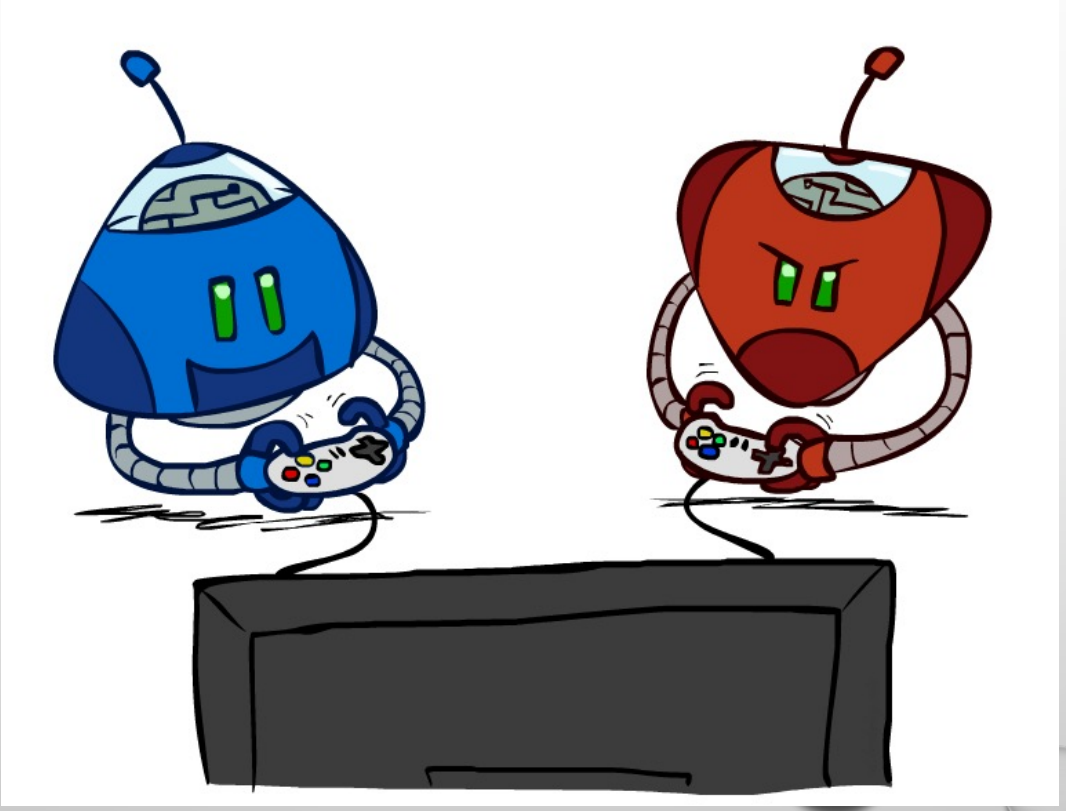
Artificial Intelligence
CE-417, Group 1
Computer Eng. Department
Sharif University of Technology

Spring 2024

By Mohammad Hossein Rohban, Ph.D.

Courtesy: Most slides are adopted from CSE-573 (Washington U.), original slides for the textbook, and CS-188 (UC. Berkeley).

Adversarial Search Methods

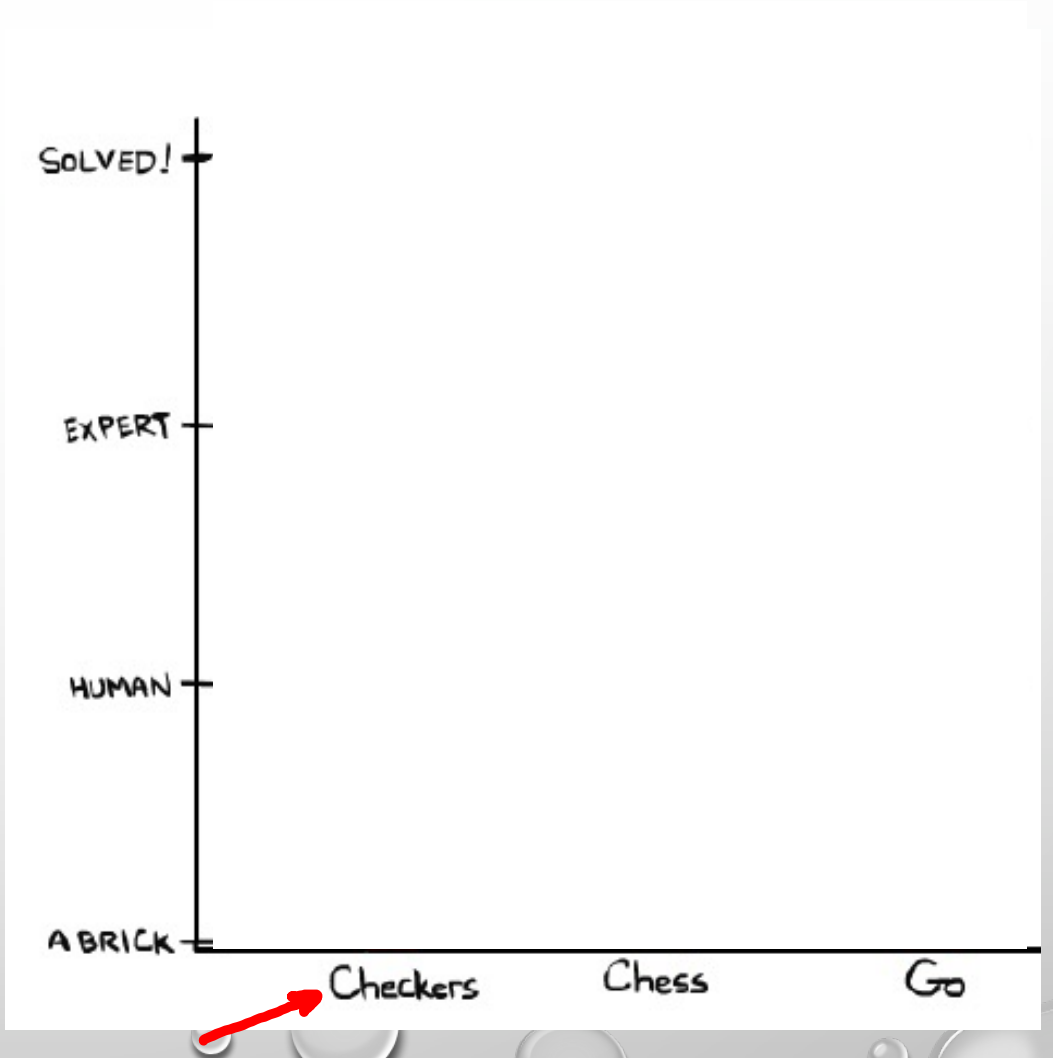


Game Playing State-of-the-Art

- **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:** AlphaGo defeats human in 2016. Uses Monte Carlo Tree Search and learned evaluation function.



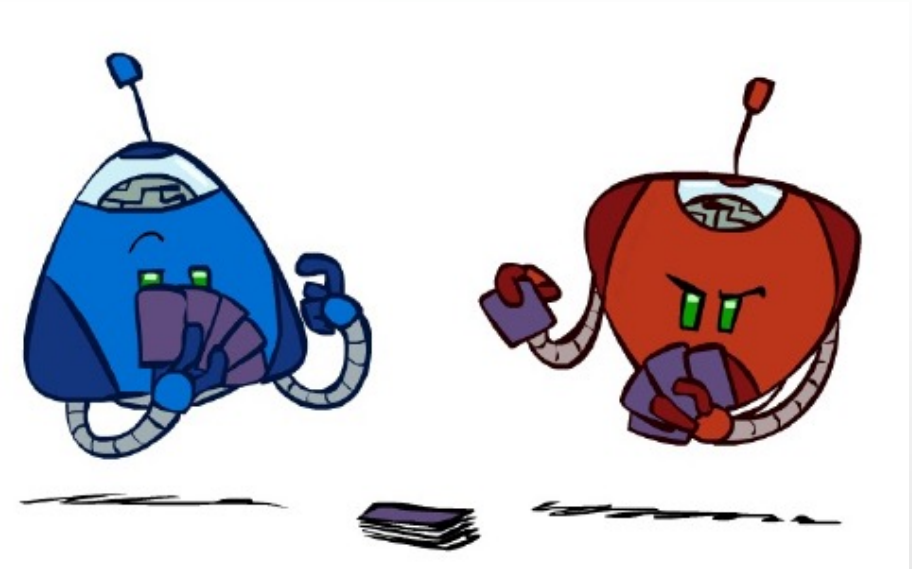
Games vs. search problems

- "Unpredictable" opponent ⇒ solution is a **strategy** specifying a move for every possible state/opponent reply
- Time limits ⇒ unlikely to find goal, must approximate

$$\pi : S \longrightarrow A$$

Types of Games

- Many different kinds of games!



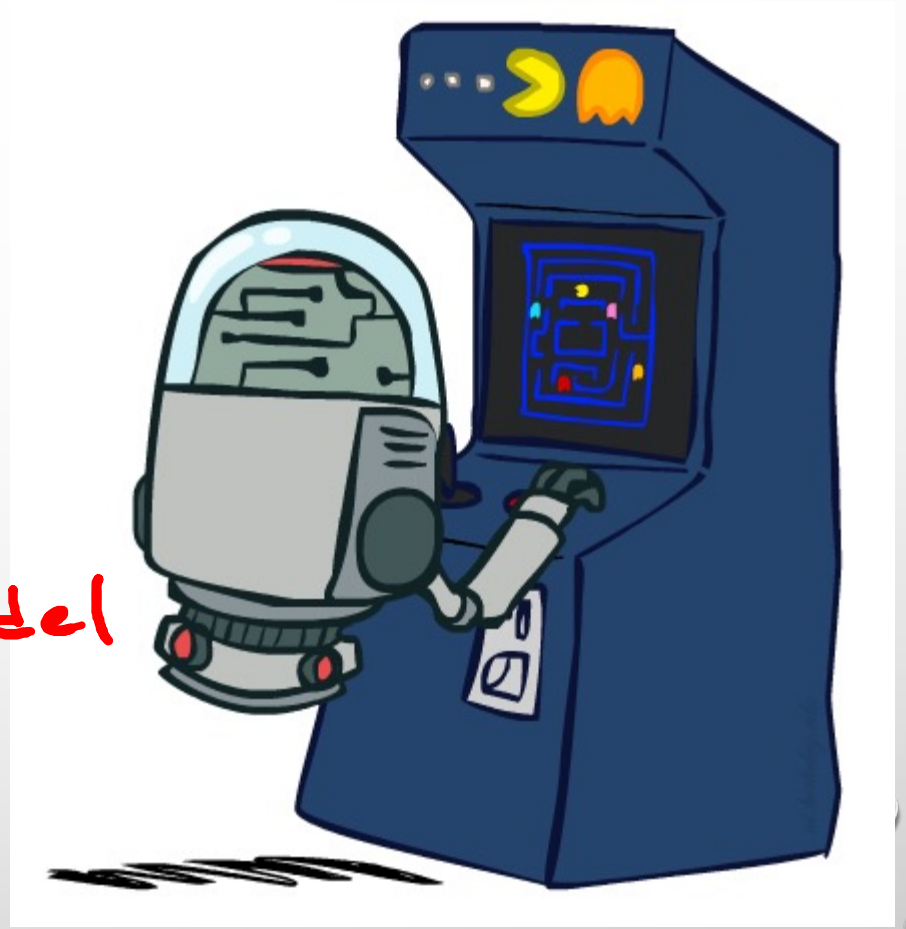
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?

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

Deterministic games

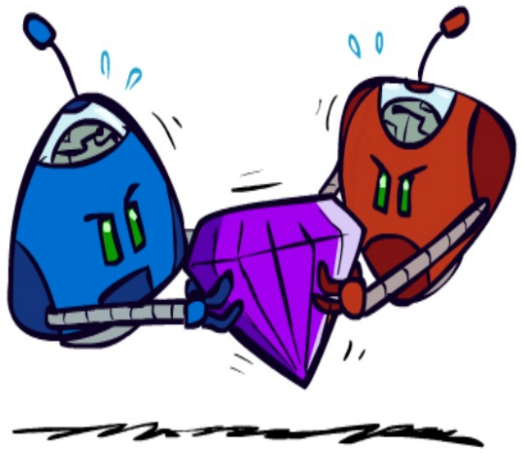
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1, … , N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition function: $S \times A \longrightarrow S$
  - Terminal test: $S \longrightarrow \{t, f\}$
  - Terminal utilities: $S \times P \longrightarrow R$

