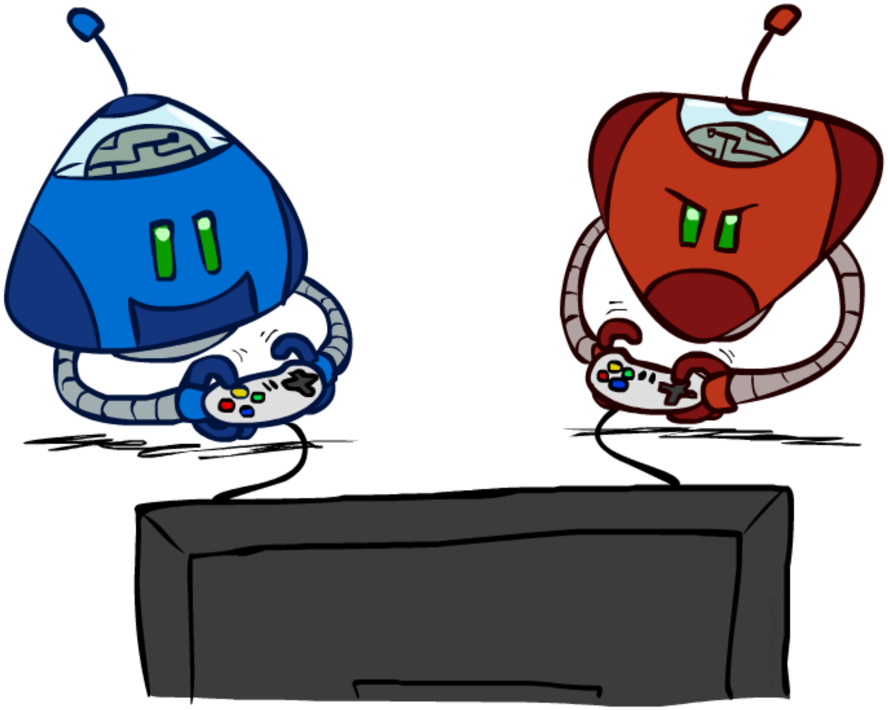
Artificial Intelligence

Adversarial Search



Logistics

- Midterm
  - Nov. 2nd, in class test
  - All questions will be related/similar to quiz/homework
- Final project
  - Will be released soon
- Final exam examples
  - Canvas

Course Topics

Search problems

Markov decision processes

Adversarial games

Constraint satisfaction problems

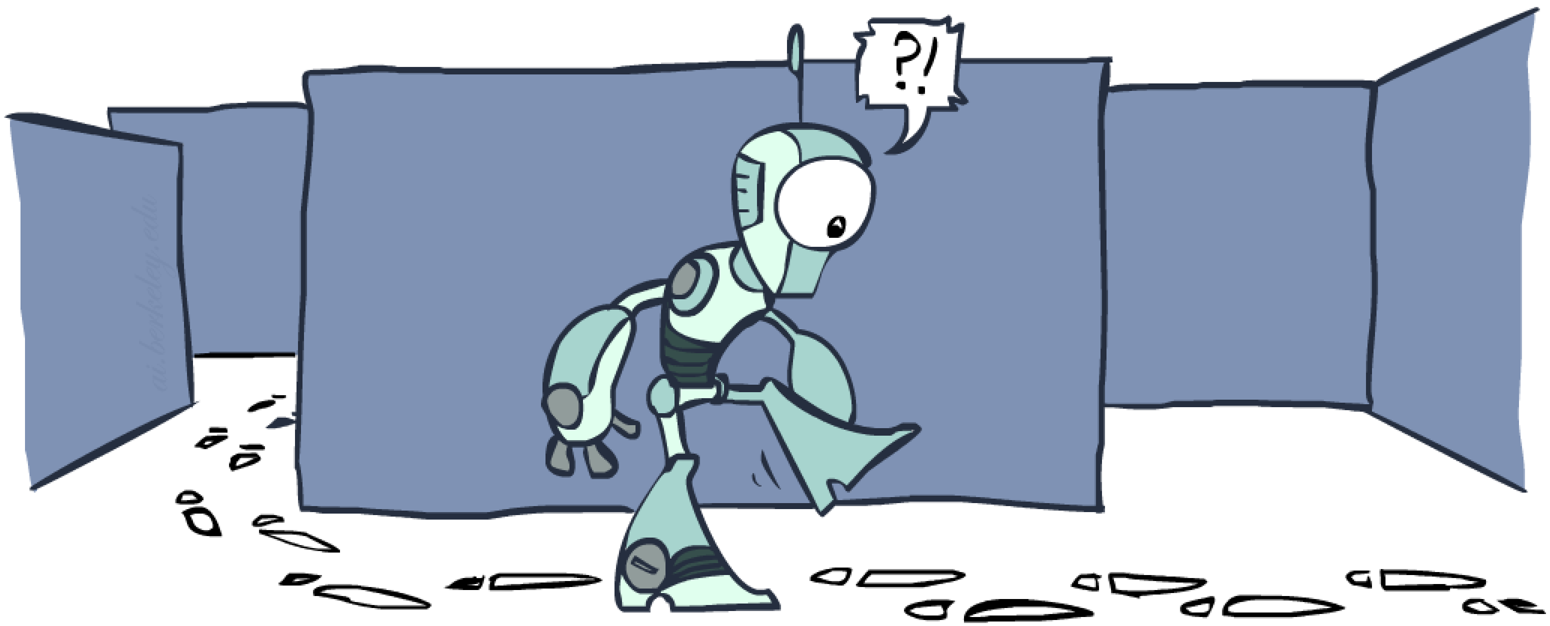
Bayesian networks

Reflex **States** **Variables** **Logic**

"Low-level intelligence" "High-level intelligence"

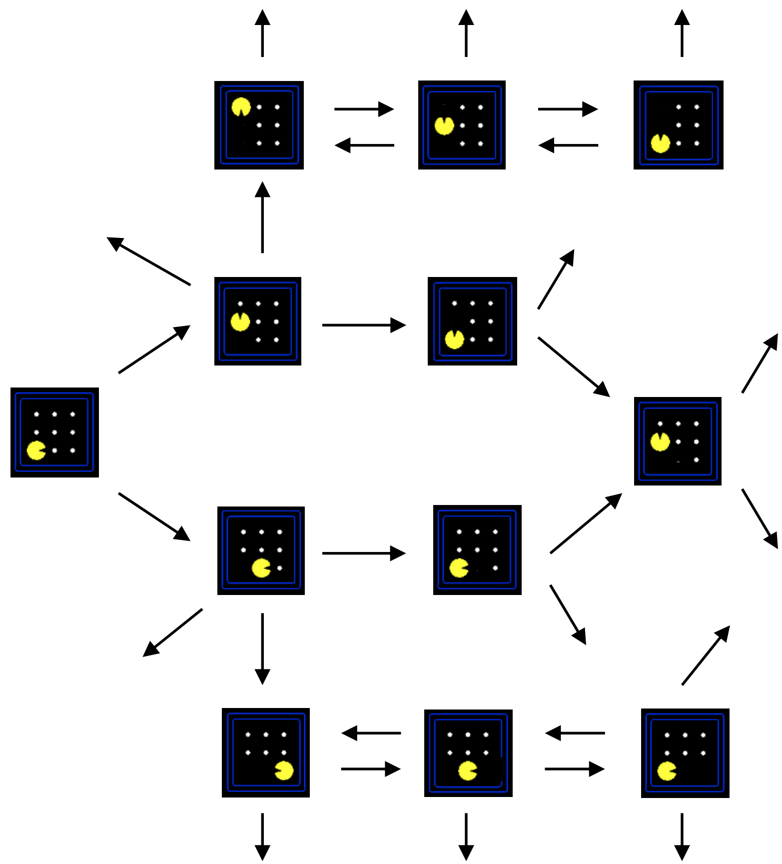
Machine learning

Graph Search



State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

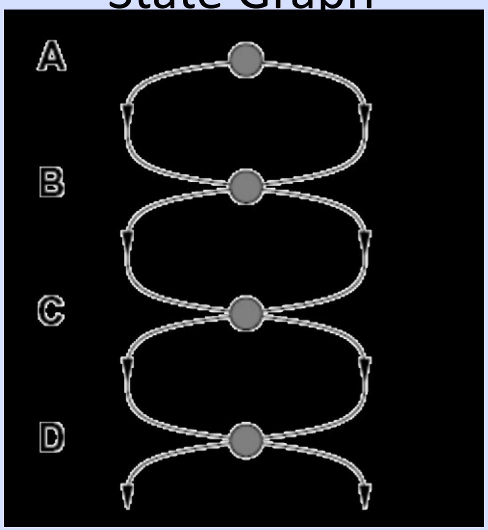


World states? $120 \mathrm{x} (2^{30}) \mathrm{x} (12^2) \mathrm{x} 4$

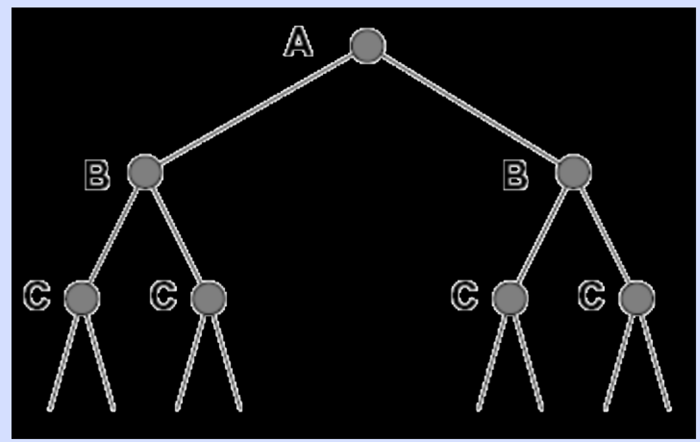
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.

