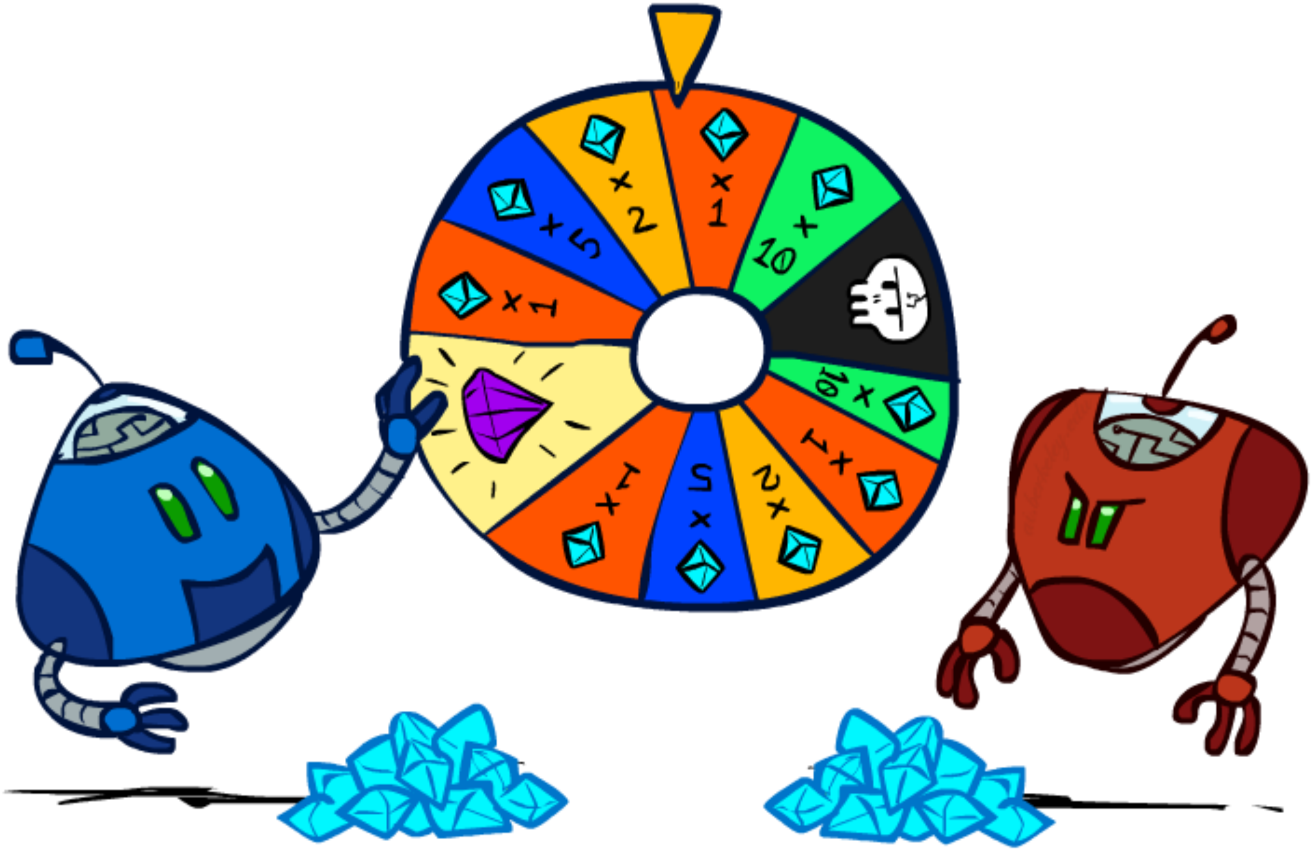
Announcements

- **Project 2** due **next Friday (Sept 22)** at 11:59pm PT

CS 188: Artificial Intelligence

Search with Other Agents: Uncertainty

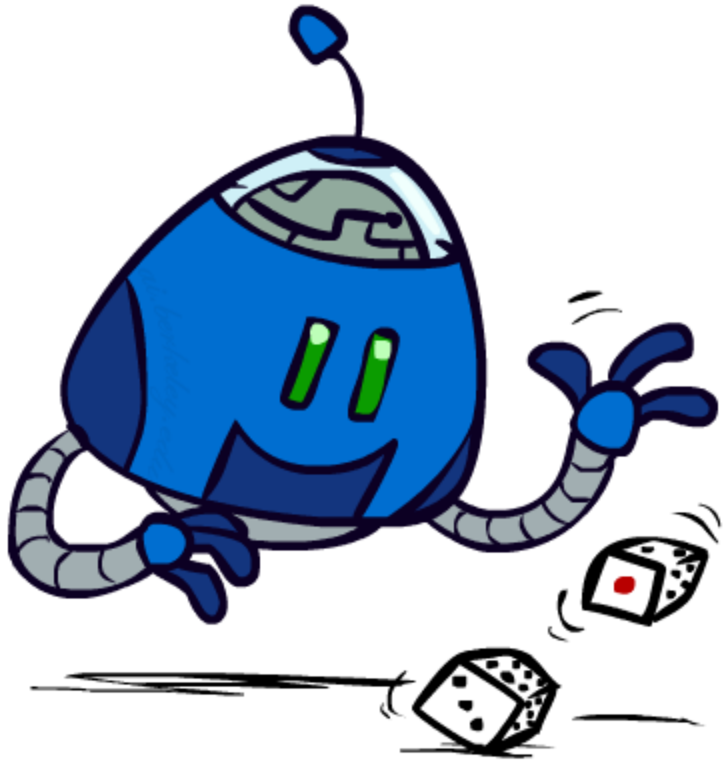


University of California, Berkeley

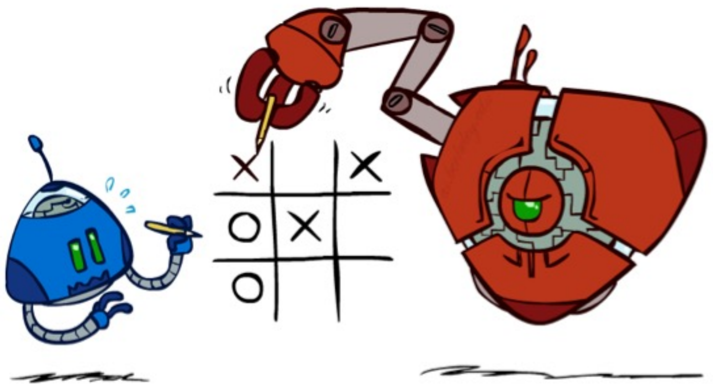
[These slides were created by Dan Klein, Pieter Abbeel for CS188 Intro to AI at UC Berkeley (ai.berkeley.edu).]

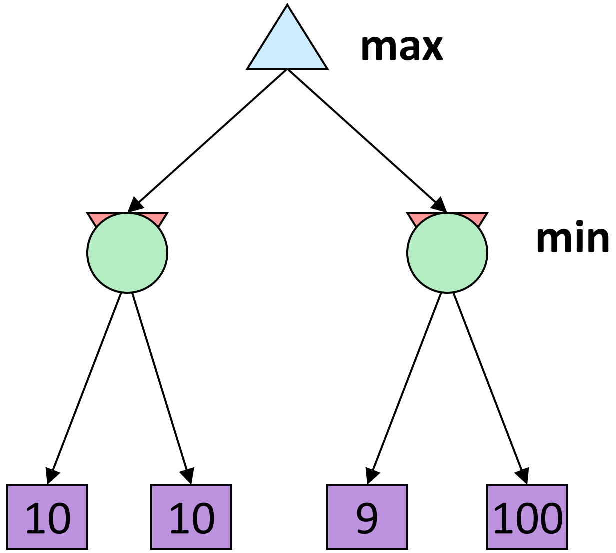
Uncertain Outcomes

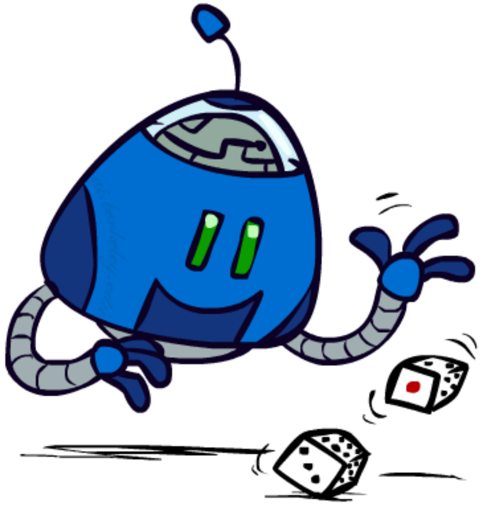
- Why do we care about uncertainty and randomness?
  - Want to model random events happening in the world
  - Build efficient algorithms with random sampling (Monte Carlo Tree Search)



Worst-Case vs. Average Case







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

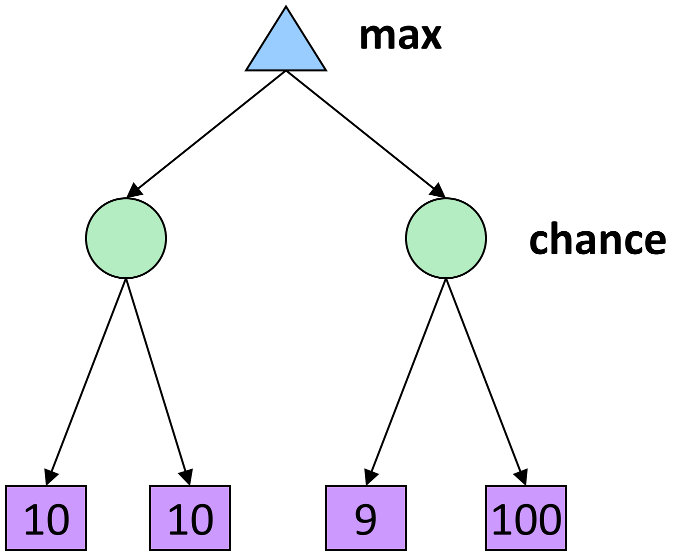
Expectimax Search

- 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

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



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

Video of Demo Minimax vs Expectimax (Min)

Video of Demo Minimax vs Expectimax (Exp)

Expectimax Pseudocode

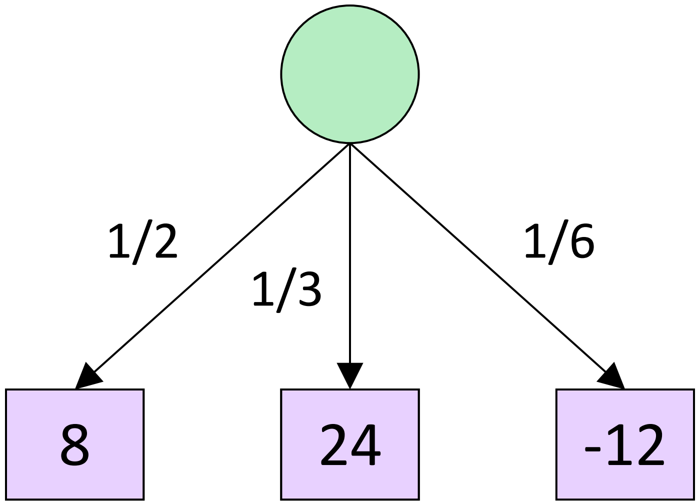
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