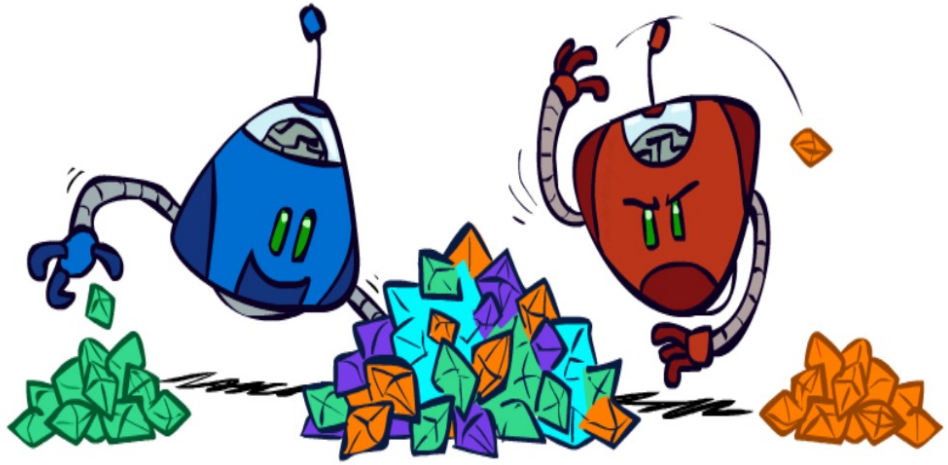
World Model



- Solution for a player is a policy: $\pi : S \longrightarrow A$

Zero-Sum Games

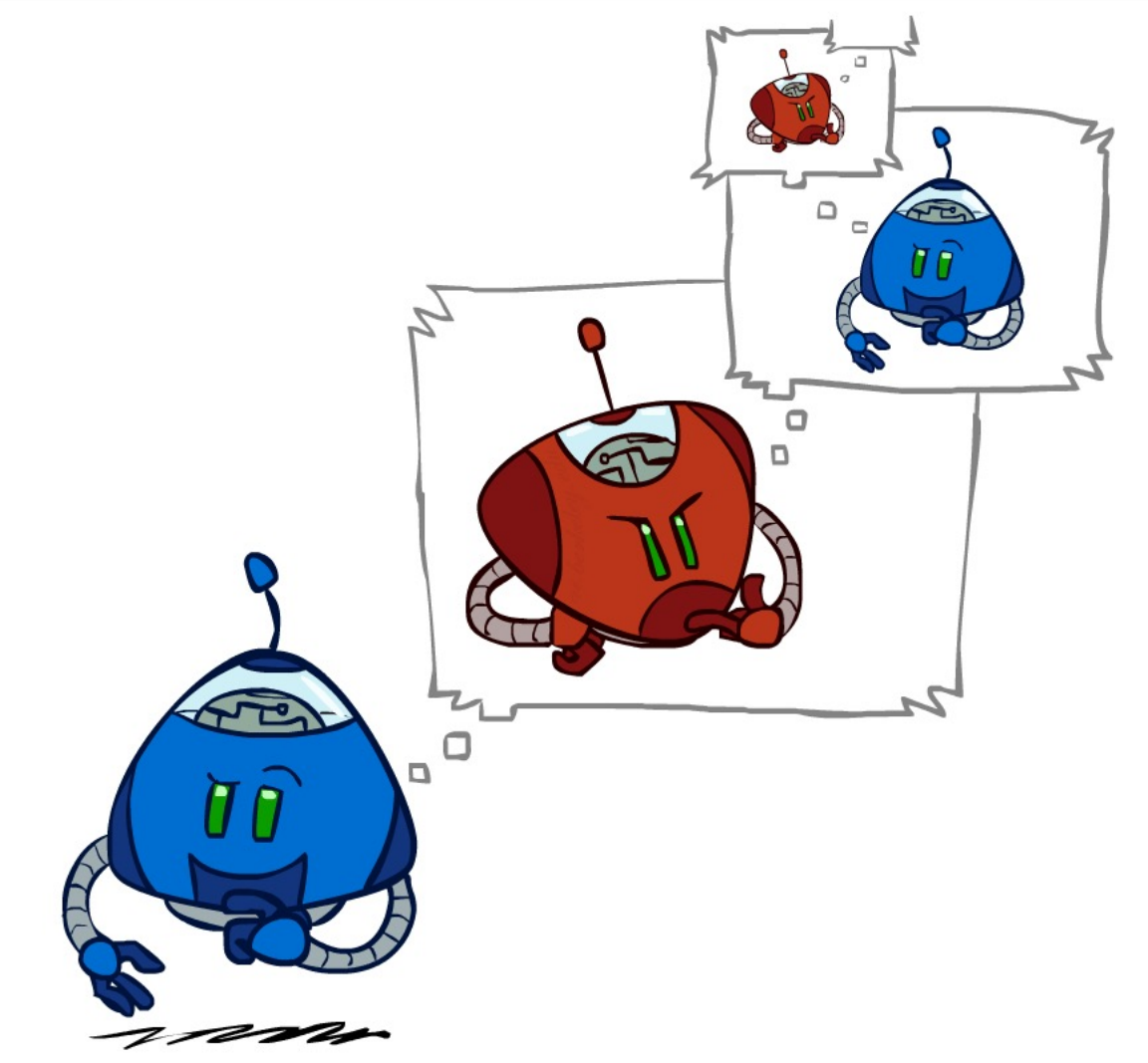




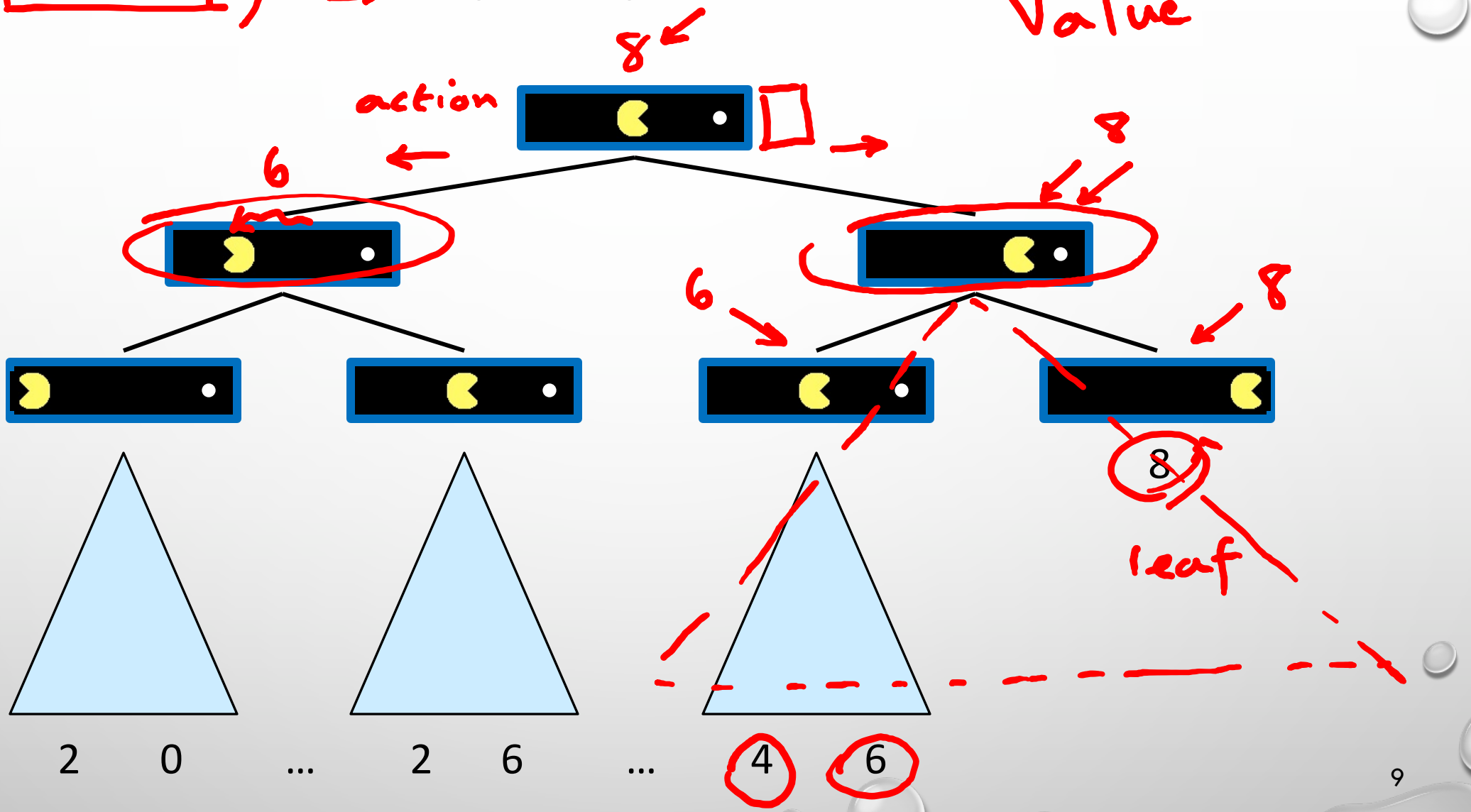
- **Zero-Sum Games**
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- **General Games**
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, & more are possible
  - More later on non-zero-sum games

Adversarial Search



Single-Agent Trees

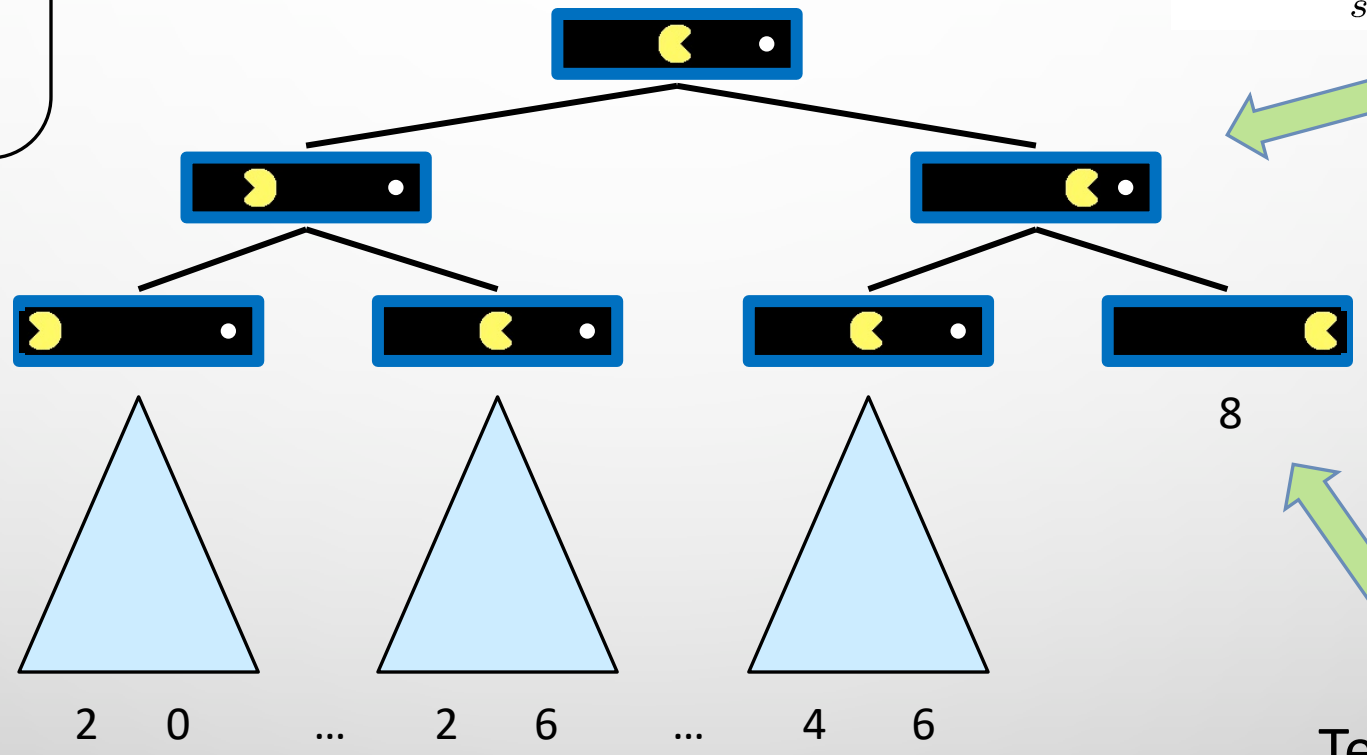


Value of a State

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

Non-Terminal States:

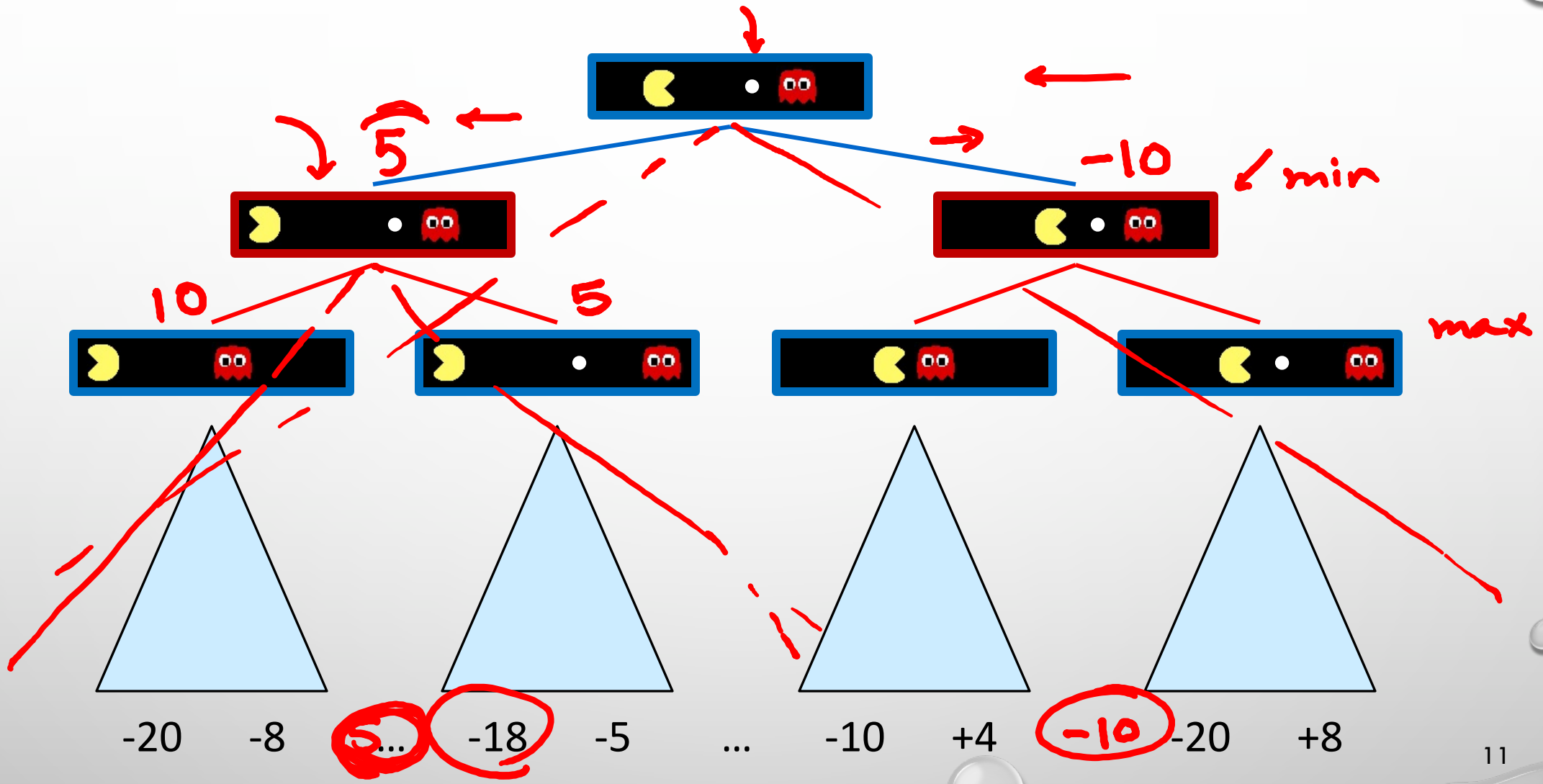
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial Game Trees



