Games

Dr. Angelica Lim
Assistant Professor
School of Computing Science
Simon Fraser University, Canada
Nov. 11, 2024

Slide Credits:Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).
Pearson 2021

Recall: Reinforcement learning

There are hundreds of approaches. We will learn basic approaches for each of the following families:

1. **Model-based**: Learn the model, solve it, execute the solution
2. **Model-free**: Learn values from experiences, use to make decisions
   - Direct evaluation
   - Temporal difference learning
   - Q-learning

Passive RL

Passive RL

Passive RL

Active RL

Midterm common issues

- What is the goal or what is the purpose of algorithm X?
  - When (in what cases) and why should we use it? **Not** what does it do.
- Mixing up K-NN with K-Means
- Cluster != classify
- SGD and backprop are not alternatives to gradients (they use them)
- Linear regression != classification

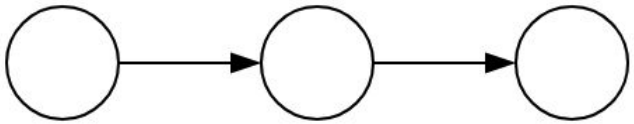
Recap

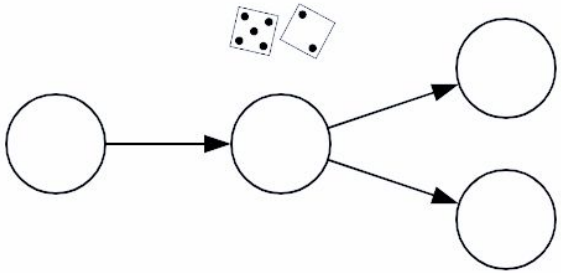
- RL solves MDPs via direct experience of transitions and rewards
- There are several approaches:
  - Learn the MDP model (transitions and rewards) and solve it
  - Learn V directly from sums of rewards, or by TD local adjustments
  - Learn Q by local Q-learning adjustments, use it directly to pick actions

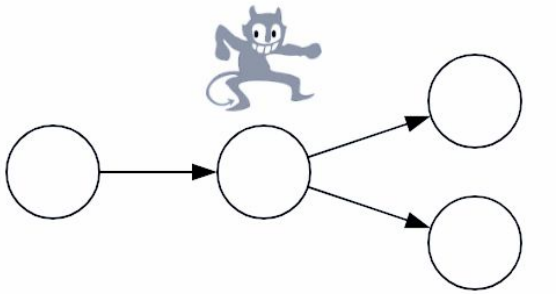
Course Overview

Week 1 : Getting to know you
Week 2 : Introduction to Artificial Intelligence
Week 3: Machine Learning I: Basic Supervised Models (Classification)
Week 4: Machine Learning II: Supervised Regression, Classification and Gradient Descent, K-Means
Week 5: Machine Learning III: Neural Networks and Backpropagation
Week 6 : Search
Week 7 : Markov Decision Processes
Week 8 : Midterm
Week 9 : Reinforcement Learning
Week 10 : Games
Week 11 : Hidden Markov Models
Week 12 : Bayesian Networks
Week 13 : Constraint Satisfaction Problems, Ethics and Explainability

State-based models





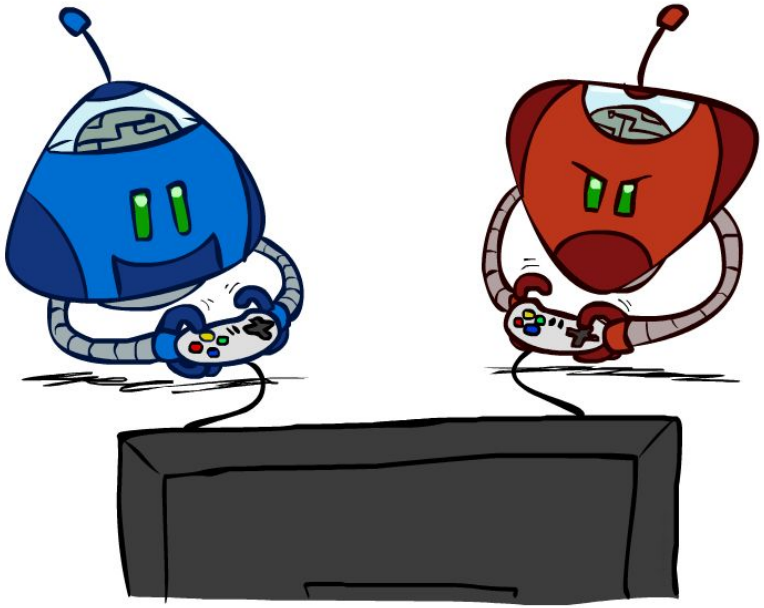


Search problems: when you have an environment with no uncertainty, ie. perfect information. But realistic settings are more complex

Markov decision processes (MDPs) handle situations with randomness, e.g. Blackjack

Game playing handles tasks where there is interaction with another agent. Adversarial games assume an opponent, e.g. Chess

Games: Minimax and Alpha-Beta Pruning

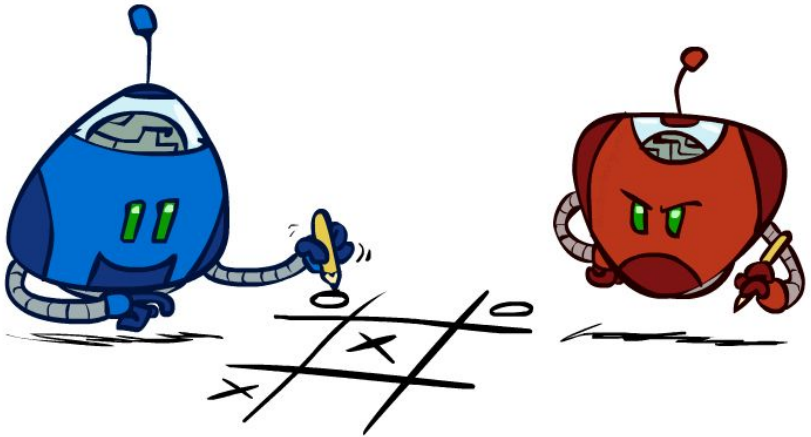


[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]

[Updated slides from: Stuart Russell and Dawn Song]

Outline

- History / Overview
- Minimax for Zero-Sum Games
- α-β Pruning
- Finite lookahead and evaluation



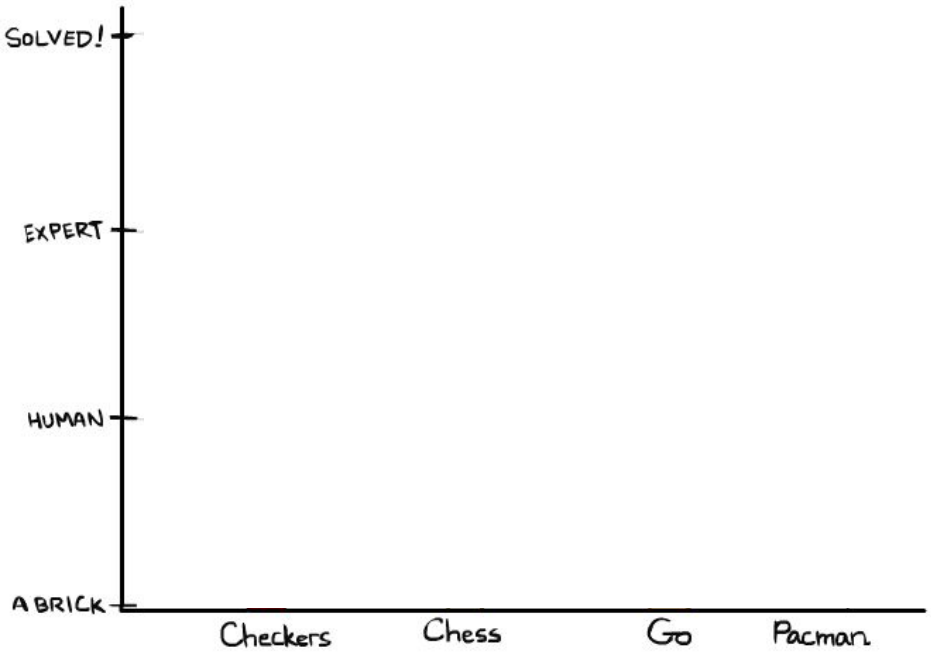
Game Playing: State of the Art

- **Checkers:**
  - 1950: First computer player
  - 1959: Samuel's self-taught program
  - 1995: First computer world champion*
  - 2007: Checkers solved!

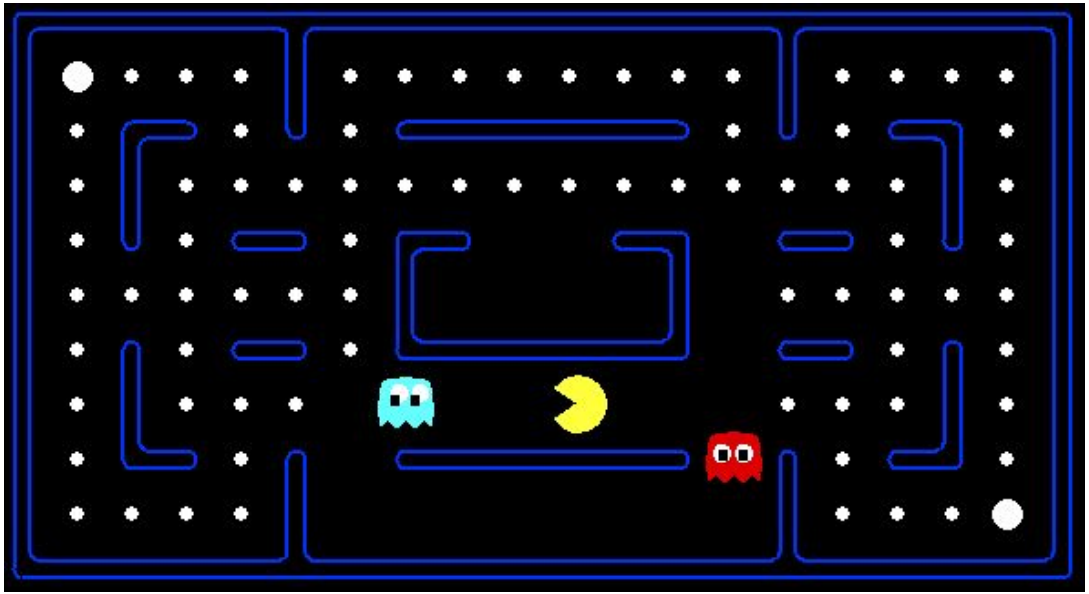
- **Chess:**
  - 1945-1960: Zuse, Wiener, Shannon, Turing, Newell & Simon, McCarthy.
  - 1960-1996: gradual improvements
  - 1997: Deep Blue defeats human champion Garry Kasparov
  - 2024: Stockfish rating 3631 (vs 2847 for Magnus Carlsen)

- **Go:**
  - 1968: Zobrist's program plays legal Go, barely (b>300!)
  - 1968-2005: various ad hoc approaches tried, novice level
  - 2005-2014: Monte Carlo tree search -> strong amateur
  - 2016-2017: AlphaGo defeats human world champions
  - 2022: Human exploits NN weakness to defeat top Go programs

- **Pacman**

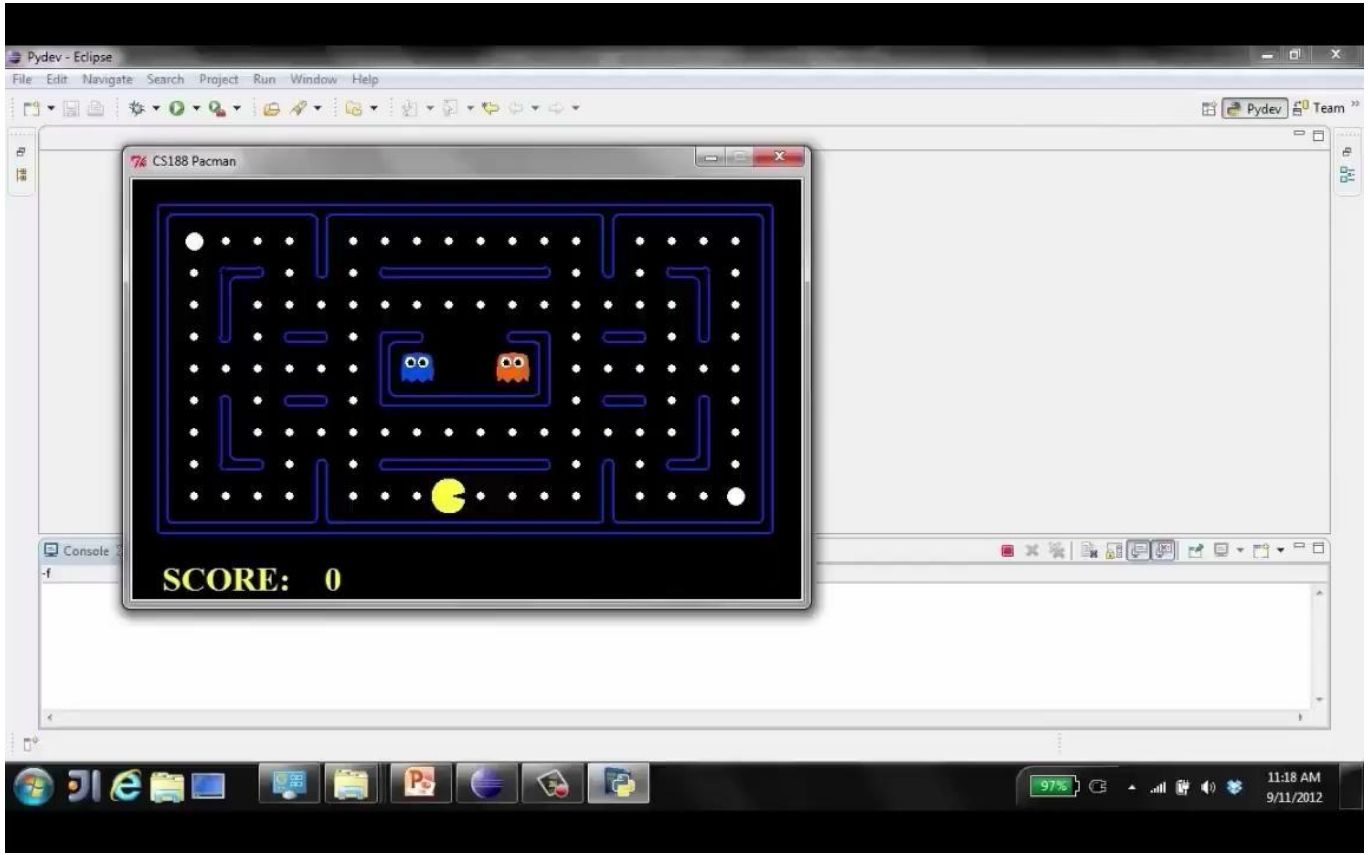


Behavior from Computation



[Demo: mystery pacman (L6D1)]

Pacman Demo


Pydev - Eclipse
File Edit Navigate Search Project Run Window Help
Pydev
Team
CS188 Pacman
SCORE: 0
Console
97%
11:18 AM
9/11/2012

Adversarial Games

Types of Games

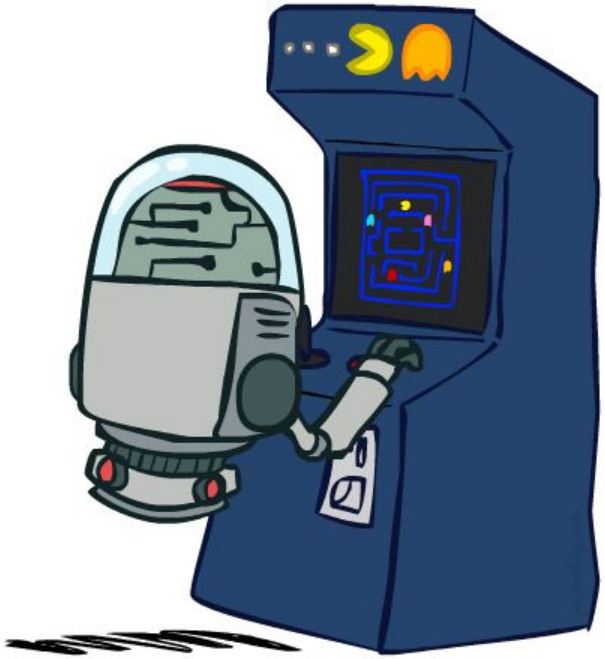
- Game = task environment with > 1 agent
- Axes:
  - Deterministic or stochastic?
  - Perfect information (fully observable)?
  - Two, three, or more players?
  - Teams or individuals?
  - Turn-taking or simultaneous?
  - Zero sum?

