Search with Other Agents

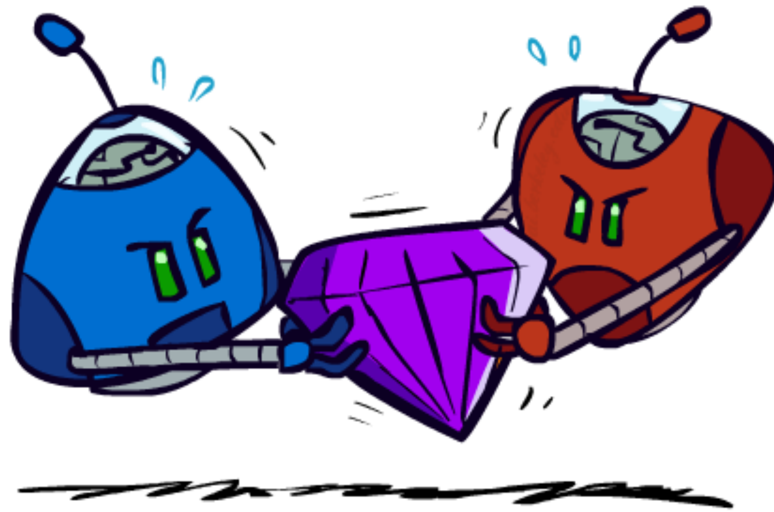
CE417: Introduction to Artificial Intelligence
Sharif University of Technology
Fall 2023

Soleymani

"Artificial Intelligence: A Modern Approach", 3rd Edition, Chapter 5

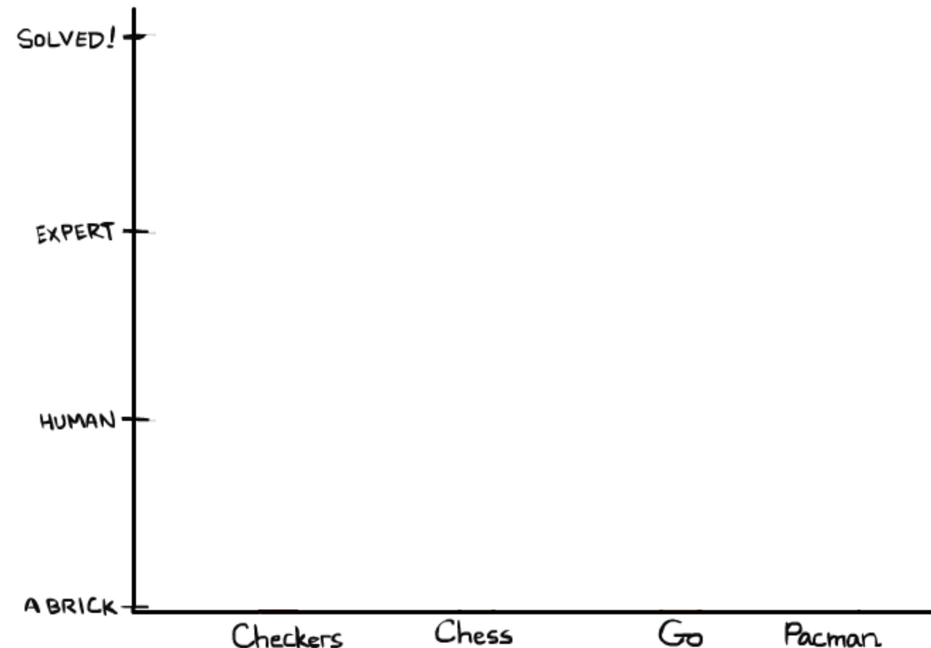
Most slides have been adopted from CS188, UC Berkeley.

Adversarial Games



Game playing progress ☺

- **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:** Human champions are now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

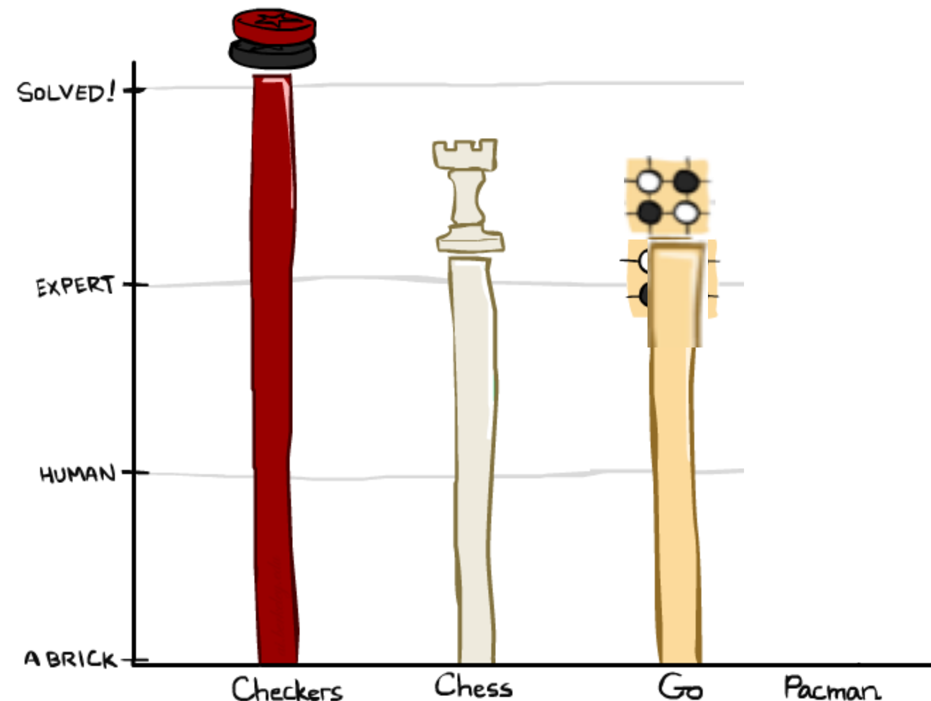


Game playing progress ☺

- **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!

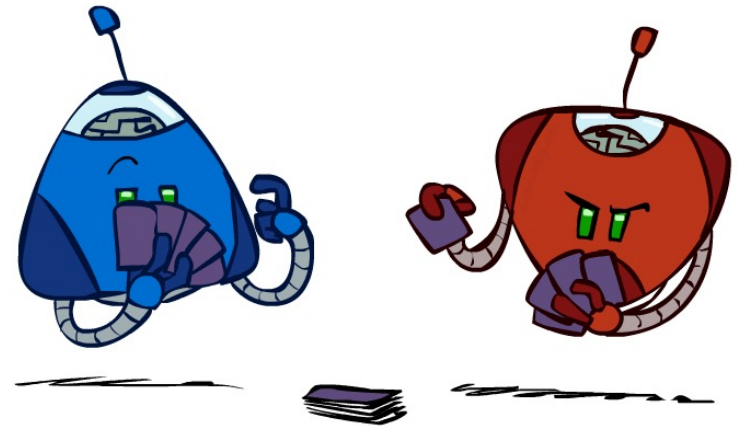
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go :2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function by deep learning.**



Types of games

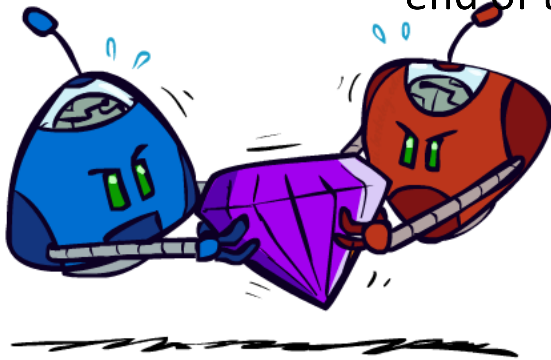
- Many different kinds of games!
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?



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

Zero-Sum games

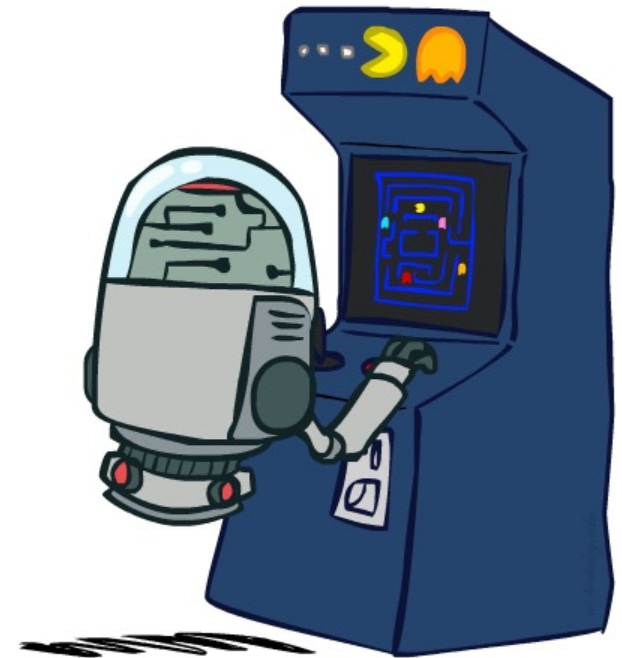
Zero-sum game: agents' goals are in conflict and sum of utility values at the end of the game is zero or constant



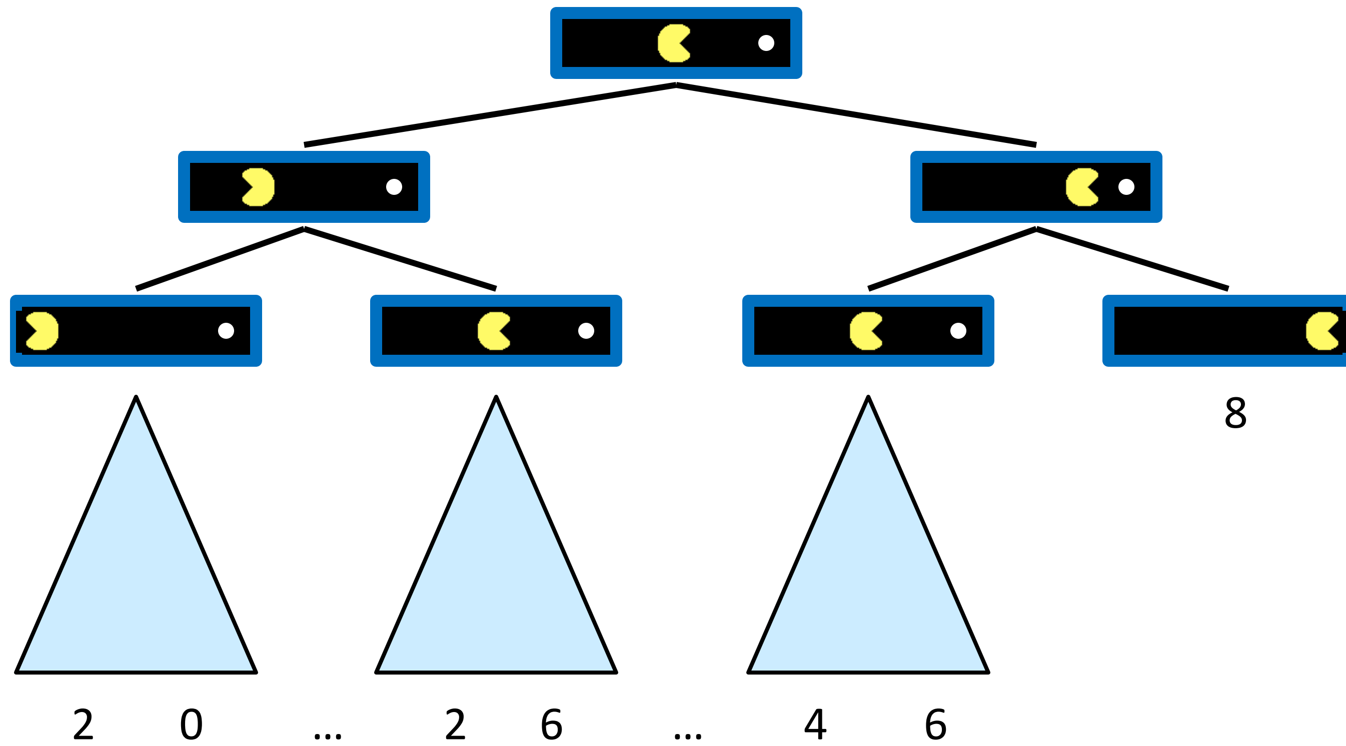
- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition,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

