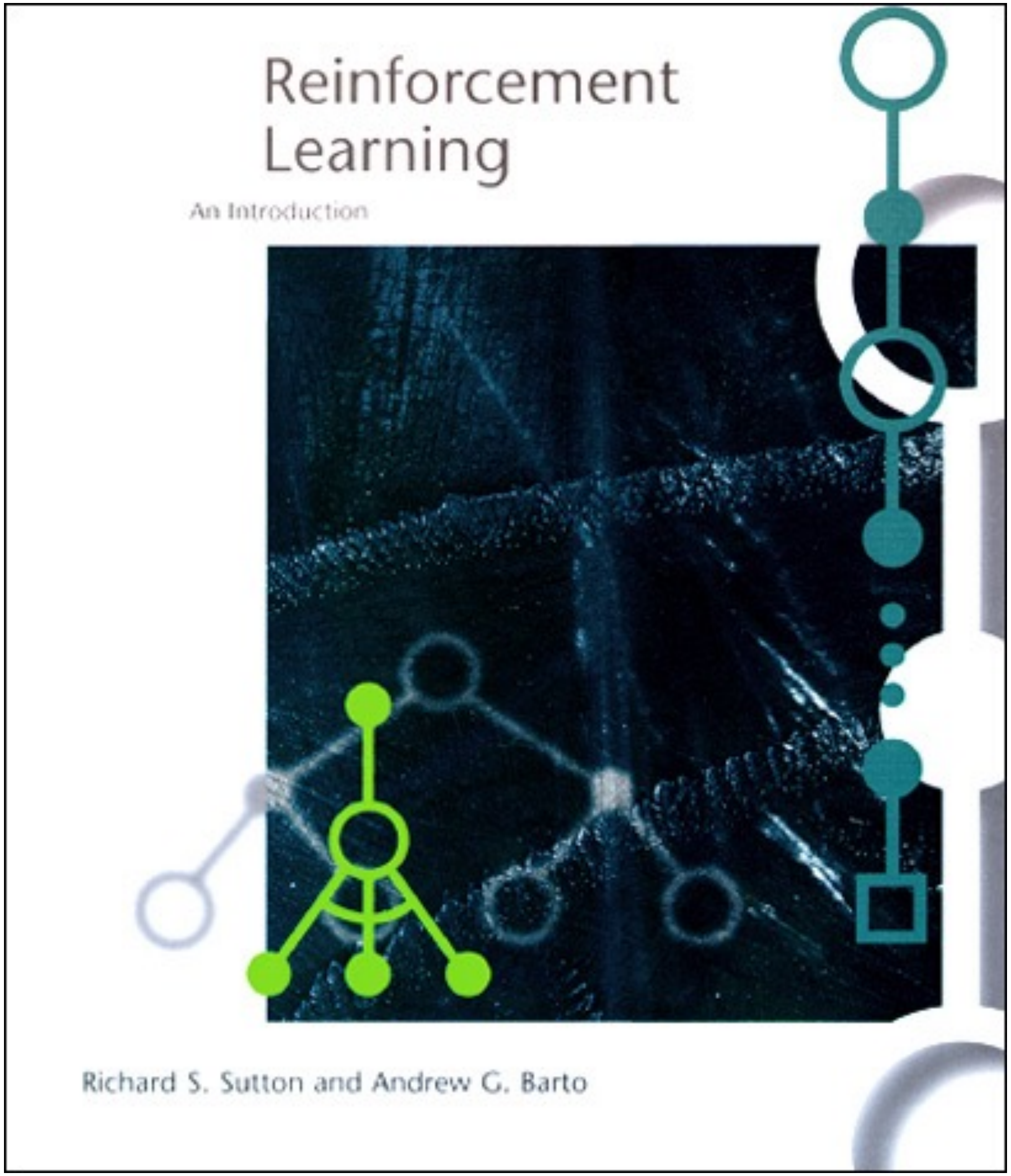
CMSC 478:
Reinforcement Learning

Some slides courtesy Cynthia Matuszek and Frank Farrero, with some material from Marie desJardin, Lise Getoor, Jean-Claude Latombe, and Daphne Koller

There's an entire book!

http://incompleteideas.net/book/the-book-2nd.html



The Big Idea

- "Planning": Find a sequence of steps to accomplish a goal.
  - Given start state, transition model, goal functions…
- This is a kind of **sequential decision making**.
  - Transitions are deterministic.
- What if they are stochastic (probabilistic)?
  - One time in ten, you drop your sock
- Probabilistic Planning: Make a plan that accounts for probability by **carrying it through the plan**.

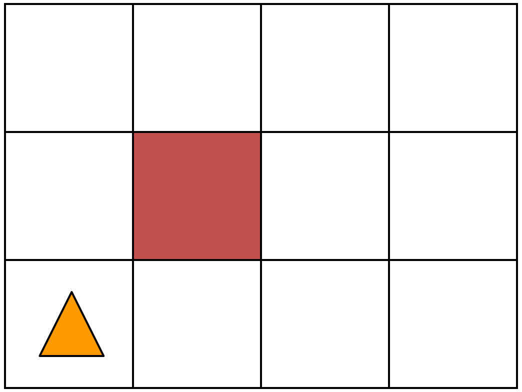
Review: Formalizing Agents

- Given:
  - A state space S
  - A set of actions $a_1, \ldots, a_k$ including their results
  - Reward value at the end of each trial (series of action) (may be positive or negative)
- Output:
  - A **mapping from states to actions**
  - Which is a **policy**, π

Reinforcement Learning

- We often have an agent which has a **task** to perform
  - It takes some actions in the world
  - At some later point, gets feedback on how well it did
  - The agent performs the same task repeatedly
- This problem is called **reinforcement learning**:
  - The agent gets positive reinforcement for tasks done well
  - And gets negative reinforcement for tasks done poorly
  - Must somehow figure out which actions to take next time

Probabilistic Transition Model



- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
  - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)
  - With probability 0.1, the robot moves right one square (if the robot is already in the rightmost row, then it does not move)
  - With probability 0.1, the robot moves left one square (if the robot is already in the leftmost row, then it does not move)
- D, R, and L have similar probabilistic effects

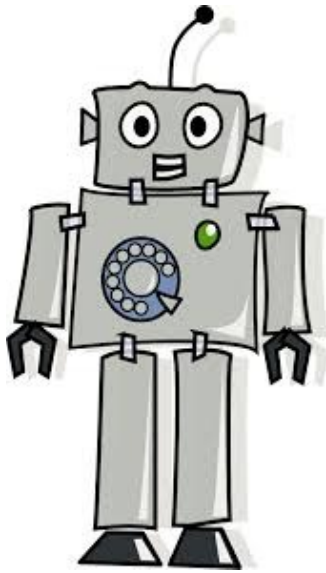
Markov Property

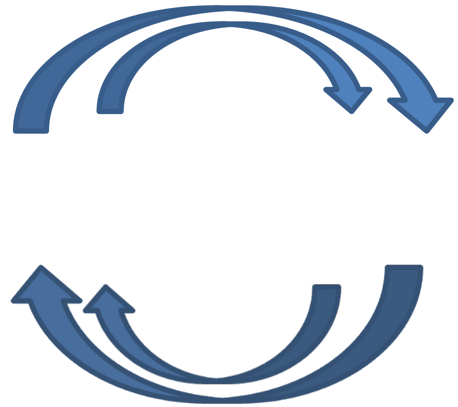
The transition properties depend only on the current state, not on the previous history (how that state was reached)

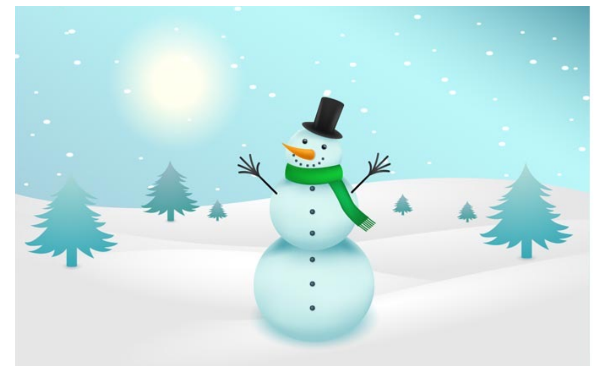
Markov assumption generally: current state only ever depends on previous state (or finite set of previous states).

