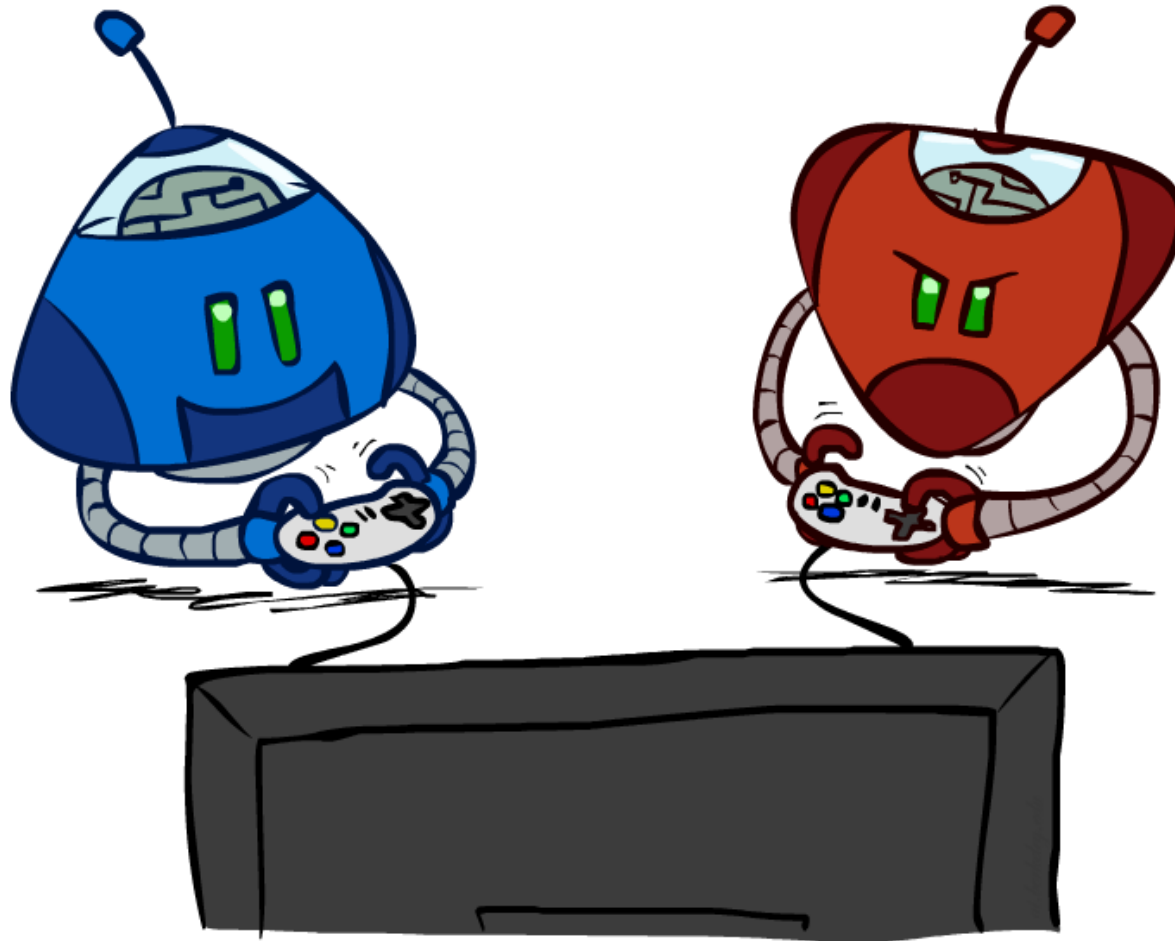


CS 188: Artificial Intelligence

Search with Other Agents I

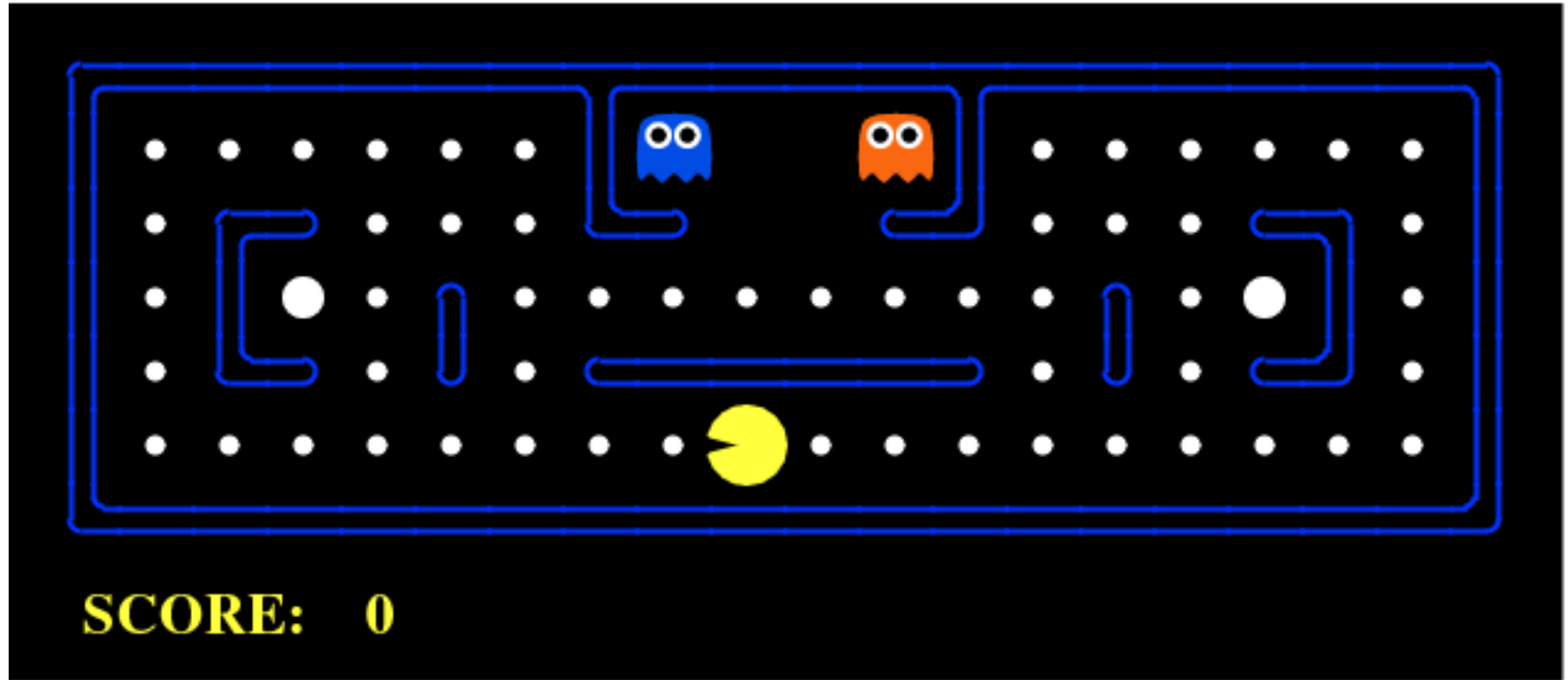


[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

Why Multiple Agents?

- AI agents that work with humans (or other AI agents)
 - Robot helpers, AI tutors, self-driving cars
- Multiple agents compete against each other to improve
 - Examples: AlphaGo
- Play multi-agent Pacman! (and other games)

Multi-Agent Pacman

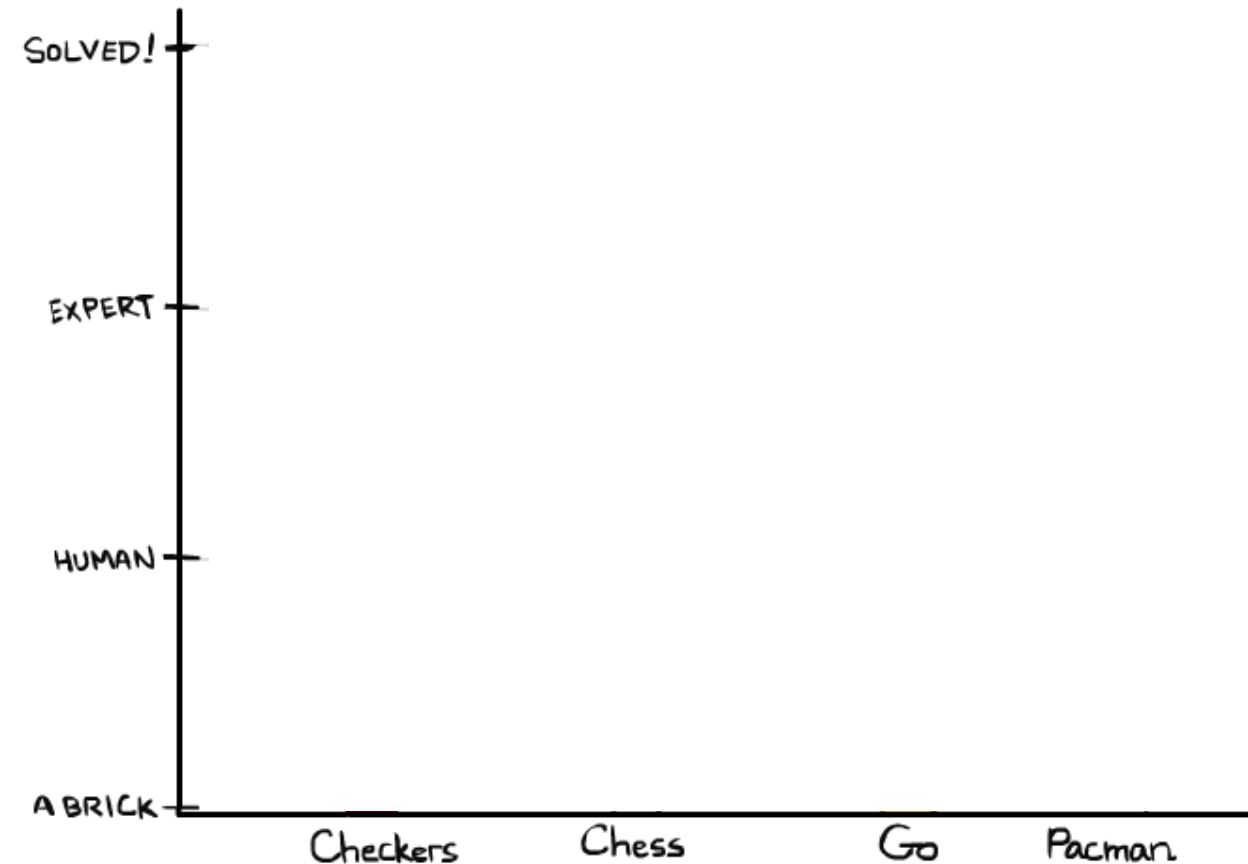


We'll focus on games in this class, but multi-agent ideas come up in many areas of AI

[Demo: mystery pacman (L6D1)]

