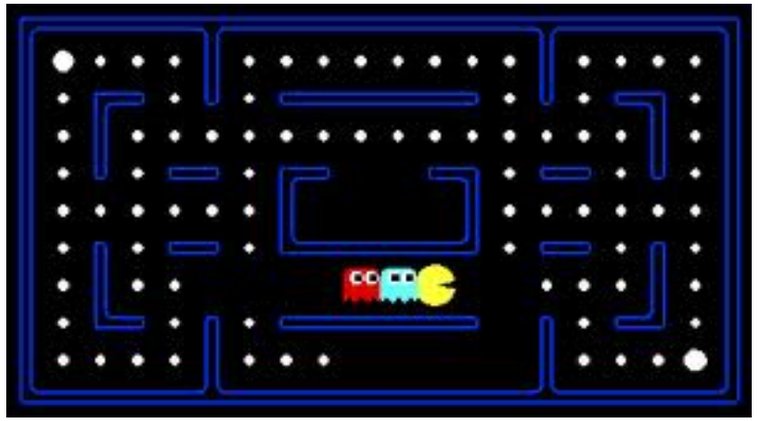
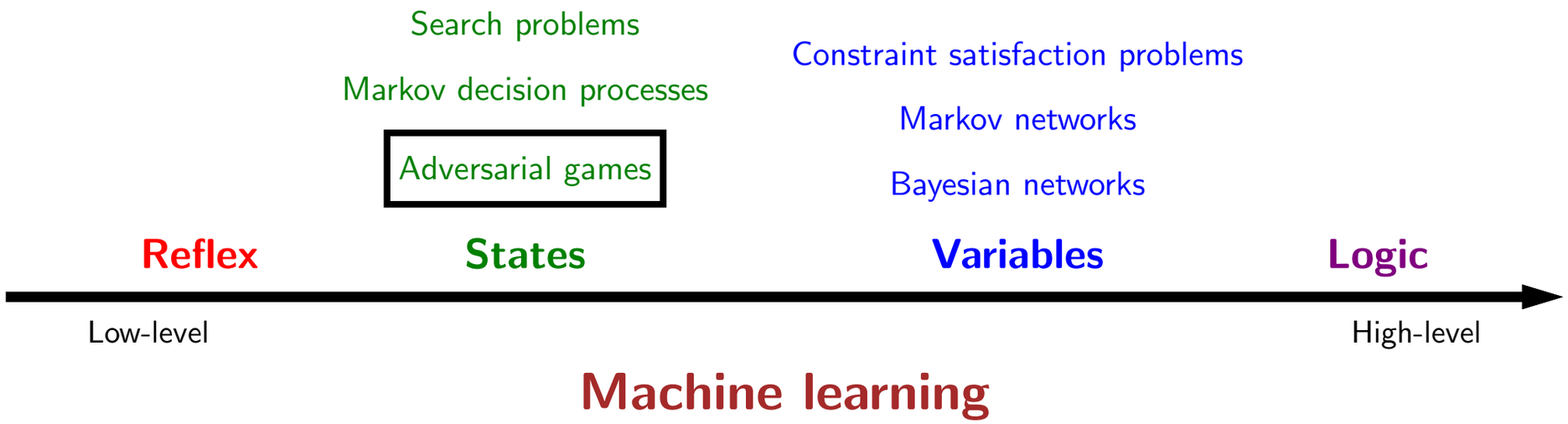
Games: overview



Course plan


Search problems
Markov decision processes
Adversarial games
Constraint satisfaction problems
Markov networks
Bayesian networks
Reflex
States
Variables
Logic
Low-level
High-level
Machine learning

- This lecture will be about games, which have been one of the main testbeds for developing AI programs since the early days of AI. Games are distinguished from the other tasks that we've considered so far in this class in that they make explicit the presence of other agents, whose utility is not generally aligned with ours. Thus, the optimal strategy (policy) for us will depend on the strategies of these agents. Moreover, their strategies are often unknown and adversarial. How do we reason about this?

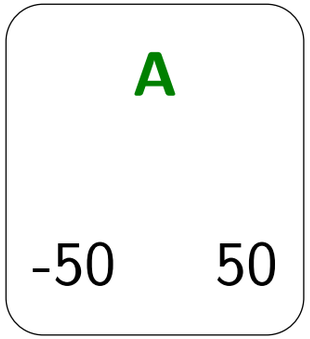
A simple game

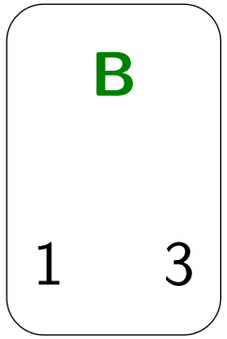
Example: game 1

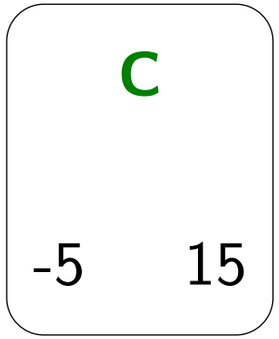
You choose one of the three bins.

I choose a number from that bin.

Your goal is to maximize the chosen number.







working with

max(A) = 50

working against

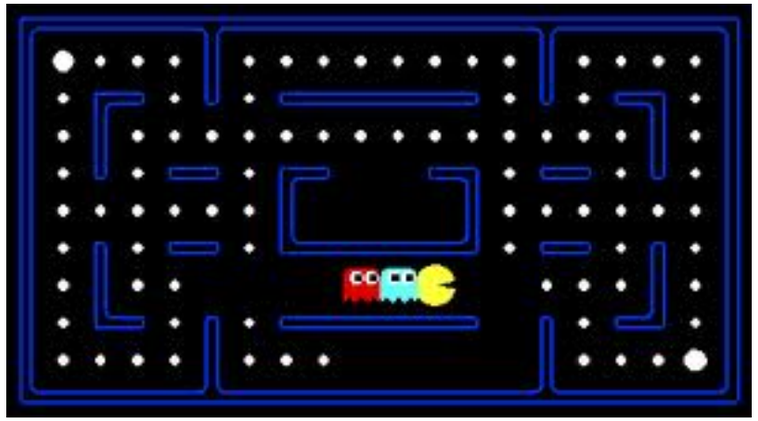
min(B) = 1

random

E[C] = 5

- Which bin should you pick? Depends on your mental model of the other player (me).
- If you think I'm working with you (unlikely), then you should pick A in hopes of getting 50. If you think I'm against you (likely), then you should pick B as to guard against the worst case (get 1). If you think I'm just acting uniformly at random, then you should pick C so that on average things are reasonable (get 5 in expectation).

