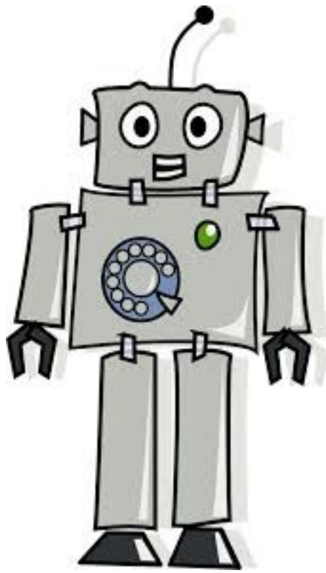
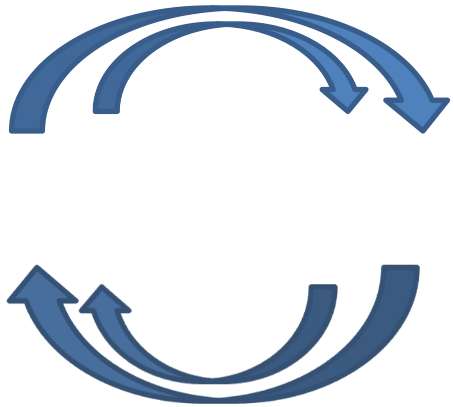
CMSC 478: Reinforcement Learning

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

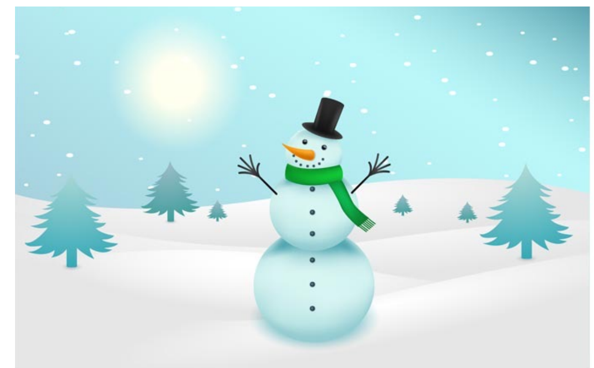
Markov Decision Process: Formalizing Reinforcement Learning

take action





get new state
and/or reward



environment

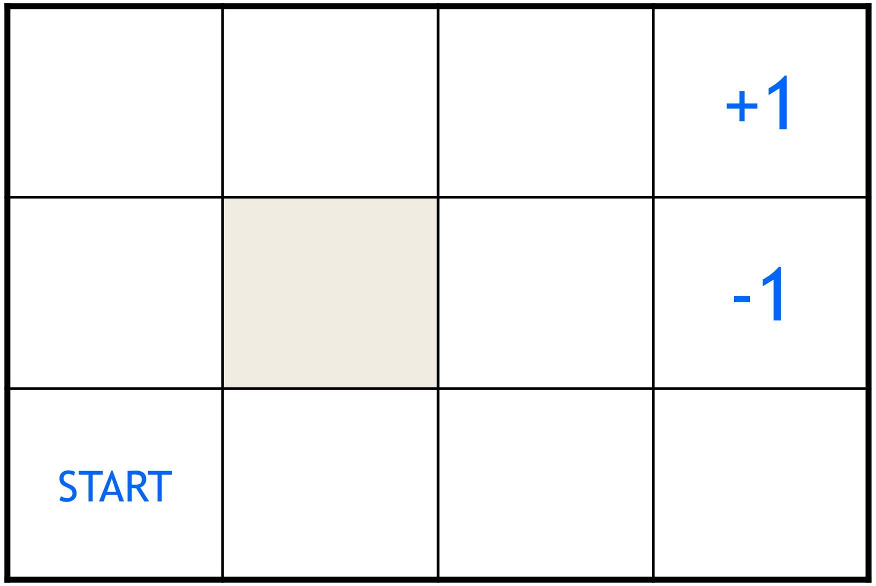
agent

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

Robot image: openclipart.org

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

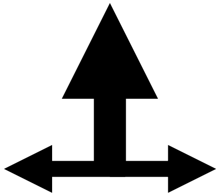
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

