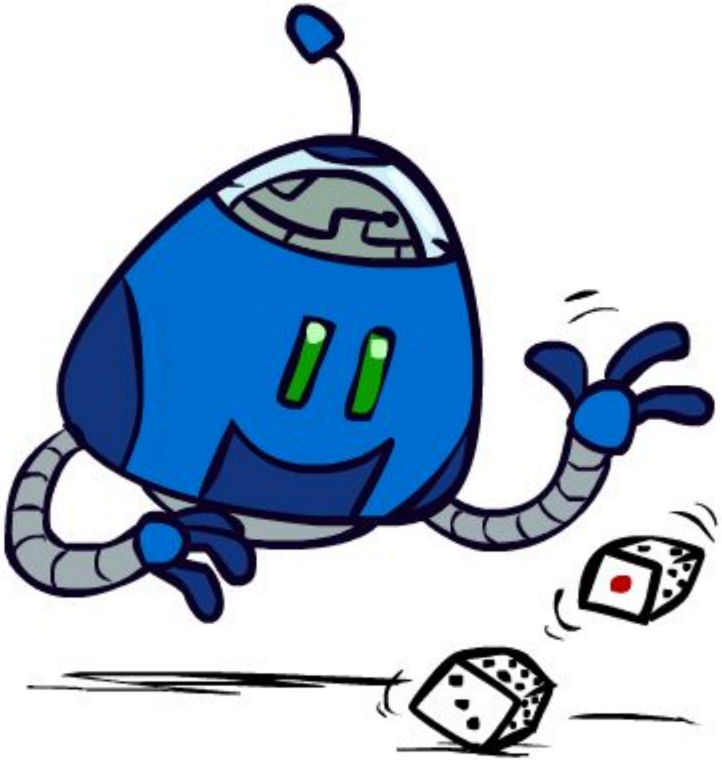
Artificial Intelligence

Lec 12: Adversarial Search (contd.)

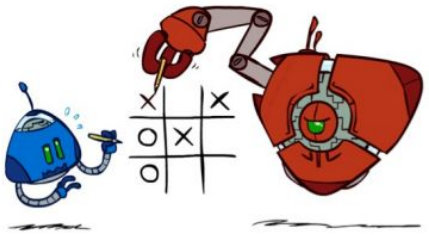
Pratik Mazumder

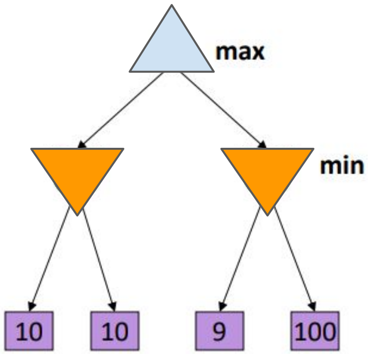
Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Uncertain Outcomes

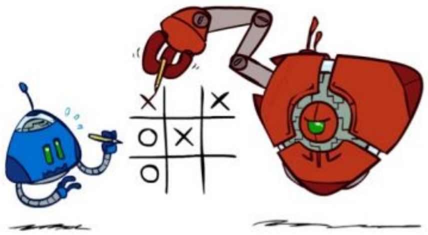


Worst-Case vs. Average Case

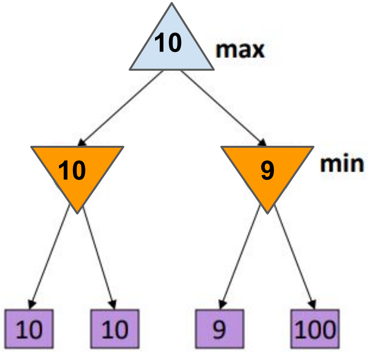




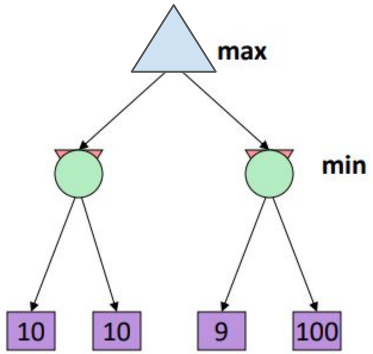
Worst-Case vs. Average Case



Ideal/Rational Adversary



Worst-Case vs. Average Case



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

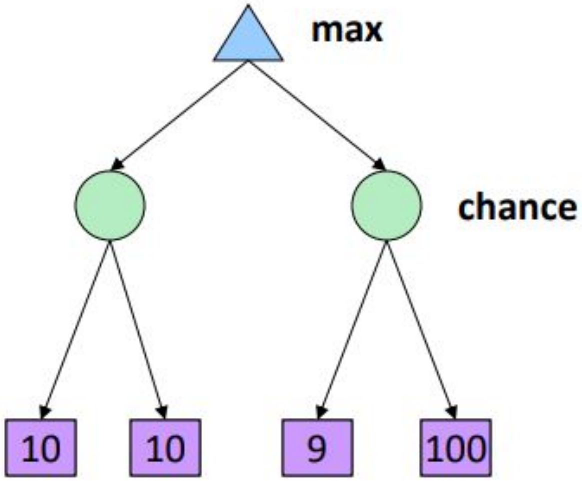
Or the adversary is naive and plays randomly moves.

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 in Pacman.
  - 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 are the same as in minimax search.
  - Chance nodes are like min nodes, but the outcome is uncertain.
  - Calculate their expected utilities
    - i.e., take the weighted average (expectation) of children.



Expectimax Search

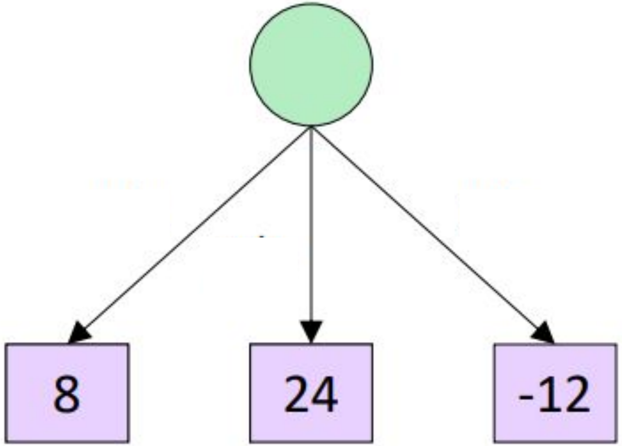
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

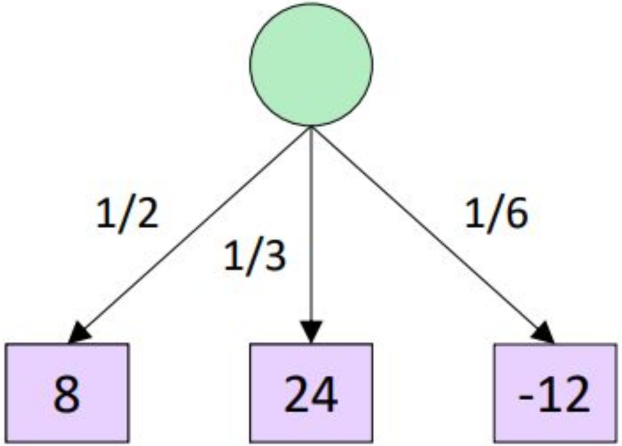
Expectimax Search

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



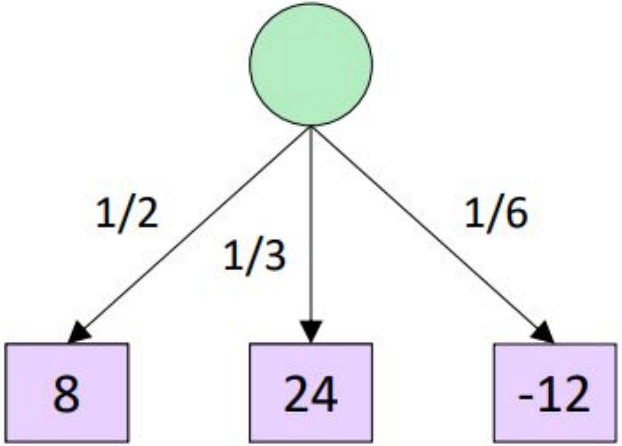
Expectimax Search

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



Expectimax Search

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



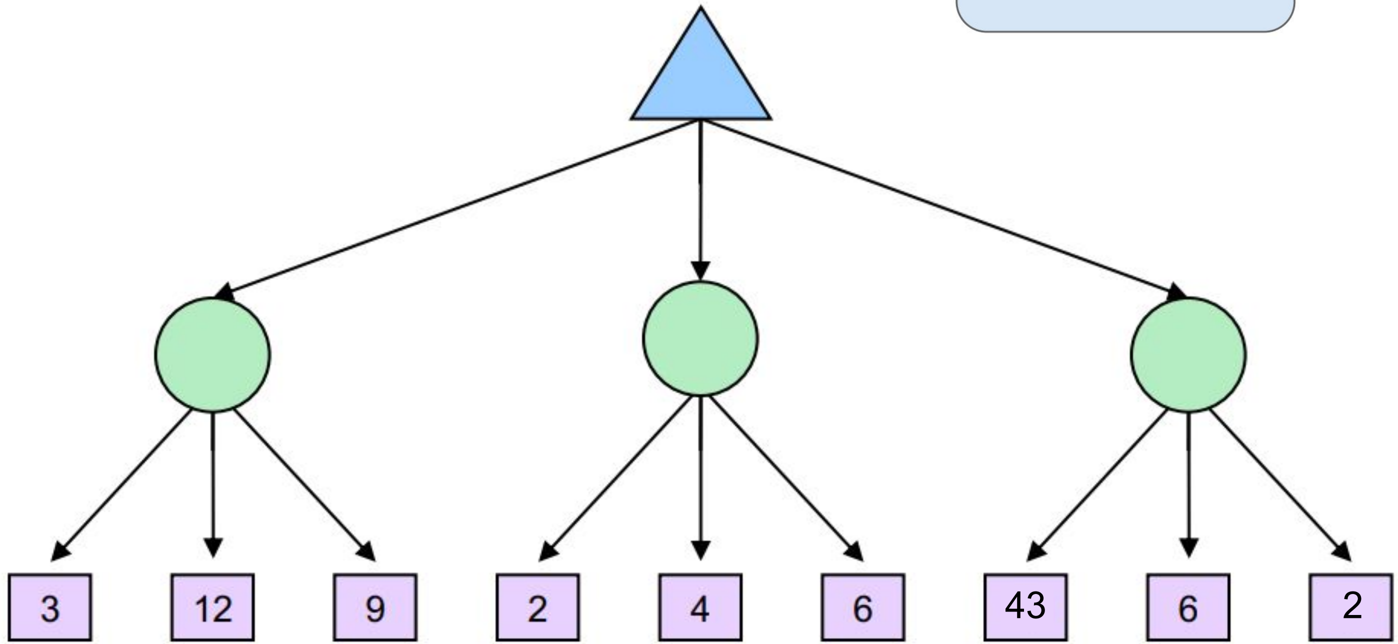
v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