Games: modeling

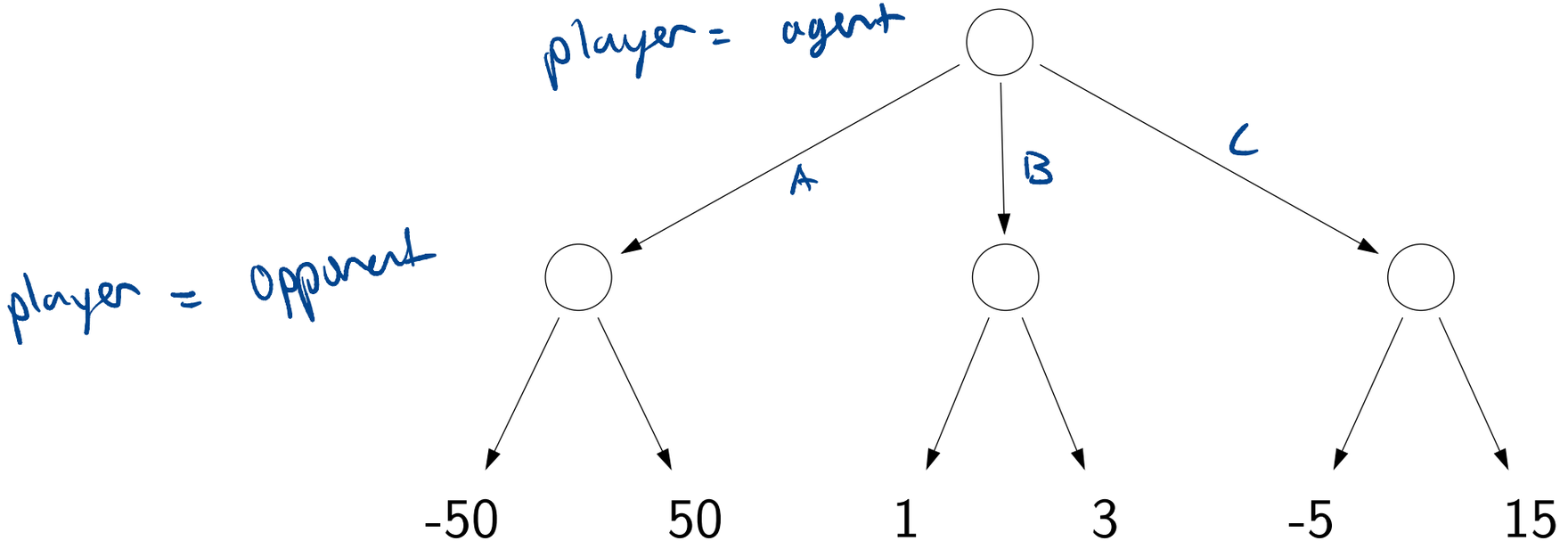


Game tree

Key idea: game tree

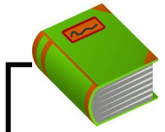
Each node is a decision point for a player.

Each root-to-leaf path is a possible outcome of the game.



Two-player zero-sum games

Players $= \{\text{agent}, \text{opp}\}$



Definition: two-player zero-sum game

s_{start}: starting state

Actions(s): possible actions from state s

Succ(s, a): resulting state if choose action a in state s

IsEnd(s): whether s is an end state (game over)

Utility(s): agent's utility for end state s

Player$(s) \in$ Players: player who controls state s

only get reward at end state

- In this lecture, we will specialize to **two-player zero-sum** games, such as chess. To be more precise, we will consider games in which people take turns (unlike rock-paper-scissors) and where the state of the game is fully-observed (unlike poker, where you don't know the other players' hands). By default, we will use the term **game** to refer to this restricted form.
- We will assume the two players are named agent (this is your program) and opp (the opponent). Zero-sum means that the utility of the agent is negative the utility of the opponent (equivalently, the sum of the two utilities is zero).
- Following our approach to search problems and MDPs, we start by formalizing a game. Since games are a type of state-based model, much of the skeleton is the same: we have a start state, actions from each state, a deterministic successor state for each state-action pair, and a test on whether a state is at the end.
- The main difference is that each state has a designated Player(s), which specifies whose turn it is. A player p only gets to choose the action for the states s such that Player$(s) = p$.
- Another difference is that instead of having edge costs in search problems or rewards in MDPs, we will instead have a utility function Utility(s) defined only at the end states. We could have used edge costs and rewards for games (in fact, that's strictly more general), but having all the utility at the end states emphasizes the all-or-nothing aspect of most games. You don't get utility for capturing pieces in chess; you only get utility if you win the game. This ultra-delayed utility makes games hard.

Example: chess

Players $=$ {white, black}

State s: (position of all pieces, whose turn it is)

Actions(s): legal chess moves that Player(s) can make

IsEnd(s): whether s is checkmate or draw

Utility(s): $+\infty$ if white wins, 0 if draw, $-\infty$ if black wins

Characteristics of games

- All the utility is at the end state

- Different players in control at different states

The halving game

Problem: halving game

Start with a number N.

Players take turns either decrementing N or replacing it with $\lfloor \frac{N}{2} \rfloor$.

The player that is left with 0 wins.

[semi-live solution: `HalvingGame`]

```python
class HalvingGame(object):
    def __init__(self, N):
        self.N = N

    # state = (player, number)

    def startState(self):
        return (+1, self.N)

    def isEnd(self, state):
        player, number = state
        return number == 0

    def utility(self, state):
        player, number = state
        assert number == 0
        return player * float('inf')

    def actions(self, state):
        return ['-', '/']

    def player(self, state):
        player, number = state
        return player

    def succ(self, state, action):
        player, number = state
        if action == '-':
            return (-player, number-1)
        elif action == '/':
            return (-player, number//2)
```

players are +1 and -1

agent

opponent

current number

agent

if agent wins, utility is $+\infty$

if agent loses, utility is $-\infty$

$n-1$ or $n/2$

player changes

Policies

policy of player p

Deterministic policies: $\pi_p(s) \in \text{Actions}(s)$

action that player p takes in state s

Stochastic policies $\pi_p(s, a) \in [0, 1]$:

probability of player p taking action a in state s

randomized

```python
# general code

def humanPolicy(game, state):
    while True:
        action = input('Input action:')
        if action in game.actions(state):
            return action

# controller

policies = {+1: humanPolicy, -1: humanPolicy}
game = HalvingGame(N=15)
state = game.startState()

while not game.isEnd(state):
    print('='*10, state)
    player = game.player(state)
    policy = policies[player]
    action = policy(game, state)
    state = game.succ(state, action)

print('utility = {}'.format(game.utility(state)))
```