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

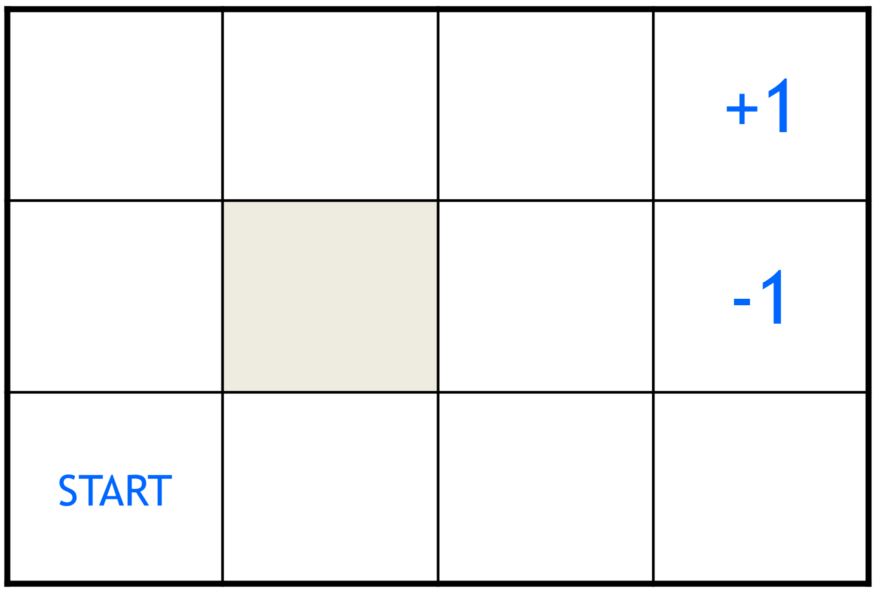


reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

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

Slide courtesy Peter Bodík

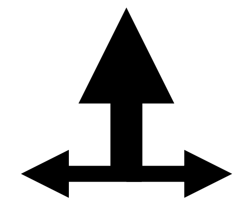
Robot in a room



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

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

Overview: Learning Strategies

Dynamic Programming

Q-learning

Monte Carlo approaches

Dynamic programming

use value functions to structure the search for good policies

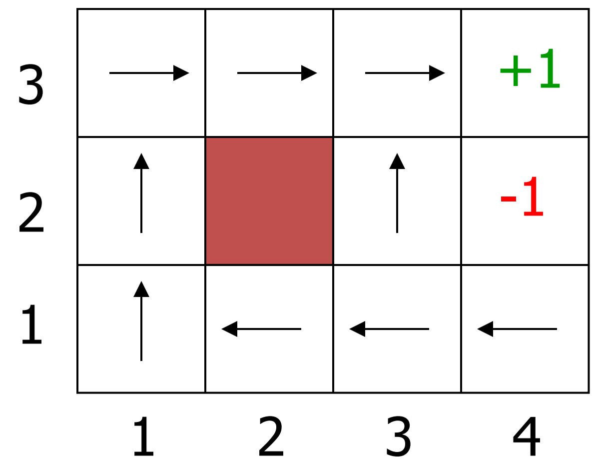
policy evaluation: compute V^{π} from π

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

start with an arbitrary policy

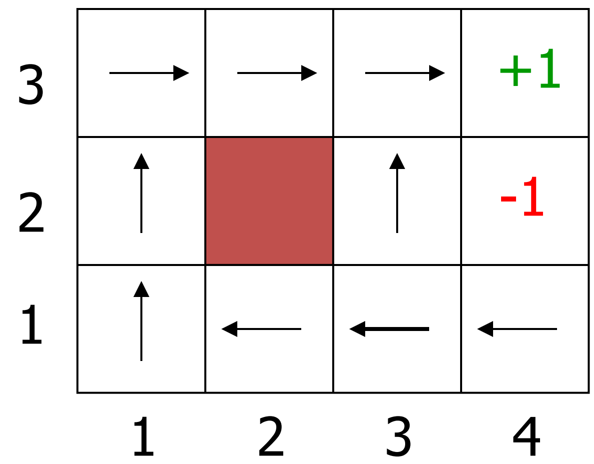
repeat evaluation/improvement until convergence

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
  - 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**:

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

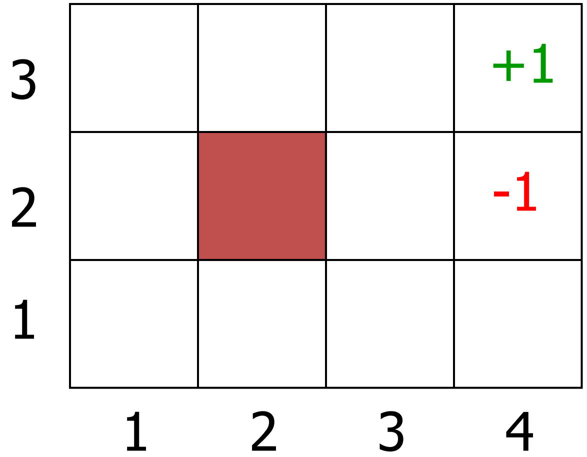
- What is the value function of a particular ***state*** in the middle of decision making?
- Need to compute ***expected value function*** of possible future histories/states

