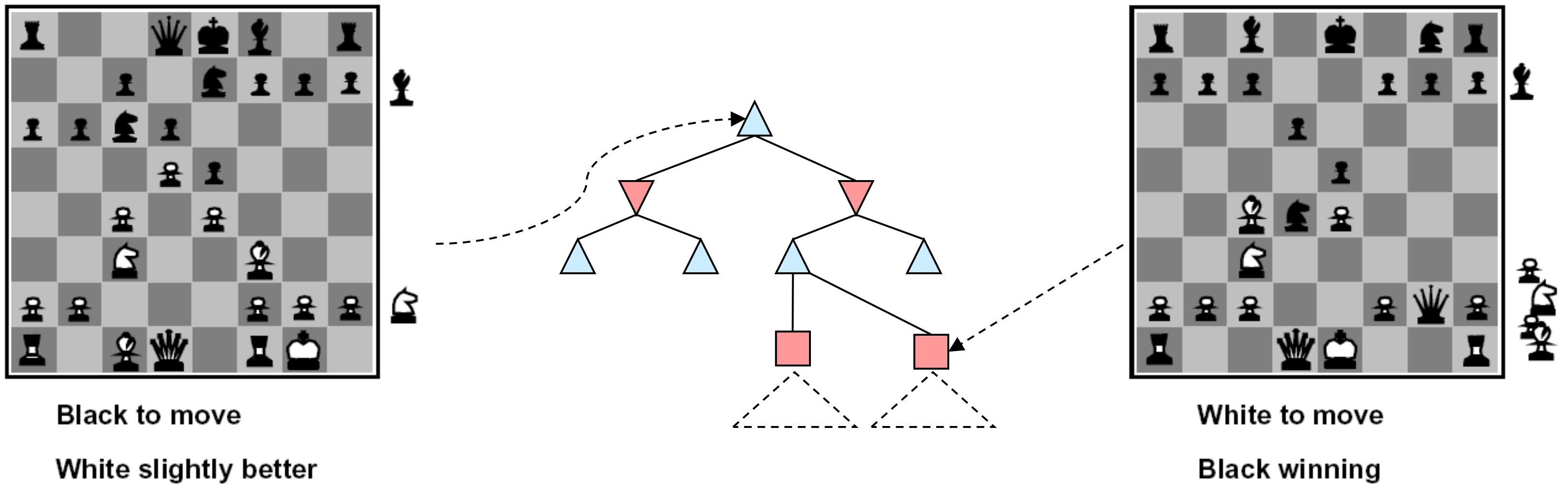
Why Pacman Starves



- A danger of replanning 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!