State Graph



Search Tree



Graph Search

- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
    - Before expanding a node, check to make sure its state has never been expanded before
    - If not new, skip it, if new add to closed set
- Important: store the closed set as a set, not a list

Tree/ Graph Search Pseudo-Code

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

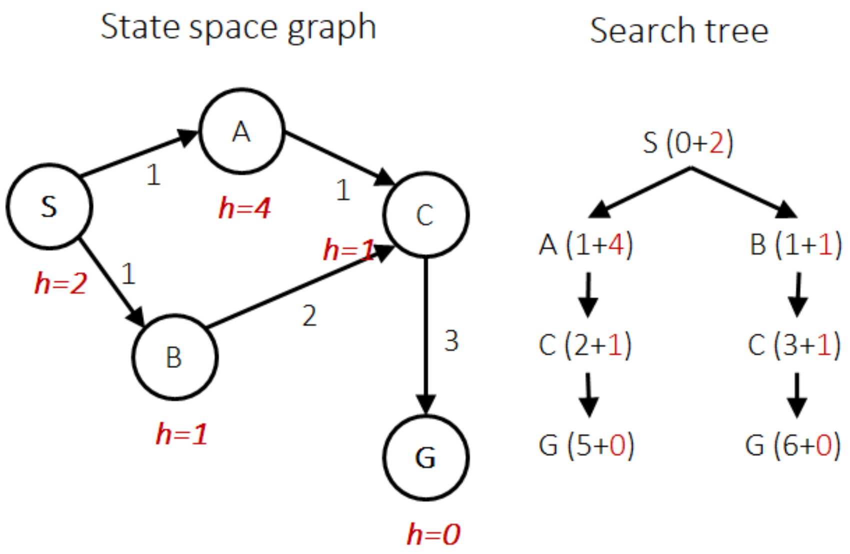
```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

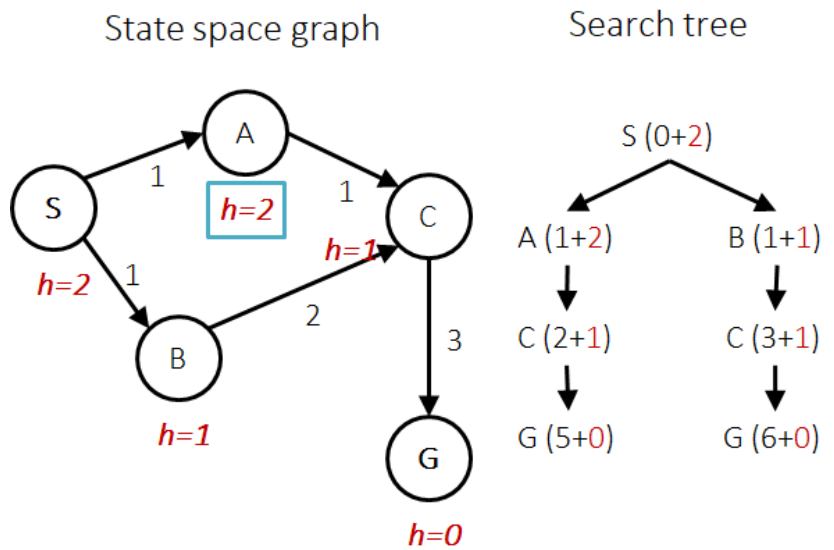
Consistency of Heuristics

- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)

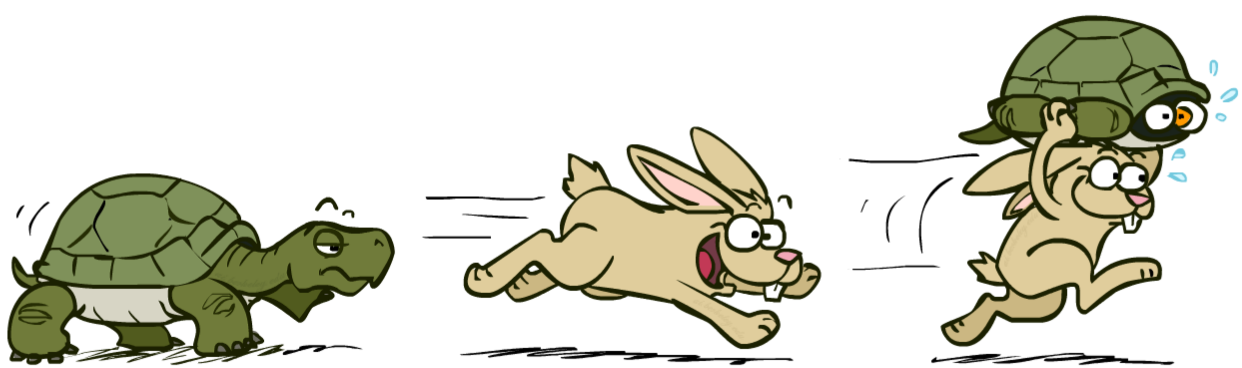
  - A* graph search is optimal





A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



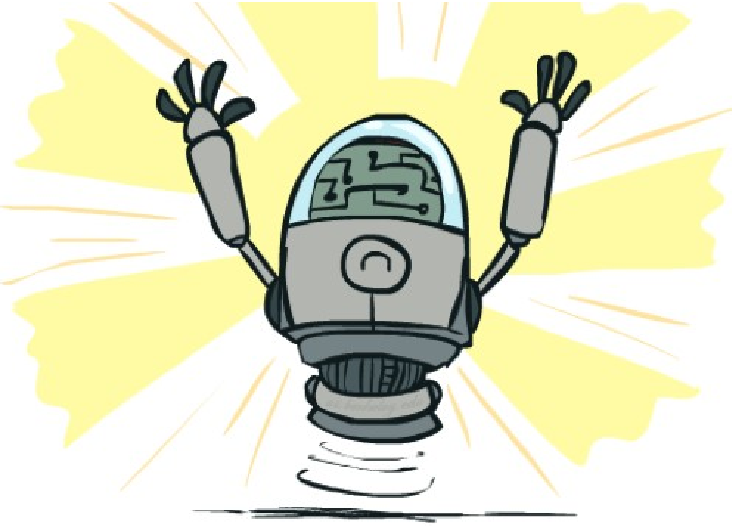
Optimality

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)

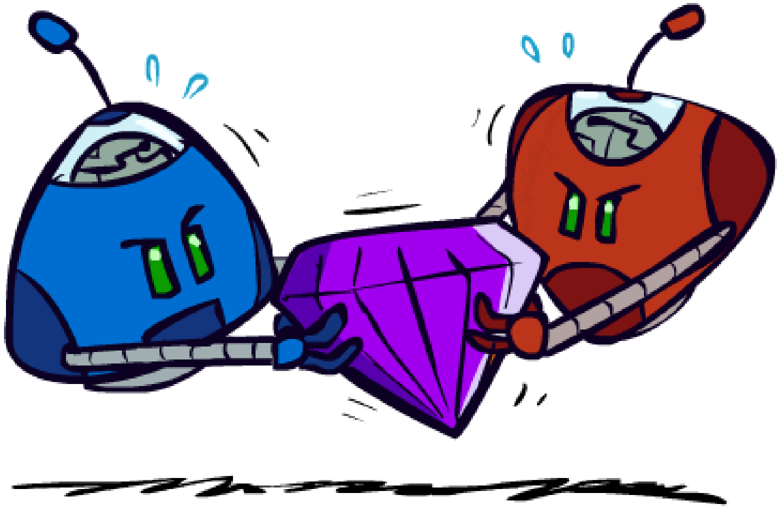
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)