- Want algorithms for calculating a **strategy** (policy) which recommends a move from every possible state

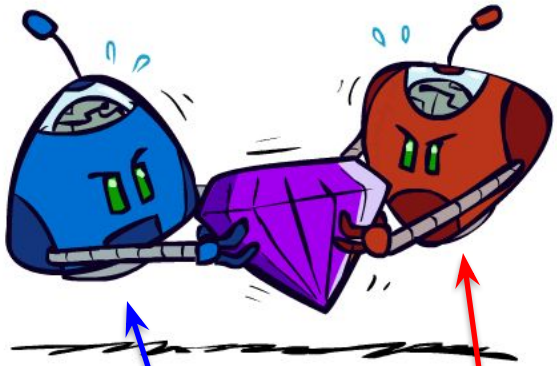
Deterministic Games

- 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 x A → S
  - Terminal test: S → {true, false}
  - Terminal utilities: S x P → R

- Similar to before, the solution for a player is a <u>policy</u>: S → A



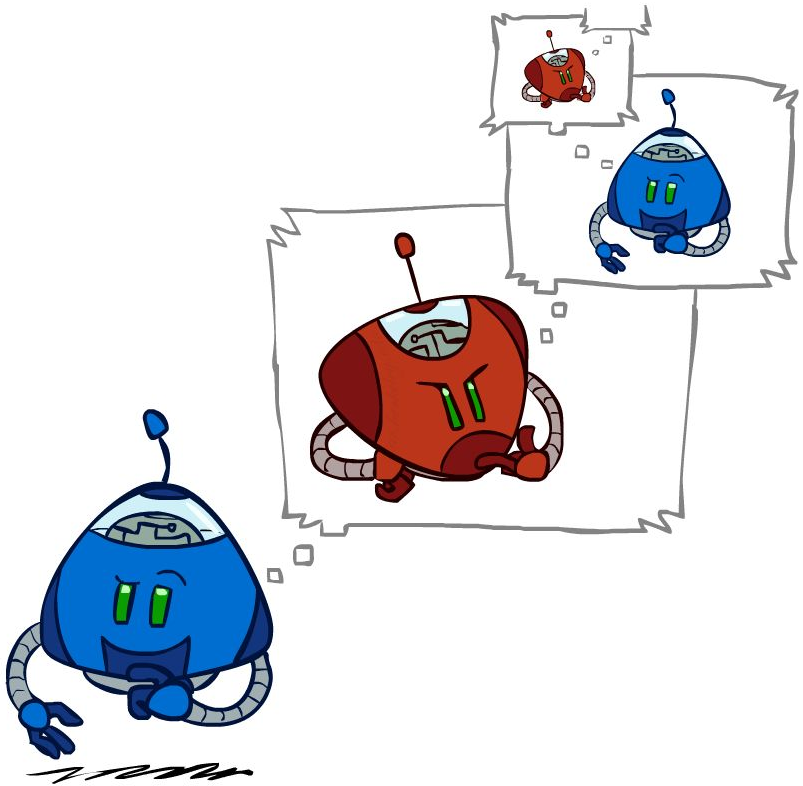
Zero-Sum Games



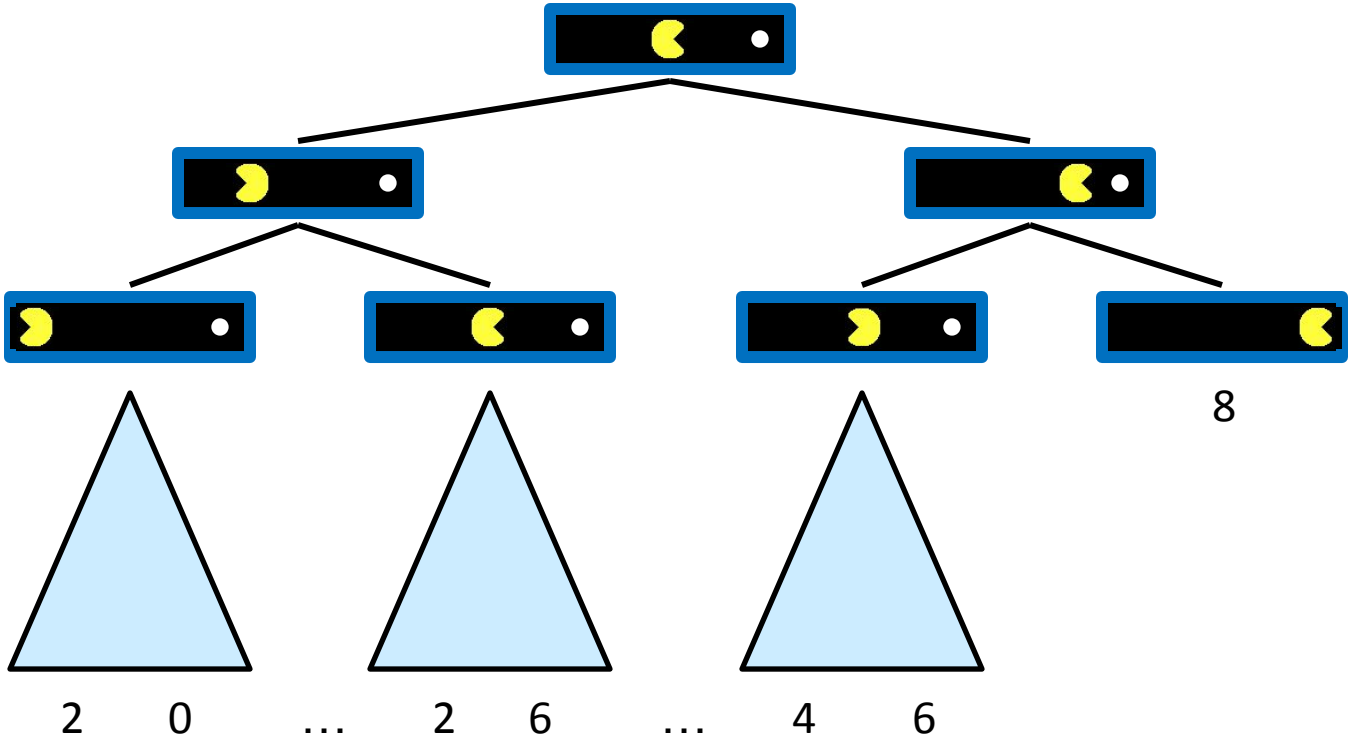
- Zero-Sum Games
  - Agents have ***opposite*** utilities
  - Pure competition:
    - One ***maximizes***, the other ***minimizes***

- General-Sum Games
  - Agents have ***independent*** utilities
  - Cooperation, indifference, competition, shifting alliances, and more are all possible
- Team Games
  - Common payoff for all team members

Adversarial Search



Recall: Single-Agent Trees

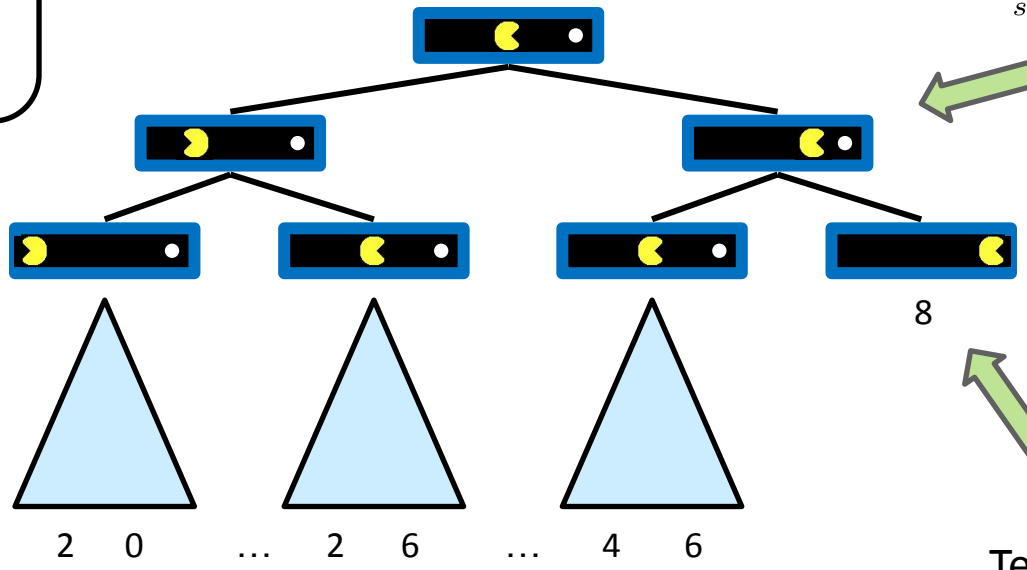


Value of a State

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

Non-Terminal States:

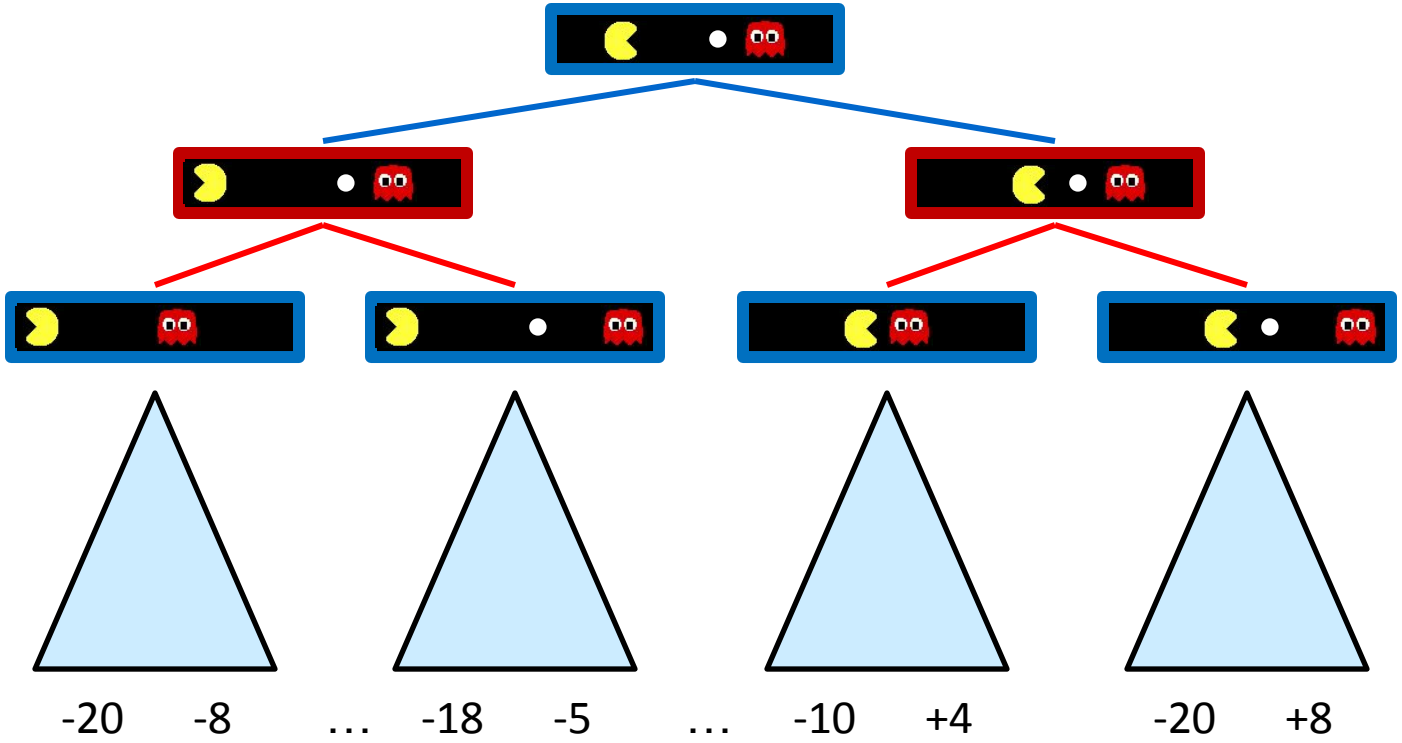
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial Game Trees



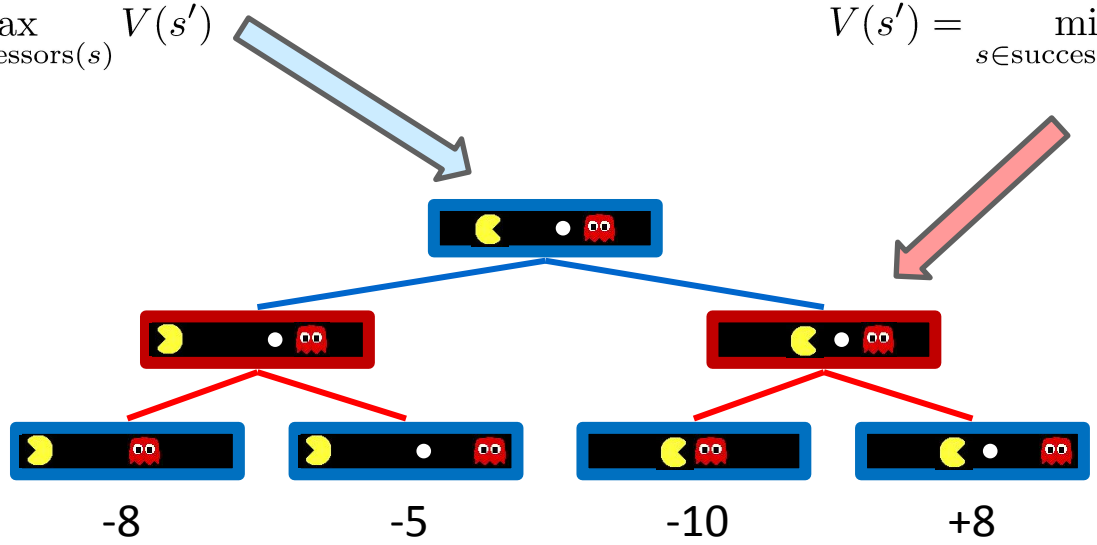
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

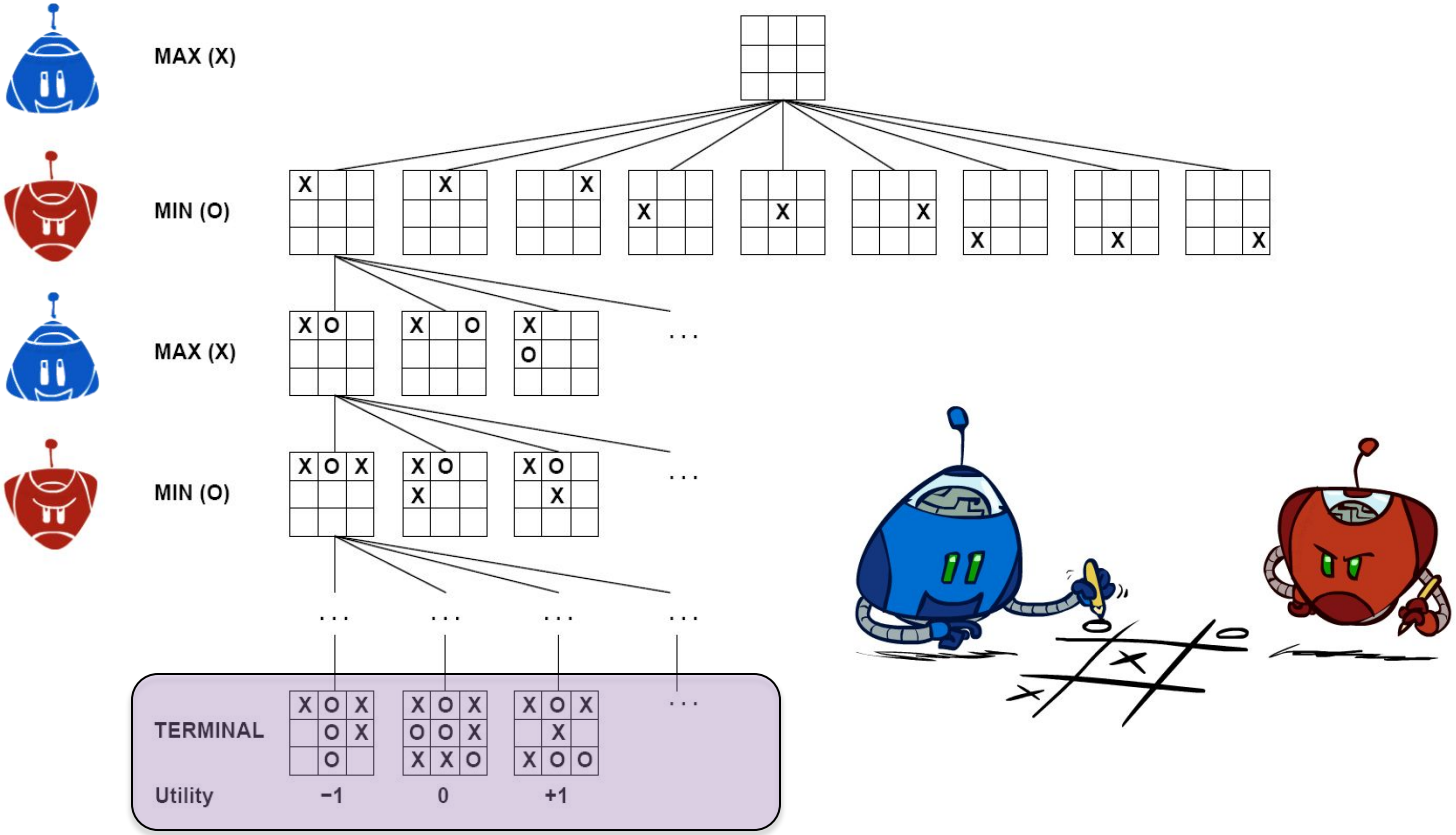
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

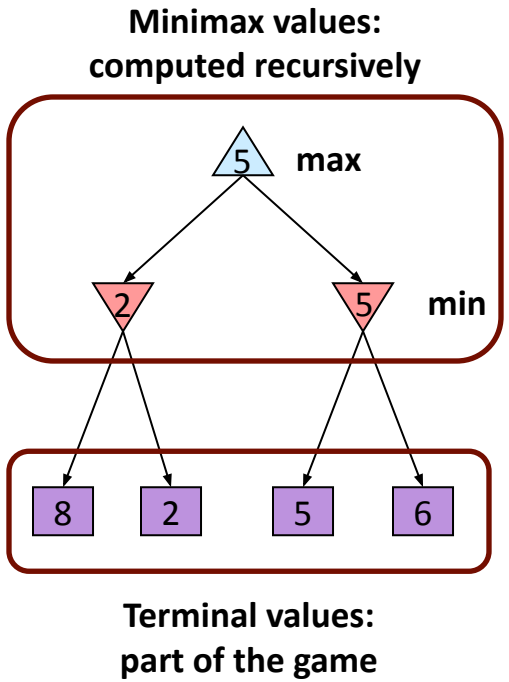
Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