Markov Decision Process: Formalizing Reinforcement Learning

take action







get new state
and/or reward

environment

agent

set of possible actions

state-action transition distribution

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

set of possible states

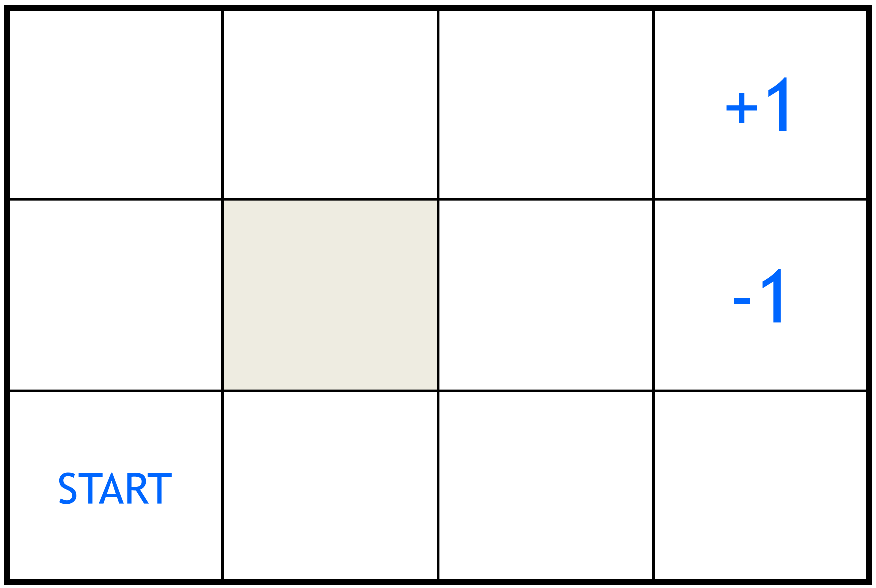
reward of (state, action) pairs

discount factor

Robot image: openclipart.org

https://static.vecteezy.com/system/resources/previews/000/090/451/original/four-seasons-landscape-illustrations-vector.jpg

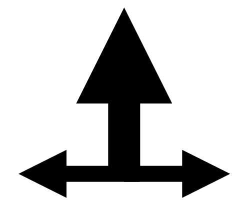
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

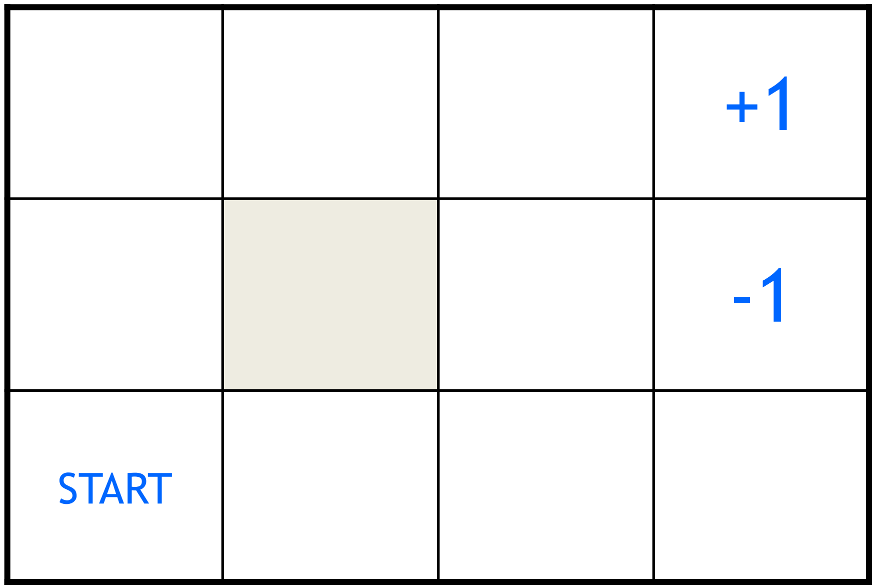
80% move UP
10% move LEFT
10% move RIGHT



reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

Goal: what's the strategy to achieve the maximum reward?

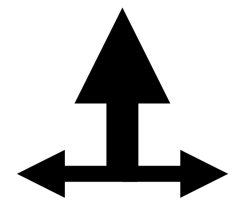
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

80% move UP
10% move LEFT
10% move RIGHT



reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

states: current location

actions: where to go next

rewards

what is the solution? Learn a mapping from (state, action) pairs to new states

Slide courtesy Peter Bodík

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- P: state-action transition distribution
- γ: discount factor

Start in initial state s_0

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

set of possible states

set of possible actions

reward of (state, action) pairs

state-action transition distribution

discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t

Markov Decision Process: Formalizing Reinforcement Learning

set of possible actions

state-action transition distribution

Markov Decision Process:

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$$

set of possible states

reward of (state, action) pairs

discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  “move” to next state $s_t \sim \pi(\cdot|s_{t-1}, a_t)$

$Policy$
π: S → A

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- P: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot|s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- P: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: **choose action over time to maximize time-discounted reward**

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- P: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

Consider all possible future times t

Discount at time t

Reward at time t

Example of Discounted Reward

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

Consider all possible future times t

Discount at time t

Reward at time t

- If the discount factor $\gamma = 0.8$ then reward

$$0.8^0 r_0 + 0.8^1 r_1 + 0.8^2 r_2 + 0.8^3 r_3 + \cdots + 0.8^n r_n + \ \ldots$$

- Allows you to consider all possible rewards in the future but preferring current vs. future self

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- P: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
 choose action a_t
 "move" to next state $s_t \sim \pi(\cdot|s_{t-1}, a_t)$
 get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

"solution": the policy π^* that maximizes the expected (average) time-discounted reward

Markov Decision Process: Formalizing Reinforcement Learning

Markov Decision Process: $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$

- $\mathcal{S}$: set of possible states
- $\mathcal{A}$: set of possible actions
- $\mathcal{R}$: reward of (state, action) pairs
- P: state-action transition distribution
- γ: discount factor

Start in initial state s_0
for t = 1 to …:
  choose action a_t
  "move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$
  get reward $r_t = \mathcal{R}(s_t, a_t)$

objective: maximize discounted reward

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

"solution" $$\pi^* = \operatorname*{argmax}_{\pi} \mathbb{E}\left[\sum_{t>0} \gamma^t r_t \,; \pi\right]$$

Markov Decision Process: Formalizing Reinforcement Learning

Mar

Here, r_t is a function of random variable s_t.

Start in initia

for t = 1 to ..

choose act

"move" to next state $s_t \sim \pi(\cdot | s_{t-1}, a_t)$

get reward $r_t = \mathcal{R}(s_t, a_t)$

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

"solution" $\pi^* = \operatorname*{argmax}_{\pi} \mathbb{E}\left[\sum_{t>0} \gamma^t r_t \,; \pi\right]$

Markov Decision Process: Formalizing Reinforcement Learning

Mar

Here, r_t is a function of random variable s_t. ➔
The expectation is over the different states s_t the agent could be in at time *t* (equiv. actions the agent could take).

Start in initia

for t = 1 to ..

choose act

"move" to next state $s_t \sim \pi(\cdot|s_{t-1}, a_t)$

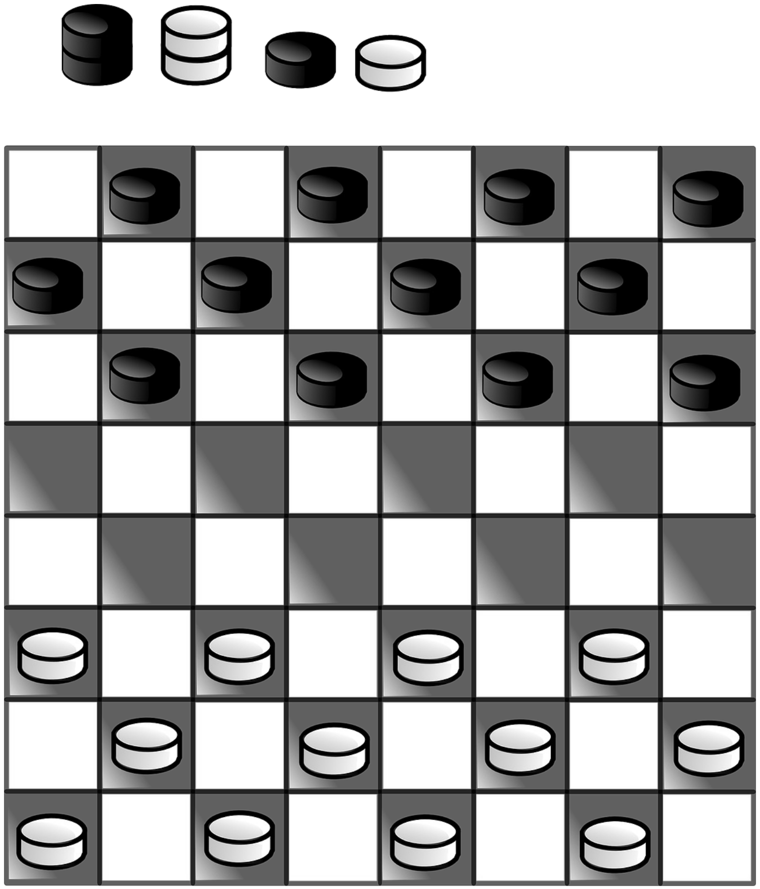
get reward $r_t = \mathcal{R}(s_t, a_t)$

$$\max_{\pi} \sum_{t>0} \gamma^t r_t$$

"solution" $\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}\left[\sum_{t>0} \gamma^t r_t \,; \pi\right]$

