Reinforcement Learning, Part 1: Dynamic Programming and Markov Decision Processes

Data 102, Lecture 21

Spring 2023

Weekly Outline

- **Announcements:**
  - HW#5 due 04/07, Project proposal due 04/10
  - Midterm 2 on 04/13

- **So far: Bandits**
  - Multi-Arm Bandits: Decision Making under uncertainty
  - Environment (possible decisions and their outcomes) are fixed

- **Next 2 lectures: Repeated/online decision making with changing environment**
  - Today: Dynamic programming and Markov Decision Processes (stateful decision making)
  - Next time: Reinforcement learning (stateful decision making with uncertainty)

Online Decision Making

- In most of what we've looked at so far, we've made decisions "offline"
  - ✦ We get a bunch of data, and then use that data to make decisions

- In online decision-making, we get feedback after every decision and update our decision accordingly. Bandits is an example
- With multi-armed bandits, we get feedback BUT the "world" stays the same throughout: the arms and their reward distributions remain the same.
- With reinforcement learning, we get feedback AND the "world" changes with each decision: reward structure changes
  - ✦ We model the "world" using **states**: decisions take us from one state to another

Agenda

- Dynamic Programming
- Markov Decision Processes

Dynamic Programming

Dynamic programming is a method for organizing computations involving recursions to minimize repeated work.

Warm-up Example: Fibonacci sequence

Recursion: $F_n = F_{n-1} + F_{n-2}$ Base cases: $F_0 = 0, F_1 = 1$

```
def fib(n):
    print(f"Computing fib({n})")
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

fib(49) + fib(48)

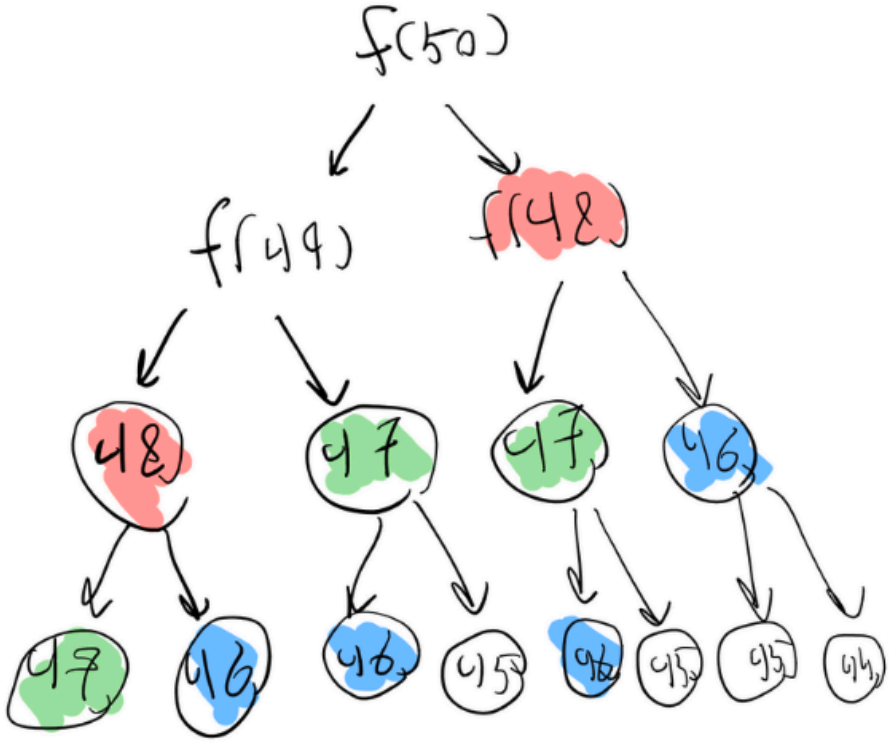
$F_2 = 0 + 1 = 1$

$F_3 = 1 + 1 = 2$

$F_4 = 2 + 1 = 3$

Question: What happens if we call `fib(50)`?

Exponential Blow-up



Computations further down the tree will be repeated even more times. So much repetitive work, slow run time.

Solution 1: Memoization

Keep track of already computed answers:

```
fib_numbers = {0: 0, 1: 1}
def fib_memo(n):
    if n in fib_numbers:
        return fib_numbers[n]
    else:
        answer = fib_memo(n-1) + fib_memo(n-2)
        fib_numbers[n] = answer
        return answer
```

- Can use decorators instead
- Much faster than naive recursion, but still slow due to dict look up each time

Solution 2: Dynamic Programming

We can replace the recursion with a for loop if computation is done in the right order:

```
import numpy as np

n = 50
fib_nums_dp = np.zeros(n_max)
fib_nums_dp[0] = 0, fib_nums_dp[1] = 1
for i in range(2, n+1):
    fib_nums_dp[i] = fib_nums_dp[i-1] + fib_nums_dp[i-2]
```

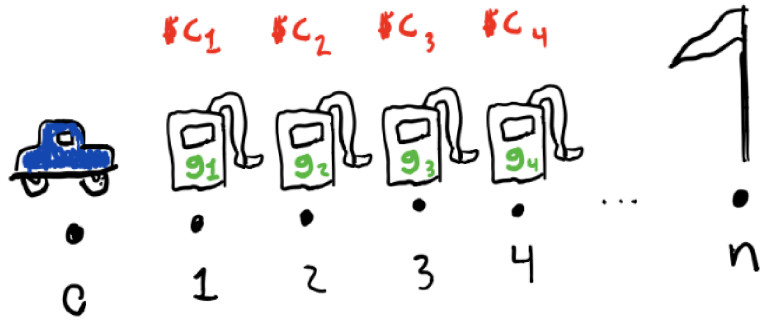
fib(50)

n+1

fibold1 = 0
fibold2 = 1
for i in range(2, n+1):
    fibnew = fibold1 + fibold2
    fibold1 = fibold2
    fibold2 = fibnew
result will be fibnew

- Dynamic programming builds up to the solution and avoids recursion by taking advantage of the **total ordering.**
- Total ordering might not always be easy to notice.
- **Pro:** fast, low-memory **Con:** more thinking, need to find the total ordering

Harder Example: Car and Gas Stations



- Locations $0,\ldots,n$
- Car starts at location 0, wants to get to location n
- Each location i has a gas station, can purchase g_i units of gas at unit cost c_i

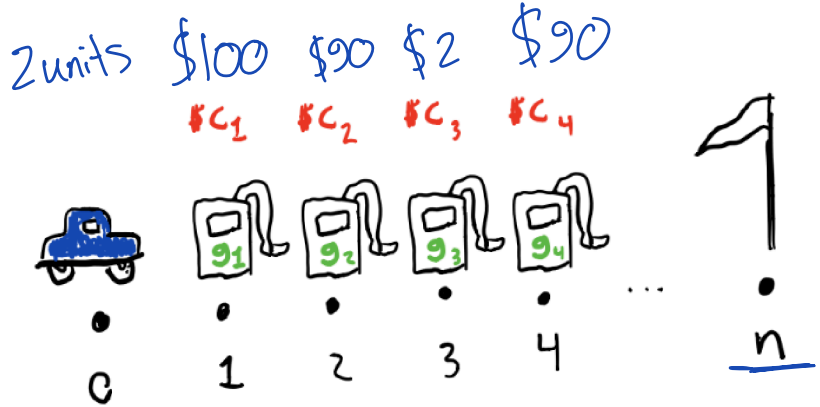
Each unit of gas moves the car 1 unit to the right

Problem:

How much gas should we buy at each location to minimize total cost for getting to location n?

Solution 1: Recursion

- Define a state as (gas left in tank, location)
- Define f(gas, loc) as minimum cost to get to end given current state
- **Two possible decisions in each state:**
  - ✦ Buy one unit of gas and stay where we are
  - ✦ Go forward



Solution 1: Recursion

- Define a state as (gas left in tank, location)
- Define f(gas, loc) as minimum cost to get to end given current state
- **Two possible decisions in each state:**
  - ✦ Buy one unit of gas and stay where we are
  - ✦ Go forward

n: final location

gas_capacity, gas_prices are given