Expectimax Search: Example

```
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
```

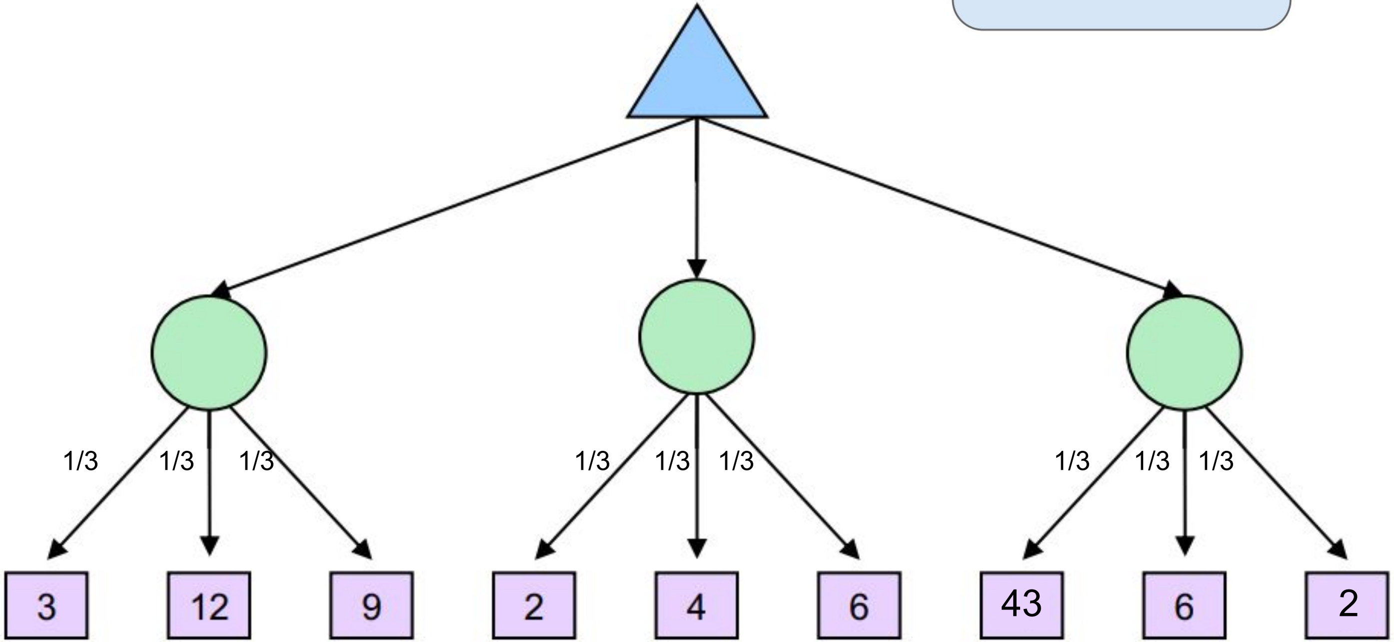


Expectimax Search: Example

```
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
```

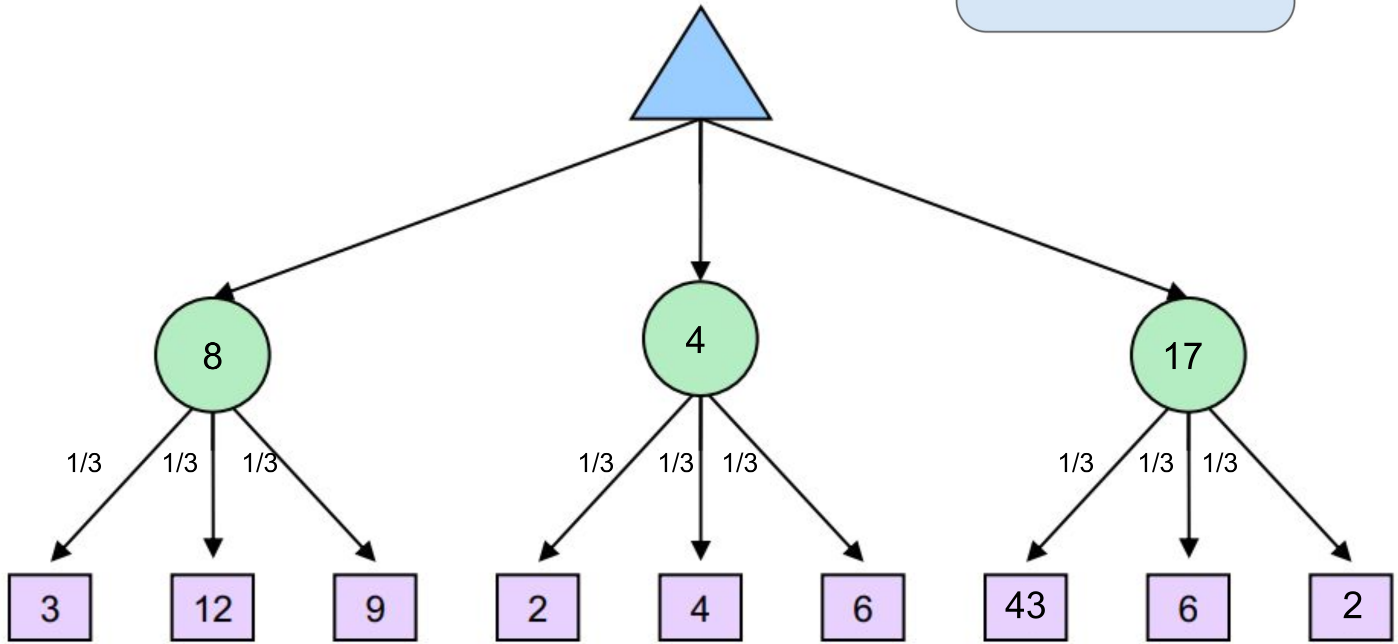


Expectimax Search: Example

```
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
```

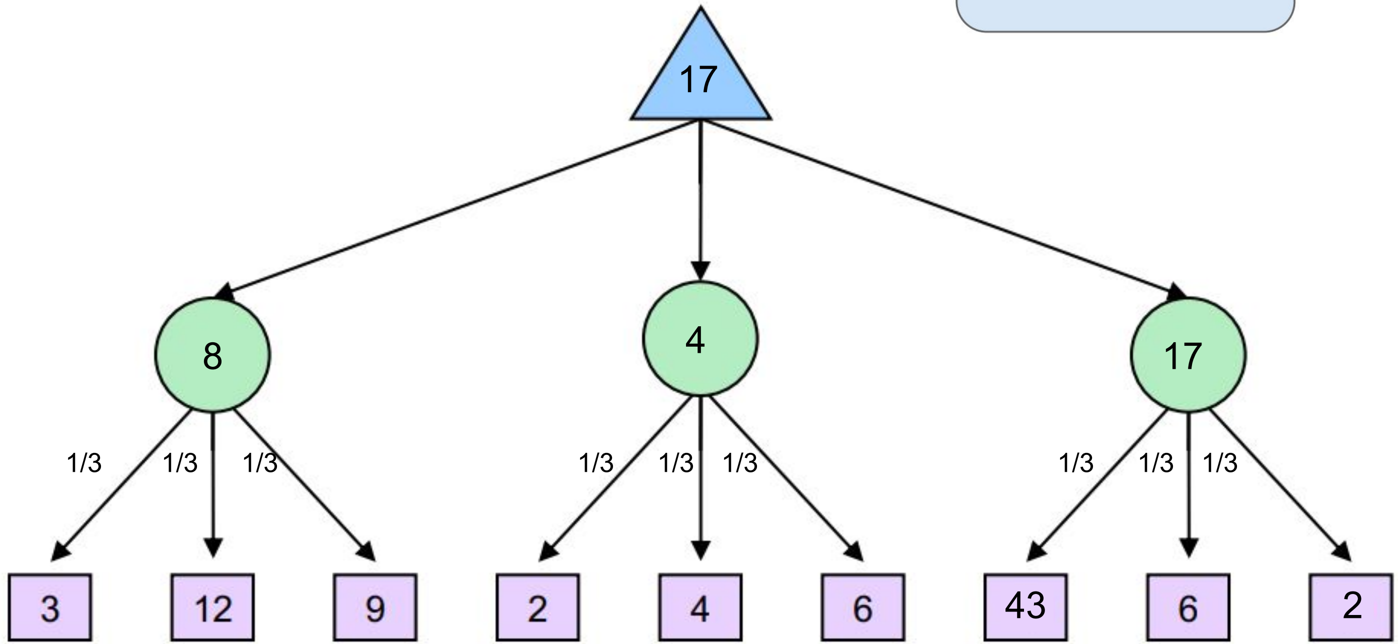


Expectimax Search: Example

```
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
```