- Consistency implies admissibility

- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems

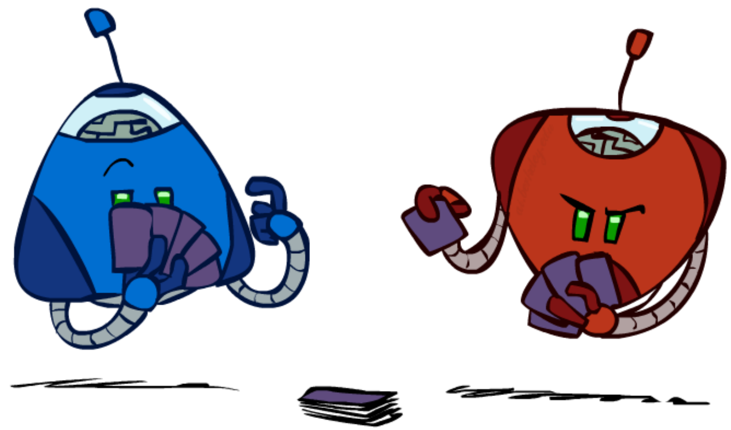


Adversarial Games



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)?

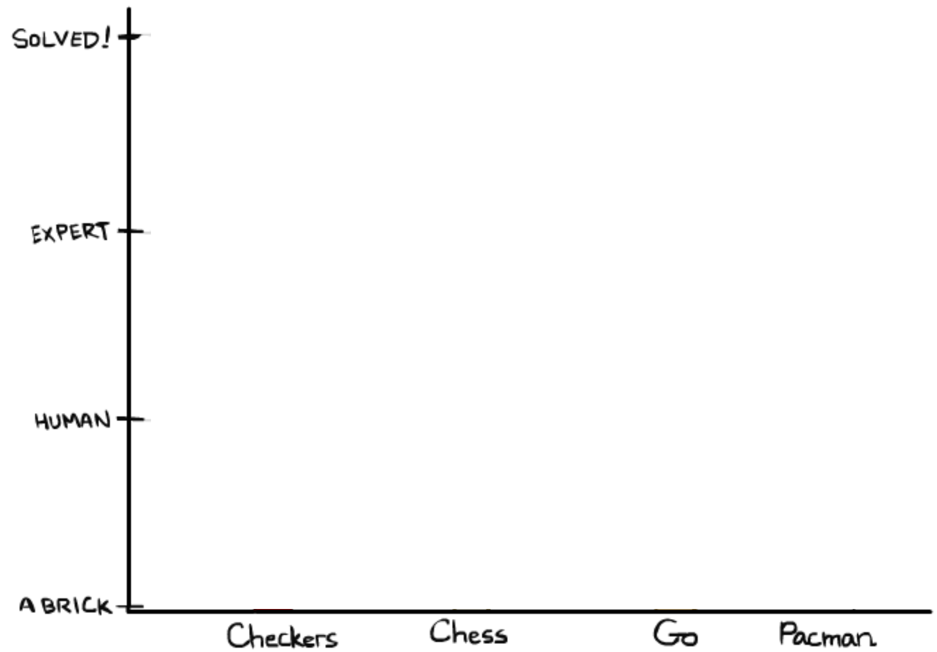
Game Playing State-of-the-Art

- **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!

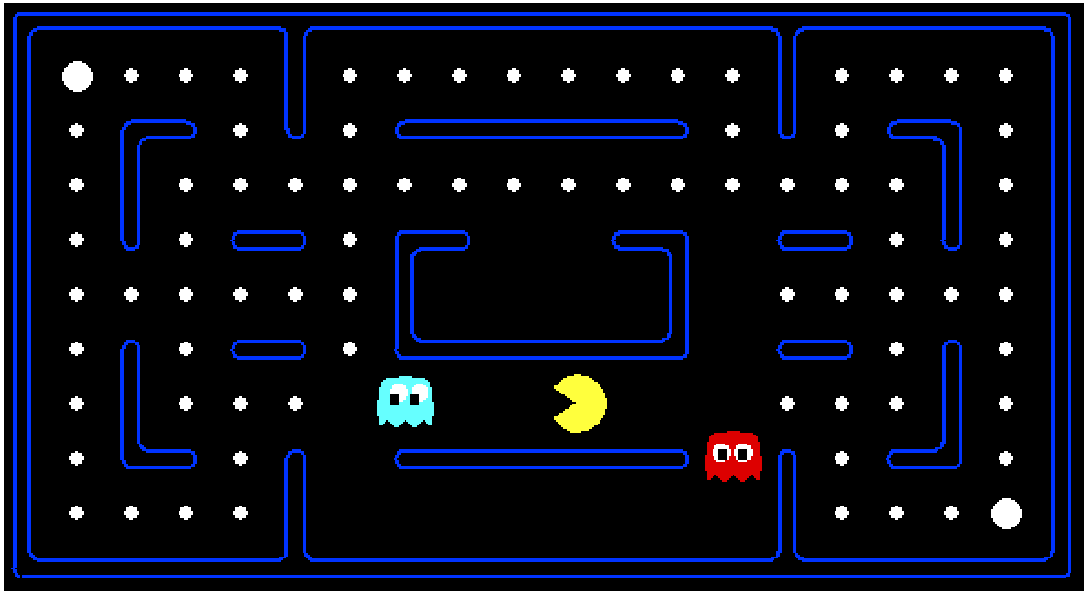
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match.

- **Go:** 2016/2017: AlphaGo defeats Korean and Chinese champions with reinforcement learning + deep learning.

- **Pacman**

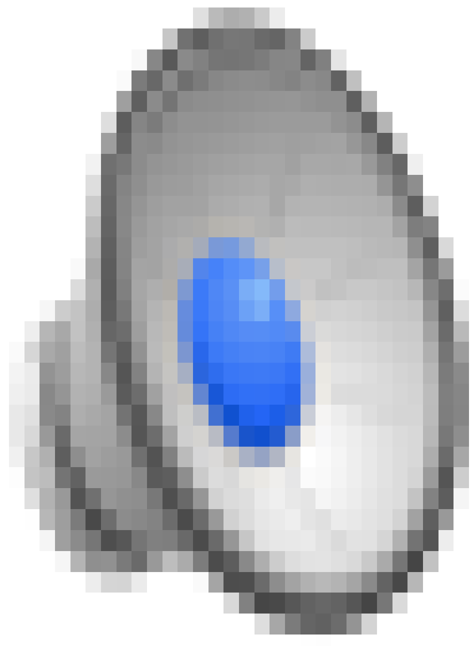


Behavior from Computation

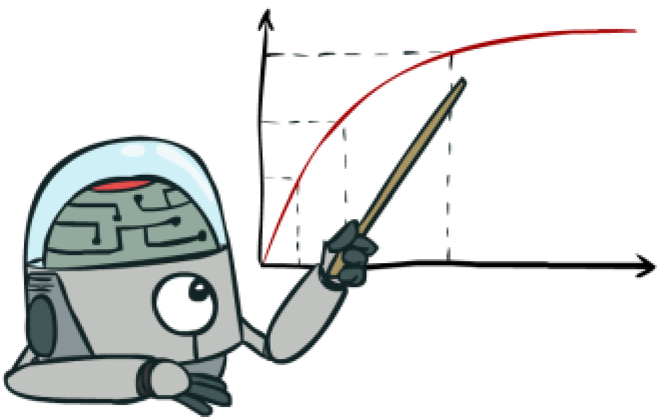


[Demo: mystery pacman (L6D1)]

Video of Demo Mystery Pacman



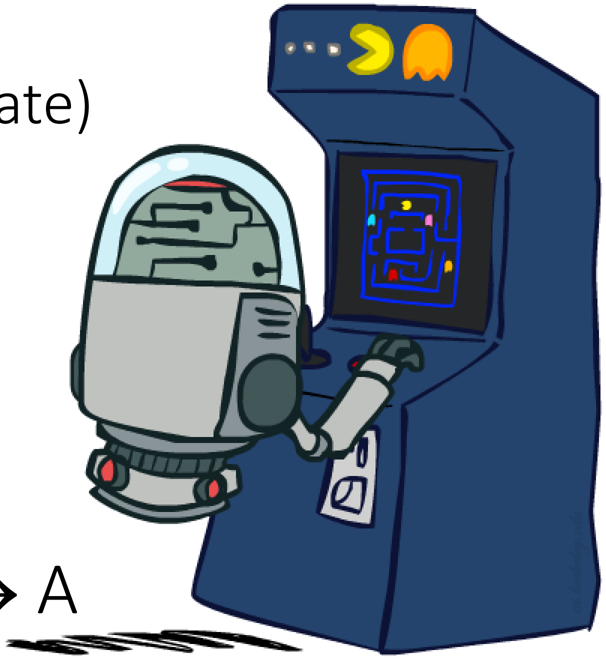
Rational Agent:
Maximize Your Expected Utility



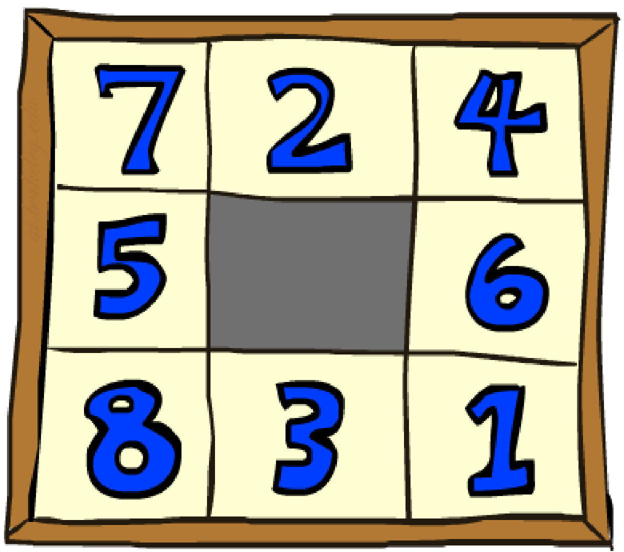