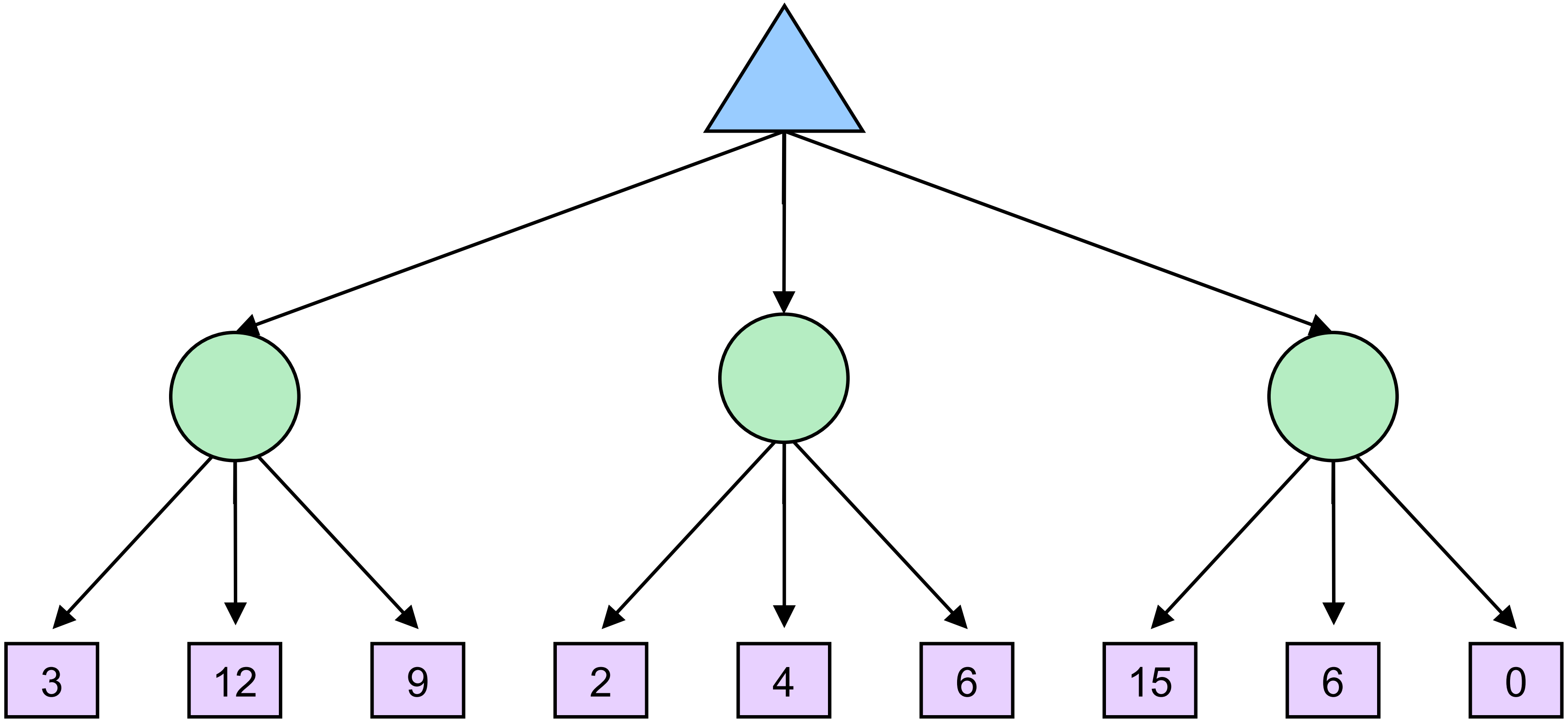
Expectimax Pseudocode

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

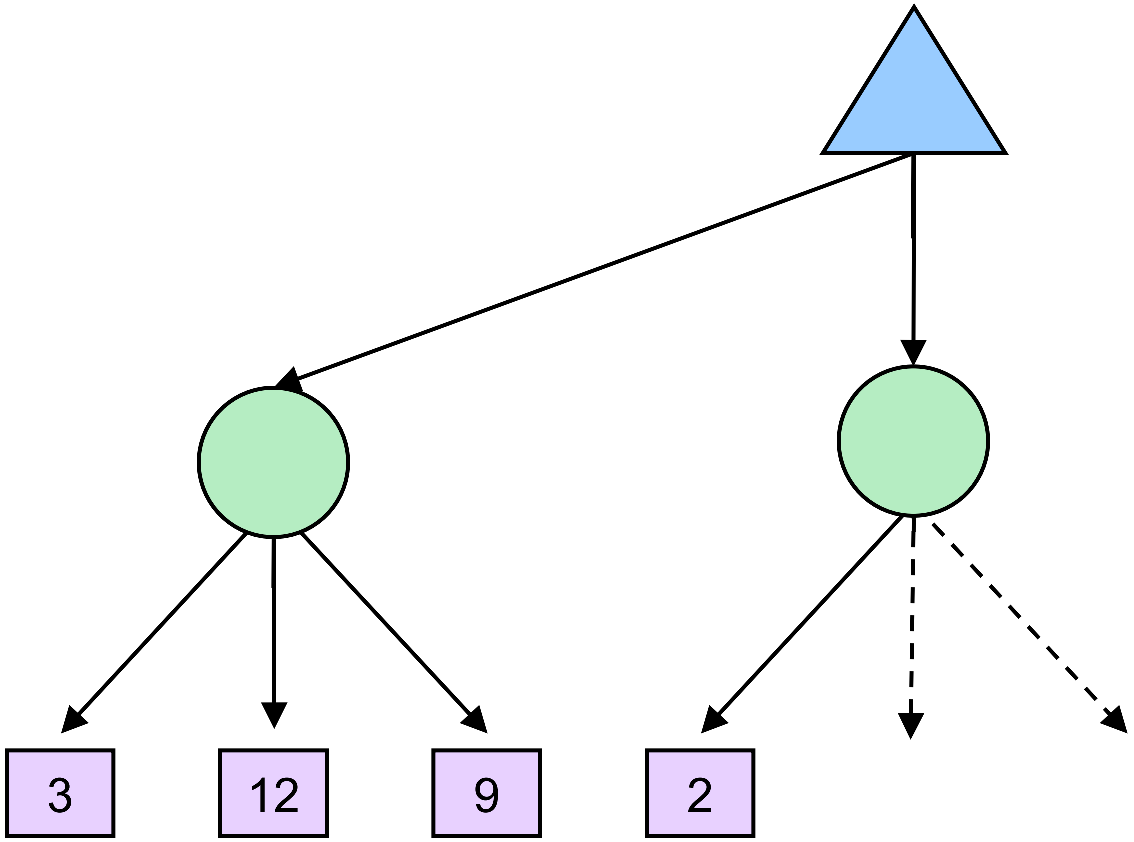


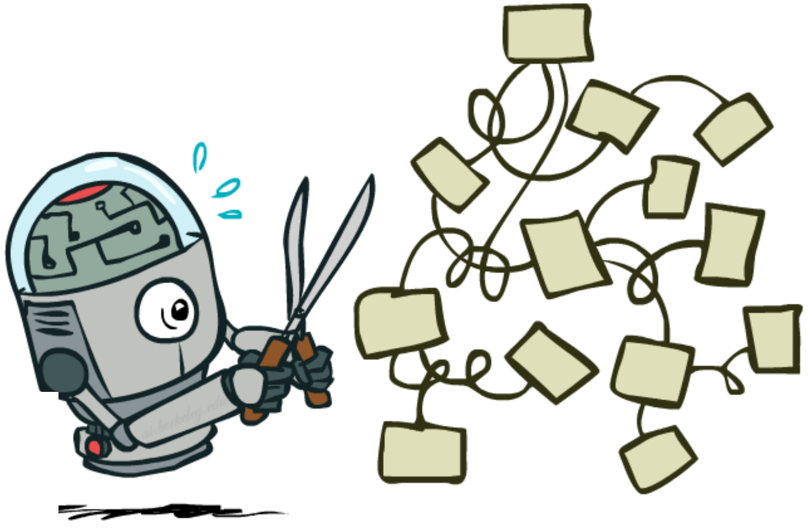
$$v = (1/2)\,(8) + (1/3)\,(24) + (1/6)\,(-12) = 10$$

Expectimax Example

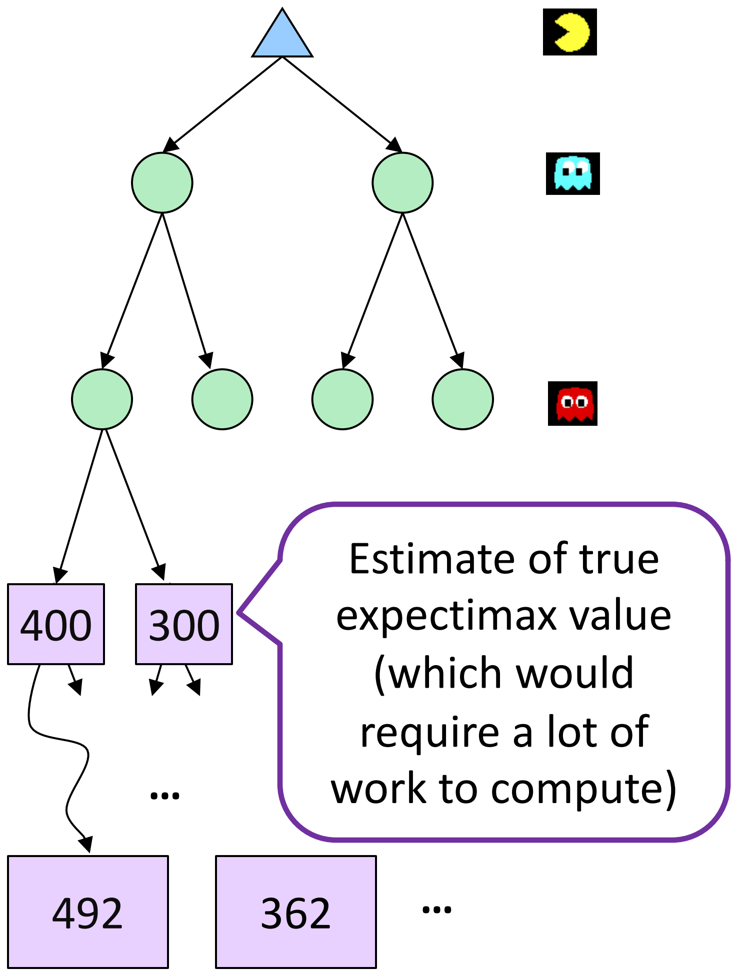


Expectimax Pruning?

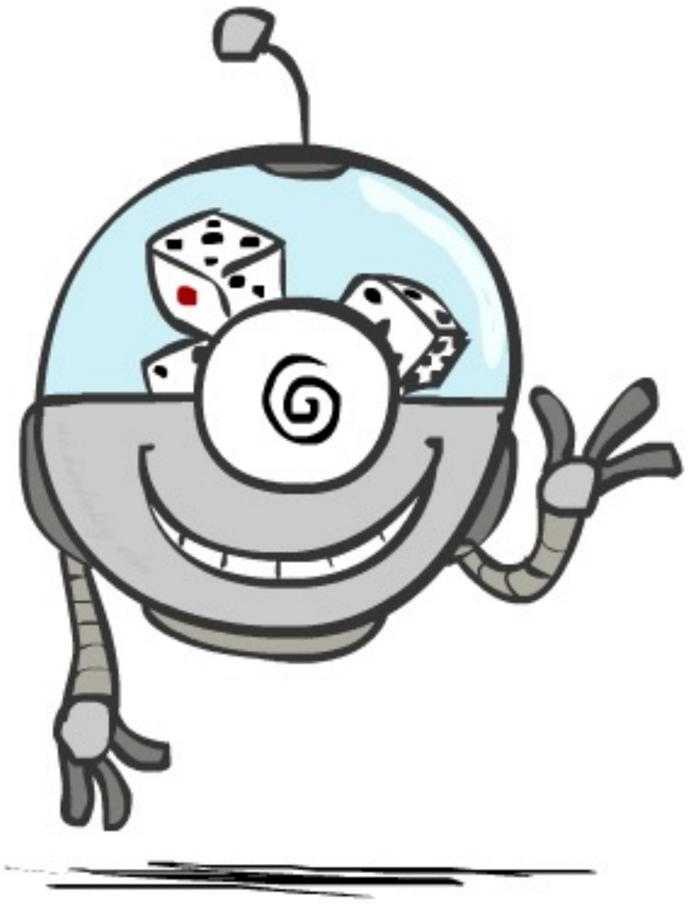




Depth-Limited Expectimax



Probabilities



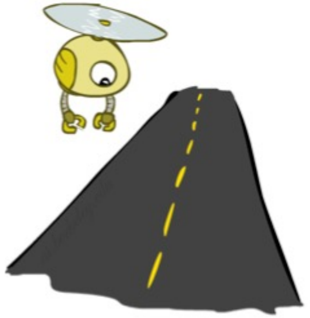
Reminder: Probabilities

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

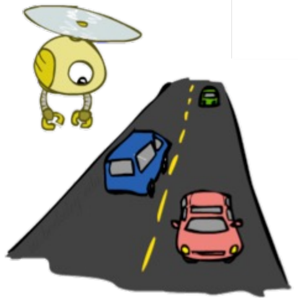
- Example: Traffic on freeway
  - Random variable: T = whether there’s traffic
  - Outcomes: T in {none, light, heavy}
  - Distribution: P(T=none) = 0.25, P(T=light) = 0.50, P(T=heavy) = 0.25

- 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

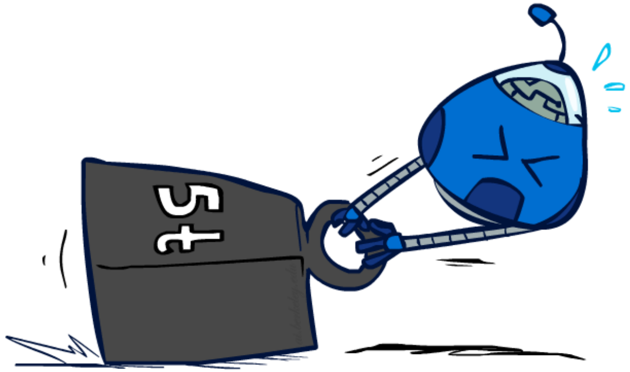


0.50

0.25

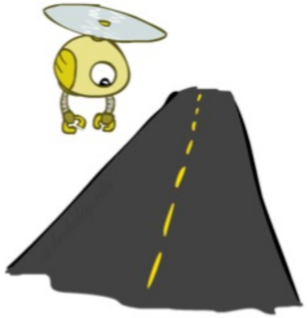
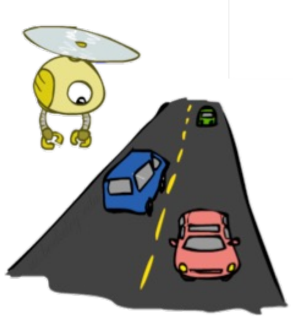
Reminder: Expectations

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



- Example: How long to get to the airport?