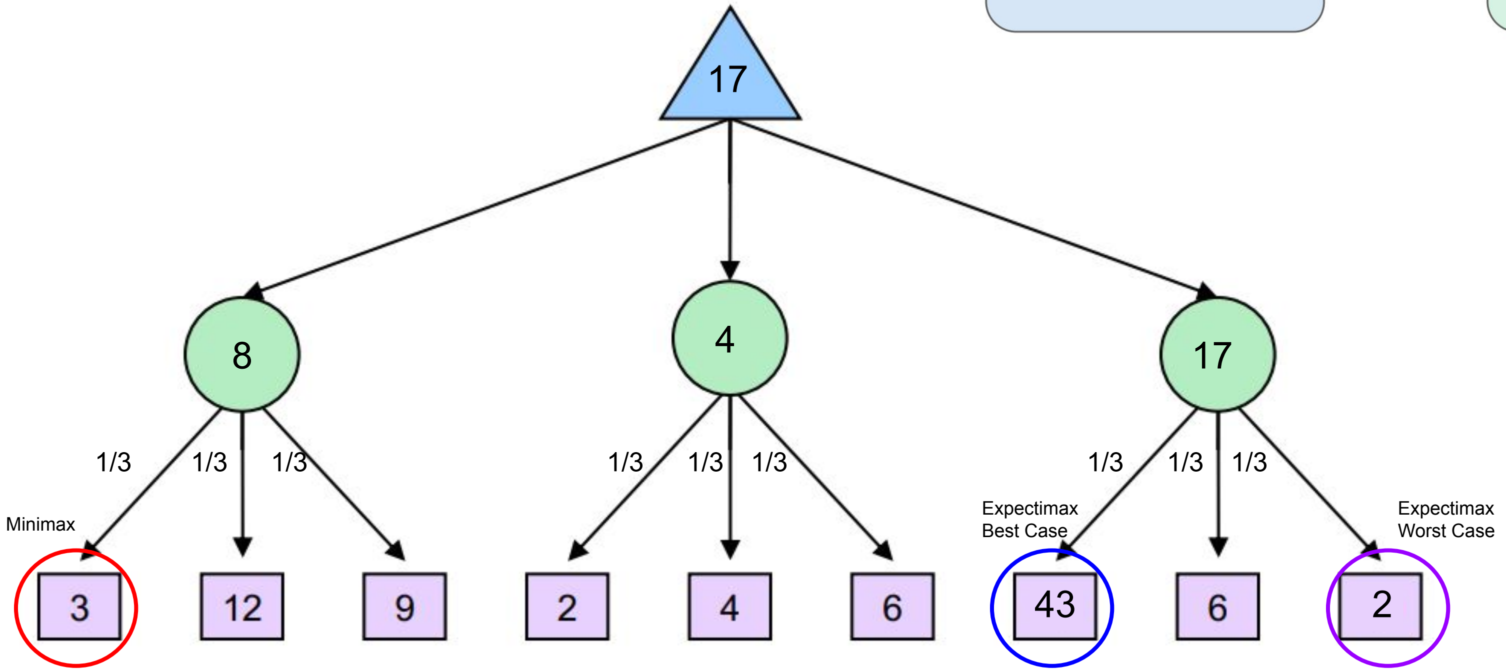


Expectimax Search: Example

```
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)
```

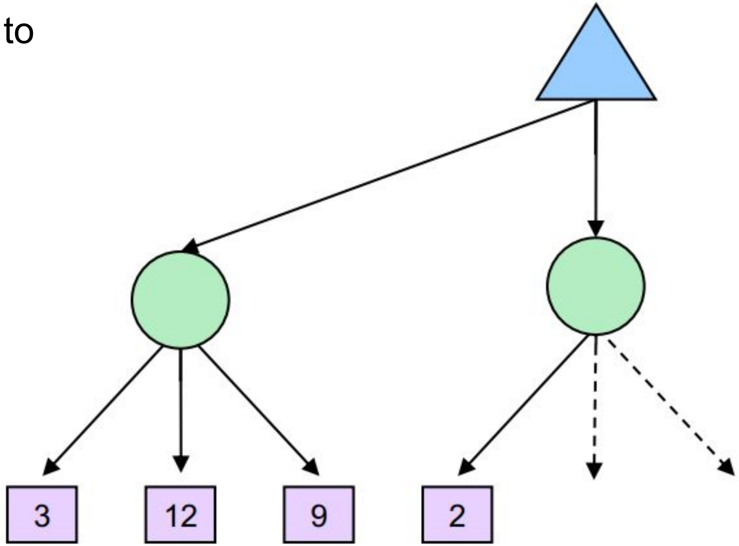
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

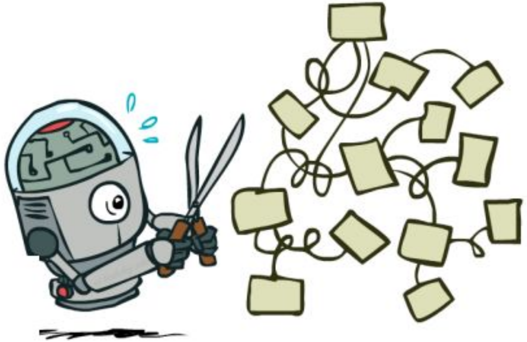
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



Expectimax Pruning?

Can we prune branches similar to what we did before?

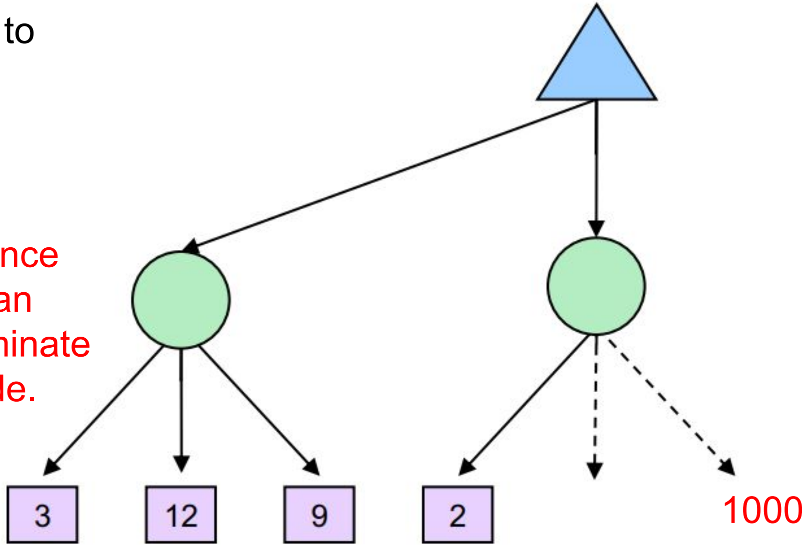


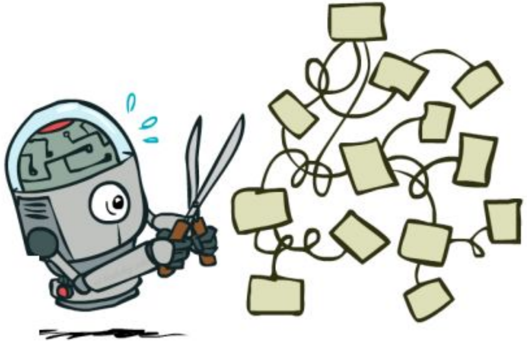


Expectimax Pruning?

Can we prune branches similar to what we did before?

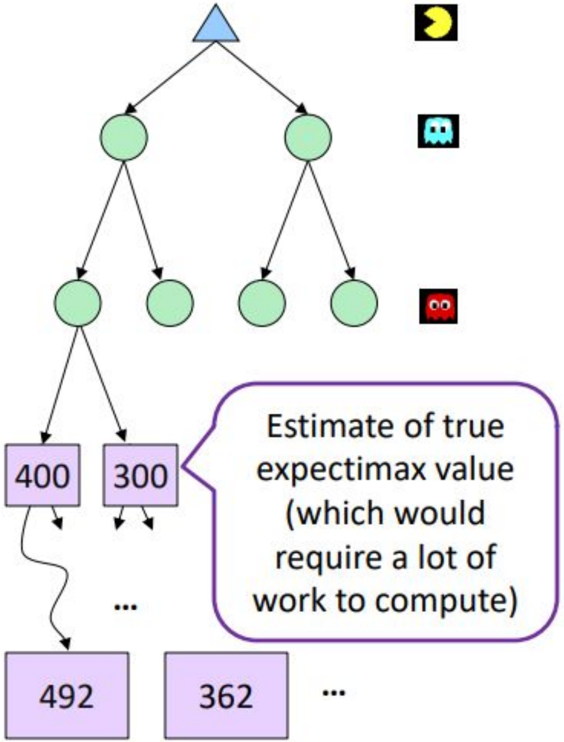
We cannot use naive pruning since one of the branches may have an extremely high value which dominate the expectimax value of the node.





Depth-Limited Expectimax

Can we do a depth-limited search?

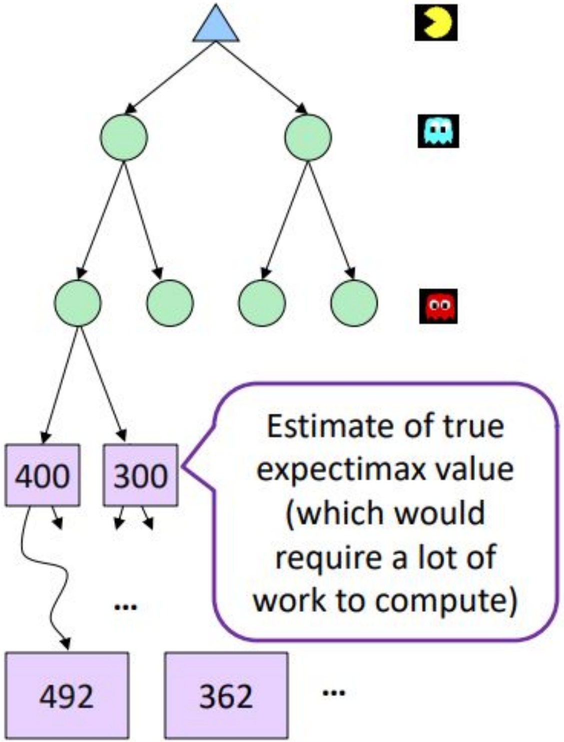


Depth-Limited Expectimax

Can we do a depth-limited search?

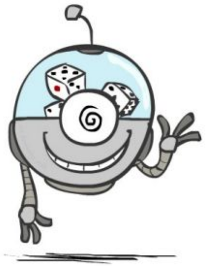
Yes.

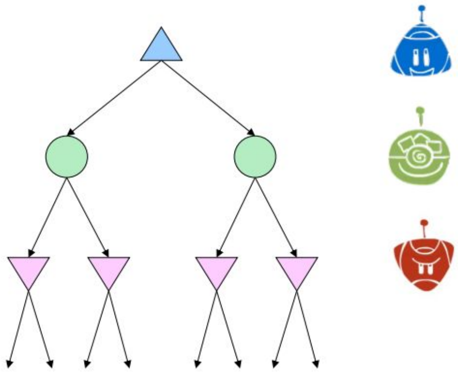
But evaluation functions may not be good. No guarantee of working well.