user input policy

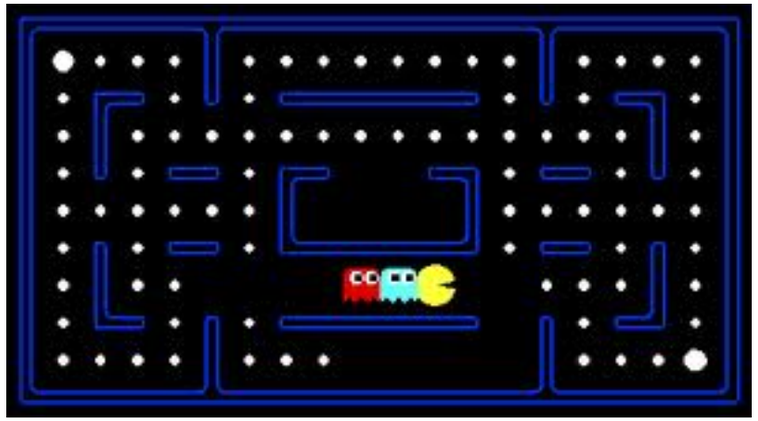
policy printed only at the end

[semi-live solution: `humanPolicy`]

- Following our presentation of MDPs, we revisit the notion of a **policy**. Instead of having a single policy π, we have a policy π_p for each player $p \in \text{Players}$. We require that π_p only be defined when it's p's turn; that is, for states s such that $\text{Player}(s) = p$.
- It will be convenient to allow policies to be stochastic. In this case, we will use $\pi_p(s, a)$ to denote the probability of player p choosing action a in state s.
- We can think of an MDP as a game between the agent and nature. The states of the game are all MDP states s and all chance nodes (s, a). It's the agent's turn on the MDP states s, and the agent acts according to π_{agent}. It's nature's turn on the chance nodes. Here, the actions are successor states s', and nature chooses s' with probability given by the transition probabilities of the MDP: $\pi_{\text{nature}}((s, a), s') = T(s, a, s')$.

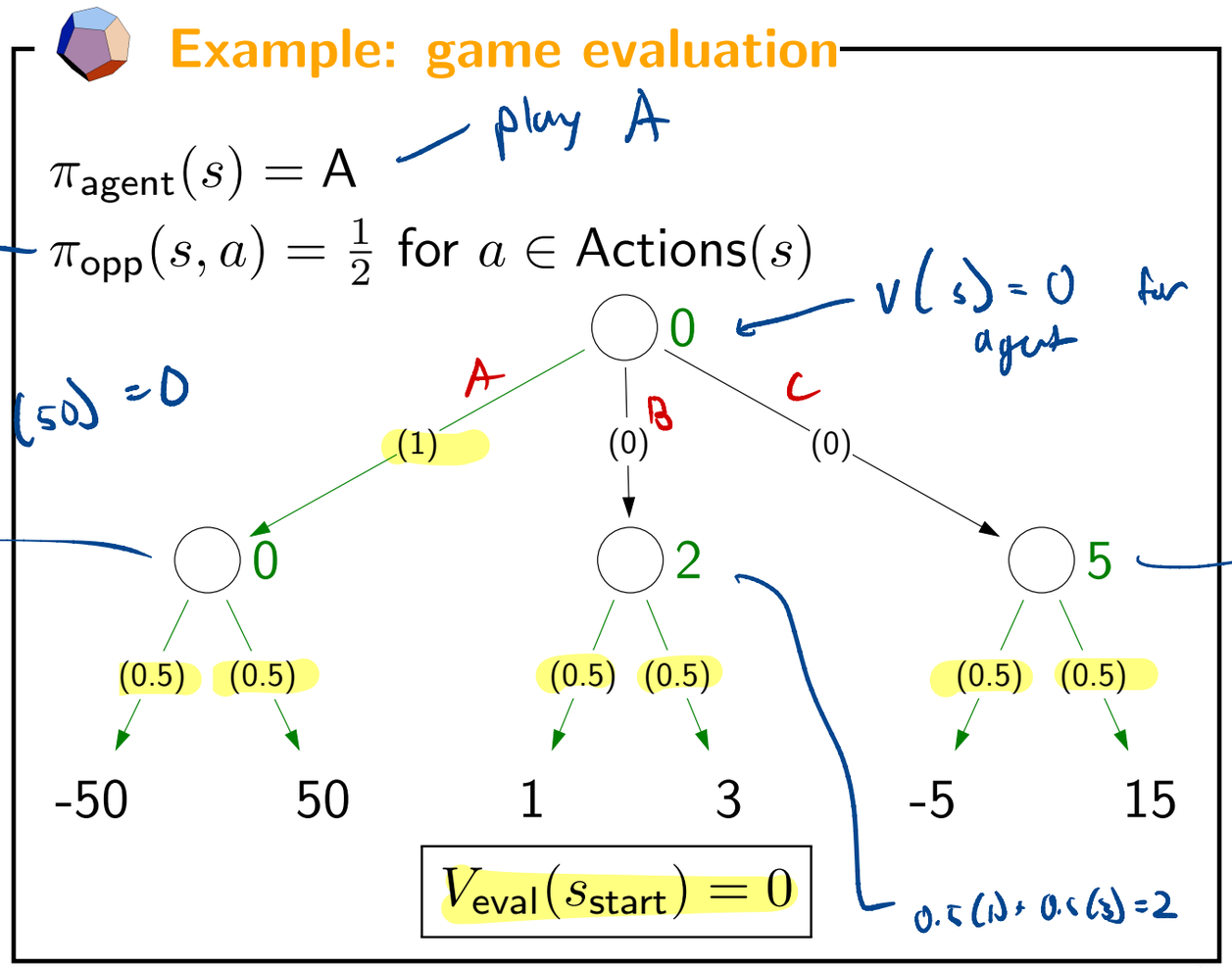
Games: game evaluation

=> policy evaluation



* know π_{agent}, π_{opp}

Game evaluation example



Example: game evaluation

$\pi_{\text{agent}}(s) = \text{A}$

$\pi_{\text{opp}}(s, a) = \frac{1}{2}$ for $a \in \text{Actions}(s)$

0

A (1) B (0) C (0)

0 2 5

(0.5) (0.5) (0.5) (0.5) (0.5) (0.5)

-50 50 1 3 -5 15

$V_{\text{eval}}(s_{\text{start}}) = 0$

play A

play randomly w/ uniform prob

V(s) = 0 for agent

V(s) = 0.5(-50) + 0.5(50) = 0

0.5(15) + 0.5(-5) = 5

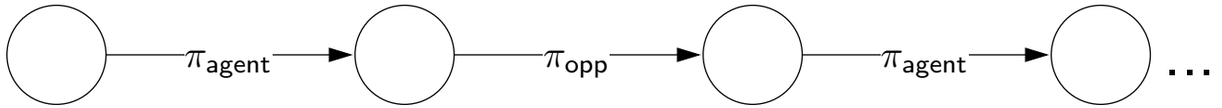
0.5(1) + 0.5(3) = 2

$V_{\text{eval}}(s_{\text{start}}) = \pi(s, A) \cdot V(s') + \pi(s, B) \cdot V(s') + \pi(s, C) \cdot V(s')$