Simple Example

- Learn to play checkers
  - Two-person game
  - 8x8 boards, 12 checkers/side
  - relatively simple set of rules: http://www.darkfish.com/checkers/rules.html
  - Goal is to eliminate all your opponent's pieces



https://pixabay.com/en/checker-board-black-game-pattern-29911

Some Challenges

1. Representing states (and actions)

2. Defining our reward

3. Learning our policy

Overview: Learning Strategies

Dynamic Programming

Q-learning

Monte Carlo approaches

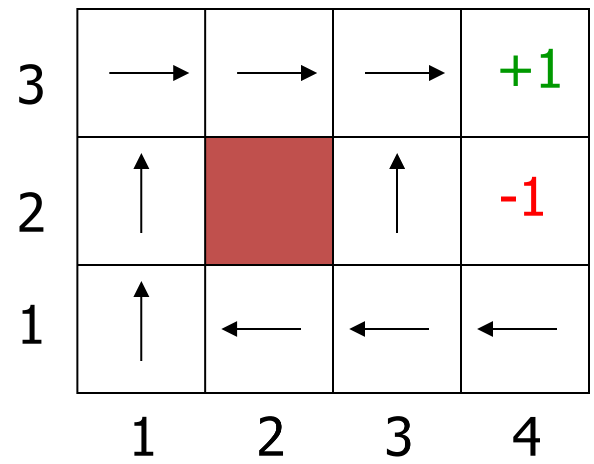
Reactive Agent Algorithm

Repeat:

- s ← sensed state
- If s is a terminal state then exit
- a ← choose action (given s)
- Perform a

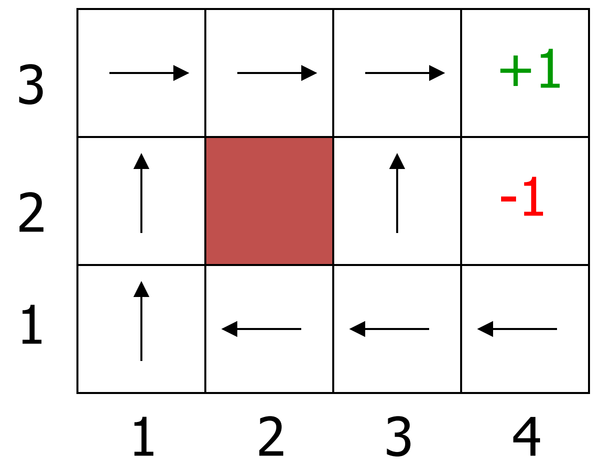
Accessible or observable state

Policy (Reactive/Closed-Loop Strategy)



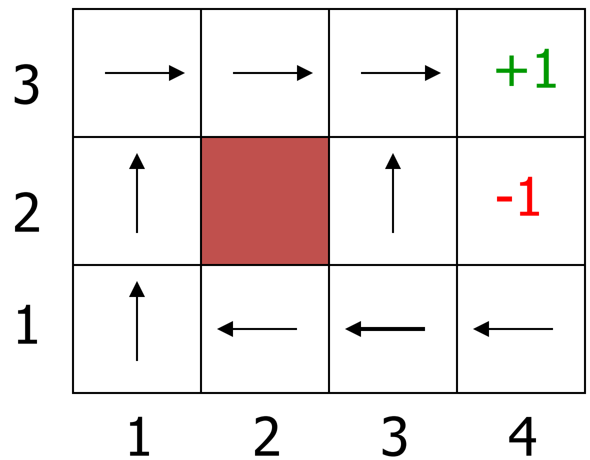
- In every state, we need to know what to do
- The **goal** doesn't change
- A policy (Π) is a complete mapping from *states* to *actions*
    - "If in [3,2], go up; if in [3,1], go left; if in…"

Optimal Policy



- A policy Π is a complete mapping from states to actions
- The optimal policy Π* is the one that always yields a history (sequence of steps ending at a terminal state) with maximal ***expected*** utility

Optimal Policy



- A policy Π is a comp ... ns
- The optimal policy Π ... history with maximal expected utility

This problem is called a Markov Decision Problem (MDP)

How to compute Π*?

Defining Value Function

- Problem:
  - When making a decision, we only know the reward so far, and the possible actions

Defining Value Function

- Problem:
  - When making a decision, we only know the reward so far, and the possible actions
  - We've defined value function retroactively (i.e., the value function/utility of a history/sequence of states is known *once we finish it*)

Defining Value Function

- Problem:
  - When making a decision, we only know the reward so far, and the possible actions
  - We've defined value function retroactively (i.e., the value function/utility of a history/sequence of states is known *once we finish it*)
  - What is the value function of a particular ***state*** in the middle of decision making?

Defining Value Function

- Problem:
  - When making a decision, we only know the reward so far, and the possible actions
  - We've defined value function retroactively (i.e., the value function/utility of a history/sequence of states is known *once we finish it*)
  - What is the value function of a particular ***state*** in the middle of decision making?
  - Need to compute ***expected value function*** of possible future histories/states

Defining Value Function

$$V^{\pi}(s) = \mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \,\middle|\, s_0 = s, \pi\right].$$

$V^{\pi}(s)$ is simply the expected sum of discounted rewards upon starting in state s, and taking actions according to π.[1]

Given a fixed policy π, its value function V^{π} satisfies the **Bellman equations**:

$$V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^{\pi}(s').$$

- What is the value function of a particular ***state*** in the middle of decision making?
- Need to compute ***expected value function*** of possible future histories/states

Dynamic programming

use value functions to structure the search for good policies

Dynamic programming

use value functions to structure the search for good policies

policy evaluation: compute V^{π} from π

policy improvement: improve π based on V^{π}

Slide courtesy/adapted: Peter Bodík

Dynamic programming

use value functions to structure the search for good policies

policy evaluation: compute V^{π} from π

policy improvement: improve π based on V^{π}

start with an arbitrary policy

repeat evaluation/improvement until convergence

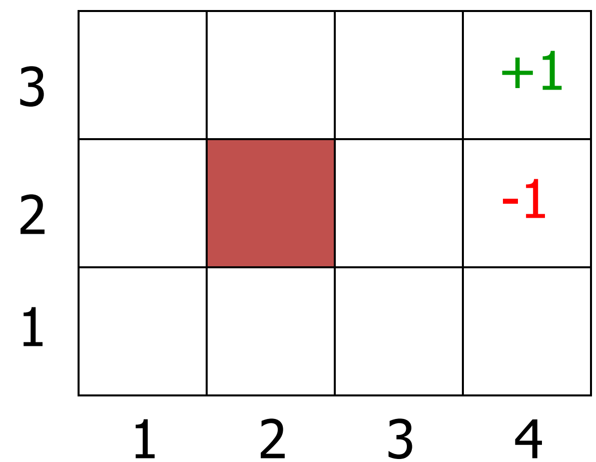
Slide courtesy/adapted: Peter Bodík

Value Iteration

Algorithm 4 Value Iteration

For each state s, initialize $V(s) := 0$.
for until convergence **do**
    For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$

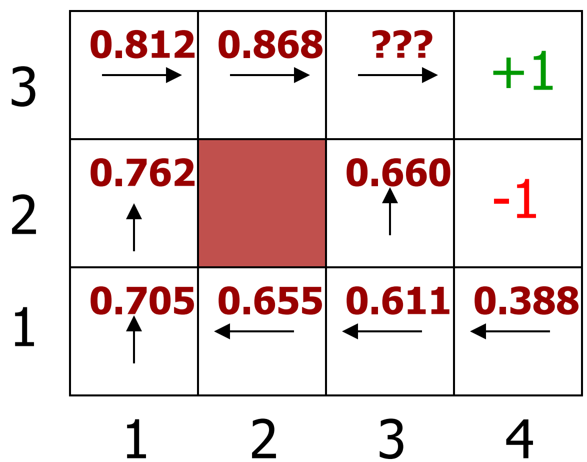


Value Iteration

Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.
2: **for** until convergence **do**
3: For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$