Mixed Layer Types

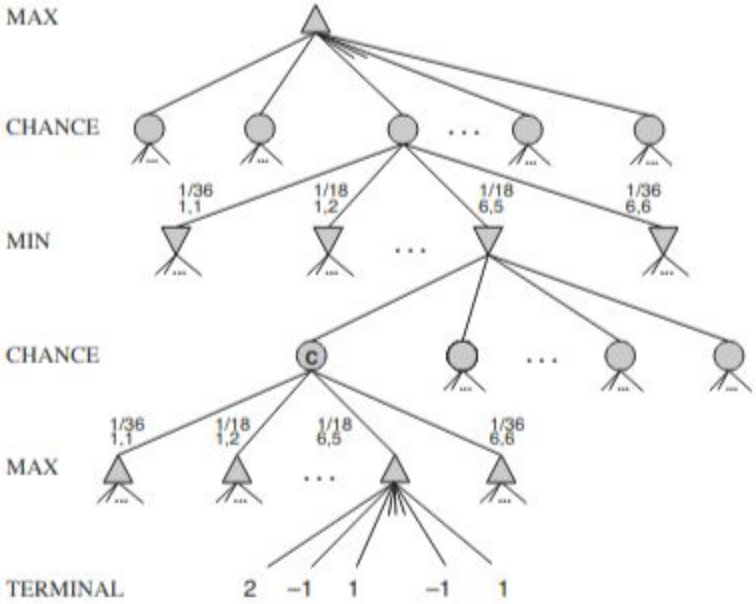
- Expectiminimax, e.g., Backgammon
- Environment is an extra “random agent” player that moves after each min/max agent.
- Each node computes the appropriate combination of its children.



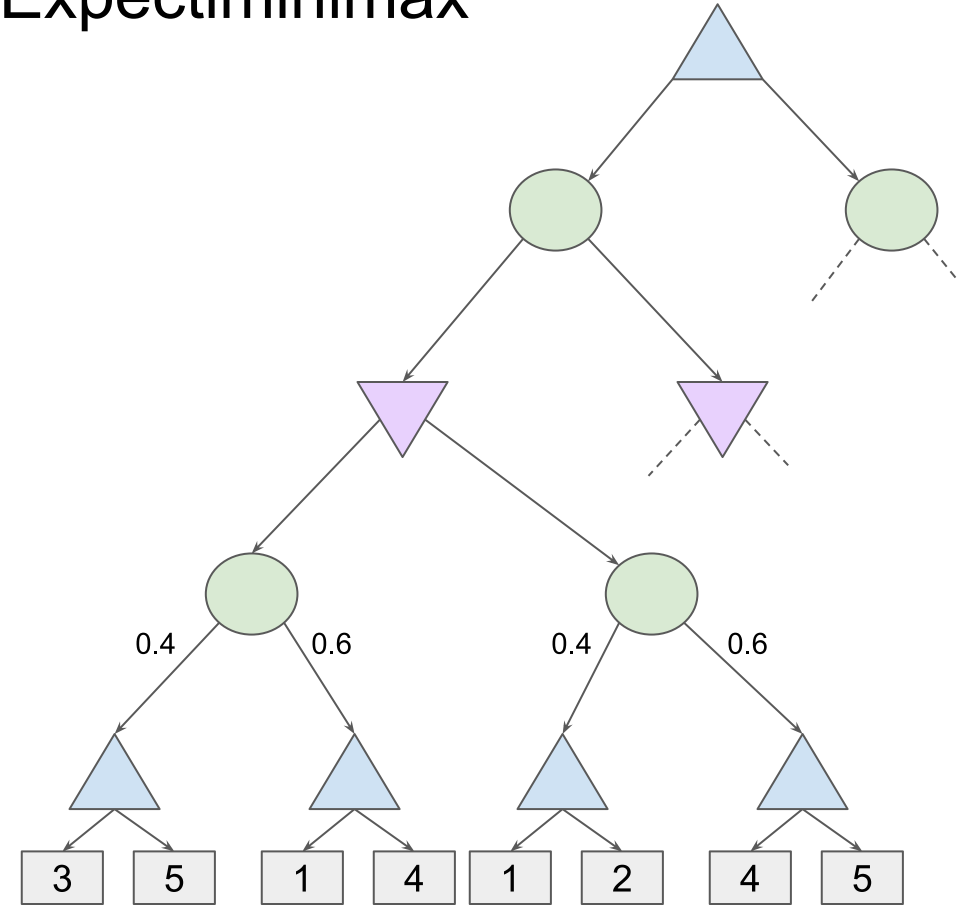


Expectiminimax

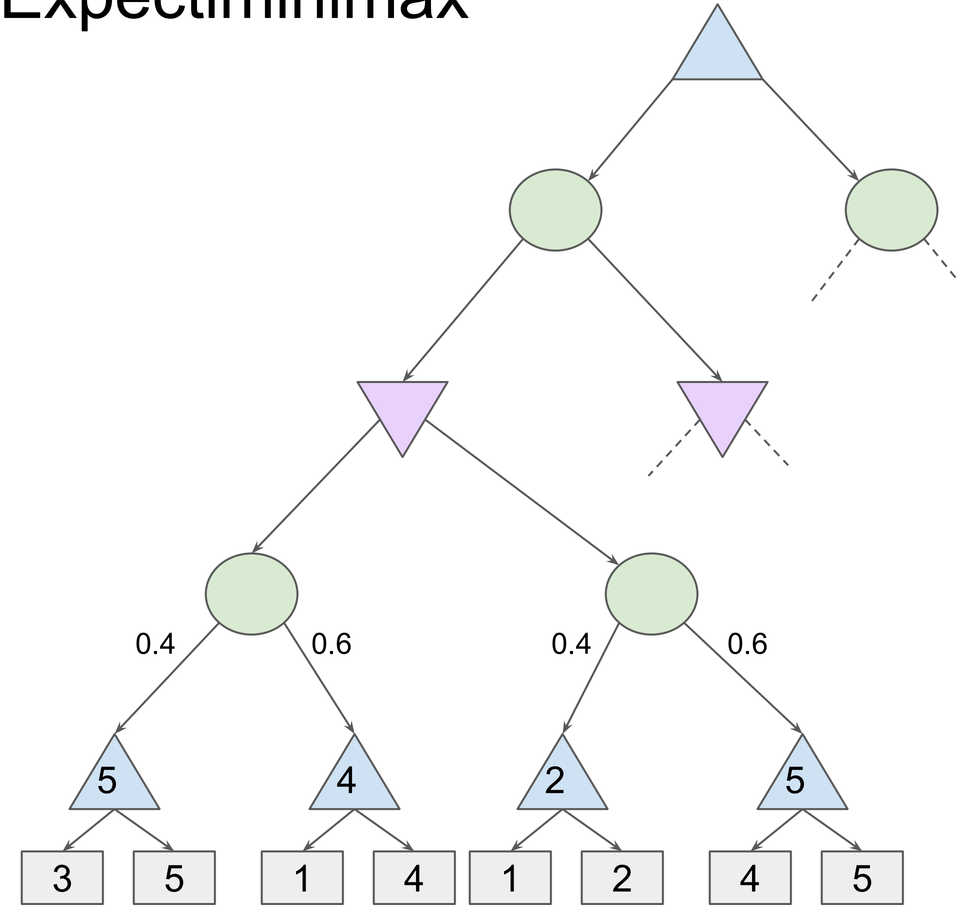
- Max nodes are the same as in minimax search.
- Min nodes are the same as in minimax search.
- Chance nodes can succeed both Max and Min nodes, but the outcome is uncertain.
    - Calculate their expected utilities, i.e., take the weighted average (expectation) of children.