Deterministic games: Adversarial search

- 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 $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R
- Solution for a player is a policy: S $\rightarrow$ A
- Examples: Tic-tac-toe, chess, checkers

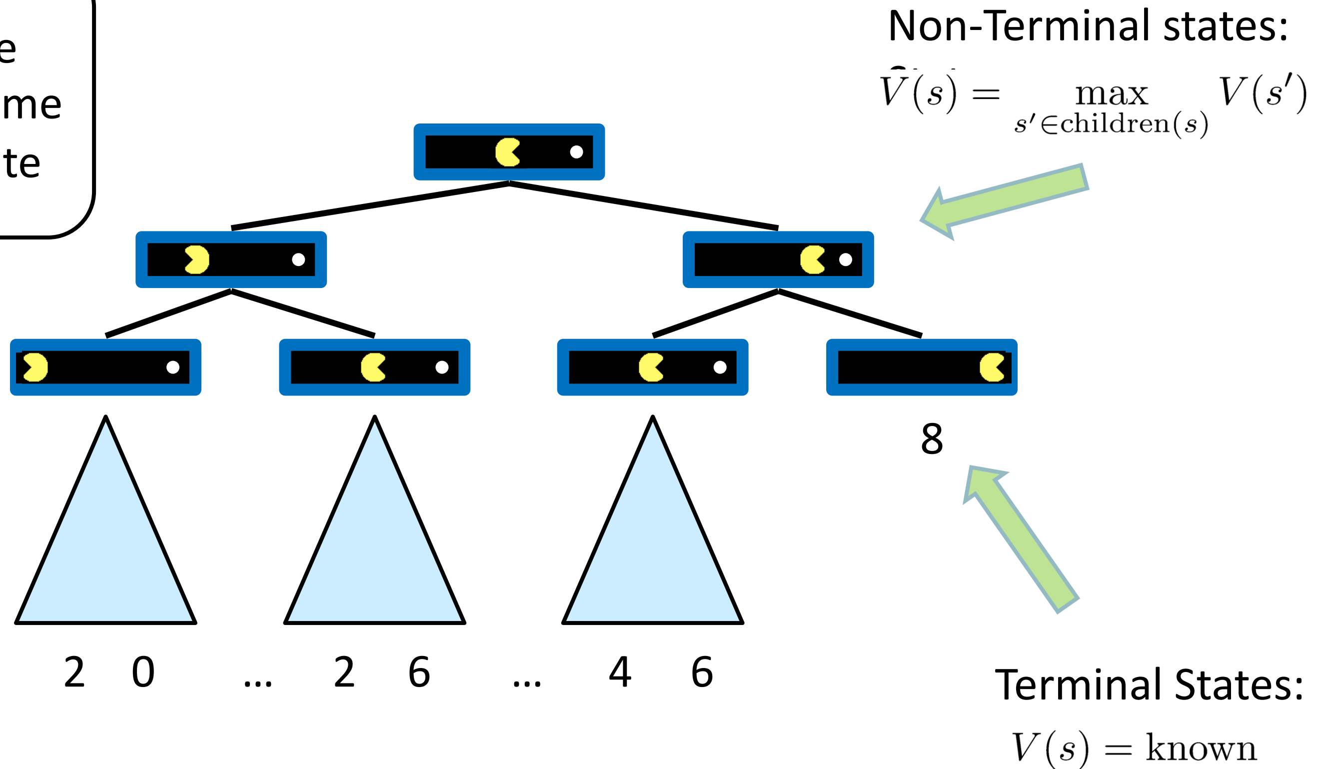


Single-agent trees

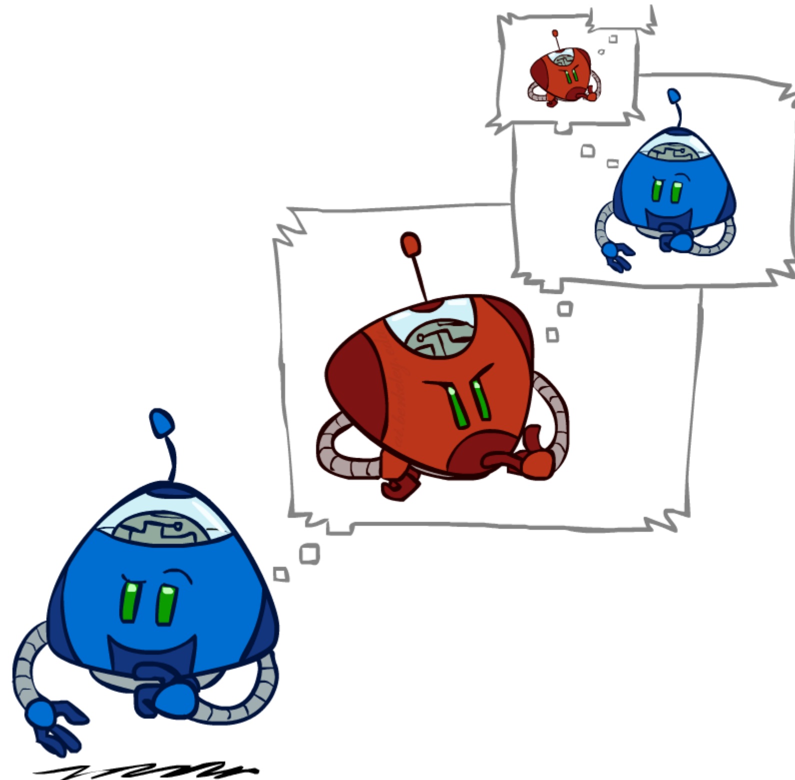


Value of a state

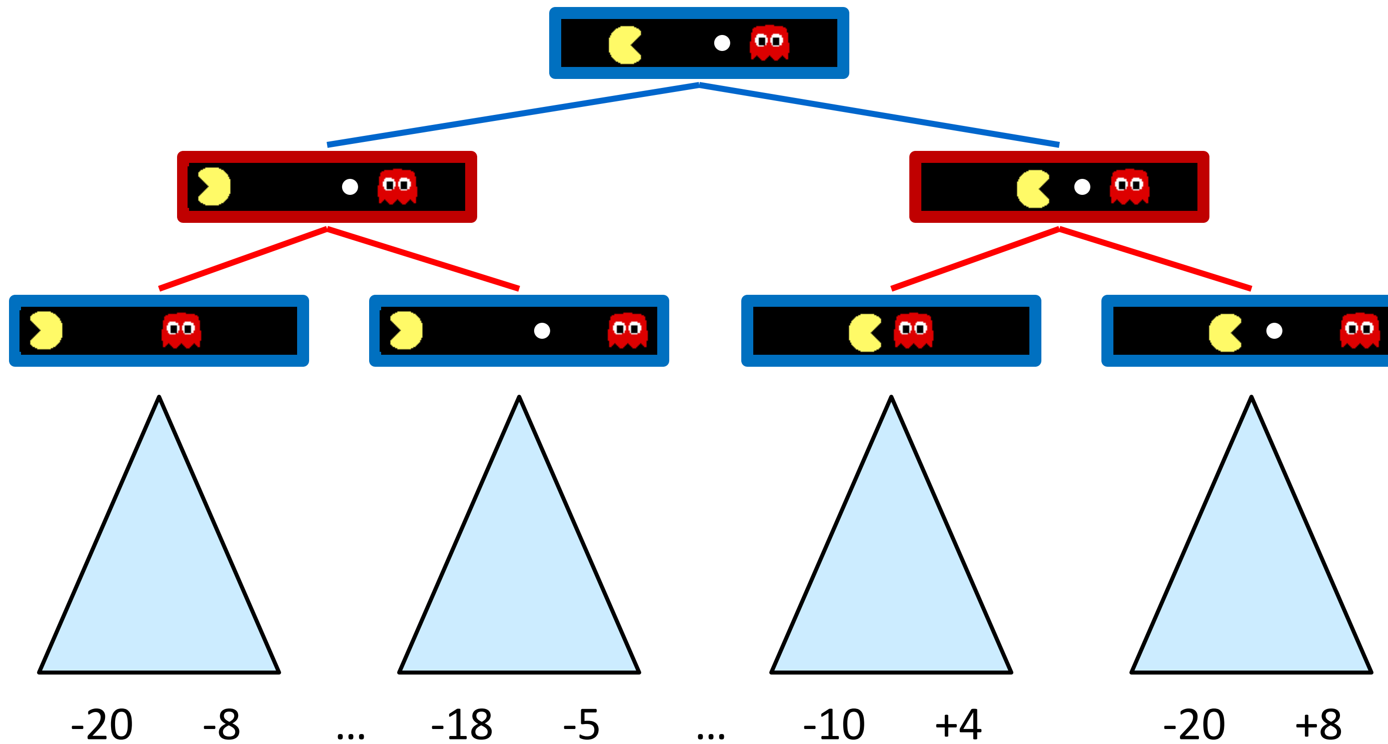
Value of a state: The best achievable outcome (utility) from that state



Adversarial search



Adversarial game trees



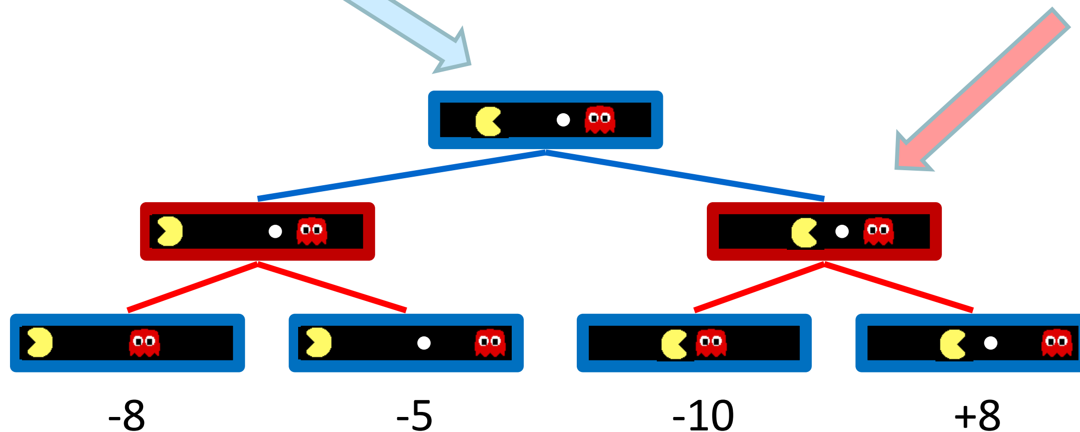
Minimax values

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:

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

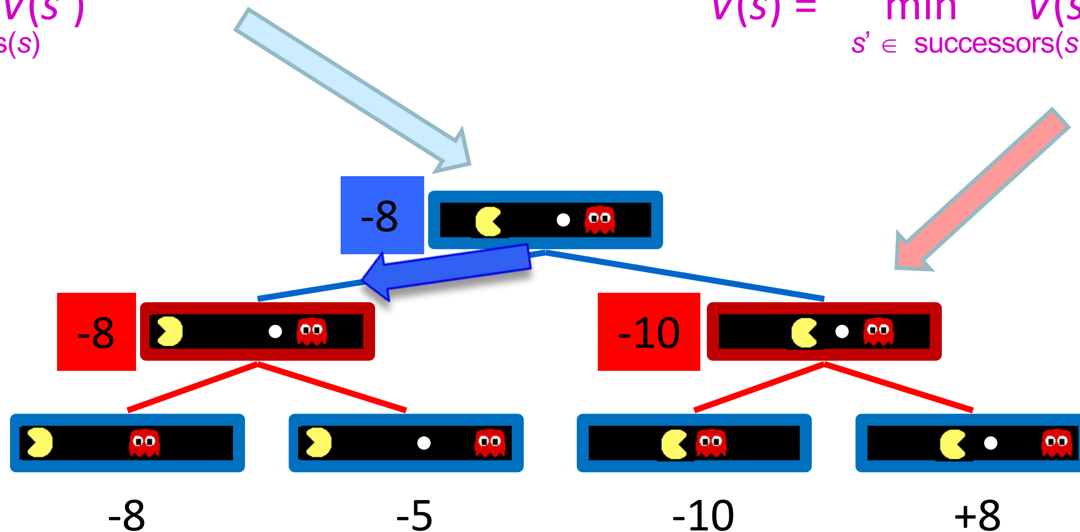
Minimax Values

MAX nodes: under Agent's control

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

MIN nodes: under Opponent's control

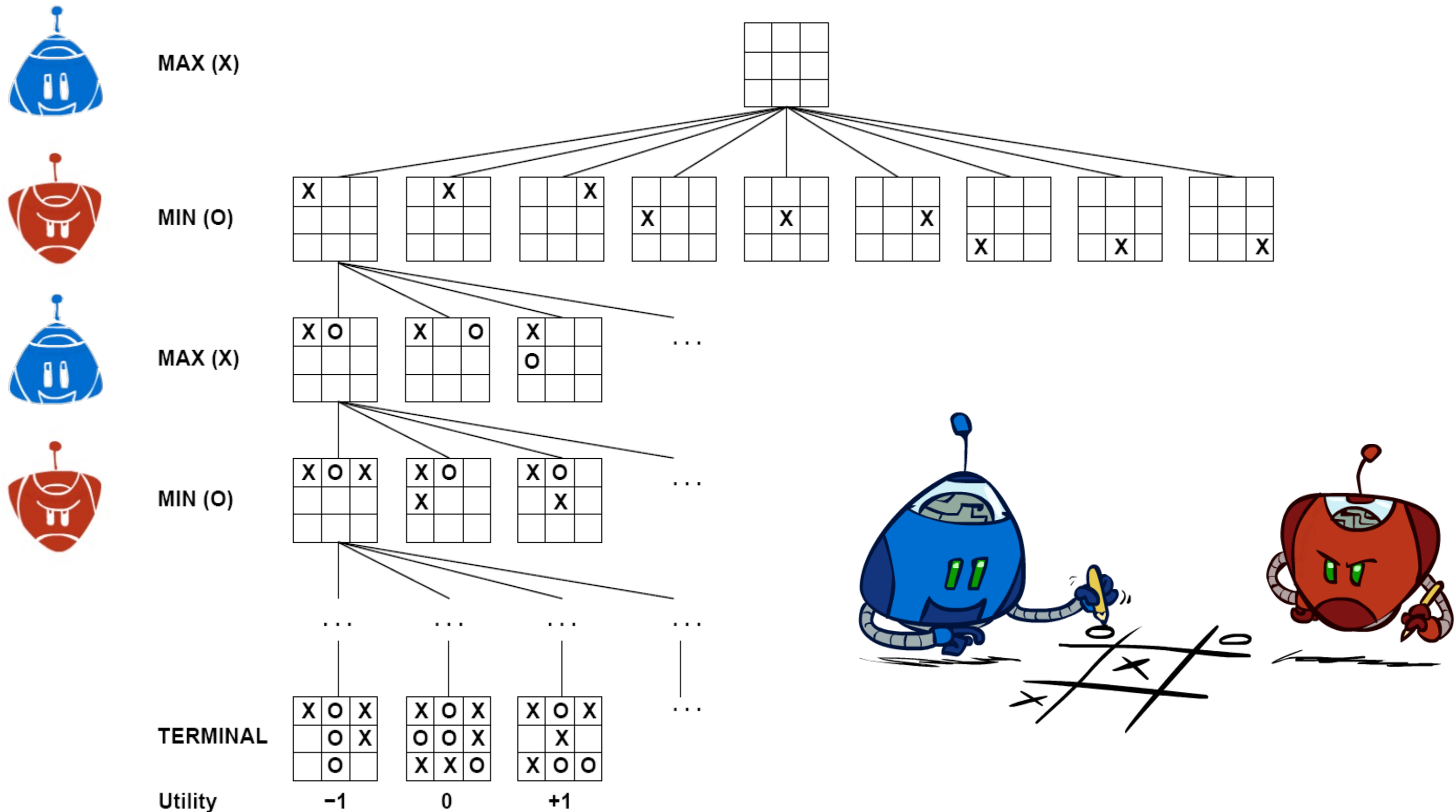
$$V(s) = \min_{s' \in \text{successors}(s)} V(s')$$



Terminal States:

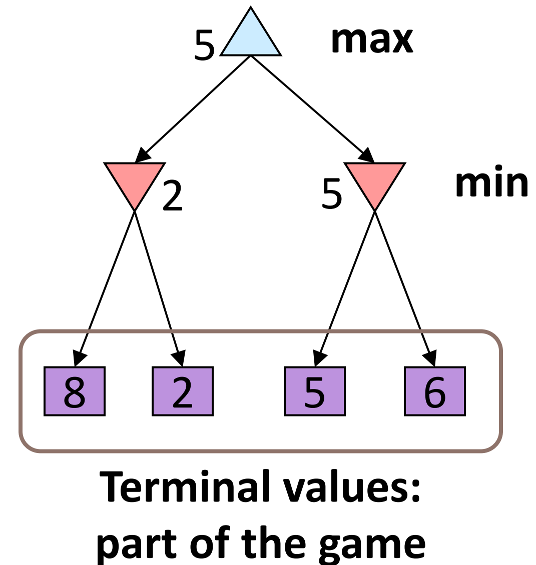
$V(s)$ = known

Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

- Opponent is assumed optimal
- 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
- Choose action leading to state with best ***minimax value***



Minimax Implementation

```
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)
```

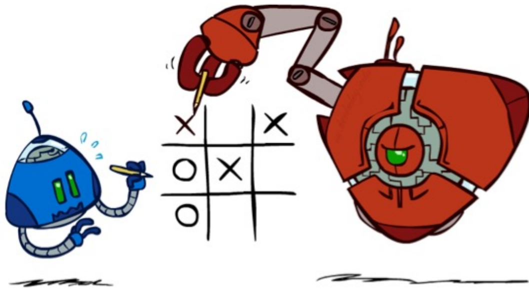
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

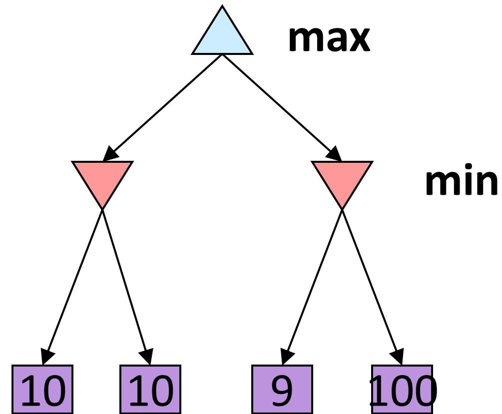
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

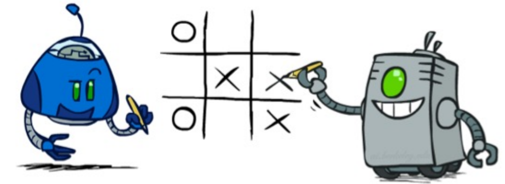
Properties of minimax

- Complete? Yes (when tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity: $O(b^m)$
- Space complexity: $O(bm)$ (depth-first exploration)
- For chess, $b \approx 35, m > 50$ for reasonable games
  - Finding exact solution is completely infeasible

Minimax Properties

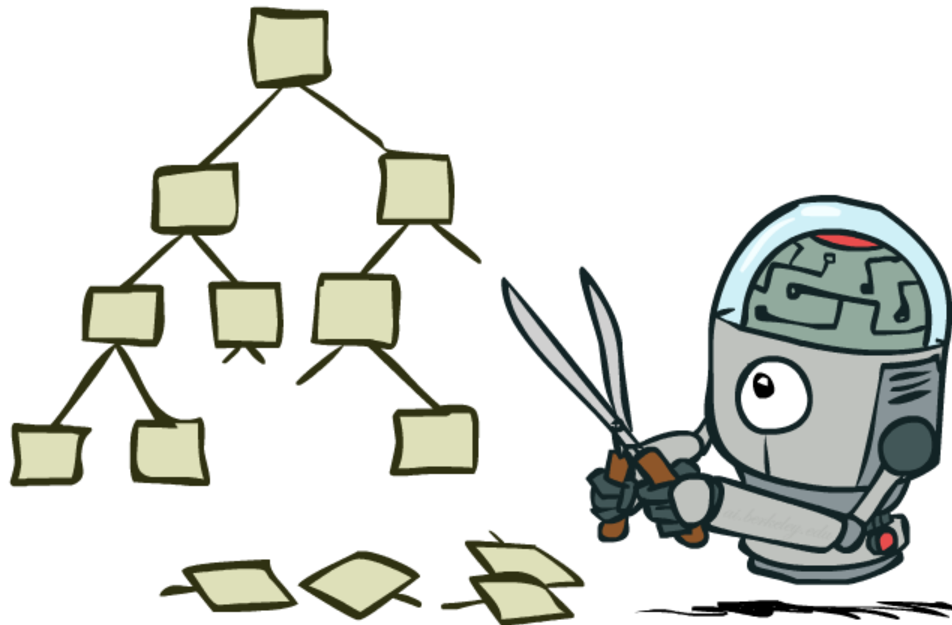






Optimal against a perfect player. Otherwise?