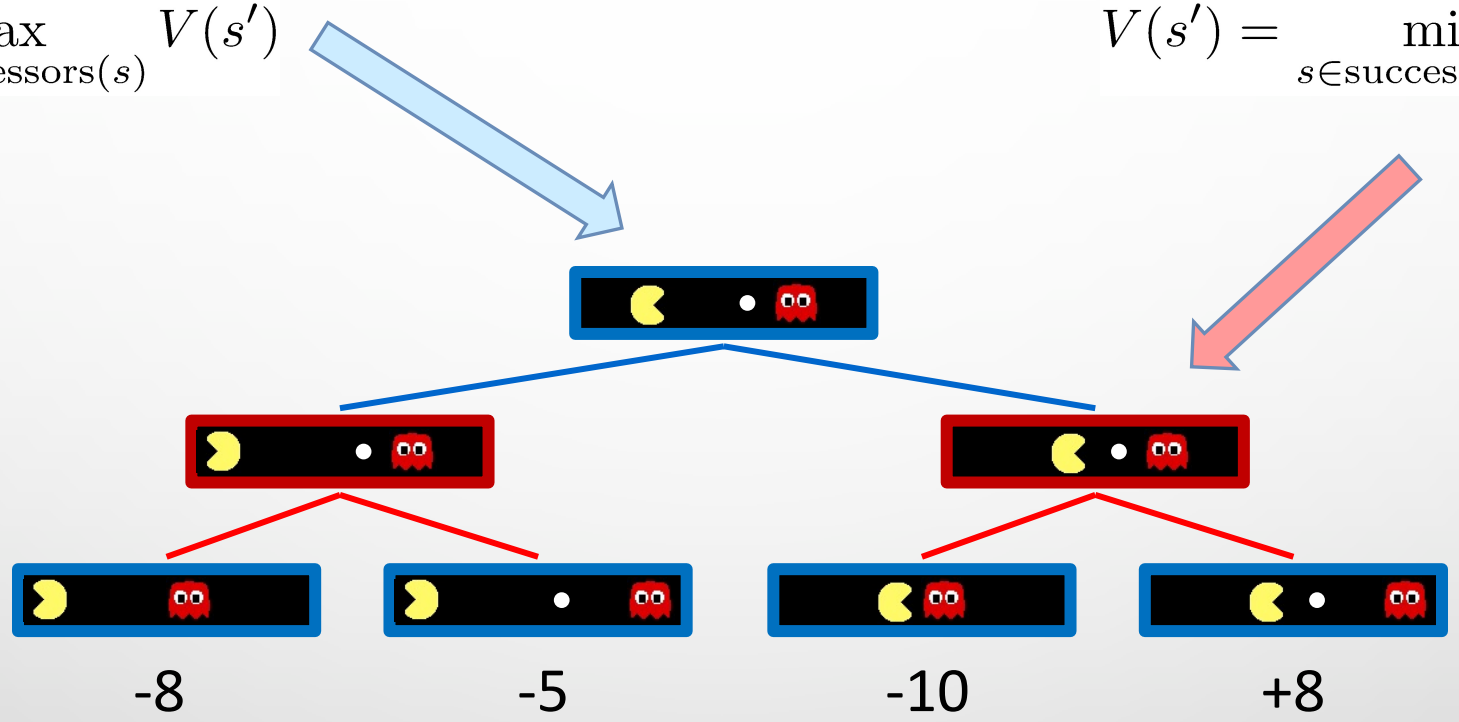
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

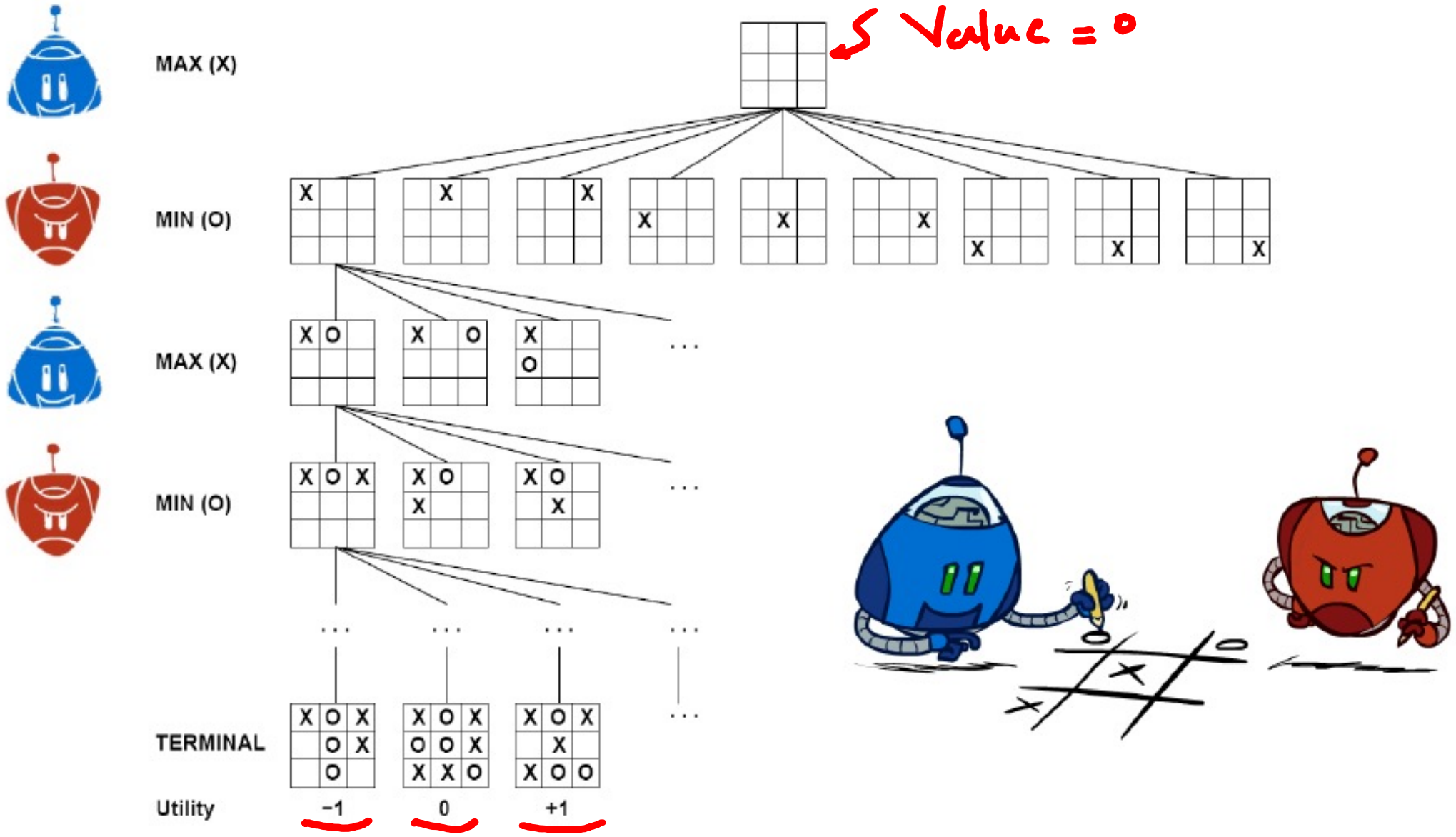
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

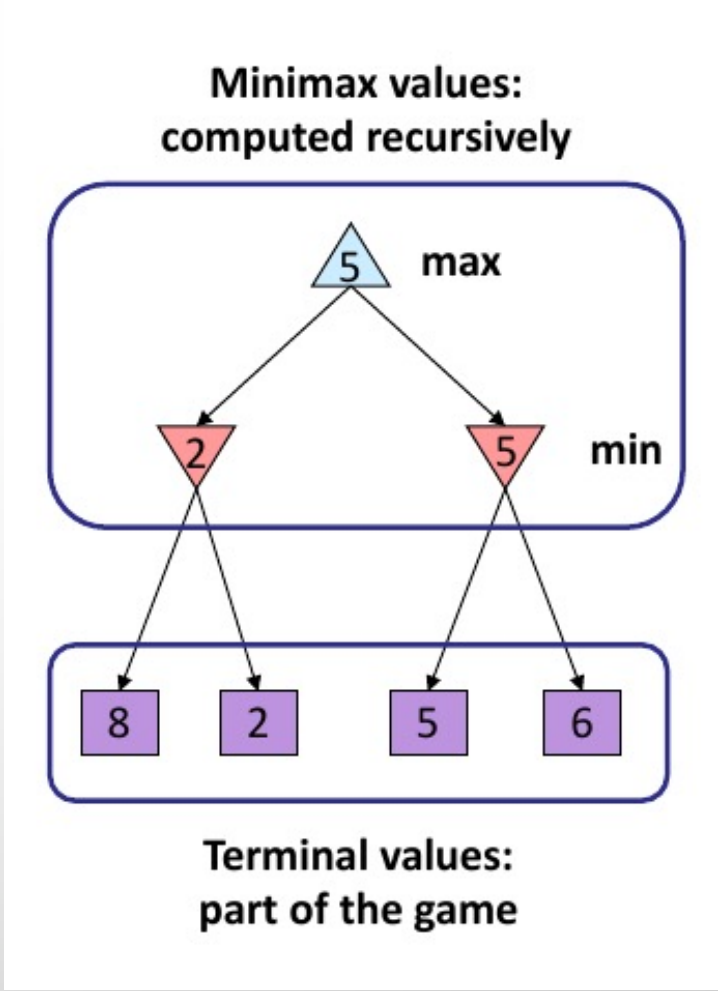
Tic-Tac-Toe Game Tree



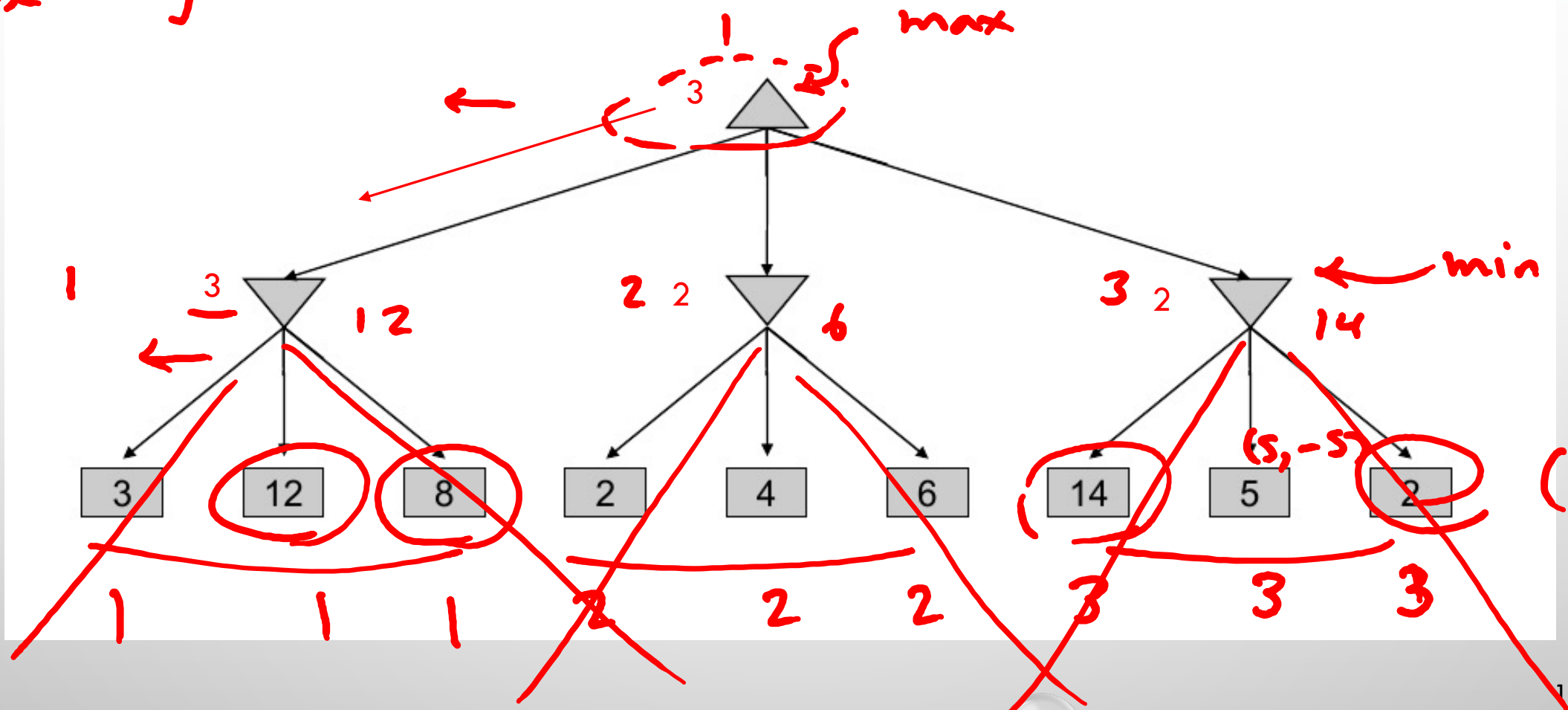
Adversarial Search (Minimax Strategy)

- Perfect play for deterministic, perfect-information, zero-sum games

- Idea:
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary
  - choose move to position with highest minimax value