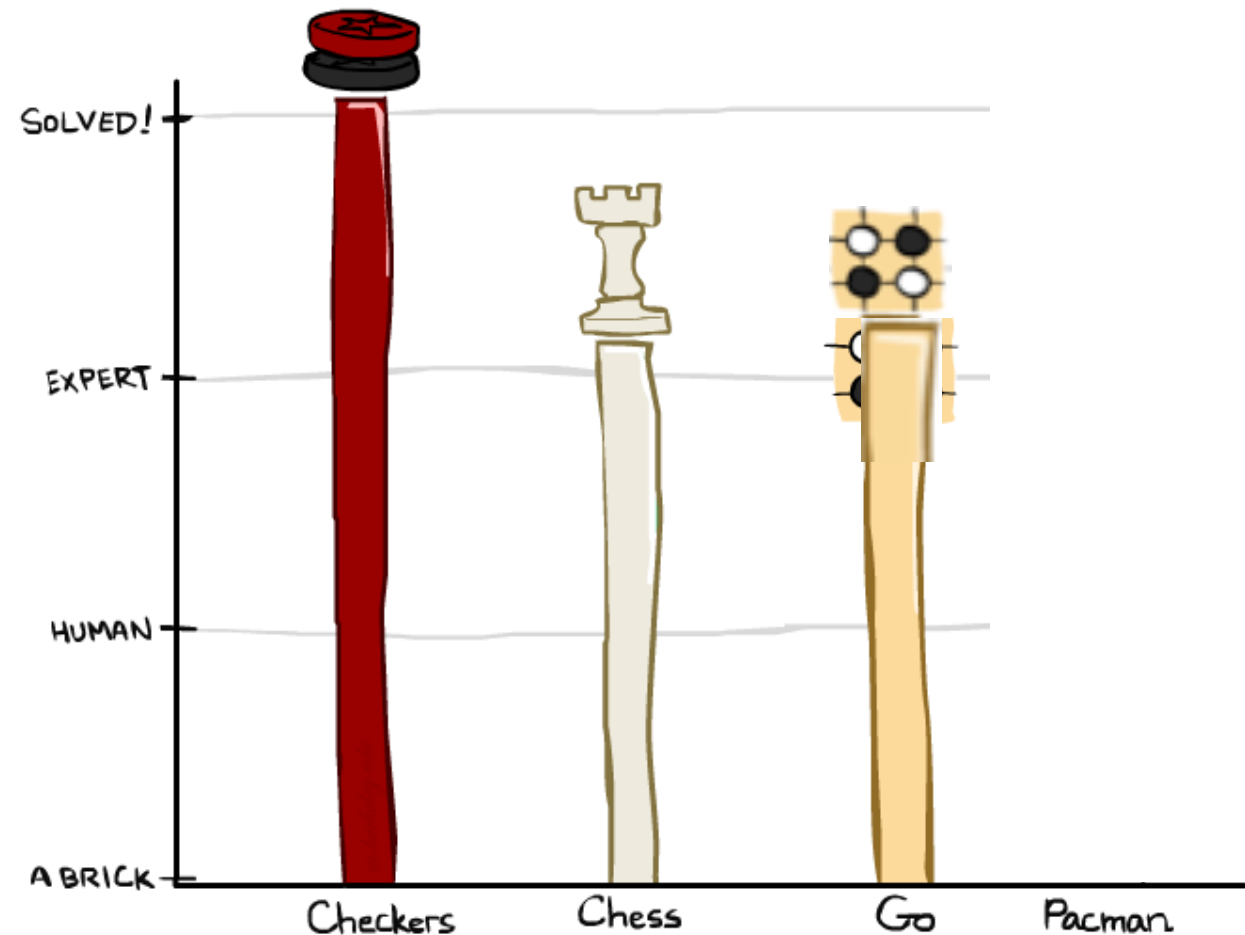
Game Playing Progress

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



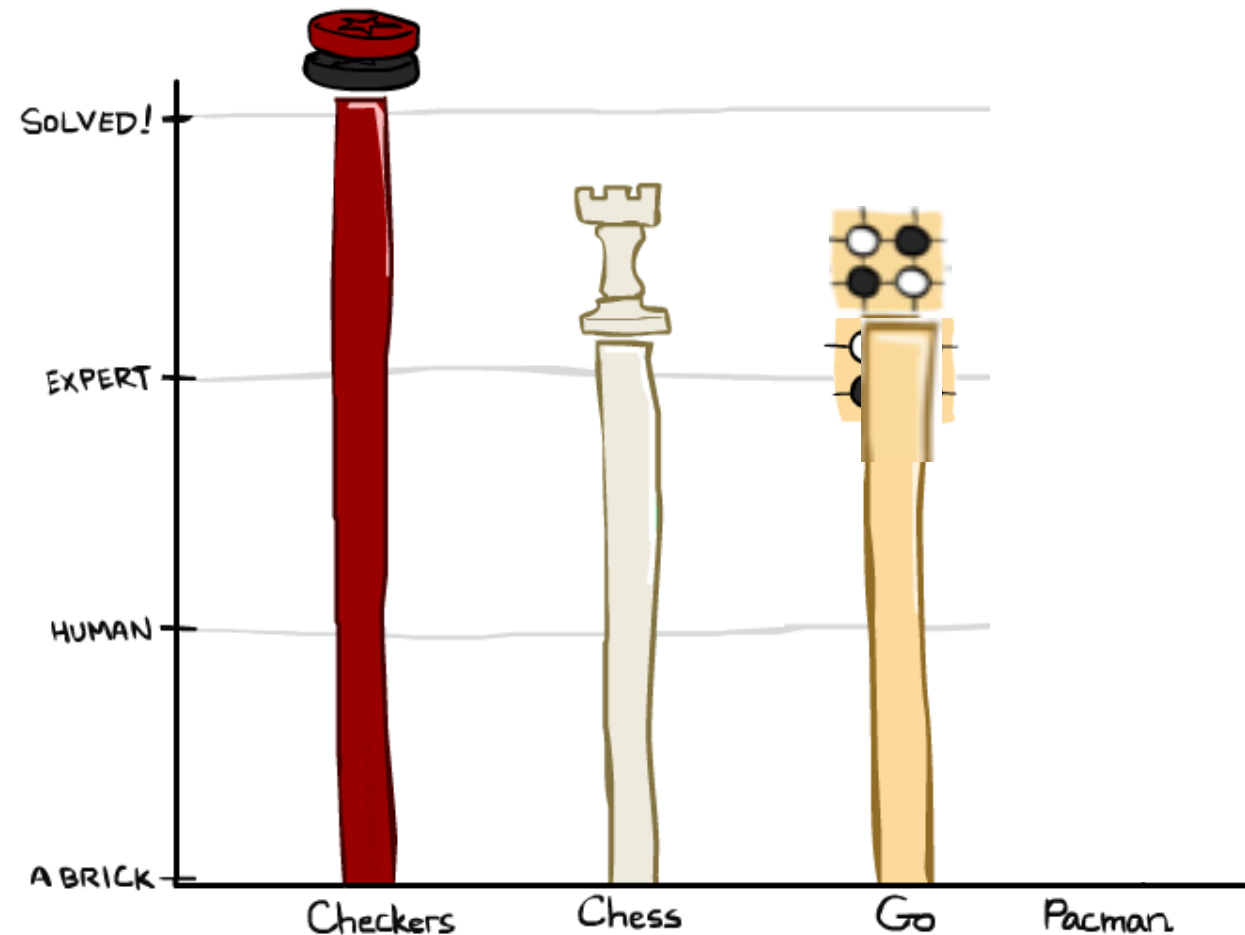
Game Playing Progress

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



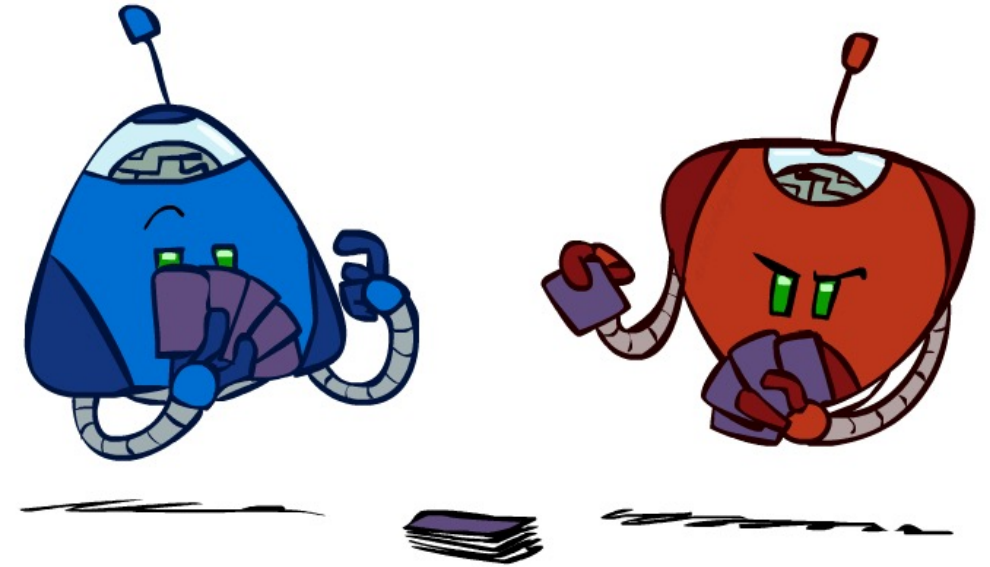
Game Playing Progress

- **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:** AlphaGo defeats human in 2016. Uses Monte Carlo Tree Search and learned evaluation function.
- **Pacman**
- **Why play games?** Helps track progress in AI



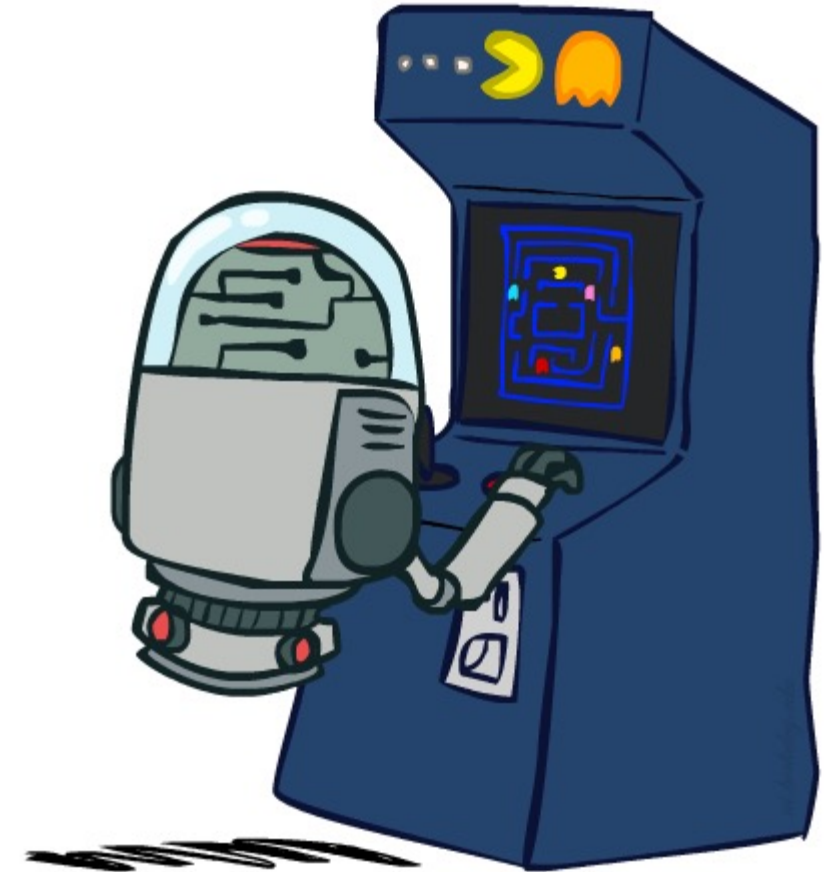
Types of Games

- 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

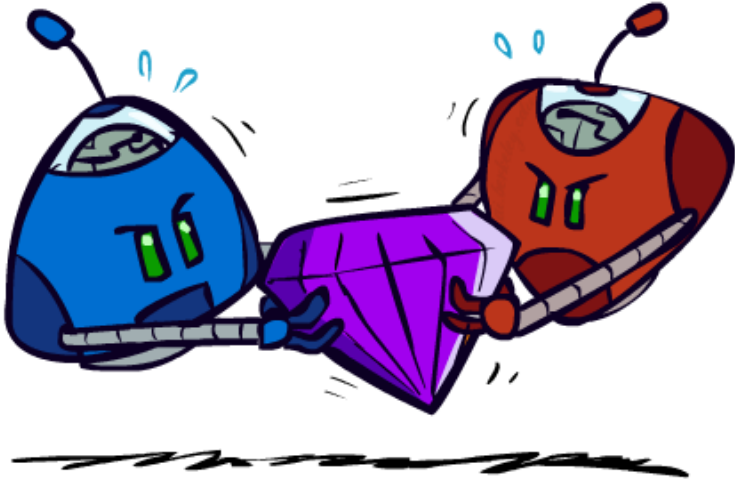


Deterministic Games

- Many possible formalizations, one is:
 - States: S (start at s_0)
 - Players: $P=\{1\dots N\}$ (usually take turns)
 - Actions: A (may depend on player / state)
 - Transition Function: $S \times A \rightarrow S$
 - Terminal Test: $S \rightarrow \{t, f\}$
 - Terminal Utilities: $S \times P \rightarrow R$
- Solution for a player is a **policy**: $S \rightarrow A$