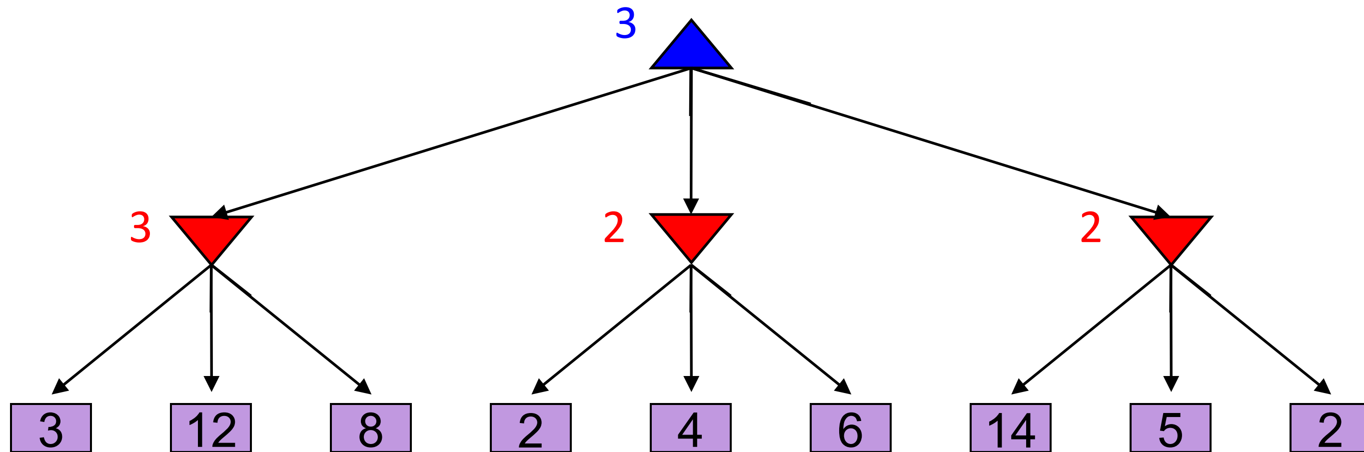
Game Tree Pruning



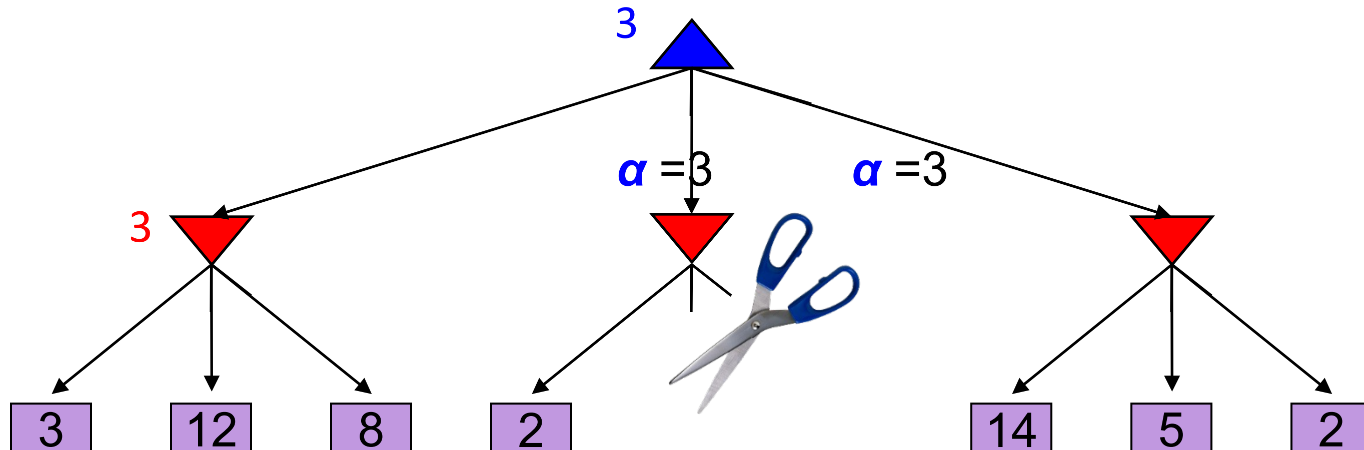
Pruning

- Correct minimax decision without looking at every node in the game tree
  - α-β pruning
    - Branch & bound algorithm
    - Prunes away branches that cannot influence the final decision

Minimax: Example

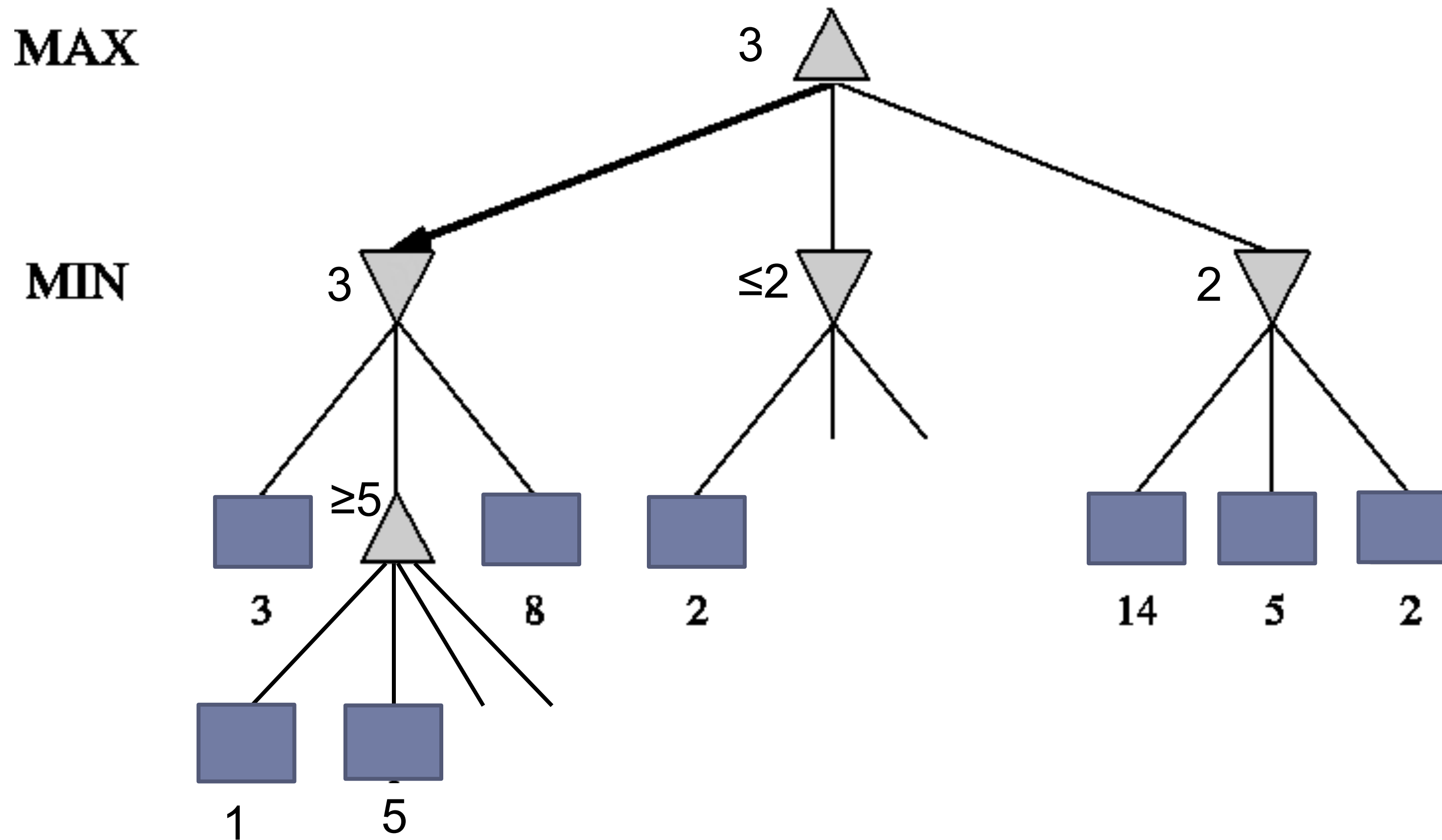


Alpha-Beta pruning: Example



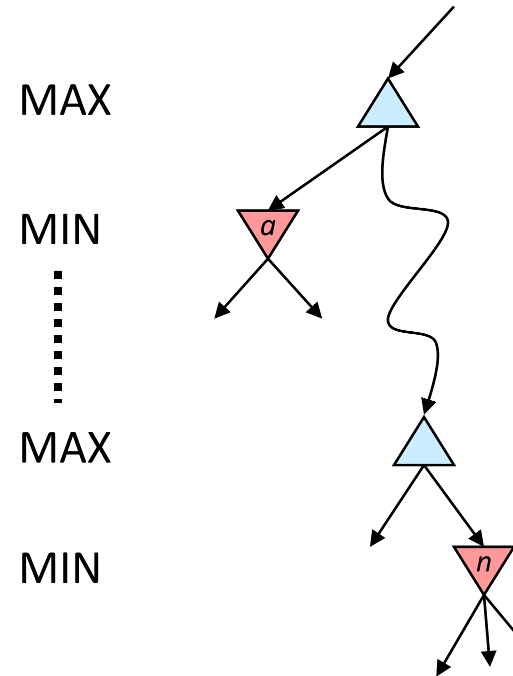
• ***The order of generation matters***: more pruning is possible if good moves come first

Alpha-Beta pruning: An other example



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



Why is it called α-β?

- α: Value of the best (<u>highest</u>) choice found so far at any choice point along the path for MAX
- β: Value of the best (<u>lowest</u>) choice found so far at any choice point along the path for MIN

- Updating α and β during the search process

- For a MAX node once the value of this node is known to be more than the current β ($v \geq \beta$), its remaining branches are pruned.

- For a MIN node once the value of this node is known to be less than the current α ($v \leq \alpha$), its remaining branches are pruned.