Value Iteration

Algorithm 4 Value Iteration

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

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

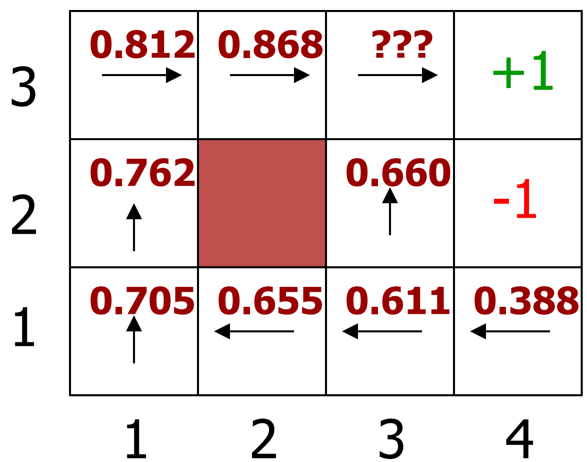


Value Iteration

Algorithm 4 Value Iteration

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

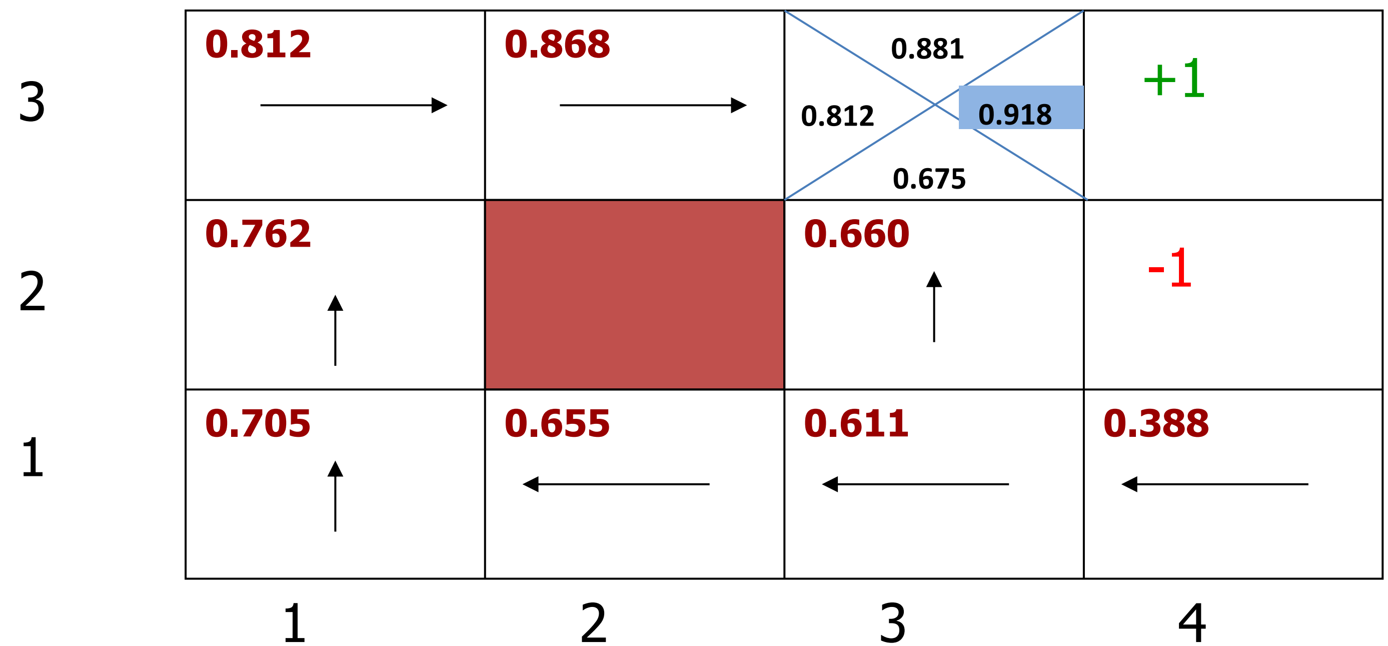
$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{15.4}$$



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.



Value Iteration

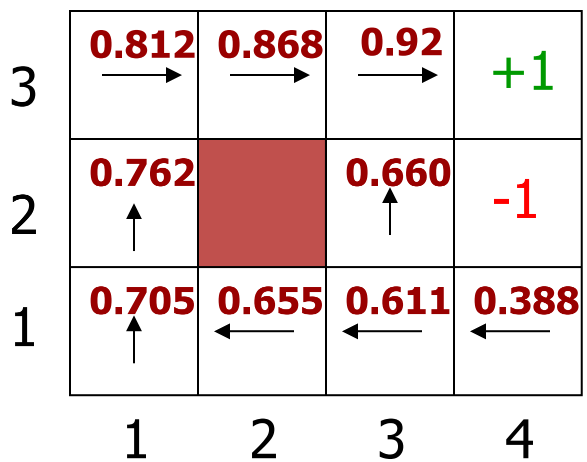
Algorithm 4 Value Iteration

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

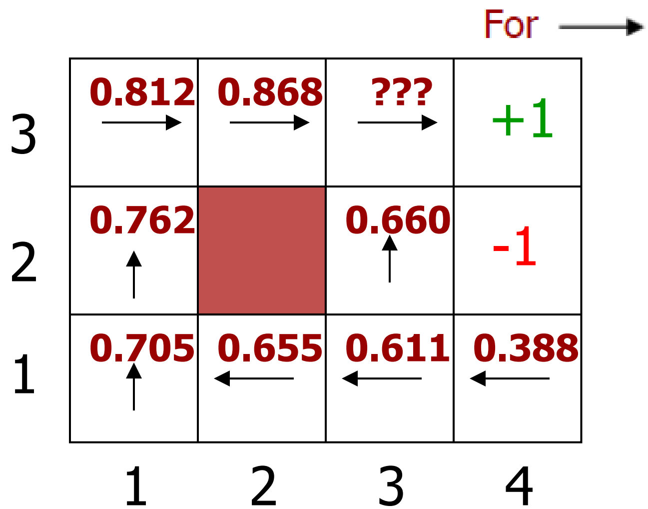
Algorithm 4 Value Iteration

1: For each state s, initialize $V(s) := 0$.

2: **for** until convergence **do**

3: For every state, update

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



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)

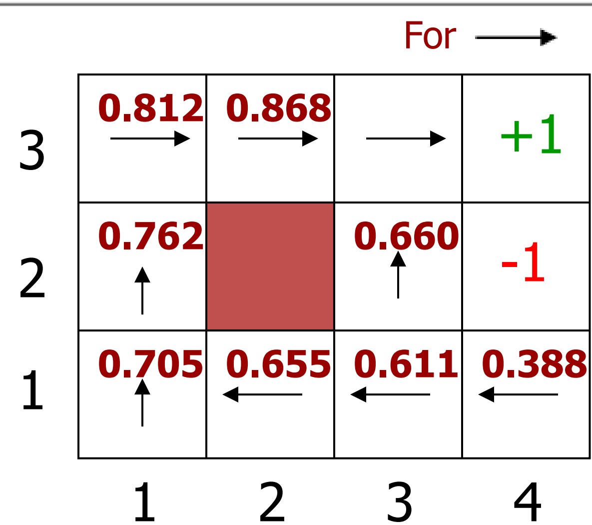
EXERCISE: What is **next** V*([3,3]) (assuming that other V* are as shown)?