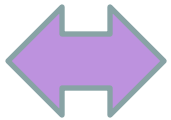
- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- 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 = -∞
    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
```

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

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

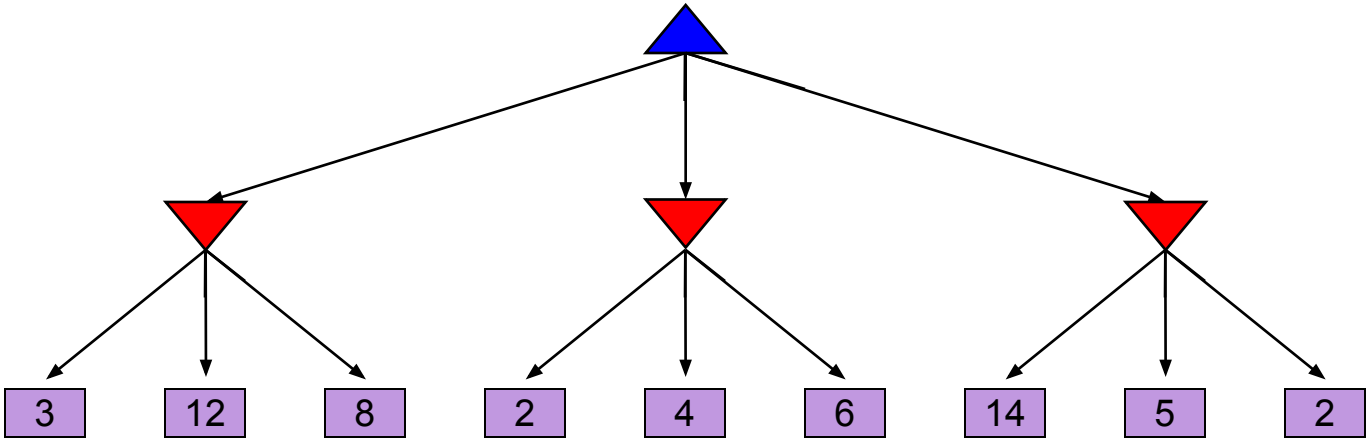
Minimax Implementation (Dispatch)