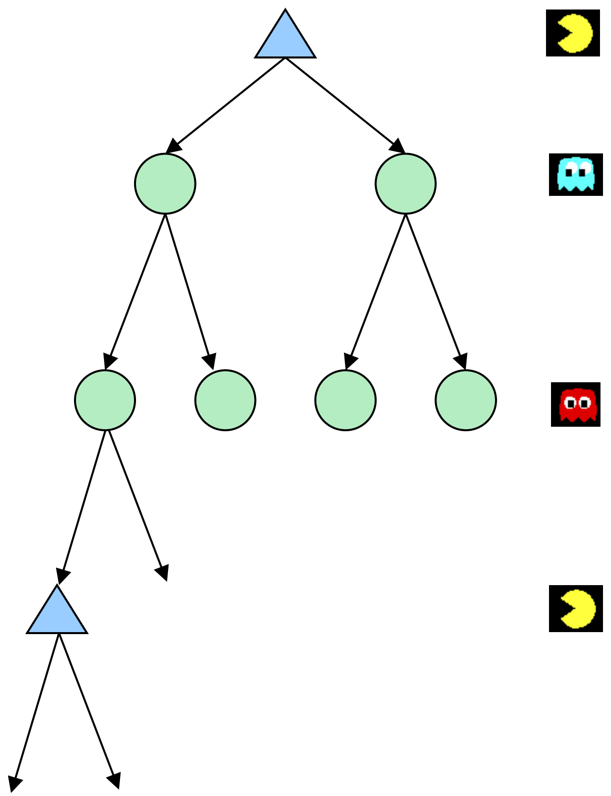
Time:	20 min		30 min		60 min	
	x	+	x	+	x	35 min
Probability:	0.25		0.50		0.25	


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!

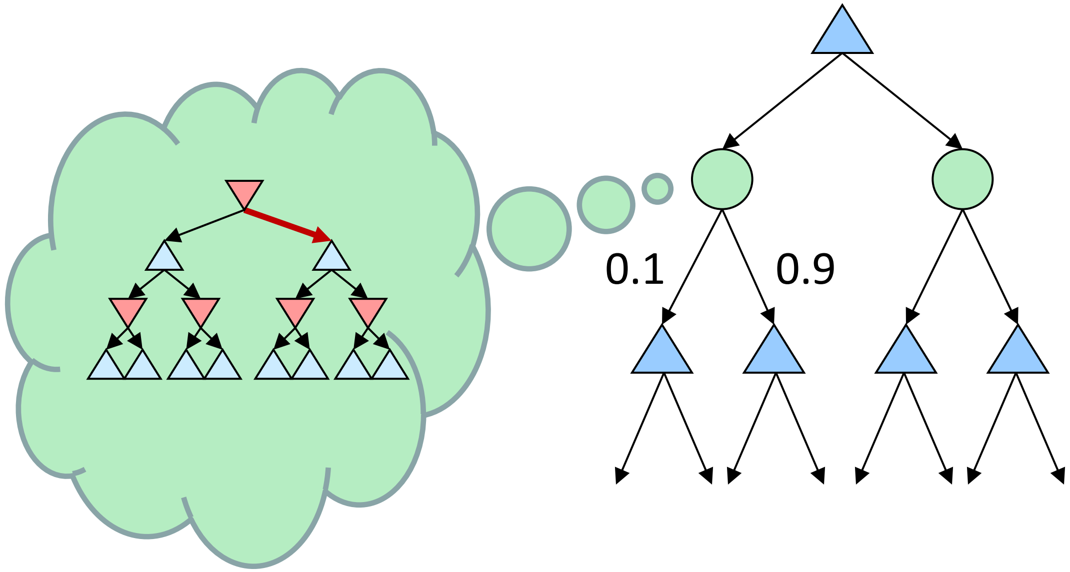
- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



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

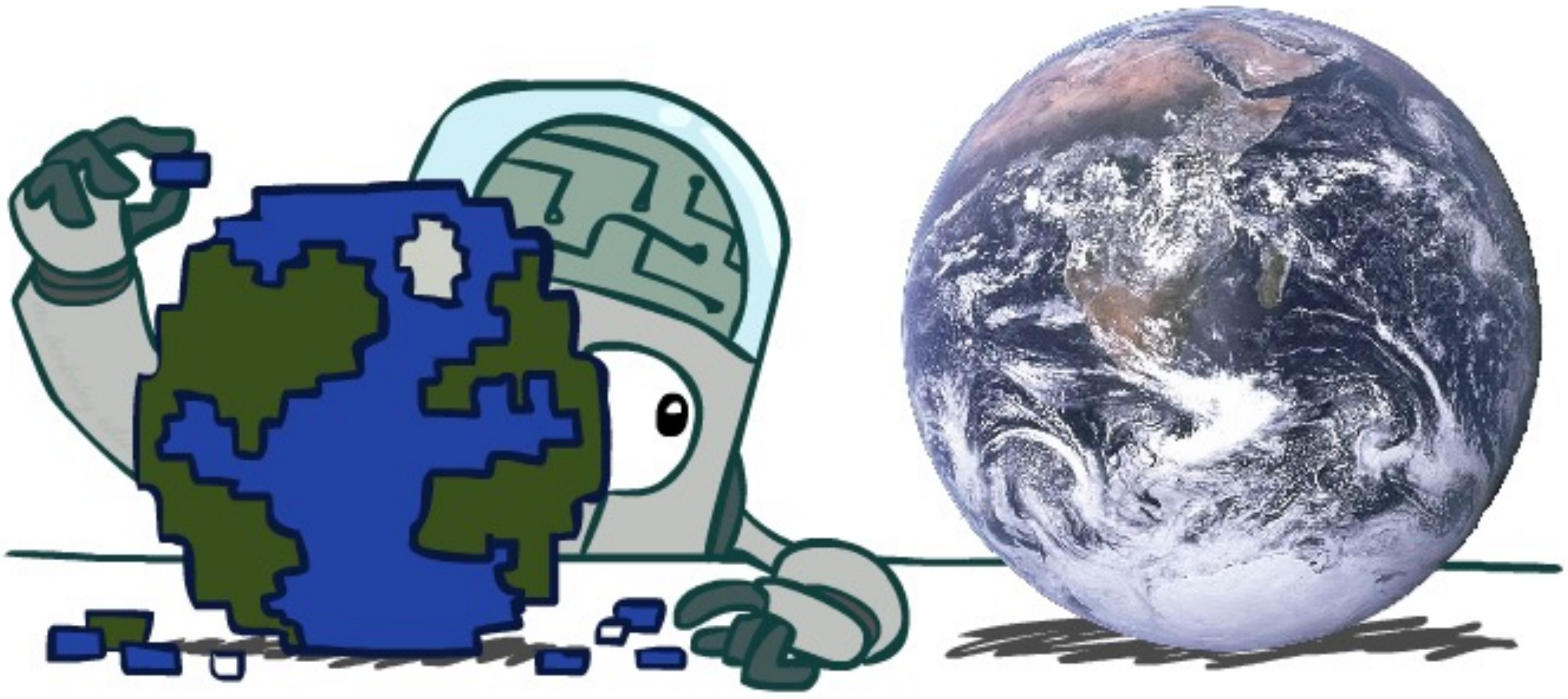
Quiz: Informed Probabilities

- 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

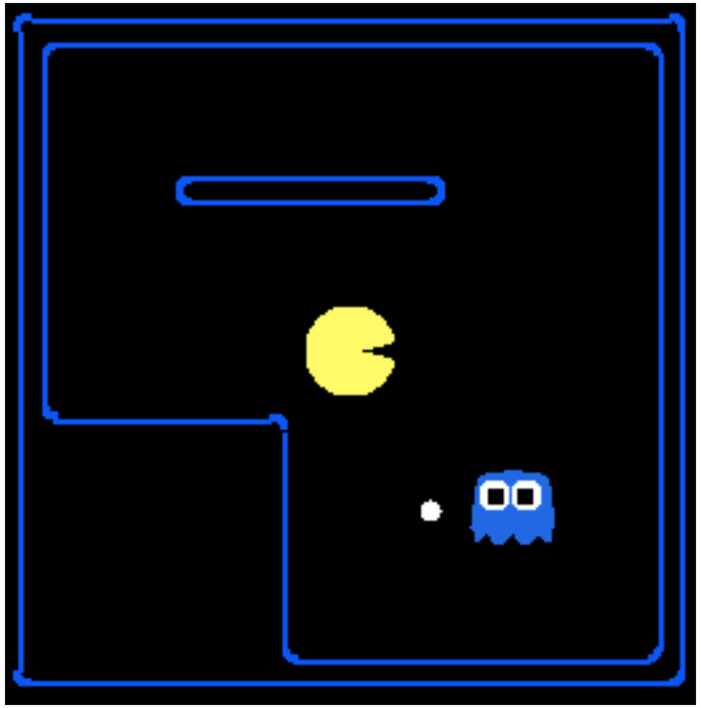
Dangerous Optimism

Assuming chance when the world is adversarial

Dangerous Pessimism

Assuming the worst case when it's not likely

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 (L7D3,4,5,6)]

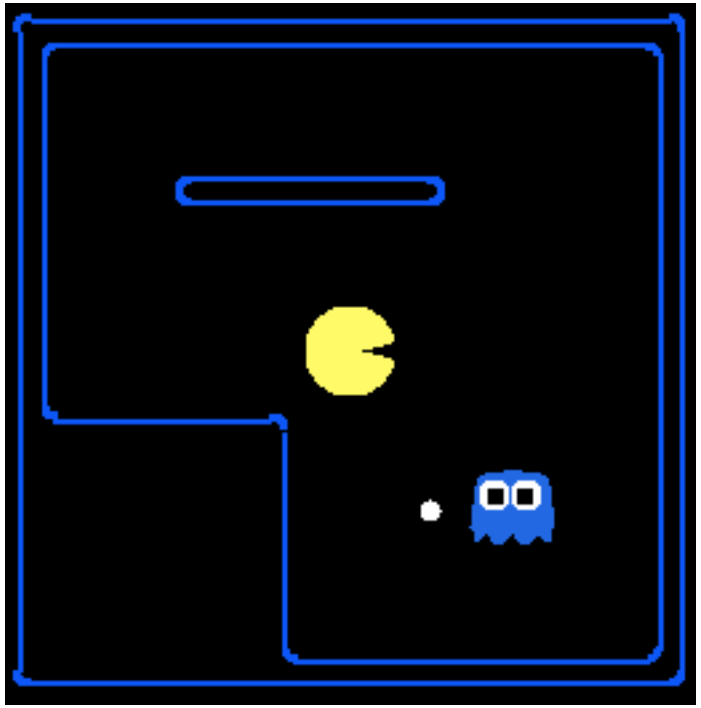
Video of Demo World Assumptions
Adversarial Ghost – Expectimax Pacman