Minimax Example



Minimax Implementation

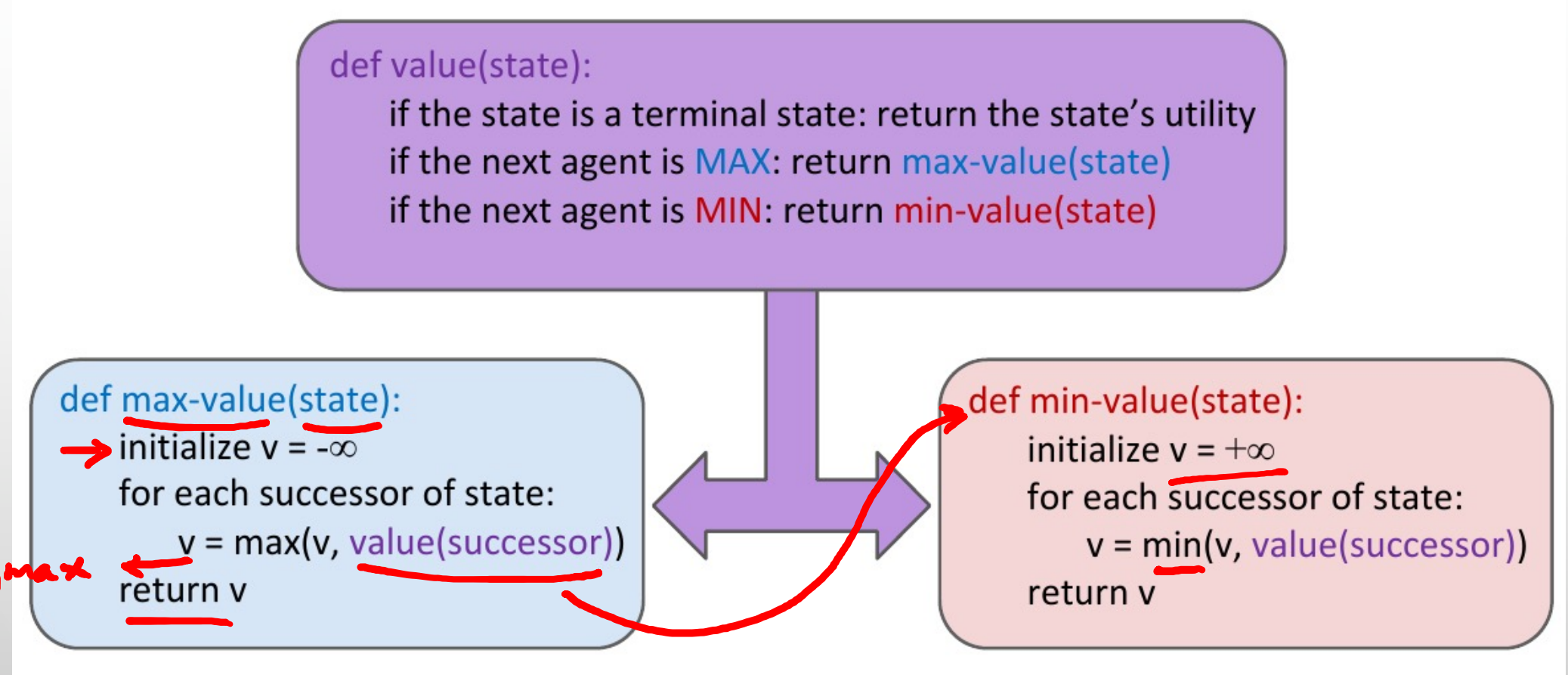
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

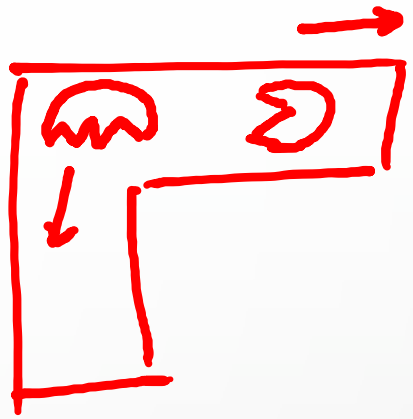
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

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

Minimax Implementation

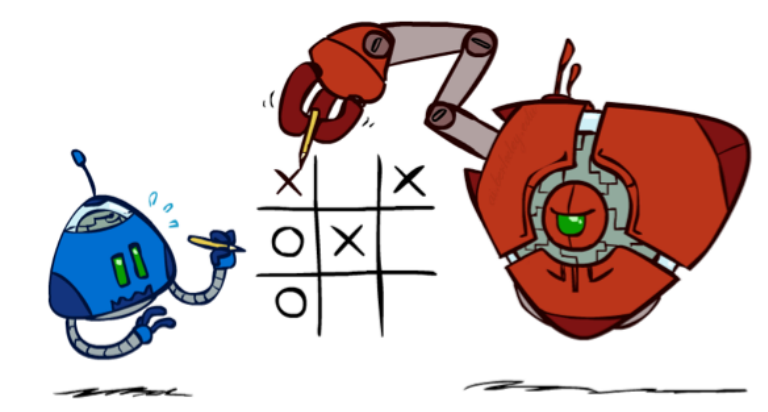


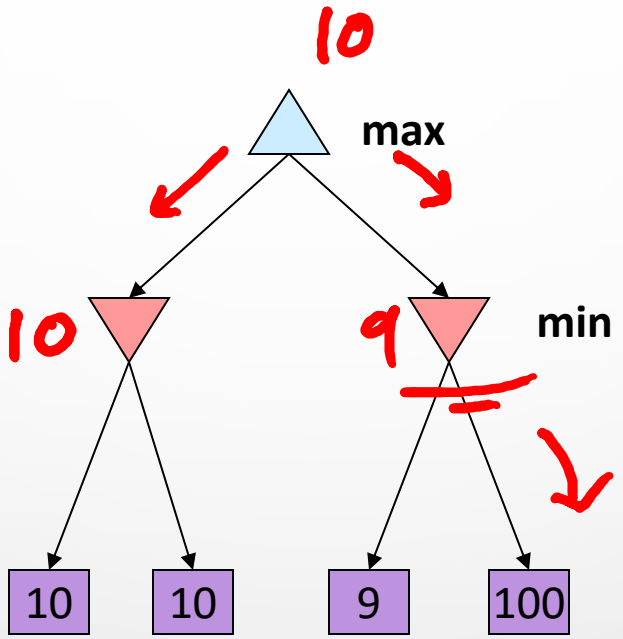
Properties of minimax

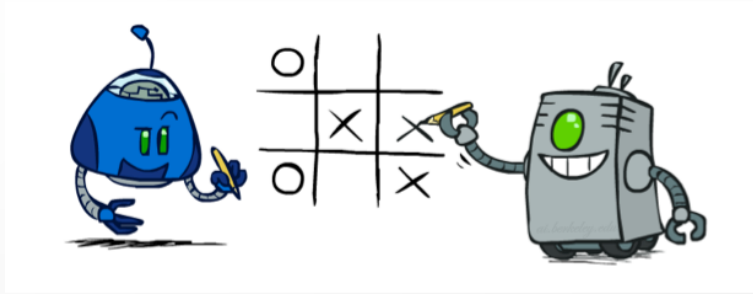


- **Complete**:
  - Yes, if tree is finite (chess has specific rules for this)
- **Optimal**:
  - Yes, against an optimal opponent. Otherwise?
- **Time complexity**:
  - $O(b^m)$
- **Space complexity**:
  - O(bm) (depth-first exploration)
- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games ⇒ exact solution completely infeasible
- But do we need to explore every path?

Properties of minimax (cont.)





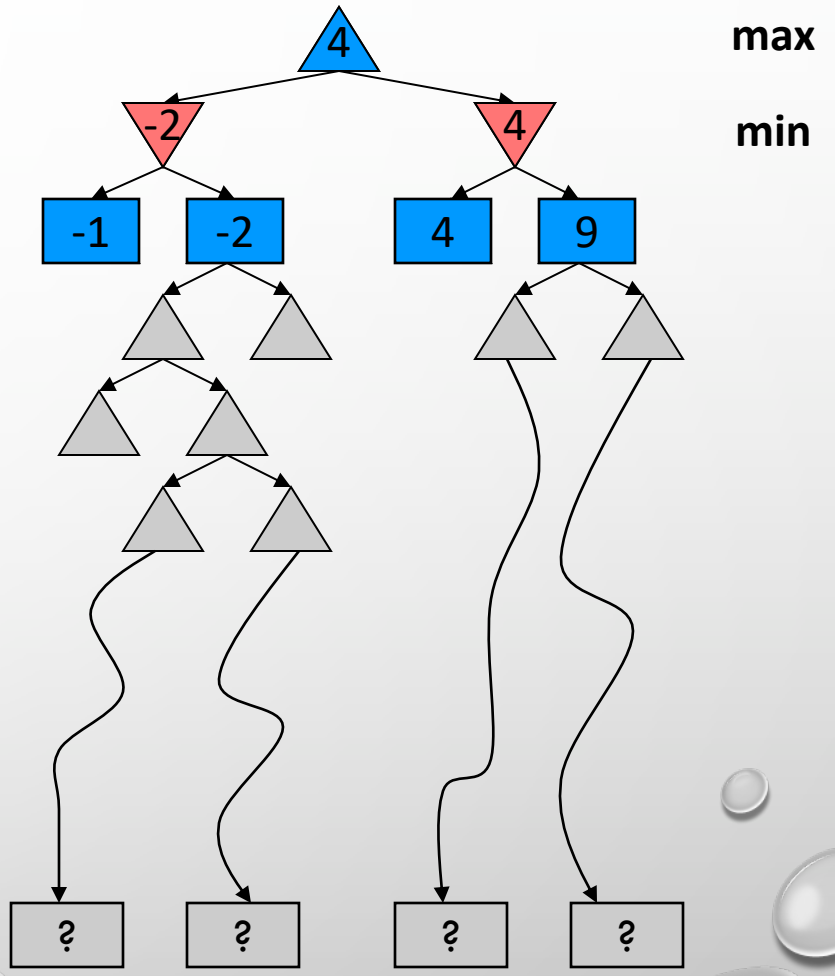


Optimal against a perfect player. Otherwise?

Resource Limits

Resource Limits

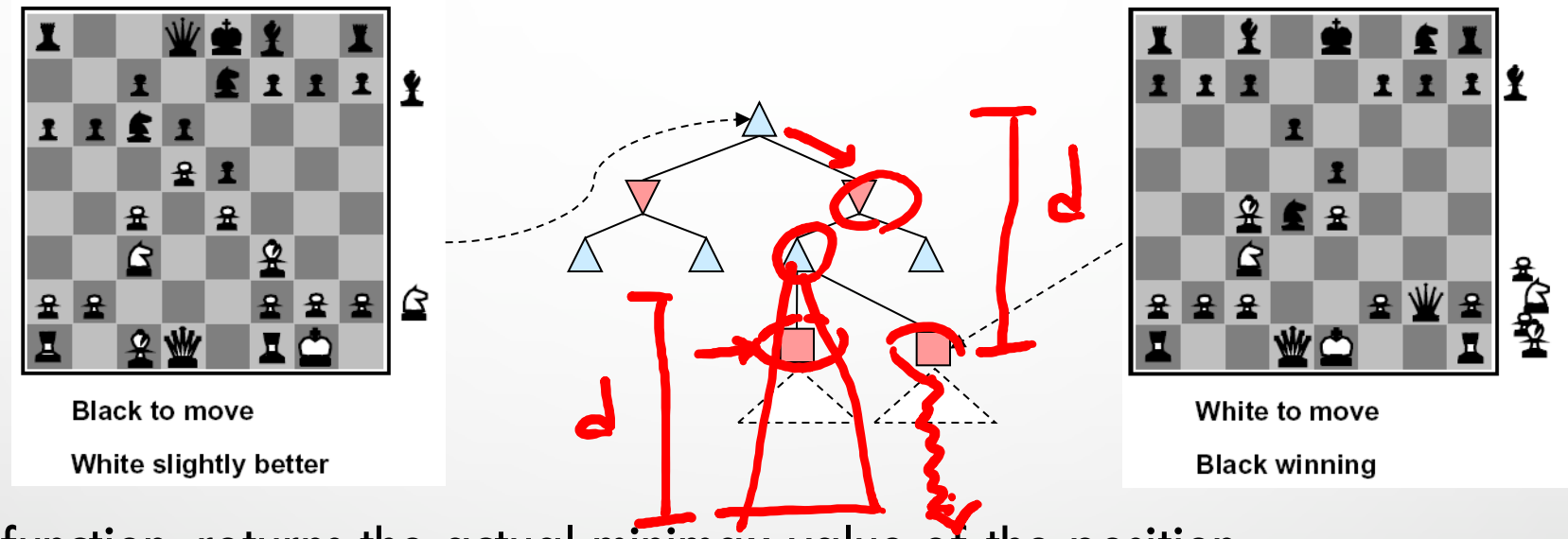
- Problem: in realistic games, cannot search to leaves!
- Solution: depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec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



Evaluation Functions

replanning

- 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:

$\underline{W}^* = (w_1^* \ldots w_n^*)$

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

Value

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

Human Intelligence vs. Comp. Resource

Example [edit]

An example handcrafted evaluation function for chess might look like the following:

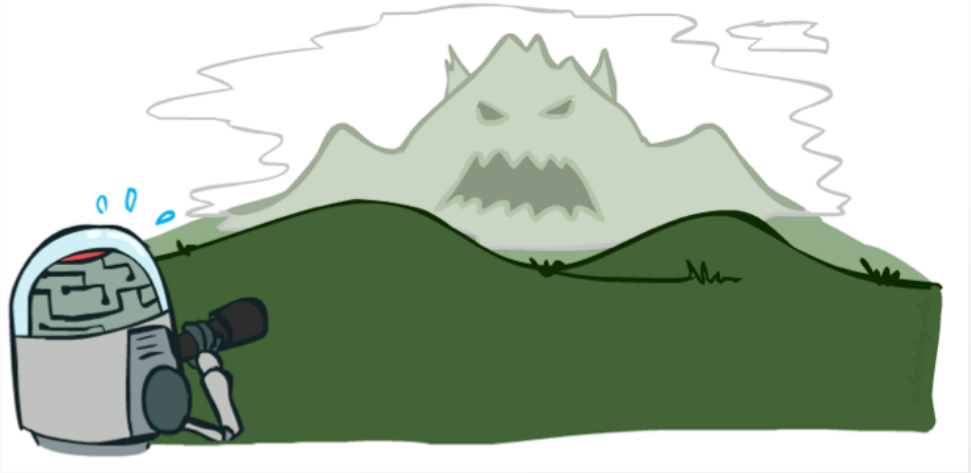
- c_1 * material + c_2 * mobility + c_3 * king safety + c_4 * center control + c_5 * pawn structure + c_6 * king tropism + ...