```
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)
```

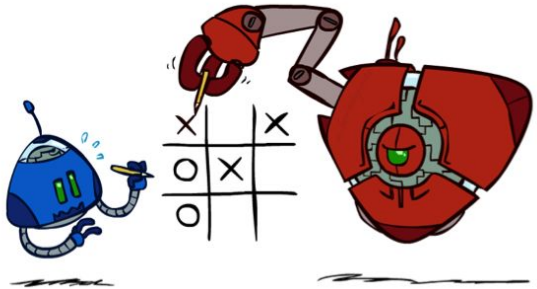
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

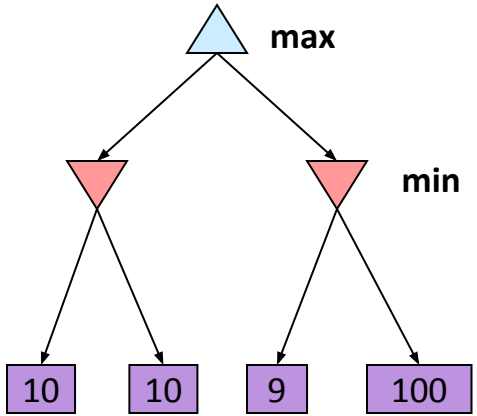
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

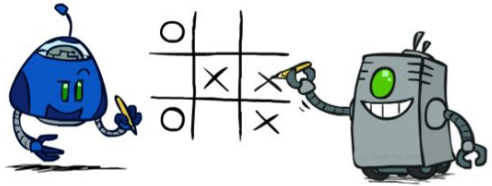
Minimax Example



Minimax Properties



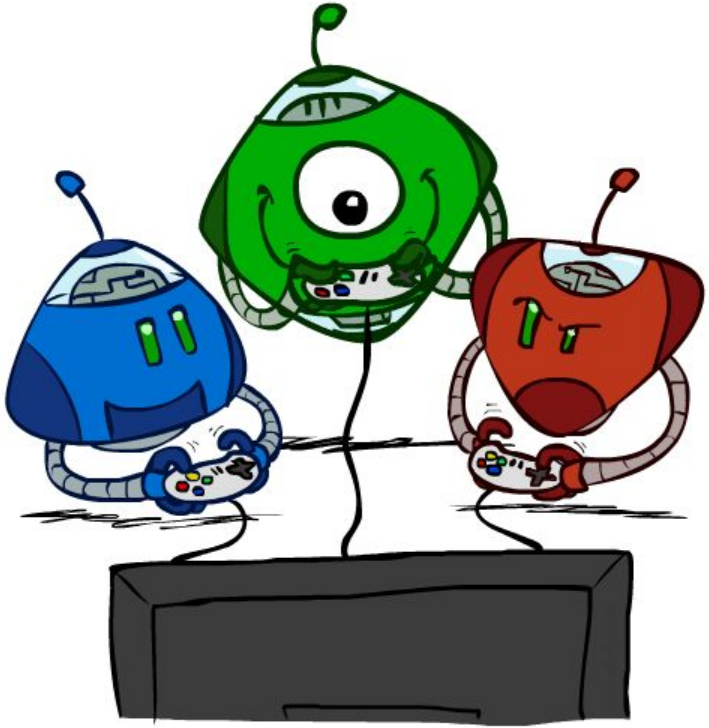




Optimal against a perfect player. Otherwise?

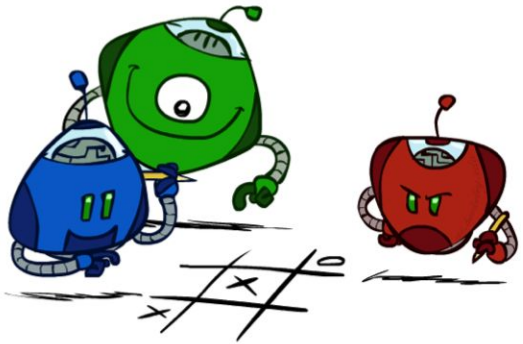
[Demo: min vs exp (L6D2, L6D3)]

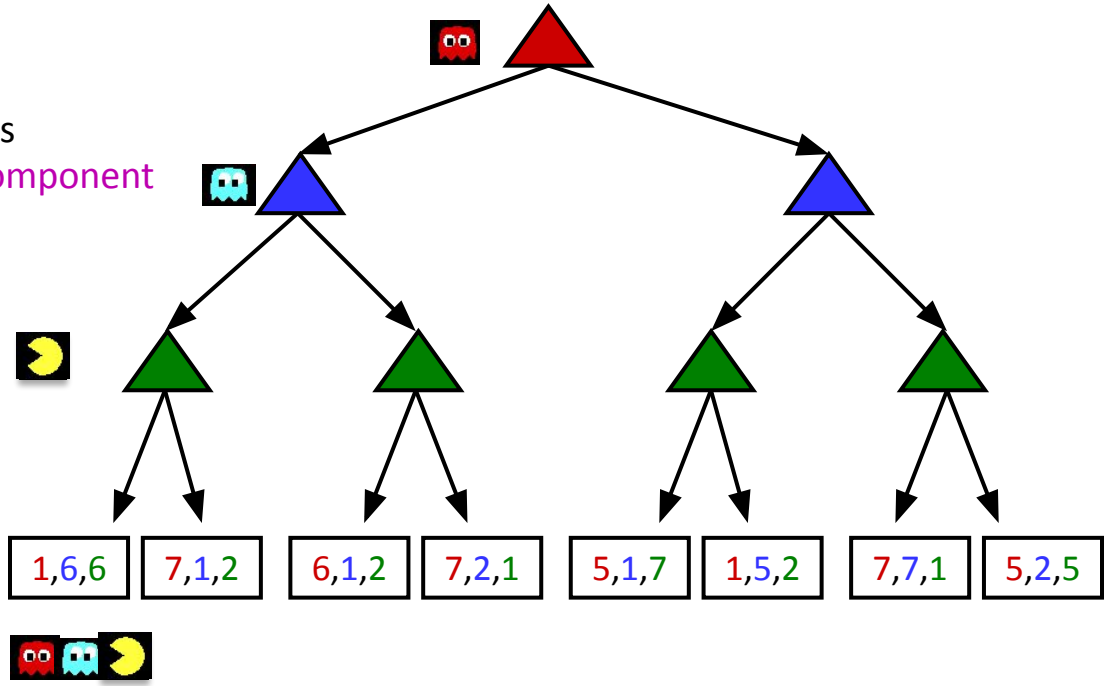
Handling games with 3+ players



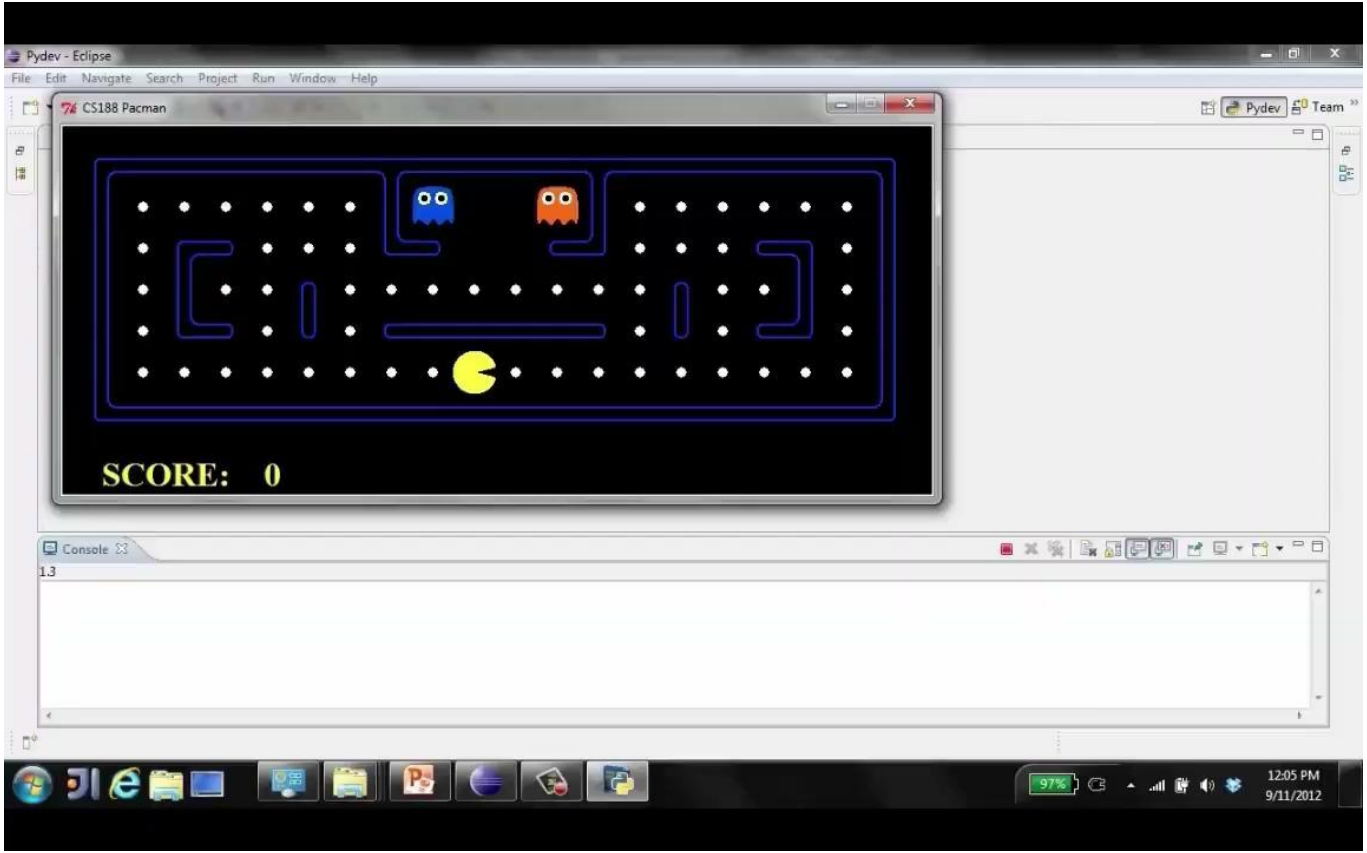
Multi-Agent Utilities

- 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…





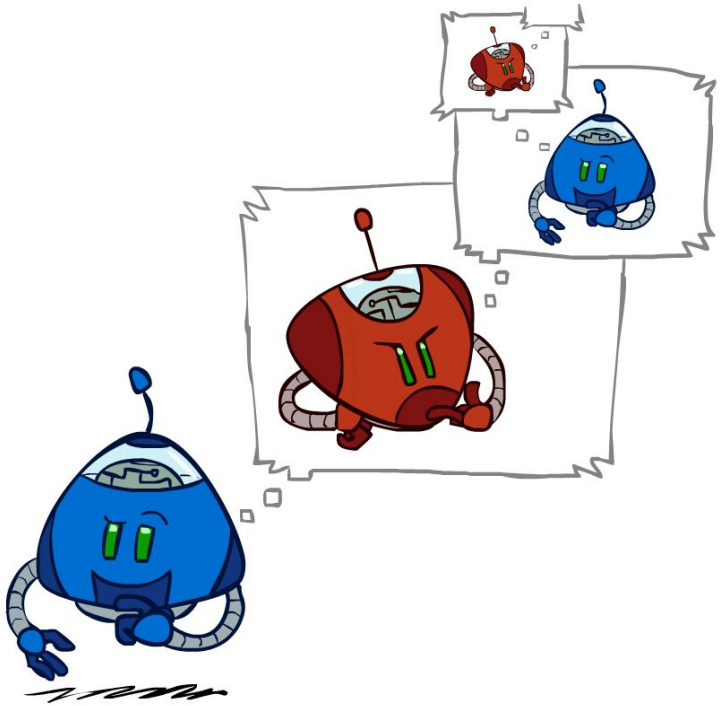
Emergent coordination in ghosts



Minimax Efficiency

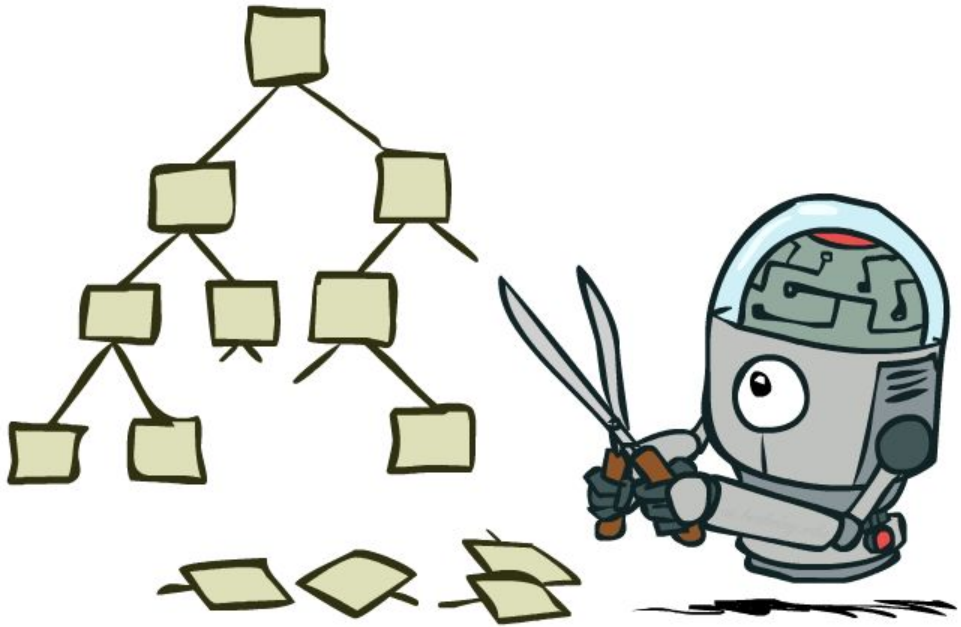
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: O(bm)

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

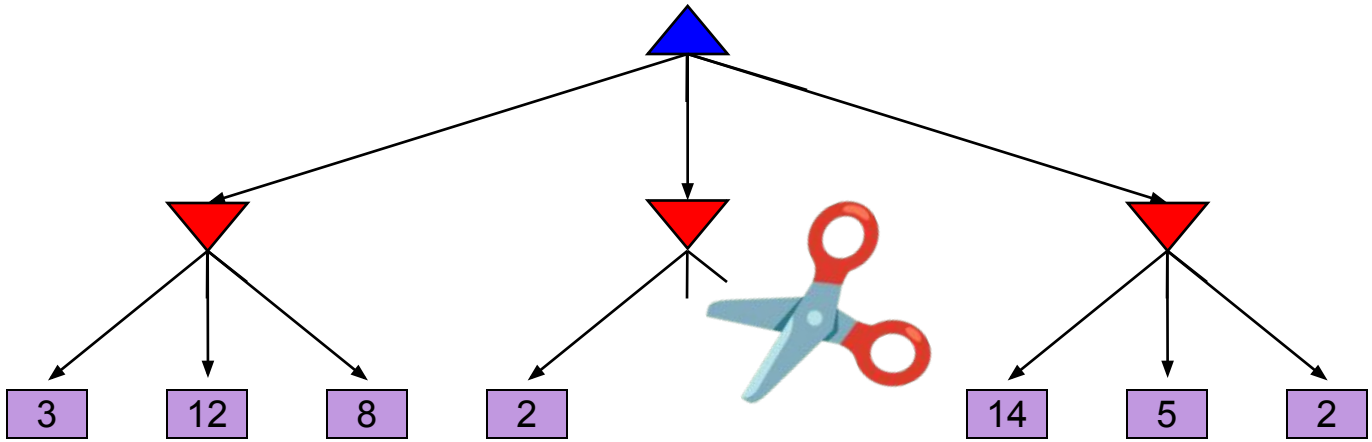


Resource Limits

Game Tree Pruning



Minimax Pruning



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

Alpha-Beta Pruning

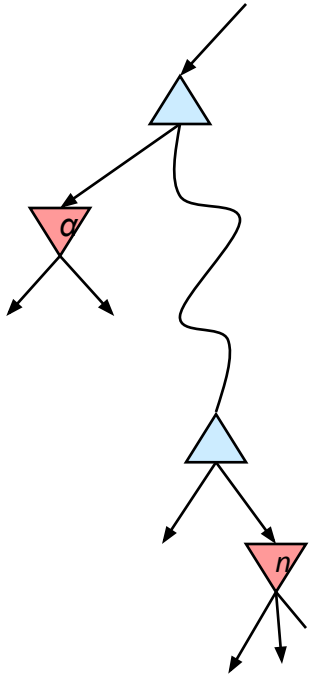
- General case (pruning children of MIN node)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let **α** be the best value that MAX can get so far at any choice point 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

MA
X

MIN

MA
X

MIN



Alpha-Beta Implementation

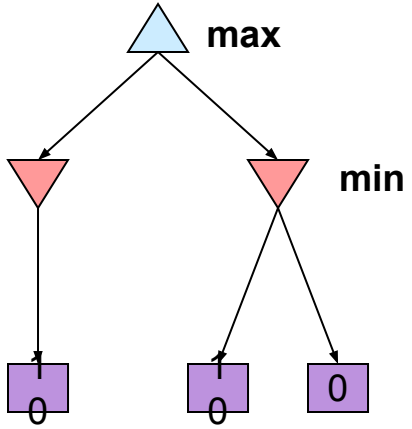
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

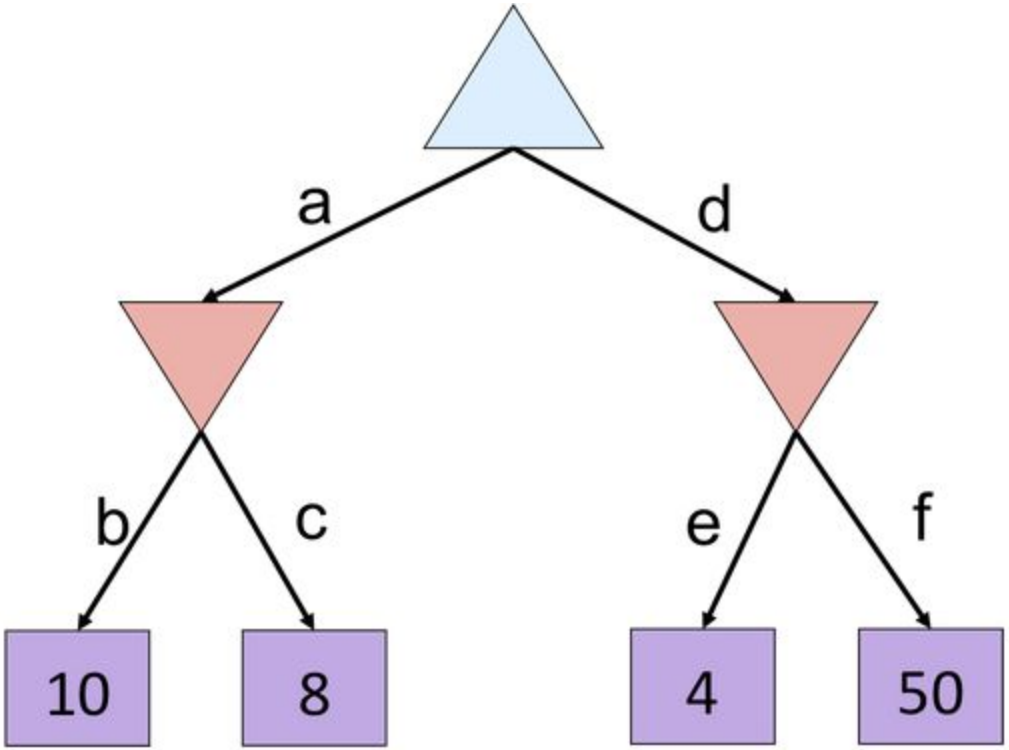
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Pruning Properties

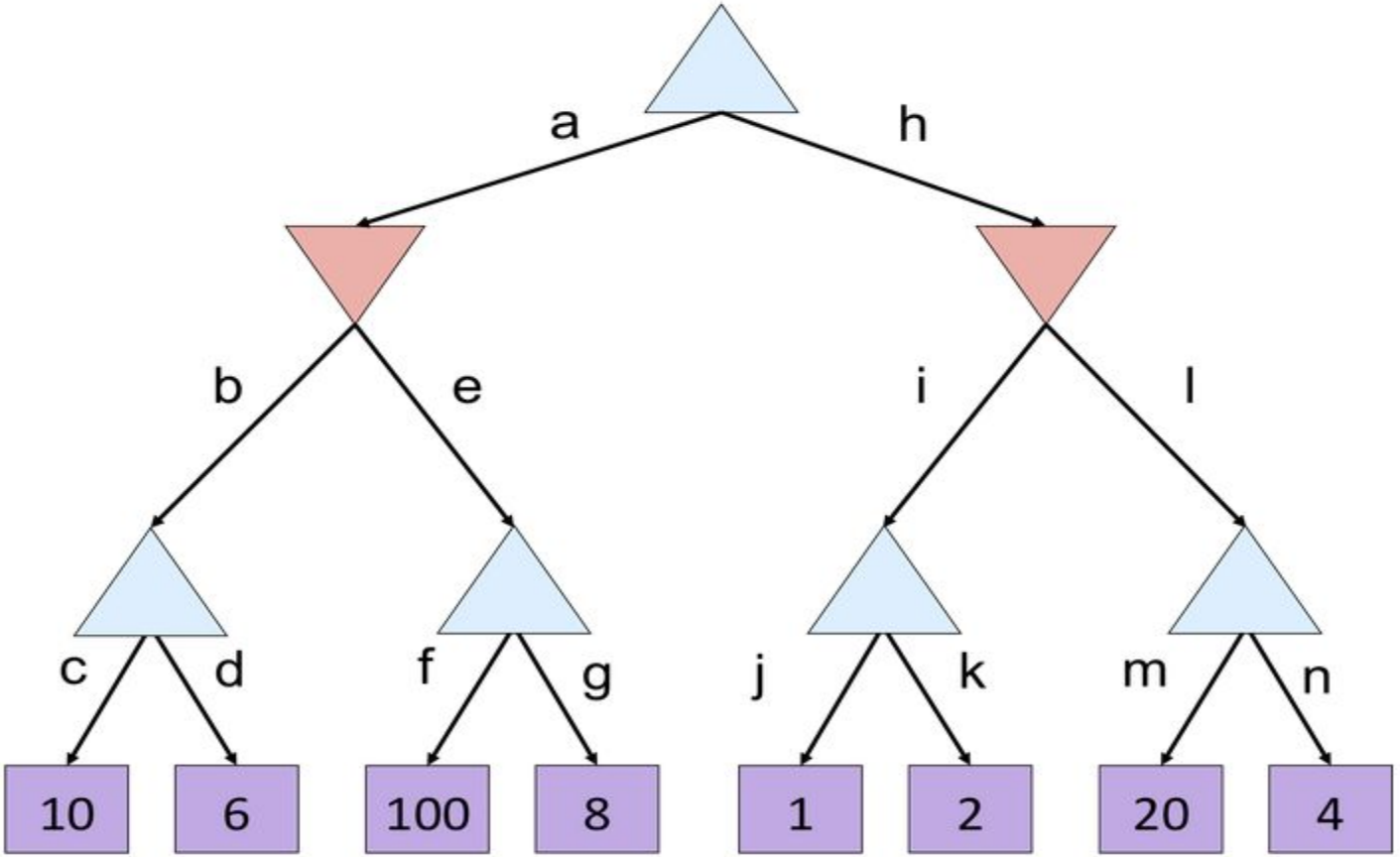
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - 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



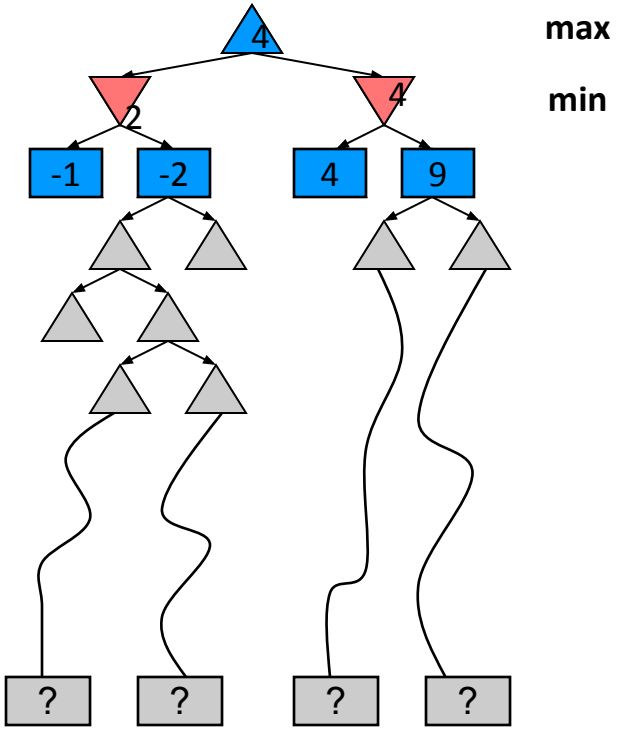
Alpha-Beta Quiz 2



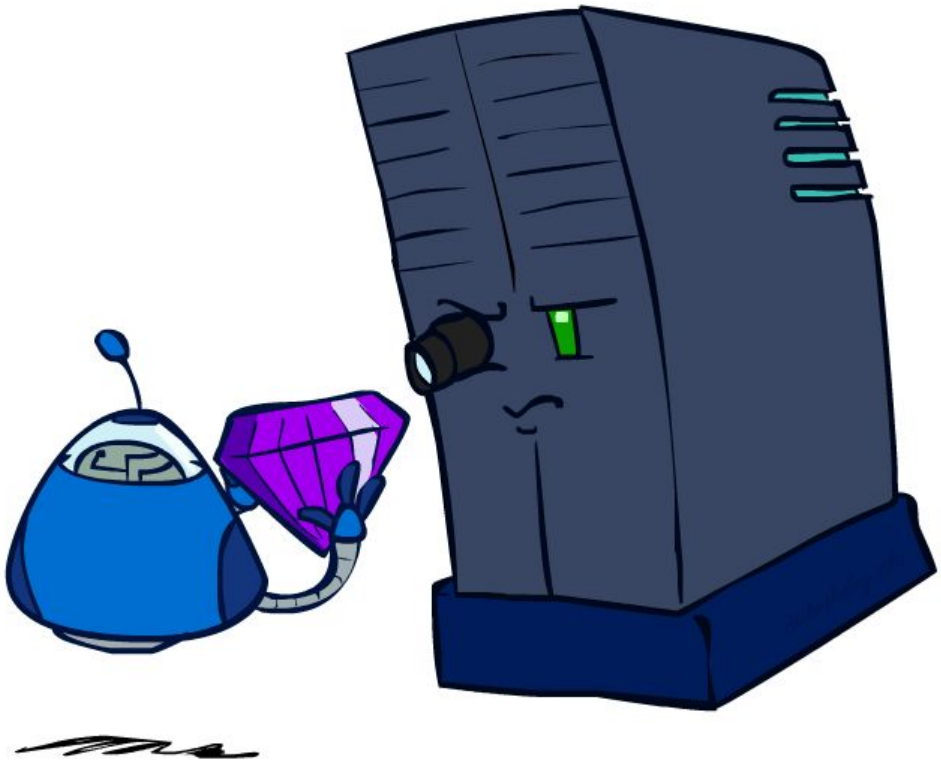
Resource Limits

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

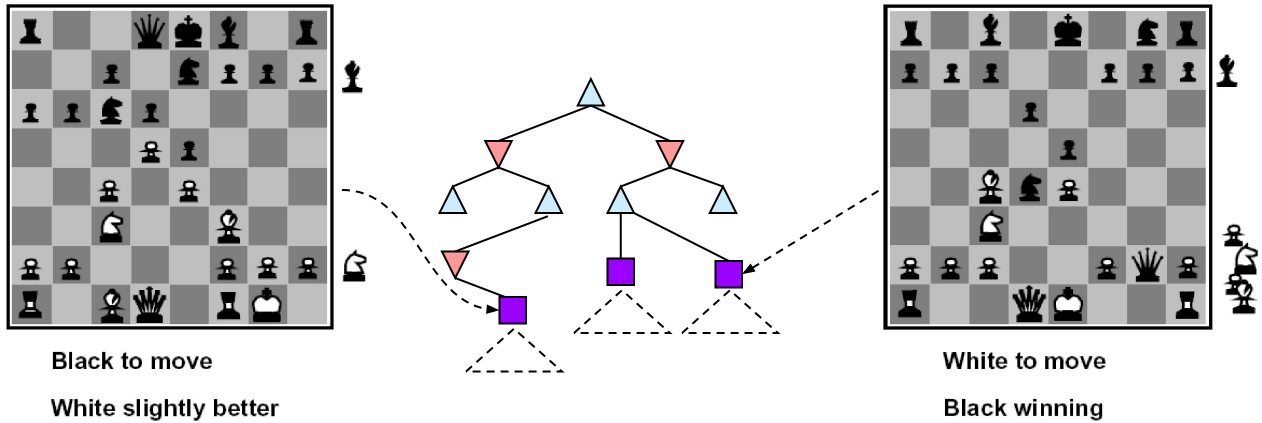


Evaluation Functions



Evaluation Functions

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

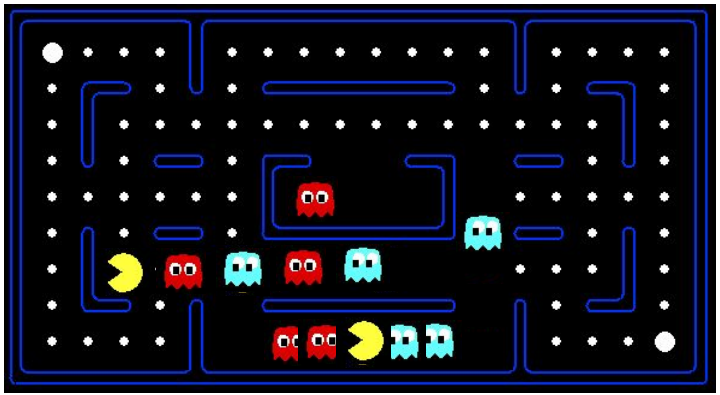


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

  - E.g. $f_1(s)$ = (num white queens – num black queens), etc.