1 0 0

Game evaluation recurrence

Analogy: recurrence for policy evaluation in MDPs



Value of the game:

* utility of the agent from perspective

recursive formula

$$V_{\text{eval}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{agent}}(s,a) V_{\text{eval}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{eval}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

value

prob of action

value of successor state

- More generally, we can write down a recurrence for $V_{\text{eval}}(s)$, which is the **value** (expected utility) of the game at state s.
- There are three cases: If the game is over ($\text{IsEnd}(s)$), then the value is just the utility $\text{Utility}(s)$. If it's the agent's turn, then we compute the expectation over the value of the successor resulting from the agent choosing an action according to $\pi_{\text{agent}}(s, a)$. If it's the opponent's turn, we compute the expectation with respect to π_{opp} instead.

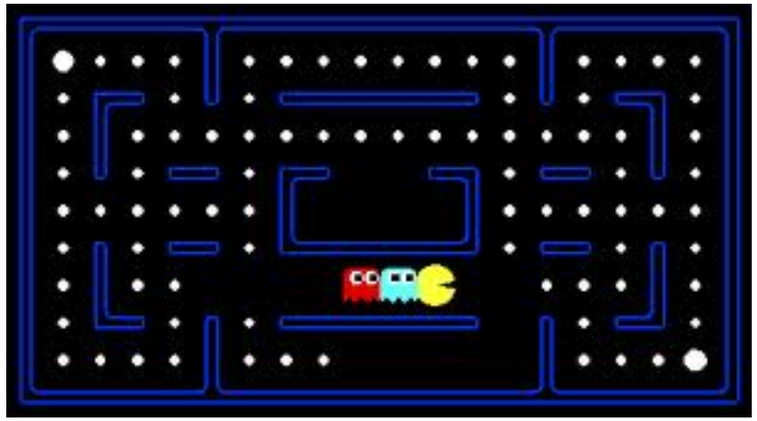
$V_{eval}(s)$ → know π_{agent}, know π_{opp}

$V_{expectimax}(s)$ → know π_{opp}

$V_{minmax}(s)$ → don't know either

Games: expectimax

=> value iteration



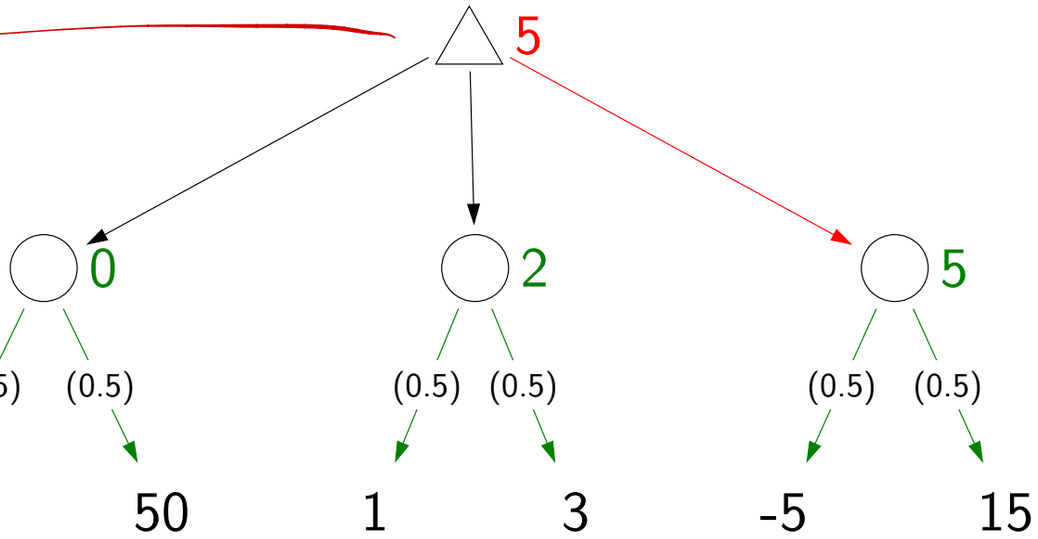
don't know π_{agent}, know π_{op}

Expectimax example

take max over successor states

Example: expectimax

$\pi_{\text{opp}}(s, a) = \frac{1}{2}$ for $a \in \text{Actions}(s)$



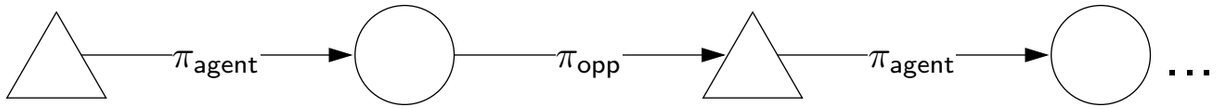
$V_{\text{exptmax}}(s_{\text{start}}) = 5$

△ is a max node, where you maximize over actions

Expectimax recurrence

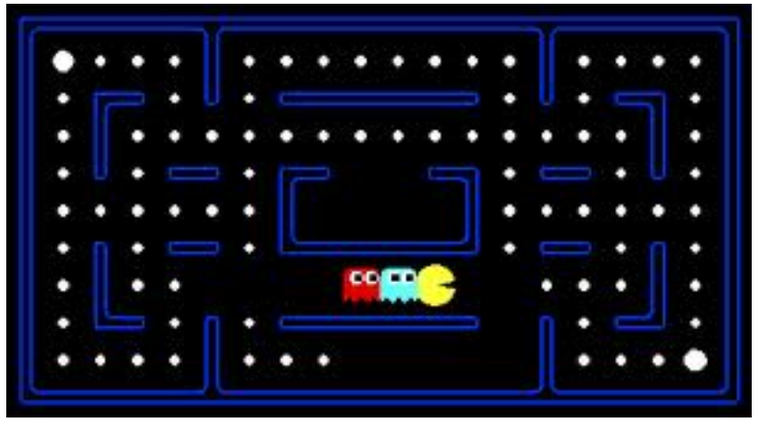
Analogy: recurrence for value iteration in MDPs