Zero-Sum Games



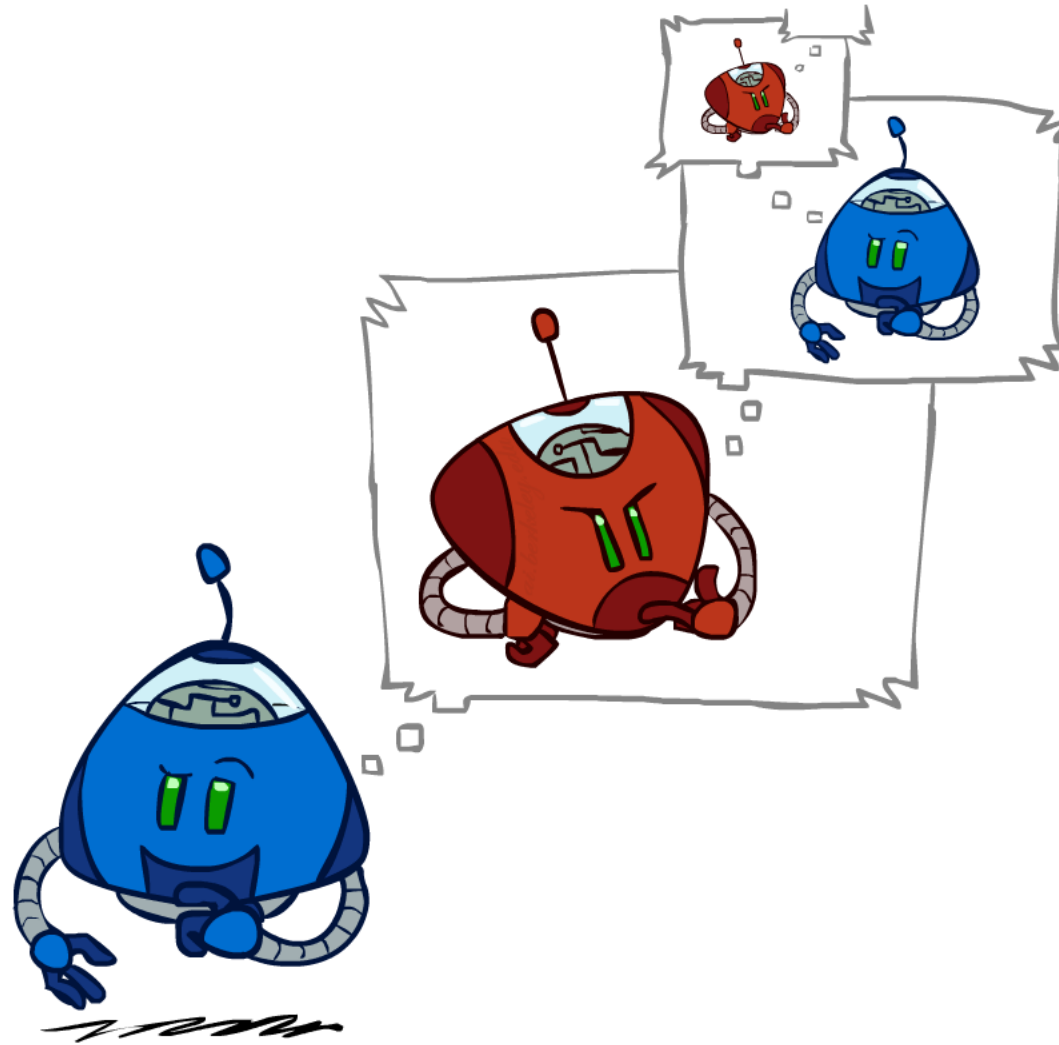
■ Zero-Sum Games

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

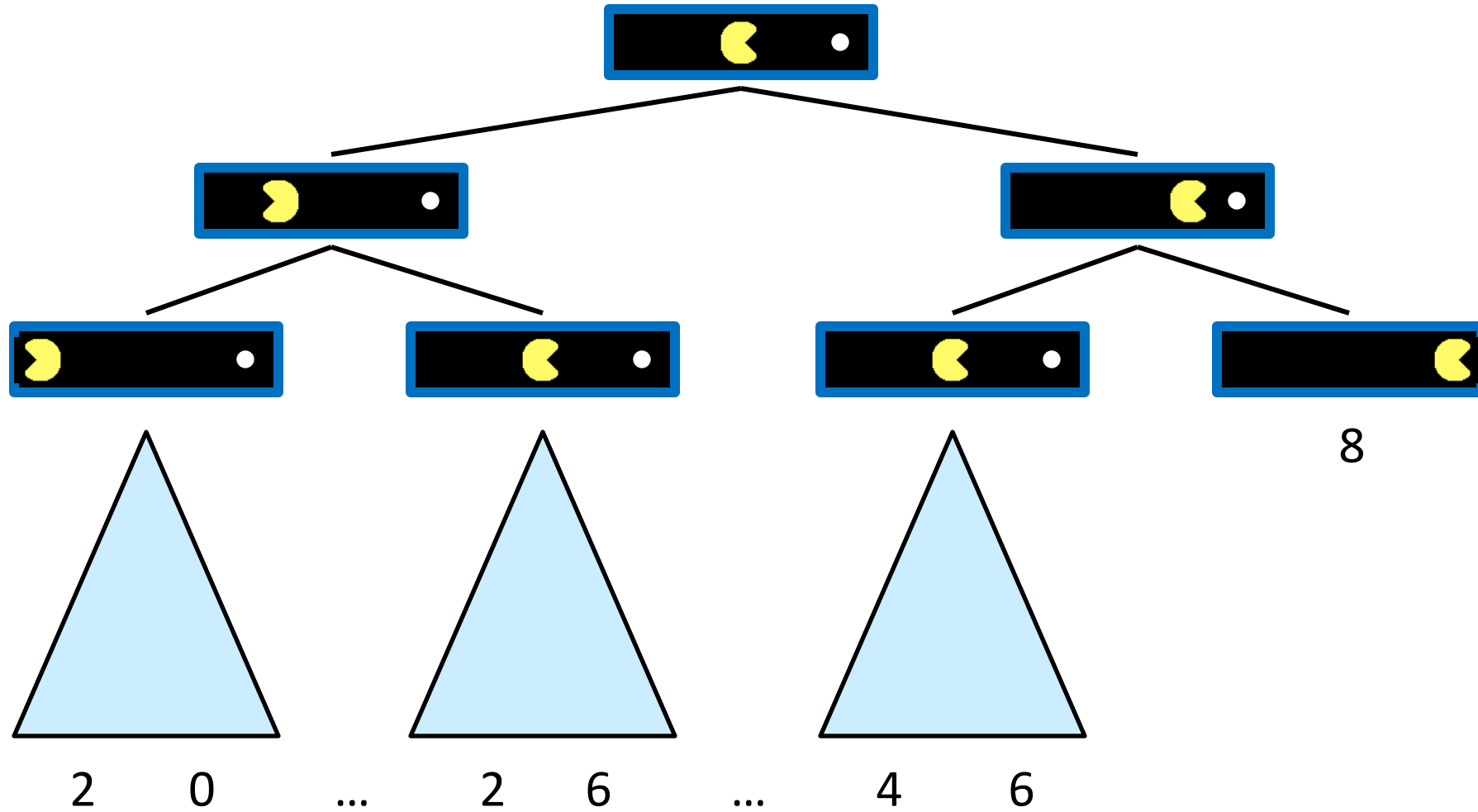
■ General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games

Solving Zero-Sum Games

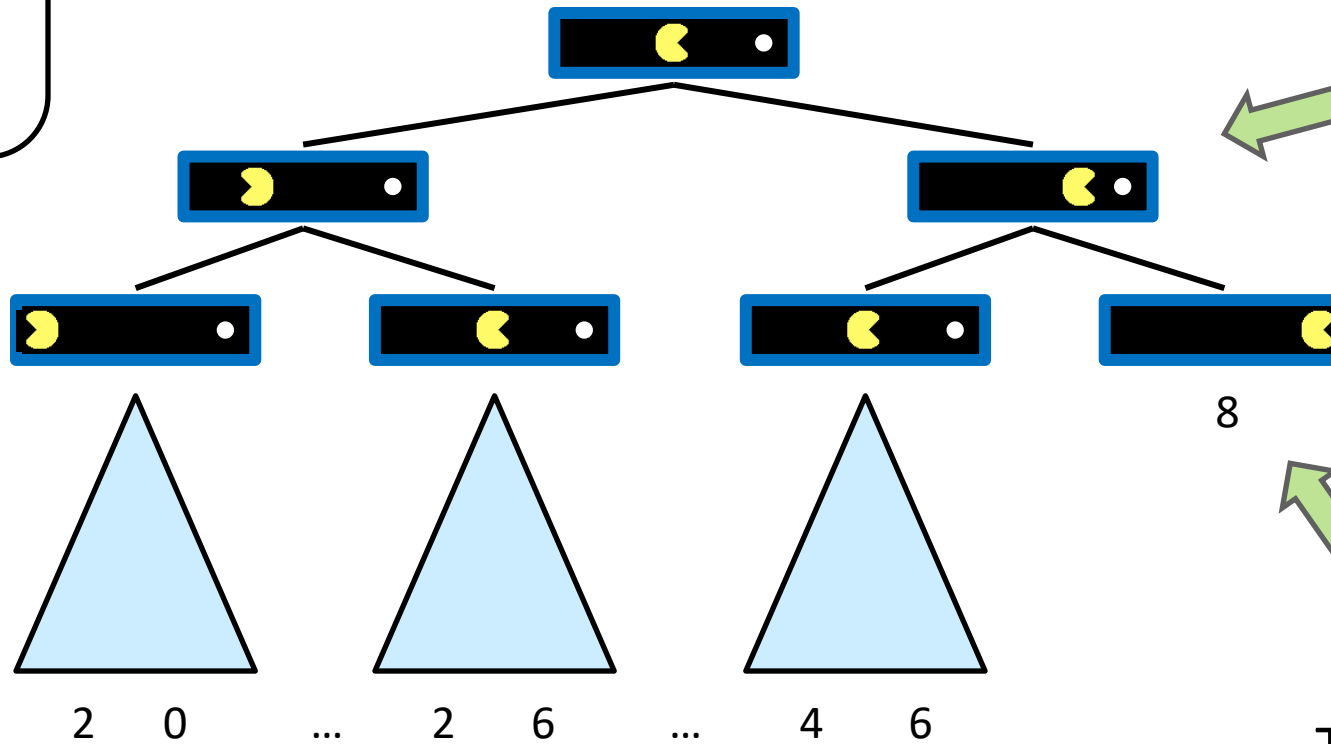


Single-Agent Search Trees



Value of a State

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



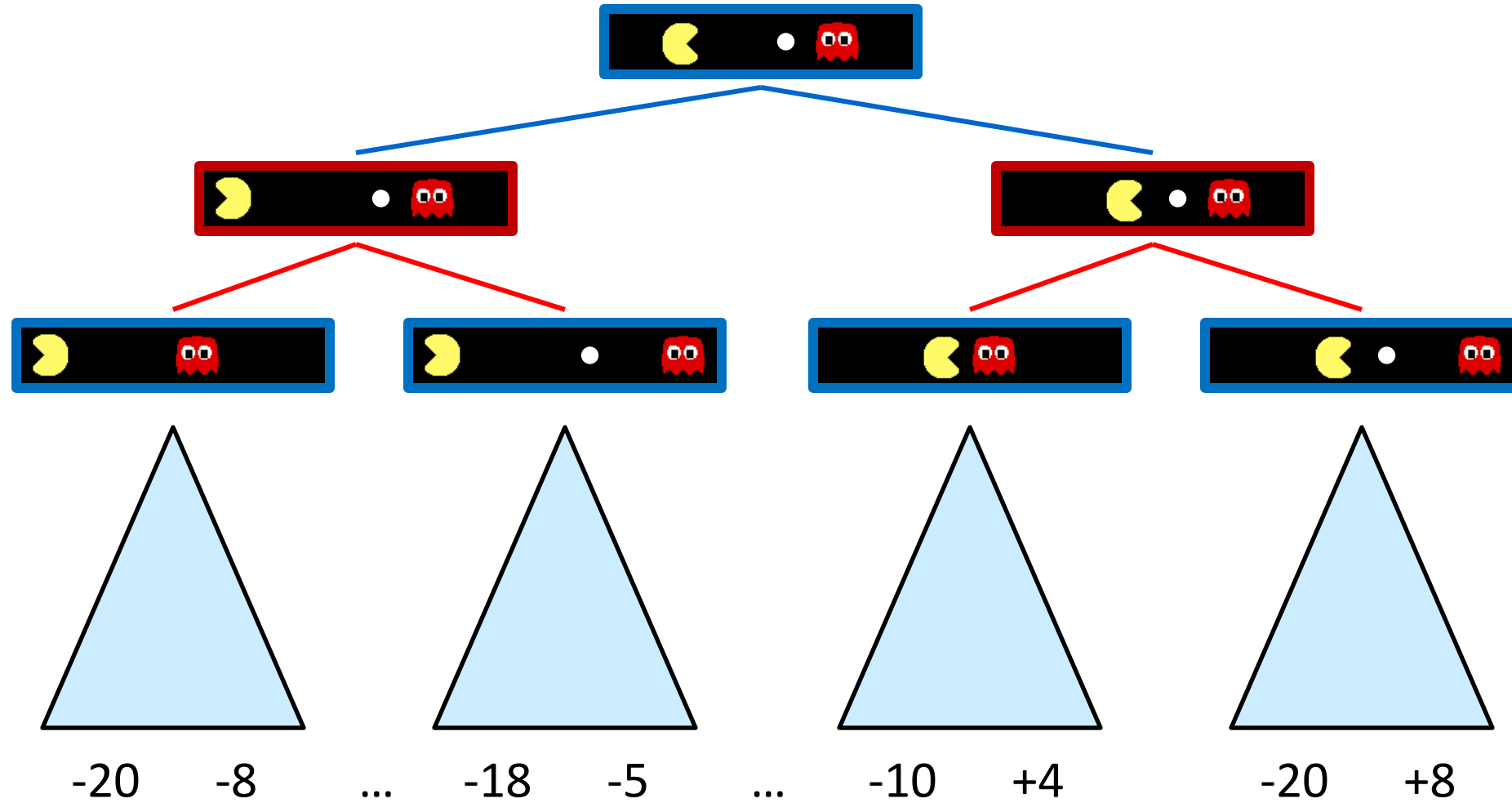
Non-Terminal States:

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

Terminal States:

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

Adversarial Game Trees



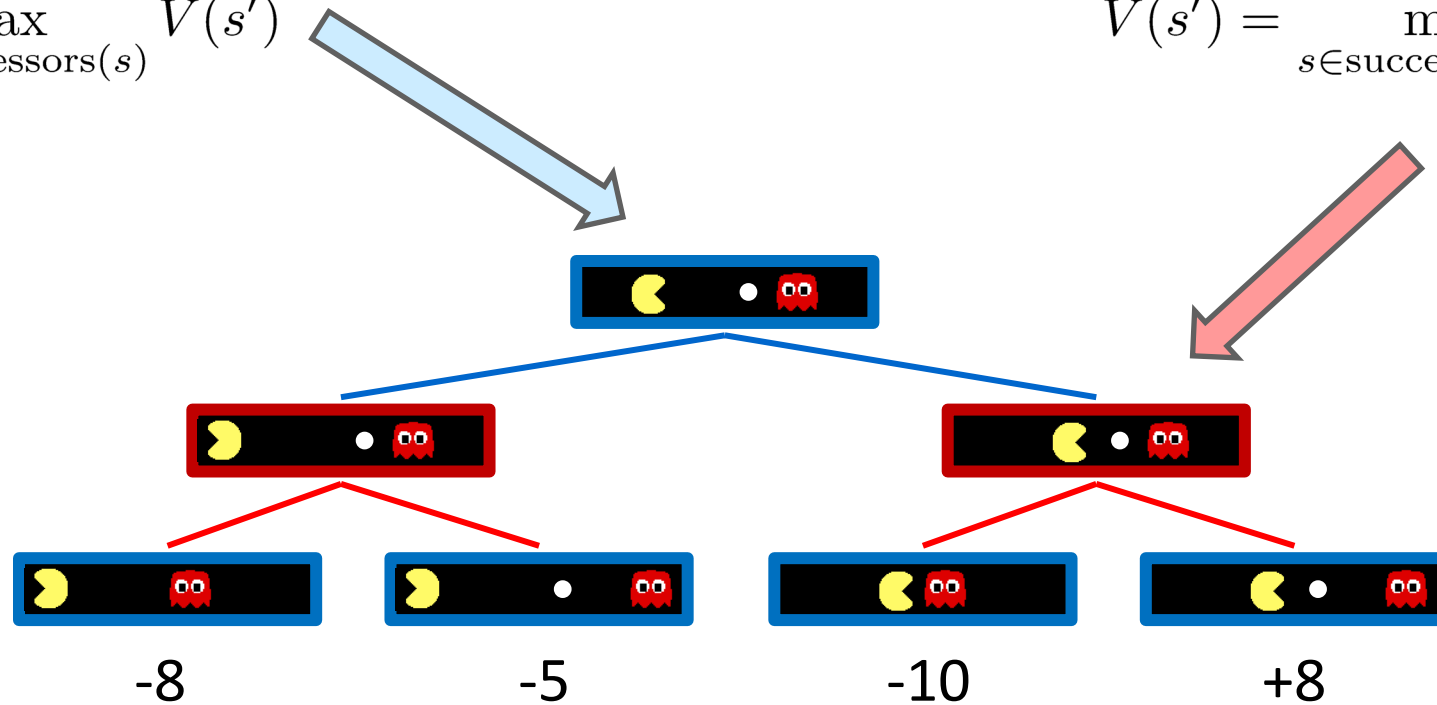
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