Each of the terms is a weight multiplied by a difference factor: the value of white's material or positional terms minus black's.

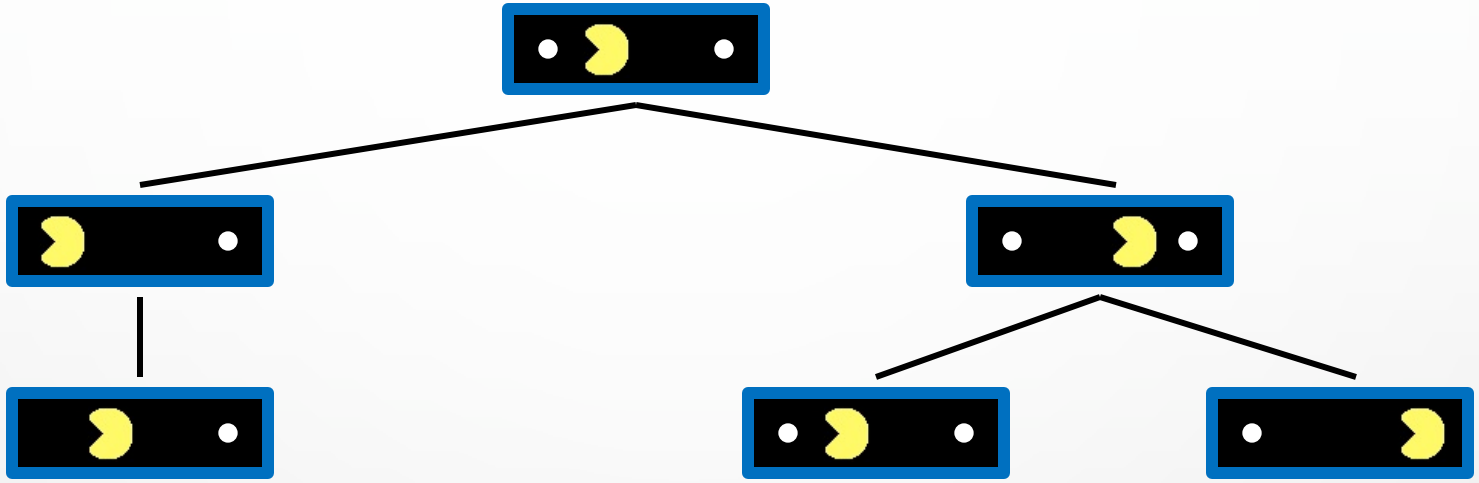
- The material term is obtained by assigning a value in pawn-units to each of the pieces.
- Mobility is the number of legal moves available to a player, or alternately the sum of the number of spaces attacked or defended by each piece, including spaces occupied by friendly or opposing pieces. Effective mobility, or the number of "safe" spaces a piece may move to, may also be taken into account.
- King safety is a set of bonuses and penalties assessed for the location of the king and the configuration of pawns and pieces adjacent to or in front of the king, and opposing pieces bearing on spaces around the king.
- Center control is derived from how many pawns and pieces occupy or bear on the four center spaces and sometimes the 12 spaces of the extended center.
- Pawn structure is a set of penalties and bonuses for various strengths and weaknesses in pawn structure, such as penalties for doubled and isolated pawns.
- King tropism is a bonus for closeness (or penalty for distance) of certain pieces, especially queens and knights, to the opposing king.

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

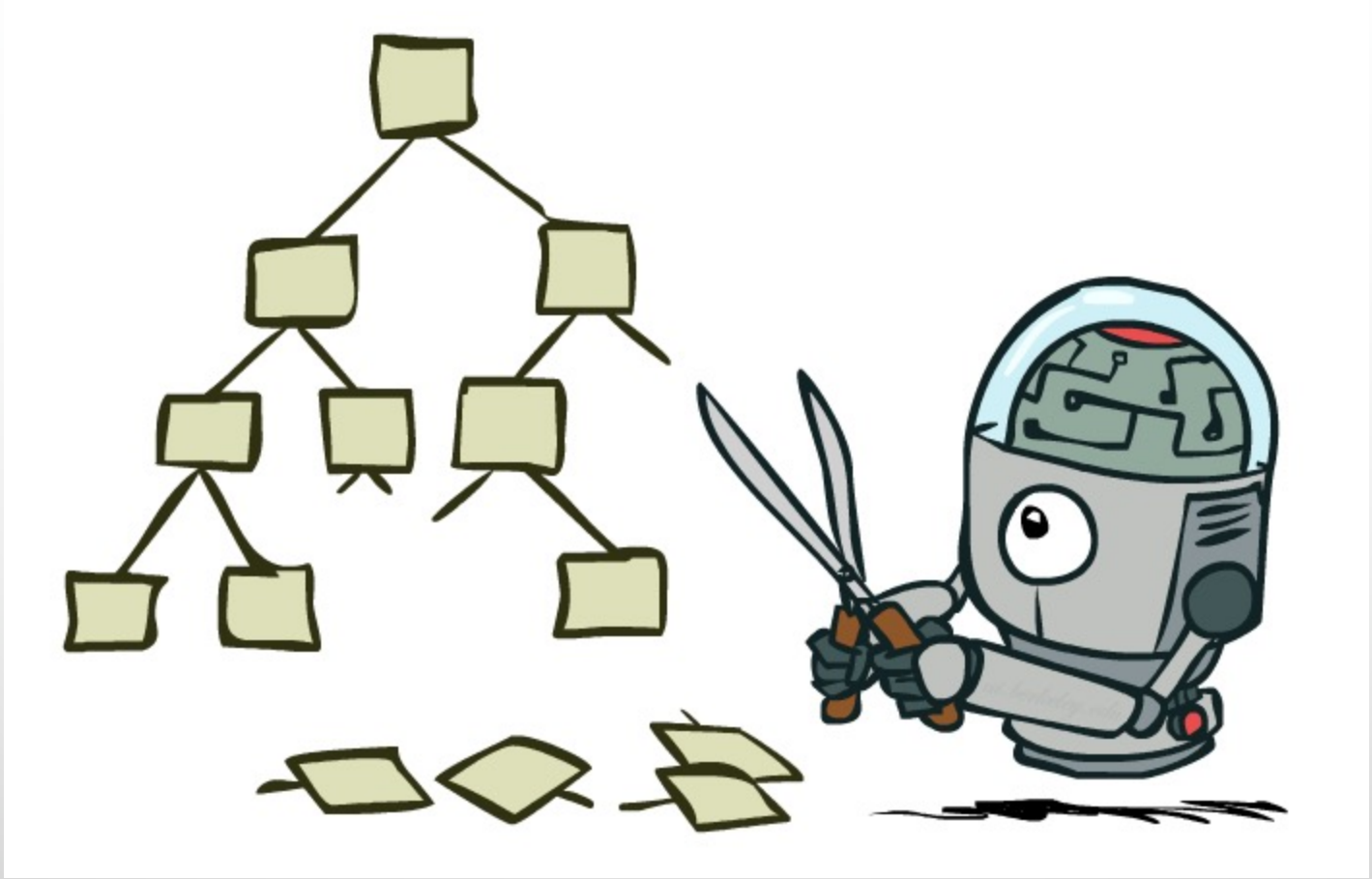


Why Pacman Starves? (evaluation function matters)

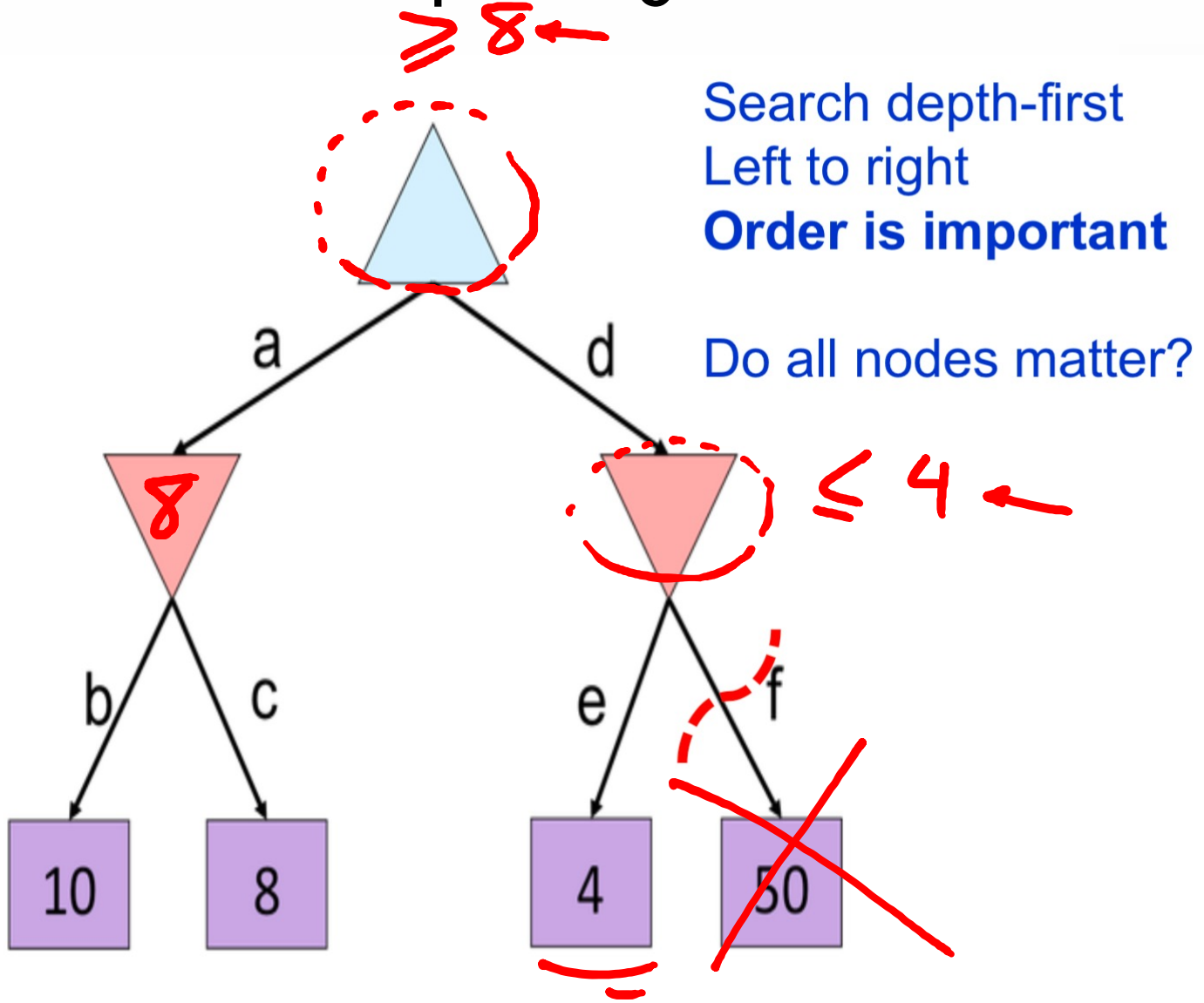


- A danger of re-planning agents! (assume that evaluation function is 10 * number of eaten dots).
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after 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!

Minimax pruning



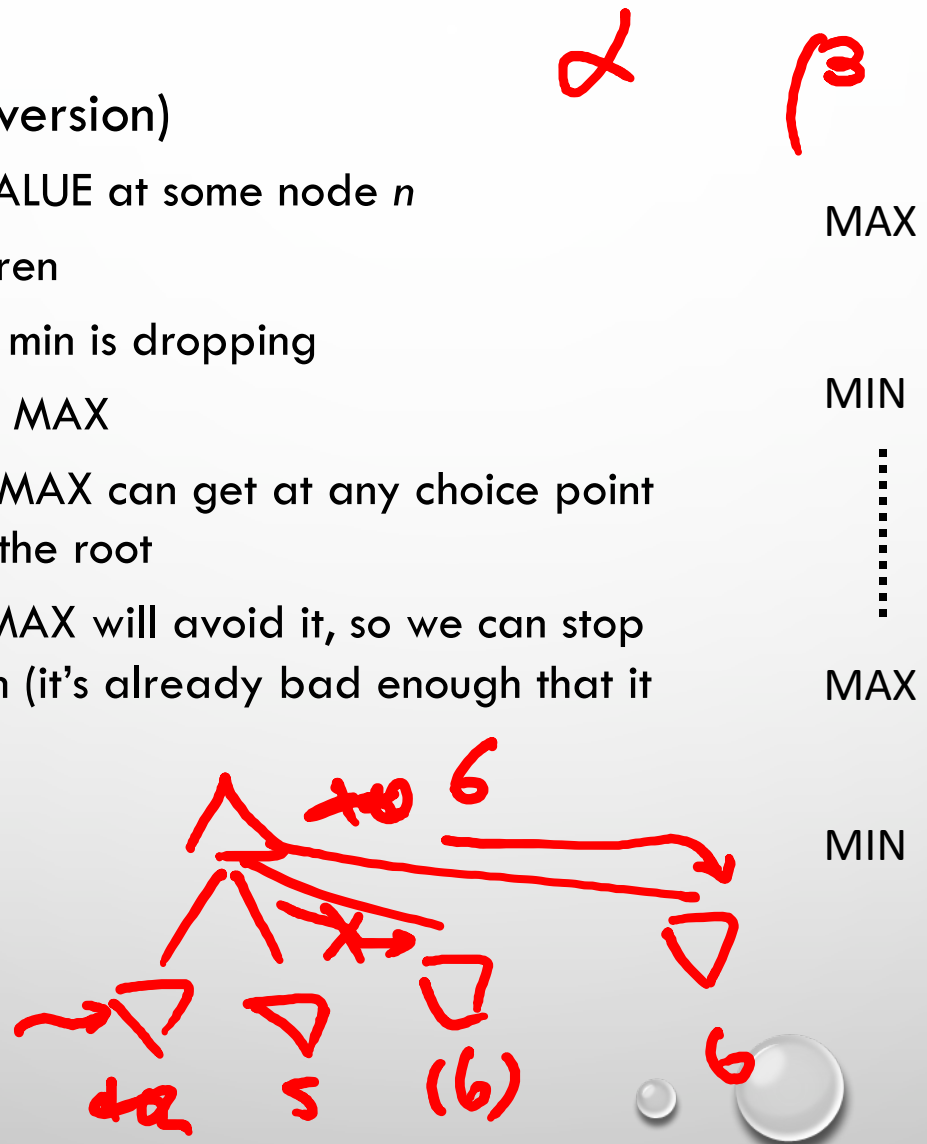
Minimax pruning



Slide adapted from Dan Klein & Pieter Abbeel - ai.berkeley.edu

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

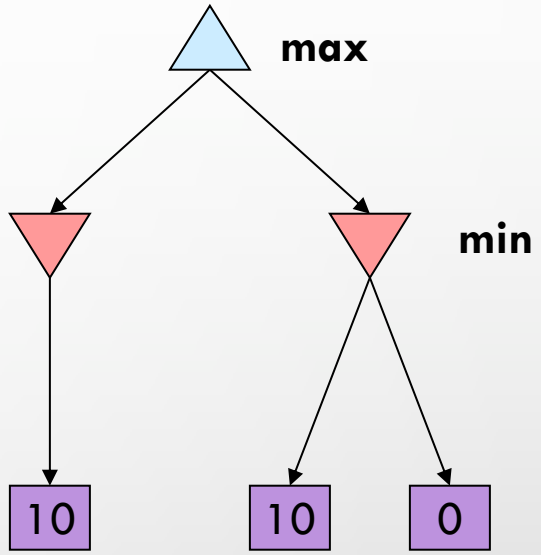
α: MAX'S BEST OPTION ON PATH TO ROOT
β: MIN'S BEST OPTION ON PATH TO ROOT

```
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
```