Act rationally: Want algorithms for calculating a strategy (policy) which recommends a move from each state

Deterministic Games

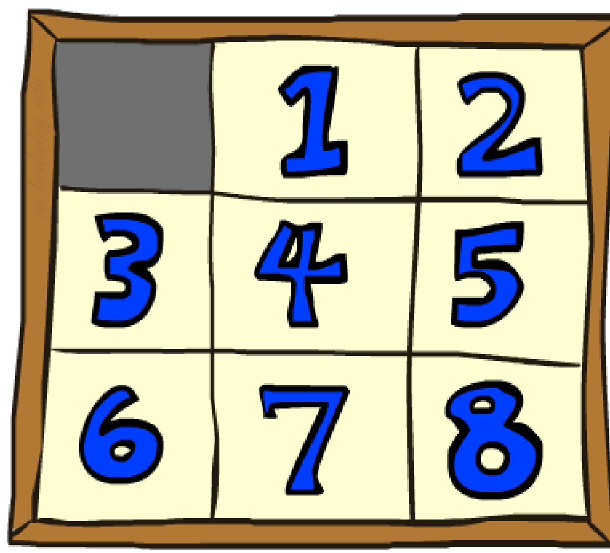
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {true,false}
  - Terminal Utilities: SxP $\rightarrow$ R



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

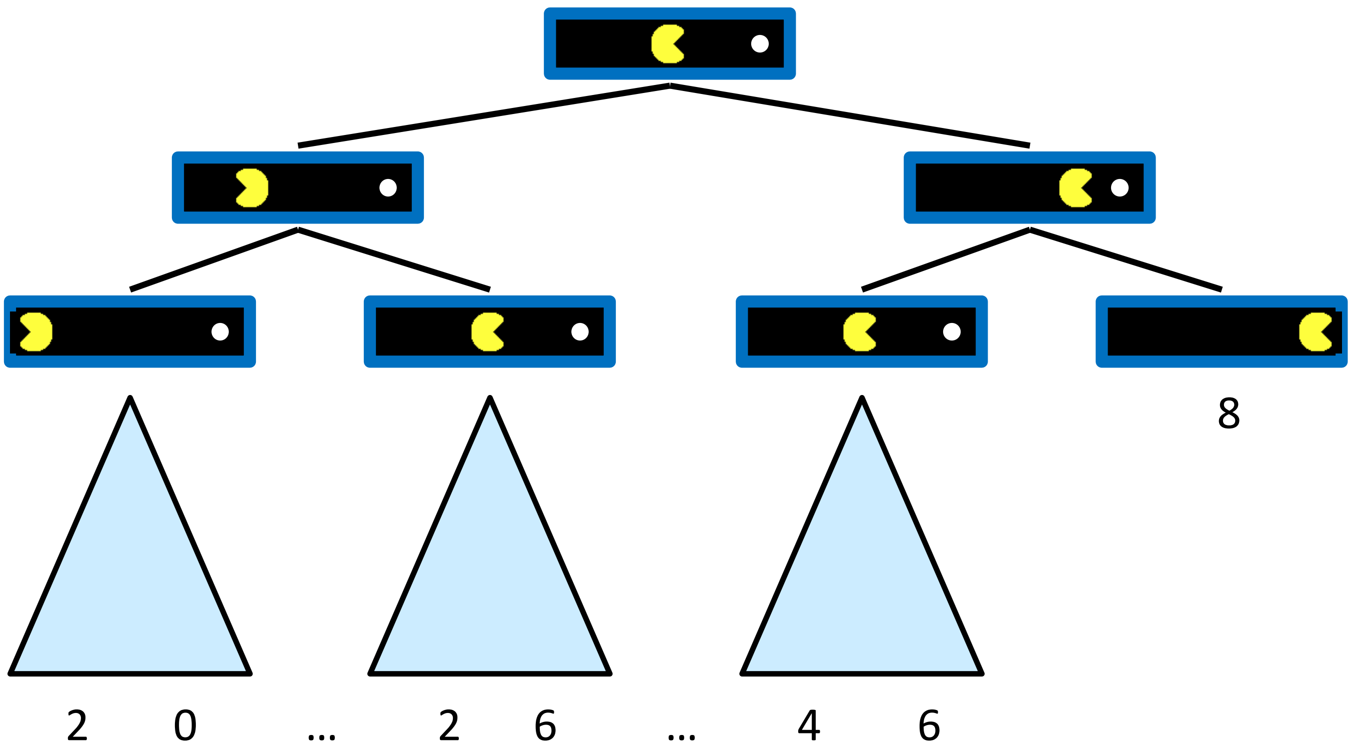


Start State



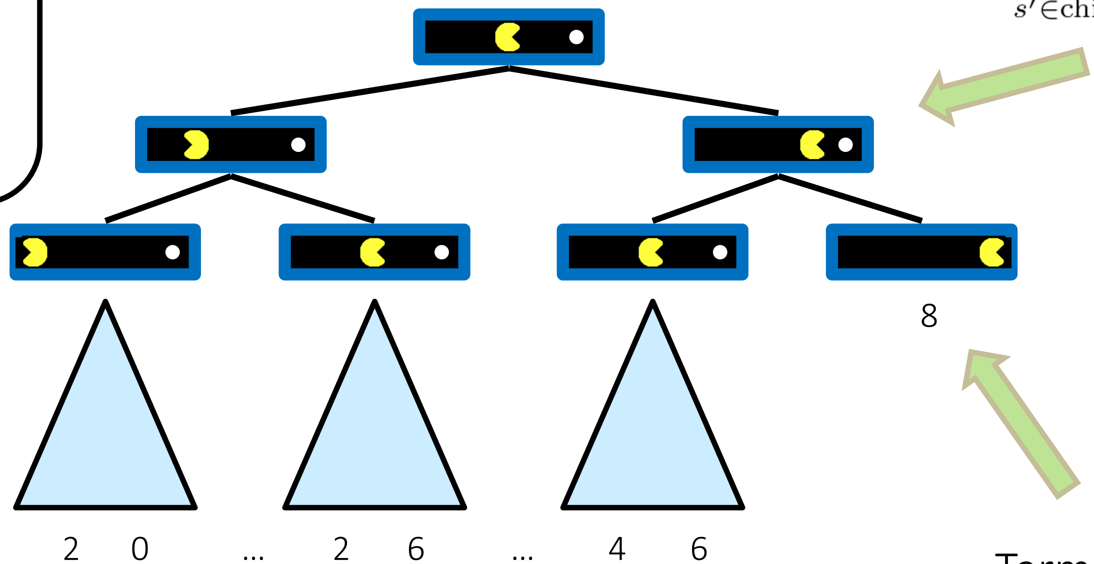
Goal State

Single-Agent Trees



Value of a State

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



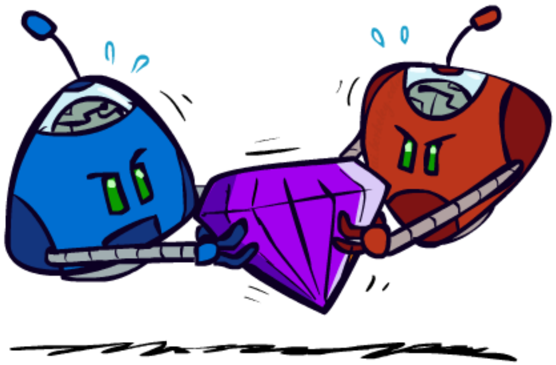
Non-Terminal States:

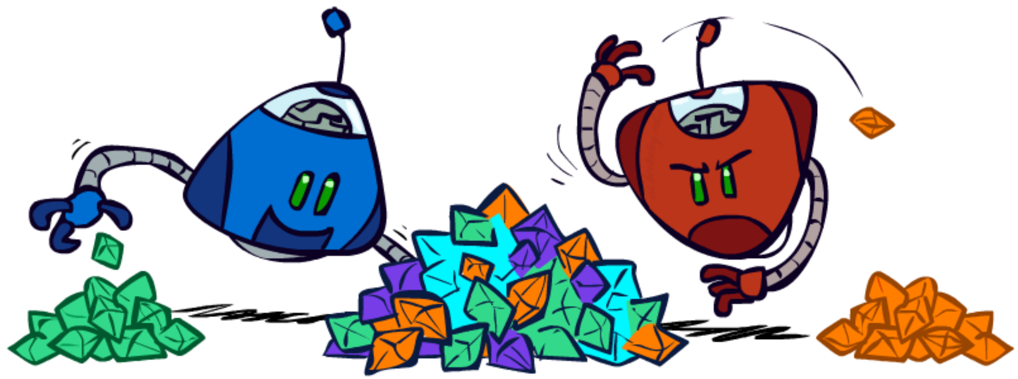
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

Terminal States:

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

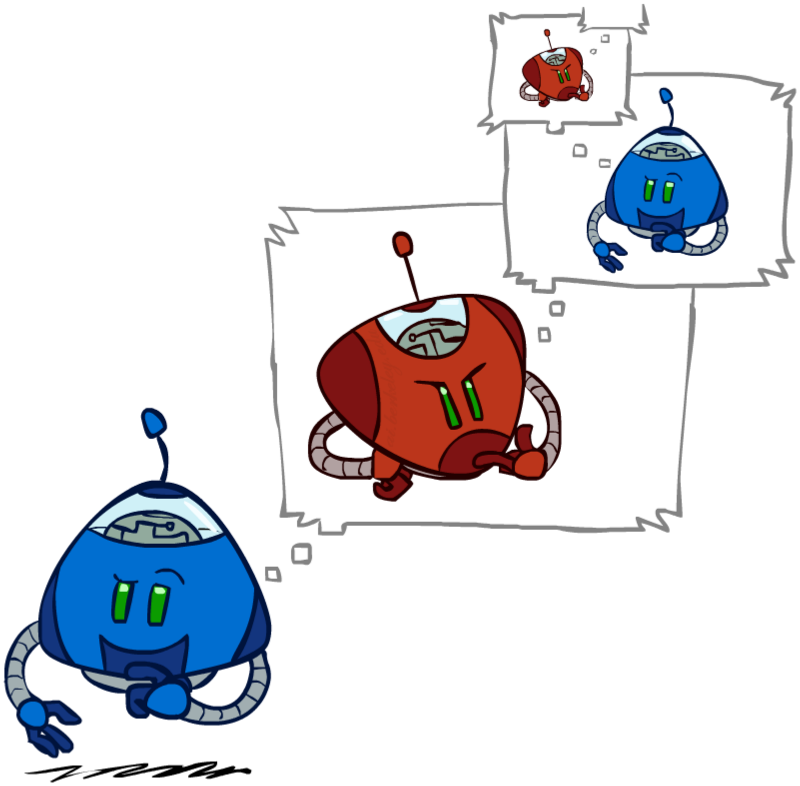
Zero-Sum Games



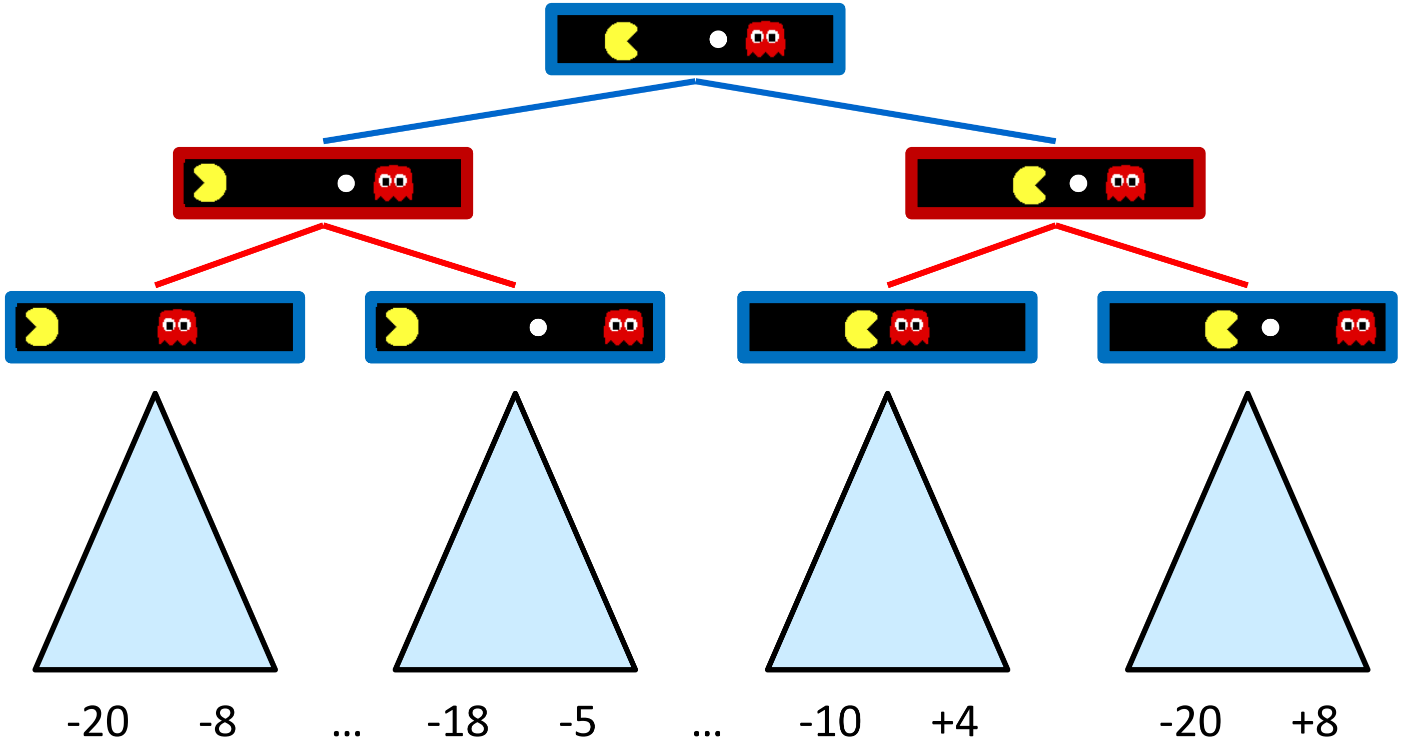


- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition,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

Adversarial Search



Adversarial Game Trees



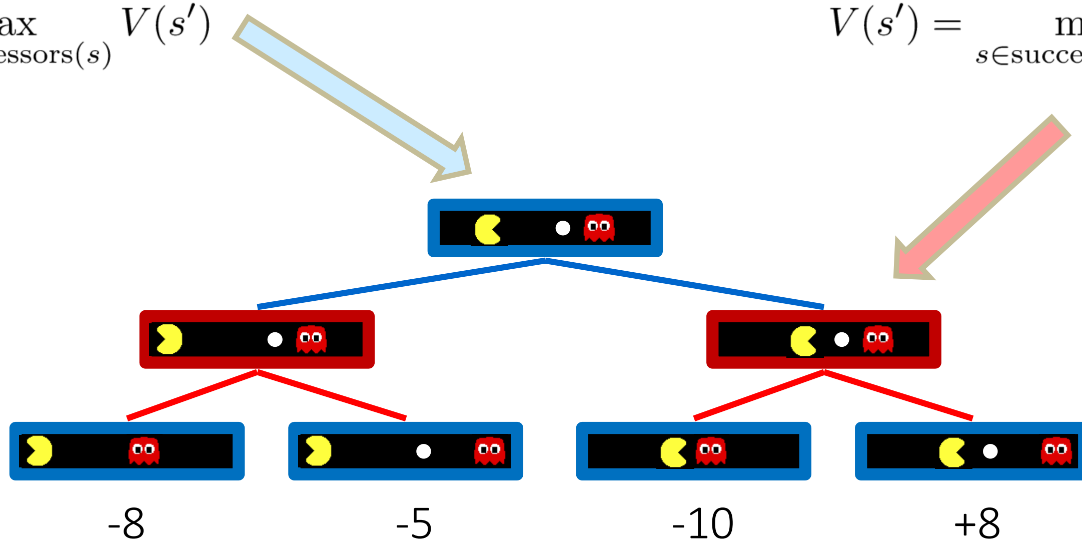
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

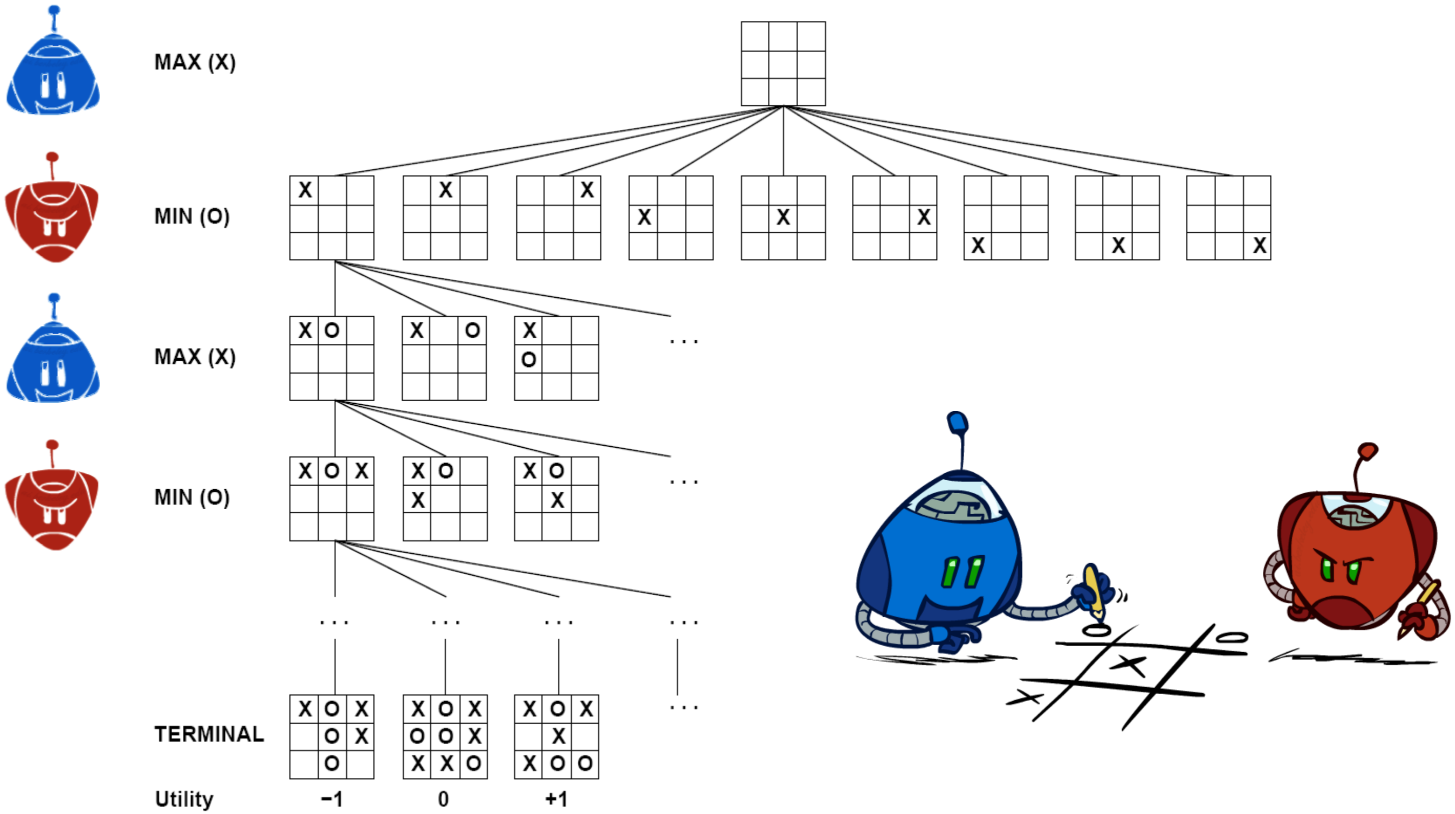
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

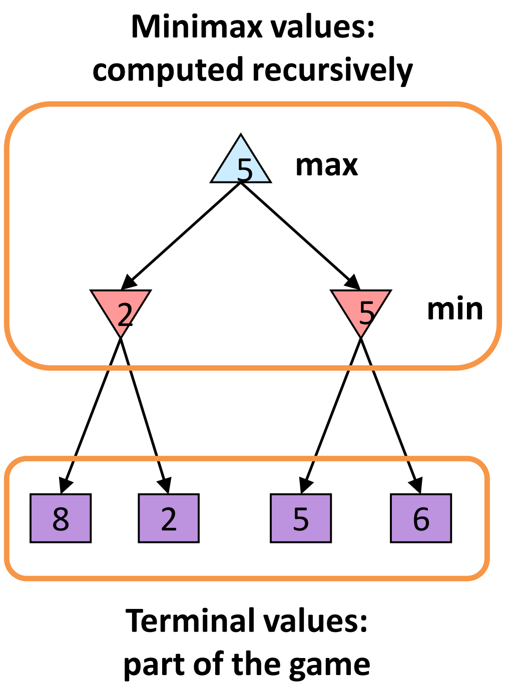
Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

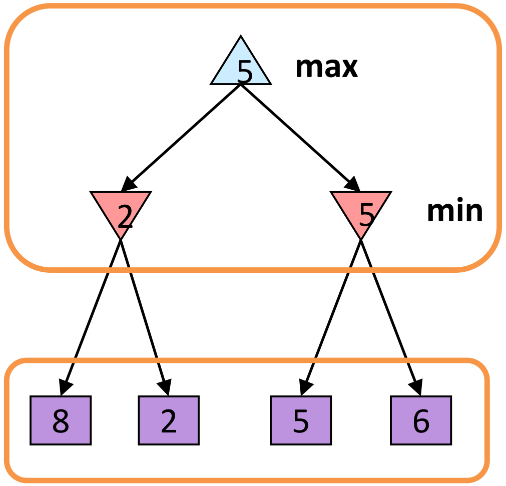
- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- 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

Minimax values:
computed recursively



Terminal values:
part of the game

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

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

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

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

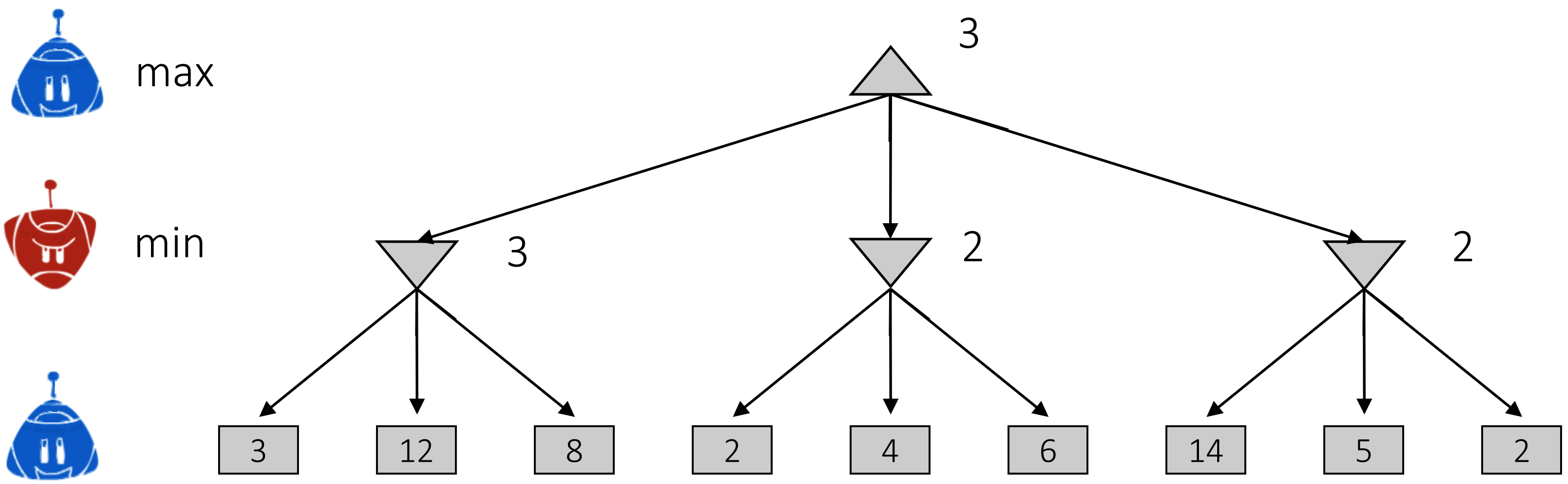
Minimax Implementation (Dispatch)