- Or a more complex nonlinear function (e.g., NN) trained by self-play RL

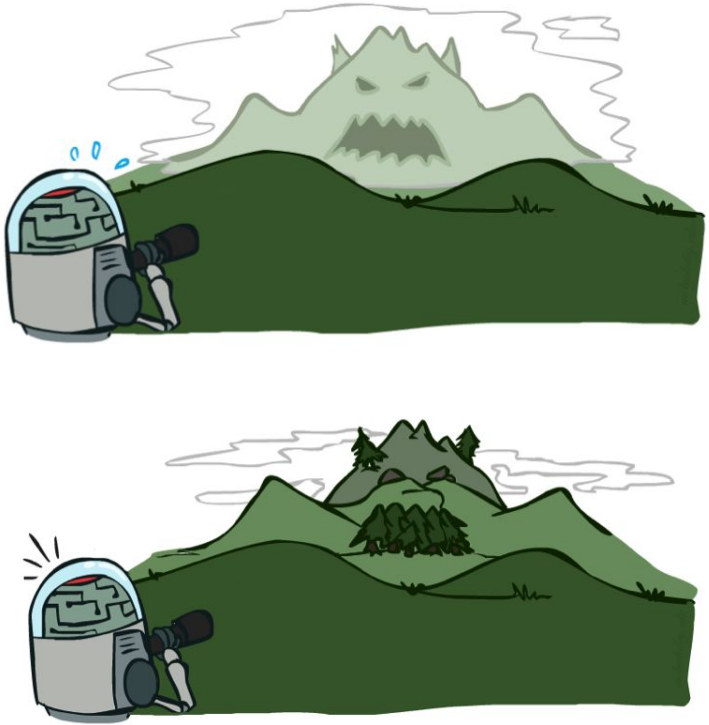
Evaluation for Pacman



[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate

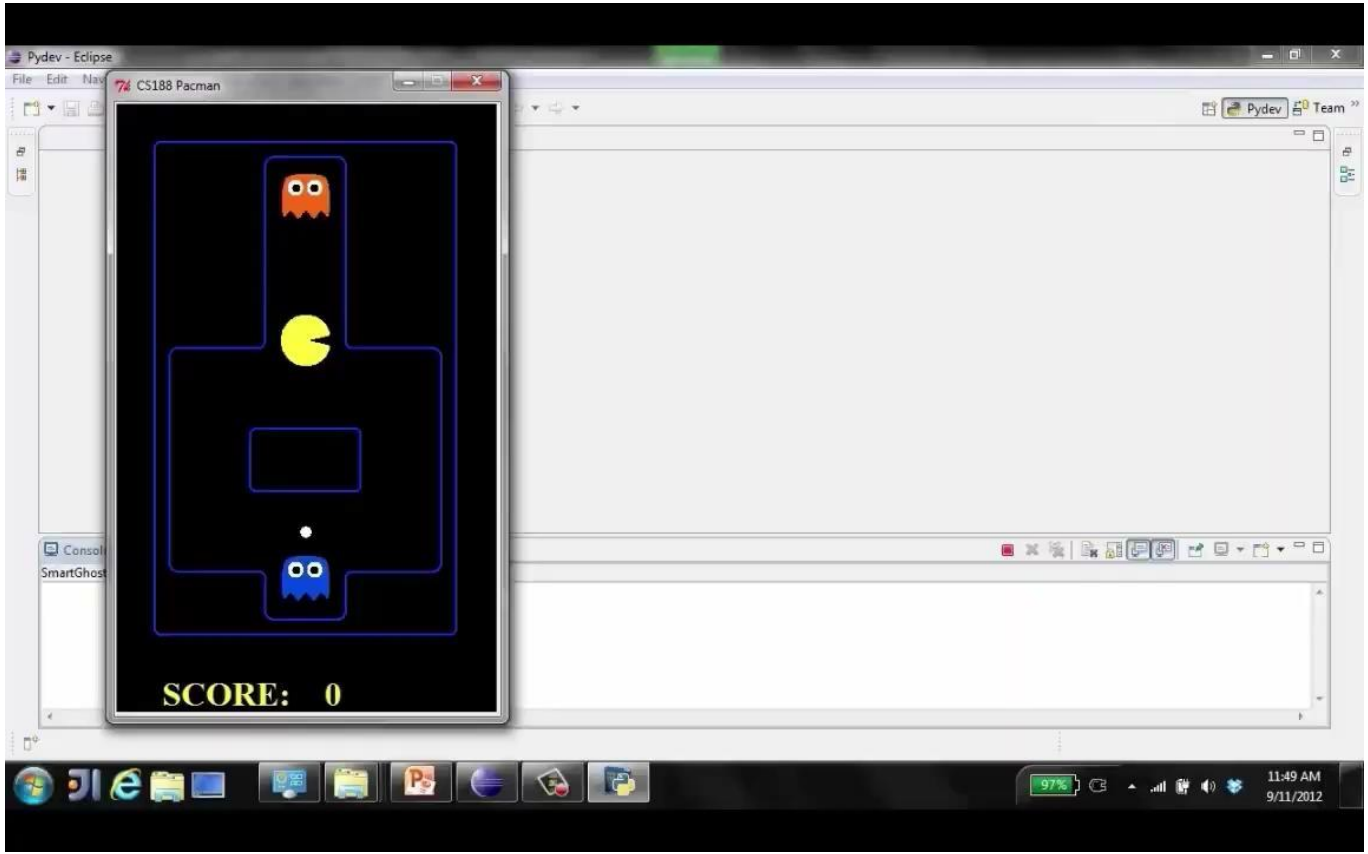
Depth Matters

- 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

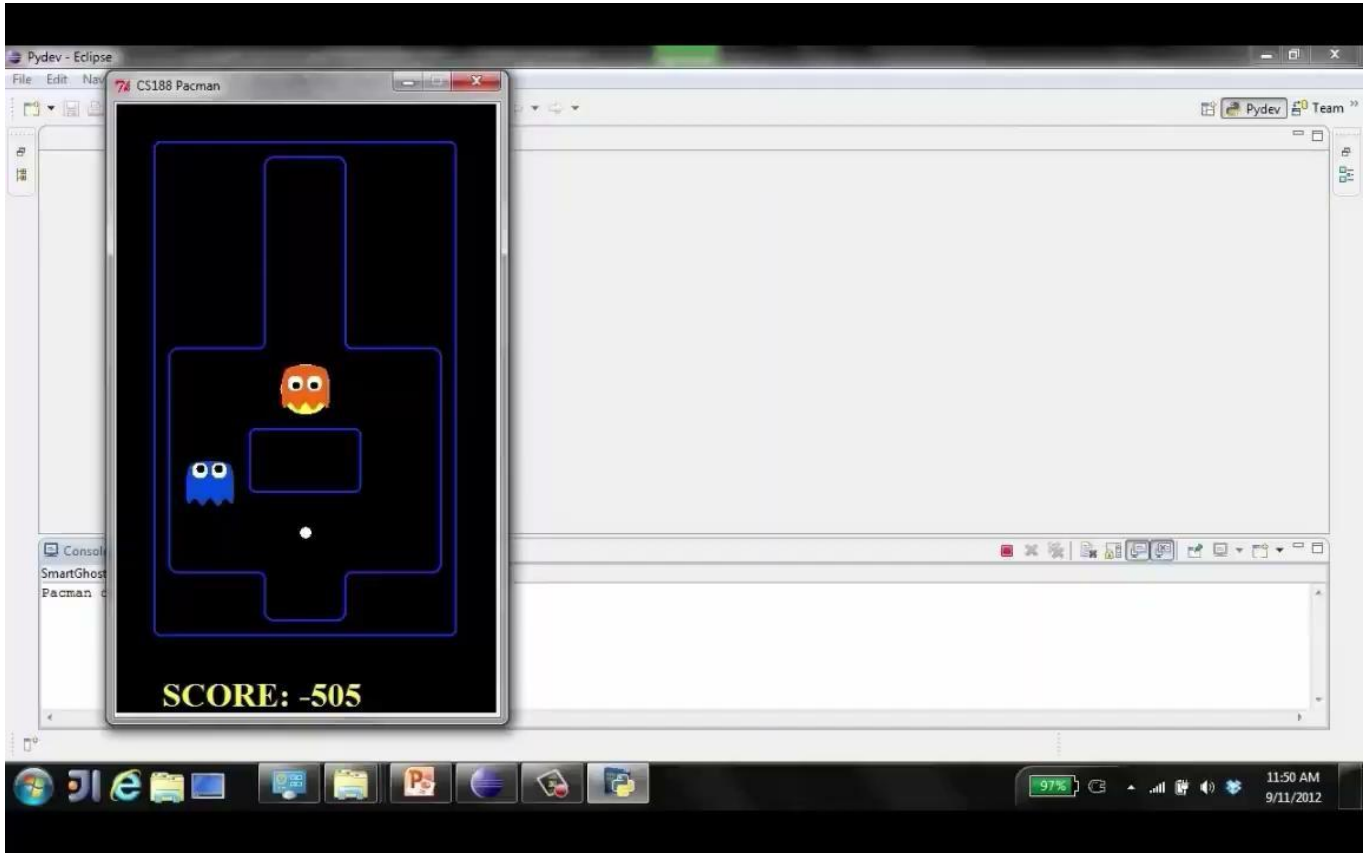


[Demo: depth limited (L6D4, L6D5)]

Video of Demo Limited Depth (2)



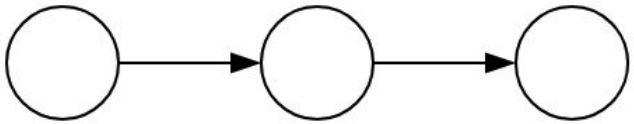
Video of Demo Limited Depth (10)

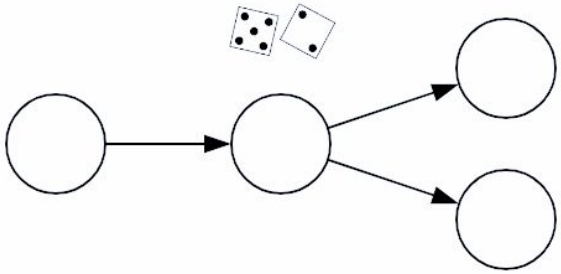


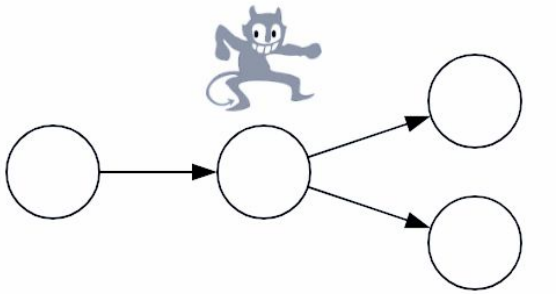
Summary

- Games are decision problems with $\geq$ 2 agents
  - Huge variety of issues and phenomena depending on details of interactions and payoffs
- For zero-sum games, optimal decisions defined by minimax
  - Simple extension to n-player "rotating" max with vectors of utilities
  - 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

State-based models







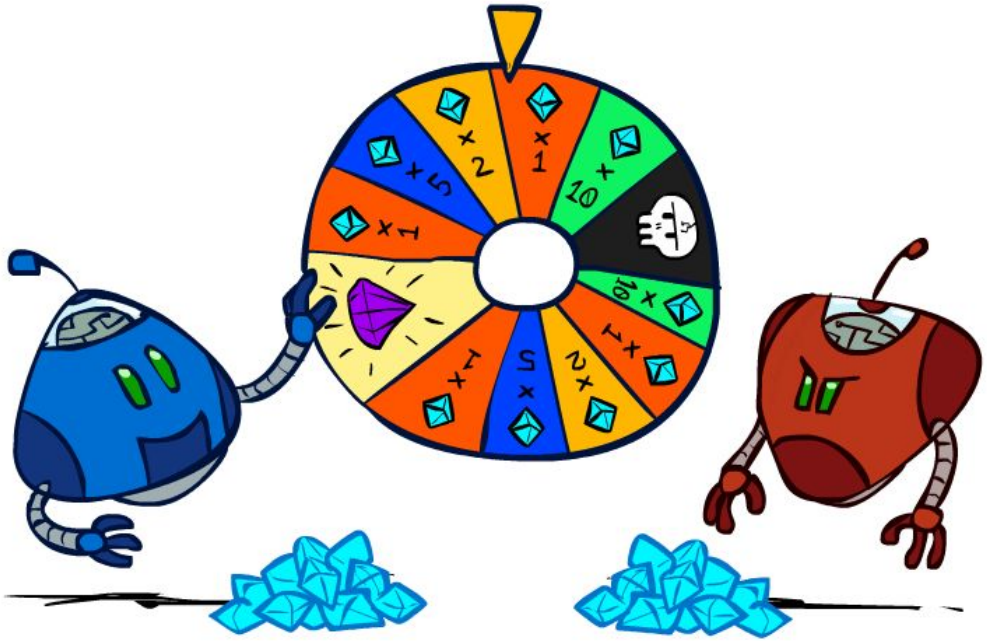
Search problems: when you have an environment with no uncertainty, ie. perfect information. But realistic settings are more complex

Markov decision processes (MDPs) handle situations with randomness, e.g. Blackjack

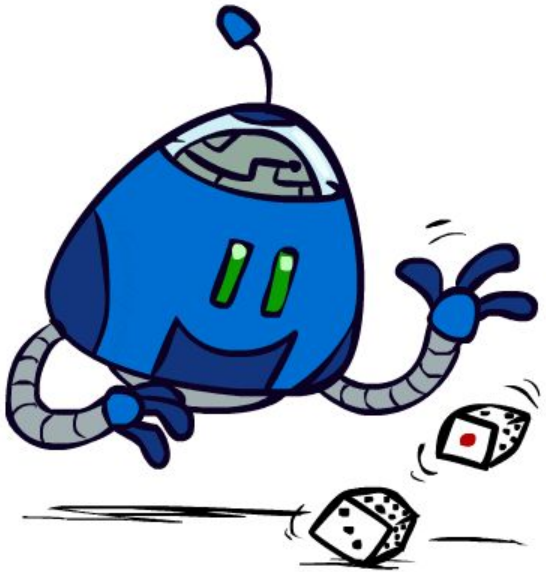
Game playing handles tasks where there is interaction with another agent. Adversarial games assume an opponent, e.g. Chess

Randomness + Game Playing

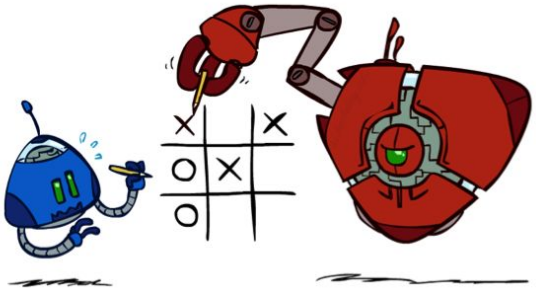
Expectimax, Monte Carlo Tree Search

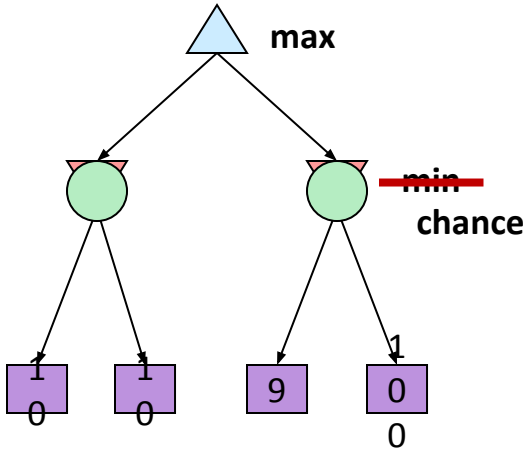


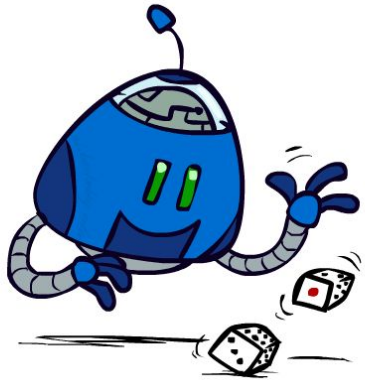
Uncertain Outcomes



Worst-Case vs. Average Case







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

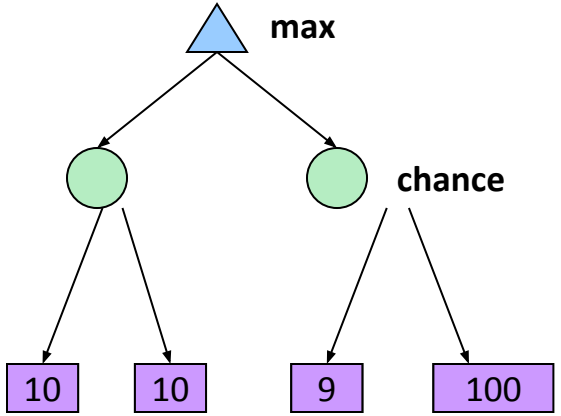
Expectimax Search

- We have already formalized the underlying uncertain-result problems as Markov Decision Processes

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Actions can fail: when moving a robot, wheels might slip
  - Unpredictable opponents: the ghosts respond randomly

- 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



[Demo: min vs exp (L7D1,2)]

Expectimax Pseudocode

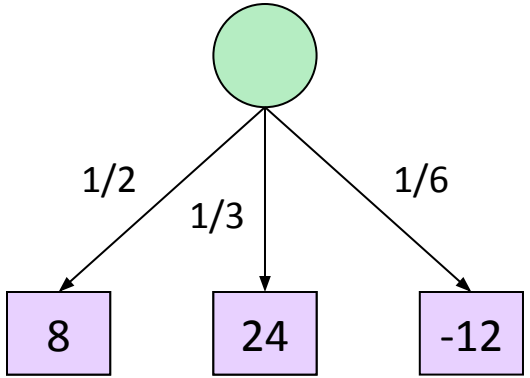
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

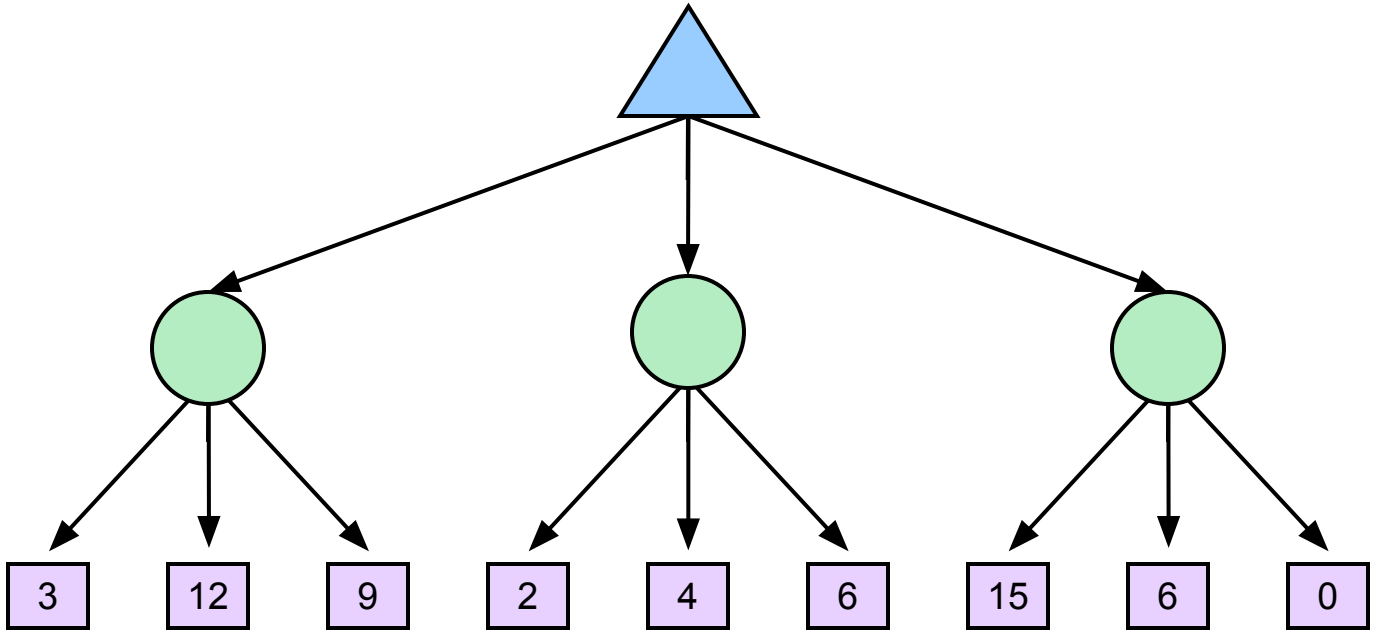
Expectimax Pseudocode

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