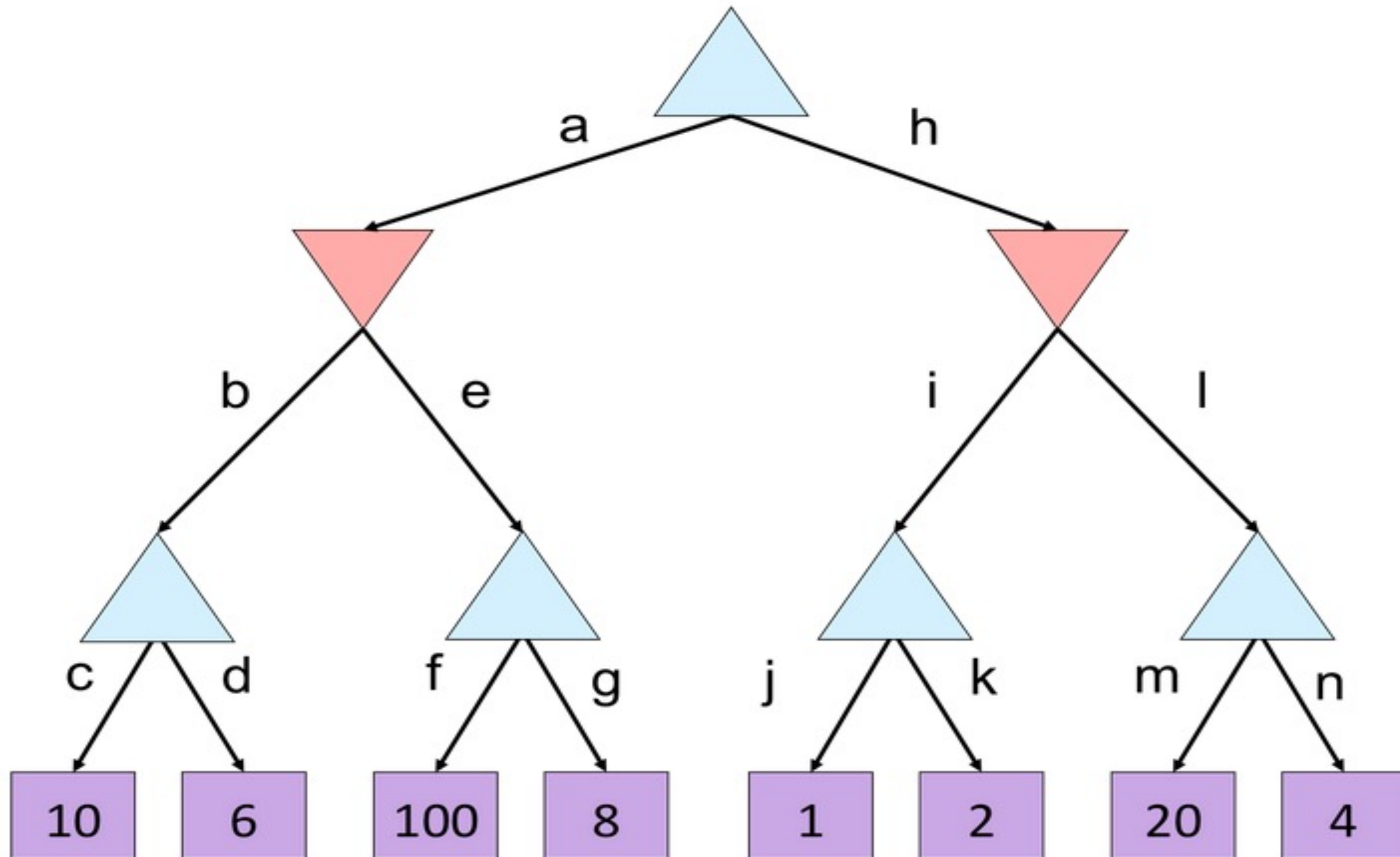
Alpha-Beta implementation

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

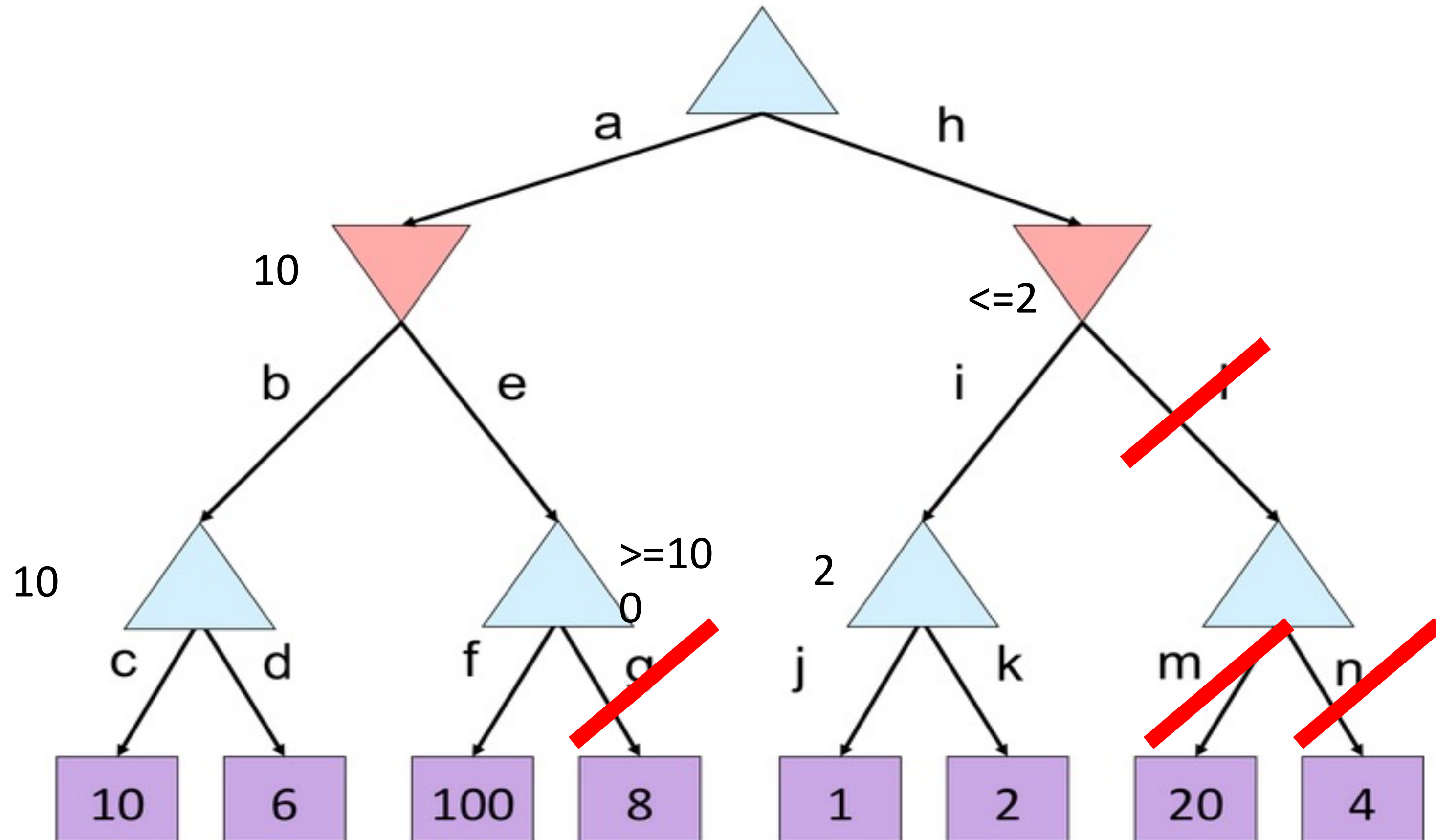
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor, α,β))
        if v ≥ β
            return v
        α = max(α, v)
    return v
```

```
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 quiz

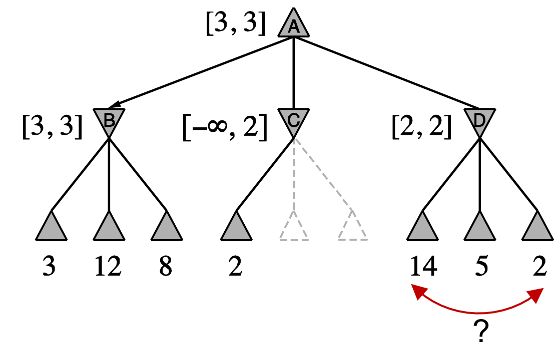


Alpha-Beta quiz



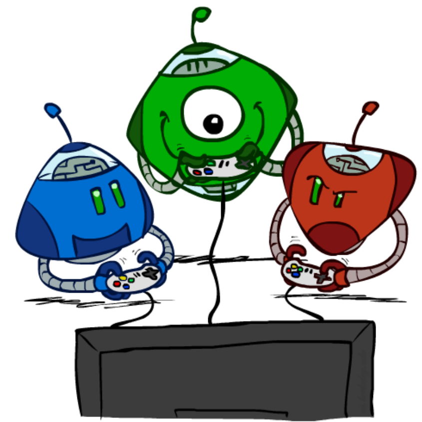
Alpha-Beta pruning properties

- 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…

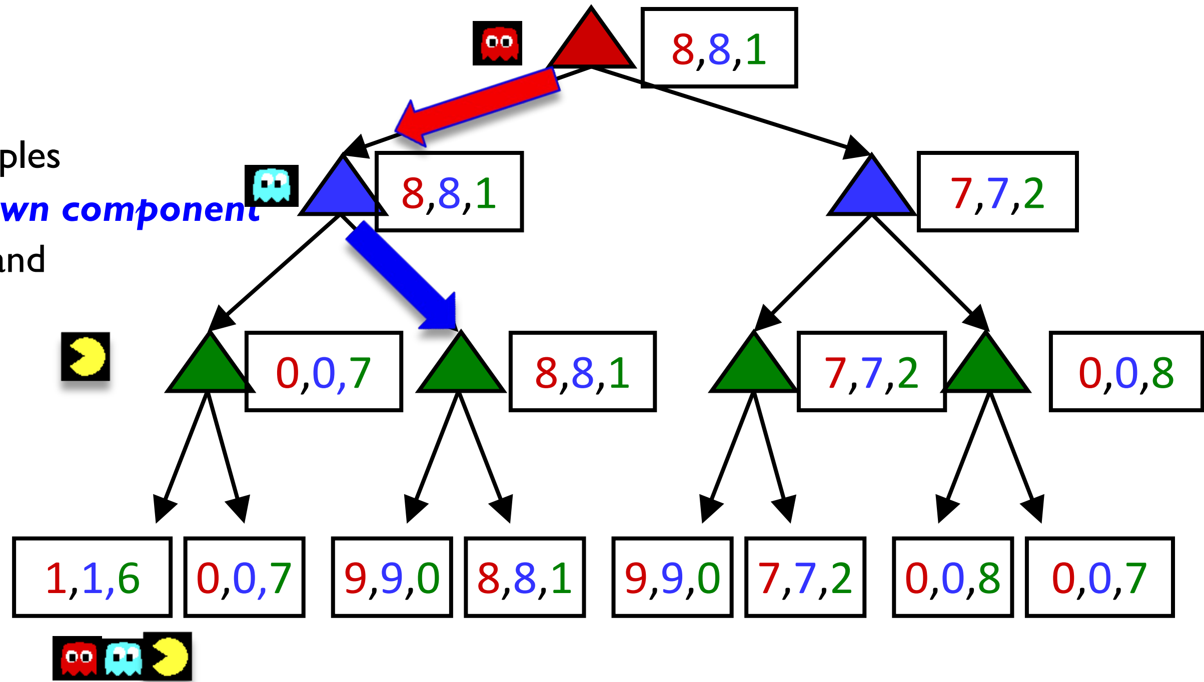


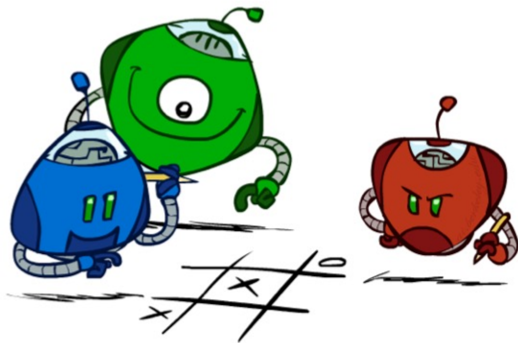
- This is a simple example of metareasoning (computing about what to compute)
- For chess: only 35^{50} instead of 35^{100}!! Yaaay!!!!!

Generalized minimax



- 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…





Video of Demo Smart Ghosts (Coordination)

Video of Demo Smart Ghosts (Coordination) – Zoomed In

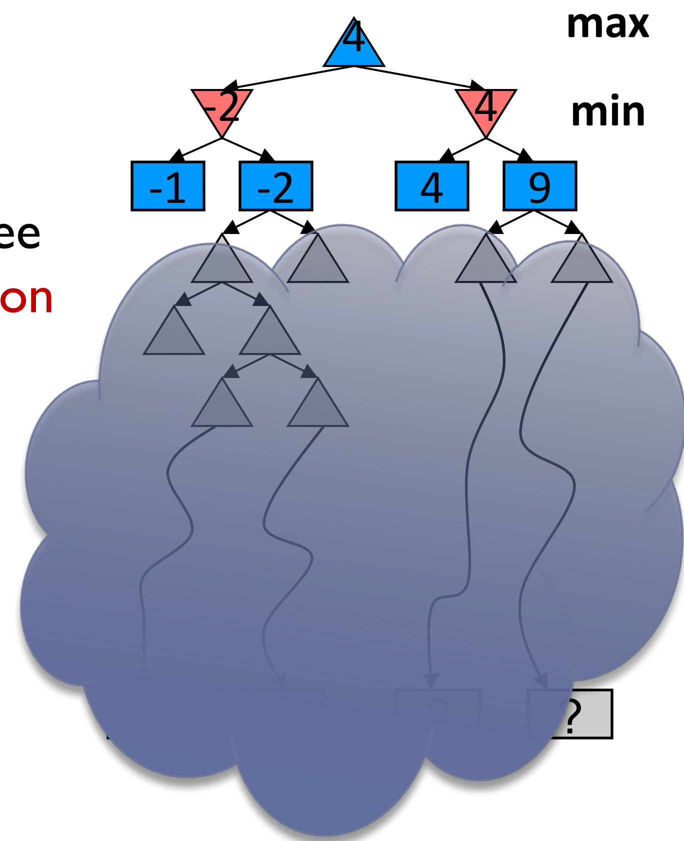
Summary

- Games are decision problems with ≥ 2 agents
  - Huge variety of issues and phenomena depending on details of interactions and payoffs
- For zero-sum games, optimal decisions defined by minimax
  - Implementable as a depth-first traversal of the game tree
  - Time complexity $O(b^m)$, space complexity $O(bm)$
- Alpha-beta pruning
  - Preserves optimal choice at the root
  - Alpha/beta values keep track of best obtainable values from any max/min nodes on path from root to current node
  - Time complexity drops to $O(b^{m/2})$ with ideal node ordering
- Exact solution is impossible even for "small" games like chess

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
- Guarantee of optimal play is gone
- More plies makes a BIG difference



Computational time limit: Solution

- Cut off the search and apply a heuristic evaluation function
  - cutoff test: turns non-terminal nodes into terminal leaves
    - Cut off test instead of terminal test (e.g., depth limit)
  - evaluation function: estimated desirability of a state
    - Heuristic function evaluation instead of utility function