Evaluation Functions

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

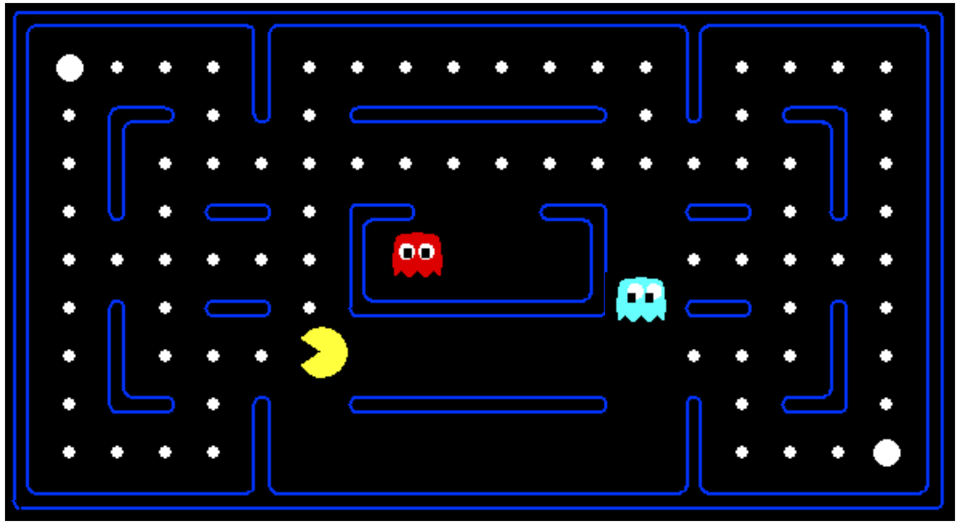


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

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

Evaluation for Pacman

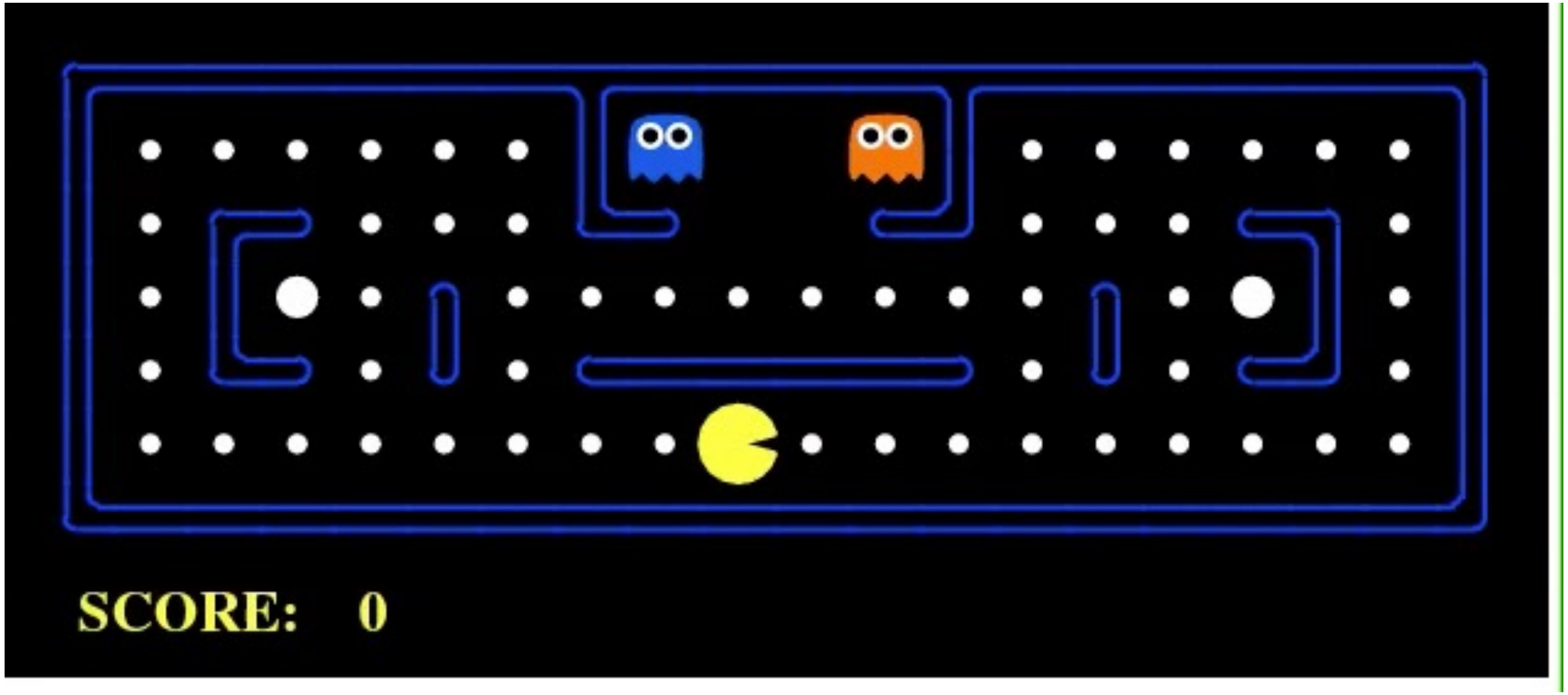


What features would be good for Pacman?