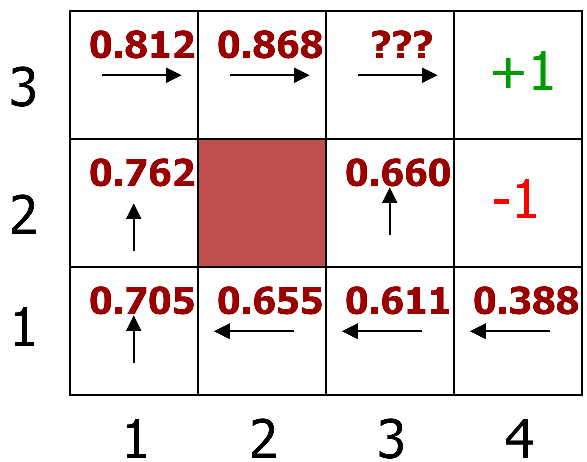


Value Iteration

Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.
2: **for** until convergence **do**
3: For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$



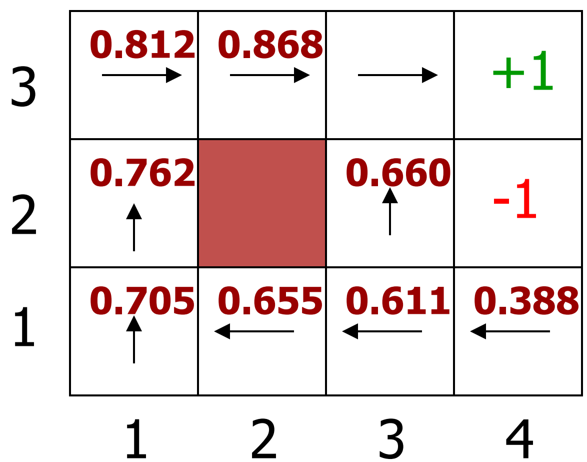
EXERCISE: What is V*([3,3]) (assuming that the other V* are as shown)?

Value Iteration

Algorithm 4 Value Iteration

For each state s, initialize $V(s) := 0$.
for until convergence **do**
    For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$



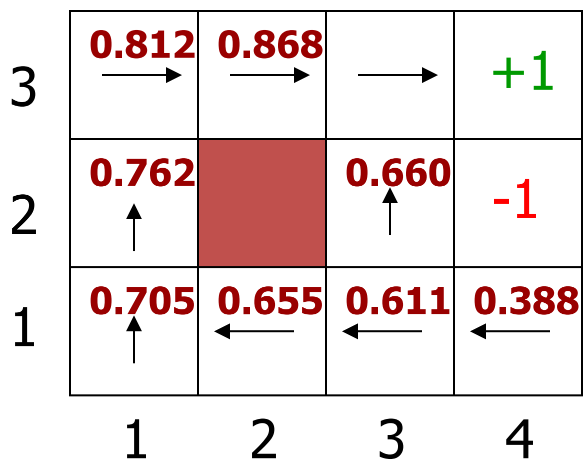
From (3, 3), 3 options: (3, 2), (4, 3), (3, 4) => but there is no (3,4) but wall, so bounced off and remains at (3, 3)

Value Iteration

Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.
2: **for** until convergence **do**
3: For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$



$V^*_{3,3} = R_{3,3} +$
$[P_{3,2}\, V^*_{3,2} + P_{3,3}\, V^*_{3,3} + P_{4,3}\, V^*_{4,3}]$

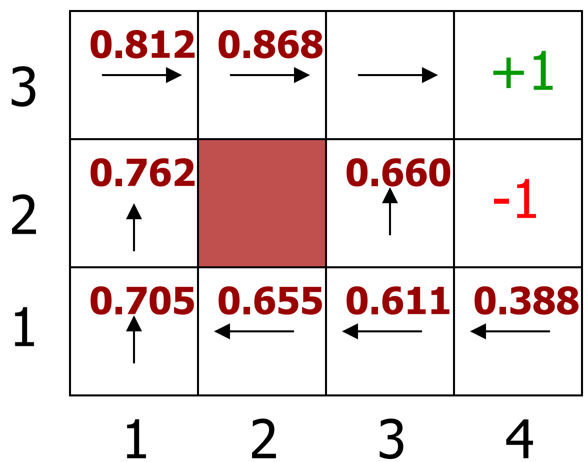
From (3, 3), 3 options: (3, 2), (4, 3), (3, 4) => but there is no (3,4) but wall, so bounced off and remains at (3, 3)

Value Iteration

Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.
2: **for** until convergence **do**
3: For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$



$V^*_{3,3} = R_{3,3} +$
$[P_{3,2}\, V^*_{3,2} + P_{3,3}\, V^*_{3,3} + P_{4,3}\, V^*_{4,3}]$
$= -0.04 +$
$[0.1*0.660 + 0.1*0.918 + 0.8*1]$

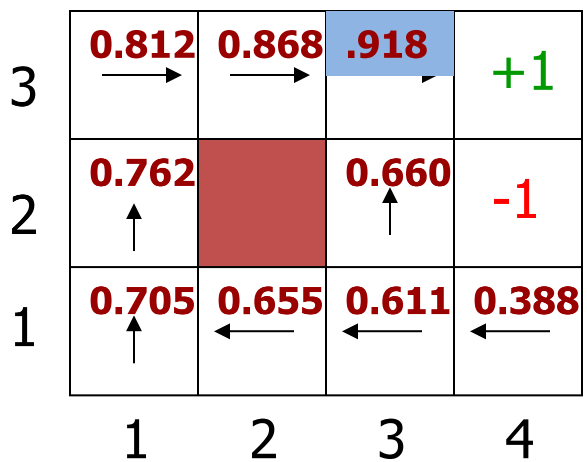
From (3, 3), 3 options: (3, 2), (4, 3), (3, 4) => but there is no (3,4) but wall, so bounced off and remains at (3, 3)

Value Iteration

Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.
2: **for** until convergence **do**
3: For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$



$V^*_{3,3} = R_{3,3} +$
$[P_{3,2}\, V^*_{3,2} + P_{3,3}\, V^*_{3,3} + P_{4,3}\, V^*_{4,3}]$
$= -0.04 +$
$[0.1*0.660 + 0.1*0.918 + 0.8*1]$

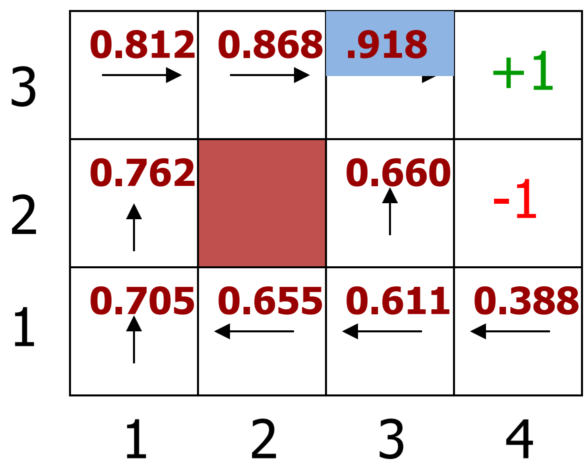
From (3, 3), 3 options: (3, 2), (4, 3), (3, 4) => but there is no (3,4) but wall, so bounced off and remains at (3, 3)

Value Iteration

Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.
2: **for** until convergence **do**
3: For every state, update

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$



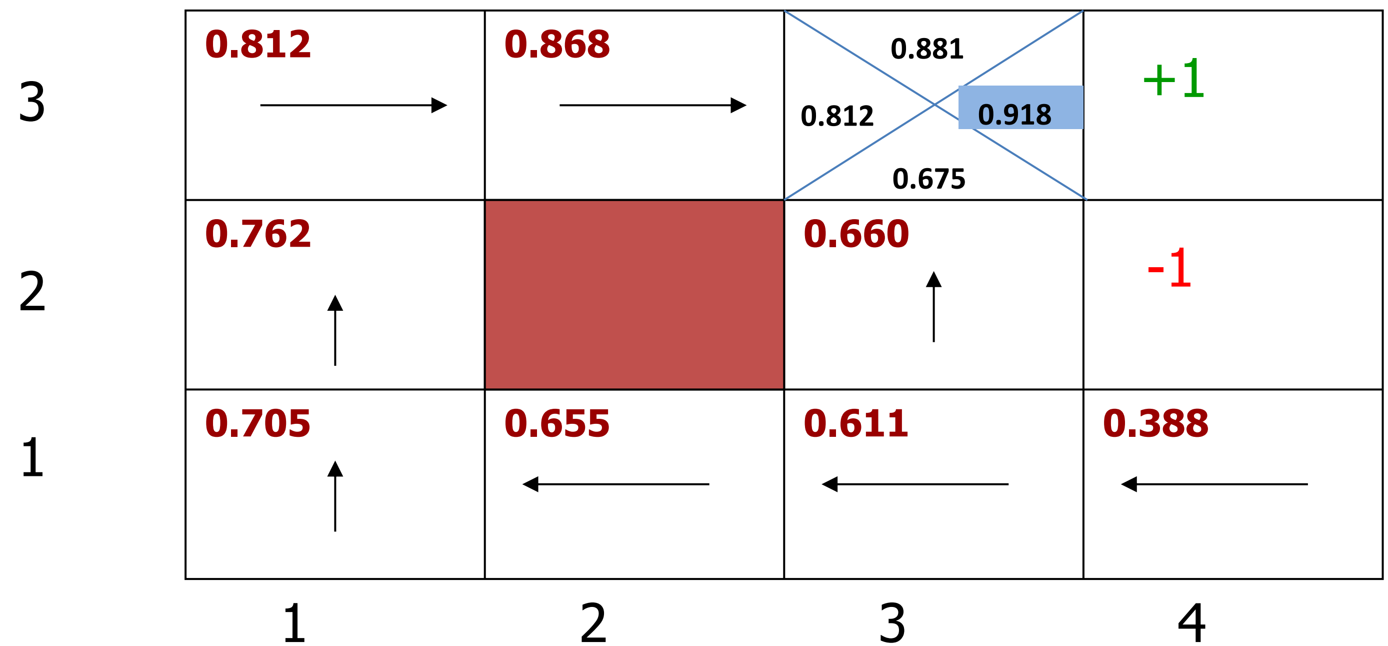
$V^*_{3,3} = R_{3,3} +$
$[P_{3,2}\ V^*_{3,2} + P_{3,3}\ V^*_{3,3} + P_{4,3}\ V^*_{4,3}]$
$= -0.04 +$
$[0.1*0.660 + 0.1*0.918 + 0.8*1]$