Video of Demo World Assumptions
Random Ghost – Minimax Pacman

Video of Demo World Assumptions
Random Ghost – Expectimax Pacman

Video of Demo World Assumptions
Adversarial Ghost – Minimax Pacman

Assumptions vs. Reality



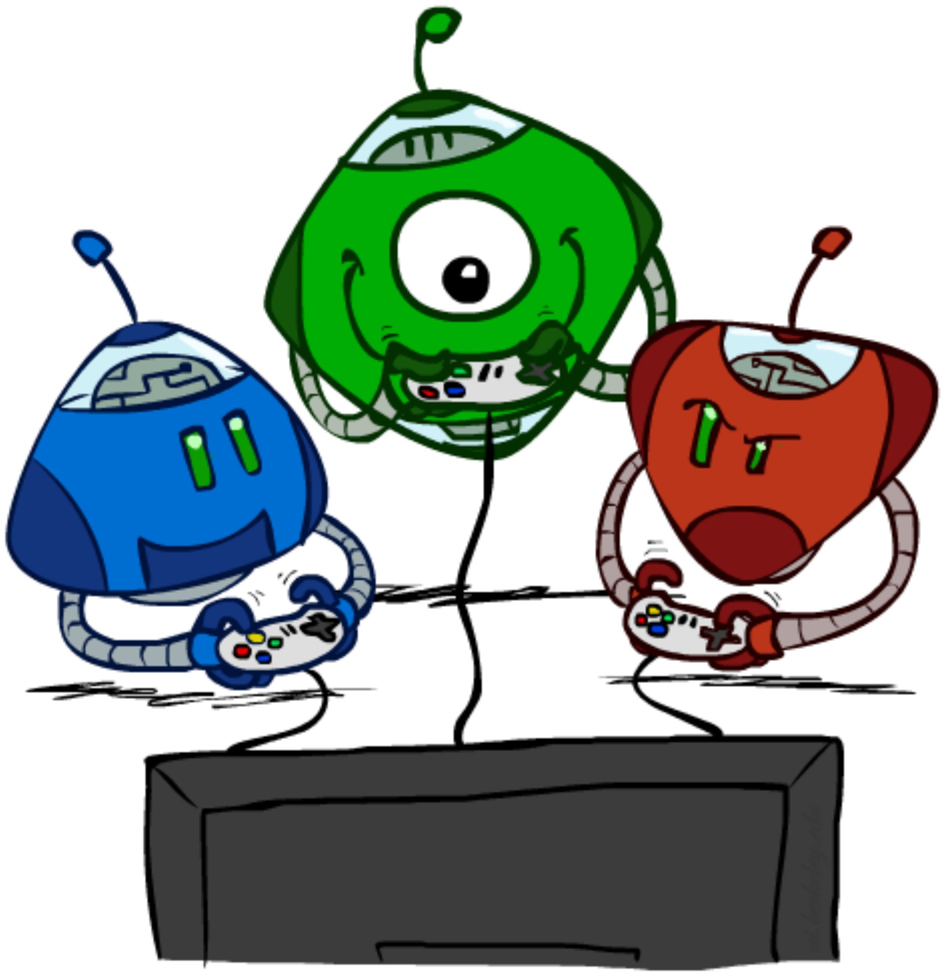
	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

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

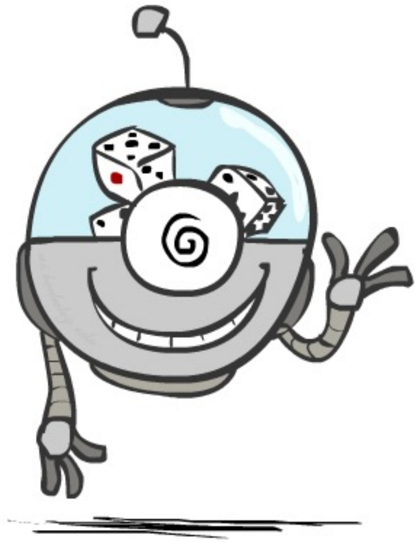
[Demos: world assumptions (L7D3,4,5,6)]

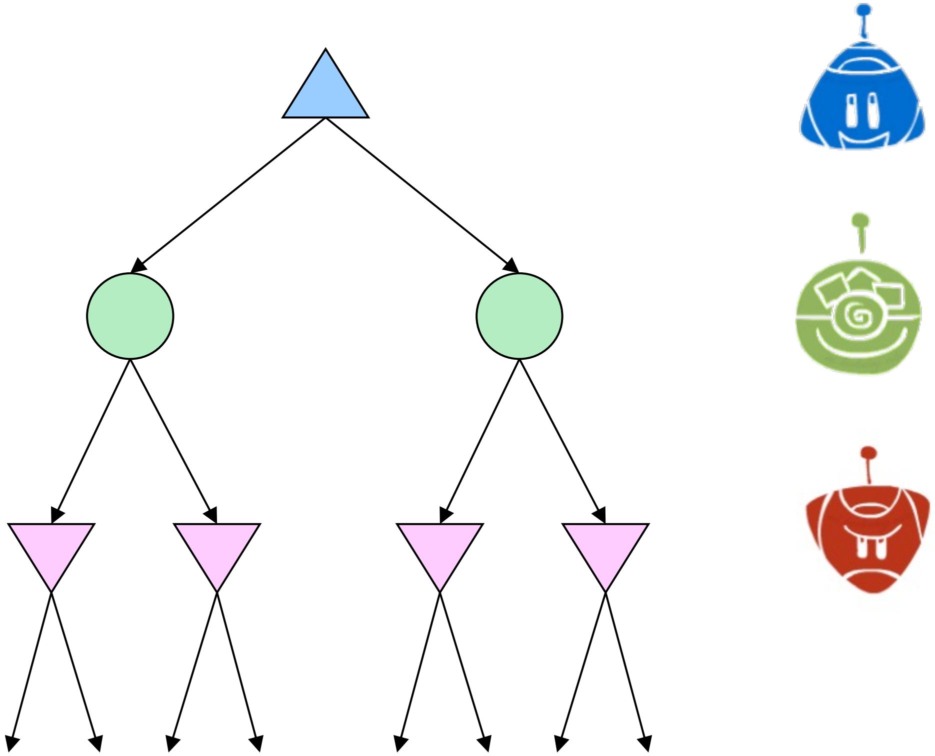
Other Game Types



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



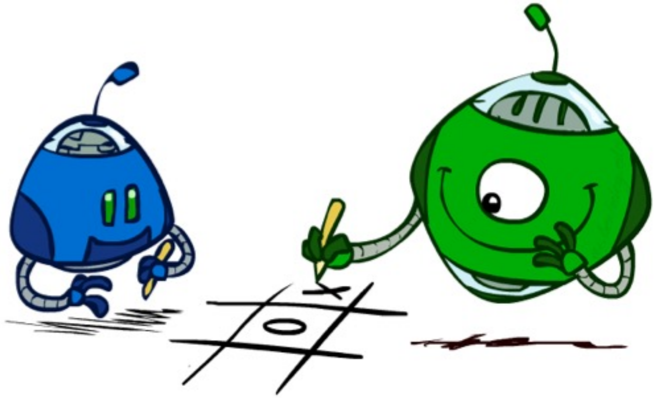


Example: Backgammon

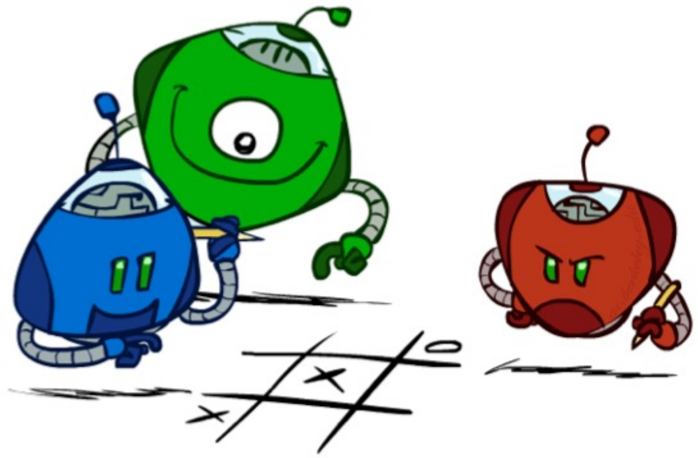
- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon ≈ 20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging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

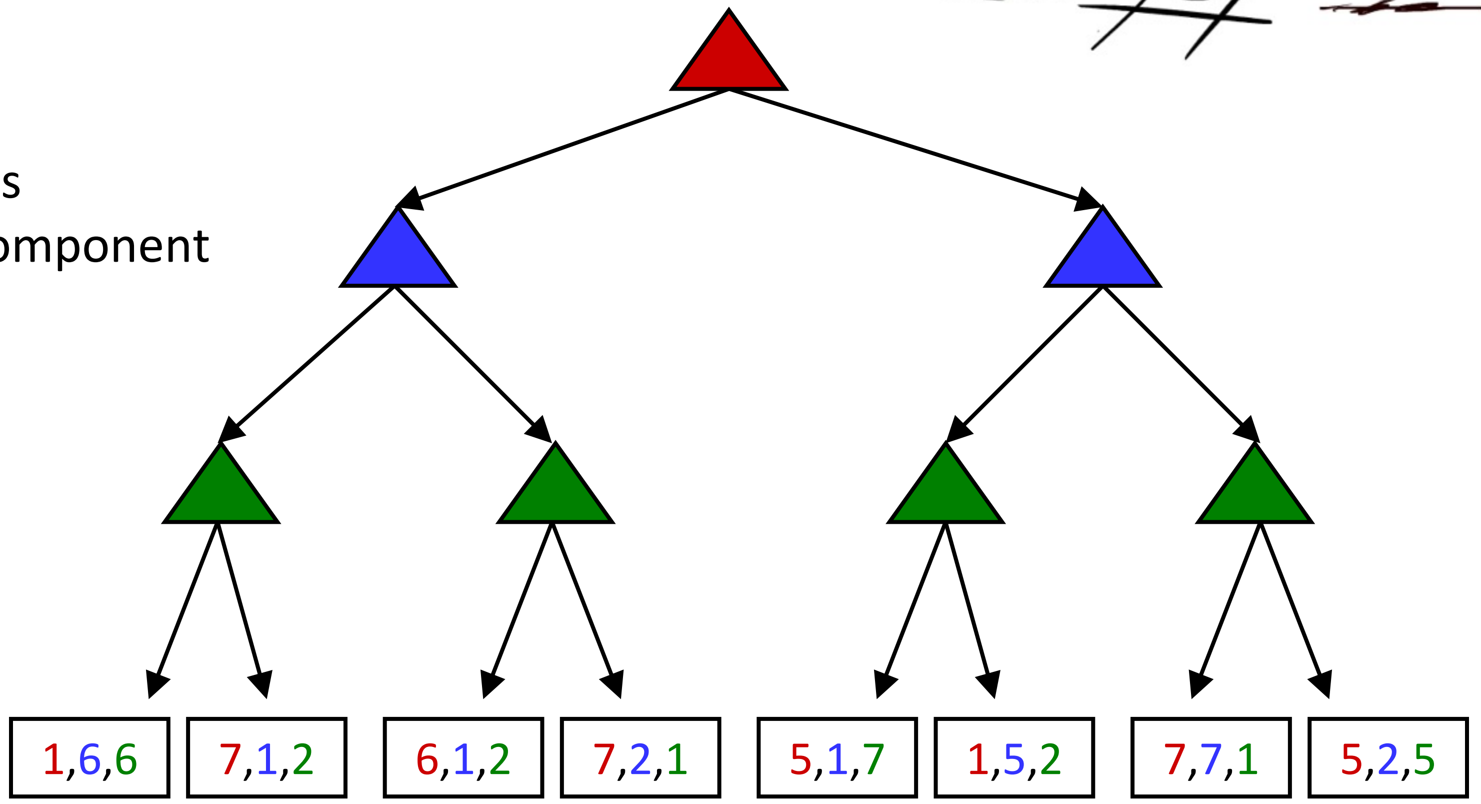
Image: Wikipedia

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…





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, $35^{(8/2)}$ =~ 1M; depth 8 is quite good
  - Difficult for Go: $300^{(8/2)}$ =~ 8 *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