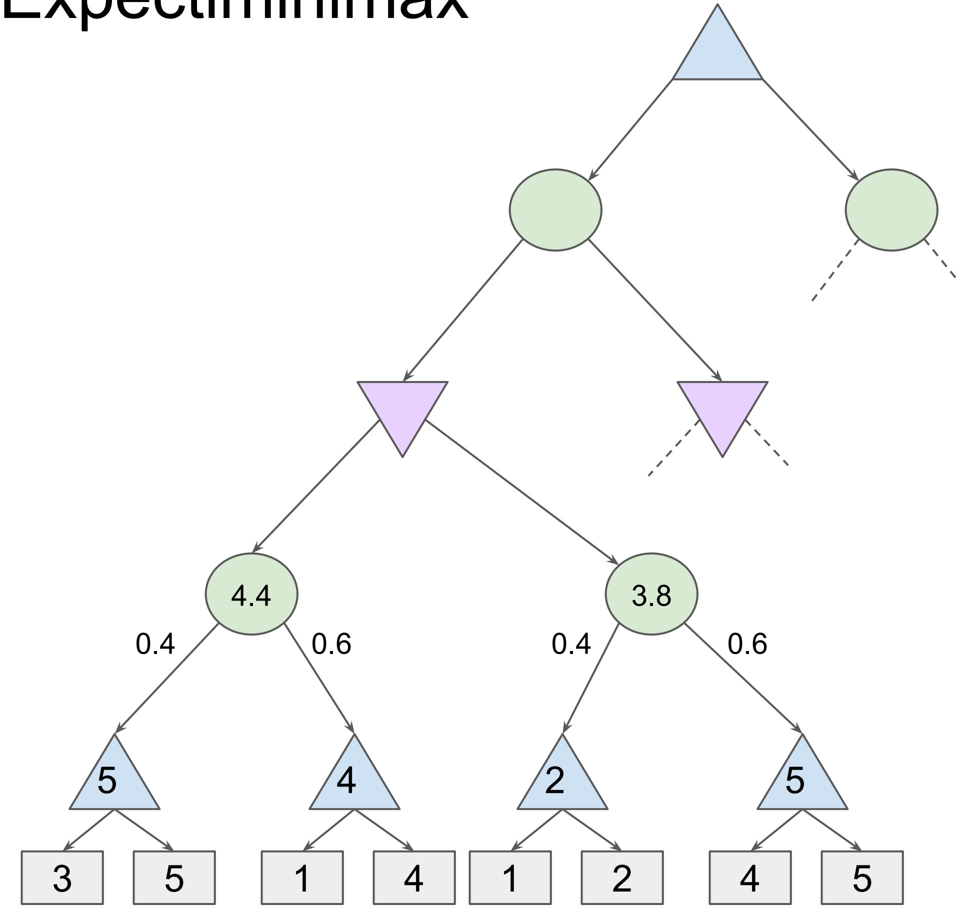
Expectiminimax



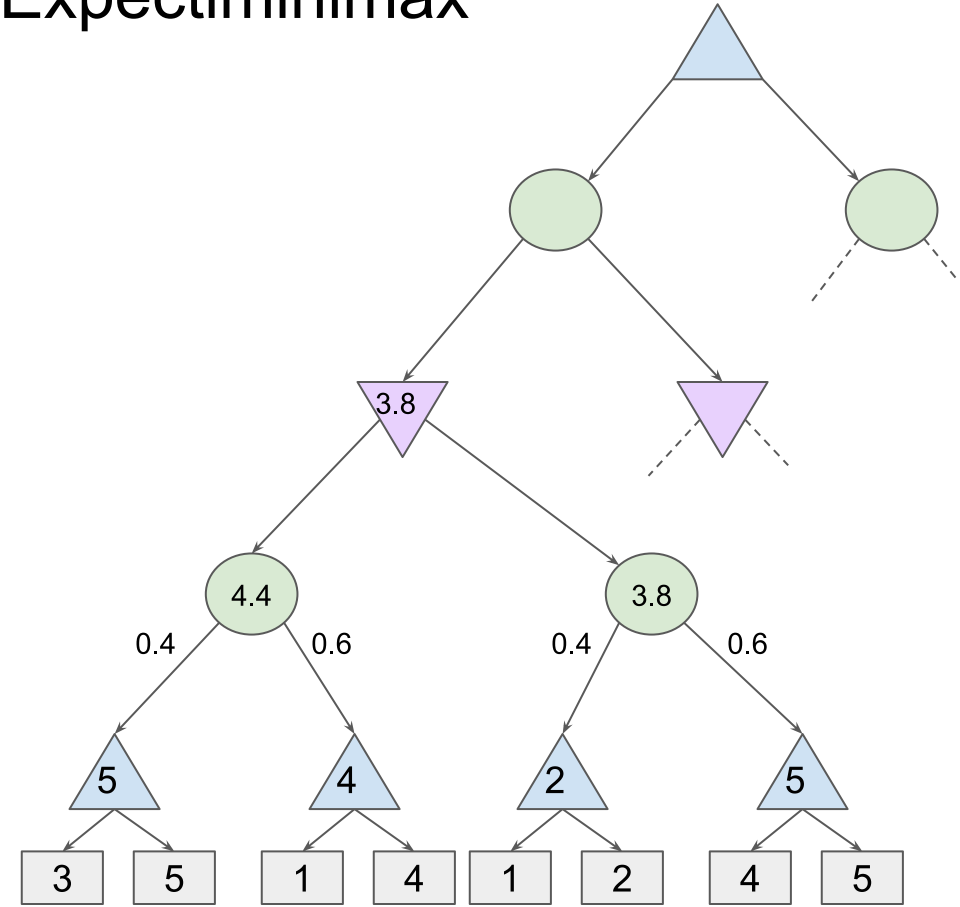
Expectiminimax



Expectiminimax



Expectiminimax



Modeling Assumptions

The Dangers of Optimism and Pessimism

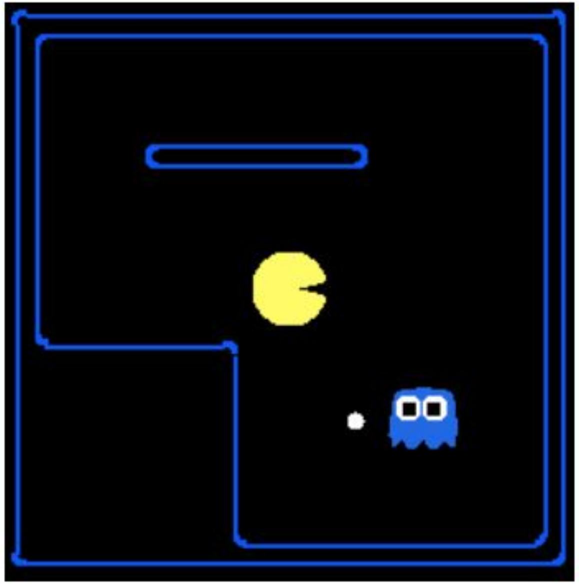
Dangerous Optimism

Assuming chance when the world is adversarial

Dangerous Pessimism

Assuming the worst case when it's not likely

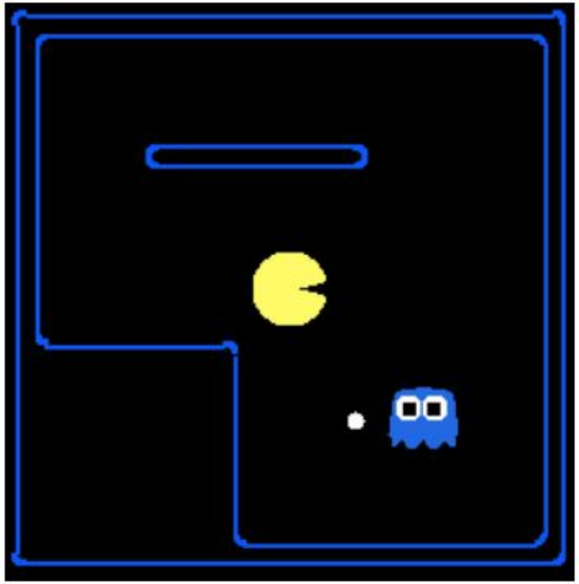
Assumptions vs. Reality



	Adversarial Ghost	Random Ghost
Minimax Pacman		
Expectimax Pacman		

Results from playing 5 games

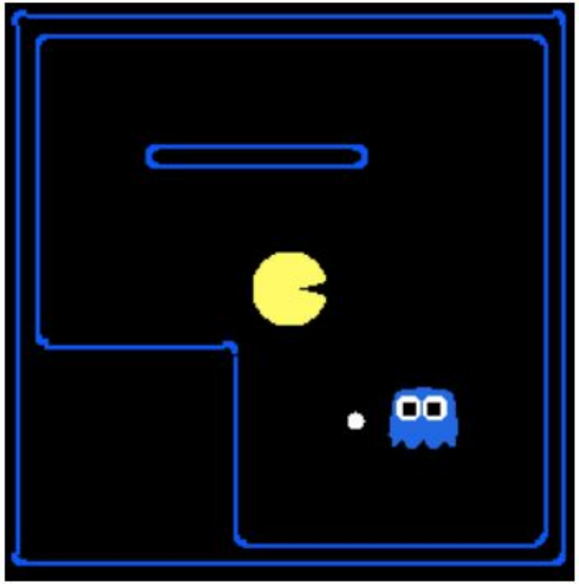
Assumptions vs. Reality



	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	
Expectimax Pacman		Won 5/5 Avg. Score: 503

Results from playing 5 games

Assumptions vs. Reality

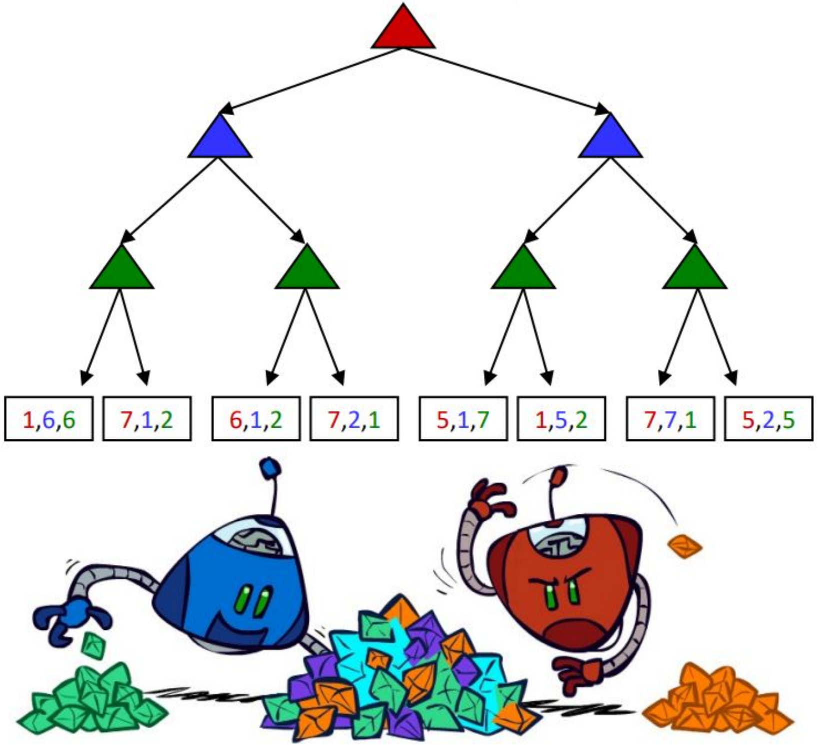


	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

Results from playing 5 games

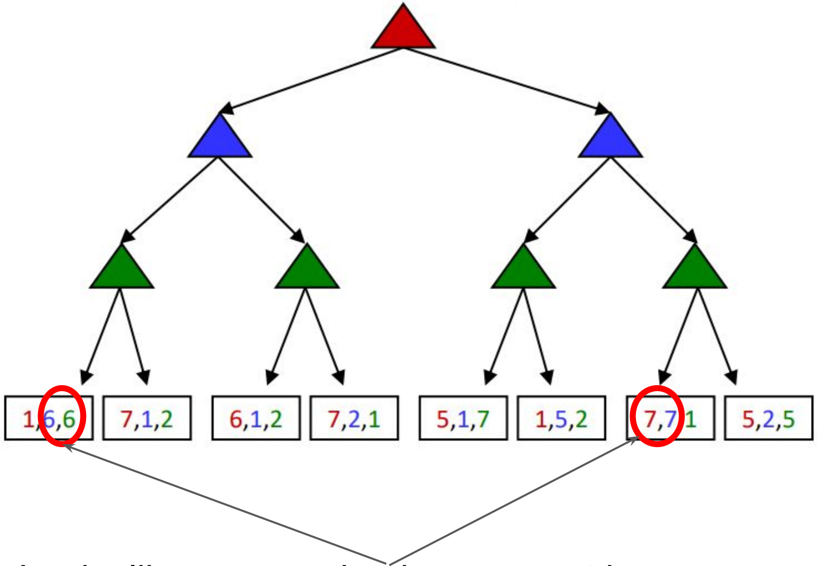
Multi-Agent Utilities

- What if the game is not zero-sum, or has multiple playe
- 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.