```
def f(gas, loc):
    if gas < 0 or gas > gas_capacity:
        return np.inf
    if loc == n:
        return 0
    cost_to_fill = f(gas+1, loc) + gas_prices[loc]
    cost_to_drive = f(gas-1, loc+1)
    return min(cost_to_fill, cost_to_drive)
```

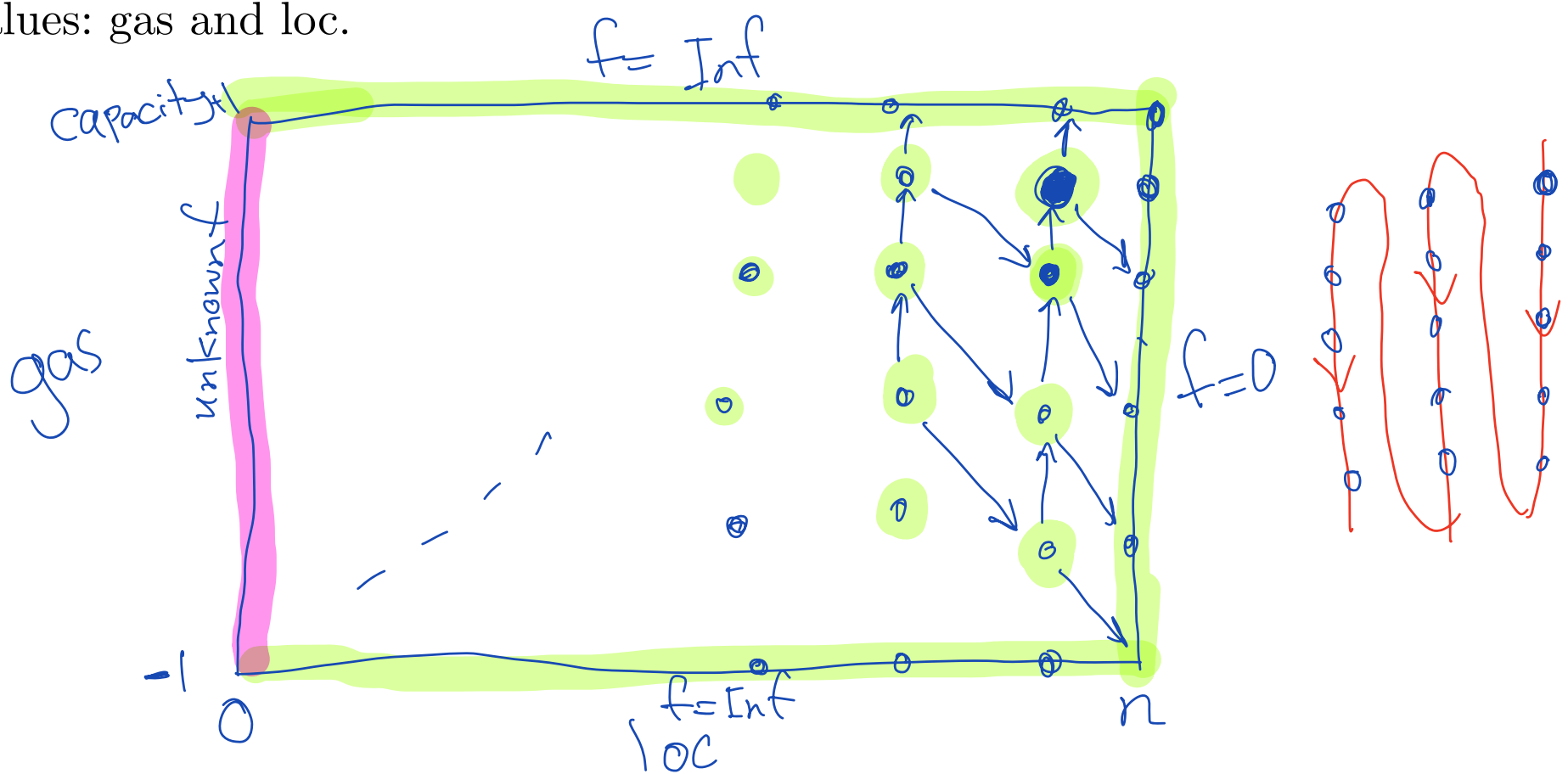
base cases 1

basecase2

Recursive calls

Solution 2: Dynamic Programming

Question: What is the total ordering in this example? It involves both state values: gas and loc.



Solution 2: Dynamic Programming