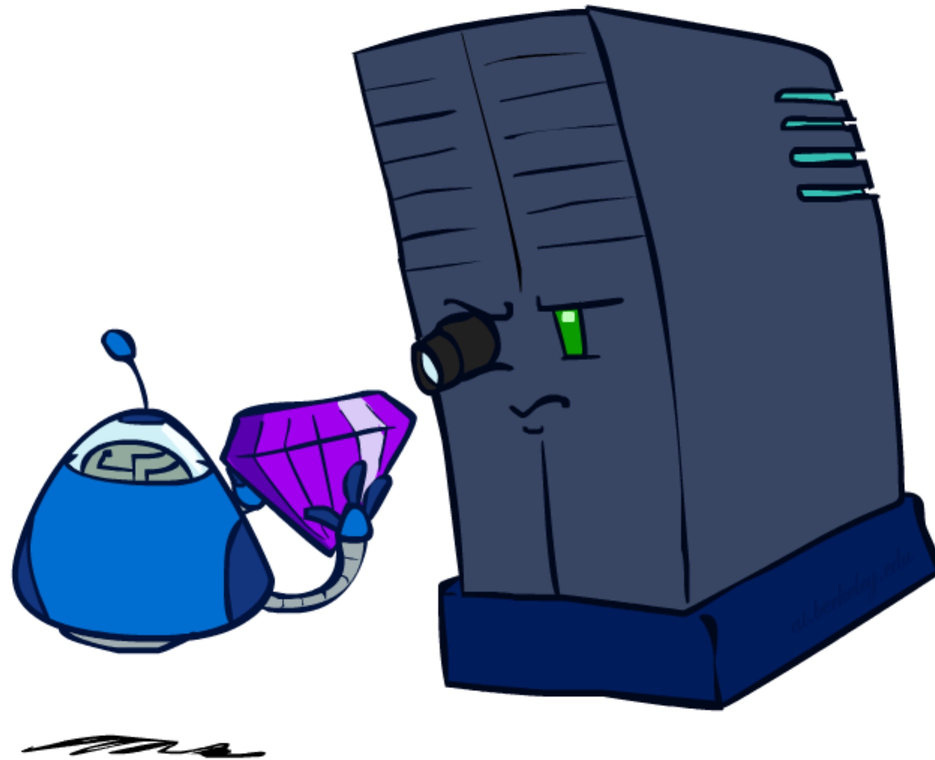
- This approach does not guarantee optimality.

Heuristic minimax

$$H_{MINIMAX(s,d)} =$$

$$\begin{cases} EVAL(s, MAX) & if\ CUTOFF_TEST(s,d) \\ max_{a \in ACTIONS(s)} H_MINIMAX(RESULT(s,a), d+1) & PLAYER(s) = \text{MAX} \\ min_{a \in ACTIONS(s)} H_MINIMAX(RESULT(s,a), d+1) & PLAYER(s) = \text{MIN} \end{cases}$$

Evaluation functions



Evaluation functions

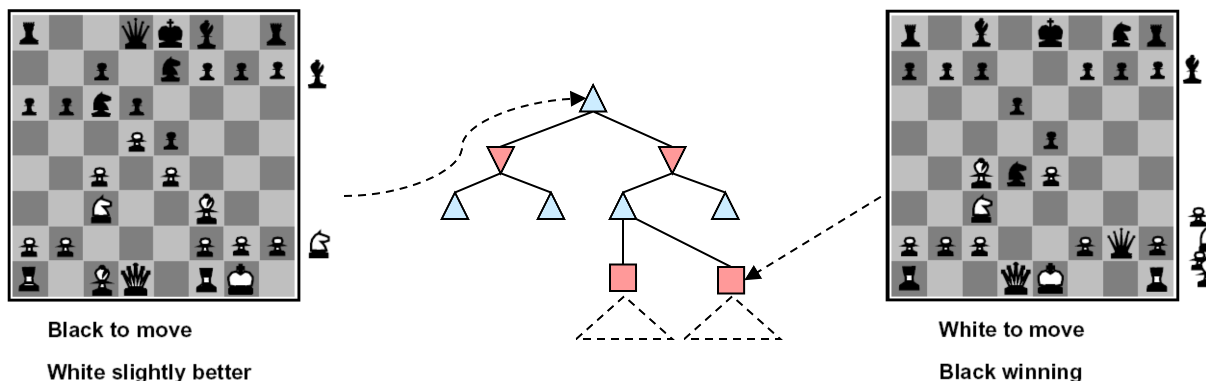
- For terminal states, it should order them in the same way as the true utility function.
- For non-terminal states, it should be strongly correlated with the actual chances of winning.
- It must not need high computational cost.

Evaluation functions based on features

- Example: features for evaluation of the chess states
  - Number of each kind of piece: number of white pawns, black pawns, white queens, black queens, etc
  - King safety
  - Good pawn structure
  - …

Evaluation functions

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

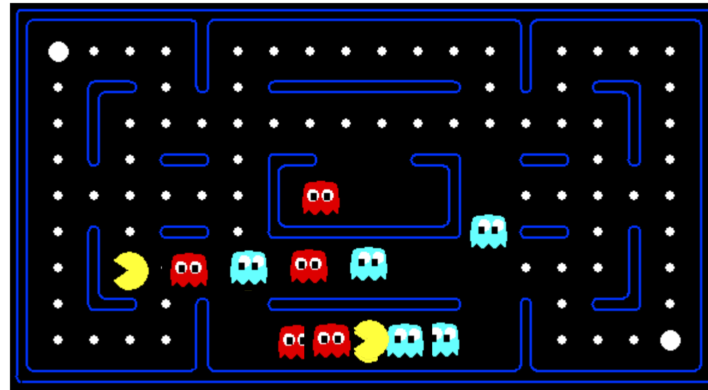


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

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

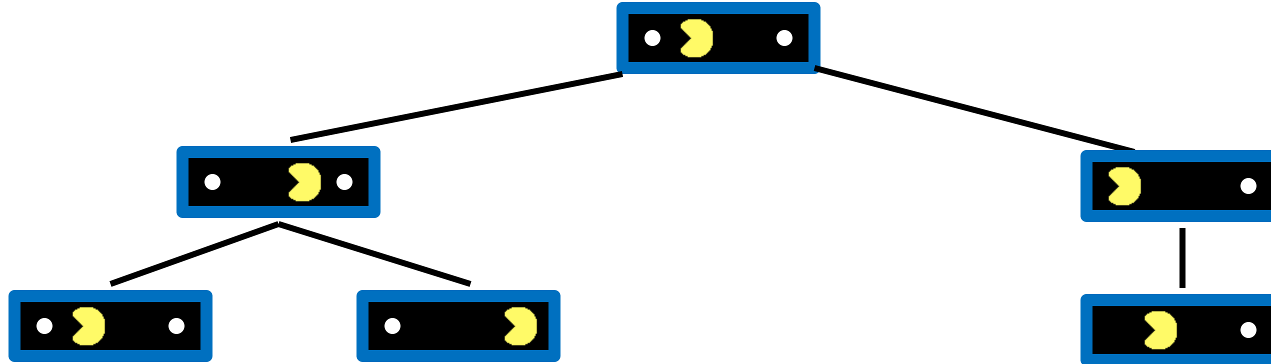
Evaluation for Pacman



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

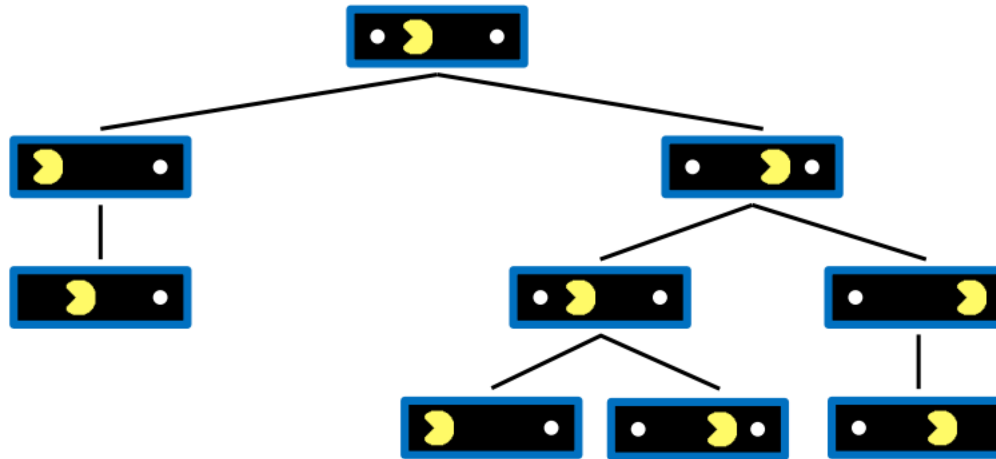
Video of Demo Thrashing (d=3)

Why Pacman Starves

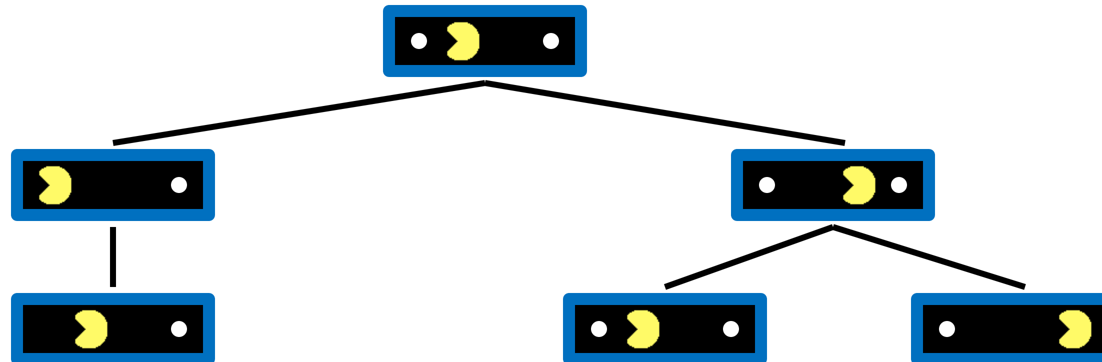


- A danger of replanning agents!
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

Why Pacman Starves



Fixing the evaluation function



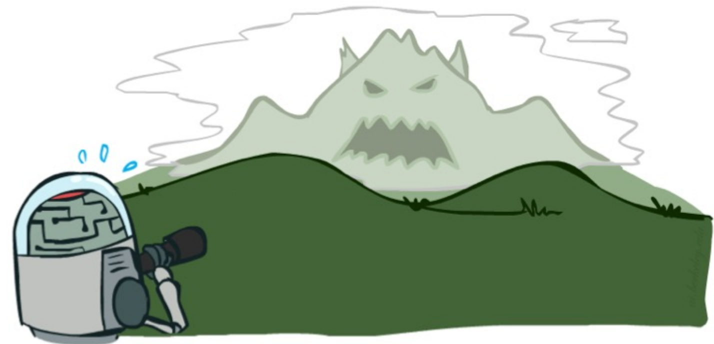
Evaluation function:

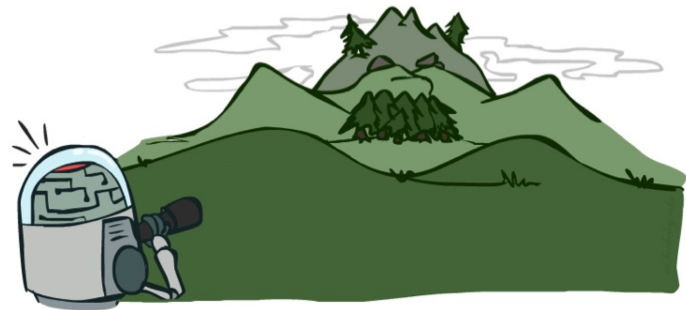
# of eaten dots + 1/(distance to the closest dot)