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

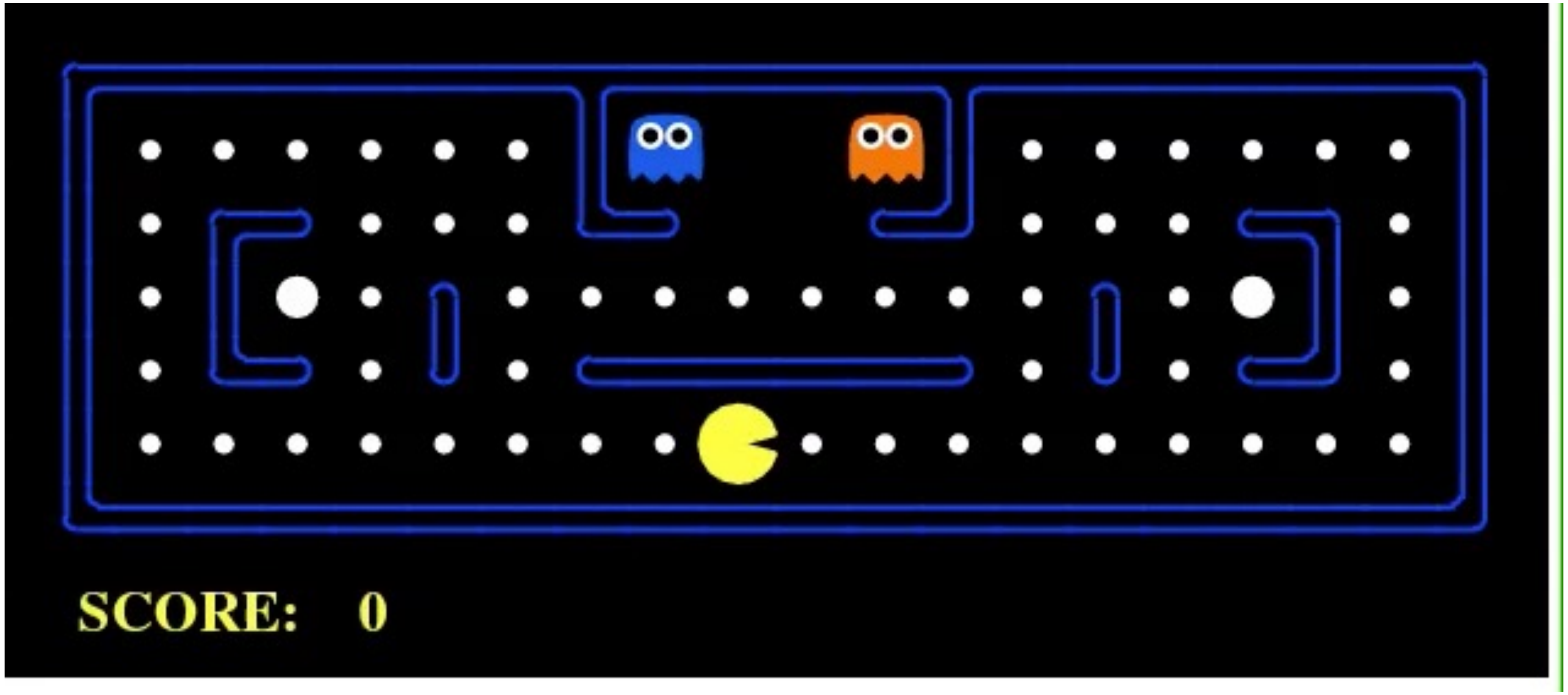
Which algorithm?