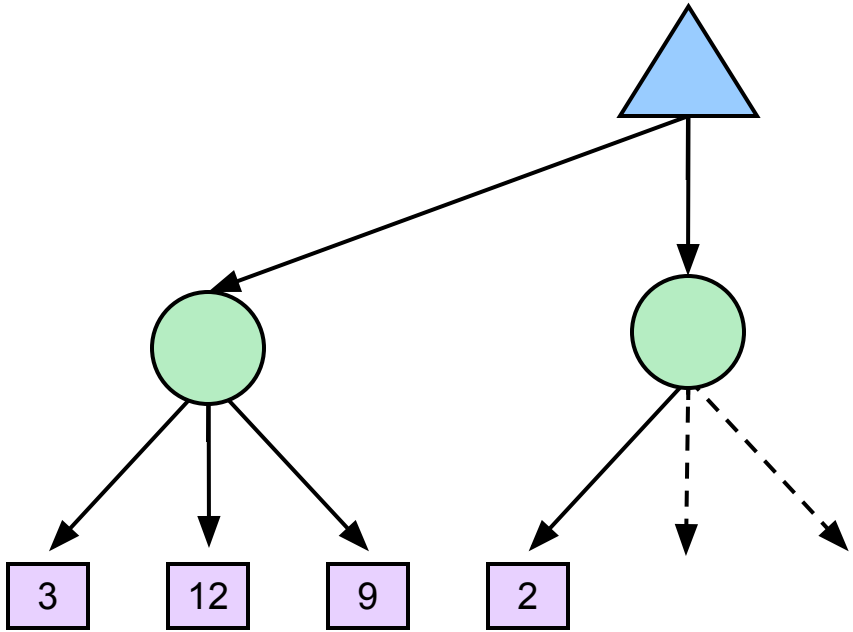


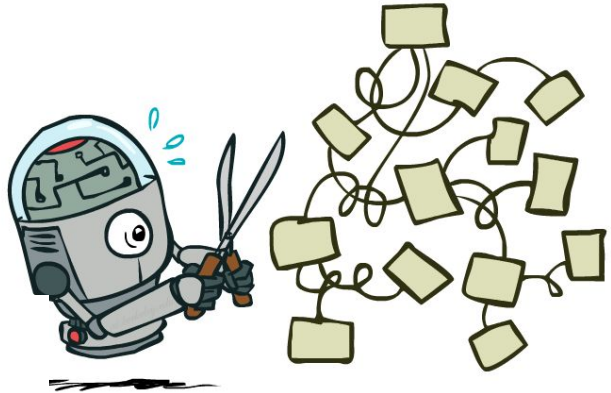
$$v = (1/2)\,(8) + (1/3)\,(24) + (1/6)\,(-12) = 10$$

Expectimax Example

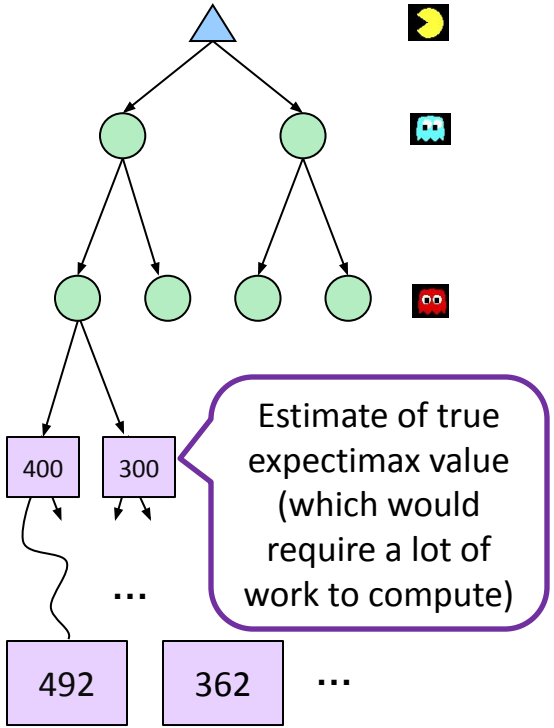


Expectimax Pruning?

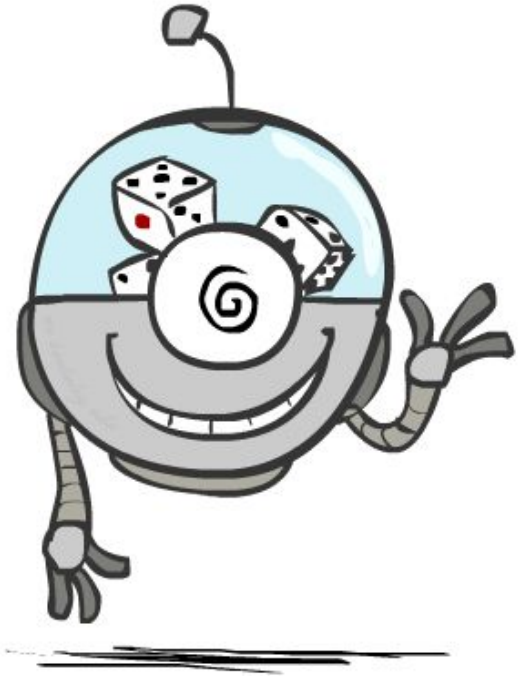




Depth-Limited Expectimax



Probabilities



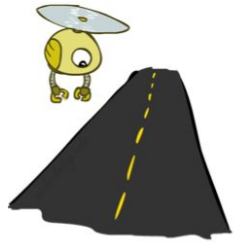
Reminder: Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

- Example: Traffic on freeway
  - Random variable: T = whether there's traffic
  - Outcomes: T in {none, light, heavy}
  - Distribution: P(T=none) = 0.25, P(T=light) = 0.50, P(T=heavy) = 0.25

- Some laws of probability (more later):
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one

- As we get more evidence, probabilities may change:
  - P(T=heavy) = 0.25, P(T=heavy | Hour=8am) = 0.60
  - We'll talk about methods for reasoning and updating probabilities later



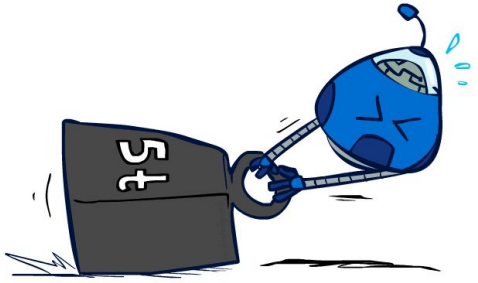
0.25

0.50

0.25

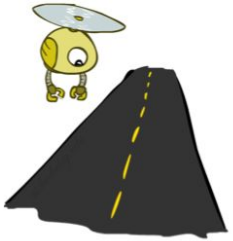
Reminder: Expectations

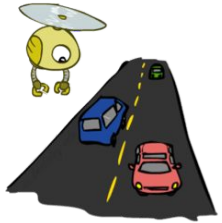
- The expected value of a function of a random variable is th average, weighted by the probability distribution over outcomes



- Example: How long to get to the airport?

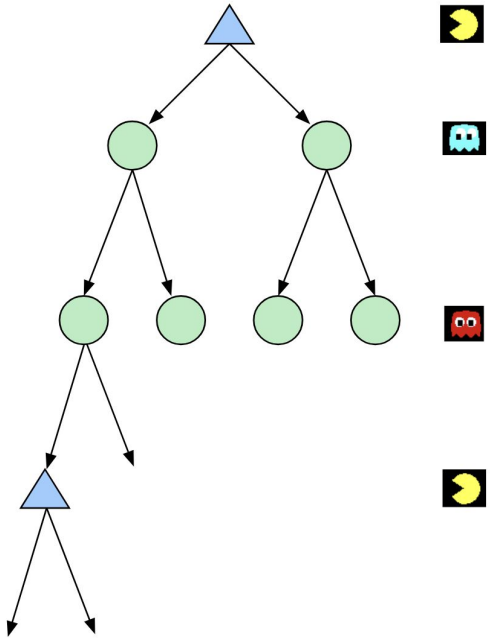
Time:	20 min		30 min		60 min		
	x	+	x	+	x		35 min
Probability:	0.25		0.50		0.25		





What Probabilities to Use?

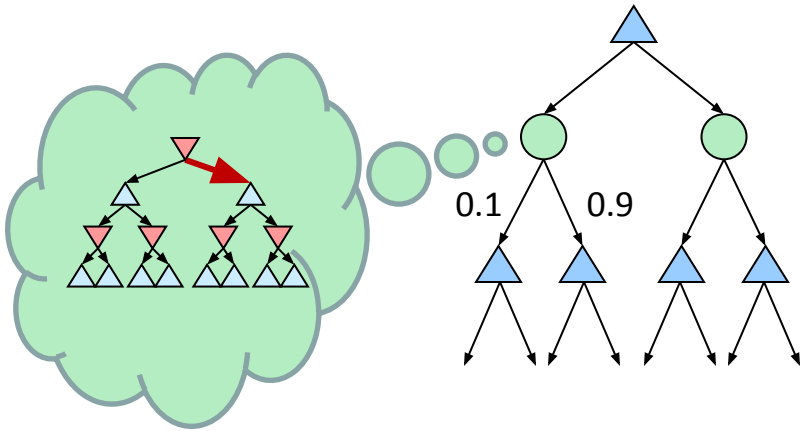
- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our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

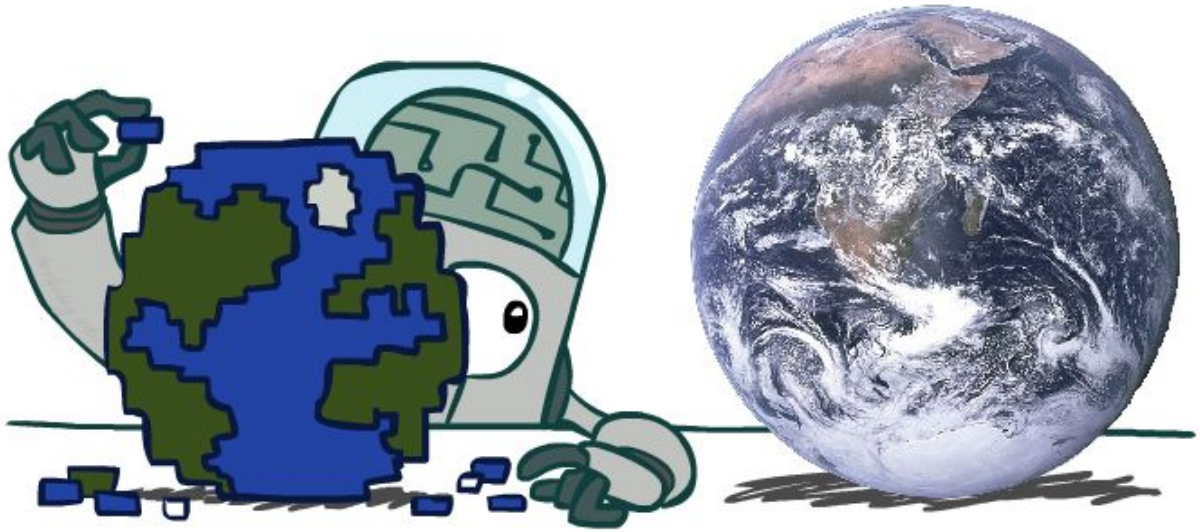
Quiz: Informed Probabilities

- 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

Modeling Assumptions



The Dangers of Optimism and Pessimism

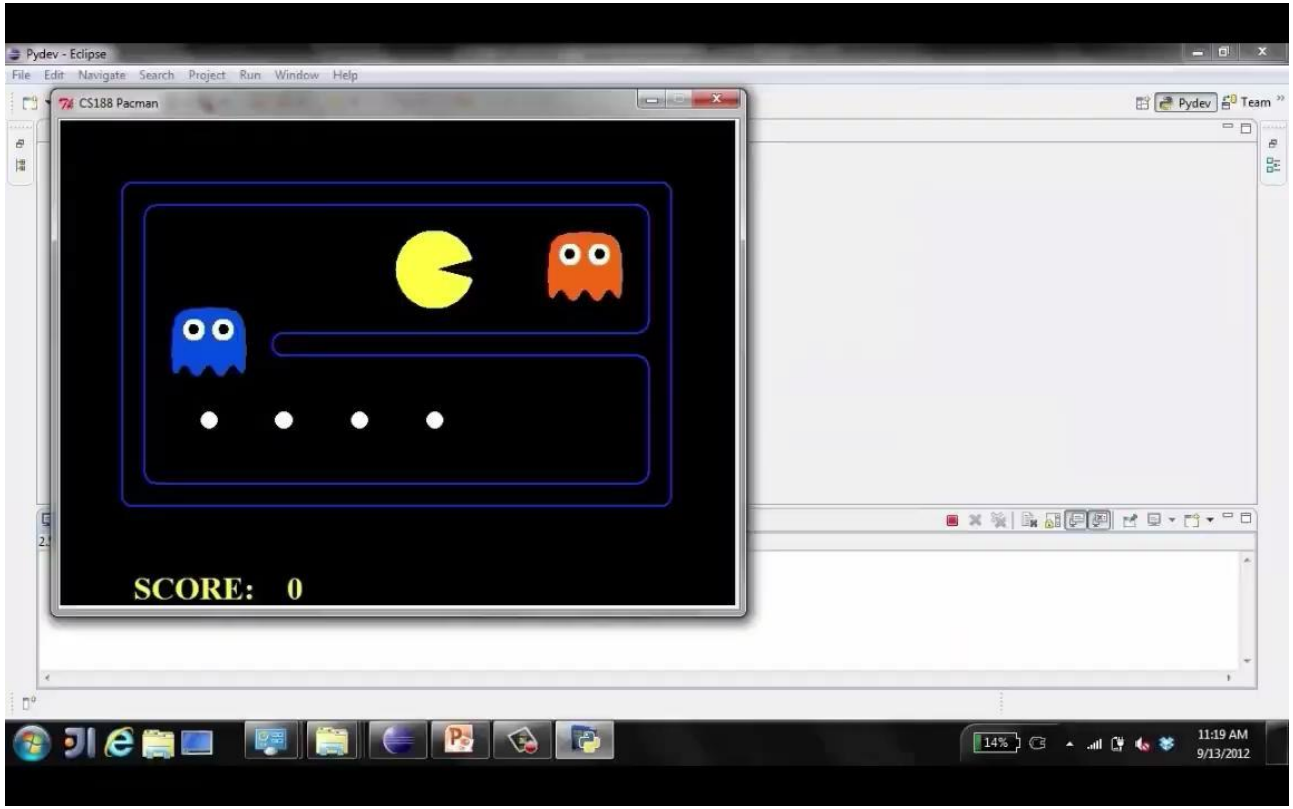
Dangerous Optimism

Assuming chance when the world is adversarial

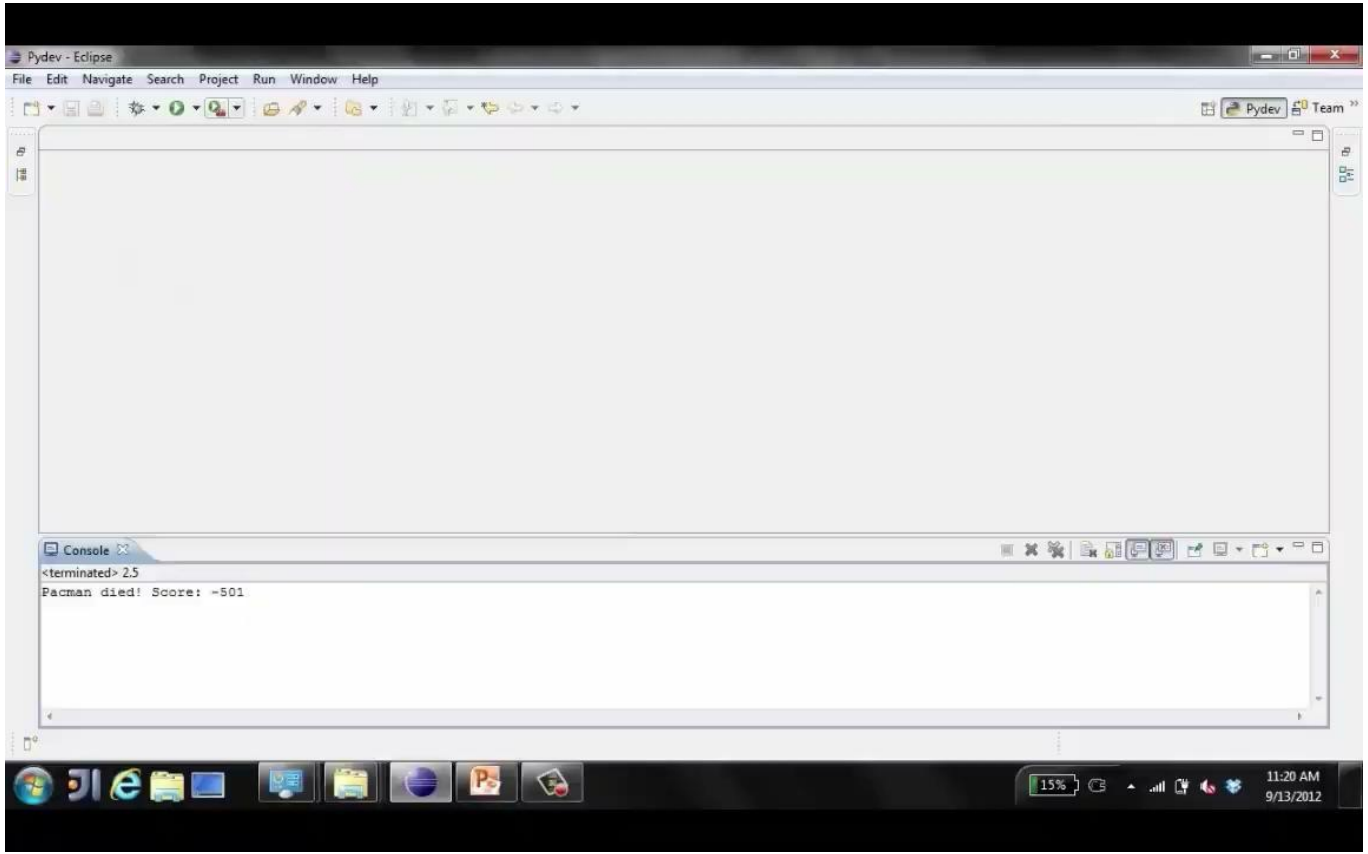
Dangerous Pessimism

Assuming the worst case when it's not likely

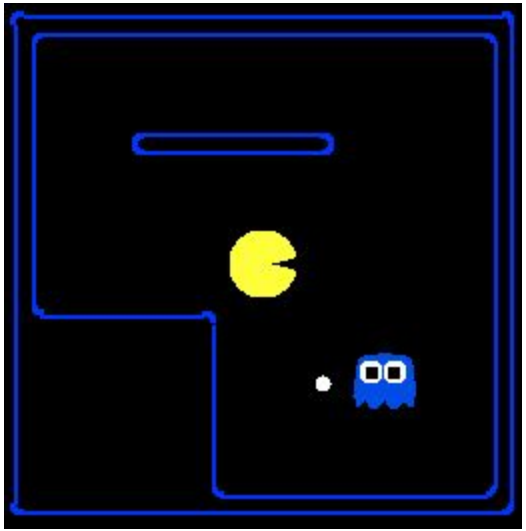
Video of Demo Minimax vs Expectimax (Min)



Video of Demo Minimax vs Expectimax (Exp)



Assumptions vs. Reality



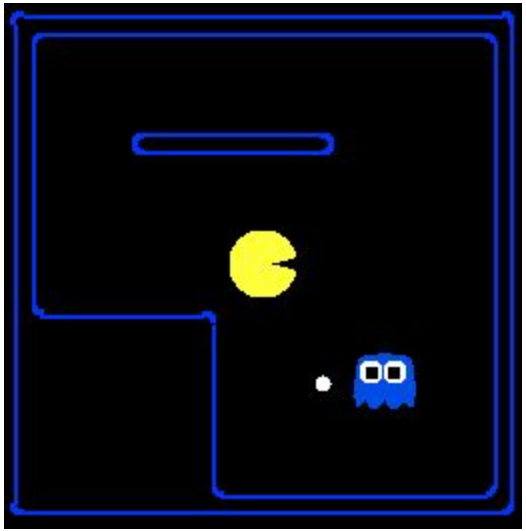
	Adversarial Ghost	Random Ghost
Minimax Pacman		
Expectimax Pacman		

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

[Demos: world assumptions

Assumptions vs. Reality