Video of Demo Thrashing -- Fixed (d=3)

Depth Matters

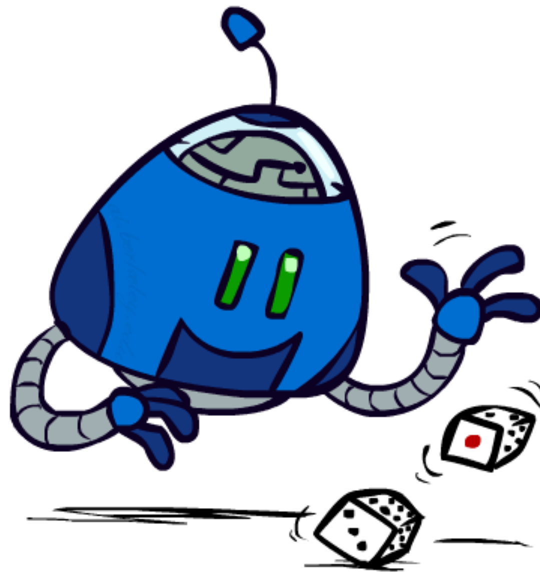
- Evaluation functions are always imperfect
- Deeper search => better play (usually)
- Or, deeper search gives same quality of play with a less accurate evaluation function
- An important example of the tradeoff between complexity of features and complexity of computation



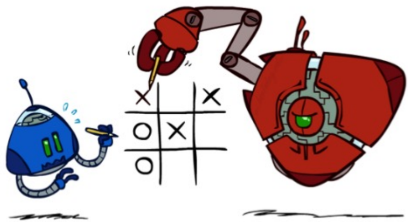


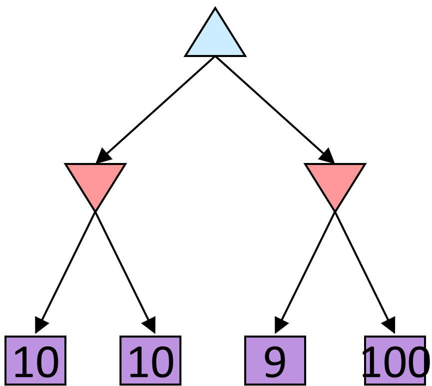
[Demo: depth limited (L6D4, L6D5)]

Uncertain Outcomes

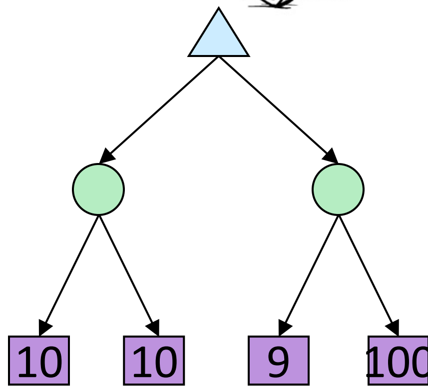


Chance outcomes in trees

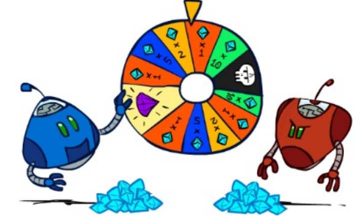


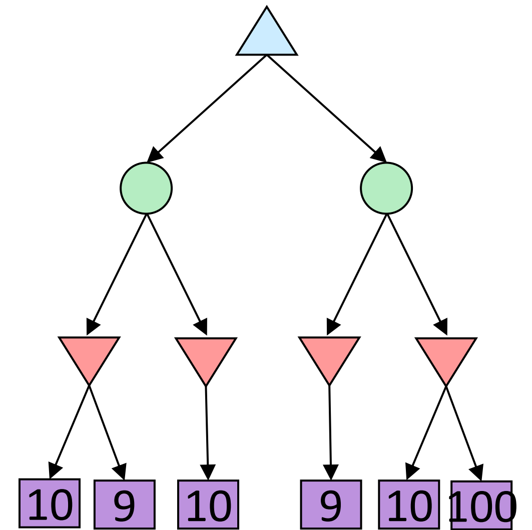


Tictactoe, chess
Minimax



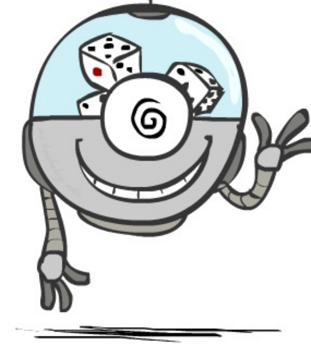
Tetris, investing
Expectimax



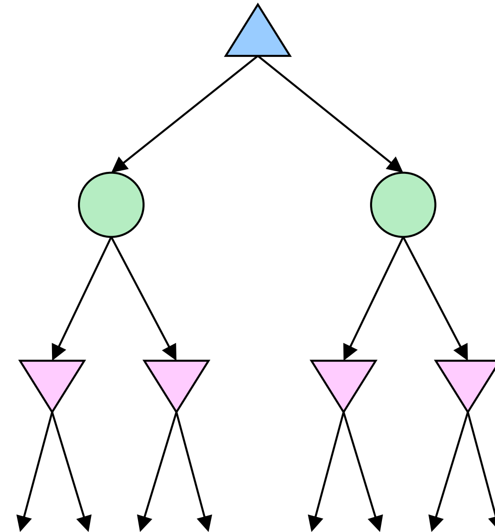


Backgammon, Monopoly
Expectiminimax

Mixed Layer Types



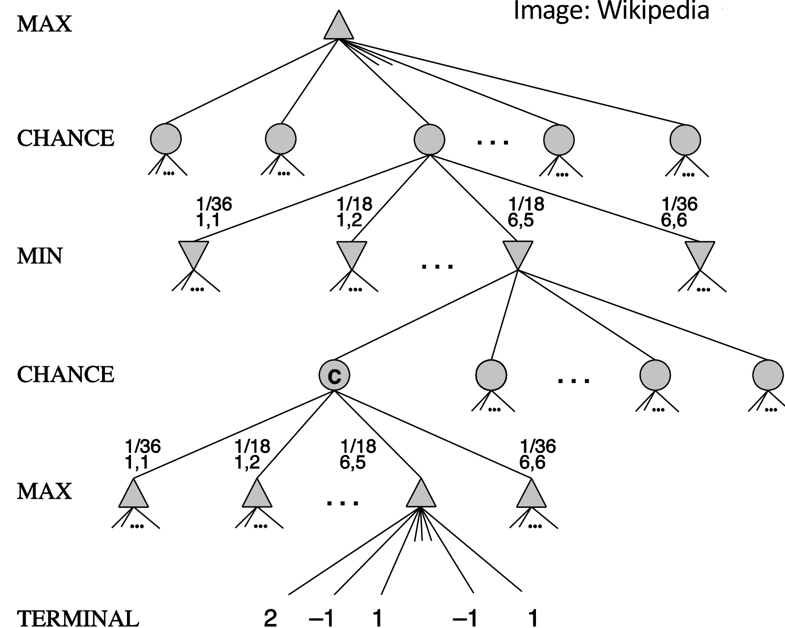
- E.g. Backgammon
- Expectiminimax
  - Environment is an extra "random agent" player that moves after each min/max agent
  - Each node computes the appropriate combination of its children



Example: Backgammon

- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon ≈ 20 legal moves
  - 4 plies = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - Usefulness of search is diminished
  - Pruning is trickier…
- Historic AI: TDGammon (1997) uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play

Image: Wikipedia

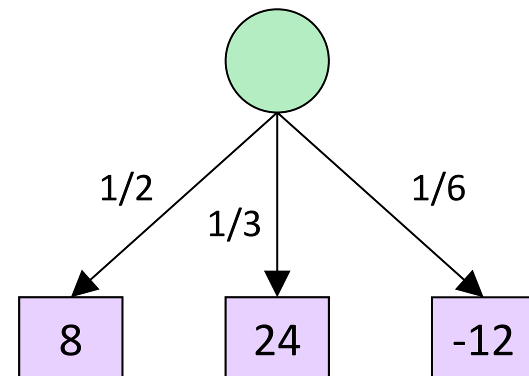


Expectiminimax

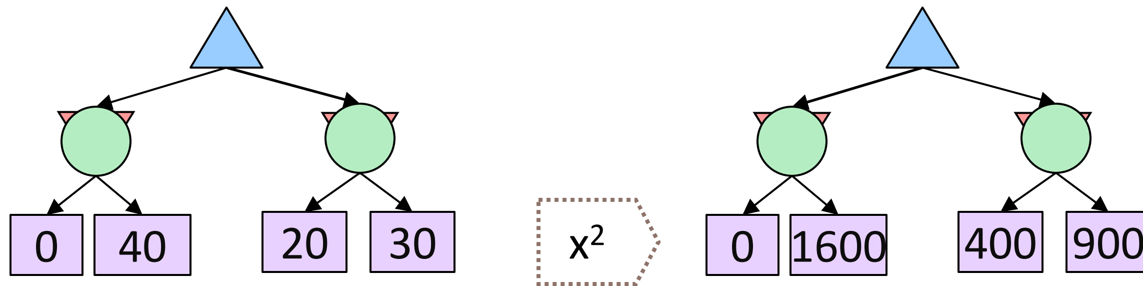
```
function decision(s) returns an action
   return the action a in Actions(s) with the highest
    value(Result(s,a))
```

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

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



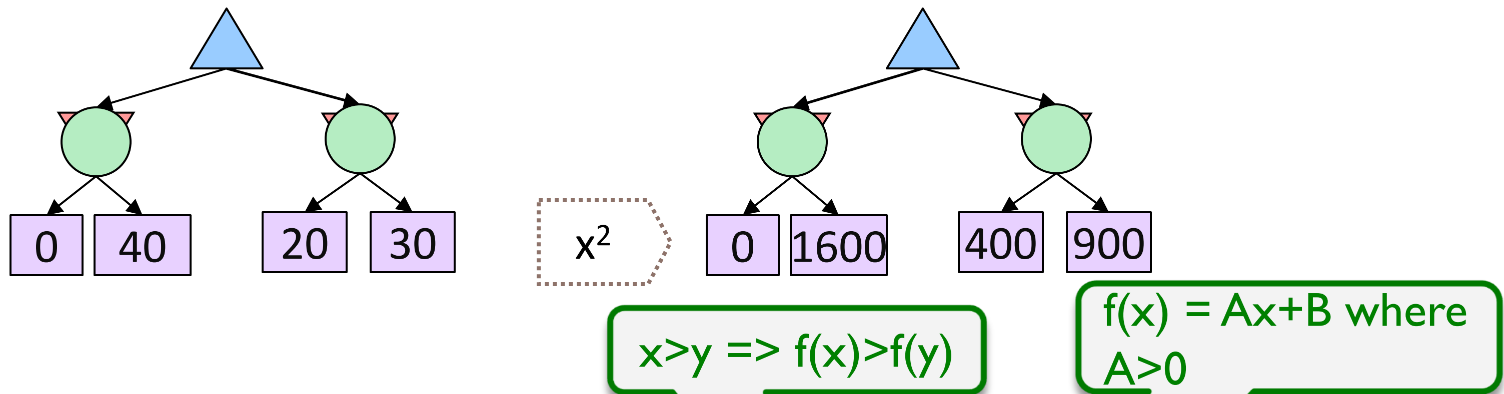
What Values to Use?



- For worst-case minimax reasoning, evaluation function scale doesn't matter
  - We just want better states to have higher evaluations (get the ordering right)
  - Minimax decisions are ***invariant with respect to monotonic transformations on values***

- Expectiminimax decisions are ***invariant with respect to positive affine transformations***
- Expectiminimax evaluation functions have to be aligned with actual win probabilities!