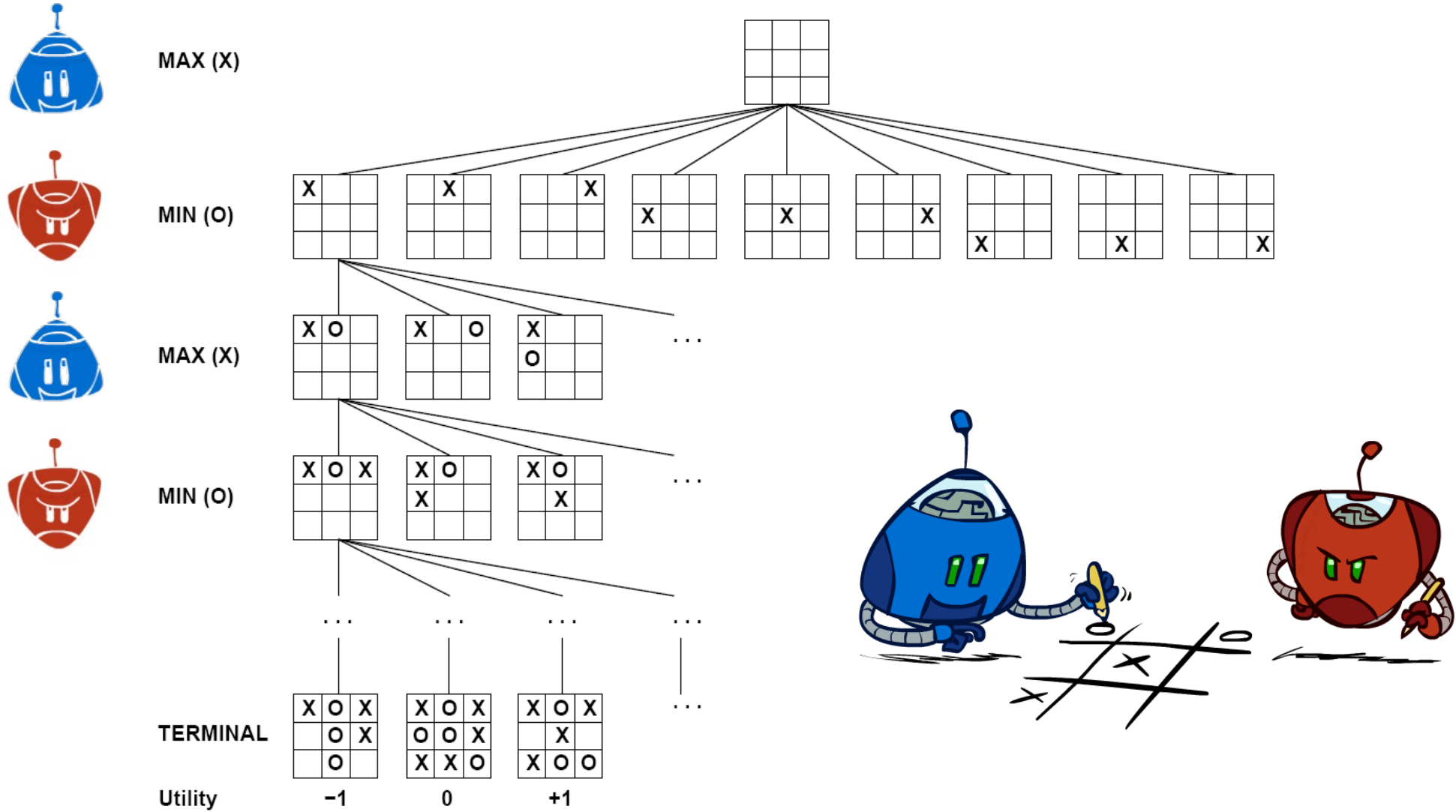
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

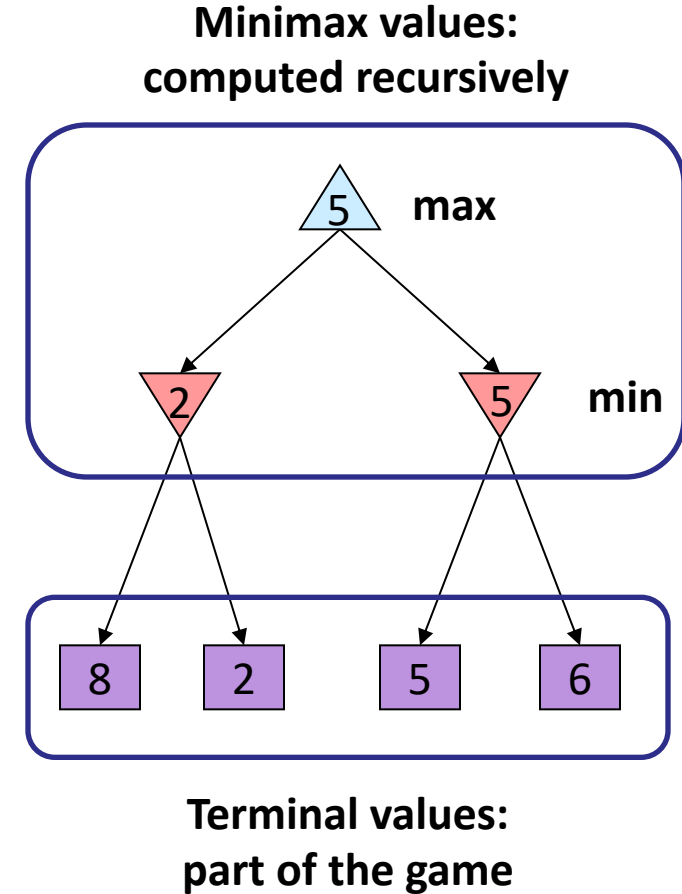
$$V(s) = \text{known}$$

Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

- **Deterministic, zero-sum games:**
 - Tic-tac-toe, chess, checkers
 - One player maximizes result
 - The other minimizes result
- **Minimax search:**
 - A state-space search tree
 - Players alternate turns
 - Compute each node's **minimax value**: the best achievable utility against a rational (optimal) adversary



Minimax Implementation

def max-value(state):

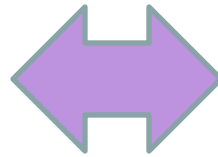
 initialize $v = -\infty$

 for each successor of state:

$v = \max(v, \text{min-value}(\text{successor}))$

 return v

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



def min-value(state):

 initialize $v = +\infty$

 for each successor of state:

$v = \min(v, \text{max-value}(\text{successor}))$

 return v

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

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 $v = -\infty$

for each successor of state:

$v = \max(v, \text{value}(\text{successor}))$

return v

```
def min-value(state):
```

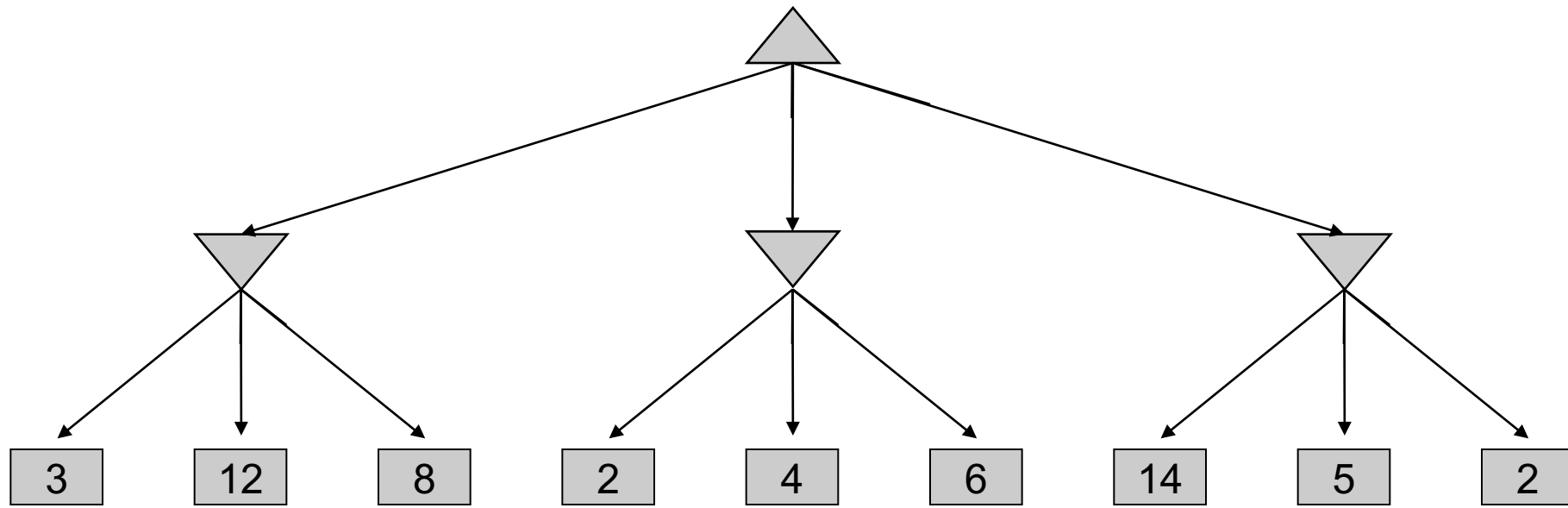
initialize $v = +\infty$

for each successor of state:

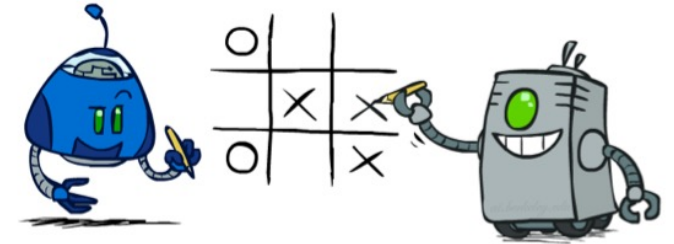
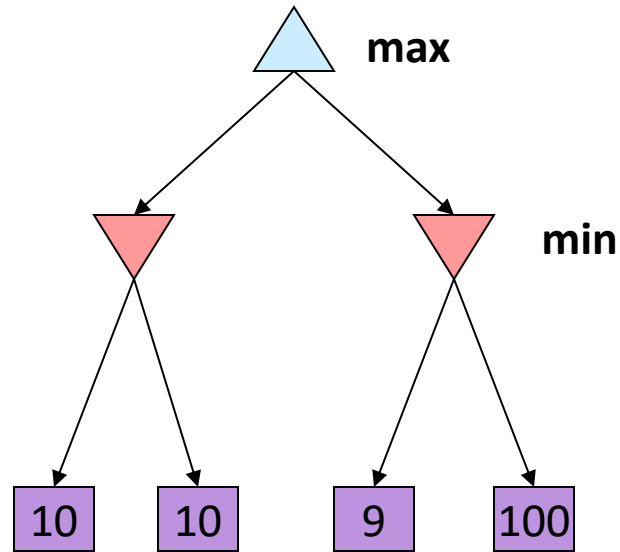
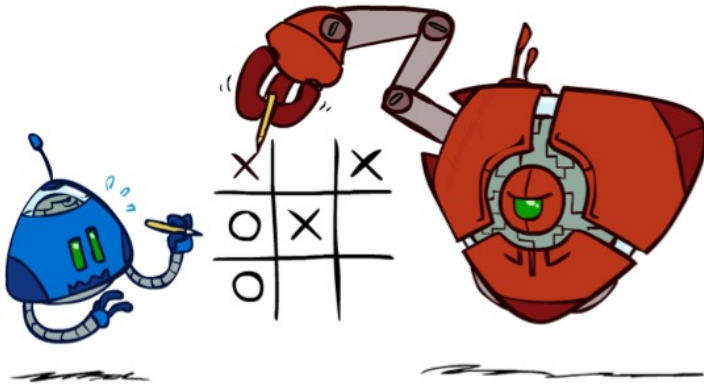
$v = \min(v, \text{value}(\text{successor}))$

return v

Minimax Example



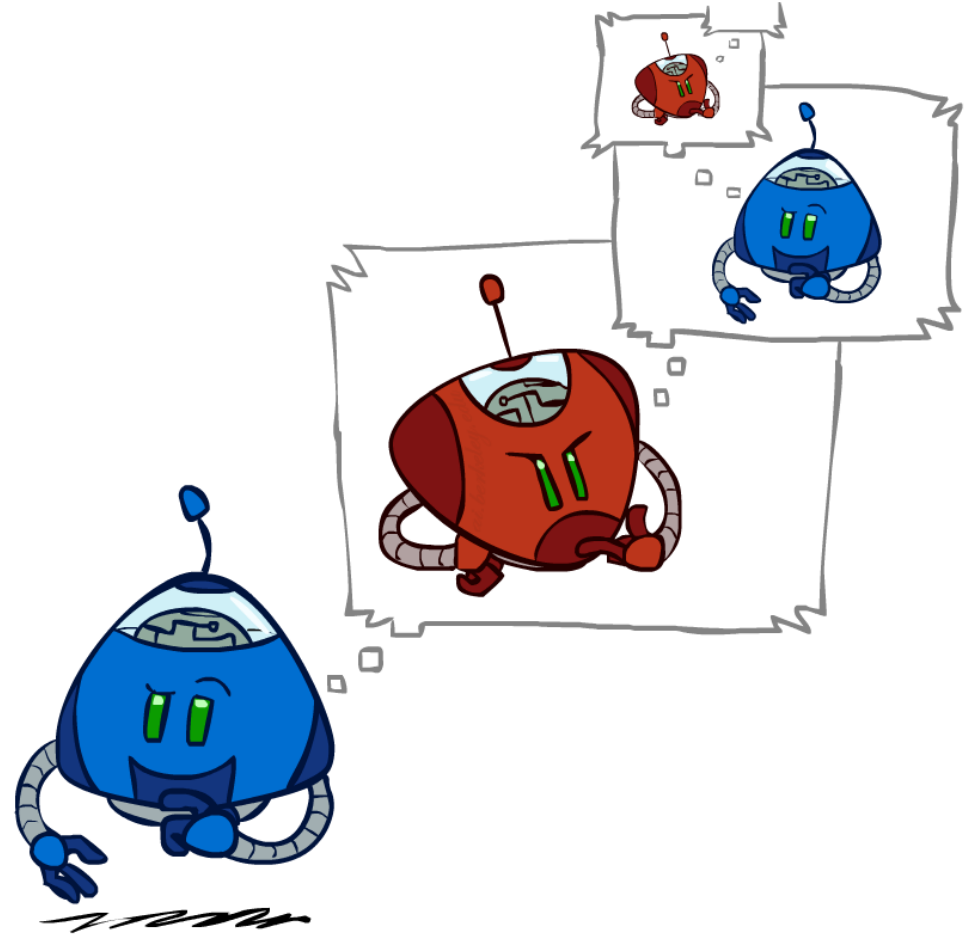
Minimax Properties



Optimal against a perfect player. Otherwise?