Multi-Agent Utilities

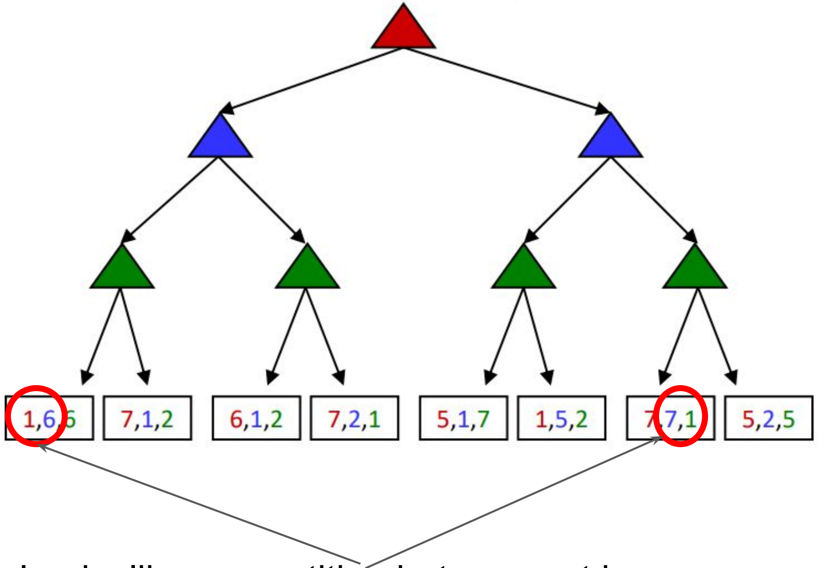
- What if the game is not zero-sum, or has multiple playe
- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically.



Looks like cooperation but may not be so

Multi-Agent Utilities

- What if the game is not zero-sum, or has multiple playe
- 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.

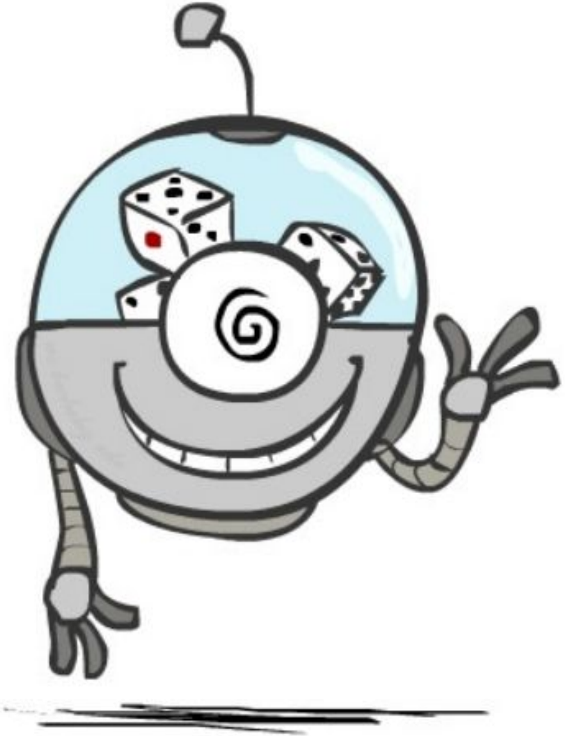


Looks like competition but may not be so

Probabilities and Randomness in Algorithm Design

- Till now, we have considered some randomness in the way the opponent or environment behaves or acts.

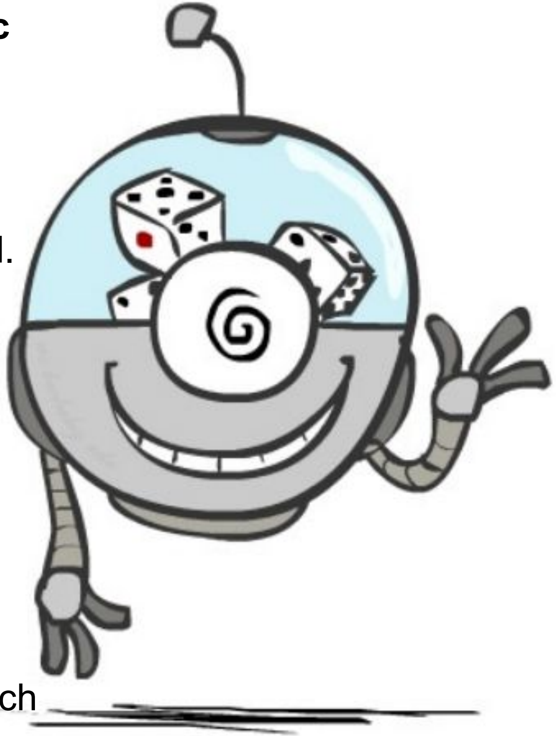
- We will now consider randomness in the algorithm design, which may help to better solve the problems at hand.



Monte Carlo Tree Search (MCTS)

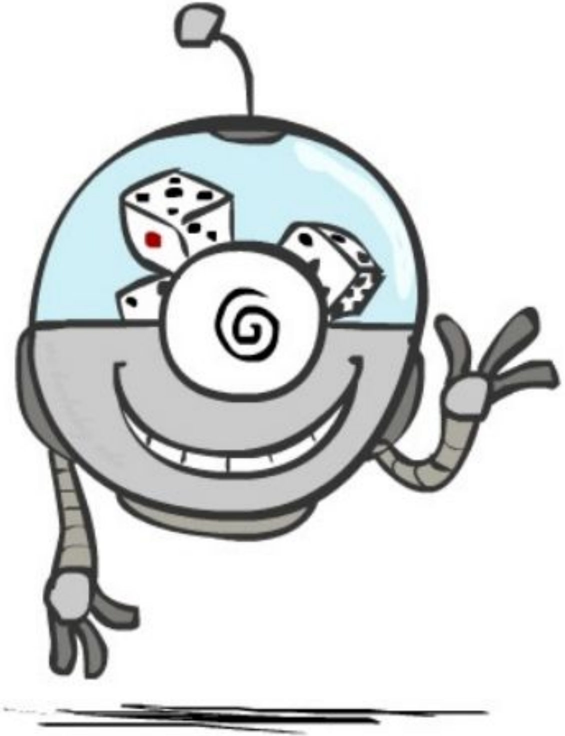
The basic Monte Carlo Tree Search (MCTS) strategy **does not use** a **heuristic evaluation function**.
The **value of a state** is estimated as the **average utility over** the **number of simulations**.

- **Playout**: simulation that chooses moves until terminal position is reached.
- **Selection**: Start from root, choose move (selection policy) repeatedly, moving down tree.
- **Expansion**: Search tree grows by generating a new explored child of the selected node.
- **Simulation**: playout from generated child node
- **Back-propagation**: use the result of the simulation to update all the search tree nodes going up to the root.



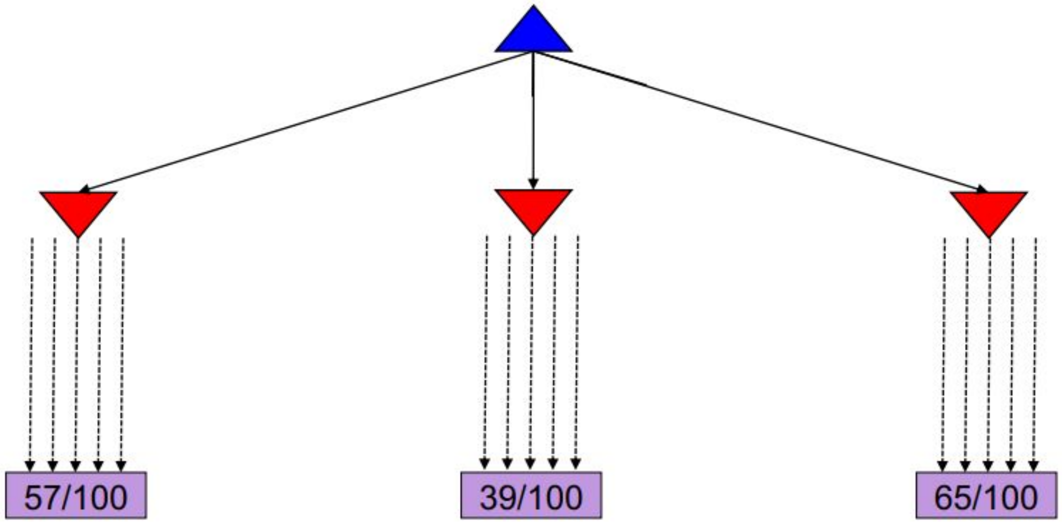
Monte Carlo Tree Search: Rollouts/Playout

- For each rollout/playout:
  - Repeat until terminal:
    - Play a move according to a fixed, fast rollout policy (may be random).
  - Record the result.
- Fraction of wins/average reward correlates with the true value of the position!
- Having a “better” rollout policy helps