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

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



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

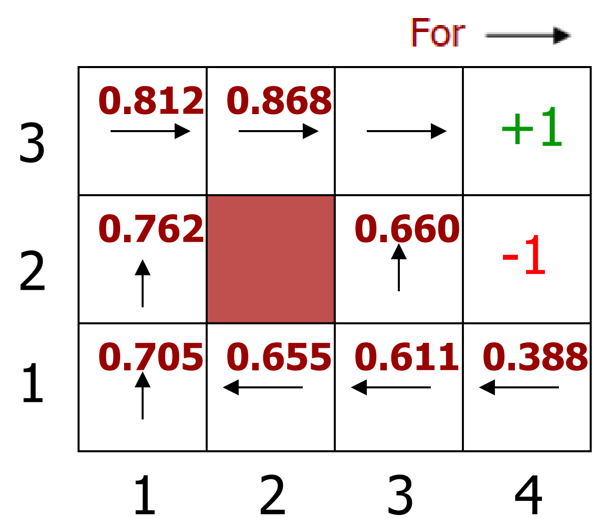
From (3, 3), 3 options: (3, 2), (4, 3), (3, 4) => but there is no (3,4) but wall, so bounced off and remains at (3, 3)

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

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



$V^*_{3,3} = R_{3,3} +$
$[P_{3,2}\ V^*_{3,2} + P_{3,3}\ V^*_{3,3} + P_{4,3}\ V^*_{4,3}]$
$= -0.04 +$
$[0.1*0.660 + 0.1*0.92 + 0.8*1]$

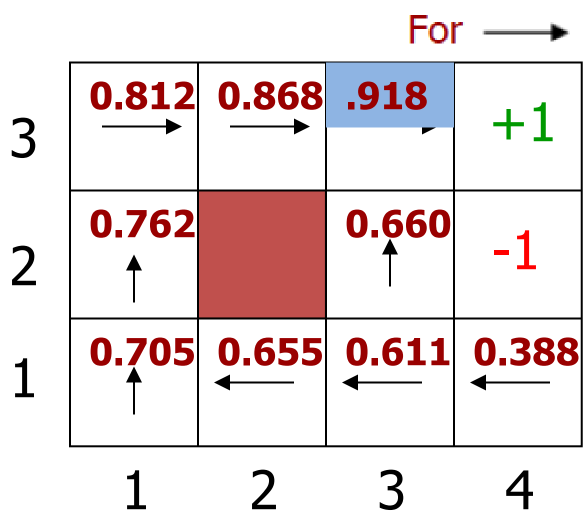
From (3, 3), 3 options: (3, 2), (4, 3), (3, 4) => but there is no (3,4) but wall, so bounced off and remains at (3, 3)

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

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

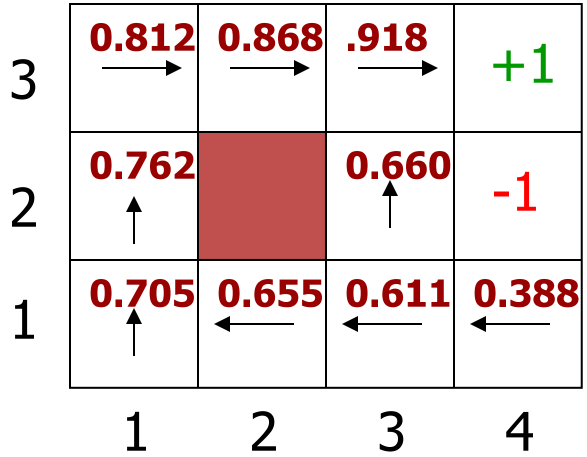


$V^*_{3,3} = R_{3,3} +$
$[P_{3,2}\, V^*_{3,2} + P_{3,3}\, V^*_{3,3} + P_{4,3}\, V^*_{4,3}]$
$= -0.04 +$
$[0.1*0.660 + 0.1*0.92 + 0.8*1]$

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)

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 $(\leftarrow)$ =

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

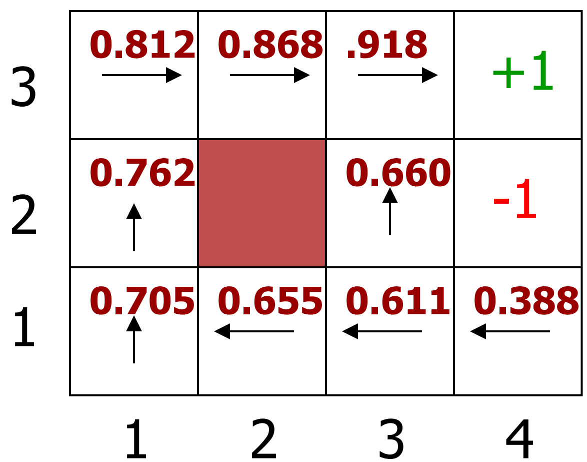
= 0.8 * 0.655 + 0.1 * 0.611 + 0.1 * 0.66

	1	2	3	4
3	0.812 →	0.868 →	.918 →	+1
2	0.762 ↑		0.660 ↑	-1
1	0.705 ↑	0.655 ←	0.611 ←	0.388 ←

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 ($\leftarrow$) =

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

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

$\pi^*_{3,1}$ being ($\uparrow$) =

$P_{up}\ V^*_{3,2} + P_{left}\ V^*_{2,1} + P_{right}\ V^*_{1,4}$

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

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

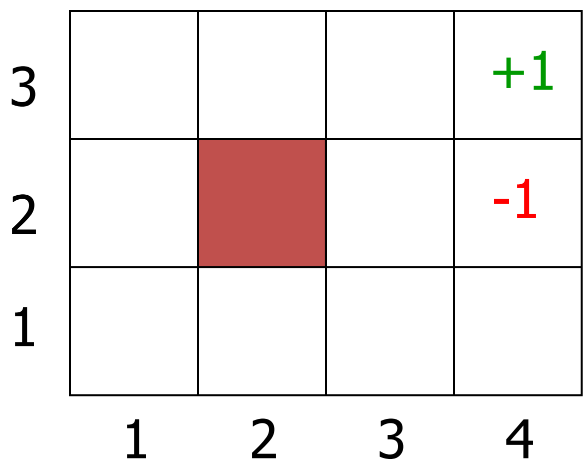
– Initialize policy randomly
– Repeatedly update state values using best action, according to current approximation of state values
– Then update policy based on new state values
– Terminate when policy stabilizes
– Resulting policy is the best policy, but state values may not be accurate (may not have 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!**