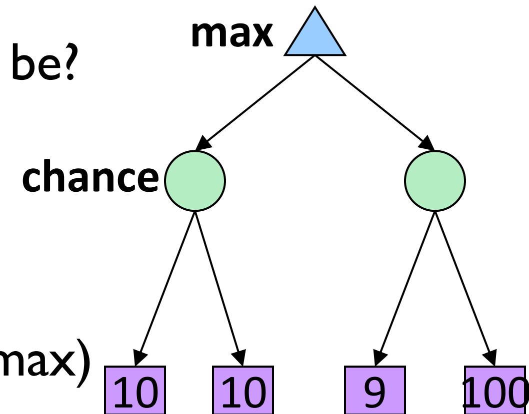
What Values to Use?



- For worst-case minimax reasoning, evaluation function scale doesn't matter
  - We just want better states to have higher evaluations (get the ordering right)
  - Minimax decisions are ***invariant with respect to monotonic transformations on values***
- Expectiminimax decisions are ***invariant with respect to positive affine transformations***
- Expectiminimax evaluation functions have to be aligned with actual win probabilities!

Expectimax Search (One Player Game)

- 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

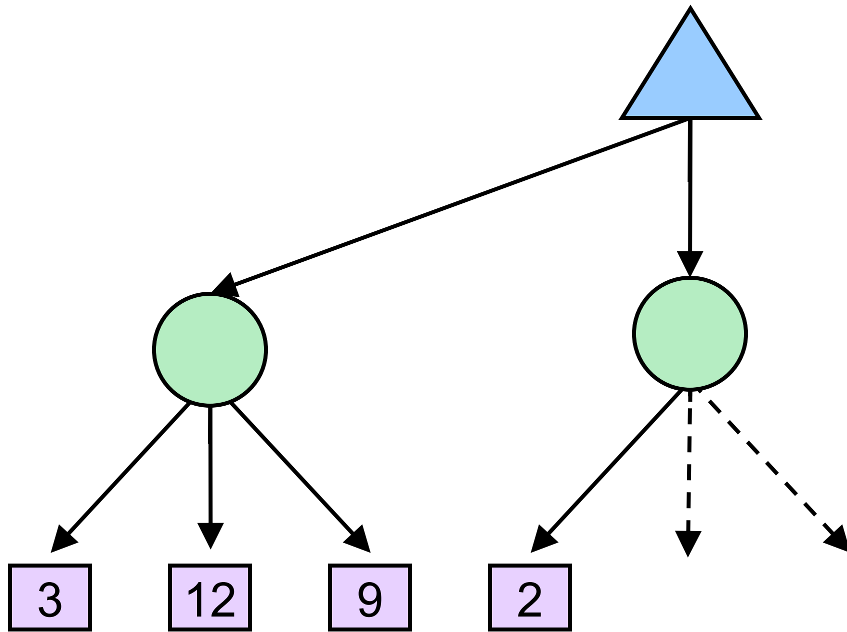


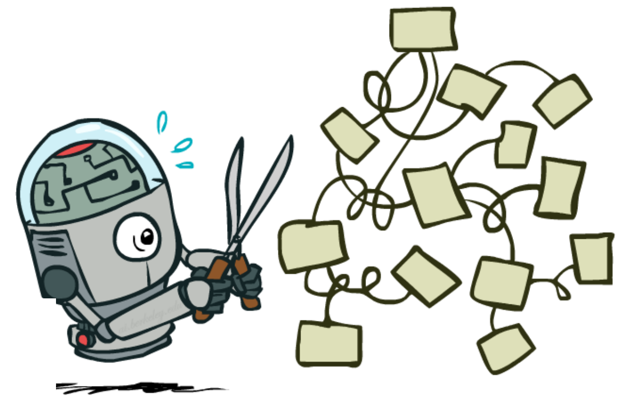
- Values should now reflect average-case (expectimax) outcomes, not worst-case outcomes

- 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

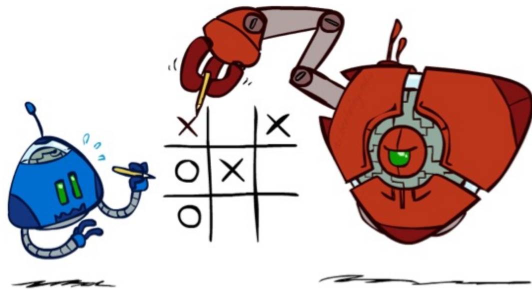
- Soon, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes

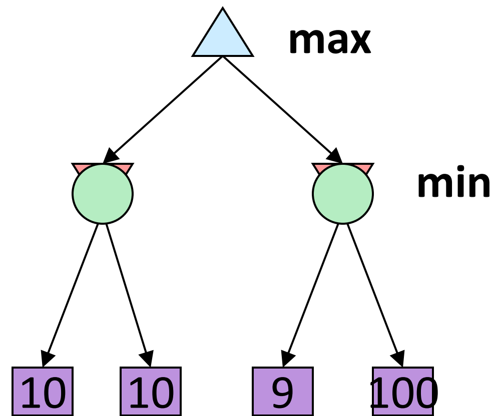
Expectimax Pruning?

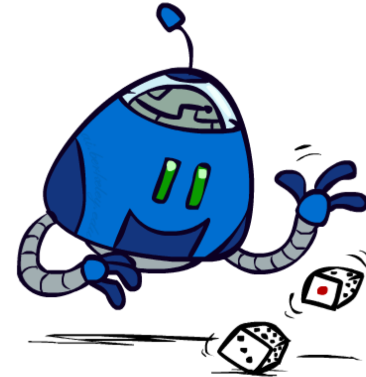




Worst-Case vs. Average Case (One player Game)



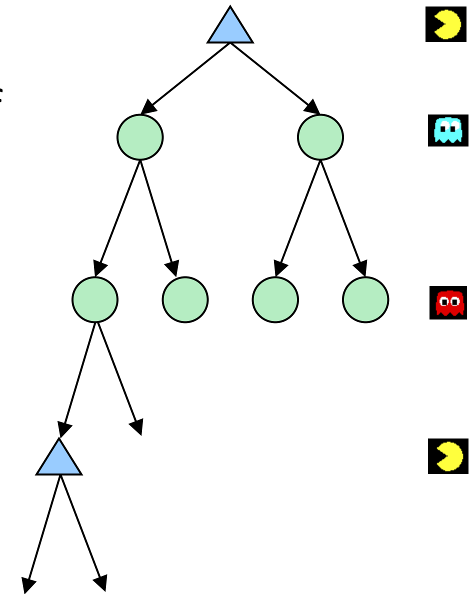




Idea: Uncertain outcomes controlled by chance, not an adversary!

What Probabilities to Use?

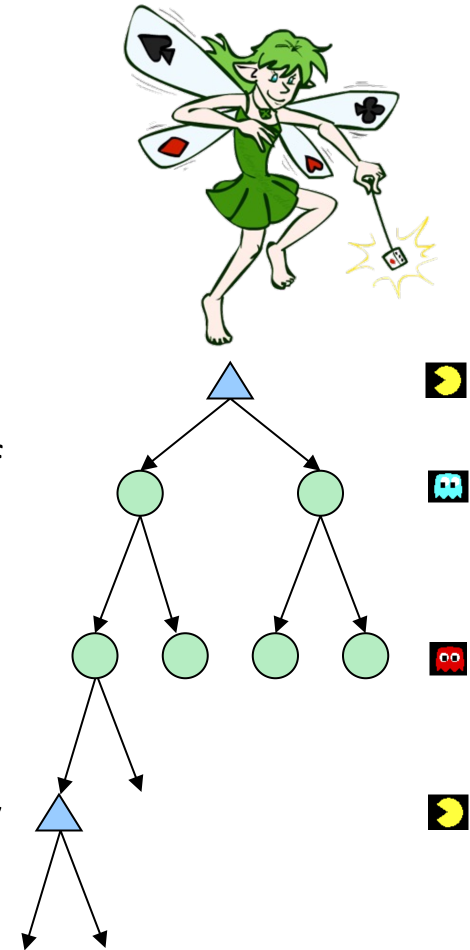
- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - could be a simple uniform distribution (roll a die)
  - could be sophisticated and require a great deal of computation
  - a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!



Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!

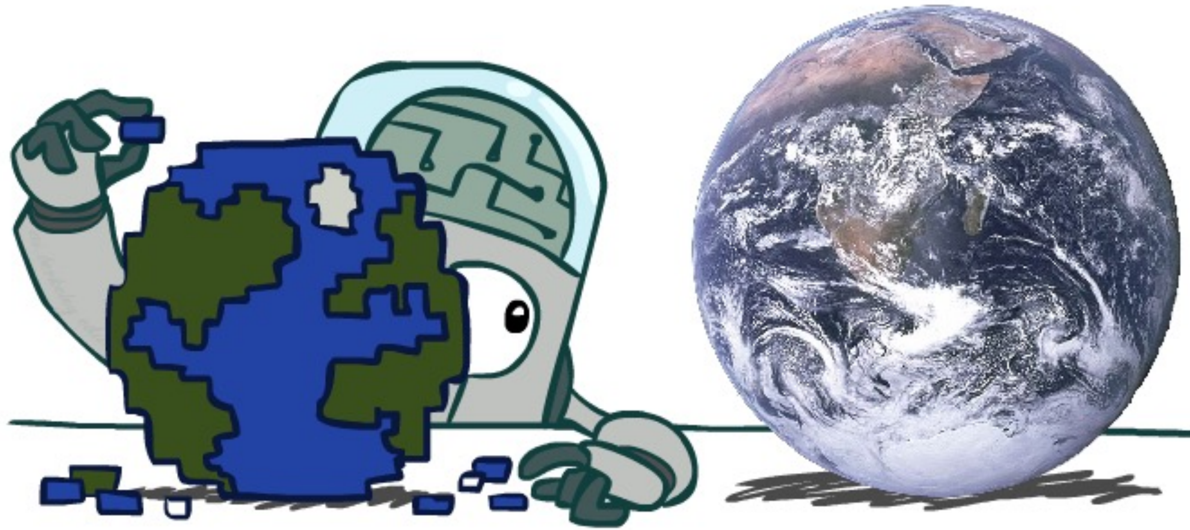
What Probabilities to Use?

- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - could be a simple uniform distribution (roll a die)
  - could be sophisticated and require a great deal of computation
  - a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!
- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!

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

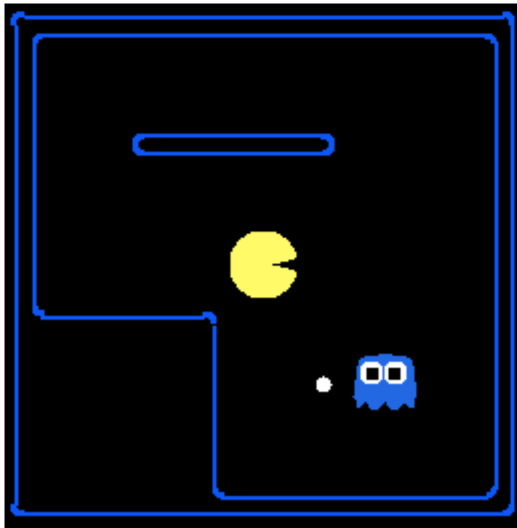
Video of Demo World Assumptions
Random Ghost – Expectimax Pacman

Video of Demo World Assumptions Random Ghost – Minimax Pacman

Video of Demo World Assumptions
Adversarial Ghost – Expectimax Pacman

Video of Demo World Assumptions
Adversarial Ghost – Minimax Pacman

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

Maximum Expected Utility

- Why should we average utilities? Why not minimax?

- Principle of maximum expected utility:
  - A rational agent should chose the action that maximizes its expected utility, given its knowledge

- Questions:
  - Where do utilities come from?
  - How do we know such utilities even exist?
  - How do we know that averaging even makes sense?
  - What if our behavior (preferences) can't be described by utilities?