```
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)
```

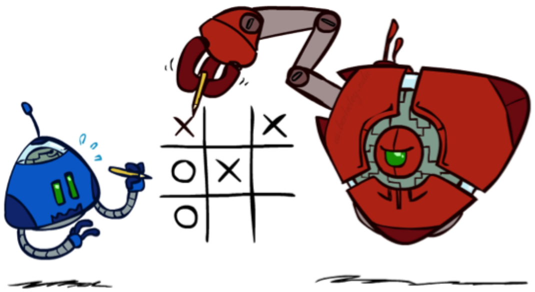
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

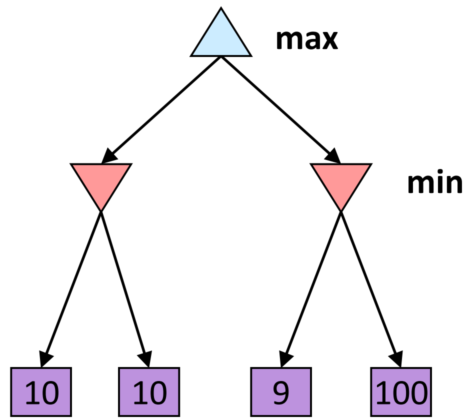
```
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

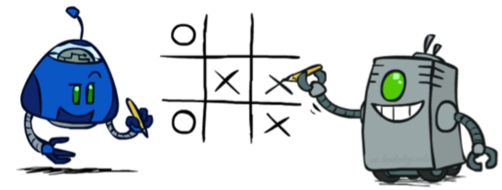
Minimax Example



Minimax Properties



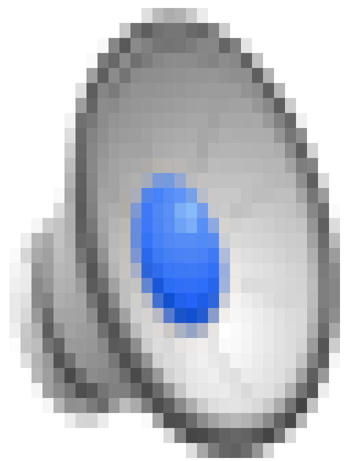




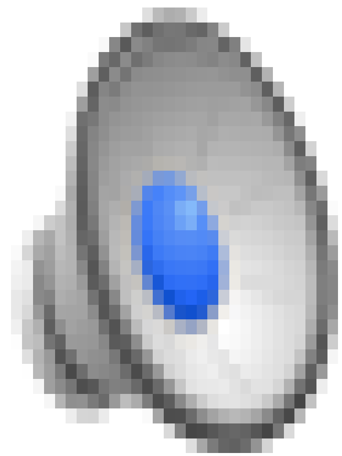
Optimal against a perfect player. Otherwise?

[Demo: min vs exp (L6D2, L6D3)]

Video of Demo Min vs. Exp (Min)

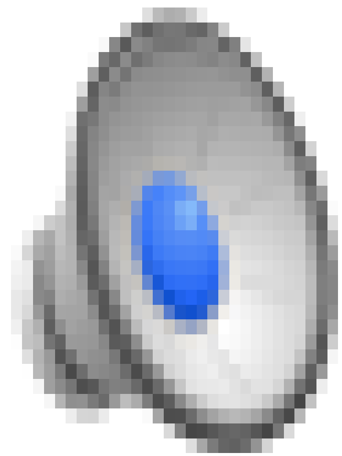


Video of Demo Min vs. Exp (Exp)



Video of Demo Smart Ghosts (Coordination)

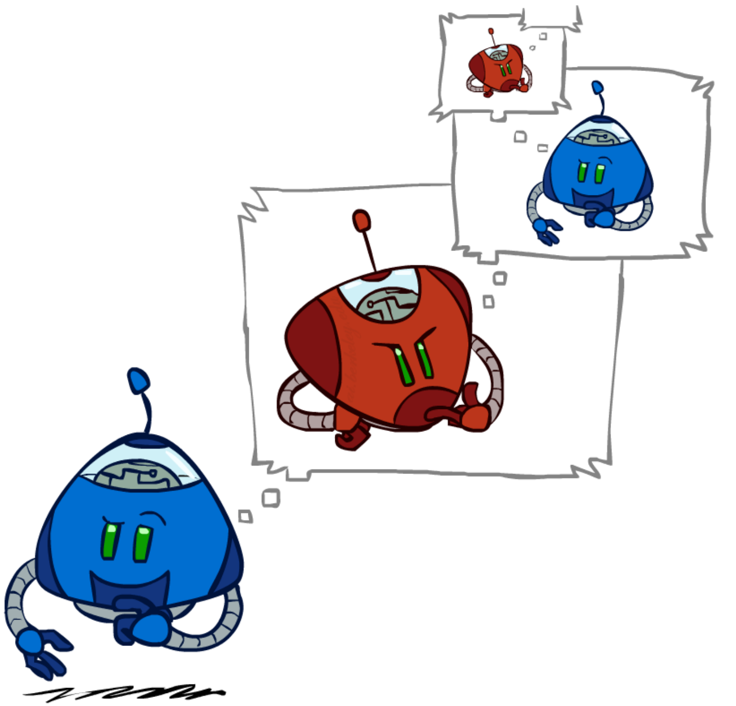
Video of Demo Smart Ghosts (Coordination) – Zoomed In