```
f = np.zeros([n+1, n+1])
for loc in range(n-1, -1, -1):
    for gas in range(gas_capacity, -1, -1):
        if gas == gas_capacity:
            cost1 = np.inf
        else:
            cost1 = f[gas+1, loc] + gas_prices[loc]
        if gas == 0:
            cost2 = np.inf
        else:
            cost2 = f[gas-1, loc+1]
        f[gas, loc] = min(cost_to_fill, cost_to_drive)
```

total ordering

- Main trick is to figure out the **total ordering** so we can build up to the solution at `f(initial_gas, 0)` without recursion

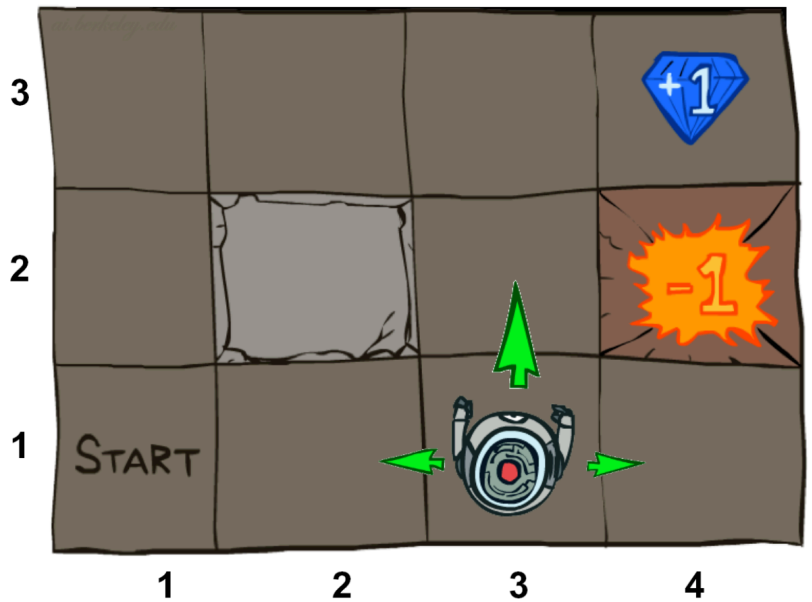
Agenda

- Dynamic Programming
- Markov Decision Processes

Markov Decision Processes

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function $T(s, a, s')$
    - Probability that a from s leads to s', i.e., $P(s' | s, a)$
    - Also called the model or the dynamics
  - A reward function $R(s, a, s')$
    - Sometimes just R(s) or R(s')
  - A start state
  - Maybe a terminal state

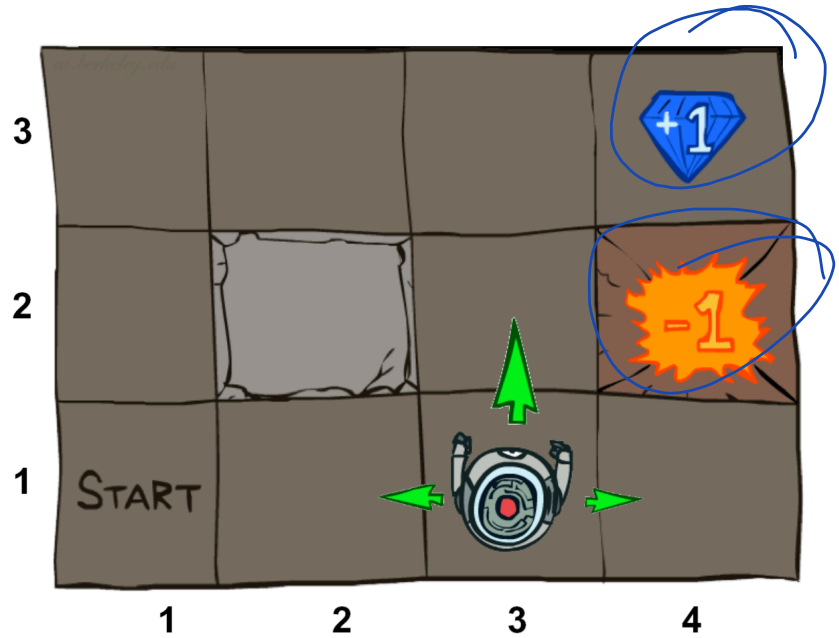
old state
action
new state



Next few slides credit: Fernando Pérez, Anca Dragan, Dan Klein, Pieter Abbeel, and the Berkeley CS188 team: ai.berkeley.edu

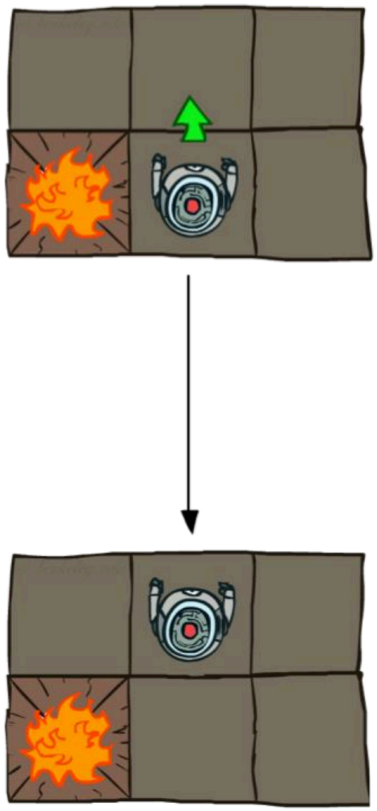
Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of rewards