* agent is a maximizing agent



$$V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Games: minimax

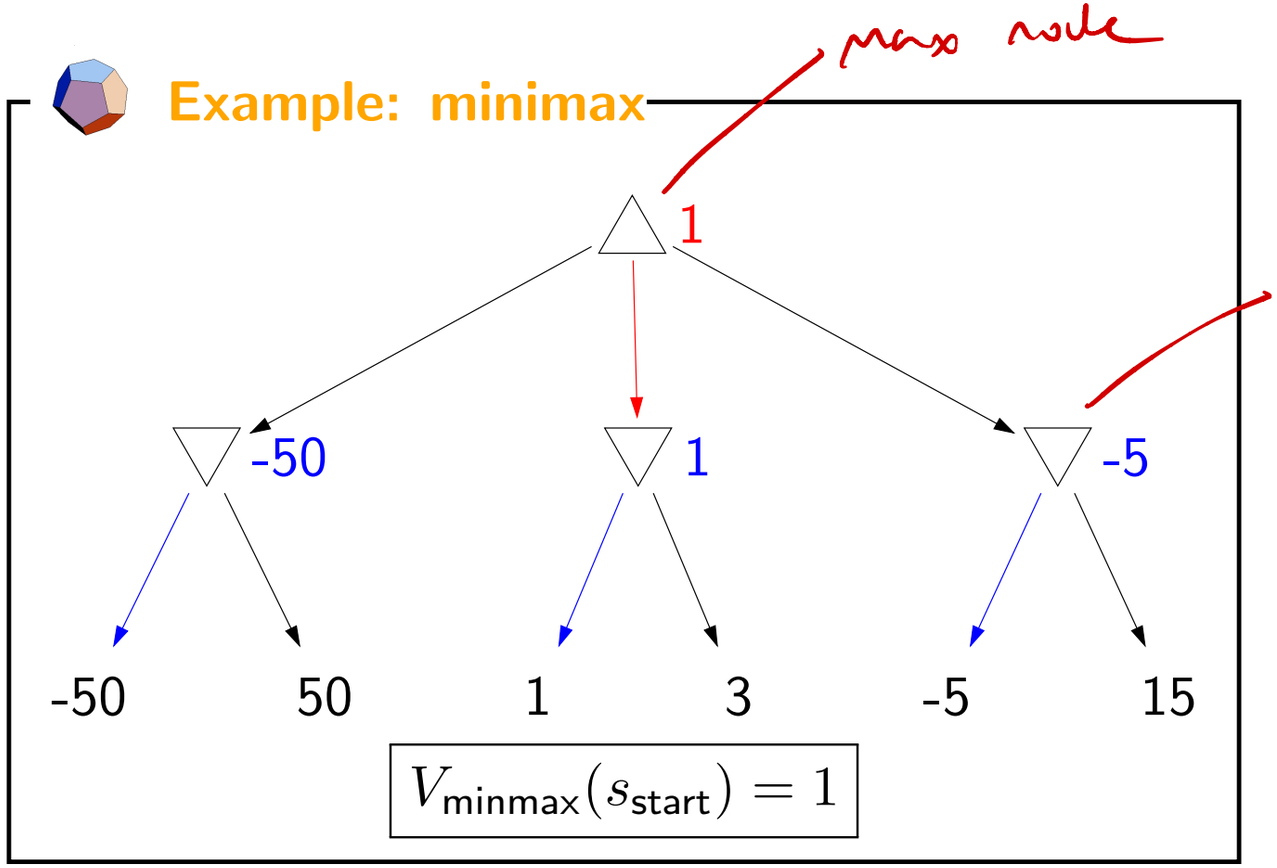


don't know π_{opp} or π_{opp}

Problem: don't know opponent's policy

Approach: assume the worst case

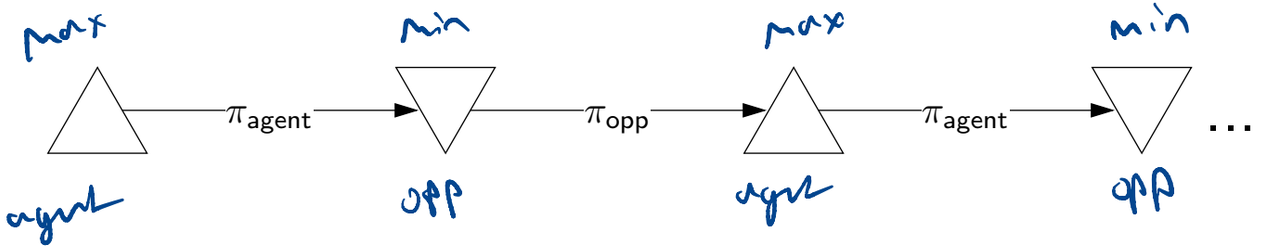
Minimax example



★ assume worst case → opponent is trying to minimize the agent's utility

Minimax recurrence

No analogy in MDPs:

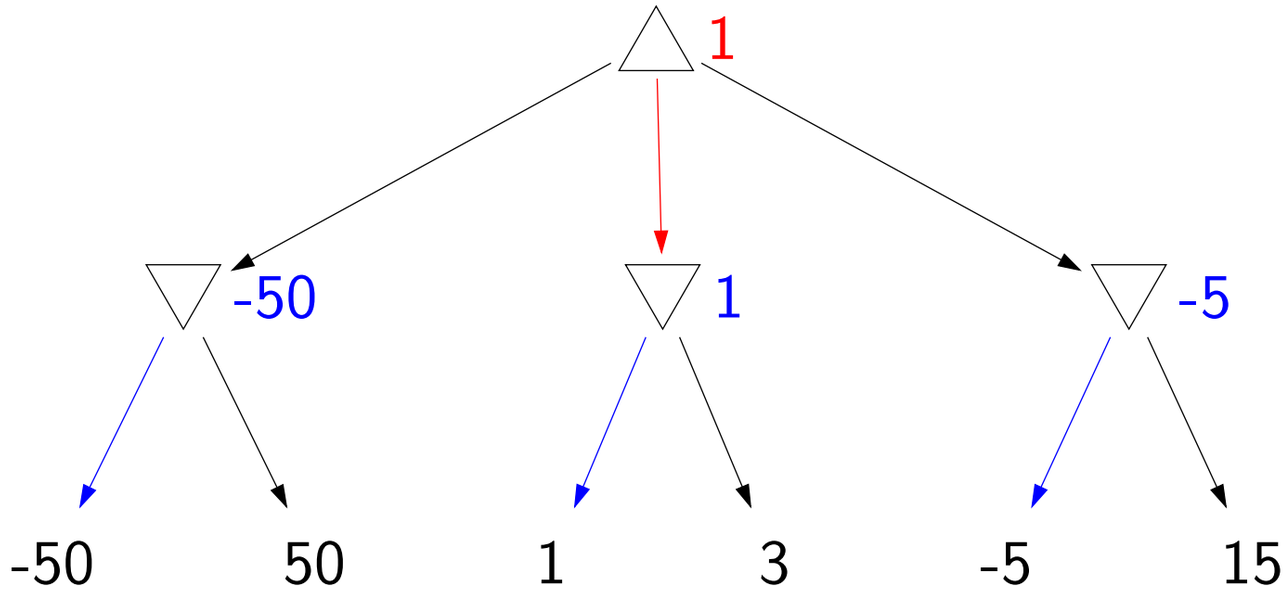


$$V_{\text{minmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Extracting minimax policies

$$\pi_{\max}(s) = \arg\max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a))$$

$$\pi_{\min}(s) = \arg\min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a))$$



The halving game

Problem: halving game

Start with a number N.