α-β, depth 4, simple eval fun



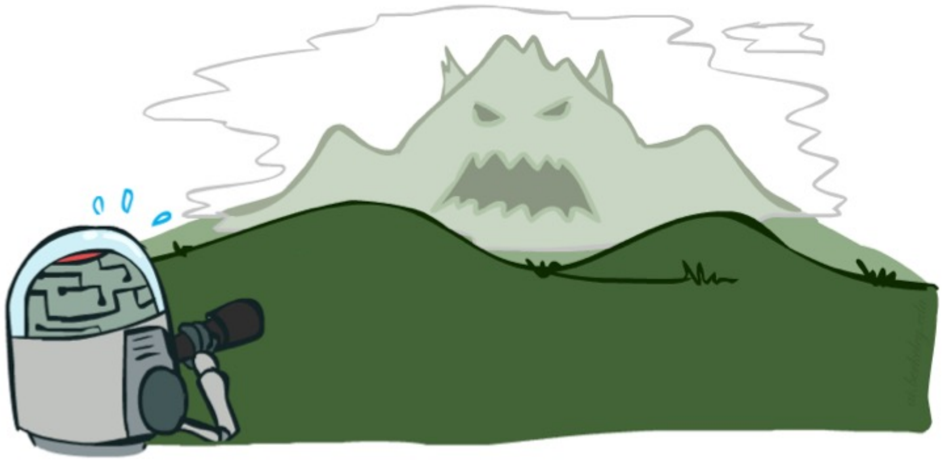
Which algorithm?