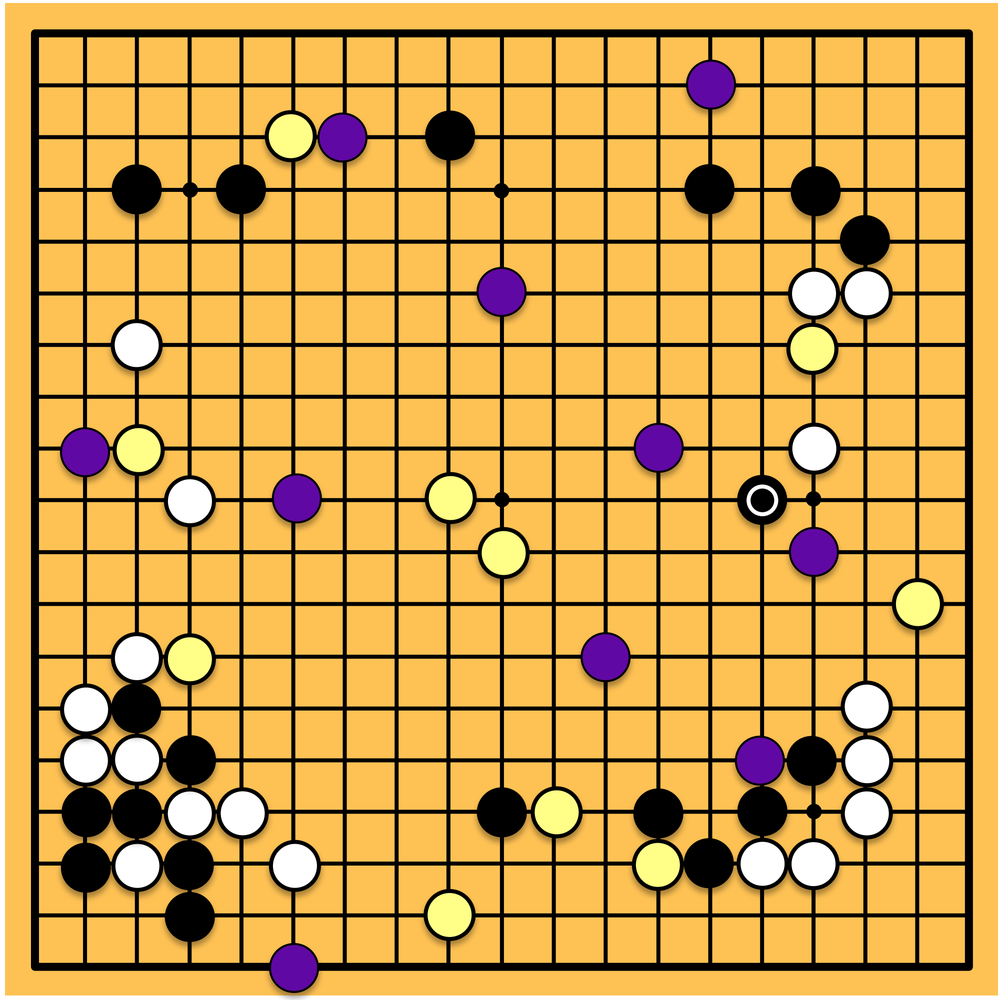
Overcoming Resource Limits with Randomization

- *Monte Carlo Tree Search (MCTS)* combines two important ideas:
  - ***Evaluation by rollouts*** – estimate value of a state by playing many games from this state by taking random actions (or some other fast policy) and count wins & losses
  - ***Selective search*** – explore parts of the tree that will help improve the decision at the root, regardless of depth

Rollouts

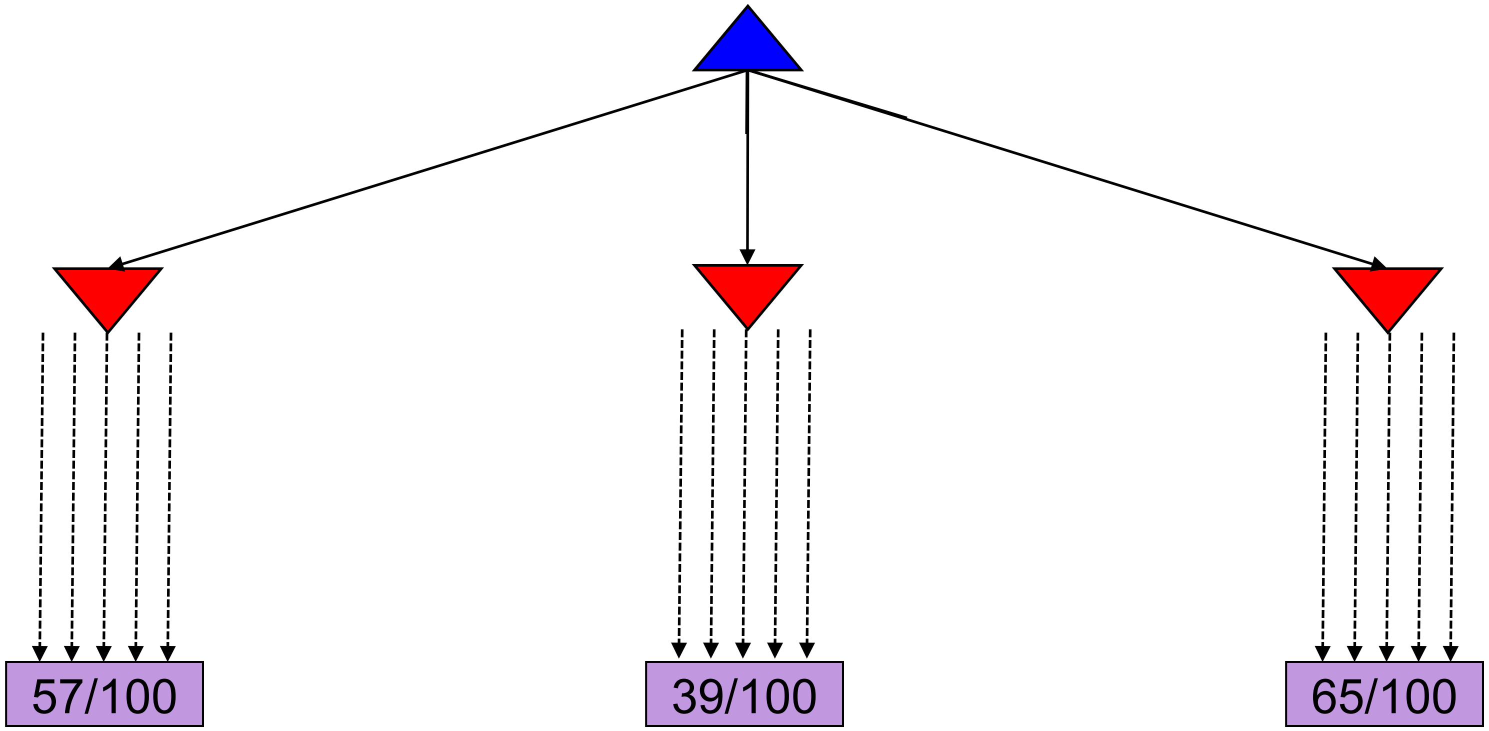
- For each rollout:
  - Repeat until terminal:
    - Play a move according to a fixed, fast rollout policy (i.e. random actions)
  - Record the result
- Fraction of wins correlates with the true value of the position!
- Having a "better" rollout policy helps

"Move 37"