Advanced topic

Infinite Horizon

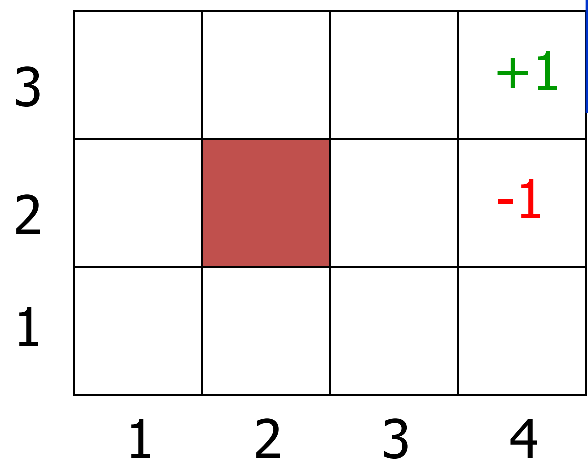
In many problems, e.g., the robot navigation example, histories are potentially unbounded and the same state can be reached many times



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

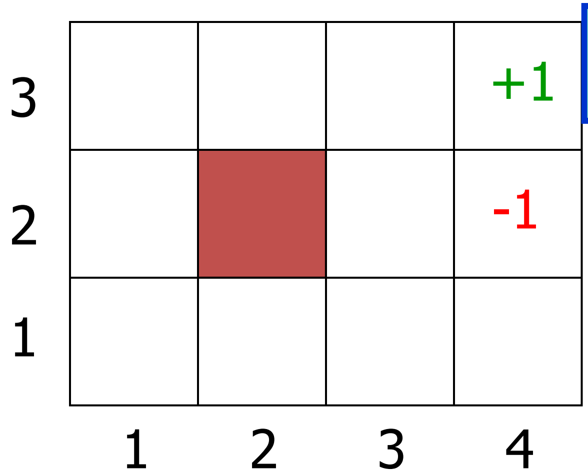


What if the robot lives forever?

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

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?

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

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

Q-learning

previous algorithms: on-policy algorithms

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

Learning rate, can be constant or a function

Q-learning: off-policy

- use any policy to estimate Q

R(s_t)

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

- 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

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 $V^{\pi}(s)$

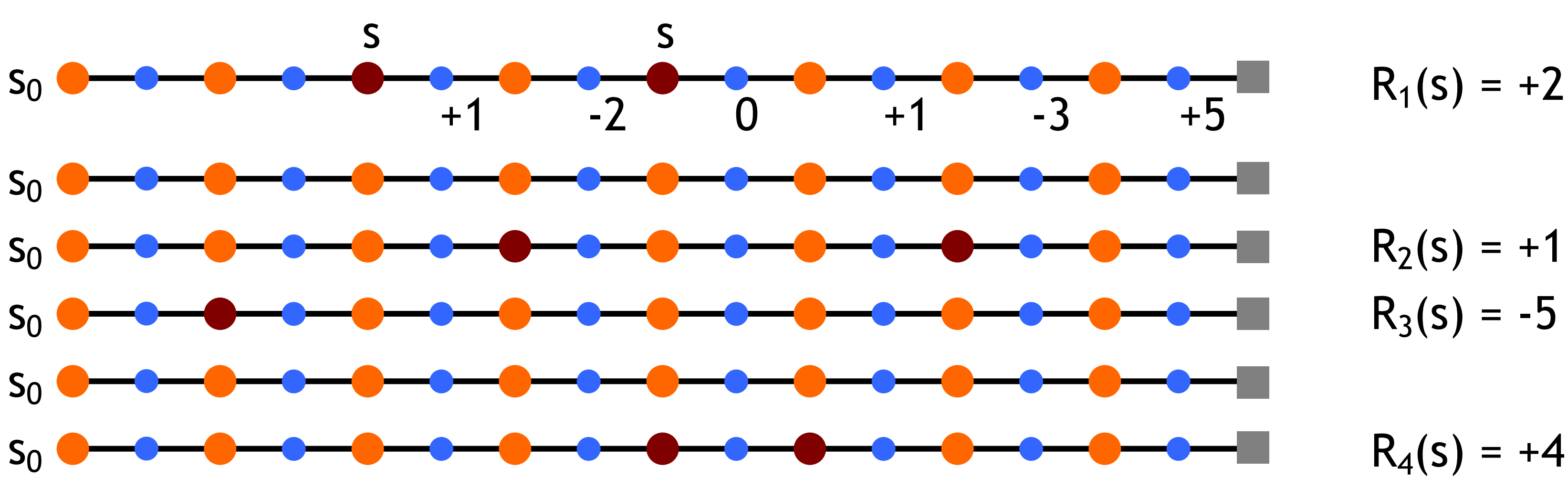
Monte Carlo policy evaluation

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



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

Slide courtesy/adapted: Peter Bodík

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:

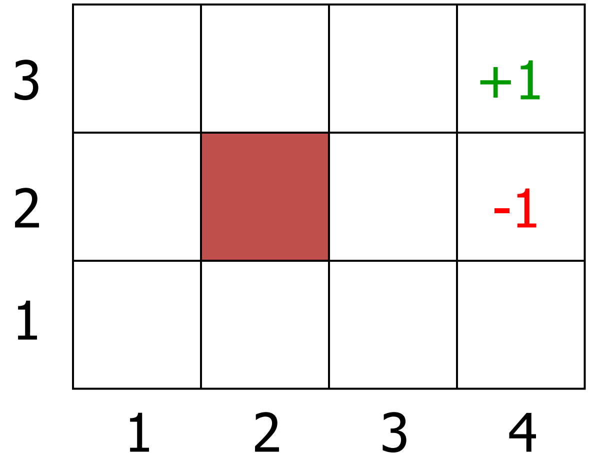
- A typical reinforcement learning system is an active agent, interacting with its environment.
- It must balance:
  - Exploration: trying different actions and sequences of actions to discover which ones work best
  - Exploitation (achievement): using sequences which have worked well so far
- Must learn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