From (3, 3), 3 options: (3, 2), (4, 3), (3, 4) => but there is no (3,4) but wall, so bounced off and remains at (3, 3)

More Breakdown

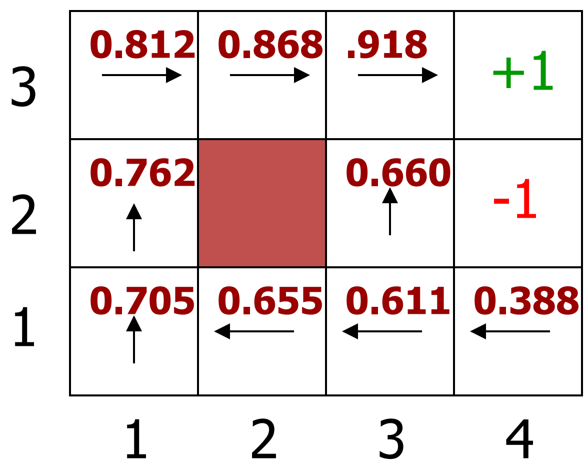
Value Iteration

In (3, 3), since → action gave us the **maximum expected future reward,** we choose to keep → in our policy. Same thing was done for all states.



Optimal Policy

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s').$$



Whichever is higher becomes next action for (3, 1)

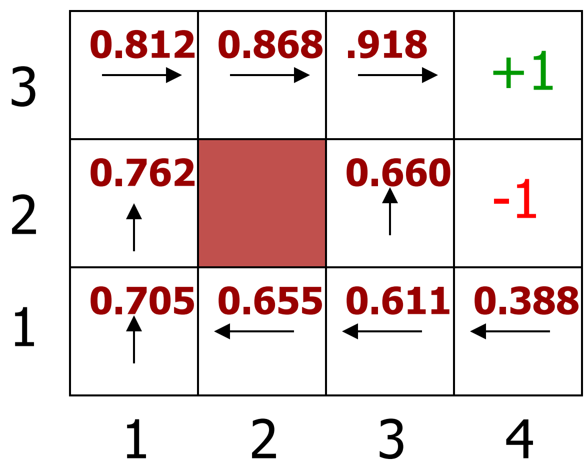
Optimal Policy

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s').$$

$\pi^*_{3,1}$ being (←) =

$P_{up}\ V^*_{2,1} + P_{left}\ V^*_{3,1}$ (Bounced off) $+ P_{right}\ V^*_{3,2}$

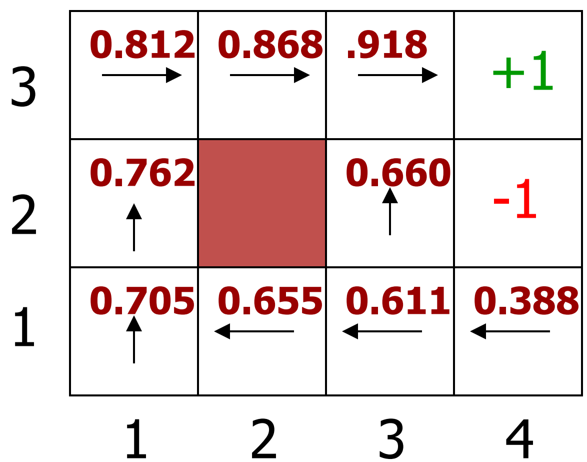
= 0.8 * 0.655 + 0.1 * 0.611 + 0.1 * 0.66



Whichever is higher becomes
next action for (3, 1)

Optimal Policy

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s').$$



$\pi^*_{3,1}$ being (←) =
$P_{up}\ V^*_{2,1} + P_{left}\ V^*_{3,1}$ (Bounced off) $+ P_{right}\ V^*_{3,2}$
$= 0.8 * 0.655 + 0.1 * 0.611 + 0.1 * 0.66$

$\pi^*_{3,1}$ being (↑) =
$P_{up}\ V^*_{3,2} + P_{left}\ V^*_{2,1} + P_{right}\ V^*_{1,4}$

Whichever is higher becomes next action for (3, 1)

Policy Iteration

- Pick a policy Π at random
- Repeat:
  - Compute Value function of each state for Π

$$V(s) := V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s') V^{\pi}(s').$$

  - Compute the policy Π' given these value functions

$$\pi'(s) := \arg\max_{a \in A} \sum_{s'} P_{sa}(s') V(s').$$

  - If $\Pi' = \Pi$ then return Π

Policy Iteration

- Pick a policy Π at random
- Repeat:
  - Compute Value function of each state for Π

$$V(s) := V^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P_{s\pi(s)}(s')V^{\pi}(s').$$

  - Compute the policy Π' given these value functions

Or solve the set of linear equations:
(often a sparse system)

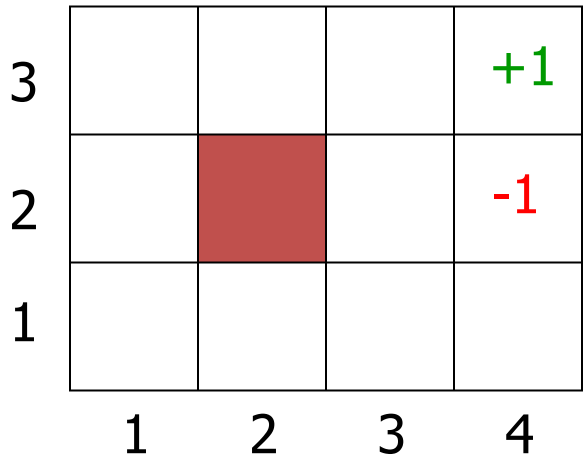
$$\pi'(s) := \arg\max_{a \in A} \sum_{s'} P_{sa}(s')V(s').$$

  - If $\Pi' = \Pi$ then return Π

Advanced topic

Infinite Horizon

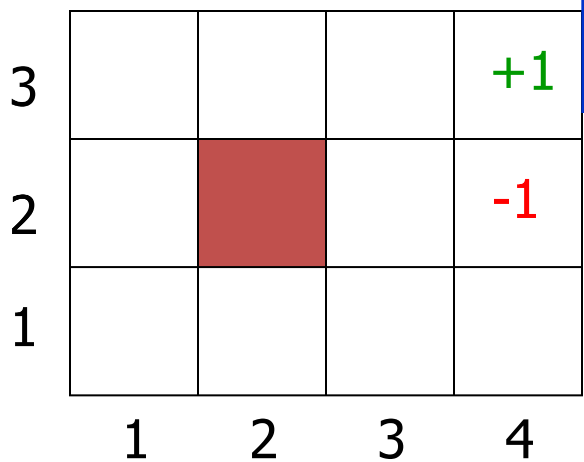
In many problems, e.g., the robot navigation example, histories are potentially unbounded and the same state can be reached many times



Advanced topic

Infinite Horizon

In many problems, e.g., the robot navigation example, histories are potentially unbounded and the same state can be reached many times

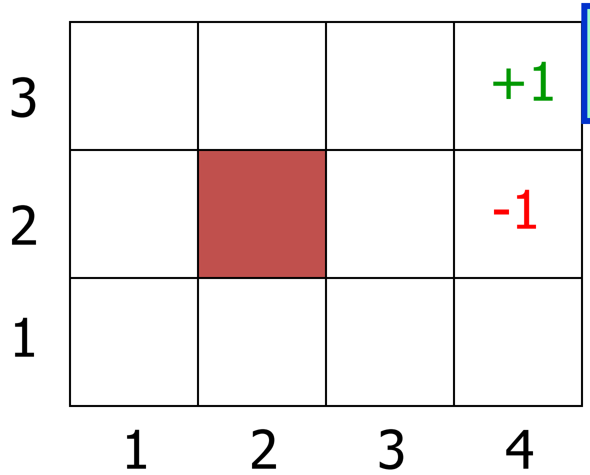


What if the robot lives forever?