Players take turns either decrementing N or replacing it with $\lfloor \frac{N}{2} \rfloor$.

The player that is left with 0 wins.

[semi-live solution: `minimaxPolicy`]

```python
def minimaxPolicy(game, state):
    def recurse(state):
        # Return (utility of that state, action that achieves that
        utility)
        if game.isEnd(state):
            return (game.utility(state), None)
        # List of (utility of succ, action leading to that succ)
        candidates = [
            (recurse(game.succ(state, action))[0], action)
            for action in game.actions(state)
        ]
        player = game.player(state)
        if player == +1:
            return max(candidates)
        elif player == -1:
            return min(candidates)
        assert False

    utility, action = recurse(state)
    print('minimaxPolicy: state {} => action {} with utility {}'.format
        (state, action, utility))
    return action
```

Utility(s)

gets the utility

$\max V_{minmax}(Succ(s, a))$

```python
policies = {+1: humanPolicy, -1: minimaxPolicy}

state = game.startState()
while not game.isEnd(state):
    # Who controls this state?
    player = game.player(state)
    policy = policies[player]
    # Ask policy to make a move
    action = policy(game, state)
    # Advance state
    state = game.succ(state, action)
```

agent is human

opponent is playing optimally w/ minimax

Face off

Recurrences produces policies:

agent knows opponent is playing π_7

$$V_{\text{exptmax}} \Rightarrow \pi_{\text{exptmax(7)}}, \pi_7 \text{ (some opponent)}$$

$$V_{\text{minmax}} \Rightarrow \pi_{\max}, \pi_{\min}$$ — opponent plays this

Play policies against each other:

Agent plays this

agent \ opponent	$\pi_{\min}$	π_7
$\pi_{\max}$	$V(\pi_{\max}, \pi_{\min})$	$V(\pi_{\max}, \pi_7)$
$\pi_{\text{exptmax(7)}}$	$V(\pi_{\text{exptmax(7)}}, \pi_{\min})$	$V(\pi_{\text{exptmax(7)}}, \pi_7)$

What's the relationship between these values?

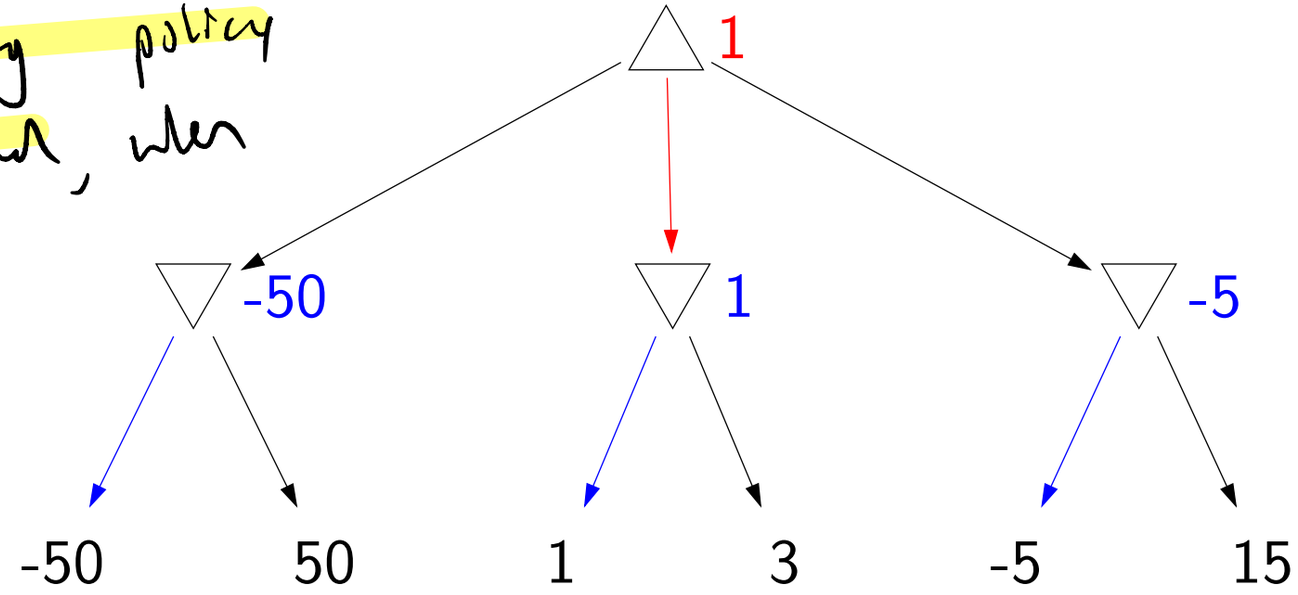
$V(\pi_{\text{exptmax(7)}}, \pi_{\min}) \leq V(\pi_{\max}, \pi_{\min}) \leq V(\pi_{\max}, \pi_7) \leq V(\pi_{\text{exptmax}}, \pi_7)$

Minimax property 1

Proposition: best against minimax opponent

$V(\pi_{\max}, \pi_{\min}) \geq V(\pi_{\text{agent}}, \pi_{\min})$ for all π_{agent}

value of maximizing policy is the upper bound, when playing against minimizing opponent