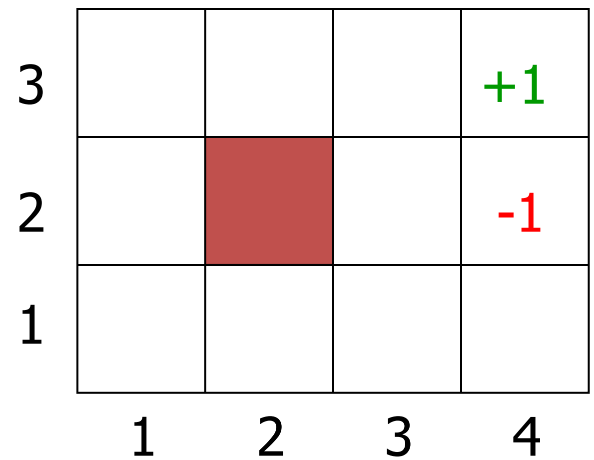
EXTRA SLIDES

Utility Function



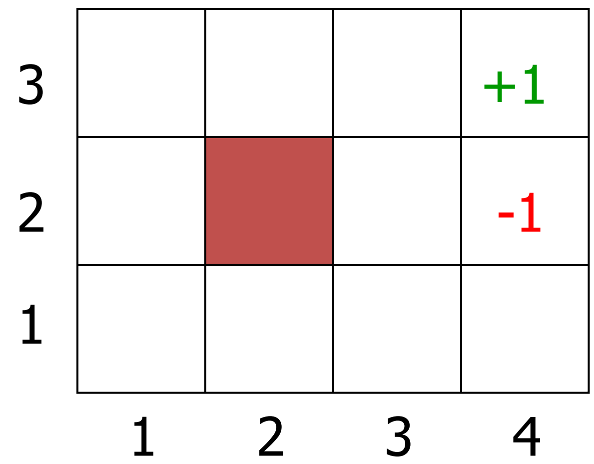
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape

Utility Function



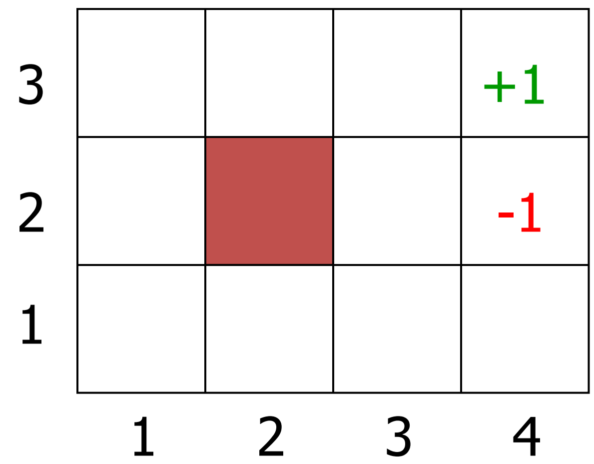
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries

Utility Function



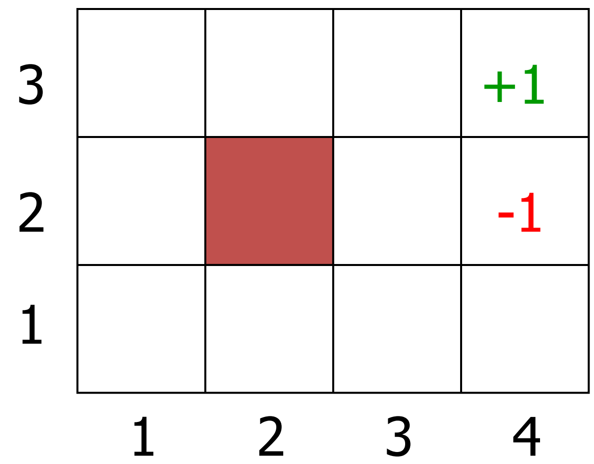
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] and [4,2] are terminal states

Utility Function



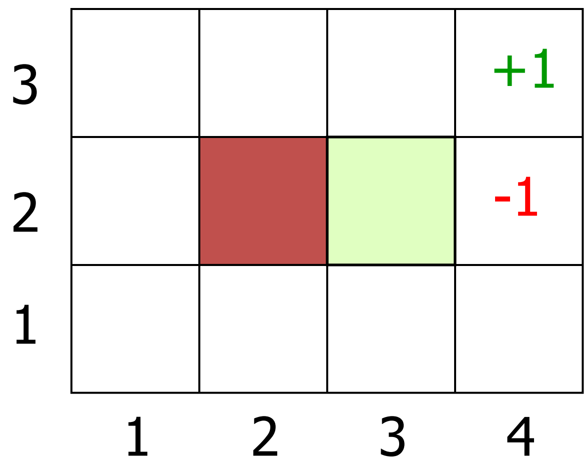
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] and [4,2] are terminal states
- Histories have utility!