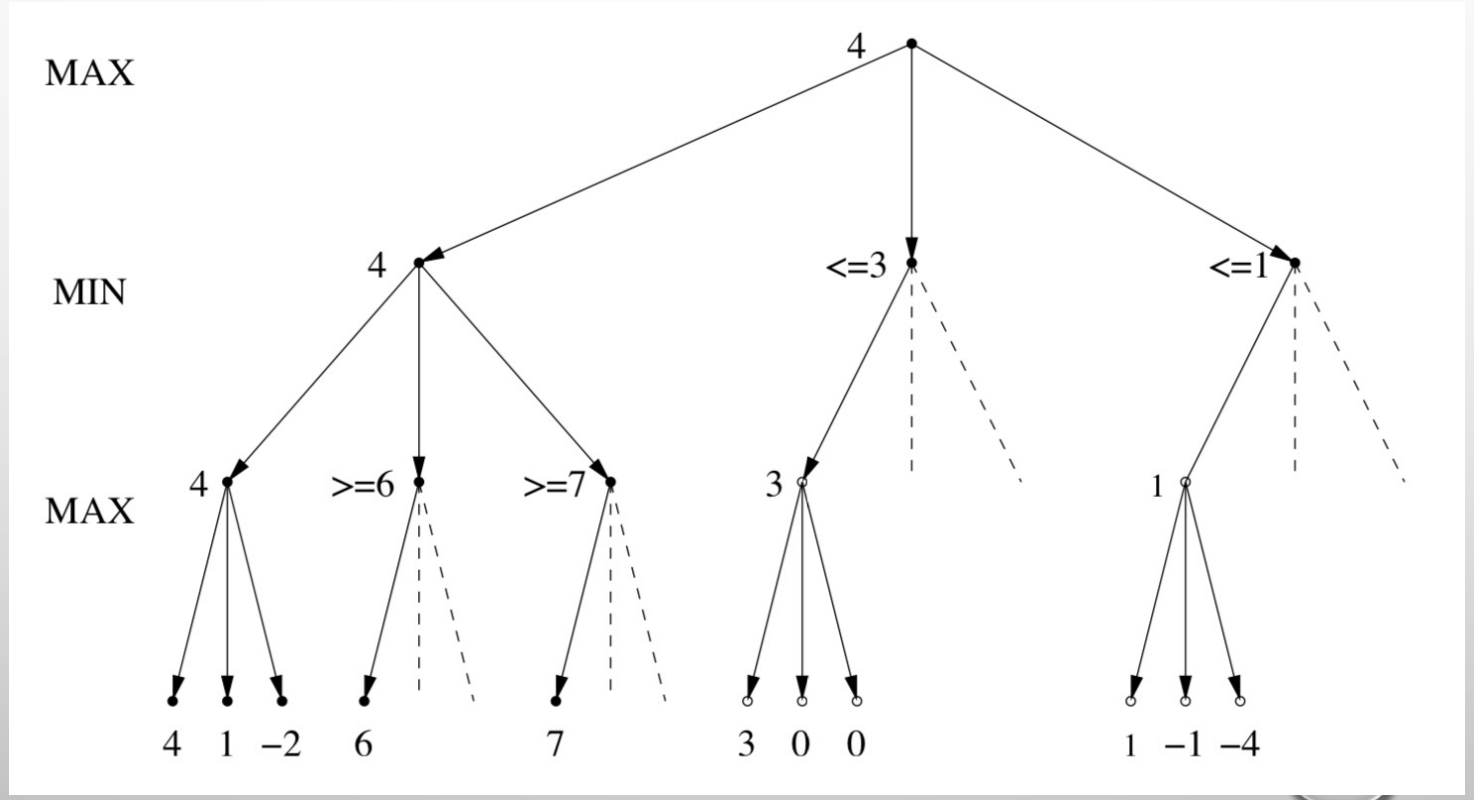
Alpha-Beta Pruning Properties

- 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 $O((2b)^{m/2})$ or better $O\left(\left(\sqrt{b}+0.5\right)^{m+1}\right)$
  - Doubles solvable depth!
  - Full search of, e.g. Chess, is still hopeless…
- With random ordering:
  - The total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b.
- This is a simple example of meta-reasoning (computing about what to compute)



Best Case Analysis of the Alpha-Beta pruning

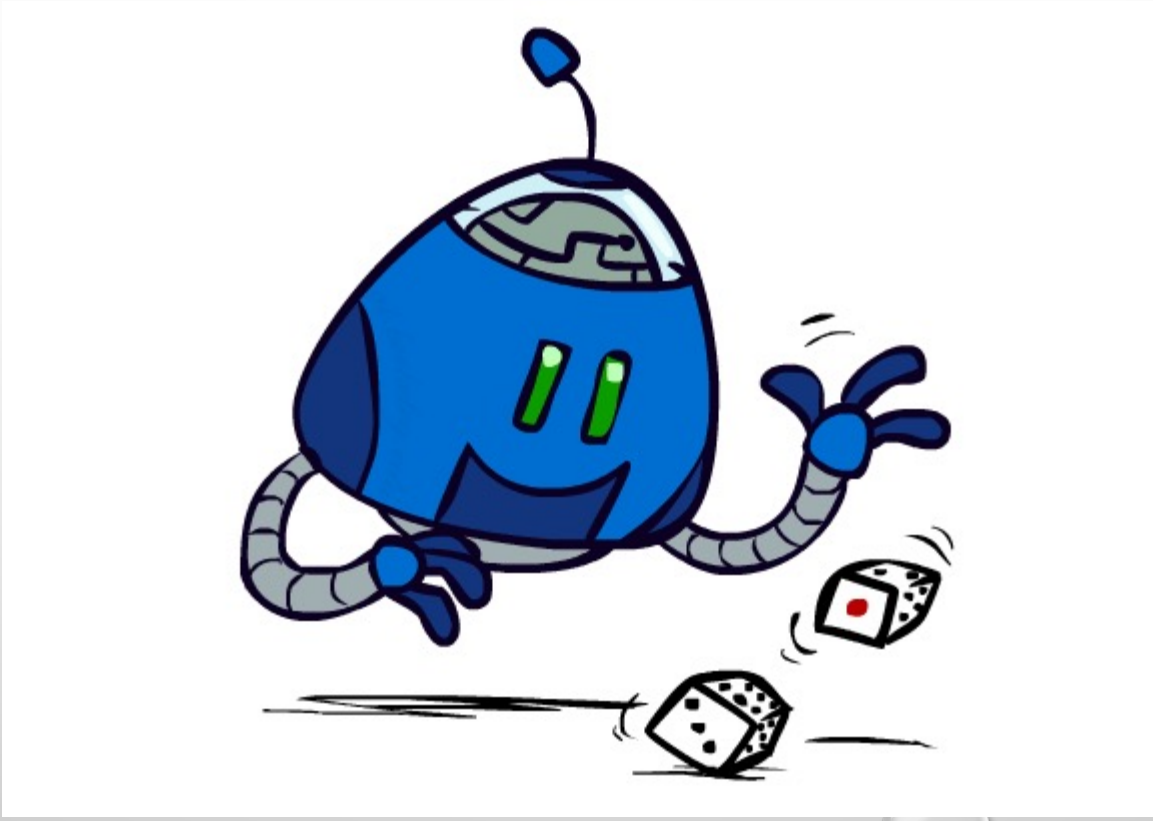
- In the best case, the minimax values are explored in descending order for MAX and in ascending order for MIN (why?)
- Can you write down a set of recursive equations to find the time complexity?



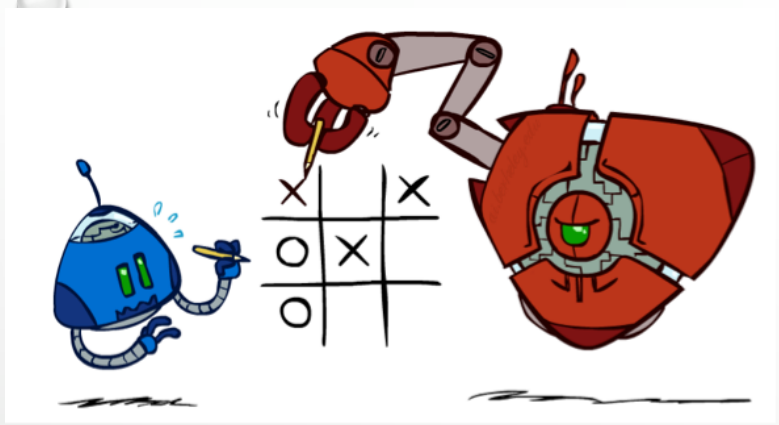
Synergies between Alpha-Beta and Evaluation Function

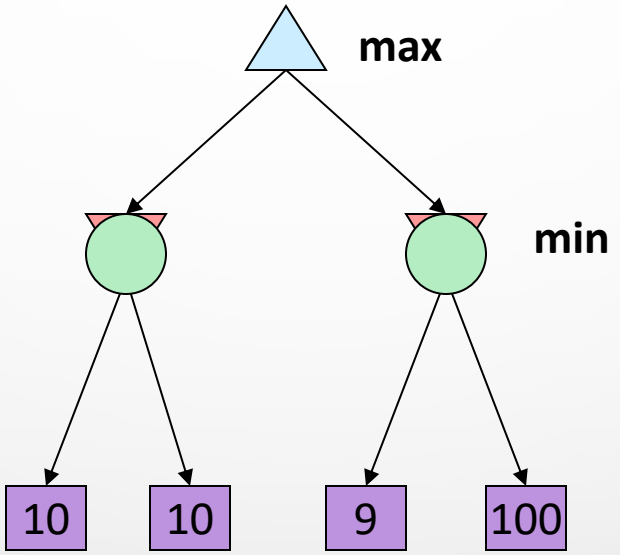
- Alpha-Beta: amount of pruning depends on expansion ordering
  - Evaluation function can provide guidance to expand most promising nodes first
- Alpha-beta:
  - Value at a min-node will only keep going down
  - Once value of min-node lower than better option for max along path to root, can prune
  - 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

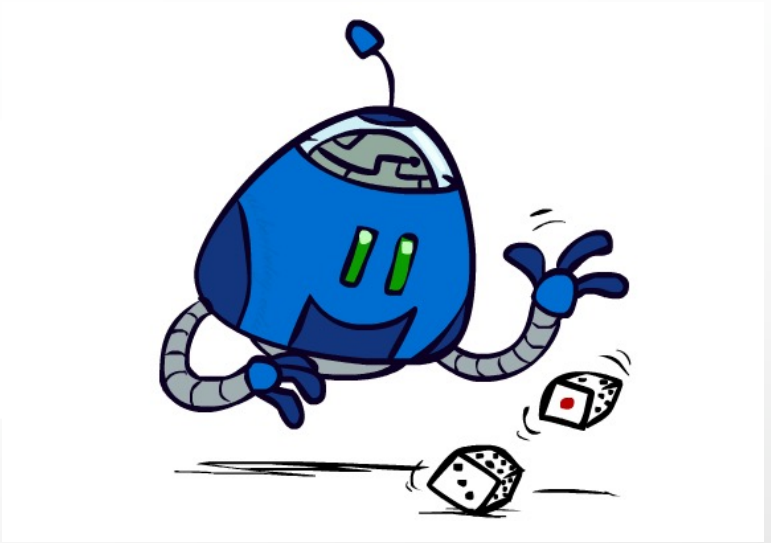
Uncertain Outcomes



Worst-Case vs. Average Case



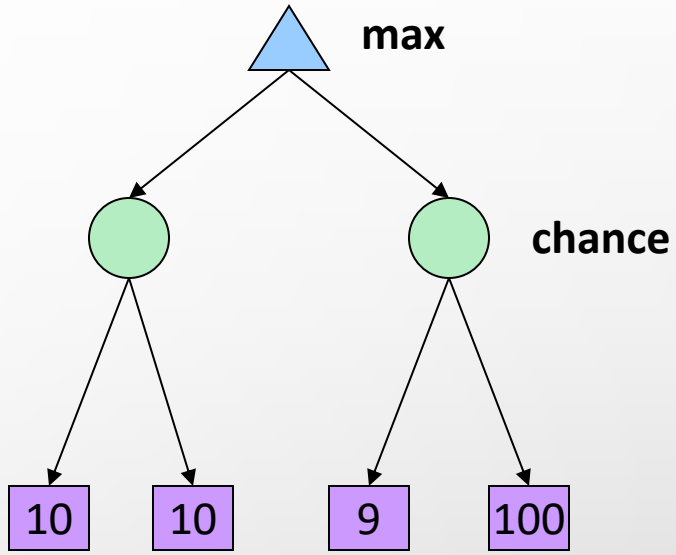




Idea: Uncertain outcomes controlled by chance, not an adversary!

Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes
- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - i.e. Take weighted average (expectation) of children
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



Expectimax Pseudocode

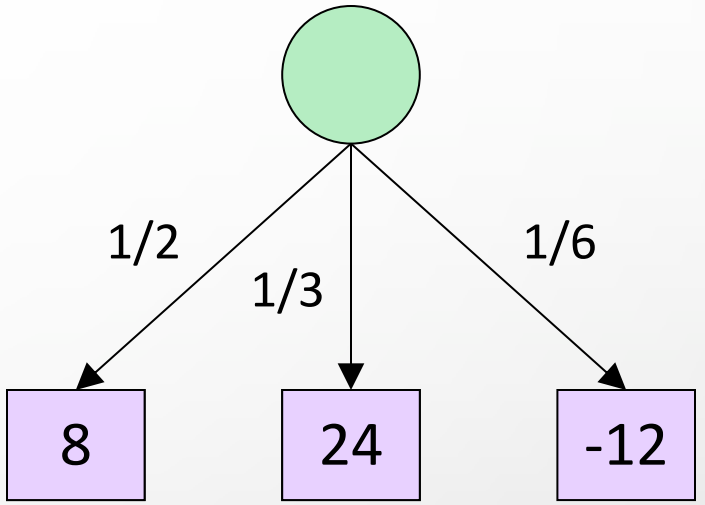
```
Def value(state):
    If the state is a terminal state: return the state's utility
    If the next agent is MAX: return max-value(state)
    If the next agent is EXP: return exp-value(state)
```

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

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

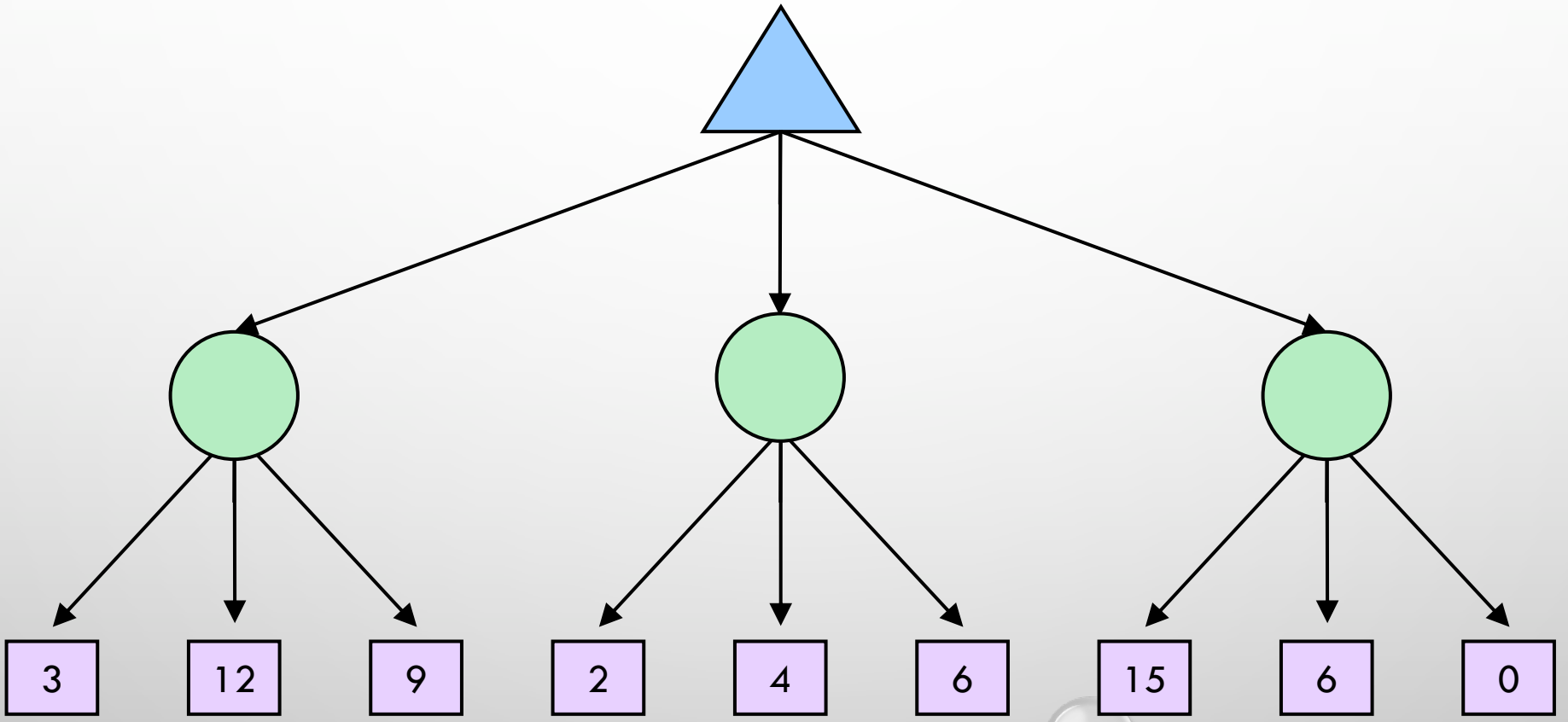
Expectimax Pseudocode

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

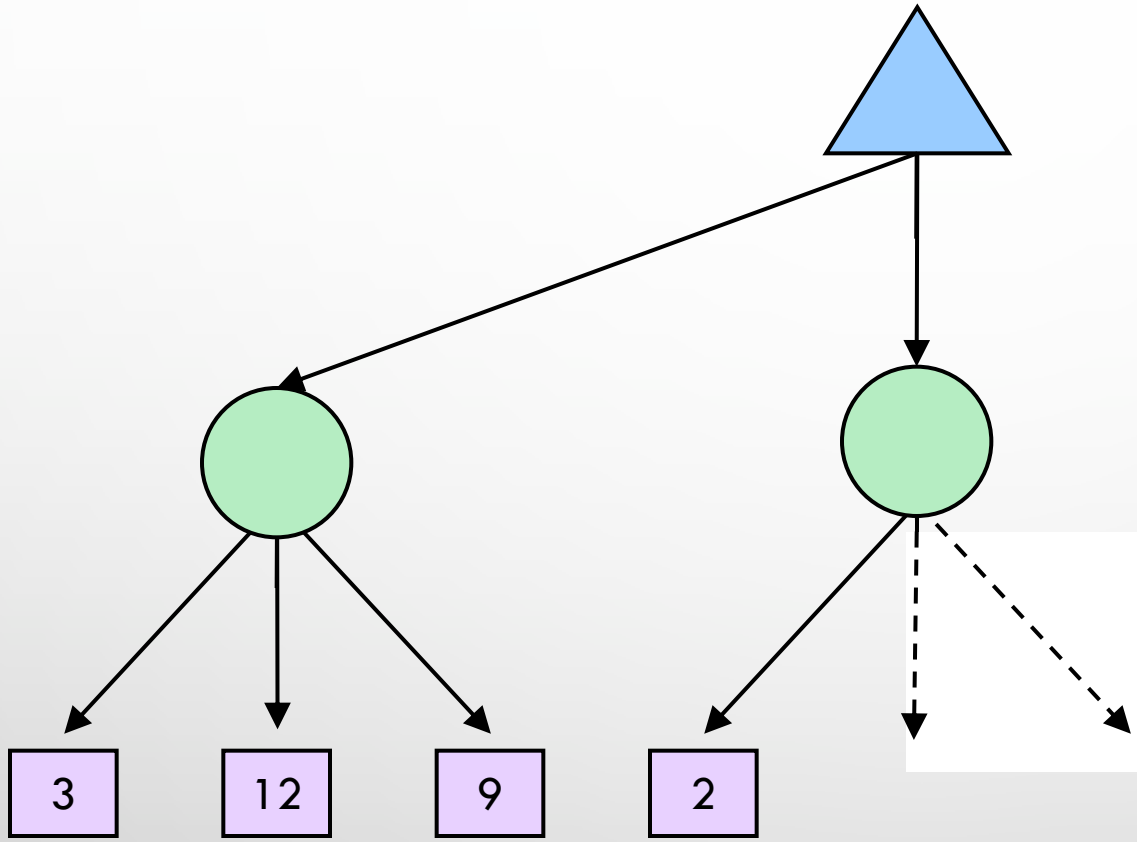


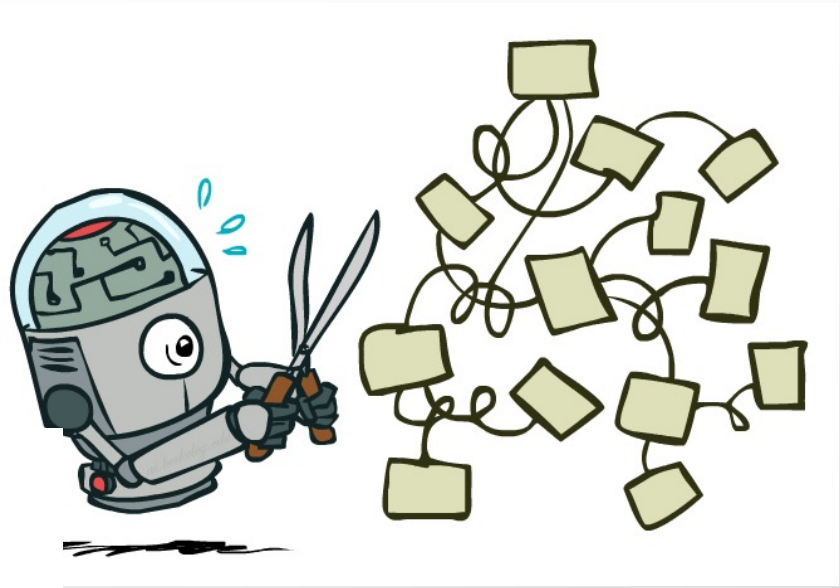
v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

Expectimax Example

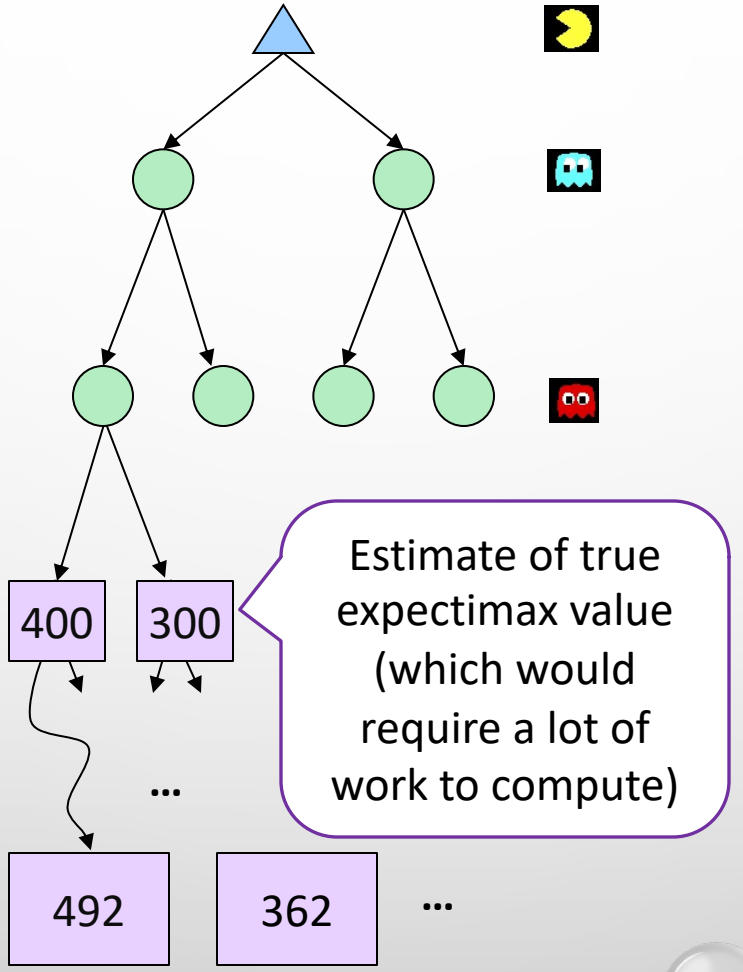


Expectimax Pruning?