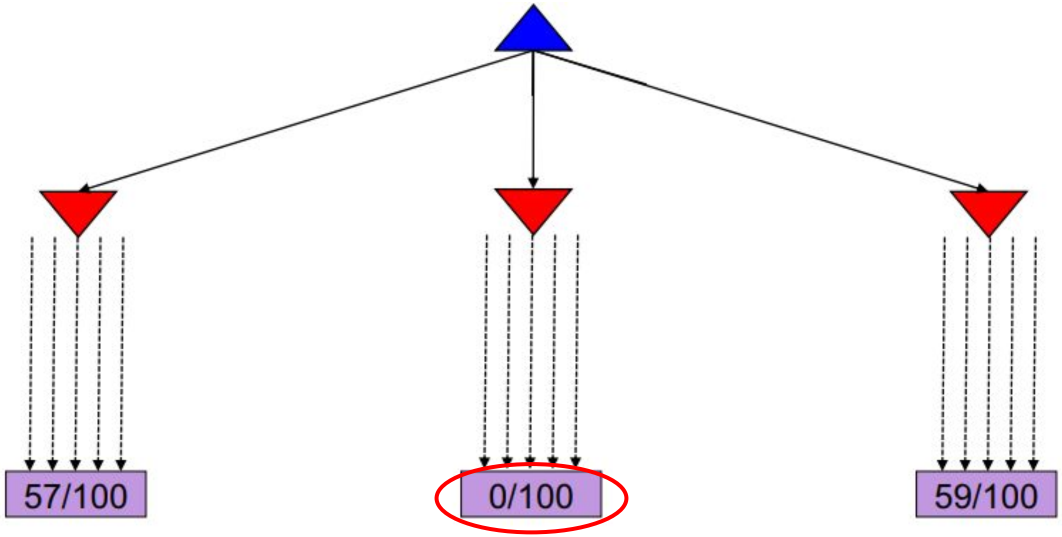
Monte Carlo Tree Search

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



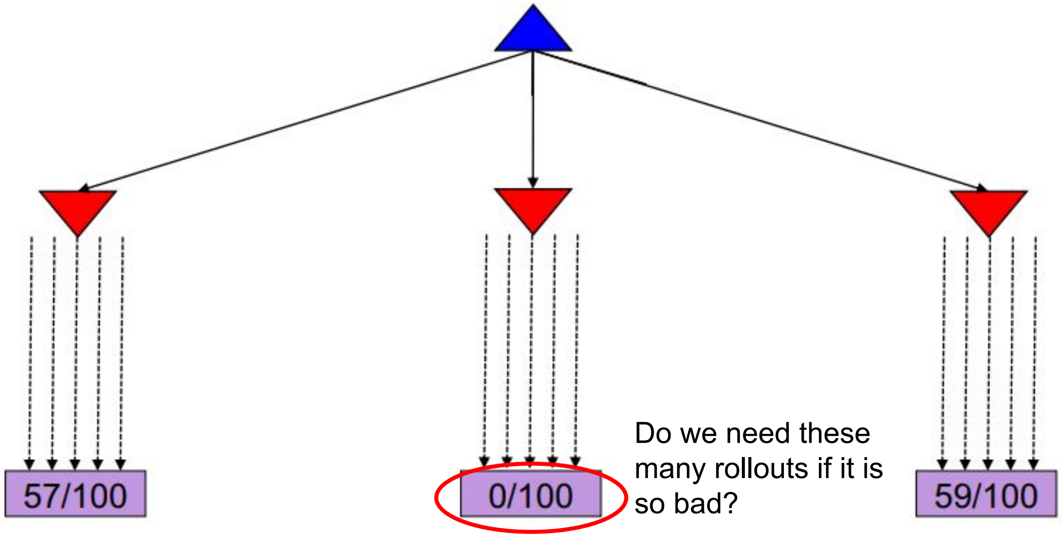
Monte Carlo Tree Search

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



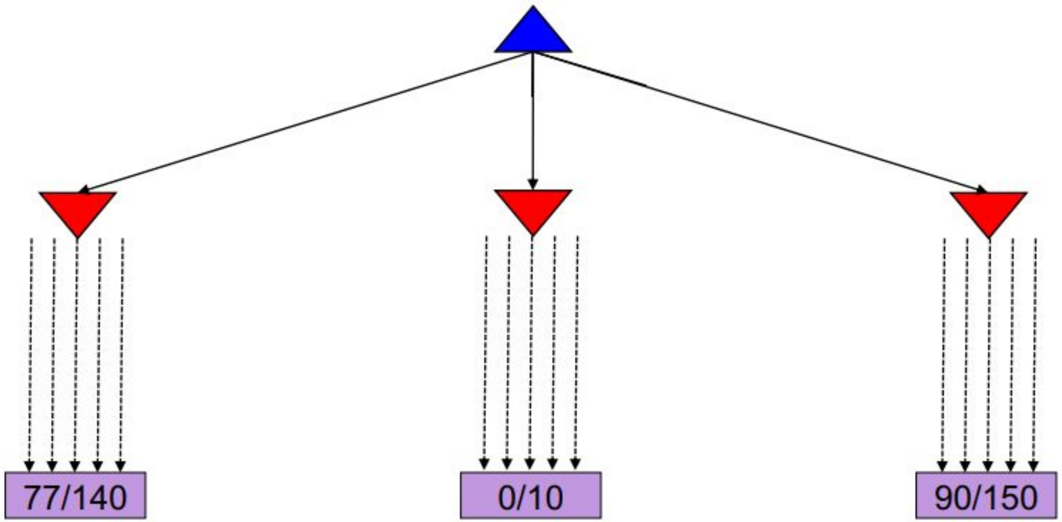
Monte Carlo Tree Search

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



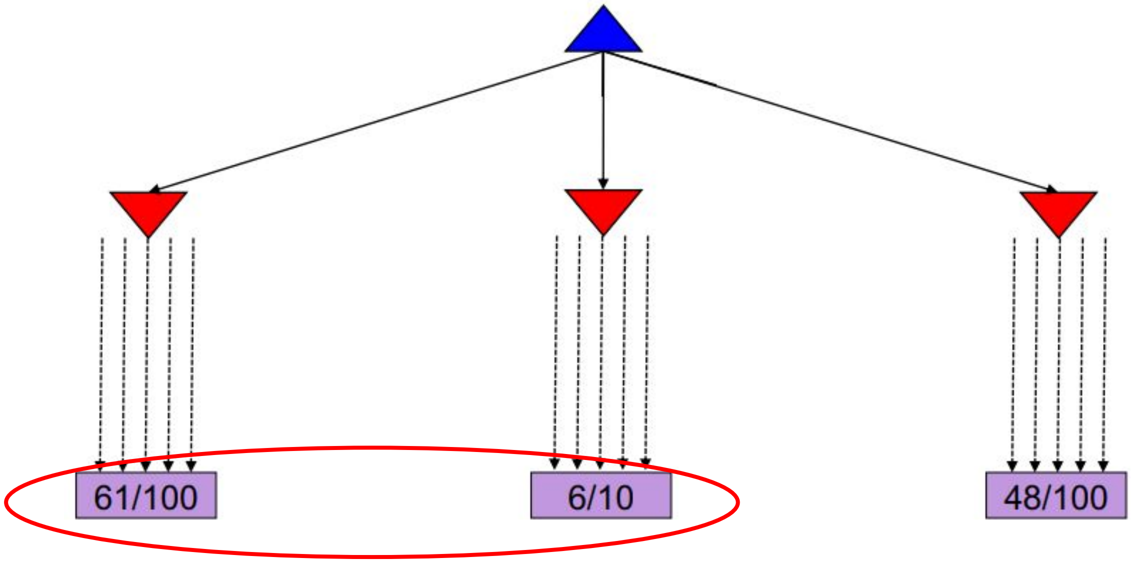
Monte Carlo Tree Search

- Let's have another version where we are more economical.
- Allocate rollouts to more promising nodes.



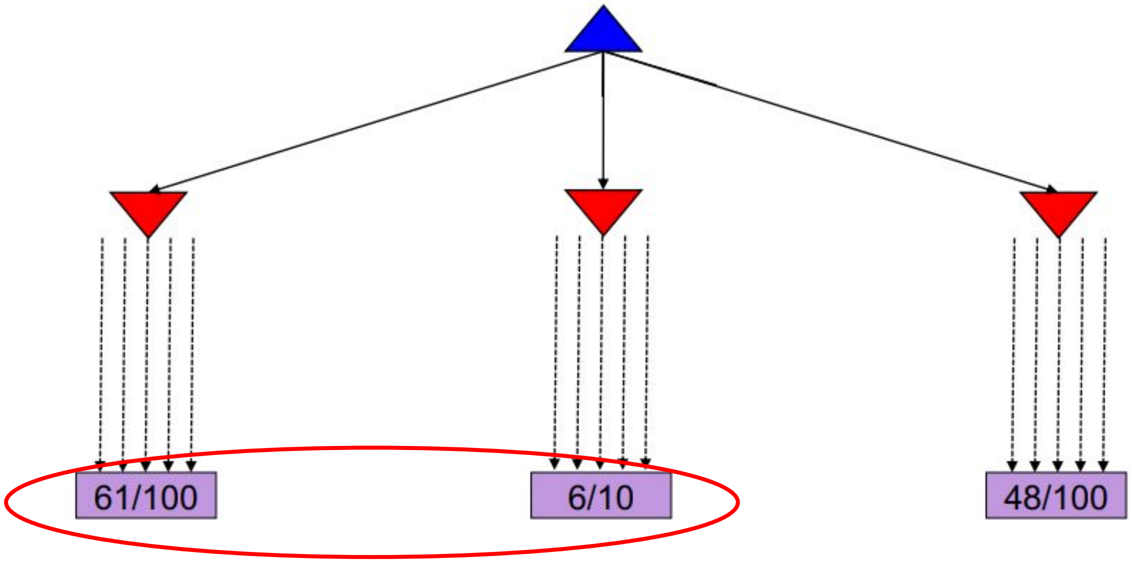
Monte Carlo Tree Search

- Let's have another version where we are more economical.
- Allocate rollouts to more promising nodes.



Monte Carlo Tree Search

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

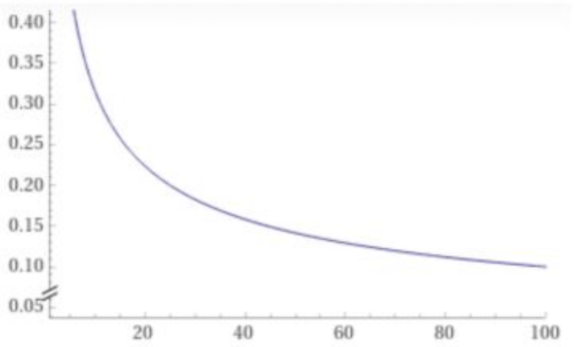


Upper Confidence Bound (UCB) heuristics

- An effective selection policy called “upper confidence bounds applied to trees” (UCT) ranks each possible move based on an upper confidence bound formula called UCB1.
- UCB1 formula combines “promising” and “uncertain”:

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

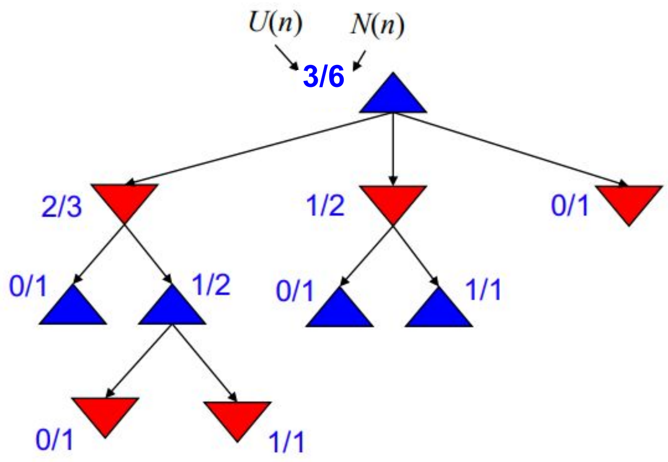
Prefer nodes that have not been played much.