Advanced topic

Infinite Horizon

In many problems, e.g., the robot navigation example, histories are potentially unbounded and the same state can be reached many times



What if the robot lives forever?

One trick:
Use discounting to make an infinite horizon problem mathematically tractable

Value Iteration: Summary

- Initialize state values (expected utilities) randomly
- Repeatedly update state values using best action, according to current approximation of state values
- Terminate when state values stabilize
- Resulting policy will be the best policy because it's based on accurate state value estimation

Policy Iteration: Summary

– Initialize policy randomly
– Repeatedly update state values using best action, according to current approximation of state values
– Then update policy based on new state values
– Terminate when policy stabilizes
– Resulting policy is the best policy, but state values may not be accurate (may not have converged yet)
– Policy iteration is often faster (because we don't have to get the state values right)

- **Both methods have a major weakness: They require us to know the transition function exactly in advance!**

Exploration vs. Exploitation

- Problem with naïve reinforcement learning:
  - What action to take?
  - **Best apparent action, based on learning to date** } Exploitation
    - Greedy strategy
    - Often prematurely converges to a suboptimal policy!
  - **Random (or unknown) action** } Exploration
    - Will cover entire state space
    - Very expensive and slow to learn!
    - When to stop being random?
  - Balance exploration (try random actions) with exploitation (use best action so far)

More on Exploration

- Agent may sometimes choose to explore suboptimal moves in hopes of finding better outcomes
  - Only by visiting all states frequently enough can we guarantee learning the true values of all the states
- When the agent is **learning**, ideal would be to get accurate values for all states
  - Even though that may mean getting a negative outcome
- When agent is **performing**, ideal would be to get optimal outcome
- A learning agent should have an **exploration policy**

Exploration Policy

- Wacky approach (exploration): act randomly in hopes of eventually exploring entire environment
  - Choose any legal checkers move
- Greedy approach (exploitation): act to maximize utility using current estimate
  - Choose moves that have in the past led to wins
- Reasonable balance: act more wacky (exploratory) when agent has little idea of environment; more greedy when the model is close to correct
  - Suppose you know no checkers strategy?
  - What's the best way to get better?

Overview: Learning Strategies

Dynamic Programming

Q-learning

Monte Carlo approaches

Q-learning

$$Q: (s, a) \rightarrow \mathbb{R}$$

Goal: learn a function that computes a "goodness" score for taking a particular action a in state s

Q-learning

previous algorithms: on-policy algorithms

start with a random policy, iteratively improve

converge to optimal

Q-learning: off-policy

use any policy to estimate Q

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Q directly approximates Q* (Bellman optimality equation)

independent of the policy being followed

only requirement: keep updating each (s,a) pair

Slide courtesy/adapted: Peter Bodík

Q-learning

previous algorithms: on-policy algorithms

- start with a random policy, iteratively improve
- converge to optimal

Q-learning: off-policy

- use any policy to estimate Q

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[\underbrace{r_{t+1}}_{R(s_t)} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

- Q directly approximates Q* (Bellman optimality equation)
- independent of the policy being followed
- only requirement: keep updating each (s,a) pair

Deep/Neural Q-learning

$$Q(s, a; \theta) \approx Q^*(s, a)$$

neural network　　　　desired optimal solution

Deep/Neural Q-learning

$$Q(s, a; \theta) \approx Q^*(s, a)$$

neural network desired optimal solution

Approach: Form (and learn) a neural network to model our optimal Q function

Deep/Neural Q-learning

Learn weights
(parameters) θ of our
neural network

$$Q(s, a; \theta) \approx Q^*(s, a)$$

neural network

desired optimal solution

Approach: Form (and learn)
a neural network to model
our optimal Q function

Overview: Learning Strategies

Dynamic Programming

Q-learning

Monte Carlo approaches

Monte Carlo policy evaluation

don’t need full knowledge of environment (just (simulated) experience)

want to estimate $V^{\pi}(s)$

Slide courtesy/adapted: Peter Bodík

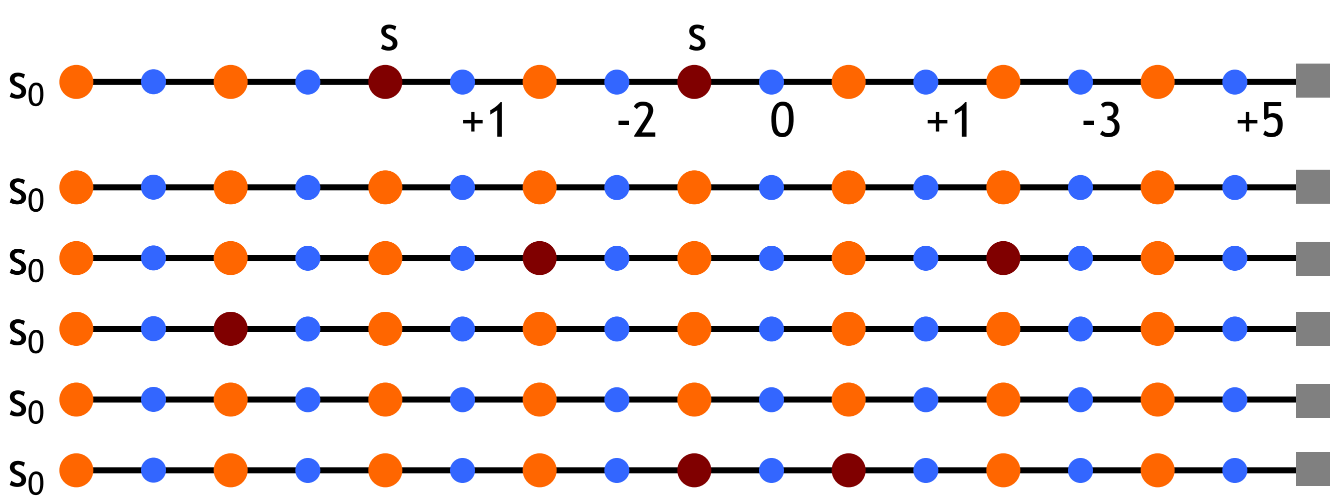
Monte Carlo policy evaluation

don't need full knowledge of environment (just **(simulated)** experience)

want to estimate $V^{\pi}(s)$

expected return starting from s and following π

estimate as average of observed returns in state s



$R_1(s) = +2$

$R_2(s) = +1$

$R_3(s) = -5$

$R_4(s) = +4$

$V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

Slide courtesy/adapted: Peter Bodík

Maintaining exploration

key ingredient of RL

deterministic/greedy policy won't explore all actions

- don't know anything about the environment at the beginning
- need to try all actions to find the optimal one

maintain exploration

- use *soft* policies instead: $\pi(s,a)>0$ (for all s,a)

ε-greedy policy

- with probability 1-ε perform the optimal/greedy action
- with probability ε perform a random action

- will keep exploring the environment
- slowly move it towards greedy policy: ε -> 0

RL Summary 1:

- **Reinforcement learning systems**
  - Learn **series** of actions or decisions, rather than a single decision
  - Based on feedback given at the end of the series
- A reinforcement learner has
  - A goal
  - Carries out trial-and-error search
  - Finds the best paths toward that goal

RL Summary 2:

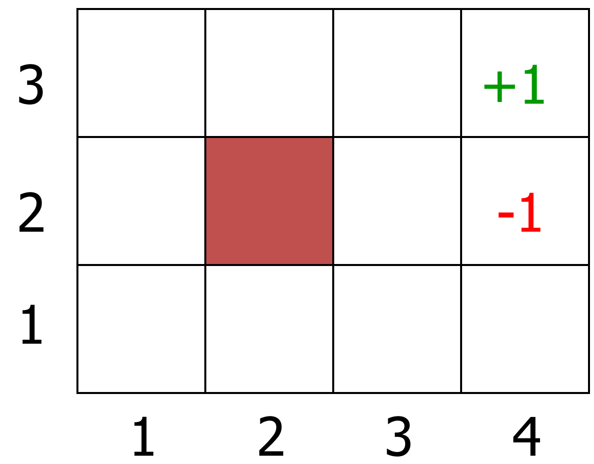
- A typical reinforcement learning system is an active agent, interacting with its environment.
- It must balance:
  - Exploration: trying different actions and sequences of actions to discover which ones work best
  - Exploitation (achievement): using sequences which have worked well so far
- Must learn **successful sequences of actions** in an uncertain environment