Utility of a History



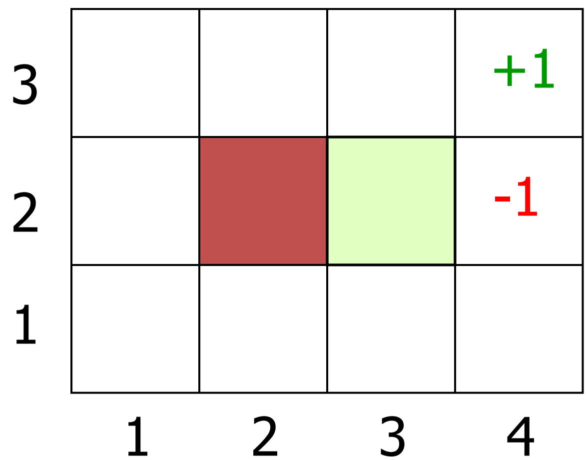
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape
- The robot needs to recharge its batteries
- [4,3] or [4,2] are terminal states
- Histories have utility!
- The utility of a history is defined by the utility of the last state (+1 or –1) minus n/25, where n is the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

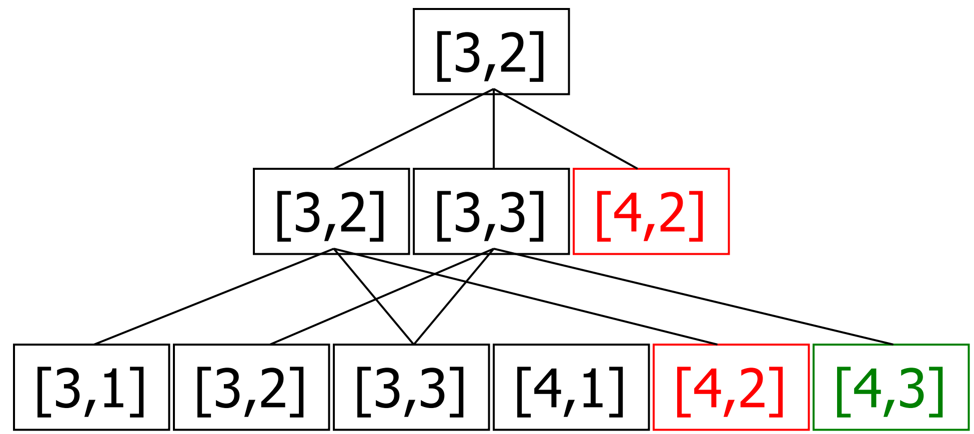
Utility of an Action Sequence



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

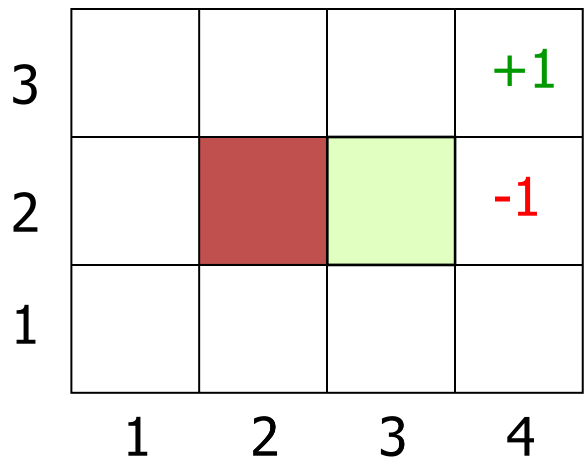
Utility of an Action Sequence

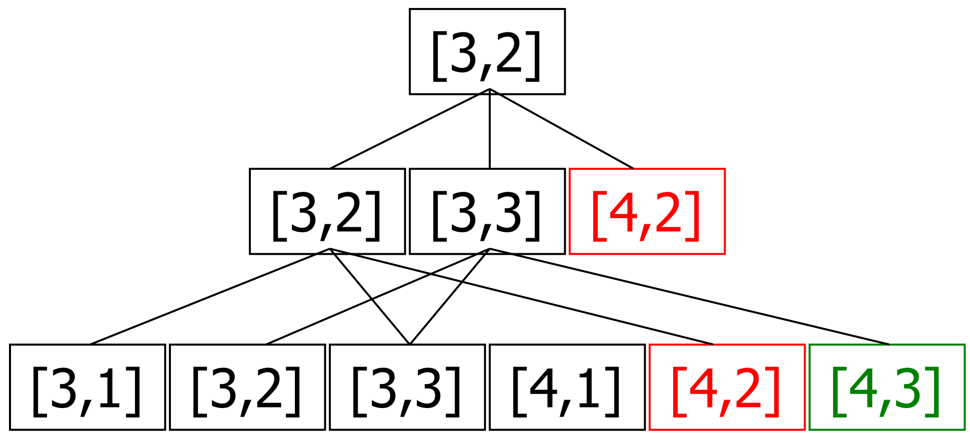




- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability

Utility of an Action Sequence

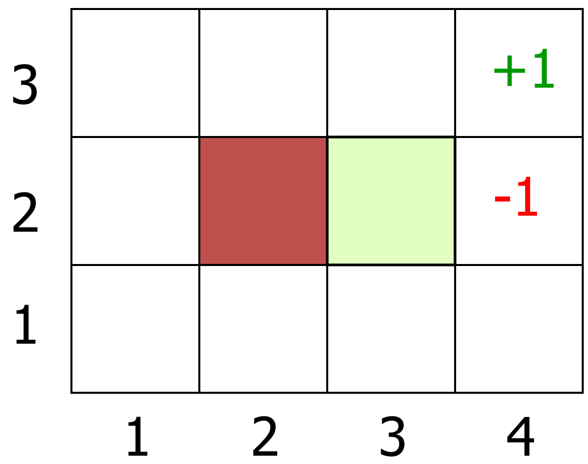