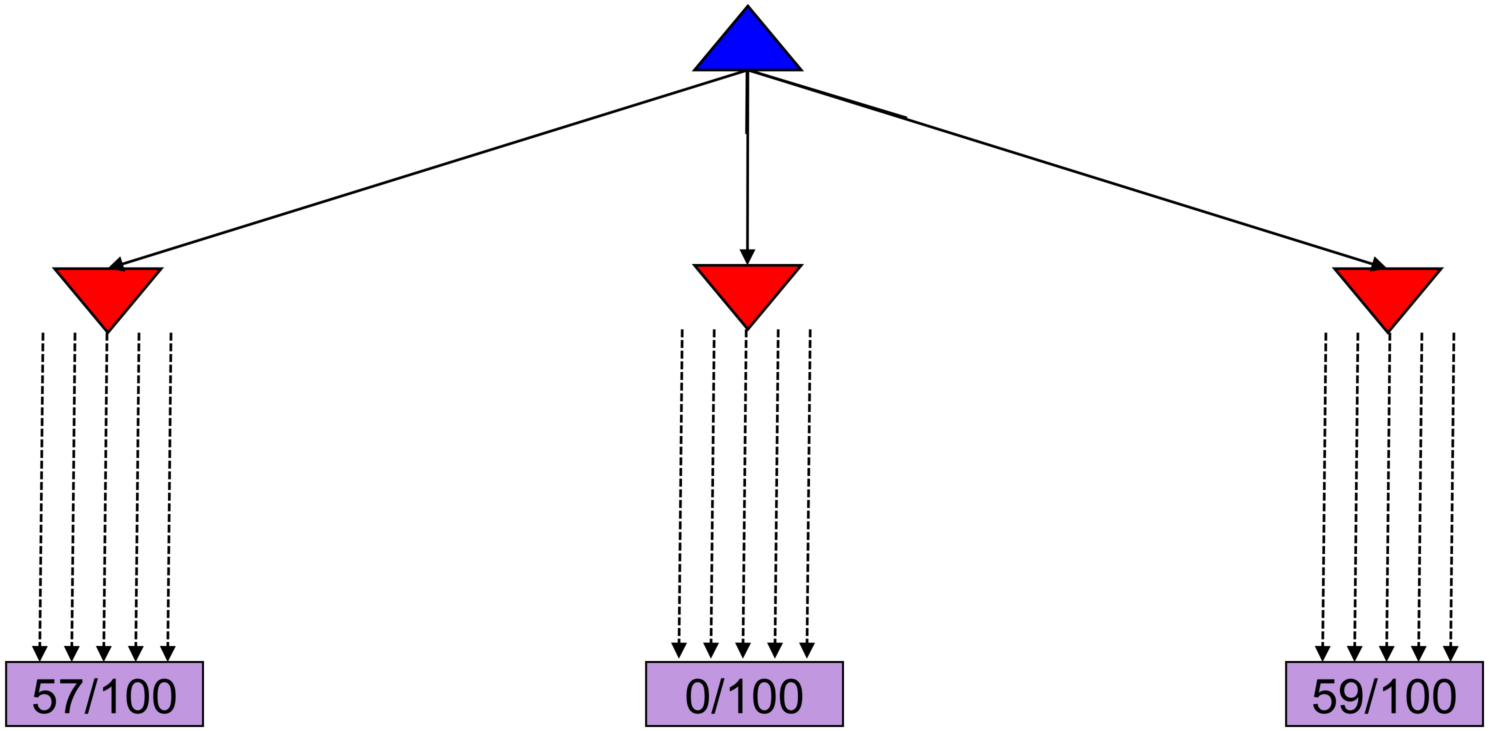
MCTS Version 0

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



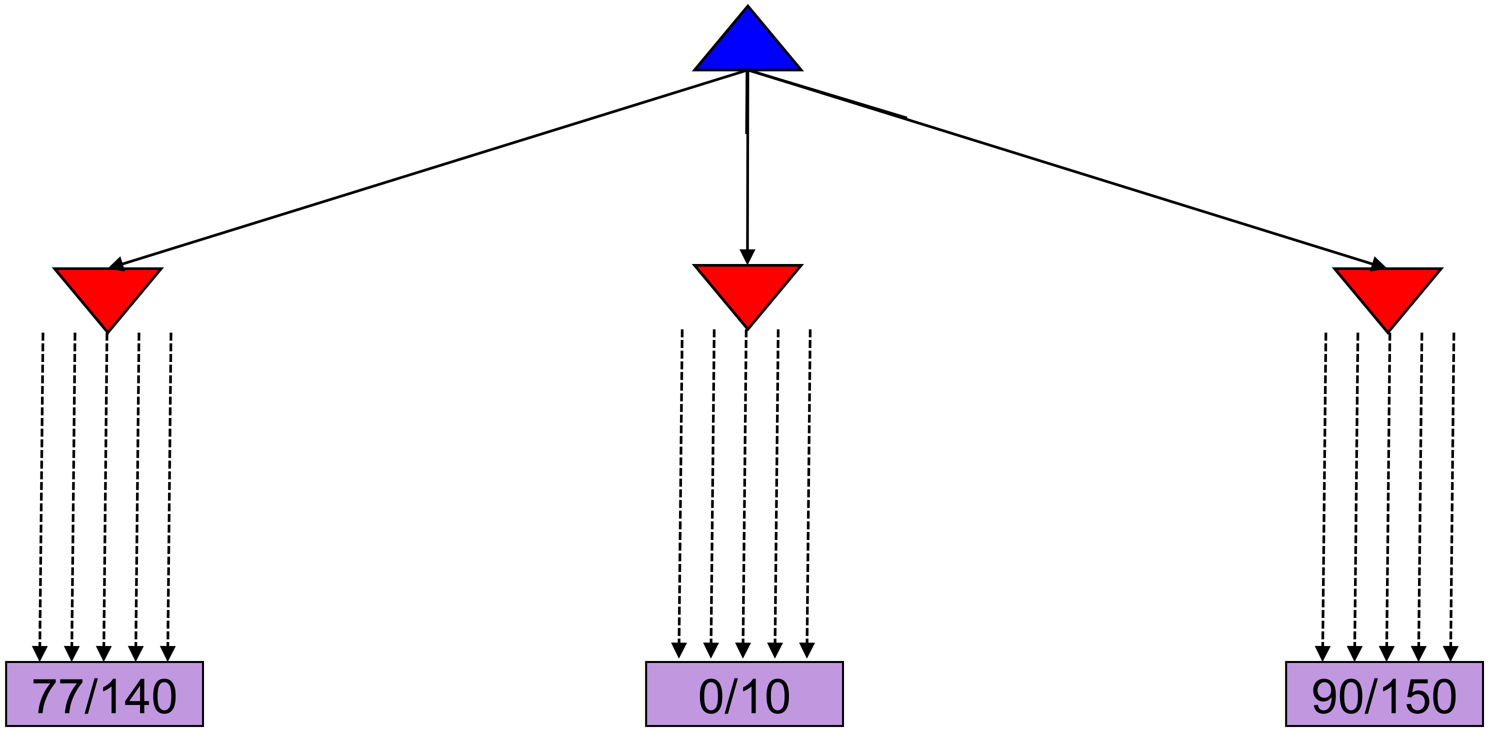
MCTS Version 0

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



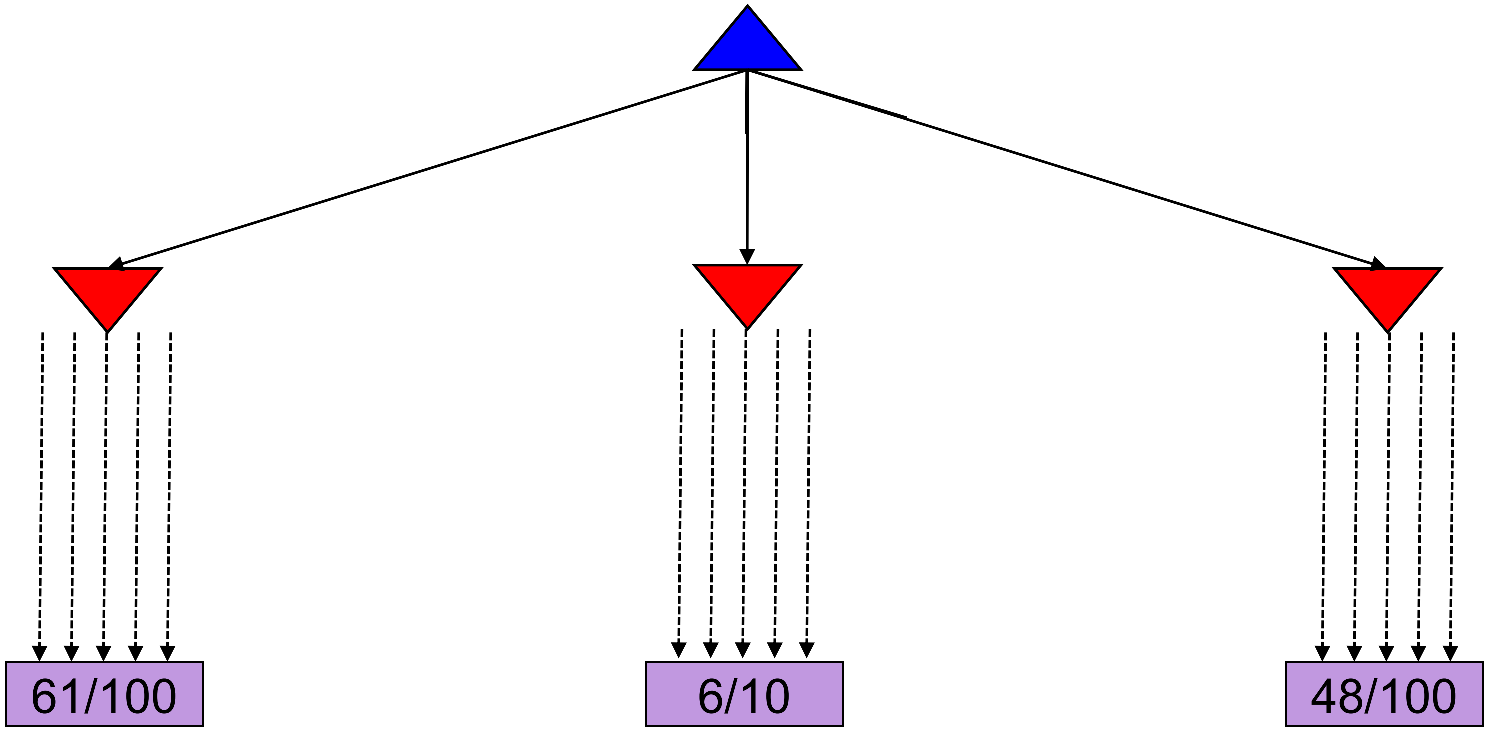
MCTS Version 0.9

- Allocate rollouts to more promising nodes



MCTS Version 1.0

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



Upper Confidence Bounds (UCB) heuristics

- UCB1 formula combines "promising" and "uncertain":
  - C is a parameter we choose to trade off between two terms

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

- $N(n)$ = number of rollouts from node *n*
- $U(n)$ = total utility of rollouts (# wins) for player of Parent(*n*)
  - Keep track of both N and U for each node

Upper Confidence Bounds (UCB) heuristics

- UCB1 formula combines "promising" and "uncertain":
  - C is a parameter we choose to trade off between two terms

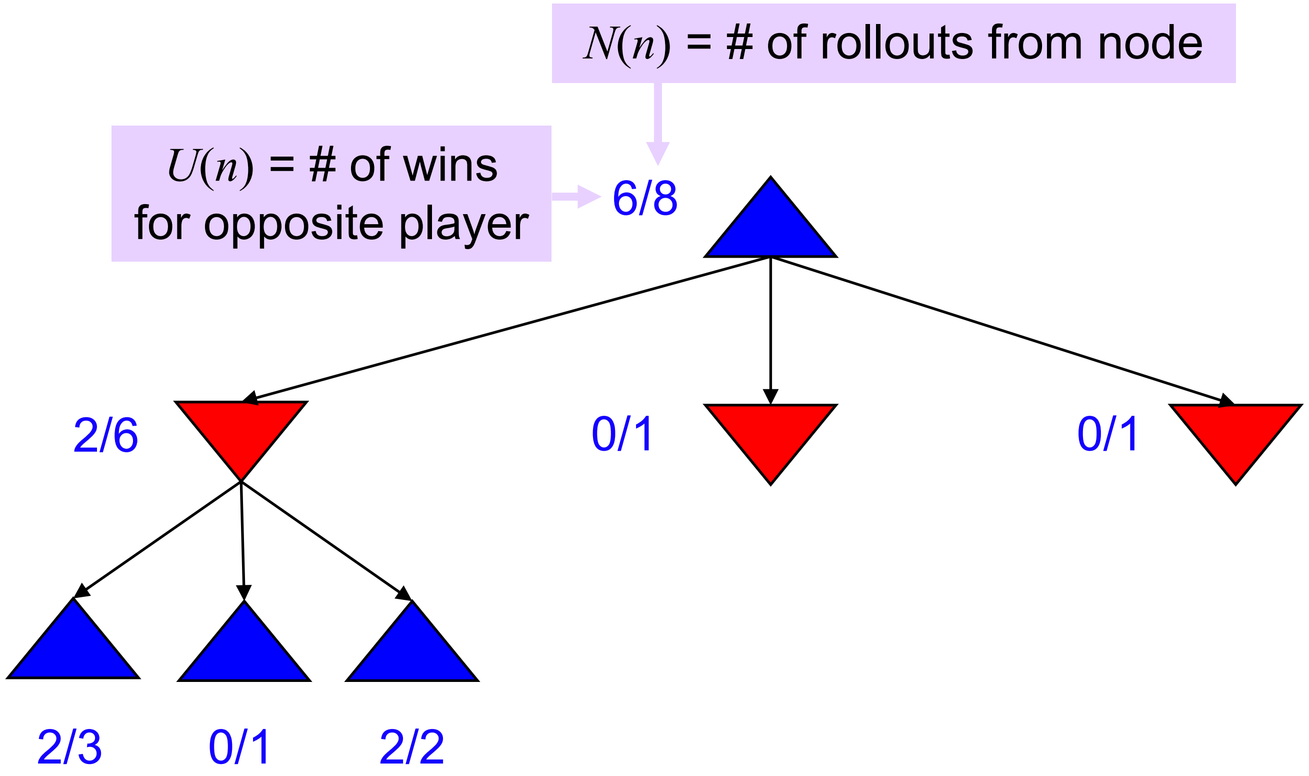
$$UCB1(n) = \frac{U(n)}{N(n)} + \boxed{C \times \sqrt{\frac{\log N(\text{Parent}(n))}{N(n)}}}$$

- High for small N
- Low for large N

- $N(n)$ = number of rollouts from node *n*
- $U(n)$ = total utility of rollouts (# wins) for player of Parent(*n*)
  - Keep track of both N and U for each node

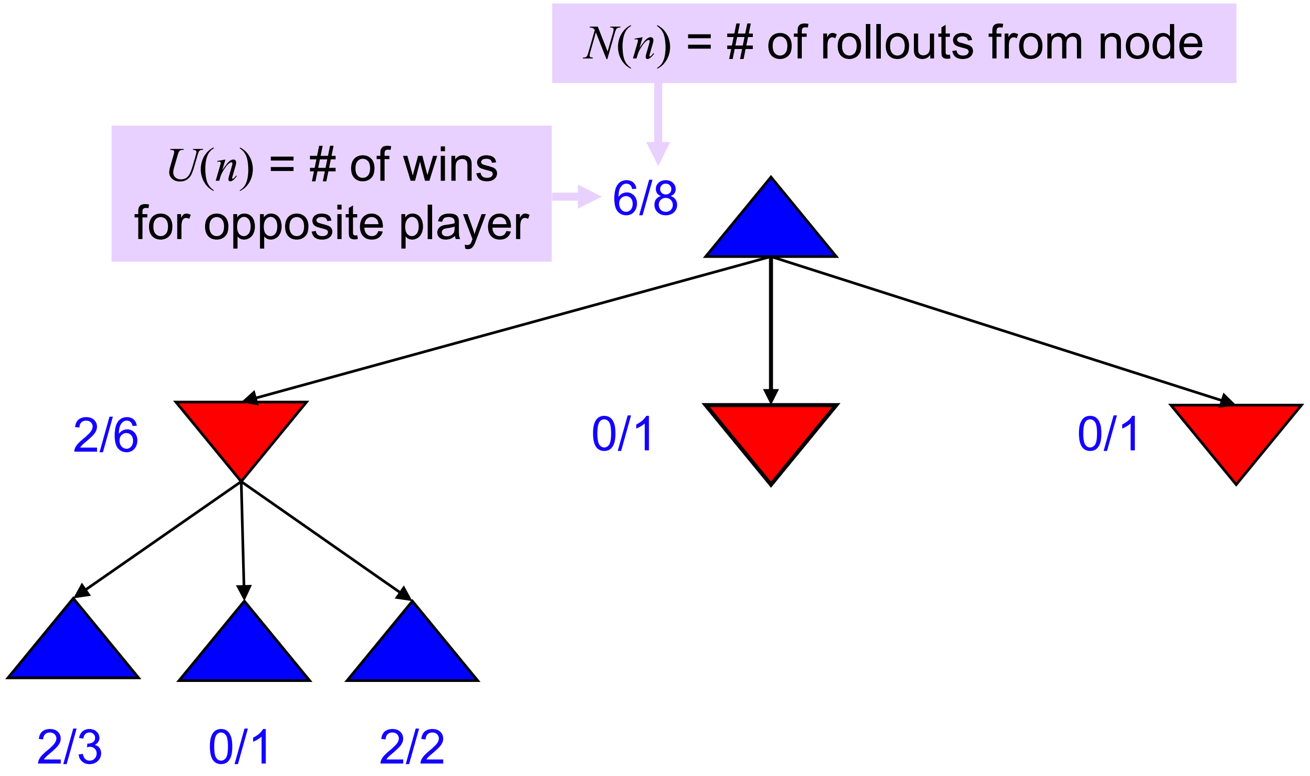
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



MCTS Algorithm

- Repeat until out of time:
  - **Selection**: recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



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

For 3 red nodes above the UCB values (with C=1) are:

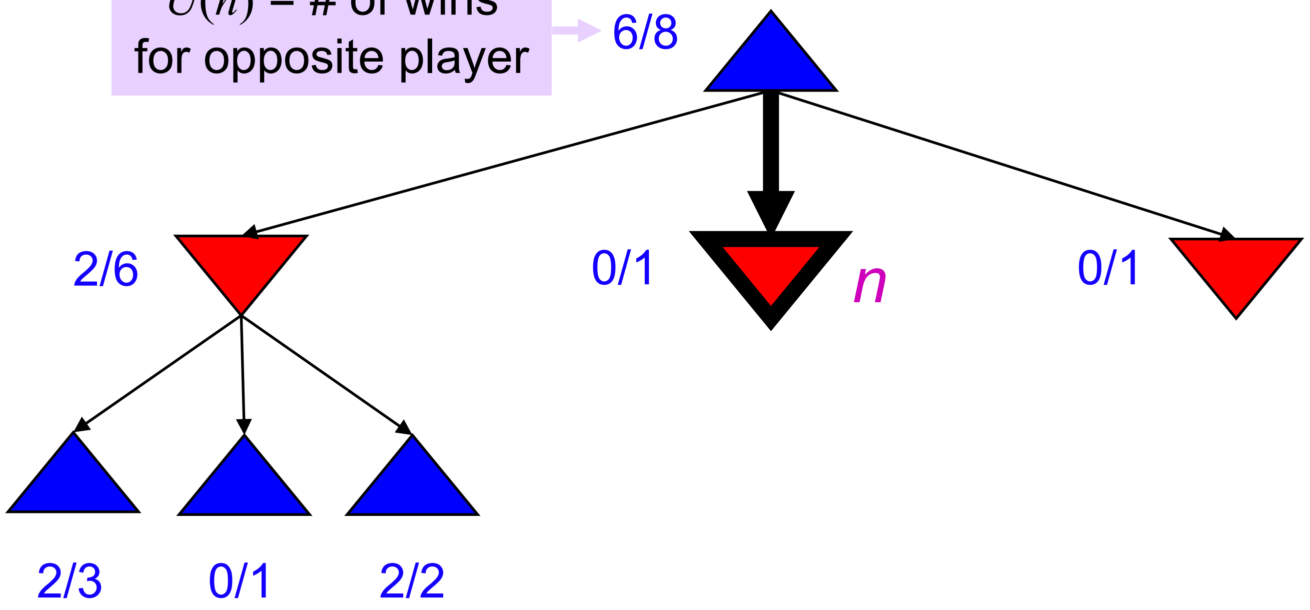
$$\frac{2}{6} + \sqrt{\frac{\log 8}{6}} \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}} \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}}$$

MCTS Algorithm

- Repeat until out of time:
  - **Selection**: recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root

$N(n)$ = # of rollouts from node

$U(n)$ = # of wins for opposite player



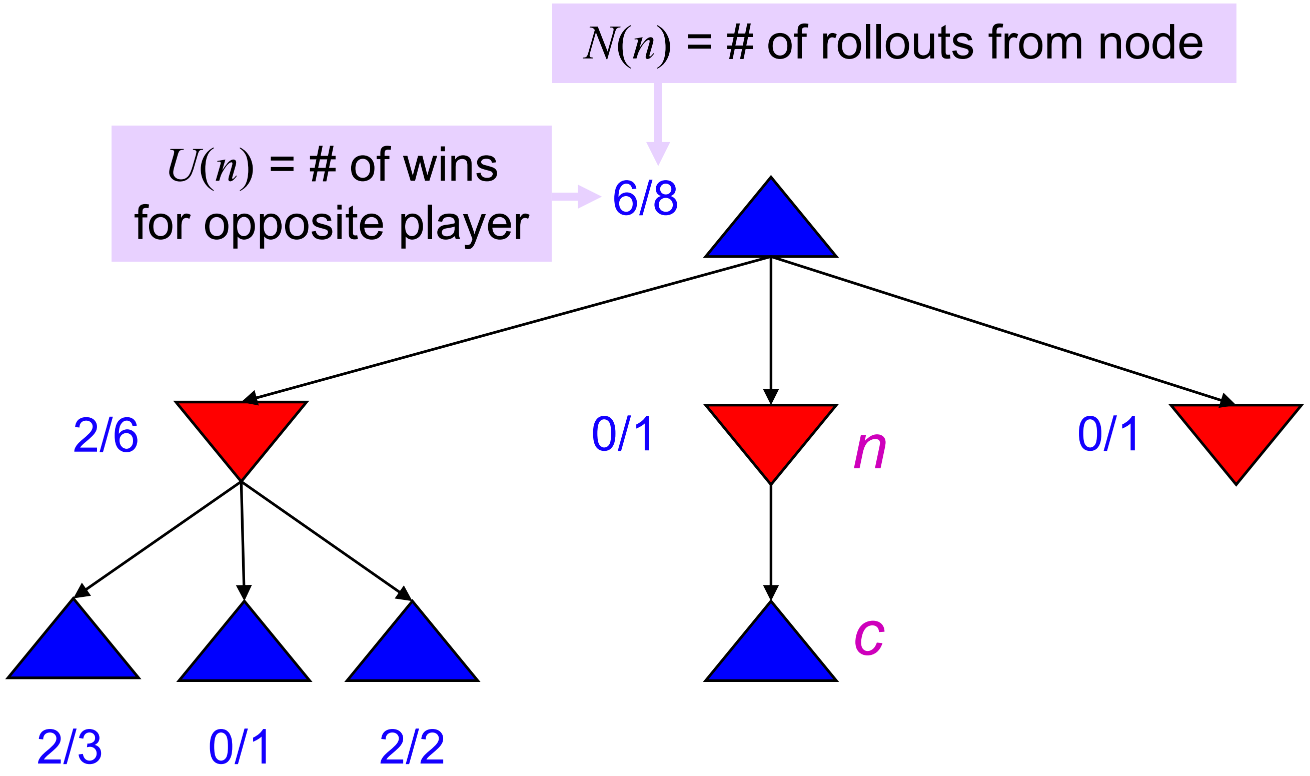
For 3 red nodes above the UCB values (with C=1) are:

$$\frac{2}{6} + \sqrt{\frac{\log 8}{6}} \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}} \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}}$$

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

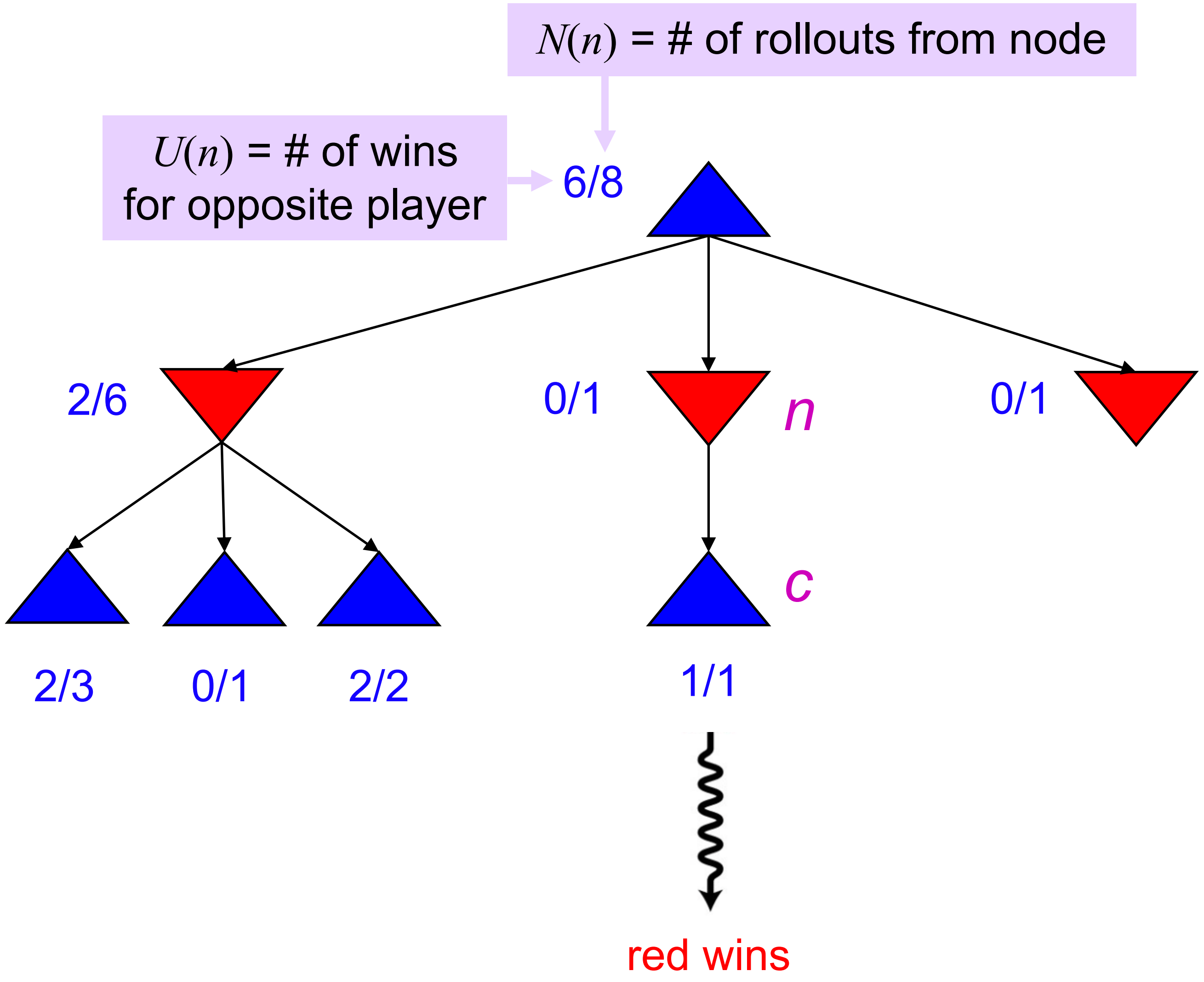
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion**: add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



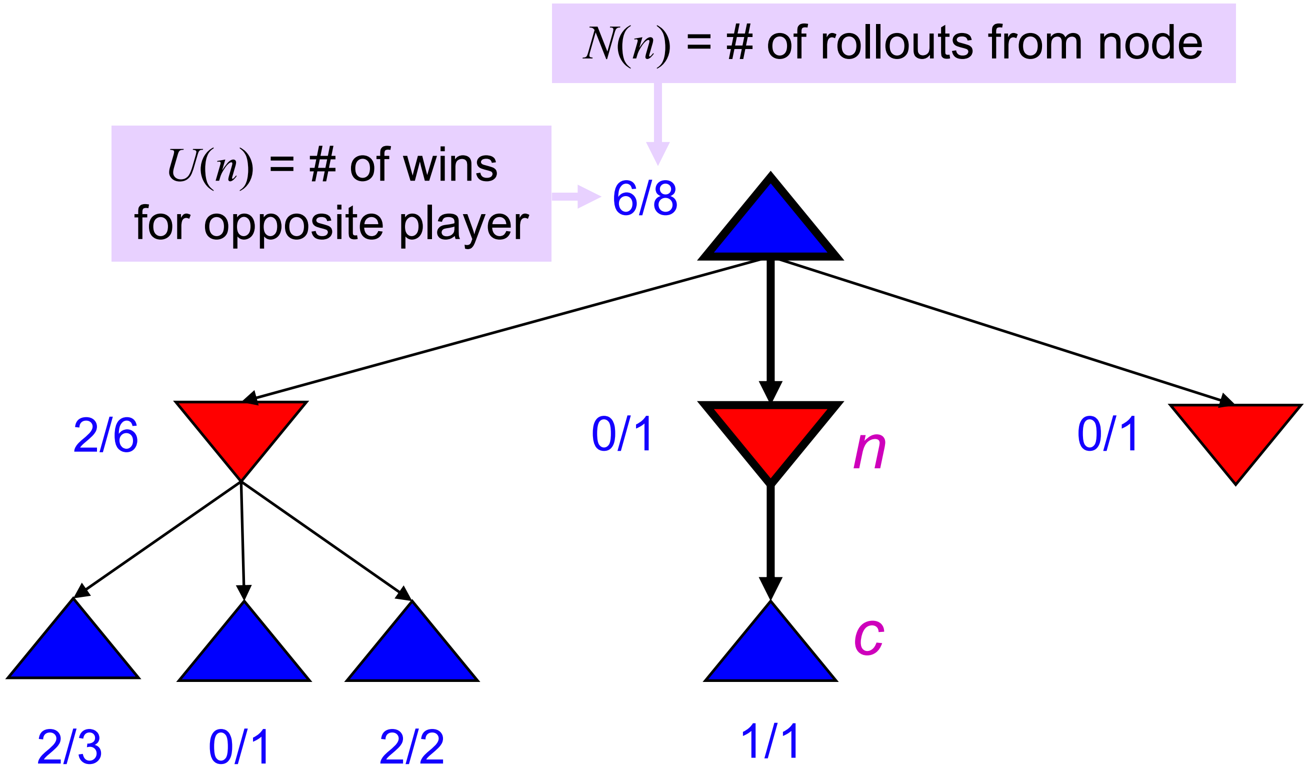
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node n
  - **Expansion:** add a new child c to n
  - **Simulation:** run a rollout from c
  - **Backpropagation:** update U and N counts from c back up to the root