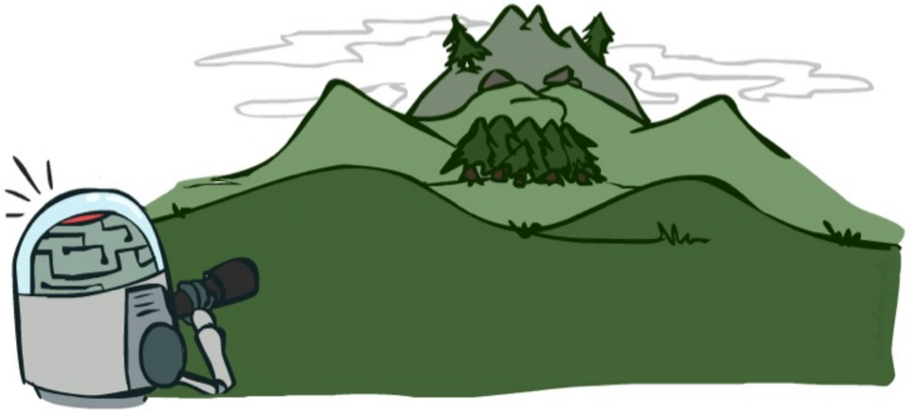
α-β, depth 4, better eval fun



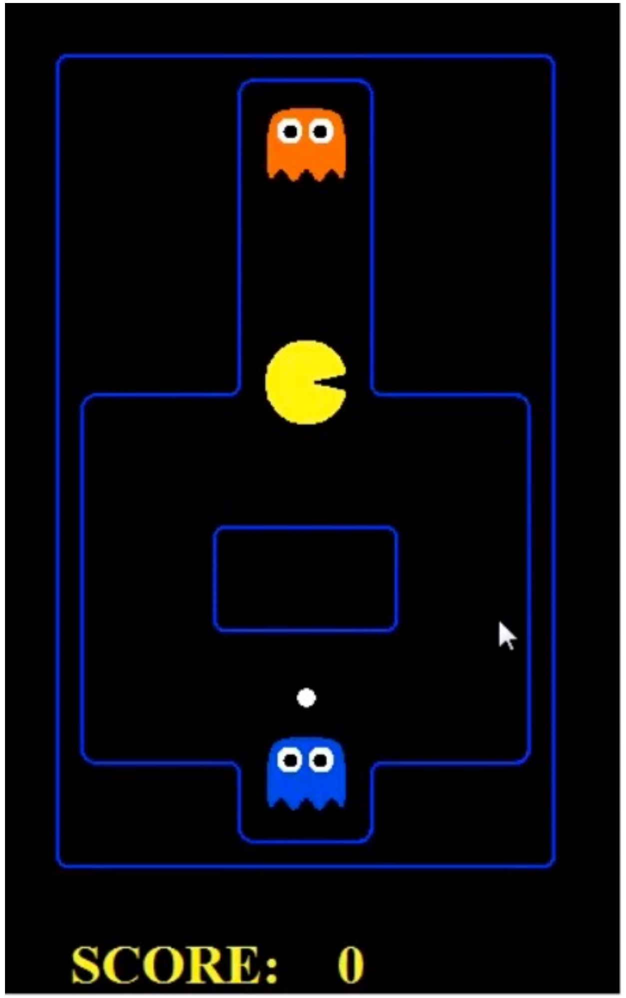
Depth Matters

- 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

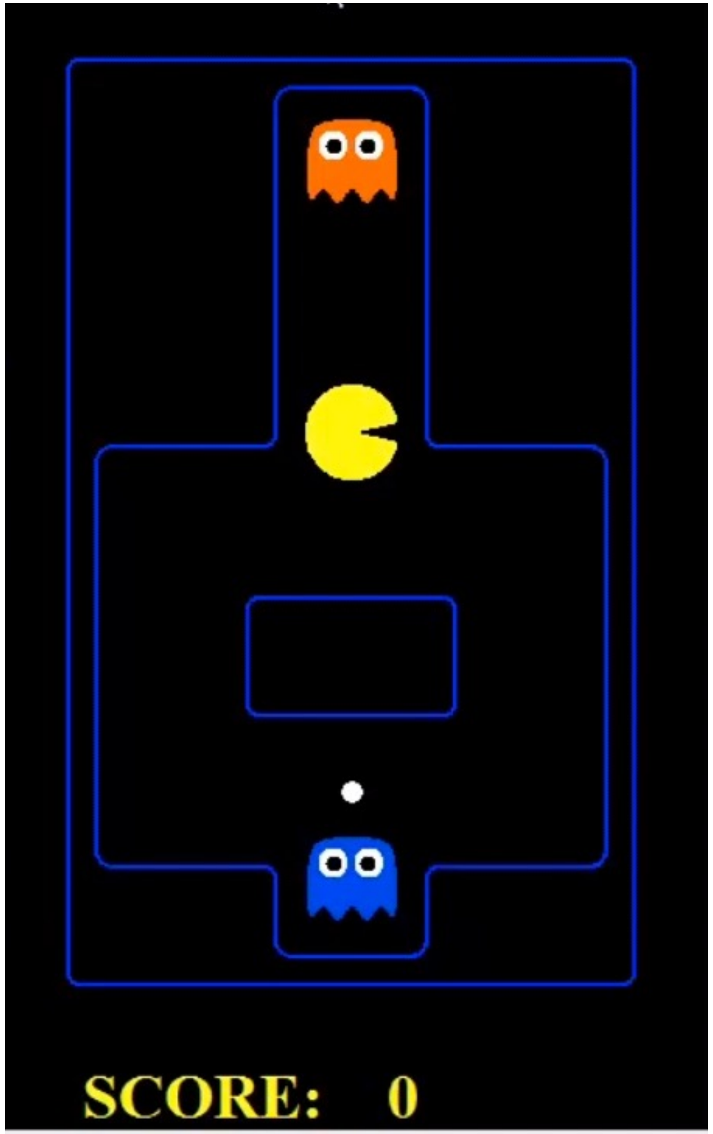




Video of Demo Limited Depth (2)



Video of Demo Limited Depth (10)



Synergies between Alpha-Beta and Evaluation Function

- Alpha-Beta: amount of pruning depends on expansion ordering
  - Evaluation function can provide guidance to expand most promising nodes first
- Alpha-beta:
  - 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