Minimax Efficiency

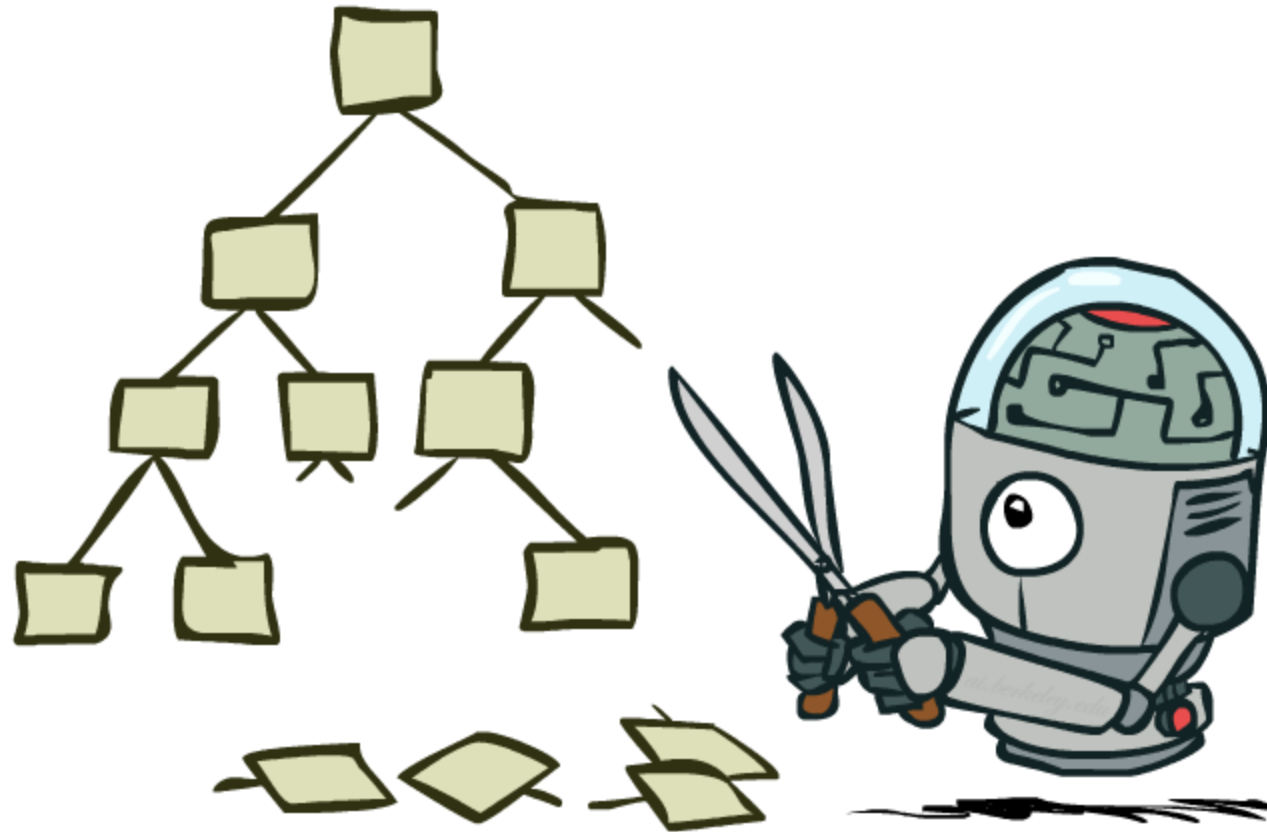
- How efficient is minimax?
 - Just like (exhaustive) DFS
 - Time: $O(b^m)$
 - Space: $O(bm)$
- Example: For chess, $b \approx 35$, $m \approx 100$
For Go, $b \approx 250-300$, $m \approx 150$
 - Exact solution is completely infeasible
 - But, do we need to explore the whole tree?



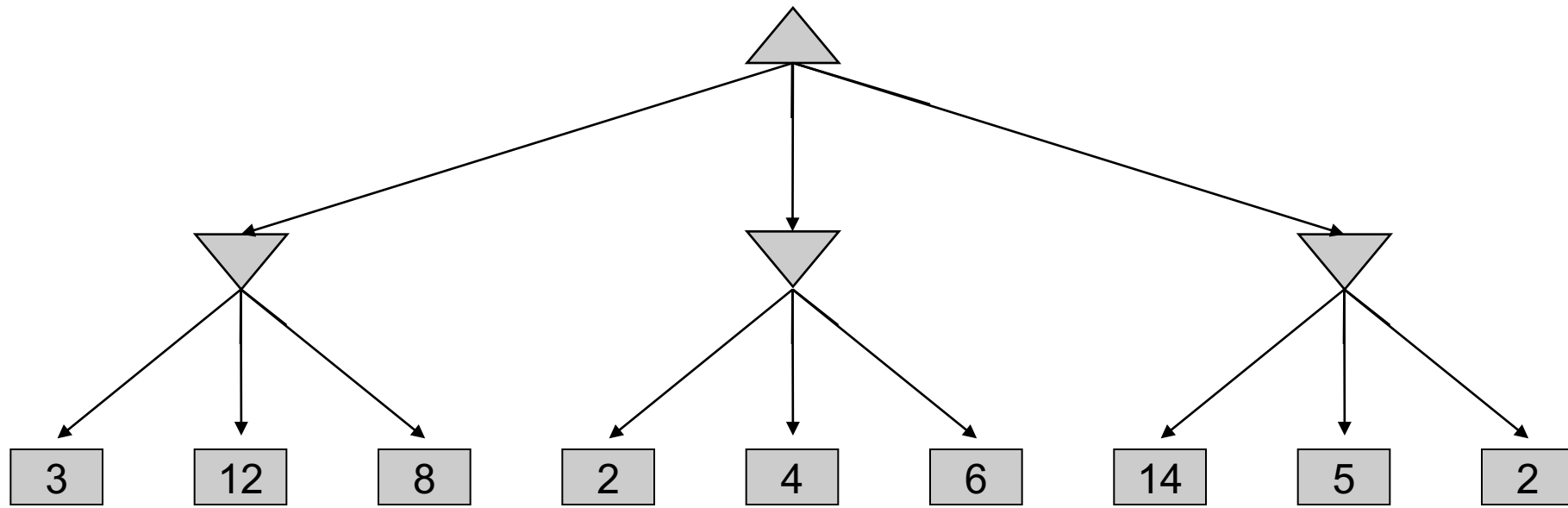
Overcoming Resource Limits



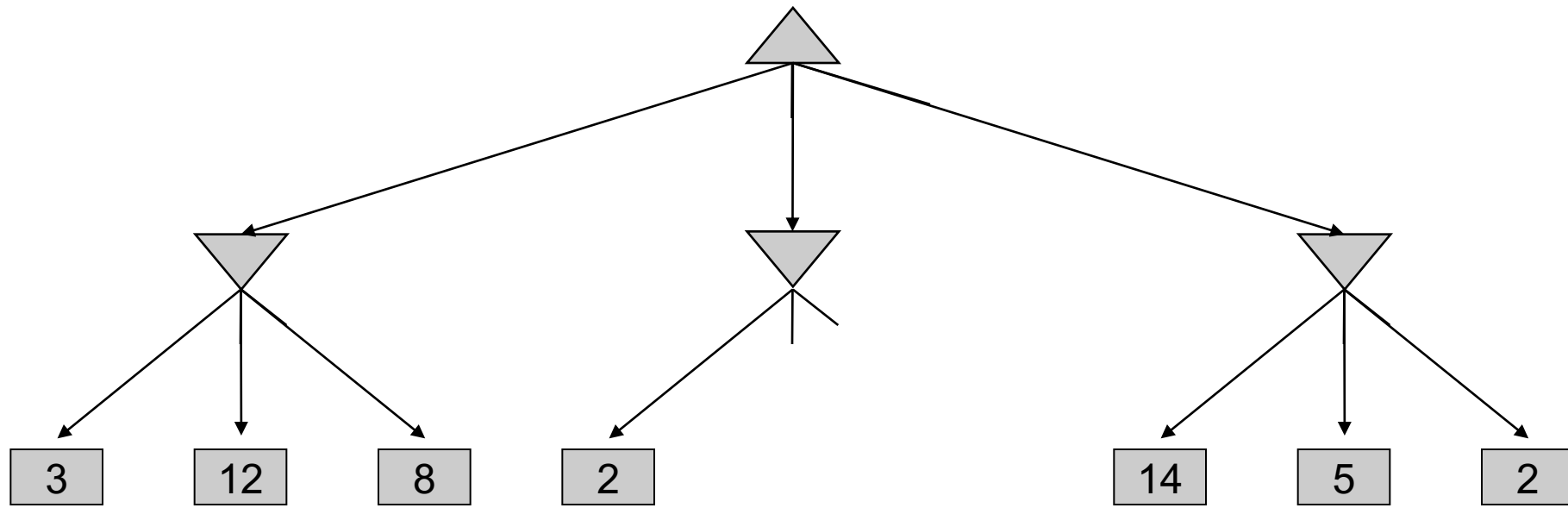
Game Tree Pruning



Minimax Example

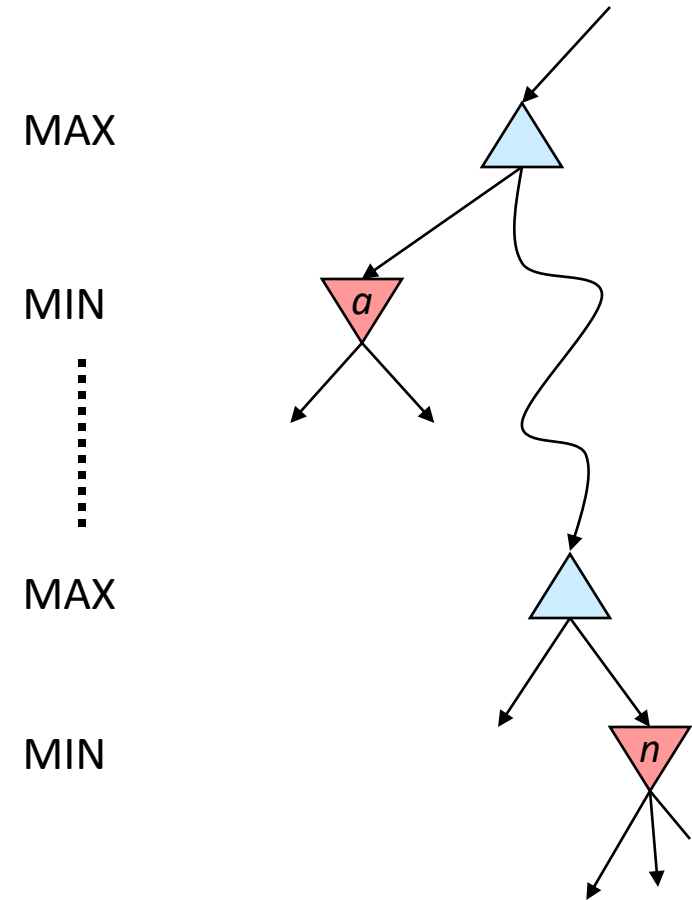


Minimax Pruning



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

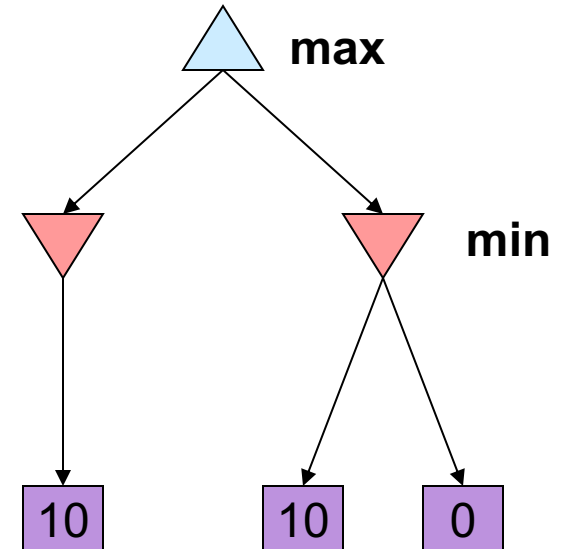
α : MAX's best option on path to root
 β : MIN's best option on path to root

```
def max-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = -\infty$   
    for each successor of state:  
         $v = \max(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v \geq \beta$  return  $v$   
         $\alpha = \max(\alpha, v)$   
    return  $v$ 
```

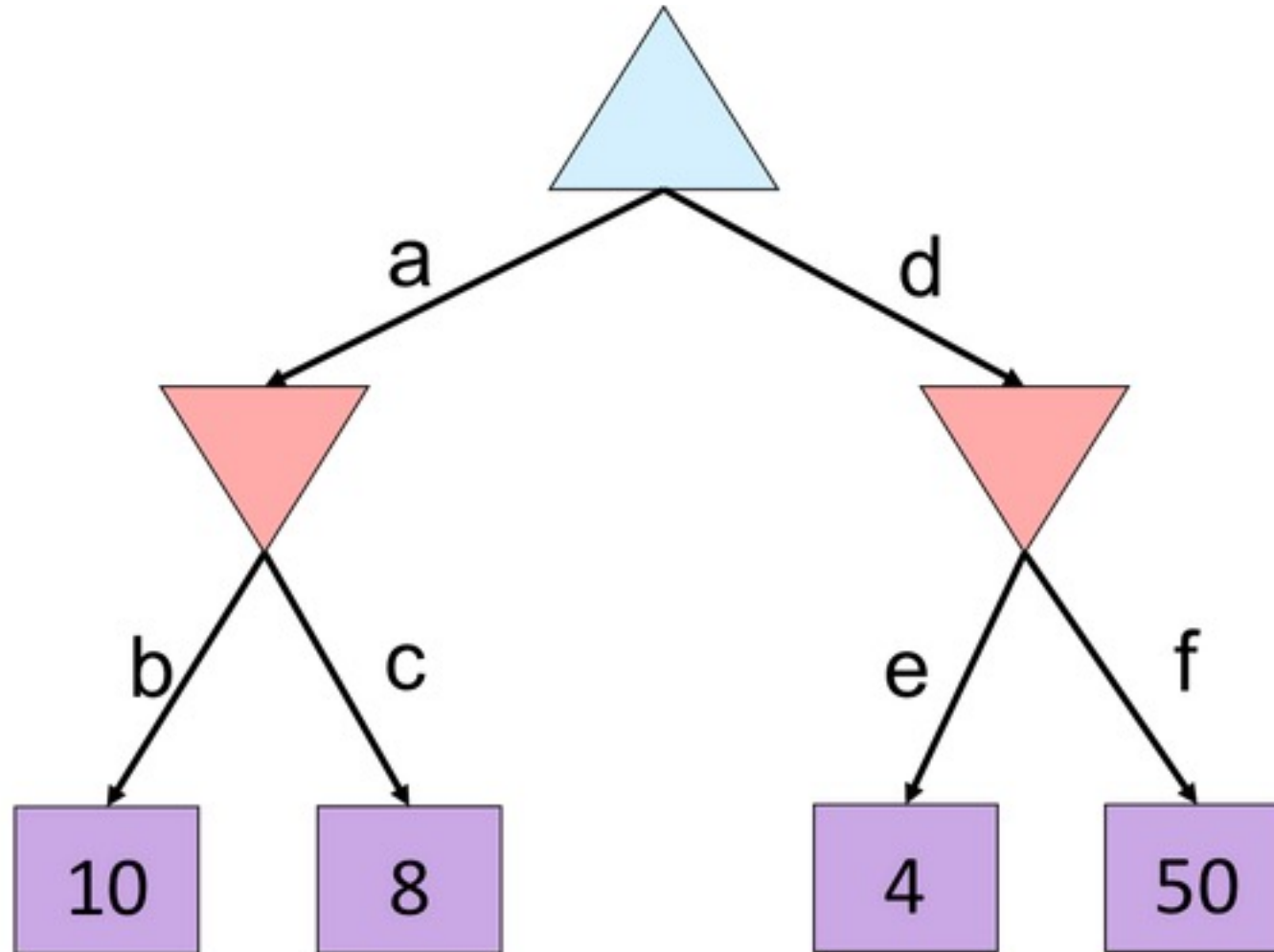
```
def min-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = +\infty$   
    for each successor of state:  
         $v = \min(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v \leq \alpha$  return  $v$   
         $\beta = \min(\beta, v)$   
    return  $v$ 
```