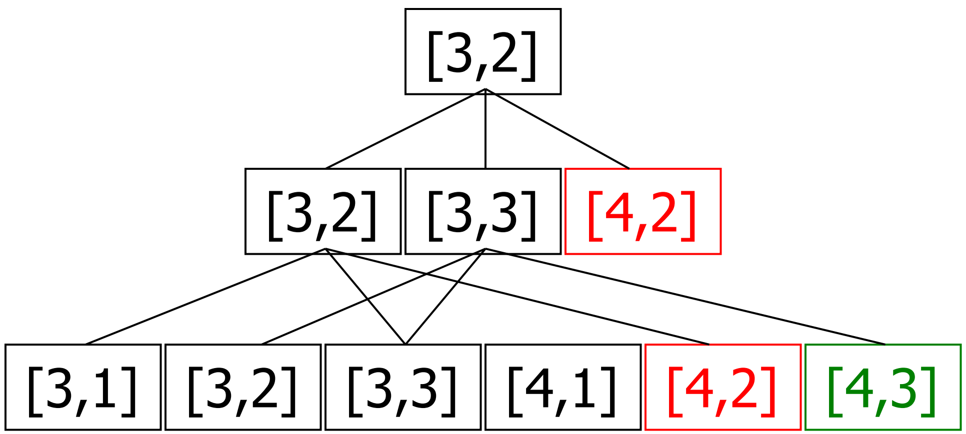


- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories:

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

Optimal Action Sequence



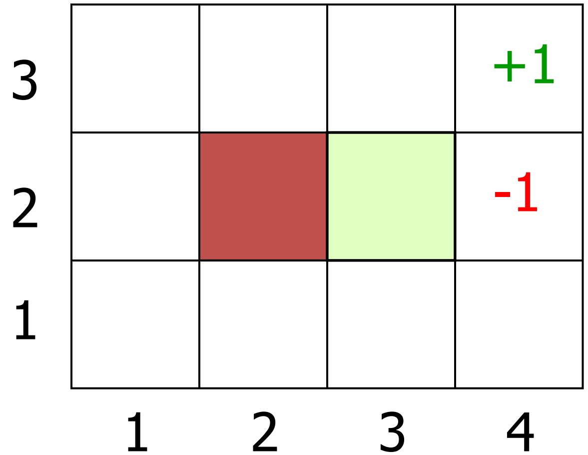


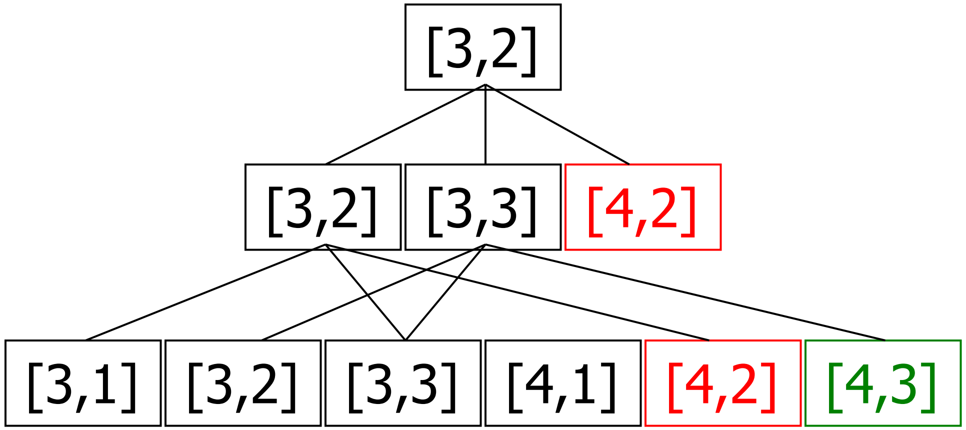
- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories:

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

- The optimal sequence is the one with maximal utility

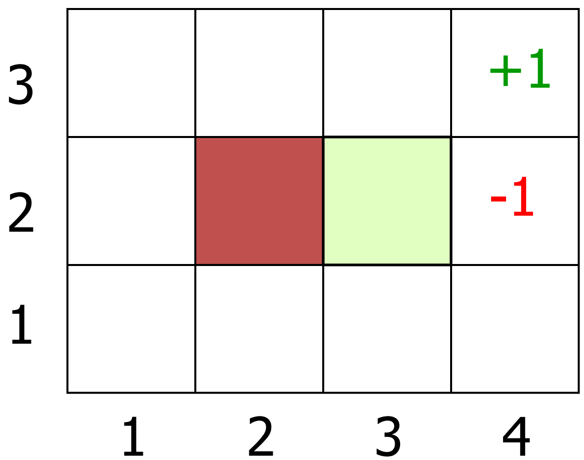
Optimal Action Sequence

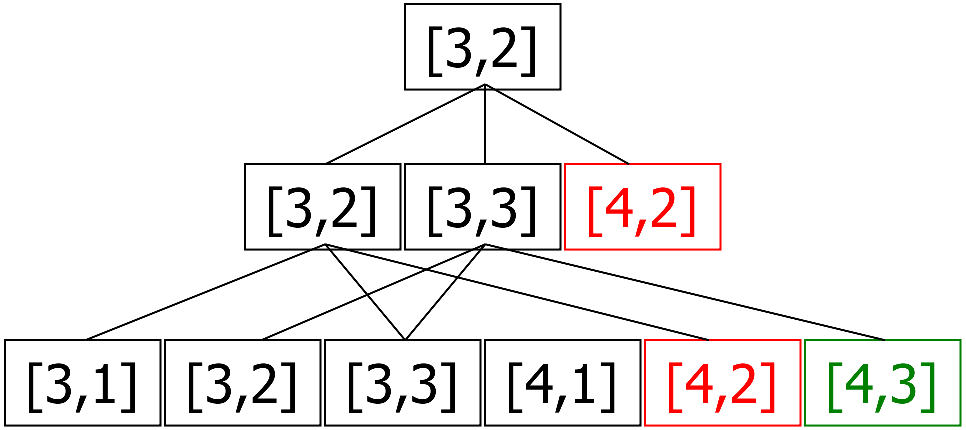




- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories
- The optimal sequence is the one with maximal utility
- **But is the optimal action sequence what we want to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