- N(n) = number of rollouts/playouts from node n.
- U(n) = total utility of all rollouts/playouts (e.g., # wins) that went through node n.
- PARENT(n) is the parent node of n in the tree.

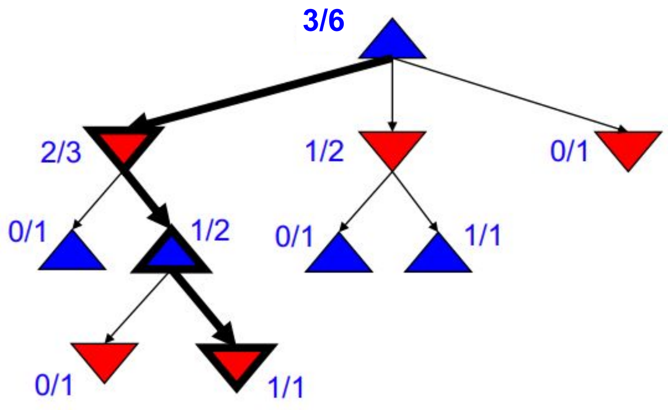
MCTS: Updated

- Repeat until out of time:
  - Selection: recursively apply UCB1 to choose a path down to a node n with unexplored children. (or leave it upto UCB to select 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.



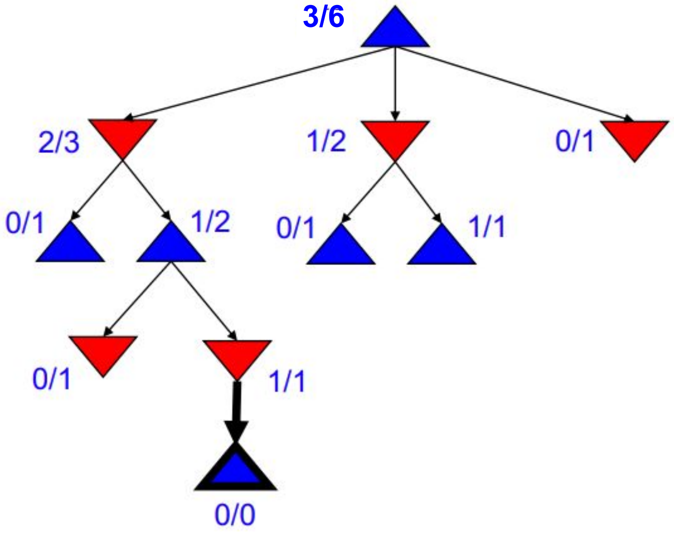
MCTS: Updated

- Repeat until out of time:
  - <u>Selection: recursively apply UCB1 to choose a path down to a node n with unexplored children. (or leave it upto UCB to select a leaf node n)</u>
  - 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



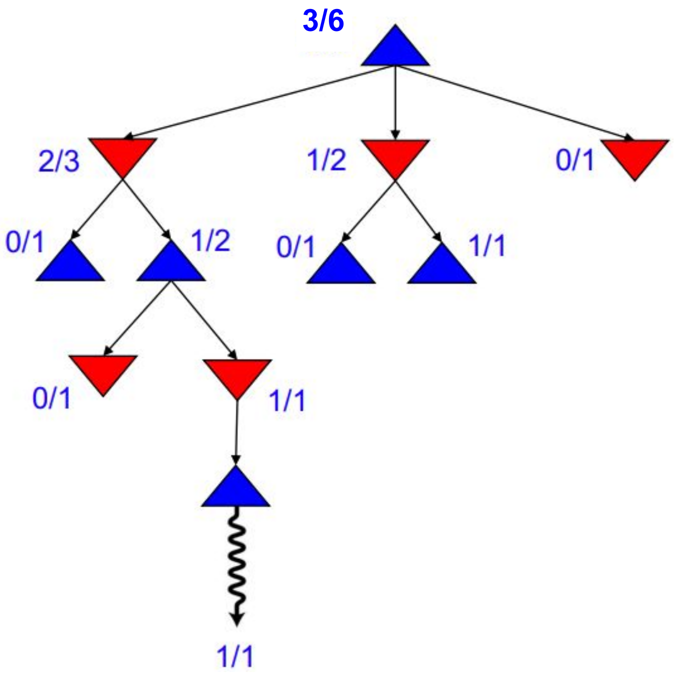
MCTS: Updated

- Repeat until out of time:
  - Selection: recursively apply UCB1 to choose a path down to a node n with unexplored children. (or leave it upto UCB to select a leaf node n)
  - <u>Expansion: add a new child c to n</u>
  - Simulation: run a rollout from c
  - Backpropagation: update U and N counts from c back up to the root



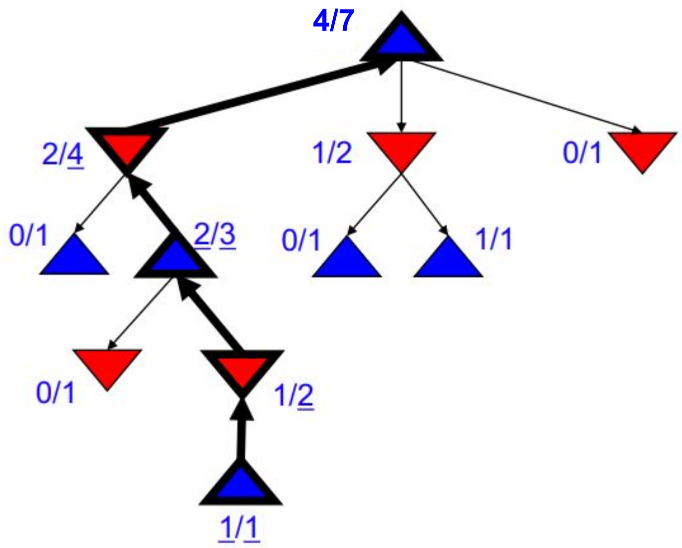
MCTS: Updated

- Repeat until out of time:
  - Selection: recursively apply UCB1 to choose a path down to a node n with unexplored children. (or leave it upto UCB to select a leaf node n)
  - Expansion: add a new child c to n
  - <u>Simulation: run a rollout from c</u>
  - Backpropagation: update U and N counts from c back up to the root



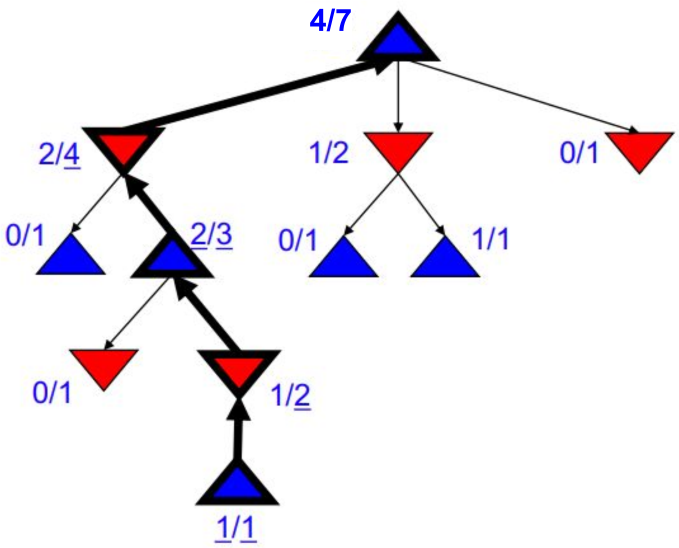
MCTS: Updated

- Repeat until out of time:
    - Selection: recursively apply UCB1 to choose a path down to a node n with unexplored children. (or leave it upto UCB to select a leaf node n)
    - Expansion: add a new child c to n
    - Simulation: run a rollout from c
    - <u>Backpropagation: update U and N counts from c back up to the root</u>

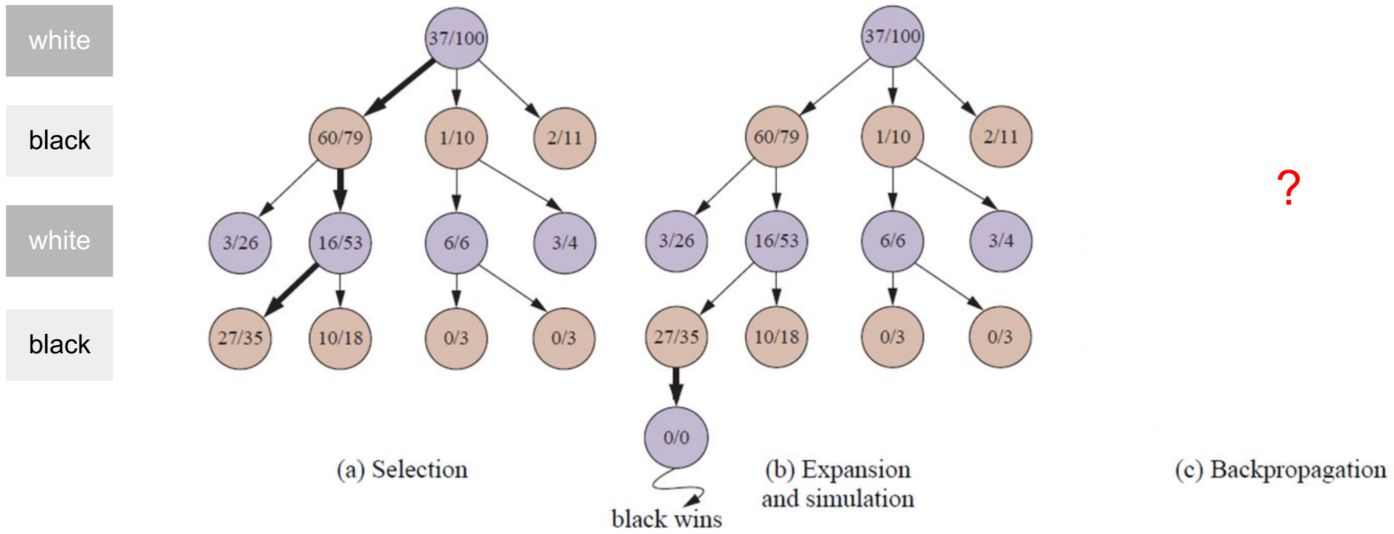


MCTS: Updated

- Repeat until out of time:
  - Selection: recursively apply UCB1 to choose a path down to a node n with unexplored children. (or leave it upto UCB to select 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
- <u>Choose the action leading to the child with the highest N (child).</u>



MCTS: Practice



MCTS Application: 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 a node's utility.

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