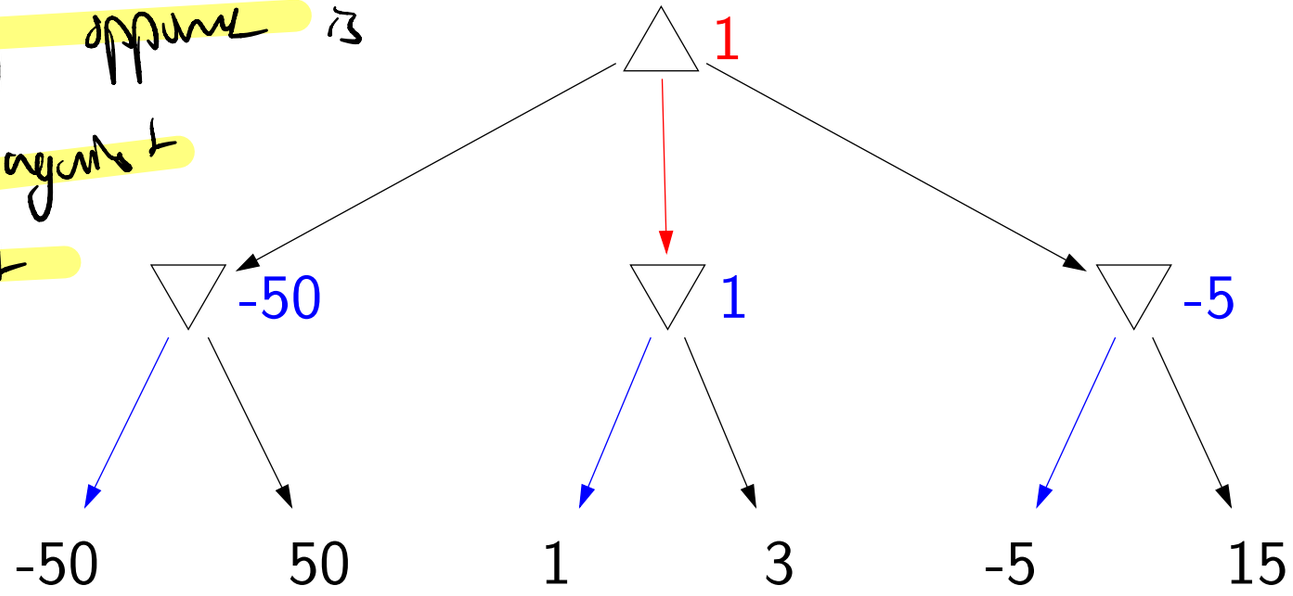


Minimax property 2

Proposition: lower bound against any opponent

$V(\pi_{\max}, \pi_{\min}) \leq V(\pi_{\max}, \pi_{\text{opp}})$ for all π_{opp}

value of minimizing opponent is lower bound when against a maximizing agent

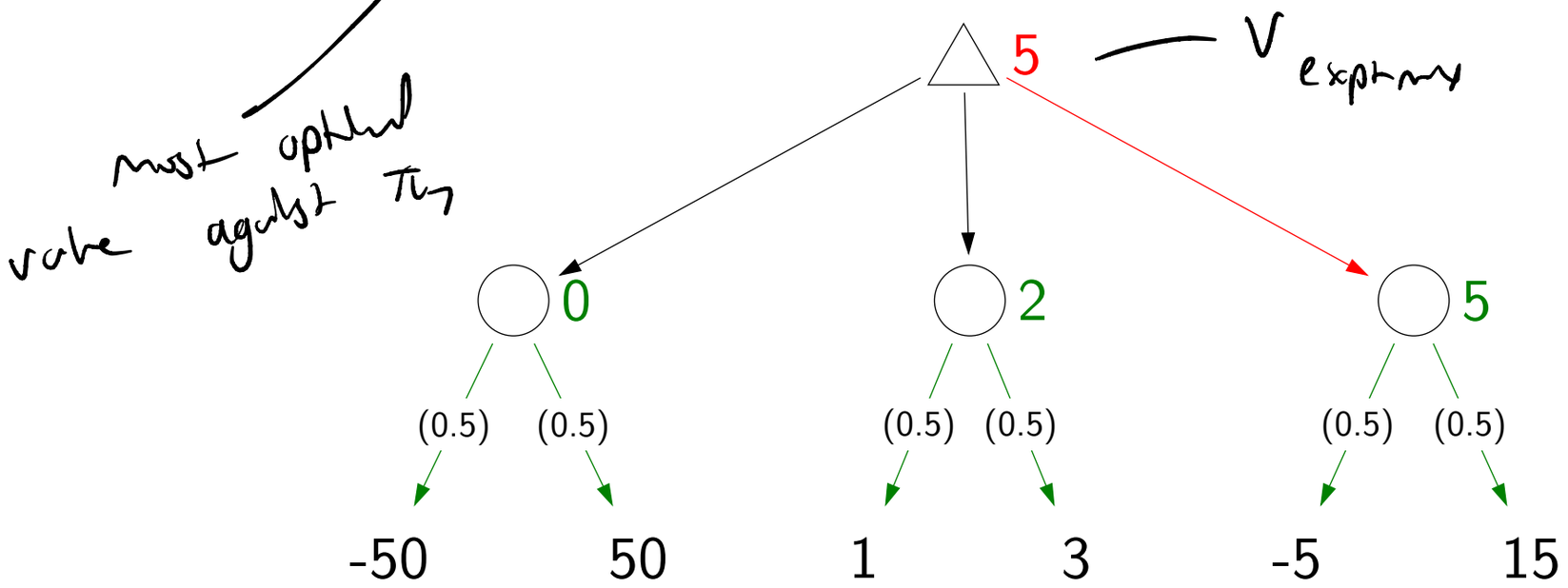


- The second property is the analogous statement for the opponent: if the opponent changes his policy from $\pi_{\min}$ to π_{opp}, then he will be no better off (the value of the game can only increase).
- From the point of the view of the agent, this can be interpreted as guarding against the worst case. In other words, if we get a minimax value of 1, that means no matter what the opponent does, the agent is guaranteed at least a value of 1. As a simple example, if the minimax value is $+\infty$, then the agent is guaranteed to win, provided it follows the minimax policy.

Minimax property 3

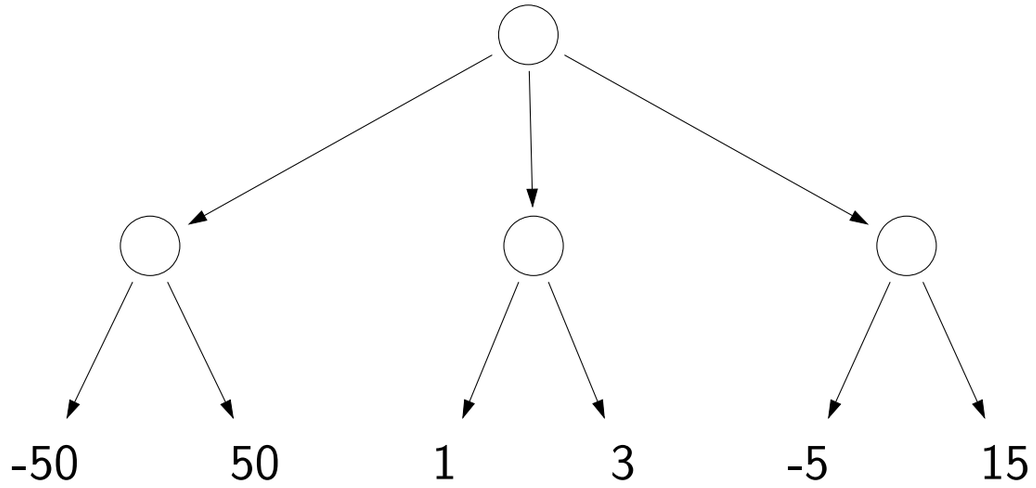
Proposition: not optimal if opponent is known