RL Summary 3

- Very hot area of research at the moment
- There are **many** more sophisticated RL algorithms
  - Most notably: probabilistic approaches
- Applicable to game-playing, search, finance, robot control, driving, scheduling, diagnosis, …

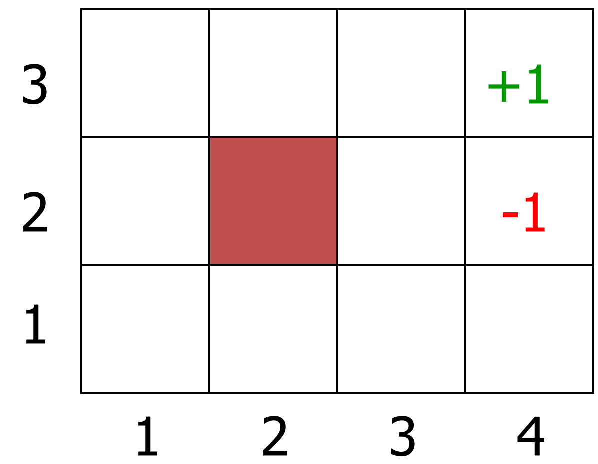
EXTRA SLIDES

Utility Function



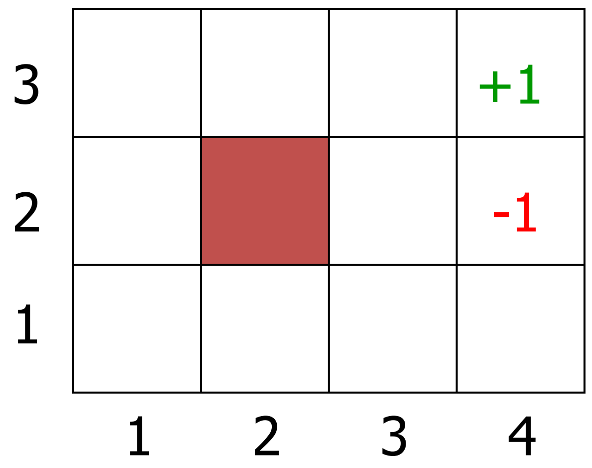
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape

Utility Function



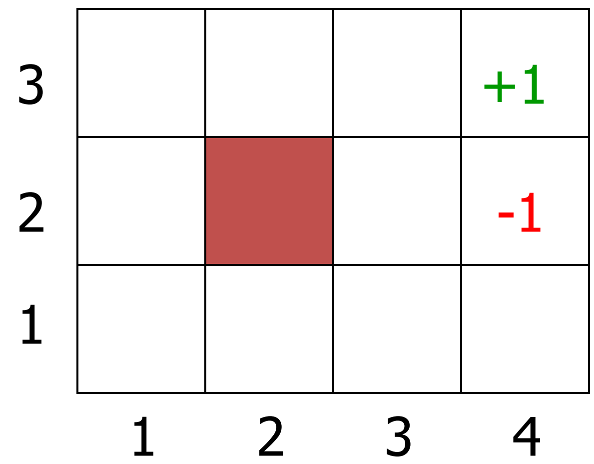
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries

Utility Function



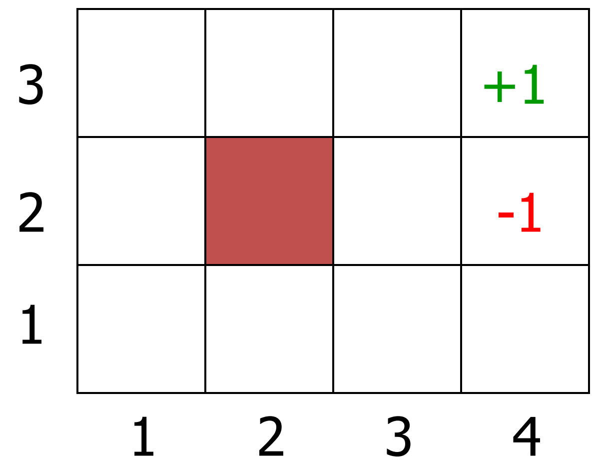
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] and [4,2] are terminal states

Utility Function



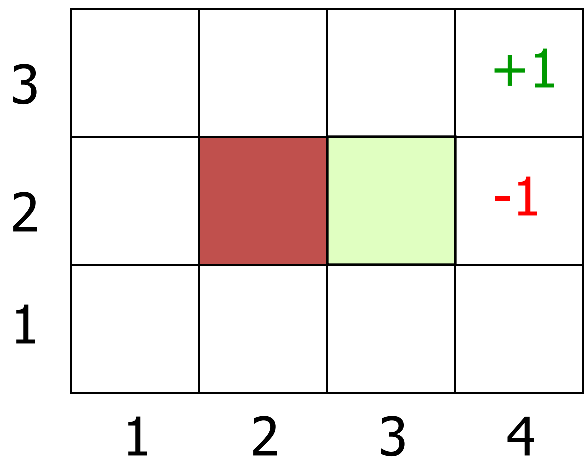
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] and [4,2] are terminal states
- Histories have utility!

Utility of a History



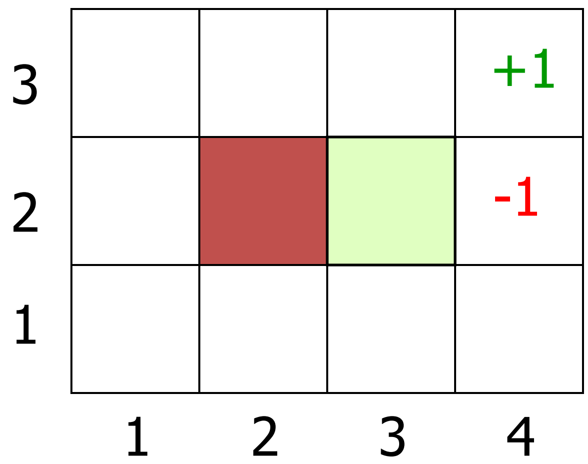
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] or [4,2] are terminal states
- Histories have utility!
- The utility of a history is defined by the utility of the last state (+1 or –1) minus n/25, where n is the number of moves
  - Many utility functions possible, for many kinds of problems.

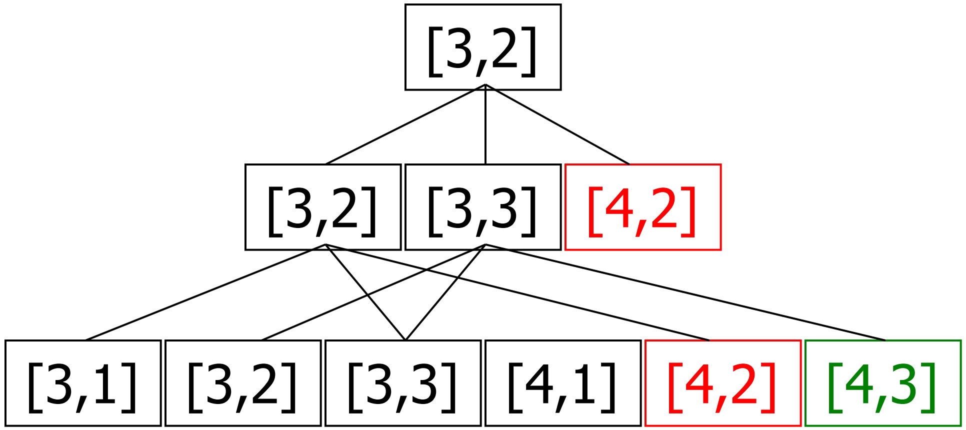
Utility of an Action Sequence



- Consider the action sequence (U,R) from [3,2]