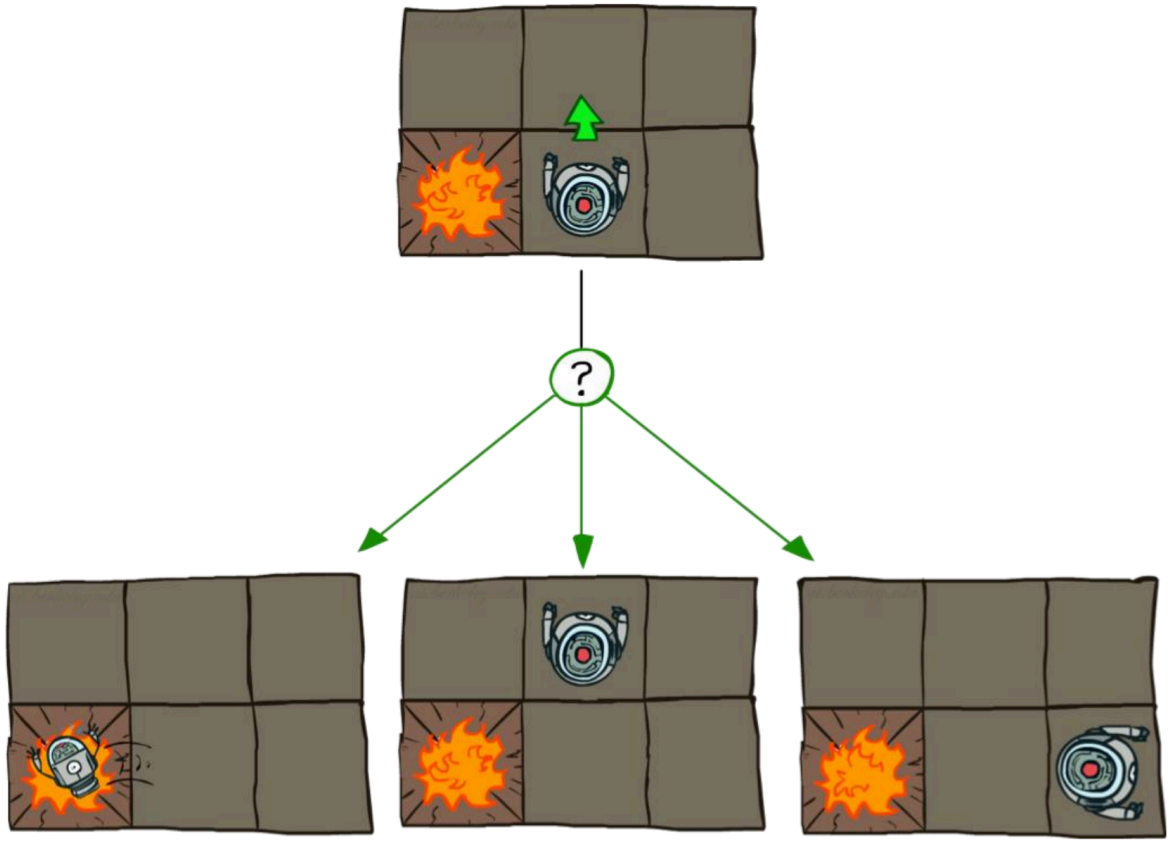


Grid World Actions

Deterministic Grid World



Stochastic Grid World



What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means action outcomes depend only on the current state:

History

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \ldots, S_0 = s_0) =$$
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

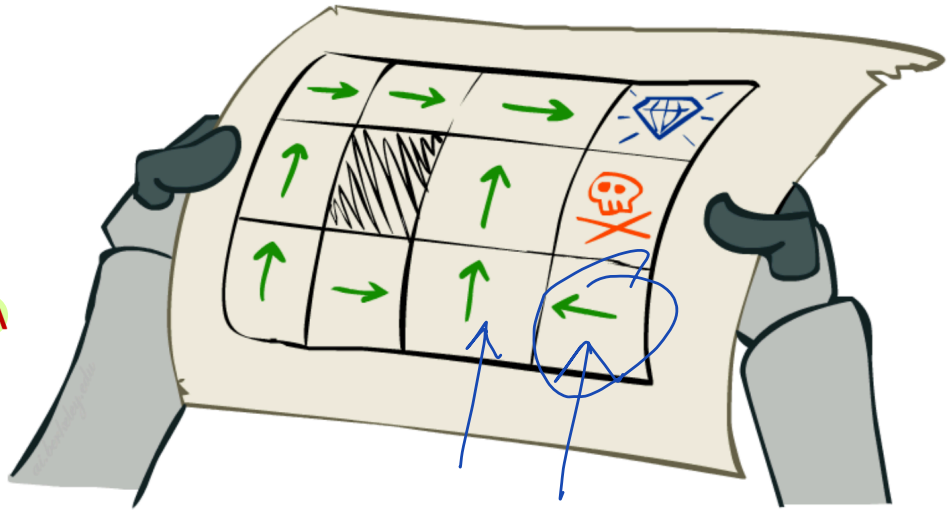
- This is just like search, where the successor function could only depend on the current state (not the history)
- Given a determined action per each state, MDP becomes a Markov chain

Andrey Markov
(1856-1922)

Policies

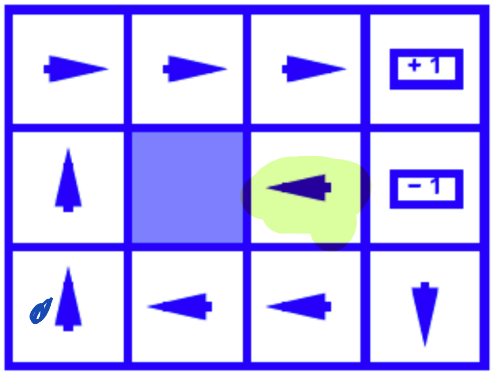
- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal

- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed

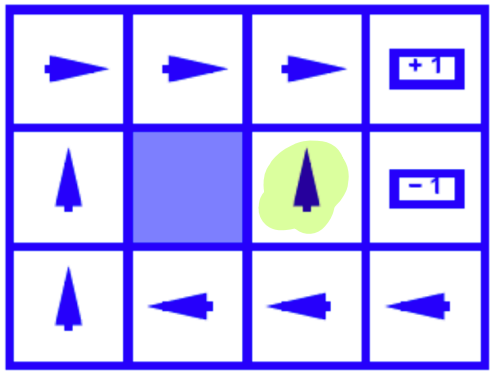


Optimal Policies

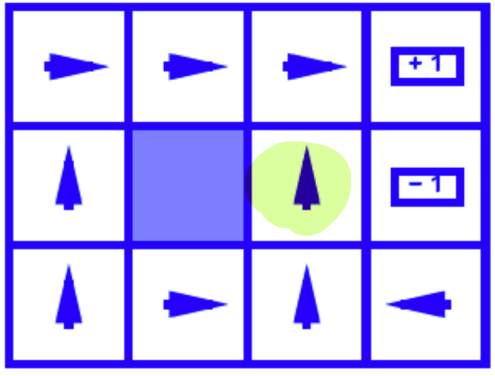
Patient



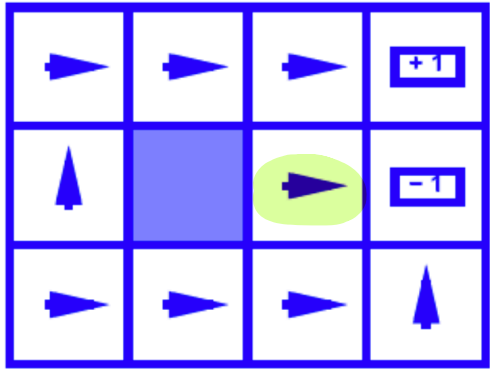
Cost at each time step ⇒ R(s) = -0.01



R(s) = -0.03



R(s) = -0.4



R(s) = -2.0

Impatient

Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later (impatience)
- **We assume values of rewards decay exponentially for one round to the next with a factor of γ.** This is common when modeling decision-making agents in economics.