Alpha-Beta Pruning Properties

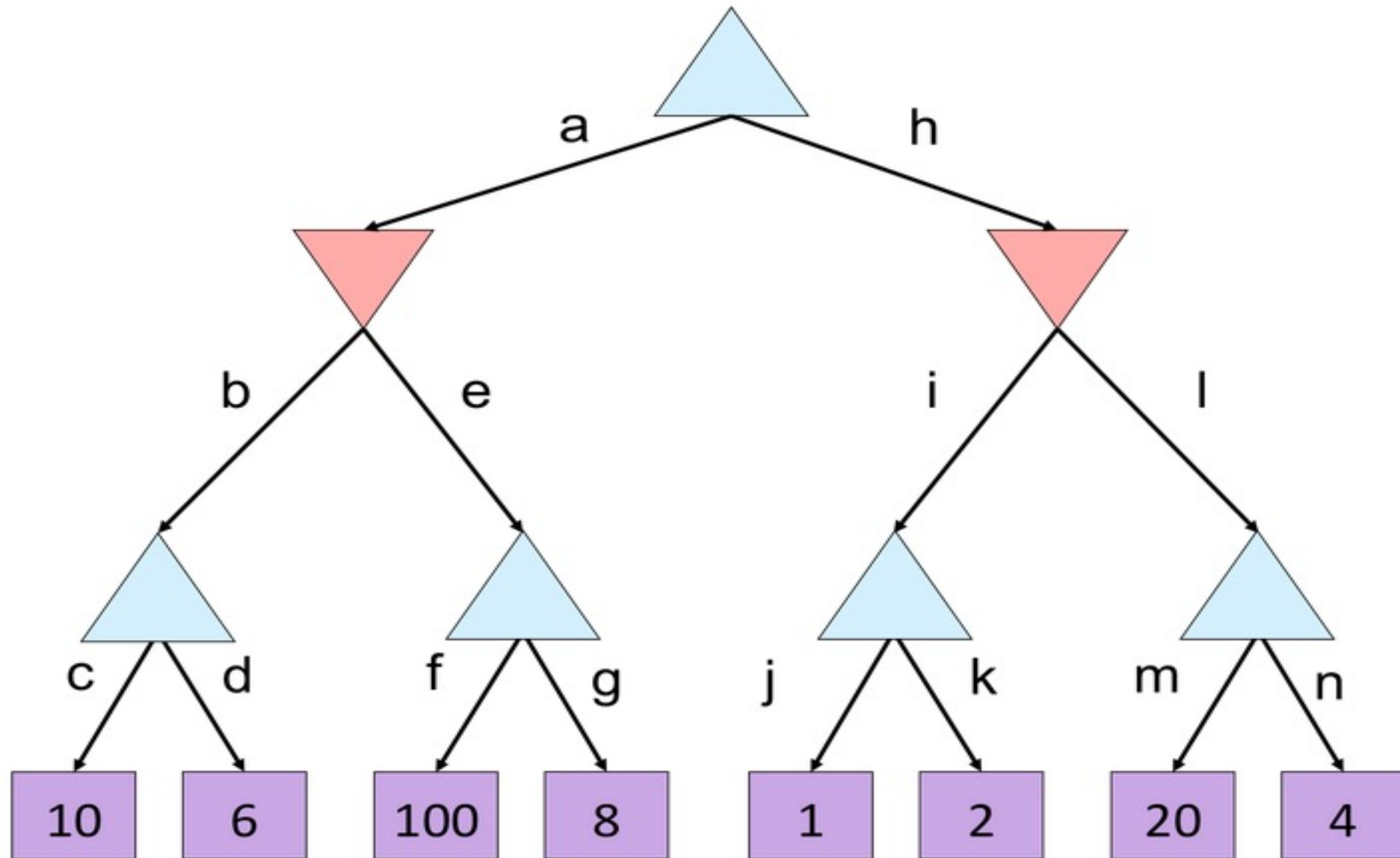
- This pruning has **no effect** on minimax value computed for the root!
- Values of intermediate nodes might be wrong
 - Important: children of the root may have the wrong value
 - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With “perfect ordering”:
 - Time complexity drops to $O(b^{m/2})$
 - Doubles solvable depth!
 - Full search of, e.g. chess, is still hopeless...
- This is a simple example of **metareasoning** (computing about what to compute)