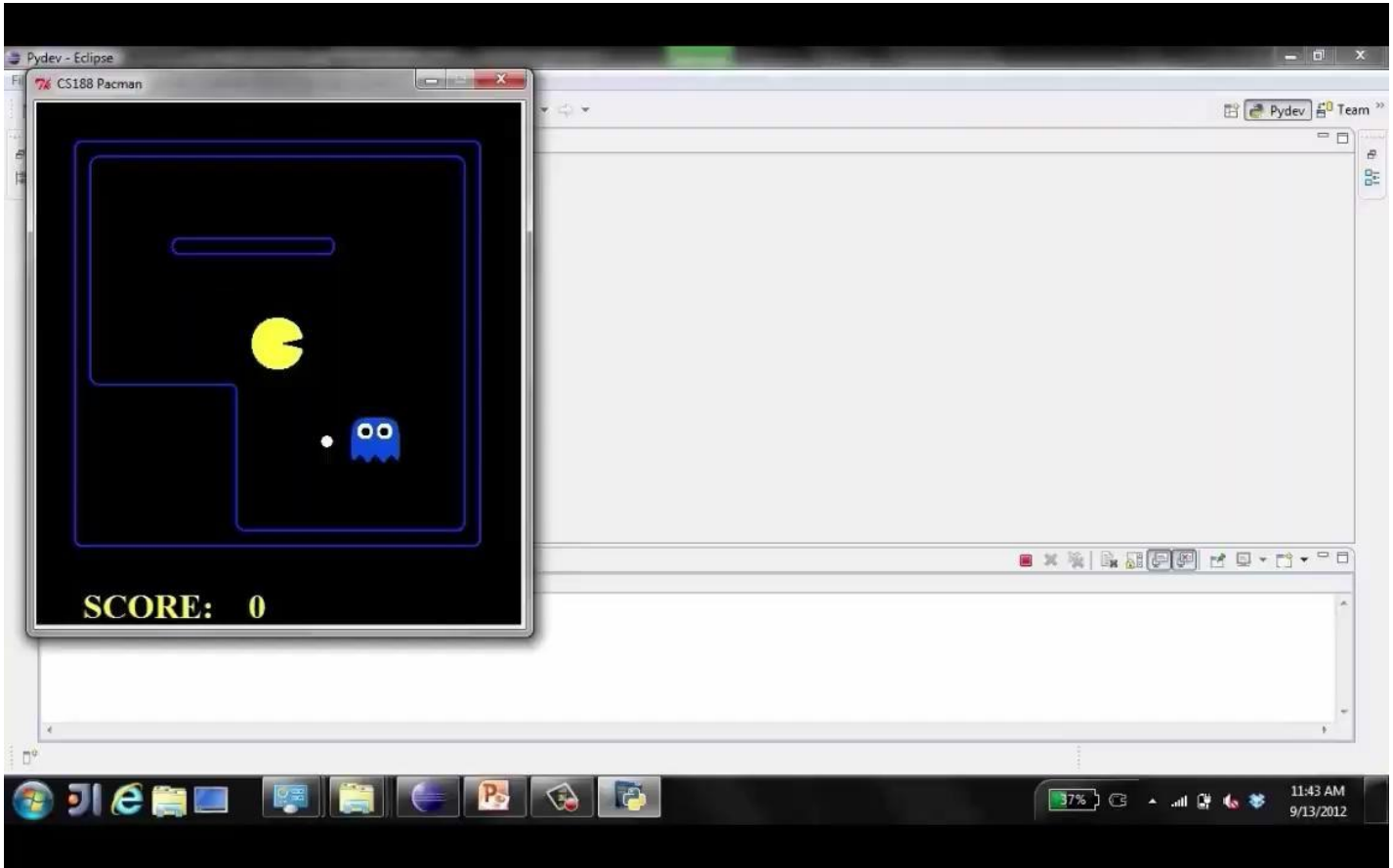
	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

Results from playing 5 games

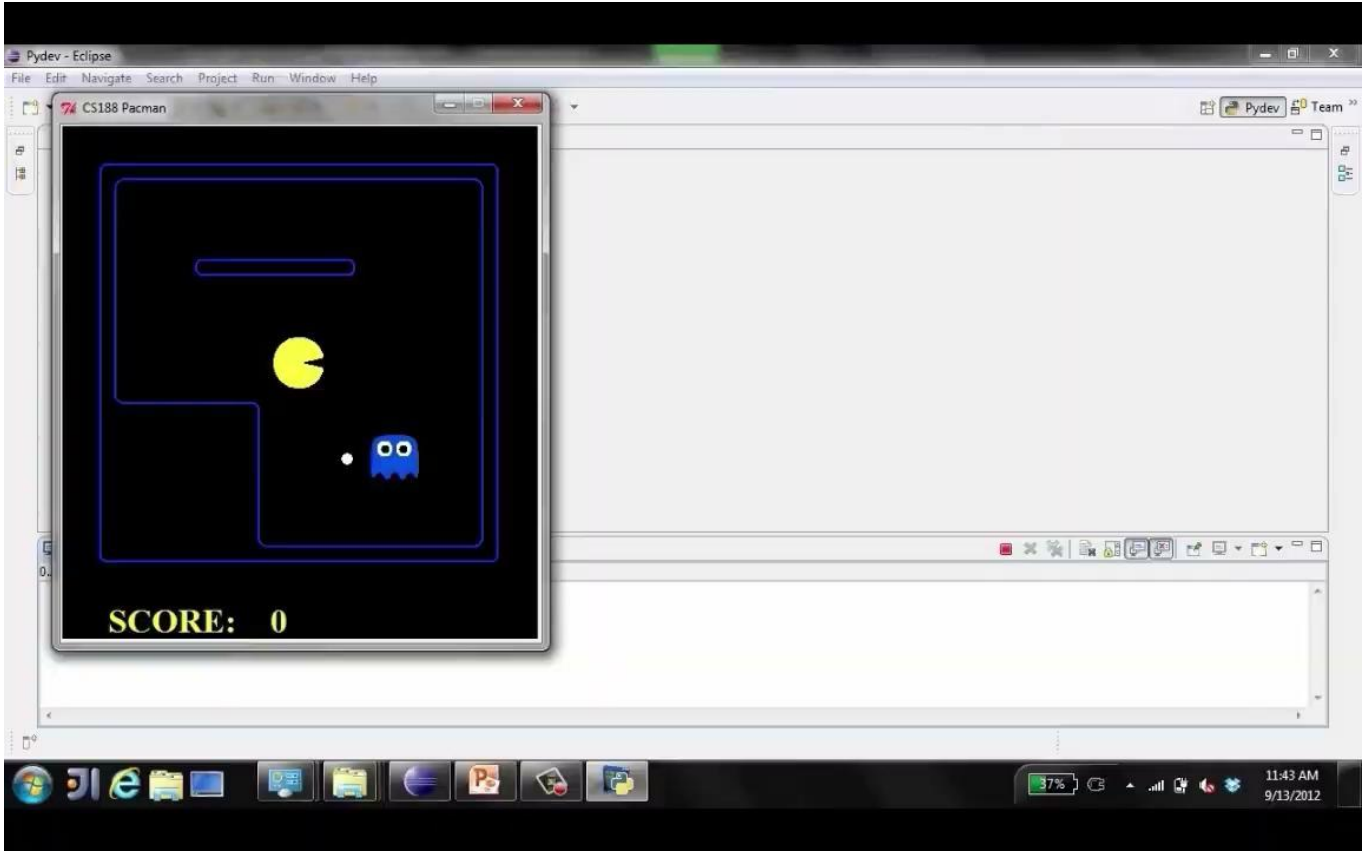
Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

[Demos: world assumptions

Video of Demo World Assumptions
Random Ghost – Expectimax Pacman

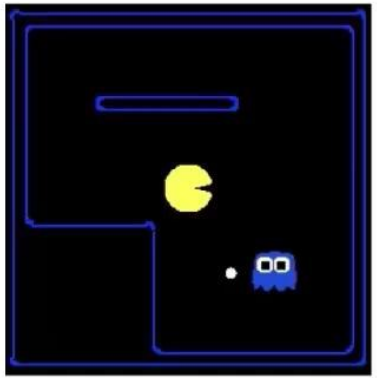


Video of Demo World Assumptions
Adversarial Ghost – Minimax Pacman



Video of Demo World Assumptions
Adversarial Ghost – Expectimax Pacman

Assumptions vs. Reality



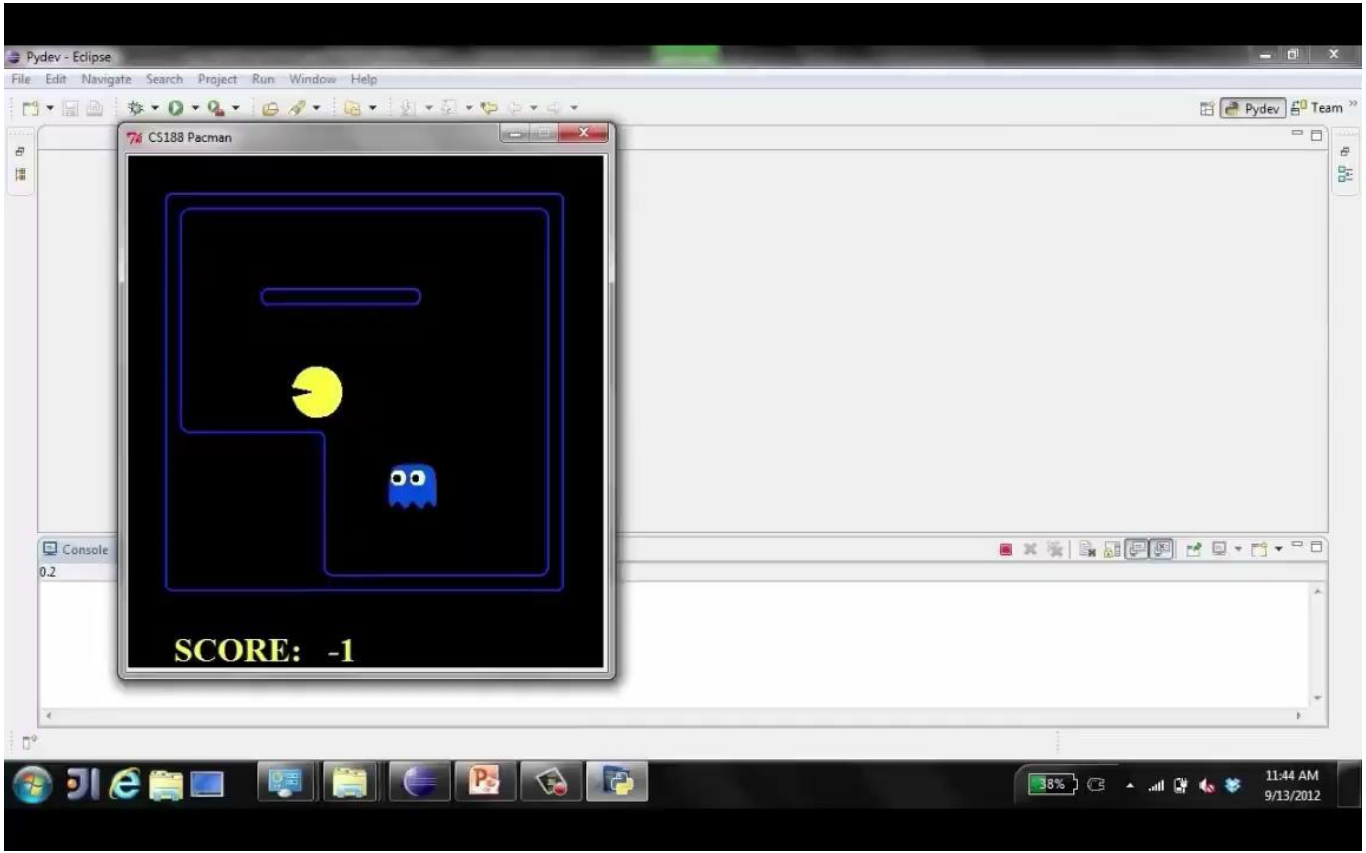
	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

Results from playing 5 games

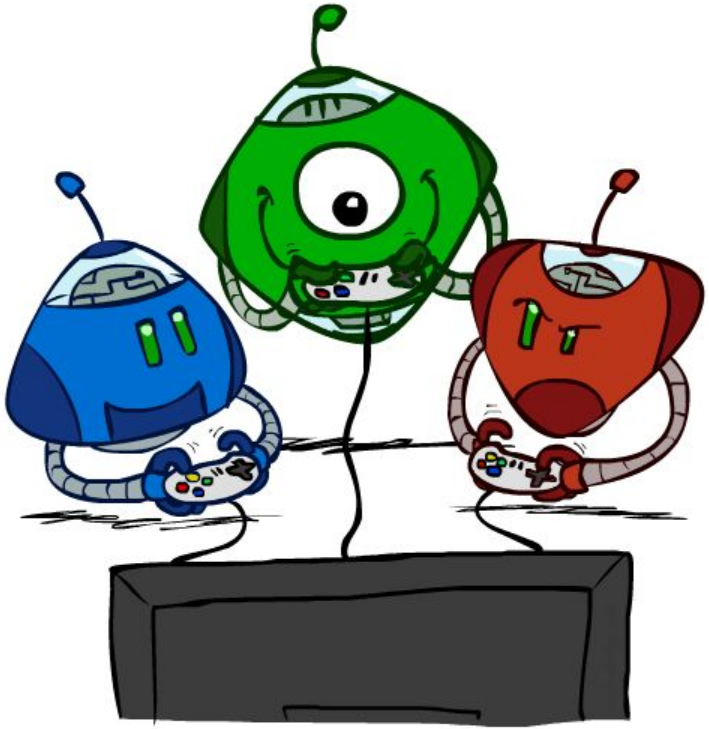
Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

[demo: world assumptions]

Video of Demo World Assumptions
Random Ghost – Minimax Pacman

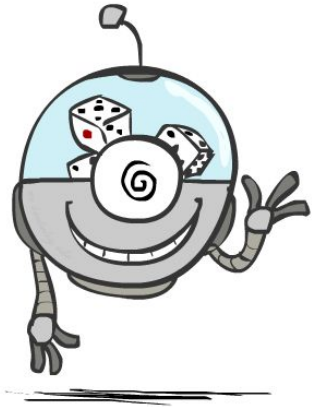


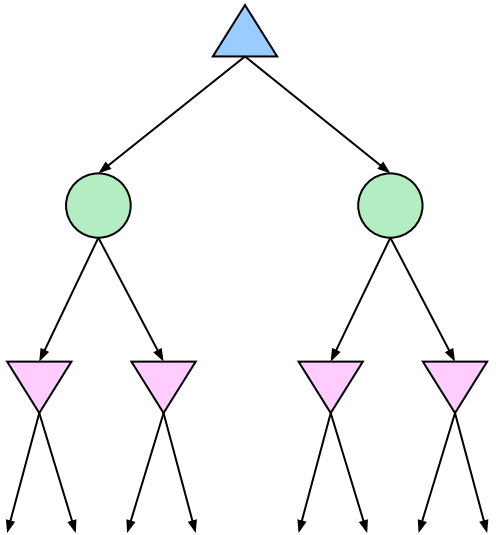
Other Game Types



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



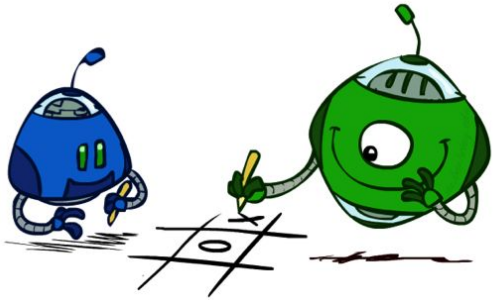


Example: Backgammon

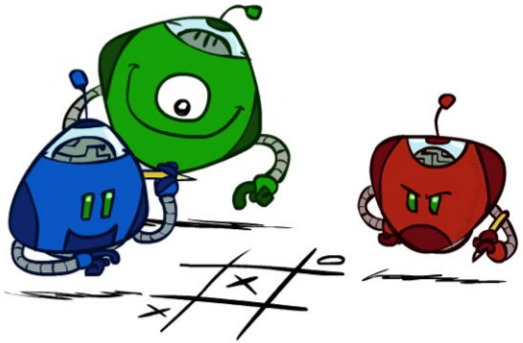
- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon ≈ 20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - 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!

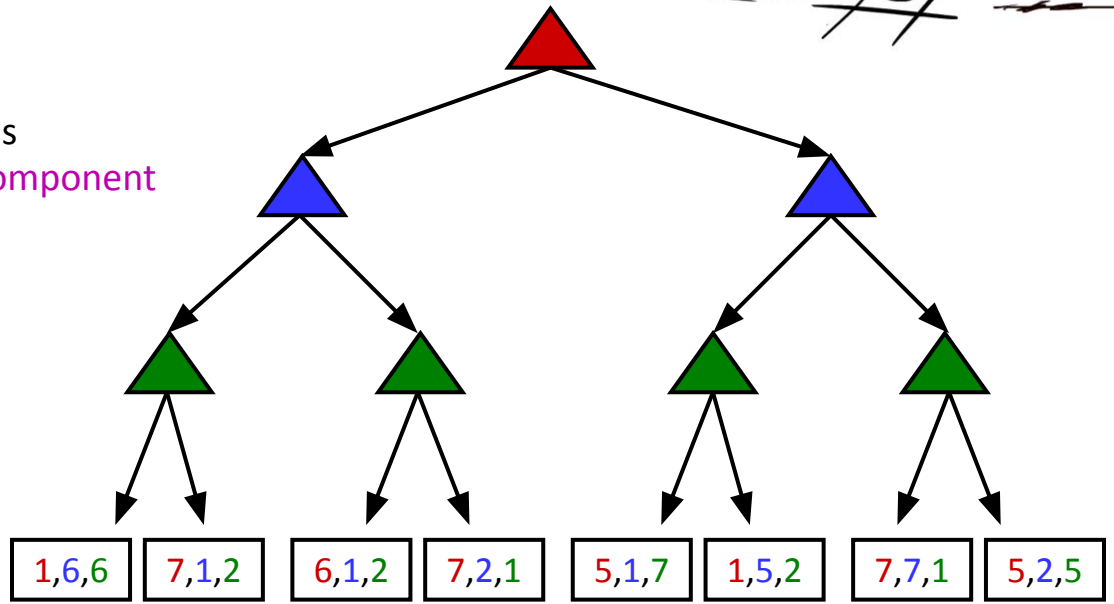
Image: Wikipedia

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…





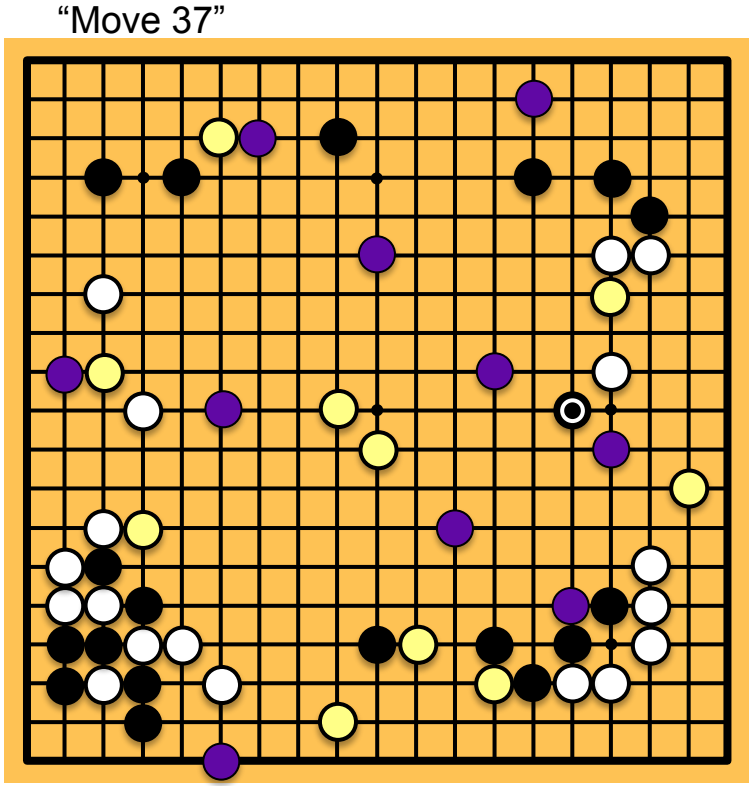
Monte Carlo Tree Search

Monte Carlo Tree Search

- Methods based on alpha-beta search assume a fixed horizon
  - Pretty hopeless for Go, with $b > 300$
- MCTS combines two important ideas:
  - ***Evaluation by rollouts*** – play multiple games to termination from a state *s* (using a simple, fast rollout policy) and count wins and losses
  - ***Selective search*** – explore parts of the tree that will help improve the decision at the root, regardless of depth

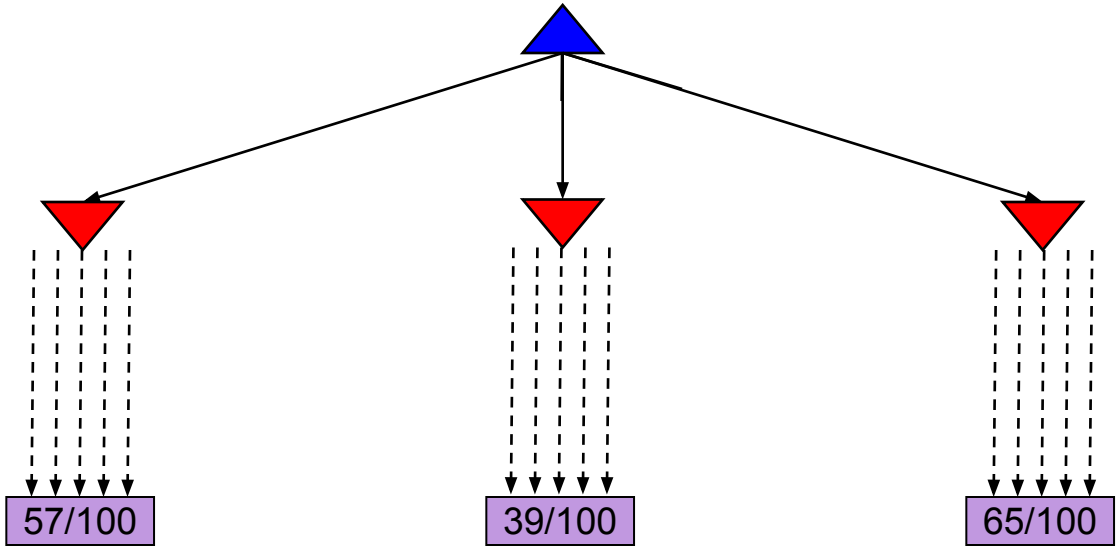
Rollouts

- For each rollout:
  - Repeat until terminal:
    - Play a move according to a fixed, fast rollout policy
  - Record the result
- Fraction of wins correlates with the true value of the position!
- Having a “better” rollout policy helps



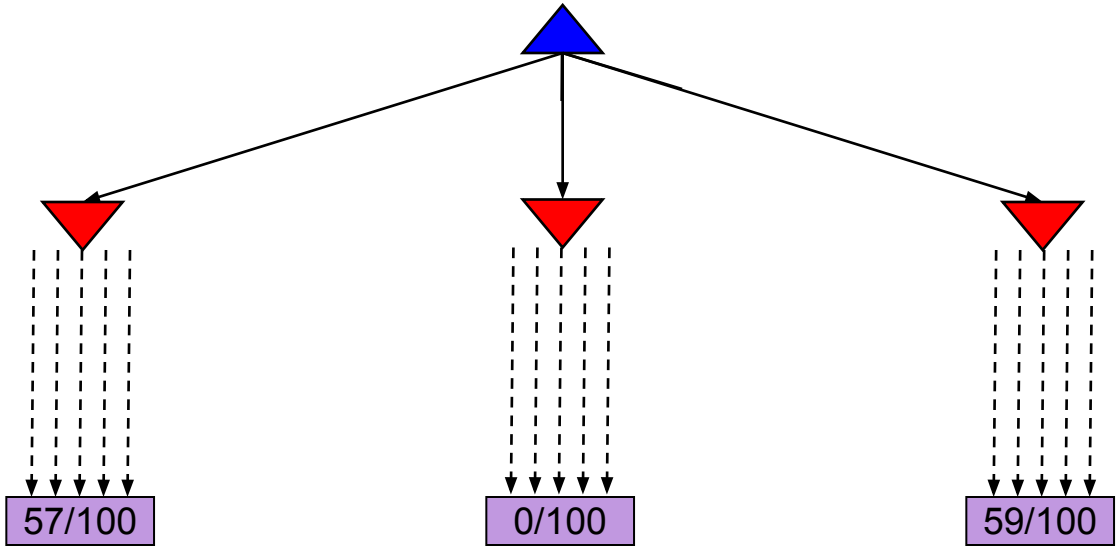
MCTS Version 0

- Do *N* rollouts from each child of the root, record fraction of wins