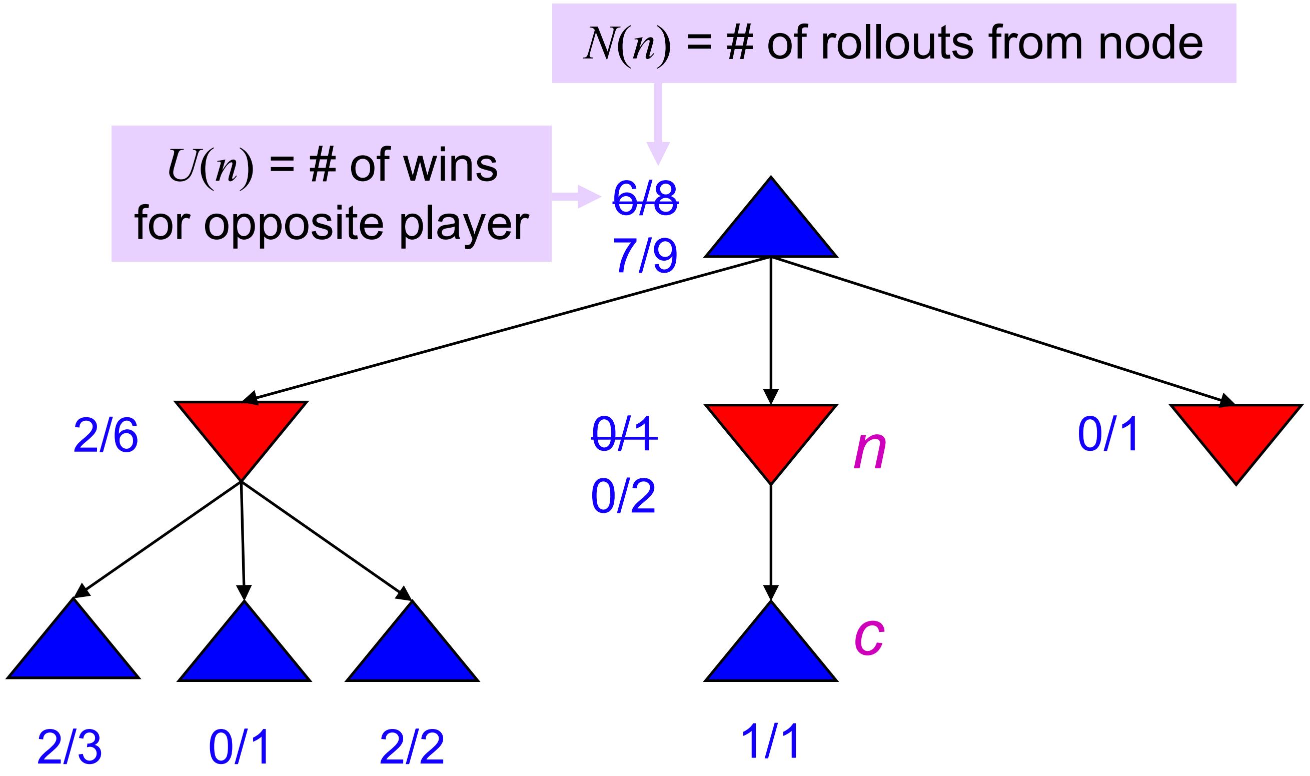
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation**: update U and N counts from *c* back up to the root



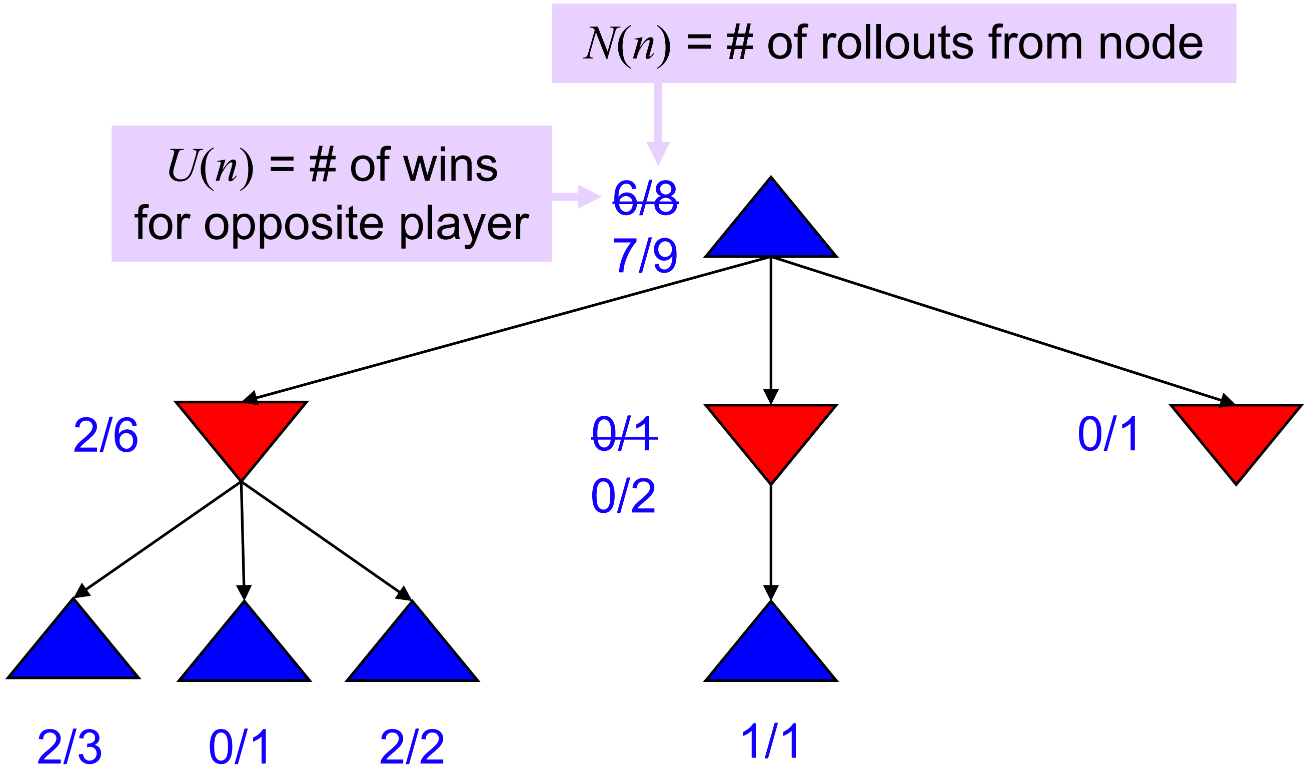
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



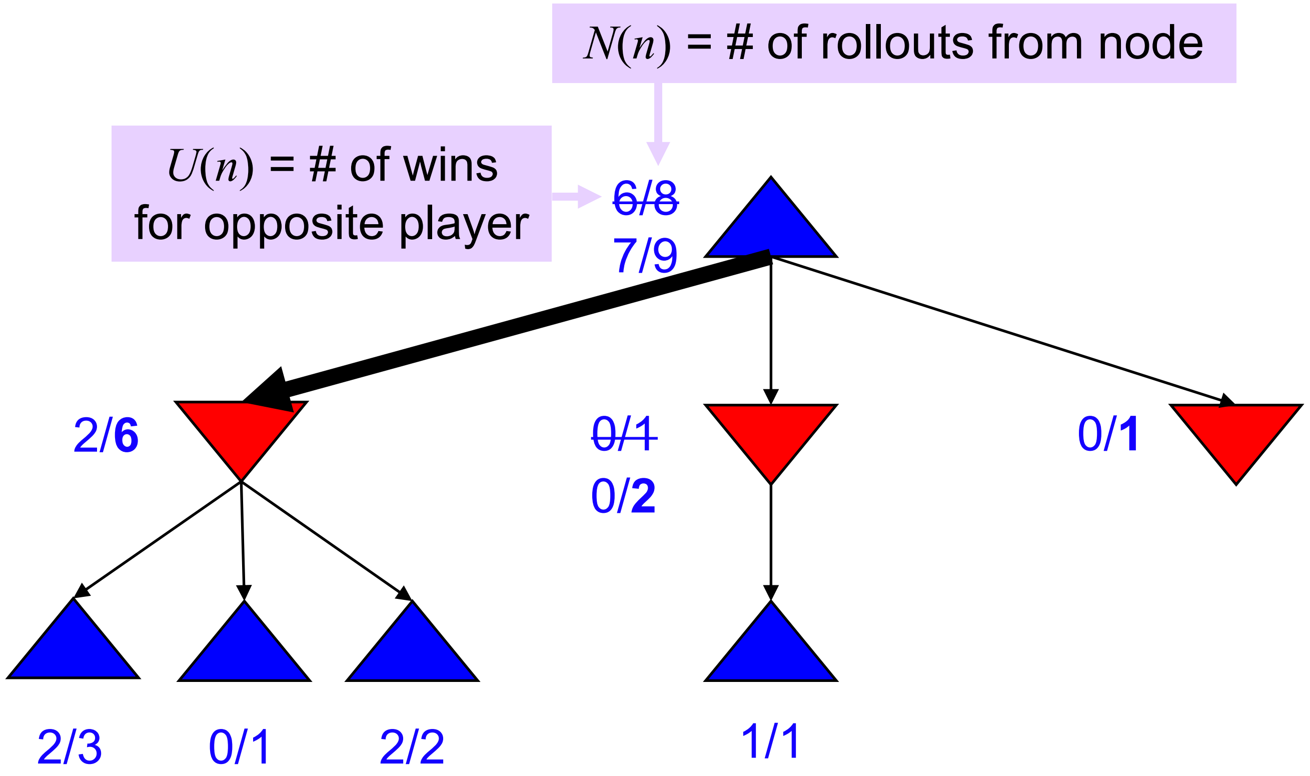
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root
- Choose the action leading to the child with highest *N*



MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root
- Choose the action leading to the child with highest *N*

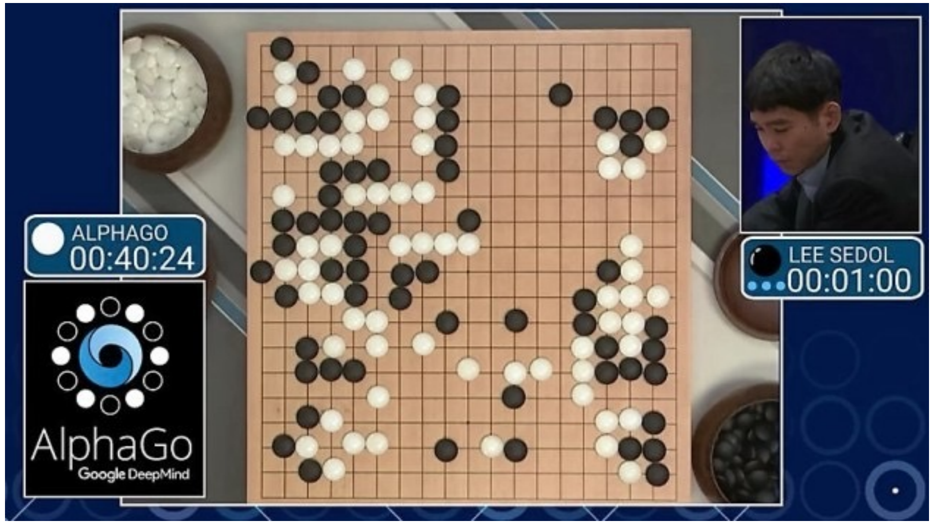


MCTS Summary

- MCTS is currently the most common tool for solving hard search problems
- Why?
  - Time complexity independent of b and m
  - No need to design evaluation functions (general-purpose & easy to use)
- Solution quality depends on number of rollouts N
  - Theorem: as $N \to \infty$ UCT selects the minimax move
- Example of using random sampling in an algorithm
  - Broadly called *Monte Carlo* methods
- MCTS can be improved further with machine learning

MCTS + Machine Learning: AlphaGo

- Monte Carlo Tree Search with additions including:
  - Rollout policy is a neural network trained with reinforcement learning and expert human moves
  - In combination with rollout outcomes, use a trained value function to better predict node's utility



[Mastering the game of Go with deep neural networks and tree search. Silver et al. Nature. 2016]

What we did today

- Extended games to include uncertain outcomes
- Modified search to reason about uncertain outcomes
  - Return *expected value* for a chance node
- Saw impact of a mismatch between model and reality in planning
  - Agent may be overly optimistic or pessimistic
  - Issue that comes up frequently in AI applications
- Saw *Monte Carlo Tree Search* algorithm
  - Practical and an example of using random sampling in an algorithm

Next Time: MDPs and Reinforcement Learning!