Alpha-Beta Quiz



Alpha-Beta Quiz 2

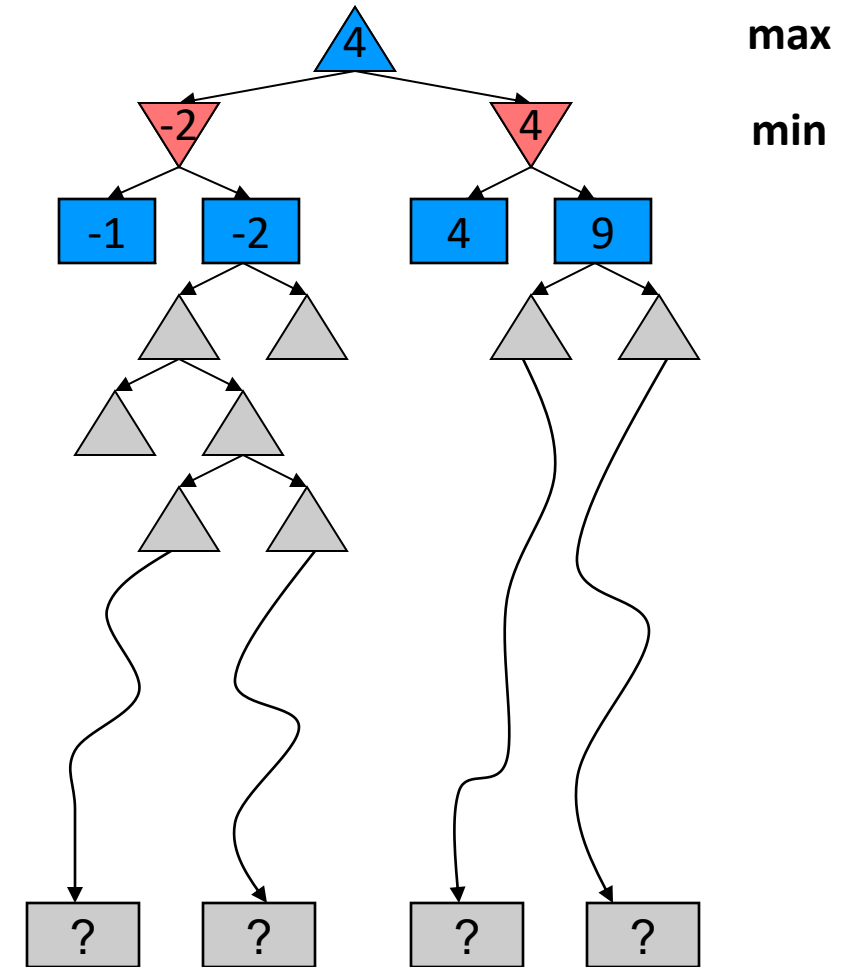


Overcoming Resource Limits

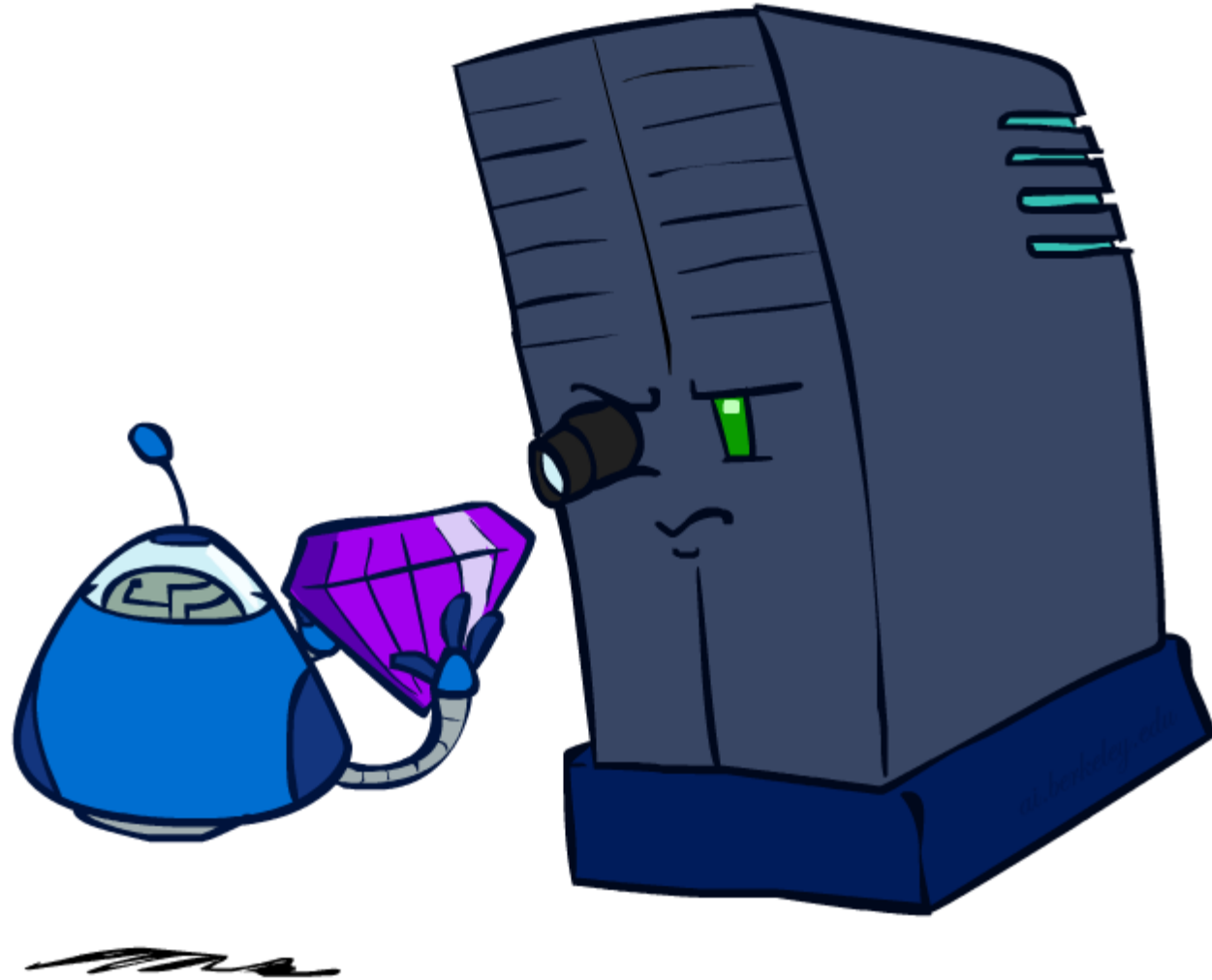


Limiting Depth

- 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

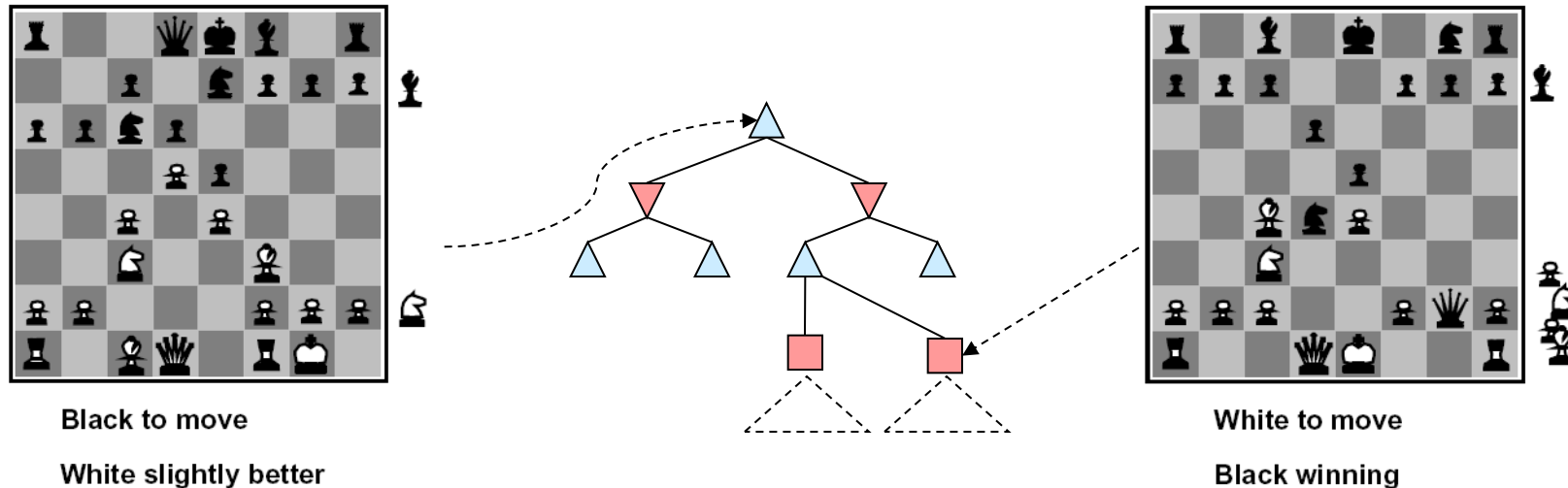


Evaluation Functions



Evaluation Functions

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



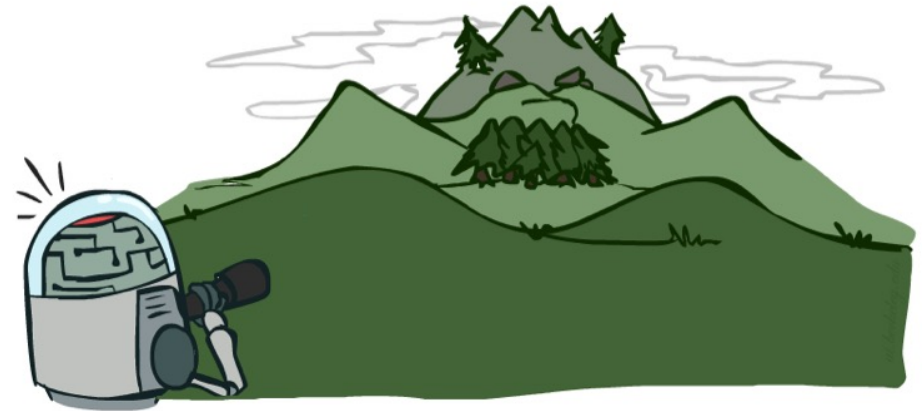
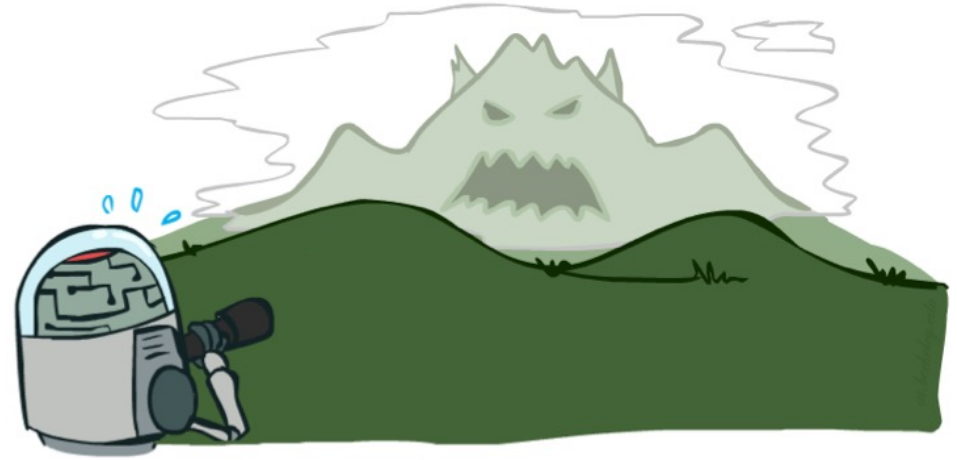
- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

- e.g. $f_1(s) = (\text{num white queens} - \text{num black queens})$, etc.

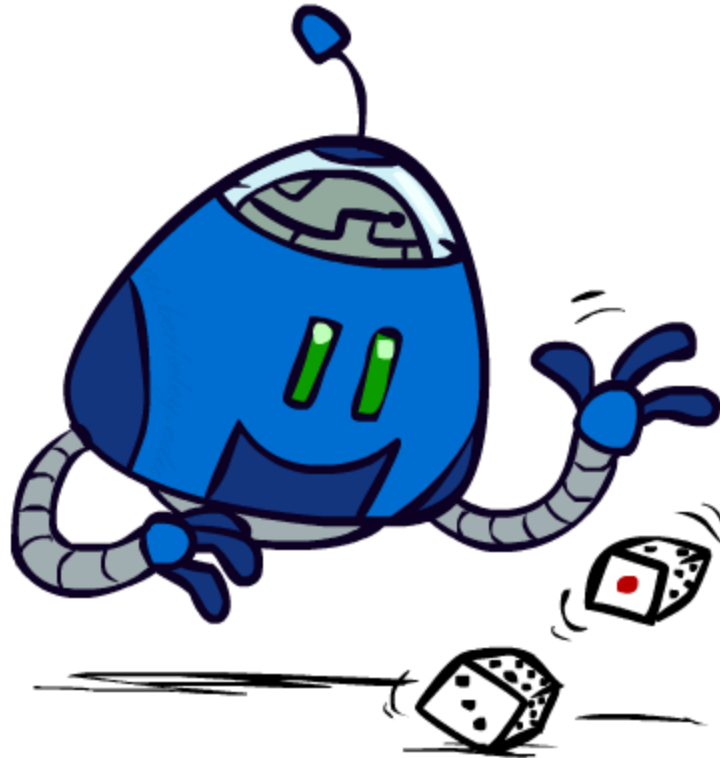
Depth Matters

- 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

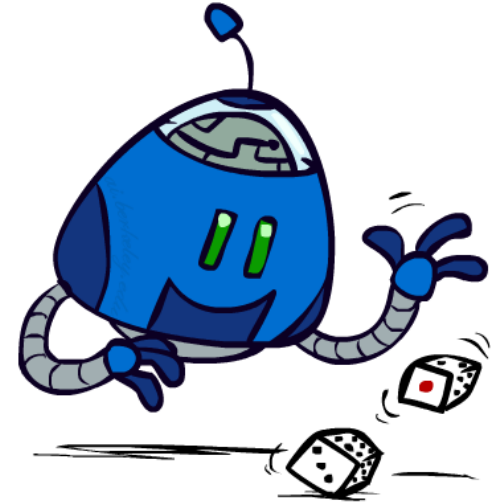
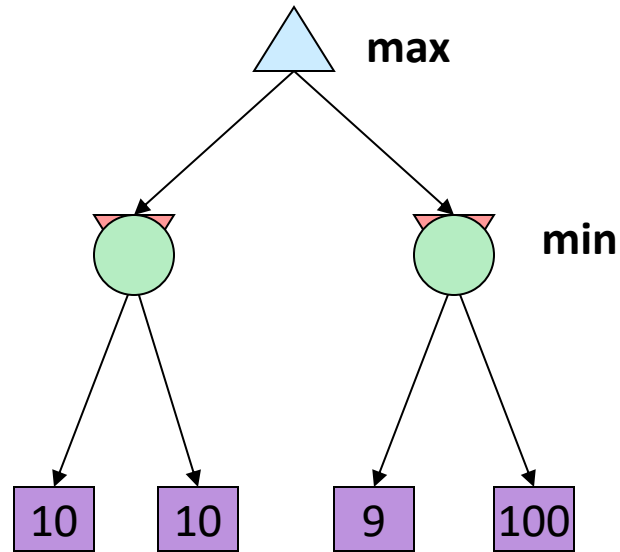
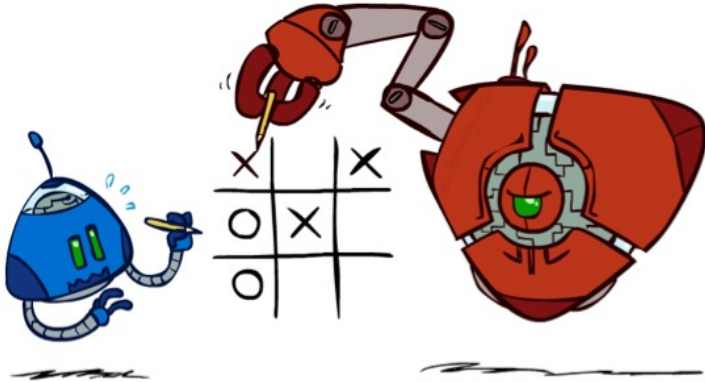


Uncertain Outcomes

- Why do we care about uncertainty and randomness?
 - Want to model random events happening in the world
 - Build efficient algorithms with random sampling (Monte Carlo Tree Search)



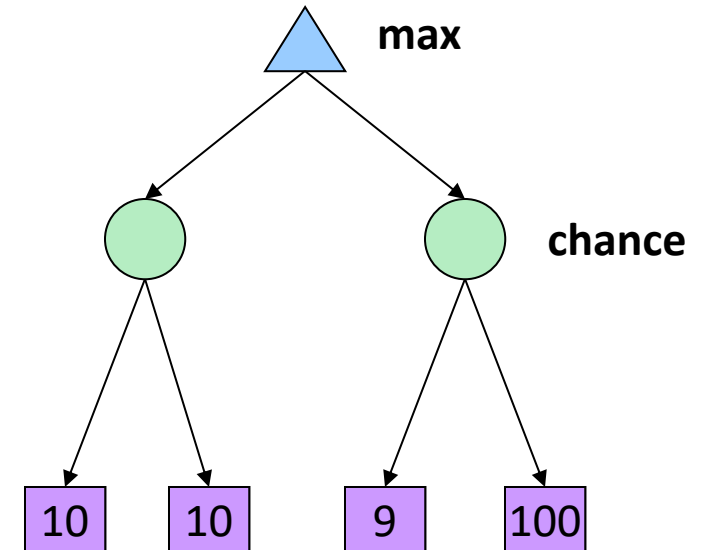
Worst-Case vs. Average Case



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



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 $v = -\infty$

for each successor of state:

$v = \max(v, \text{value}(\text{successor}))$

return v

```
def exp-value(state):
```

initialize $v = 0$

for each successor of state:

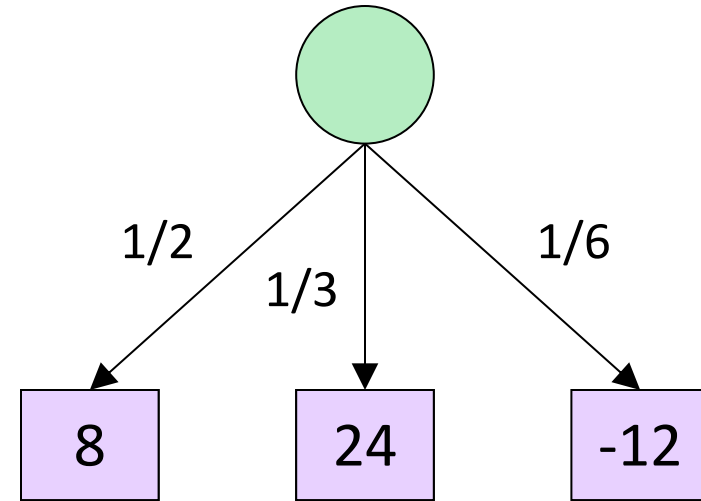
$p = \text{probability}(\text{successor})$