1

Worth Now

γ

Worth Next Step

γ^2

Worth In Two Steps

Optimal Quantities

- The value (utility) of a state s:

  $V^*(s)$ = expected utility starting in s and acting optimally

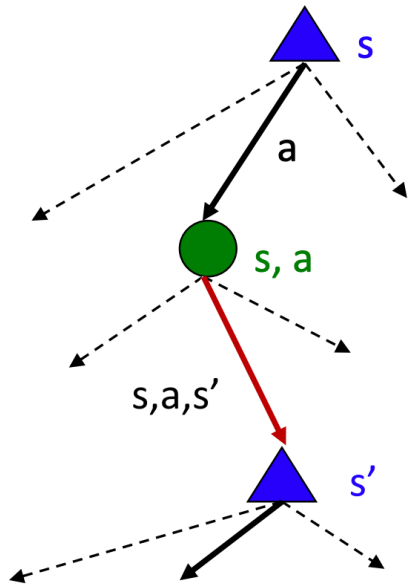
- The value (utility) of a q-state (s,a):

  $Q^*(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) acting optimally

- The optimal policy:

  $\pi^*(s)$ = optimal action from state s

$\pi^*: S \rightarrow A$



s is a *state*

(s, a) is a *q-state*

(s,a,s') is a *transition*

Principle of Optimality

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." (Bellman, 1957)

- Used to convert an optimal decision making problems to recursive problems or dynamic programming
- It implies that the following quantities are the same (assuming an MDP when outcome of each action is deterministic):

$$V^*(s_0) = \max_{\{a_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1})$$

Our goal in its original form

utility from where I end up

$$V^*(s_0) = \max_a R(s_0, a, s') + \gamma V(s')$$

Principle of Optimality

optimize action now

Values of States

Recursive definition of value:

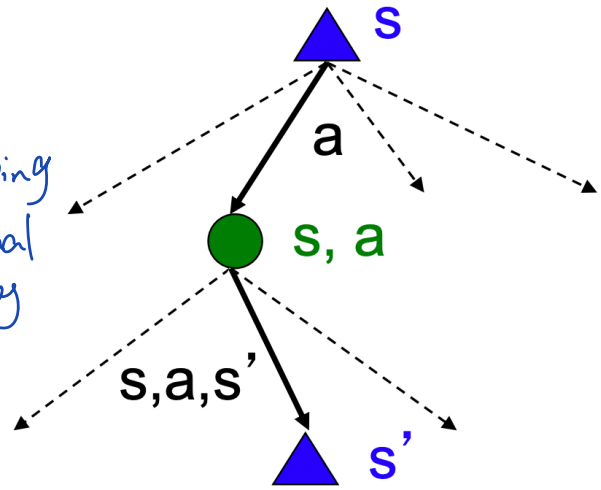