- Pick the move that gives the best outcome by this metric



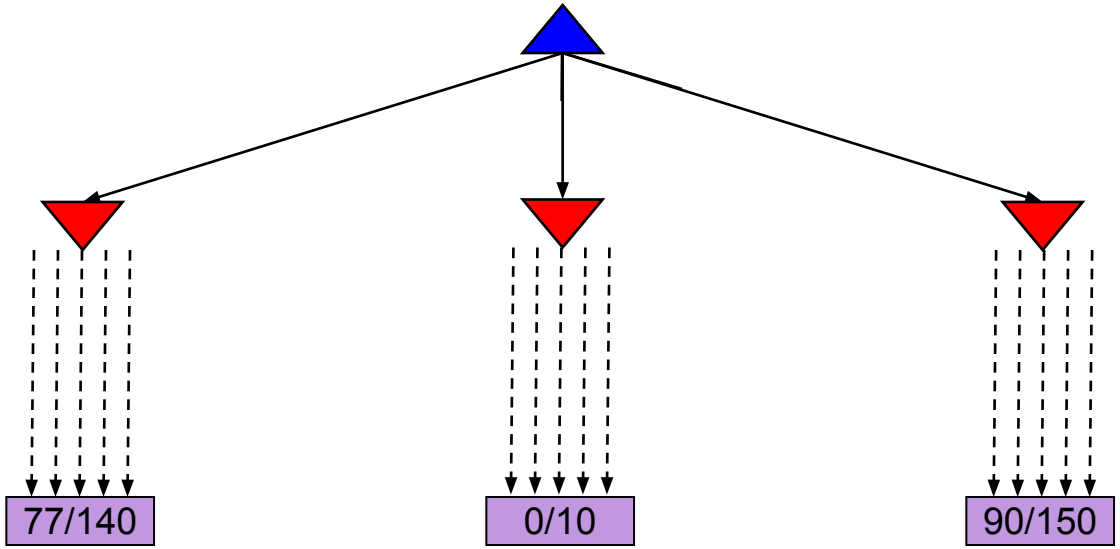
MCTS Version 0

- Do *N* rollouts from each child of the root, record fraction of wins
- Pick the move that gives the best outcome by this metric



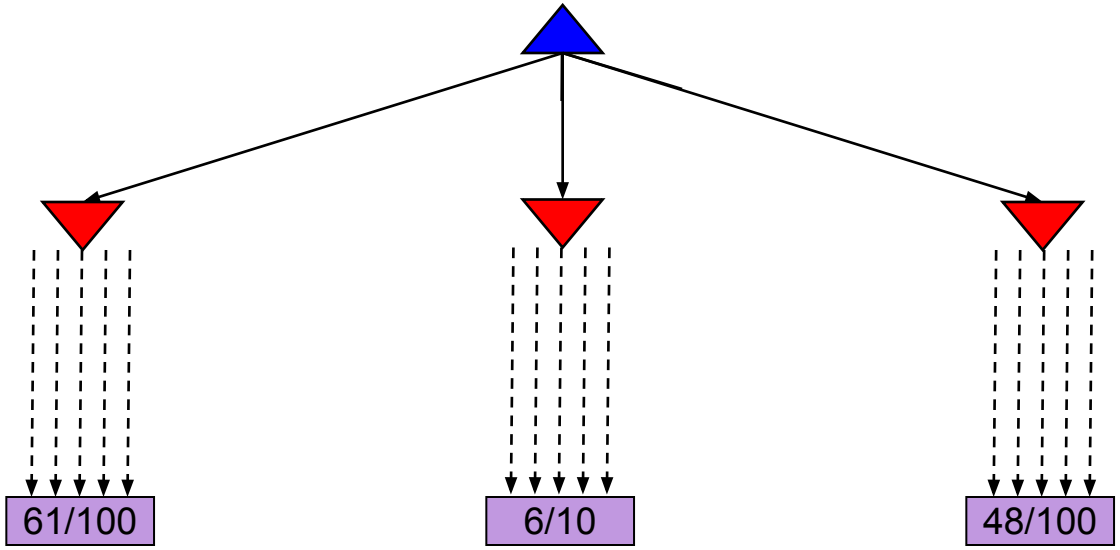
MCTS Version 0.9

- Allocate rollouts to more promising nodes



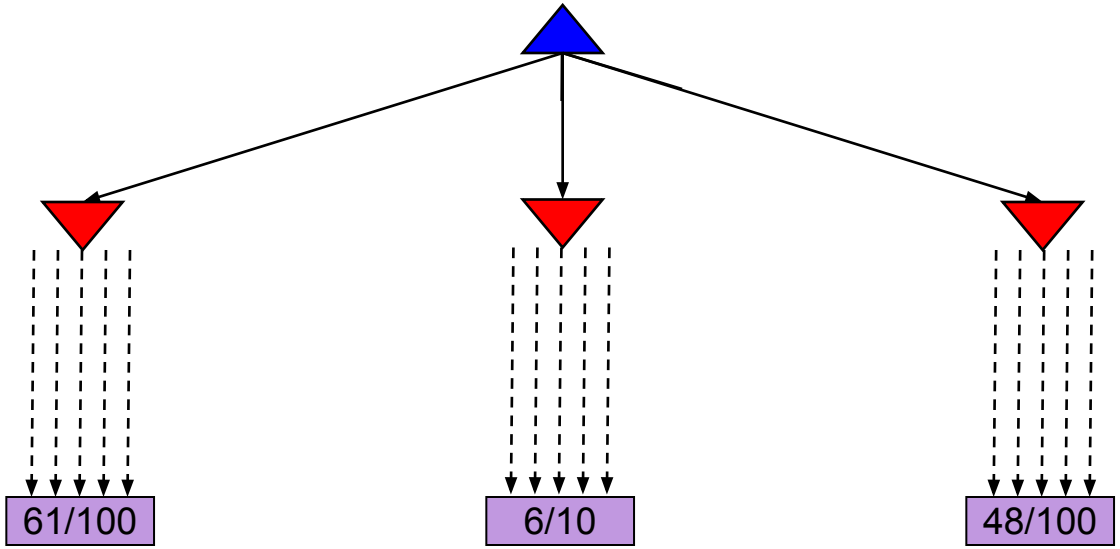
MCTS Version 0.9

- Allocate rollouts to more promising nodes



MCTS Version 1.0

- Allocate rollouts to more promising nodes
- Allocate rollouts to more uncertain nodes



UCB heuristics

- UCB1 formula combines "promising" and "uncertain":

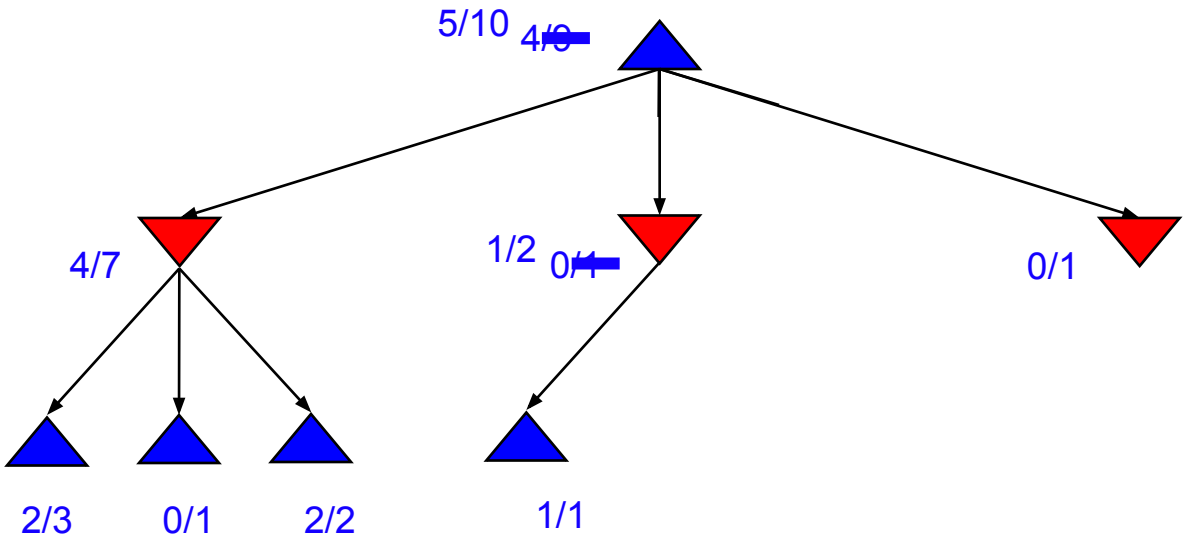
$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

- $N(n)$ = number of rollouts from node n
- $U(n)$ = total utility of rollouts (e.g., # wins) for Player(Parent(n))
- A provably not terrible heuristic for ***bandit problems***
  - (which are not the same as the problem we face here!)

MCTS Version 2.0: UCT

- Repeat until out of time:
  - Given the current search tree, recursively apply UCB to choose a path down to a leaf (not fully expanded) node *n*
  - Add a new child *c* to *n* and run a rollout from *c*
  - Update the win counts from *c* back up to the root
- Choose the action leading to the child with highest *N*

UCT Example



Why is there no min or max?

- "Value" of a node, $U(n)/N(n)$, is a weighted ***sum*** of child values!
- Idea: as $N \rightarrow \infty$, the vast majority of rollouts are concentrated in the best child(ren), so weighted average $\rightarrow$ max/min
- Theorem: as $N \rightarrow \infty$ UCT selects the minimax move
  - (but N never approaches infinity!)

Summary

- Games require decisions when optimality is impossible
  - Bounded-depth search and approximate evaluation functions
- Games force efficient use of computation
  - Alpha-beta pruning, MCTS
- Game playing has produced important research ideas
  - Reinforcement learning (checkers)
  - Iterative deepening (chess)
  - Rational metareasoning (Othello)
  - Monte Carlo tree search (chess, Go)
  - Solution methods for partial-information games in economics (poker)
- Video games present much greater challenges – lots to do!
  - $b = 10^{500}$, $|S| = 10^{4000}$, $m = 10{,}000$, partially observable, often > 2 players