$v += p * \text{value}(\text{successor})$

return v

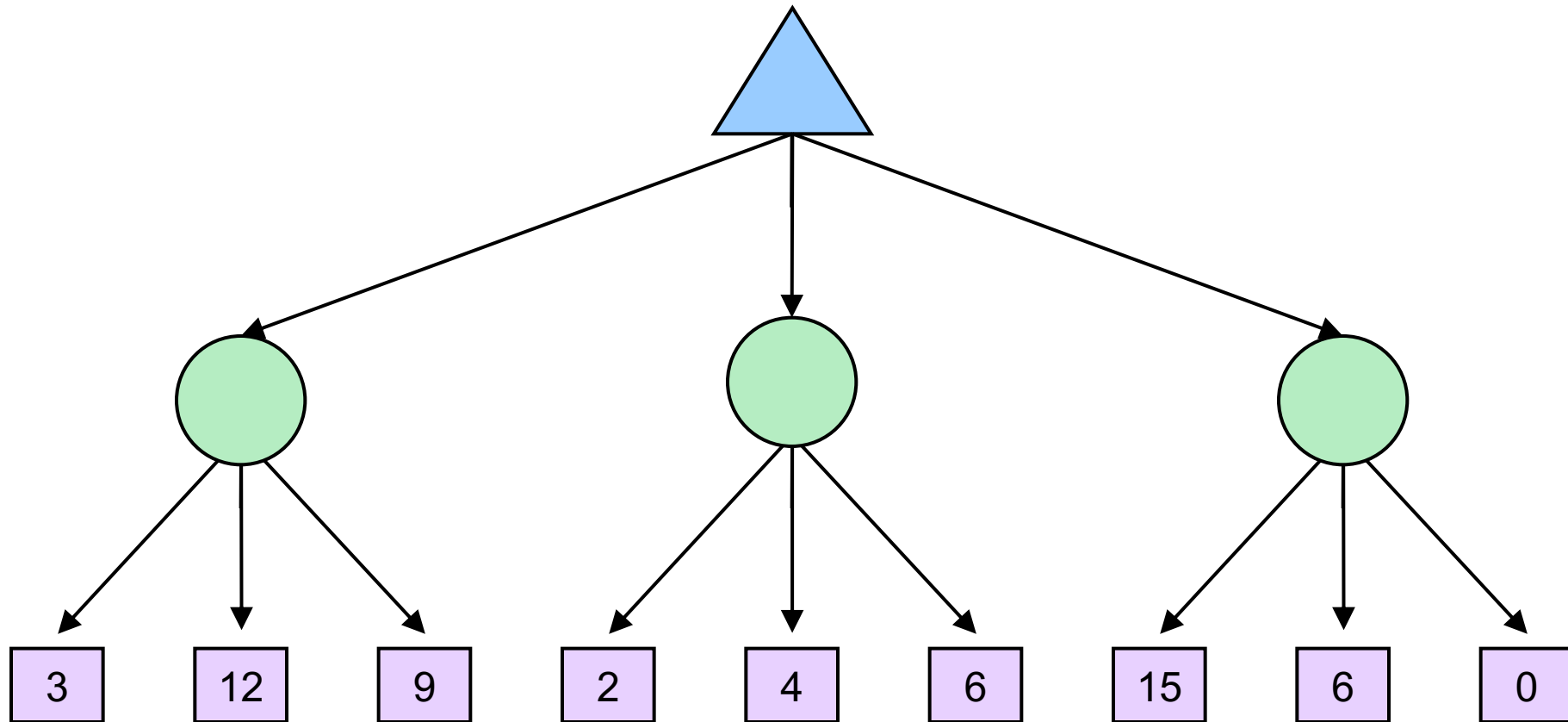
Expectimax Pseudocode

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

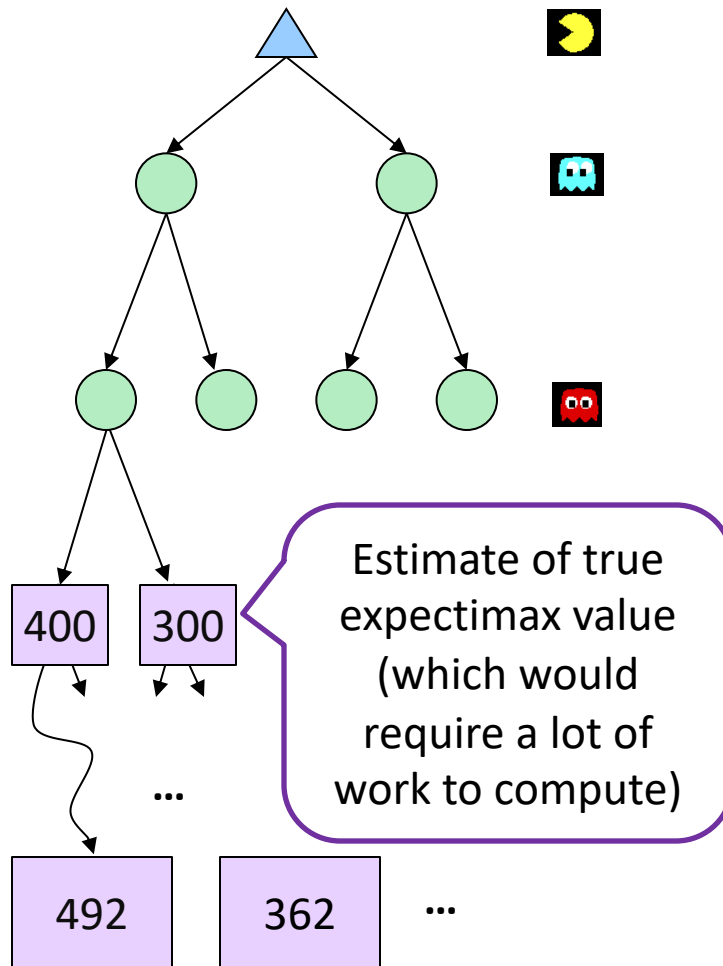


$$v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10$$

Expectimax Example

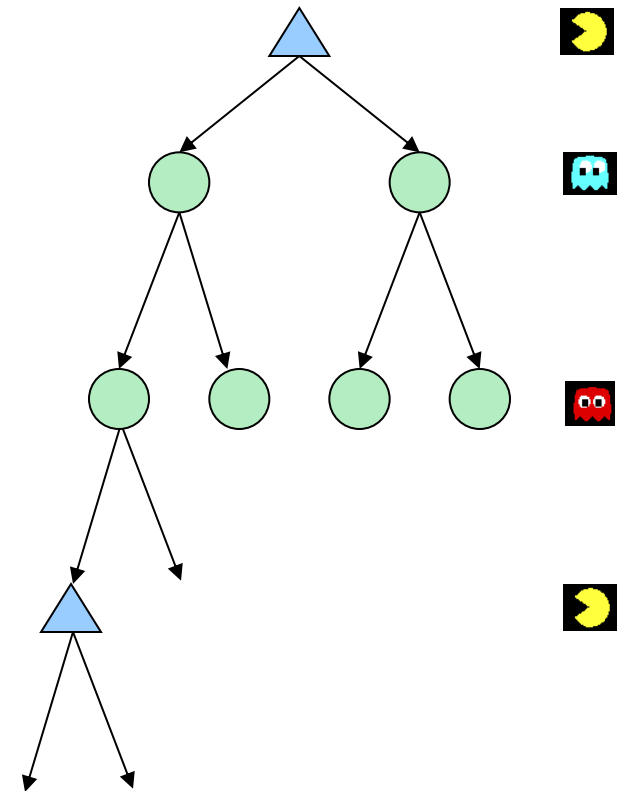


Depth-Limited Expectimax



What Probabilities to Use?

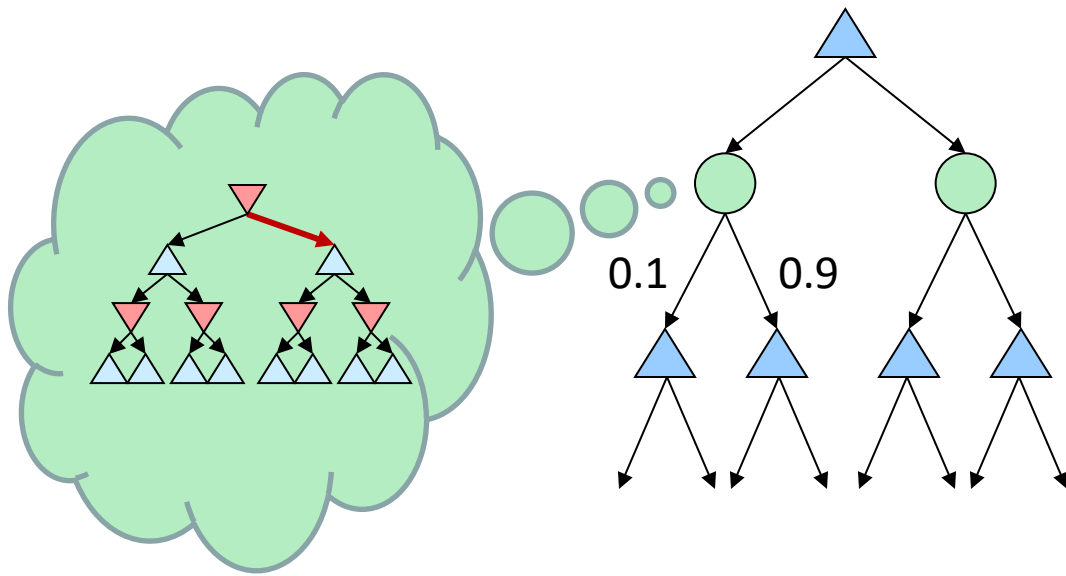
- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
 - Model could be a simple uniform distribution (roll a die)
 - Model could be sophisticated and require a great deal of computation
 - We have a chance node for any outcome out of our 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!

Quiz: Informed Probabilities

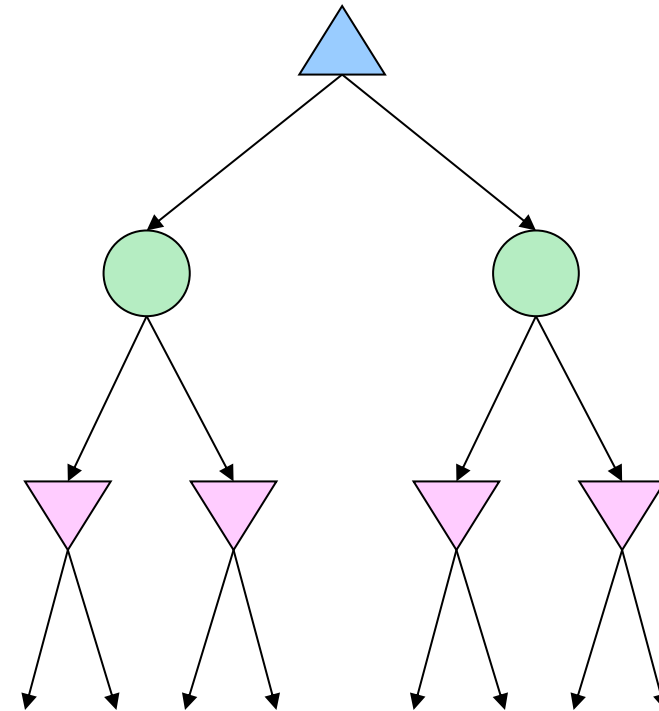
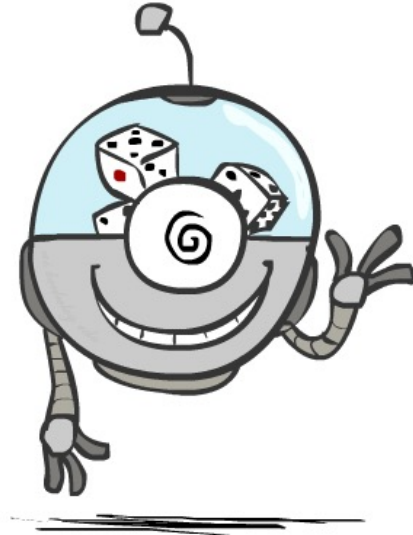
- Let's say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise
- Question: What tree search should you use?



- 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

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



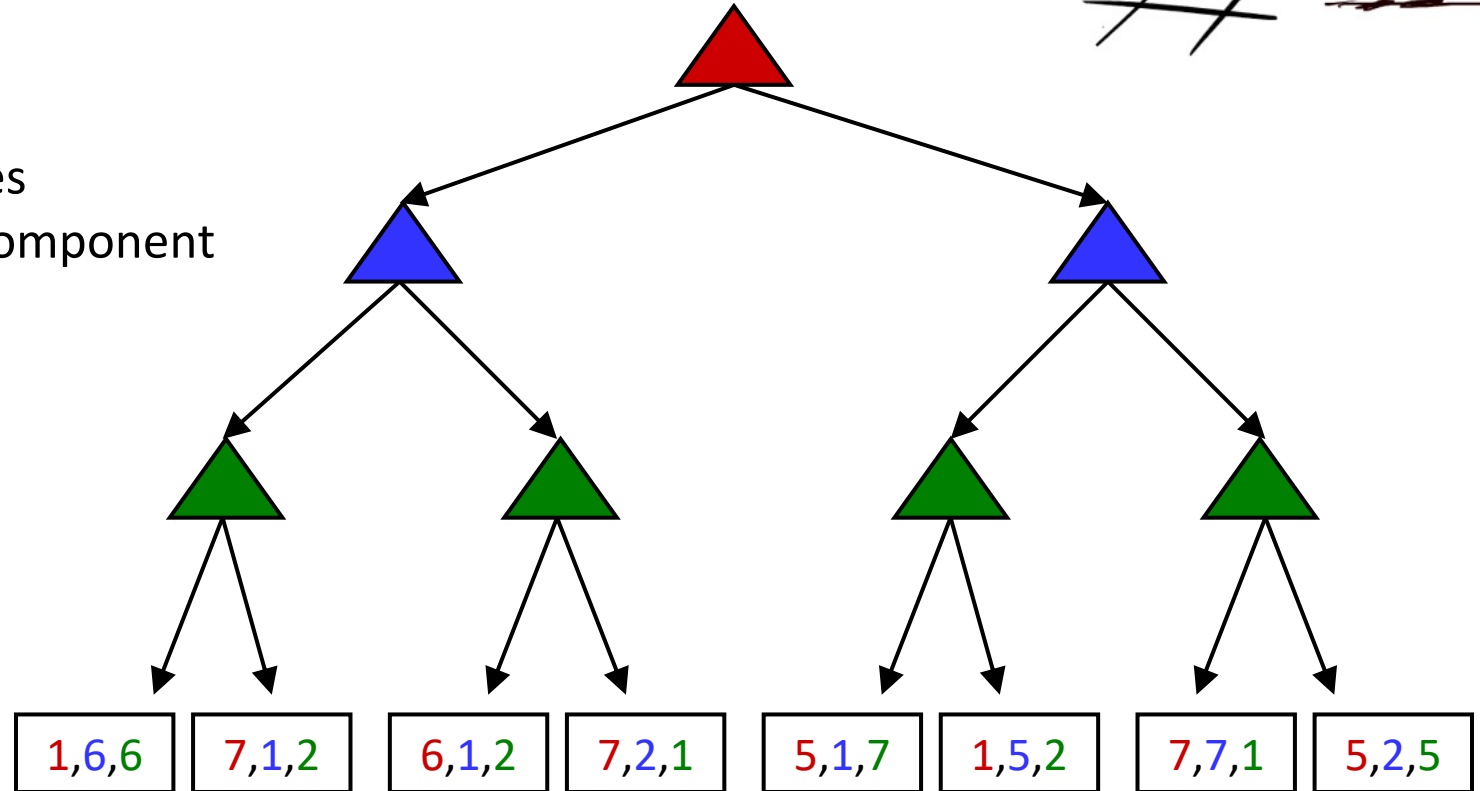
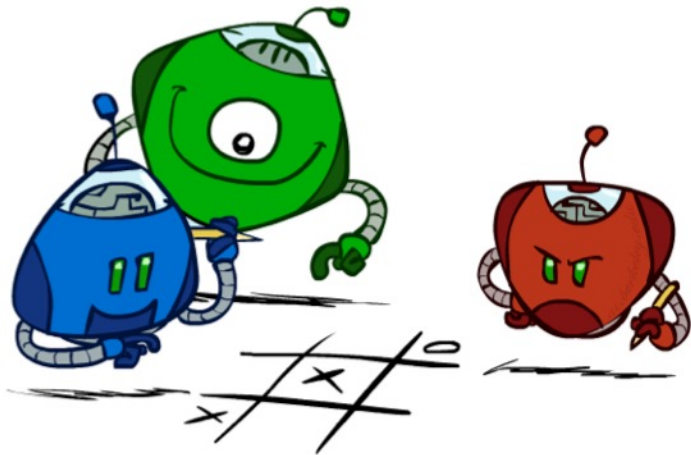
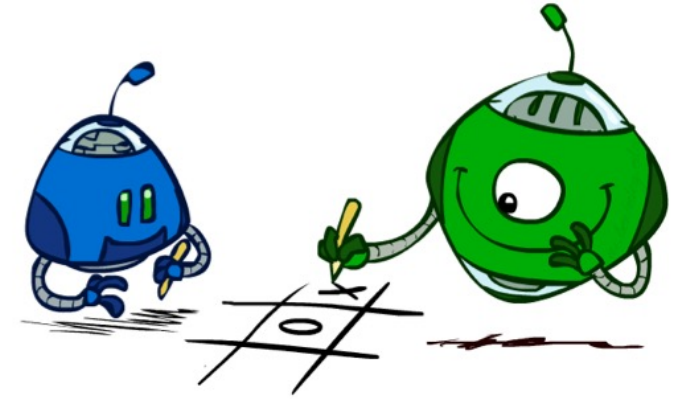
Example: Backgammon

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



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...



Difficulties with Search so Far

- Even with alpha-beta pruning and limited depth, large **b** is an issue (recall best-case time complexity is $b^{m/2}$)
 - Possible for chess: with alpha-beta, $35^{(8/2)} \approx 1M$; depth 8 is quite good
 - Difficult for Go: $300^{(8/2)} \approx 8 \text{ billion}$
- Limiting depth requires us to design good evaluation functions
 - Not a general solution: need to design new evaluation function for each new problem
 - Bad evaluation function may make solutions inefficient or biased
 - The trend in AI is to prefer general methods and less human tweaking

What we did today

- Introduced multi-agent games

- Come up in many places in AI
- Different types of games (focused on *zero-sum* today)



- Modified search trees to include opponent actions

- Assumed opponent acts in a way that is worst for you
- Called this *Minimax Search*

- Looked at efficiency (not good) and suggested two ways to improve it

- Alpha-Beta pruning (exact, some gains but not huge)
- Limiting tree depth (big gains, but not exact and needs heuristics)

What we did today

- Extended games to include uncertain outcomes
- Modified search to reason about uncertain outcomes
 - Return *expected value* for a chance node