$V(\pi_{\max}, \pi_7) \leq V(\pi_{\text{exptmax(7)}}, \pi_7)$ for opponent π_7



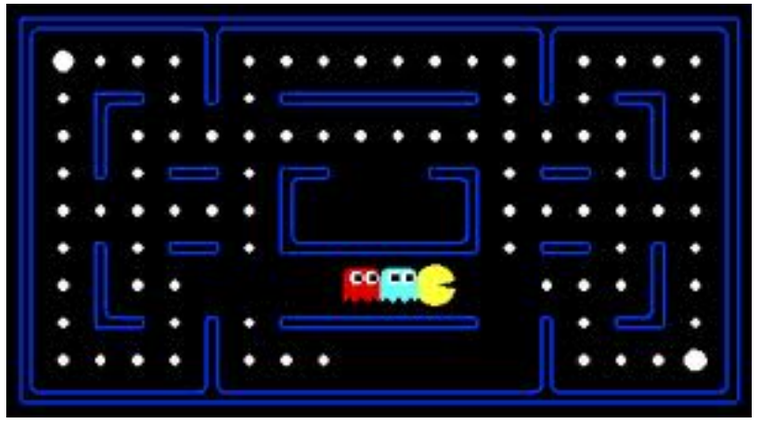
- However, following the minimax policy might not be optimal for the agent if the opponent is known to be not playing the adversarial (minimax) policy.
- Consider the running example where the agent chooses A, B, or C and the opponent chooses a bin. Suppose the agent is playing $\pi_{\max}$, but the opponent is playing a stochastic policy π_7 corresponding to choosing an action uniformly at random.
- Then the game value here would be 2 (which is larger than the minimax value 1, as guaranteed by property 2). However, if we followed the expectimax $\pi_{\text{exptmax}(7)}$, then we would have gotten a value of 5, which is even higher.

Relationship between game values



	$\pi_{\min}$		π_7
$\pi_{\max}$	$V(\pi_{\max}, \pi_{\min})$	$\leq$	$V(\pi_{\max}, \pi_7)$
	1		2
	$\vert\vee$		$\vert\wedge$
$\pi_{\text{exptmax(7)}}$	$V(\pi_{\text{exptmax(7)}}, \pi_{\min})$		$V(\pi_{\text{exptmax(7)}}, \pi_7)$
	-5		5

Games: expectiminimax



What if there is uncertainty in nature?

A modified game

Example: game 2

You choose one of the three bins.

Flip a coin; if heads, then move one bin to the left (with wrap around).

I choose a number from that bin.

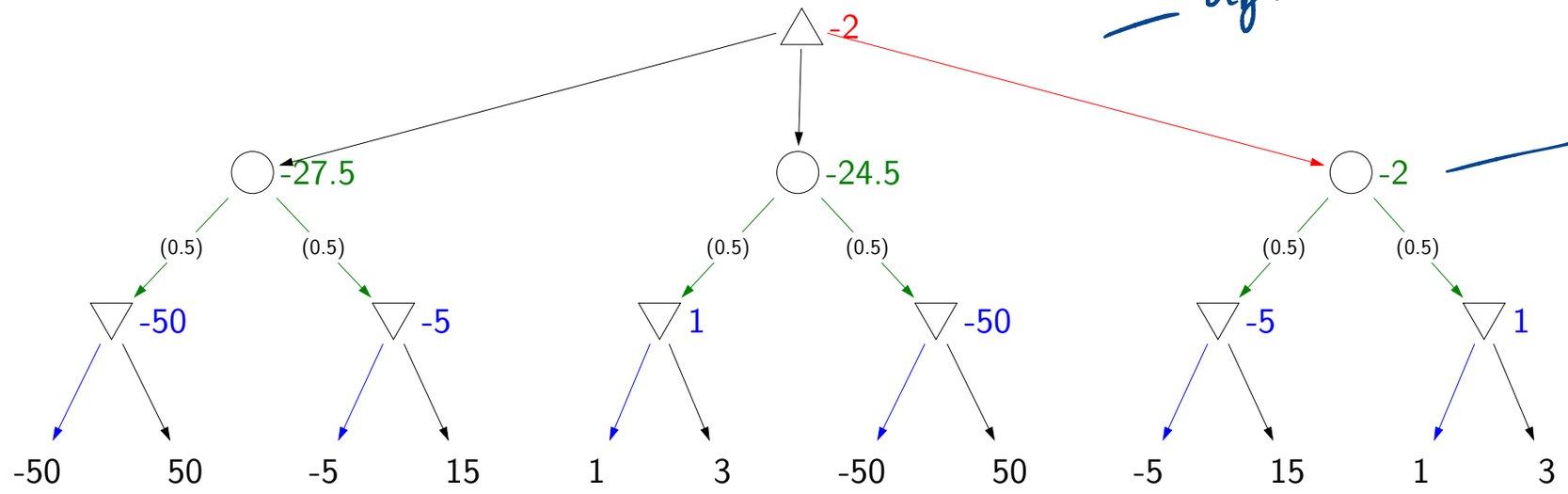
Your goal is to maximize the chosen number.

A	B	C
-50 50	1 3	-5 15

Expectiminimax example

Example: expectiminimax

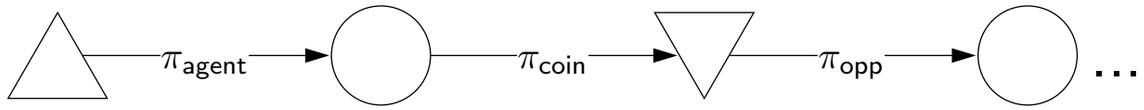
$\pi_{\text{coin}}(s, a) = \frac{1}{2}$ for $a \in \{0, 1\}$



$$V_{\text{exptminmax}}(s_{\text{start}}) = -2$$

Expectiminimax recurrence

Players $= \{\text{agent}, \text{opp}, \text{coin}\}$



$$V_{\text{exptminmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{coin}}(s,a) V_{\text{exptminmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{coin} \end{cases}$$

same as before

$\sum$ we know policy of coin, just evaluate it

Summary so far

Primitives: max nodes, chance nodes, min nodes

Composition: alternate nodes according to model of game

Value function $V_{...}(s)$: recurrence for expected utility

Scenarios to think about:

- What if you are playing against multiple opponents?
- What if you and your partner have to take turns (table tennis)?
- Some actions allow you to take an extra turn?