Minimax Efficiency

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

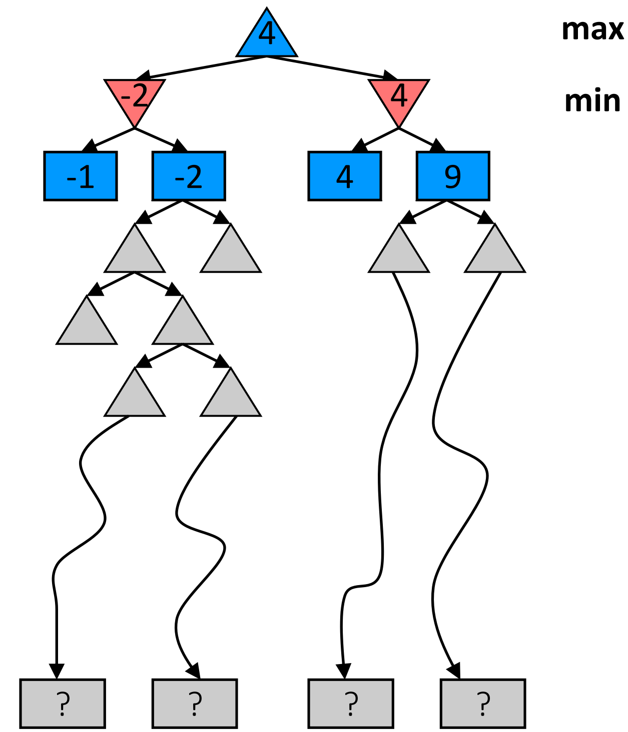
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



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

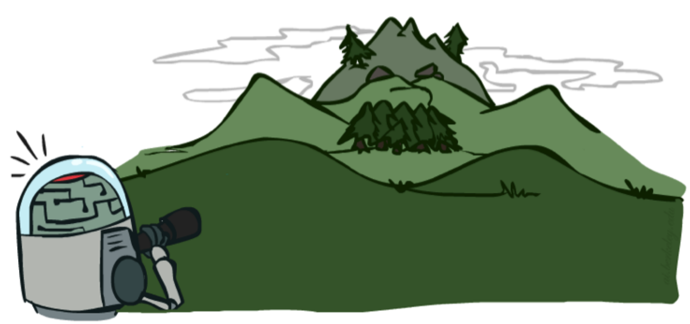


Depth Matters

- Evaluation functions are always imperfect

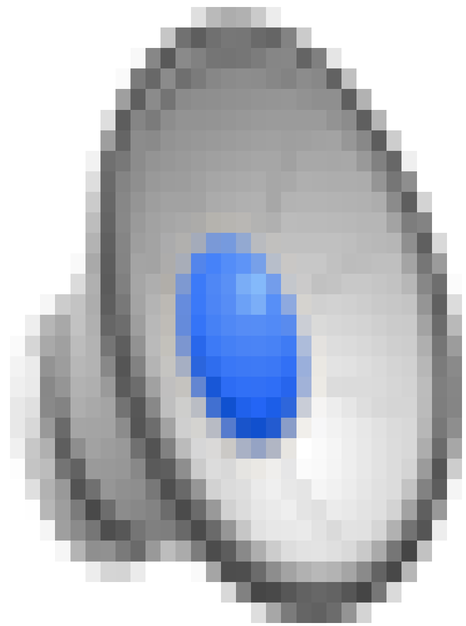
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters

- An important example of the tradeoff between complexity of features and complexity of computation

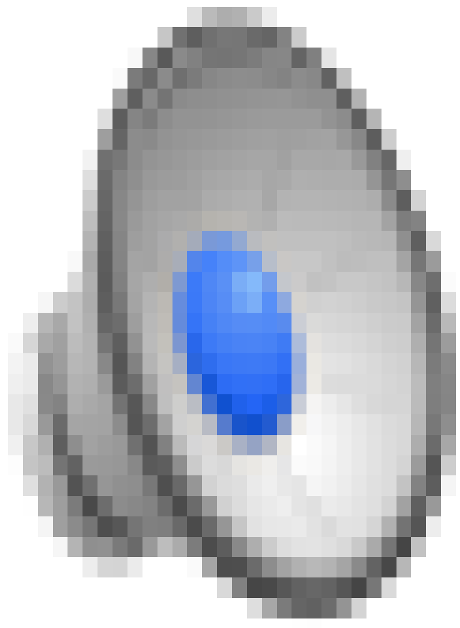


[Demo: depth limited (L6D4, L6D5)]

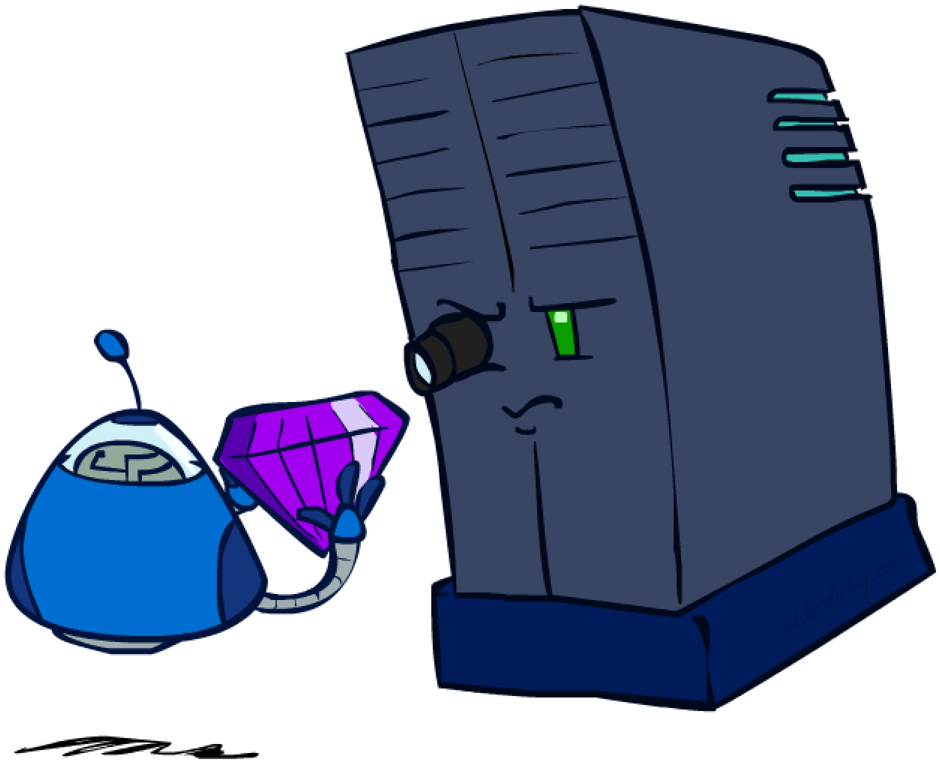
Video of Demo Limited Depth (2)



Video of Demo Limited Depth (10)

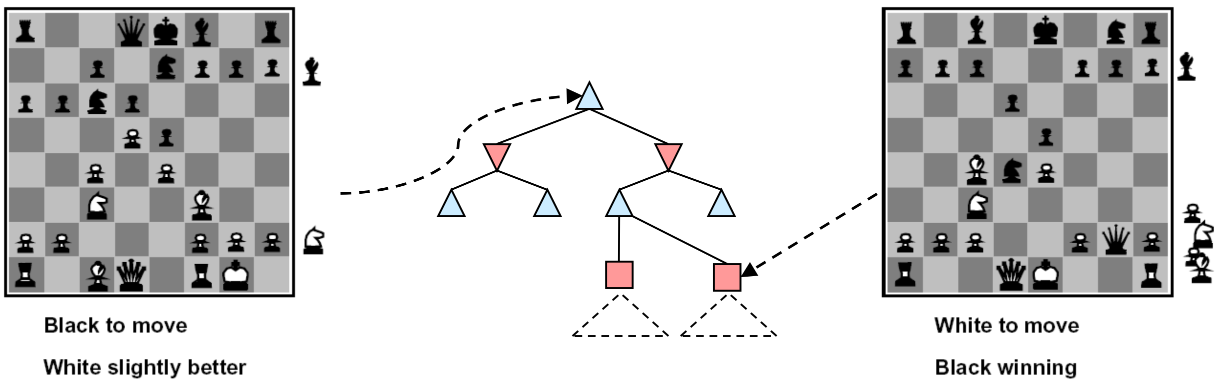


Evaluation Functions



Evaluation Functions

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



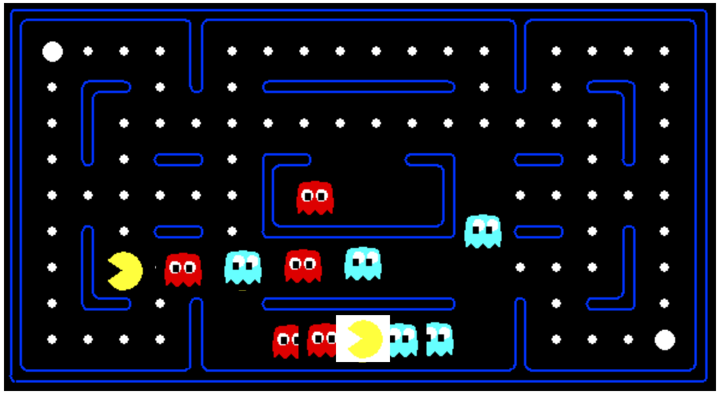
- Ideal function: returns the actual minimax value of the position

- In practice: typically weighted linear sum of features:

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

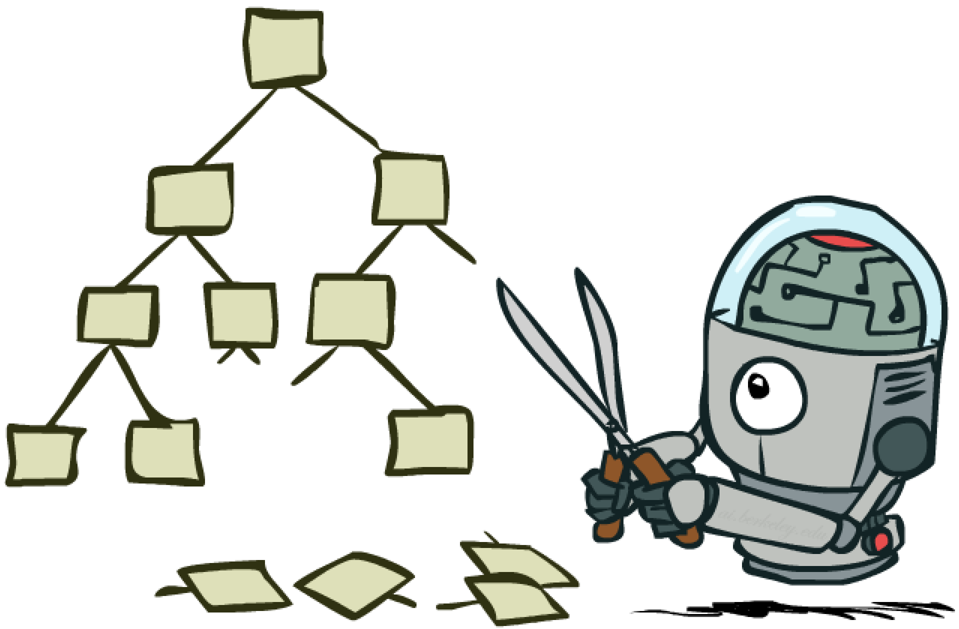
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Evaluation for Pacman

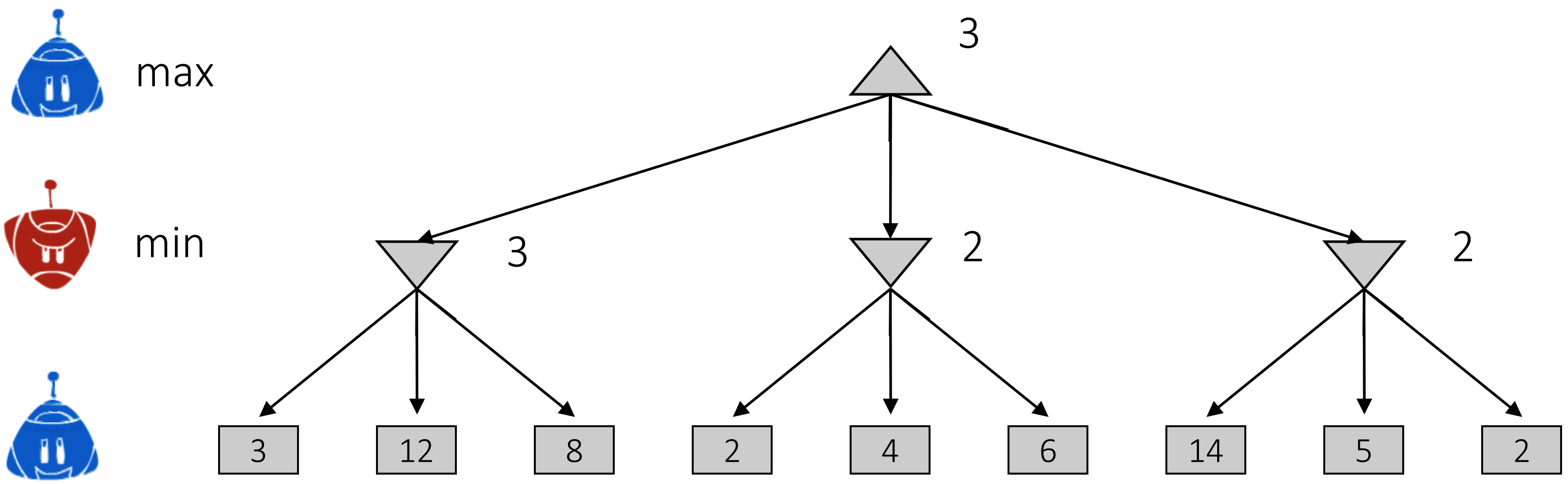


[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate (L6D6,7,8,10)]

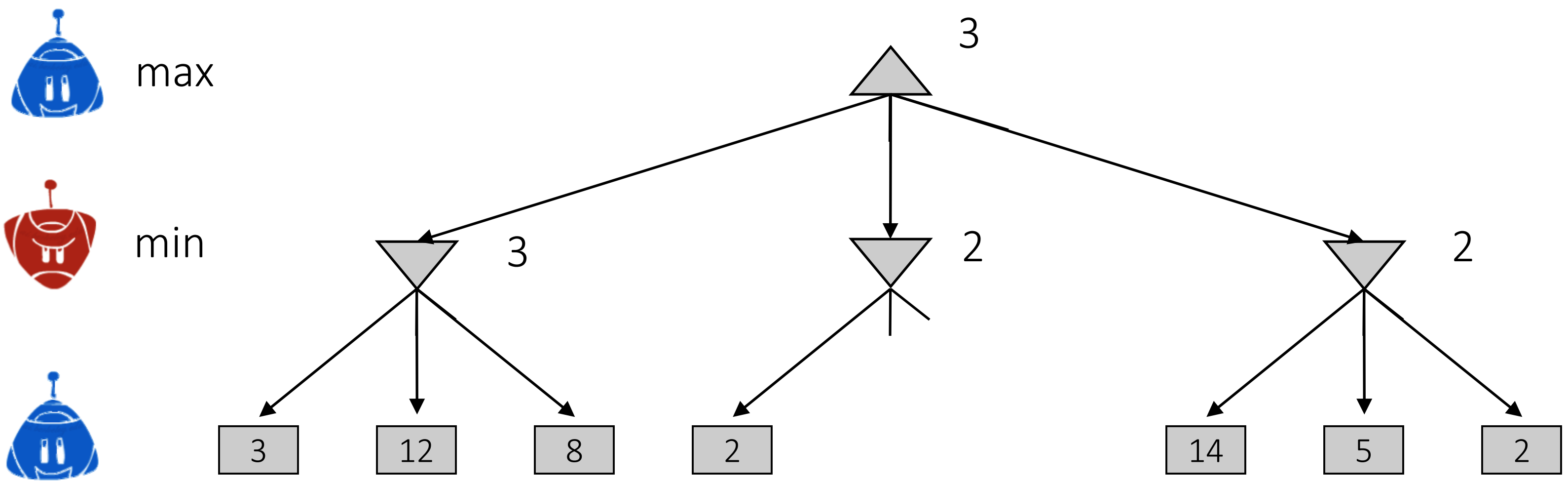
Game Tree Pruning



Minimax Example

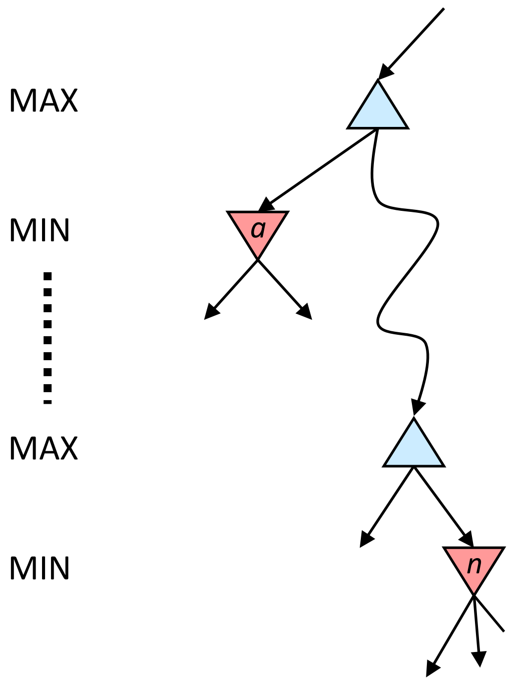


Minimax Pruning



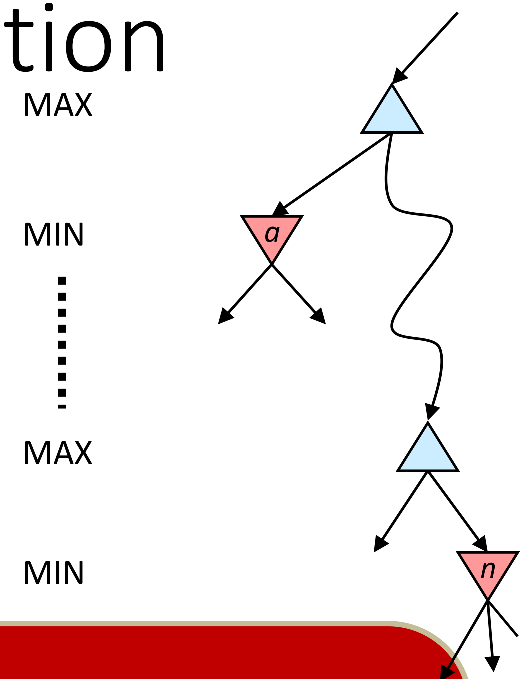
Alpha-Beta Pruning

- 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 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



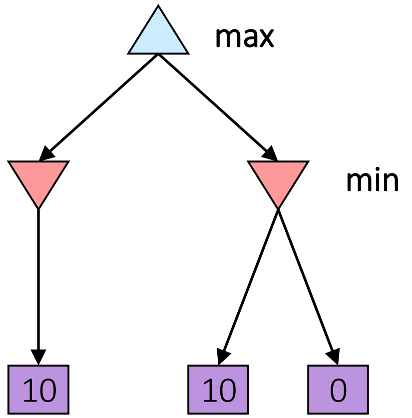
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

```
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!
- 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…
- 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



- This is a simple example of metareasoning (computing about what to compute)

Course Topics

Search problems

Markov decision processes

Adversarial games

Constraint satisfaction problems

Bayesian networks

Reflex **States** **Variables** **Logic**

"Low-level intelligence" "High-level intelligence"

Machine learning