$$V^*(s) = \max_a Q^*(s,a)$$

principle of optimality

$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

immediate reward; following optimal policy

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$



The last equation is called the **"Bellman equation"**

Solving the Bellman Equation

circular Recursion

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right]$$

- We can use dynamic programming to solve the Bellman equation!
- But the recursion for V^* is circular: $V^*(s)$ could depend on itself.
- This was not a problem for gas stations because there was a total ordering.
- **Solution?**

Solving the Bellman Equation

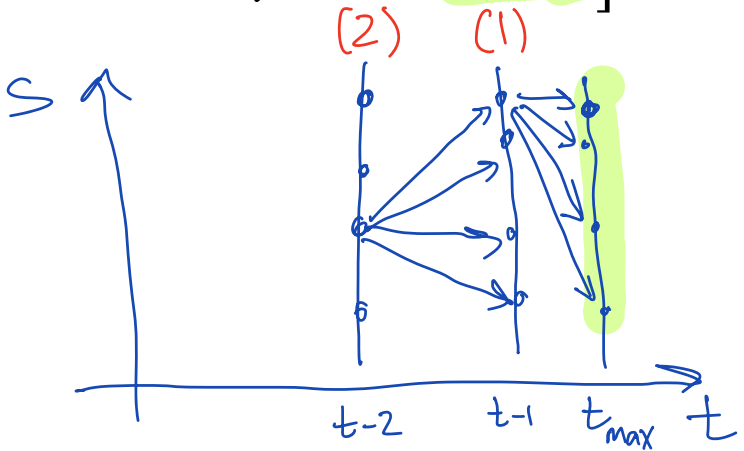
$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s') \right]$$

- We can use dynamic programming to solve the Bellman equation!
- But the recursion for V^* is circular: $V^*(s)$ could depend on itself.
- This was not a problem for gas stations because there was a total ordering.
- **Solution: Add a time component**

$$V^*(s, t) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s', t+1) \right]$$

$$V^*(s, t_{max}) = 0$$

- Time t creates total ordering!
- Can recover $V^*(s)$ by letting $t_{max} \to \infty$



Value Learning via Naive Recursion