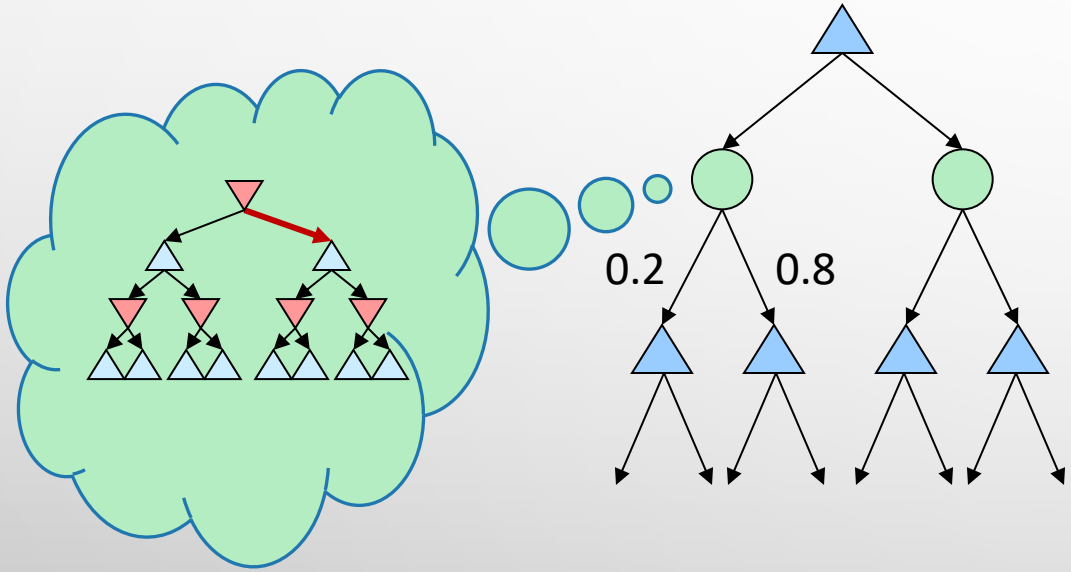


Depth-Limited Expectimax



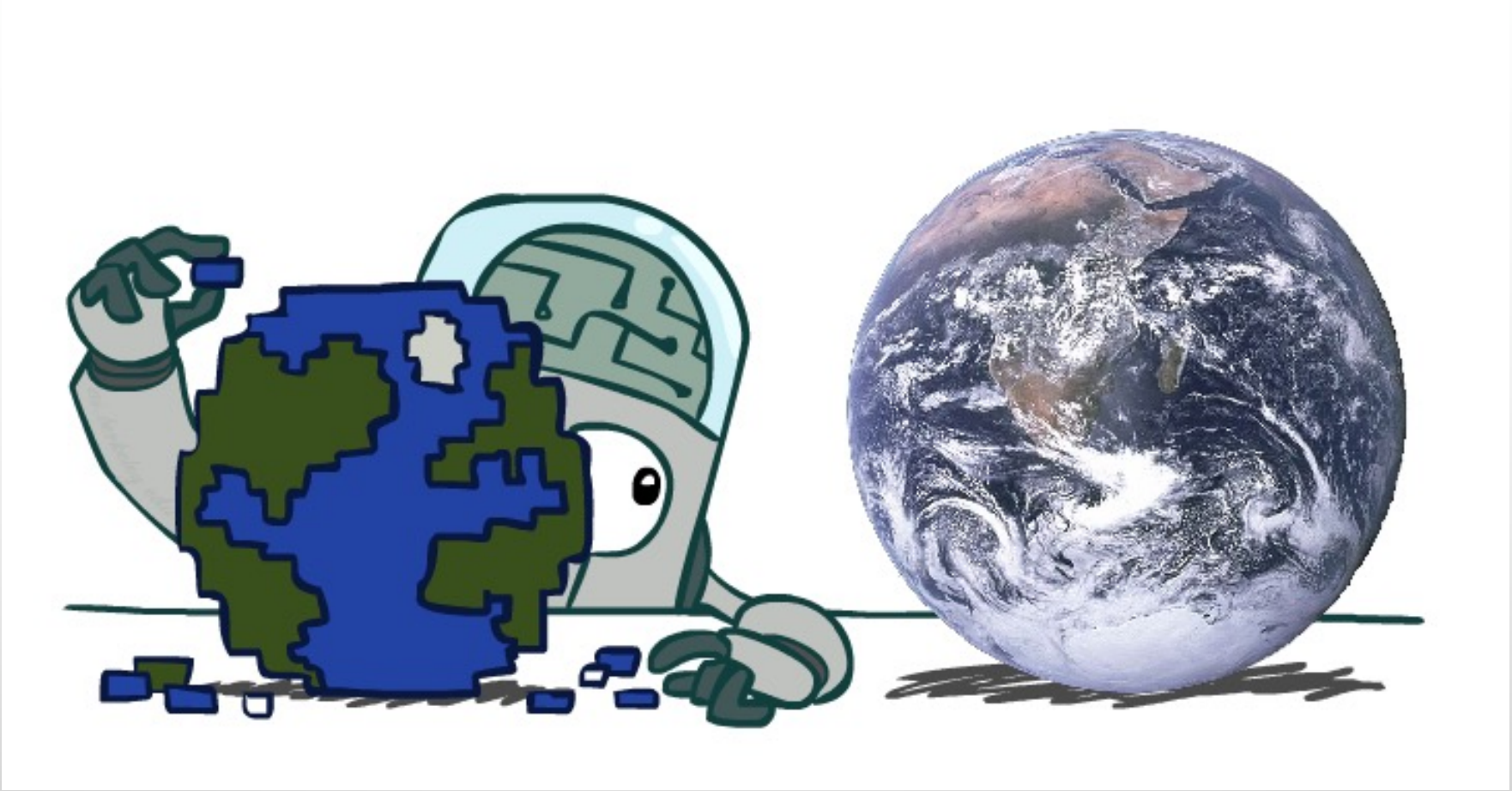
Informed Probabilities

- Let's say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise
- Question: what tree search should you use?



- Answer: Expectimax!
  - To figure out EACH chance node's probabilities, you have to run a simulation of your opponent
  - This kind of thing gets very slow very quickly
  - Even worse if you have to simulate your opponent simulating you…
  - … except for minimax, which has the nice property that it all collapses into one game tree

Modeling Assumptions



The Dangers of Optimism and Pessimism

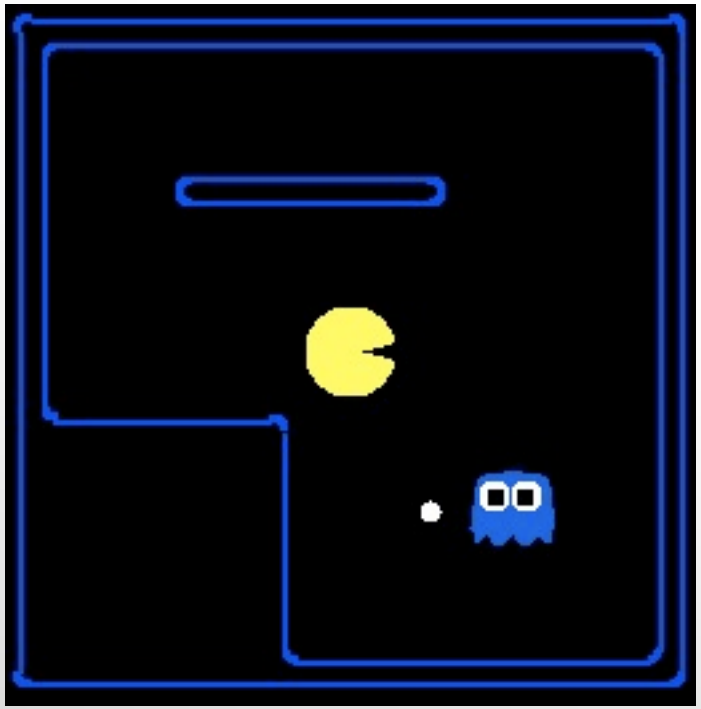
Dangerous Optimism

Assuming chance when the world is adversarial

Dangerous Pessimism

Assuming the worst case when it's not likely

Assumptions vs. Reality

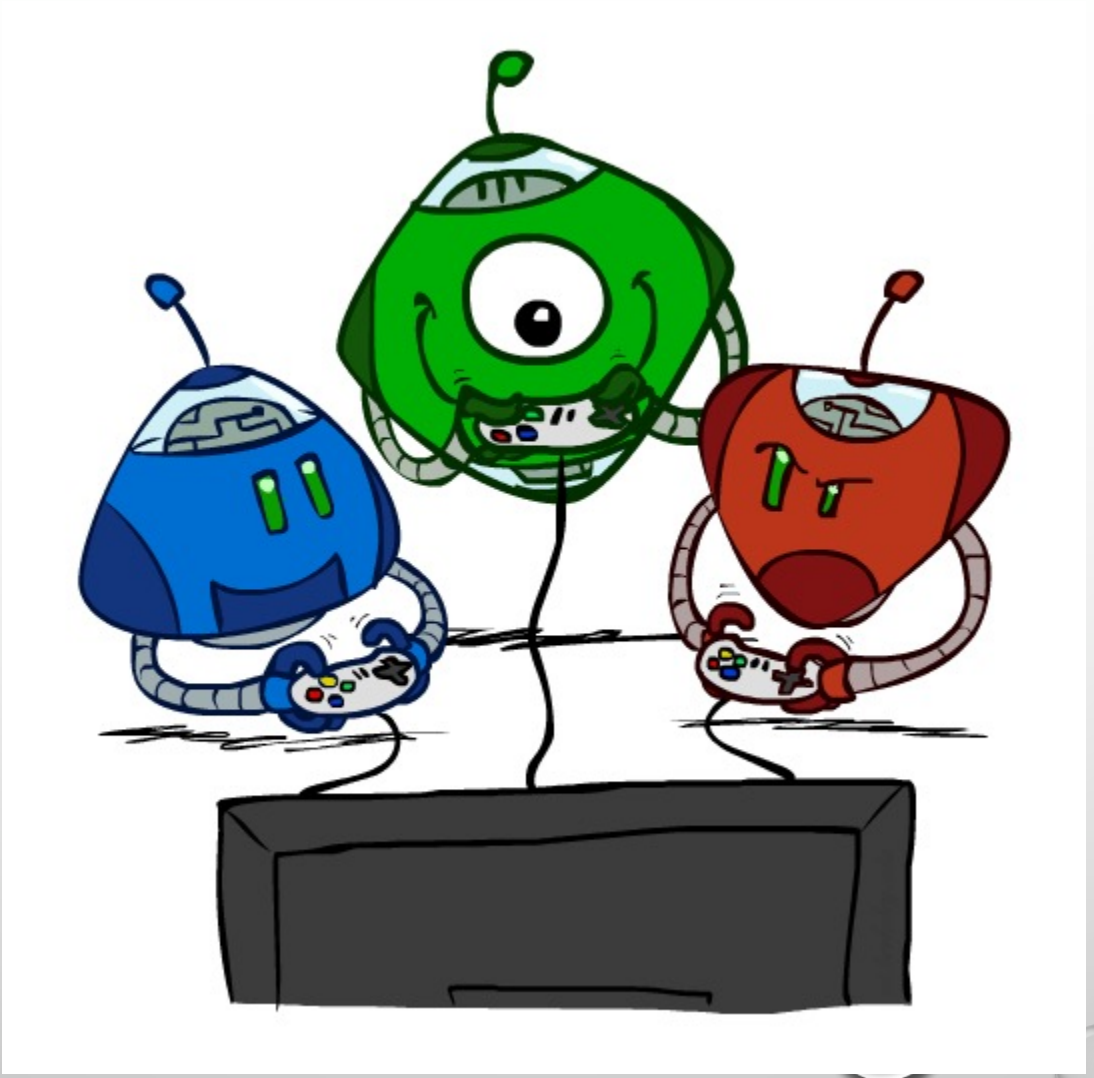


	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

Results from playing 5 games

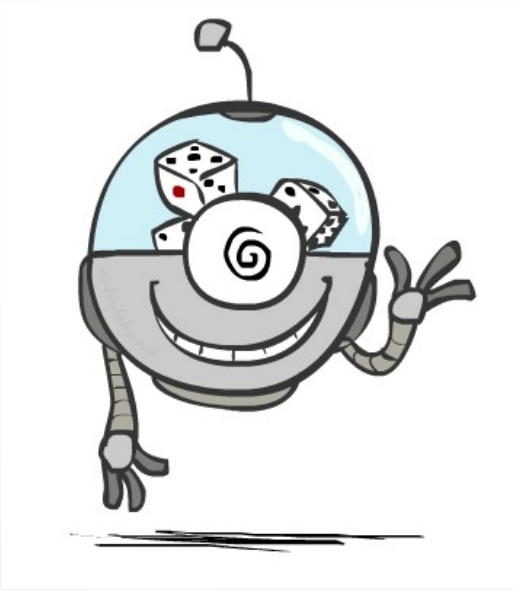
Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

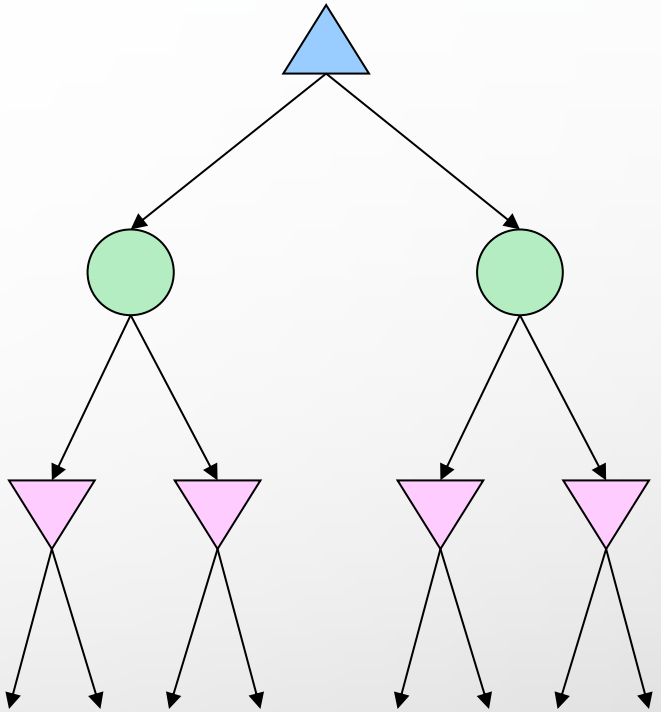
Other Game Types

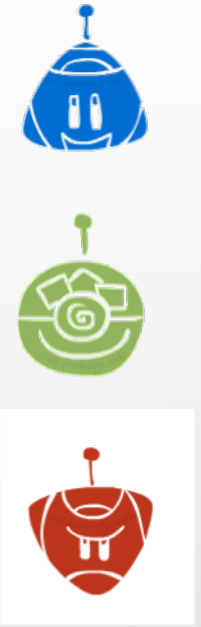


Mixed Layer Types

- e.g. Backgammon
- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children







Example: Backgammon

- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon ≈ 20 legal moves
  - Depth 2 = 20 x $(21 \times 20)^3 = 1.2 \times 10^9$

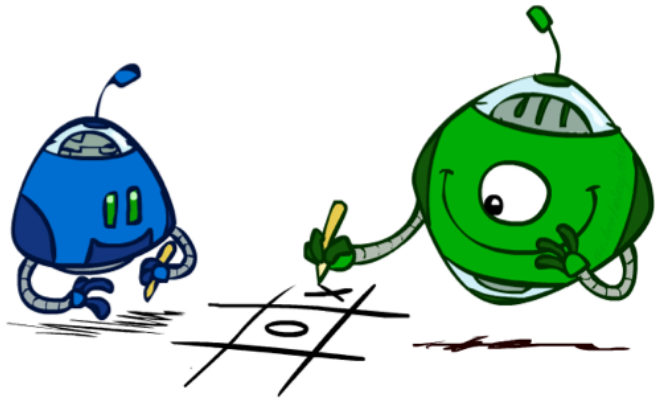
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier…

- Historic AI: TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play

- 1st AI world champion in any game!

Image: Wikipedia

Multi-Agent Utilities



- What if the game is not zero-sum, or has multiple players?

- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and Competition dynamically…