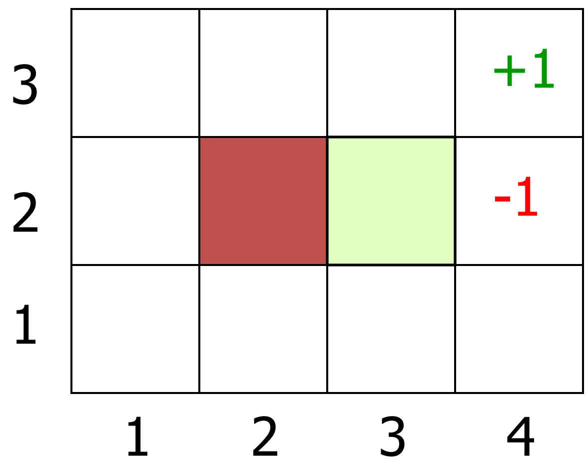
Utility of an Action Sequence

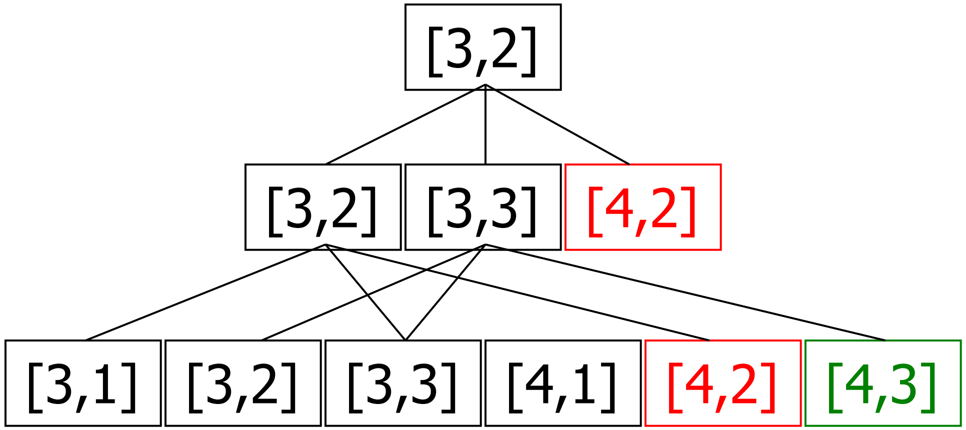




- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability

Utility of an Action Sequence

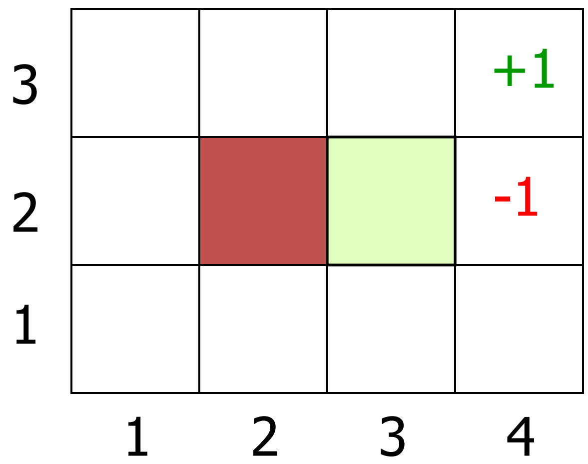


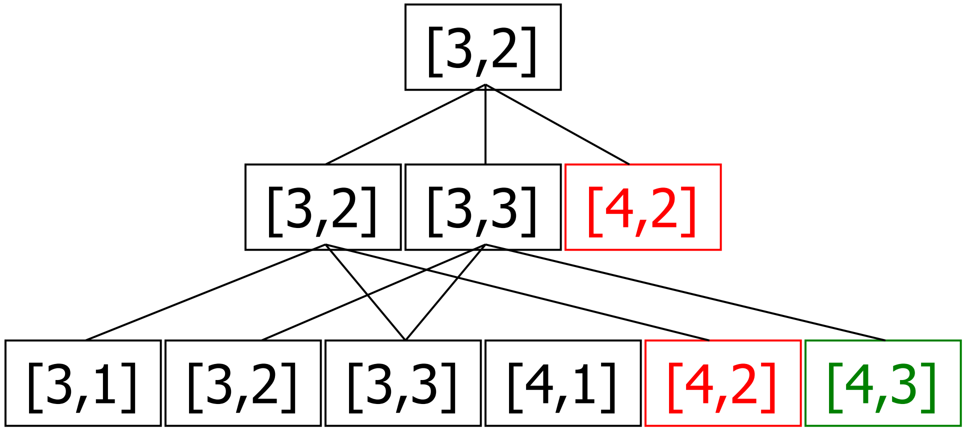


- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories:

$$\mathcal{U} = \Sigma_h \mathcal{U}_h \, \mathbf{P}(h)$$

Optimal Action Sequence



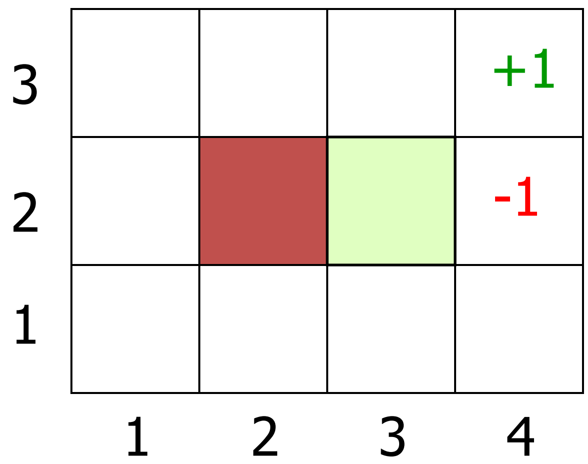


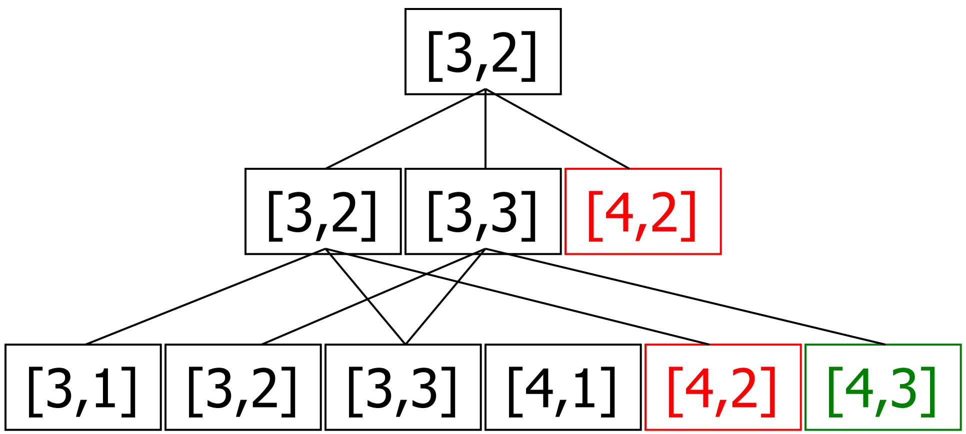
- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories:

$$\mathcal{U} = \Sigma_h \mathcal{U}_h \mathbf{P}(h)$$

- The optimal sequence is the one with maximal utility

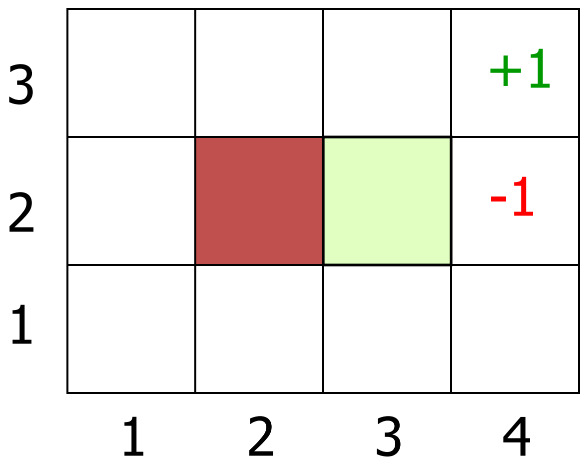
Optimal Action Sequence

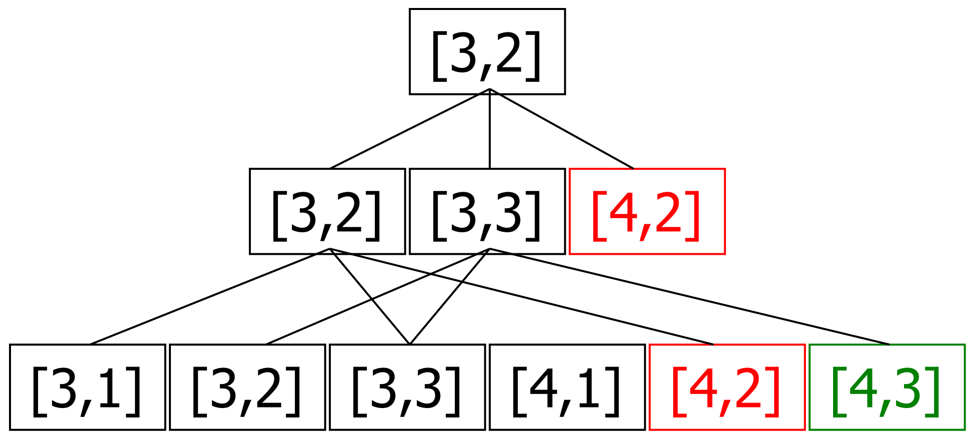




- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories
- The optimal sequence is the one with maximal utility
- **But is the optimal action sequence what we want to compute?**

Optimal Action Sequence





- Consider the action sequence (U,R) from [3,2]
- A run produc... ...ability
- The utility of the sequence is the expected utility of the histories
- The optimal sequence is the one with maximal utility
- **But is the optimal action sequence what we want to compute?**

only if the sequence is executed blindly!