```
def V(s,t):
  if t >= t_max:
    return 0
  else:
    best = max([sum([T(s,a,s')*(R(s,a,s')+gamma*V(s',t+1))
                     for s' in states
                    ])
                for a in actions
               ])
    return best
```

transition P; immediate value; discounted future value

base case

Problem with this solution? Redundant calculations similar to Fib

Value Learning via Dynamic Programming

```
V = np.zeros(shape=(num_states, t_max+1))
for t in range(t_max-1, -1, -1):
  for s in states:
    V[s,t] = max([sum([T(s,a,s')*(R(s,a,s')+gamma*V[s',t+1])
                       for s' in states
                      ])
                  for a in actions
                 ])
```

look up into array

t=2 ... V[S, 2+1]

I only need keep track of a 1D array V_old

Much faster, by taking advantage of time total ordering.
This is called the **"Value Iteration"** algorithm.

We want $V^*(S, t=0)$ as final answer

Value Learning via Dynamic Programming

Can save memory with the "sliding window" trick:

```
V = np.zeros(num_states)
for t in range(t_max-1, -1, -1):
  V_old = np.copy(V)
  for s in states:
    V[s] = max([sum([T(s,a,s')*(R(s,a,s')+gamma*V_old[s'])
                     for s' in states
                    ])
                for a in actions
               ])
```

Can improve even further by "in-place" updating until convergence.

Q-Iteration

- We are interested in finding the optimal policy. The value function does not give any information on the optimal policy.
- Instead, we need the Q^* function to find the optimal policy.
- **Question:** How do we use Q^* function to get π^*?

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}}\ Q^*(s,a)$$

Q-Iteration

- We are interested in finding the optimal policy. The value function does not give any information on the optimal policy.
- Instead, we need the Q^* function to find the optimal policy.
- **Question:** How do we use Q^* function to get π^*?
- Recall Q^* and V^* are closely related: $V^*(s) = \max_{a \in A} Q^*(s, a)$
- Compute Q^* similar to V^* by using time component and dynamic programming:

$$
\begin{aligned}
Q^*(s, a, t) &= \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s', t+1) \right] \\
&= \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a' \in A} Q^*(s', a', t+1) \right] \\
Q^*(s, a, t_{max}) &= 0
\end{aligned}
$$