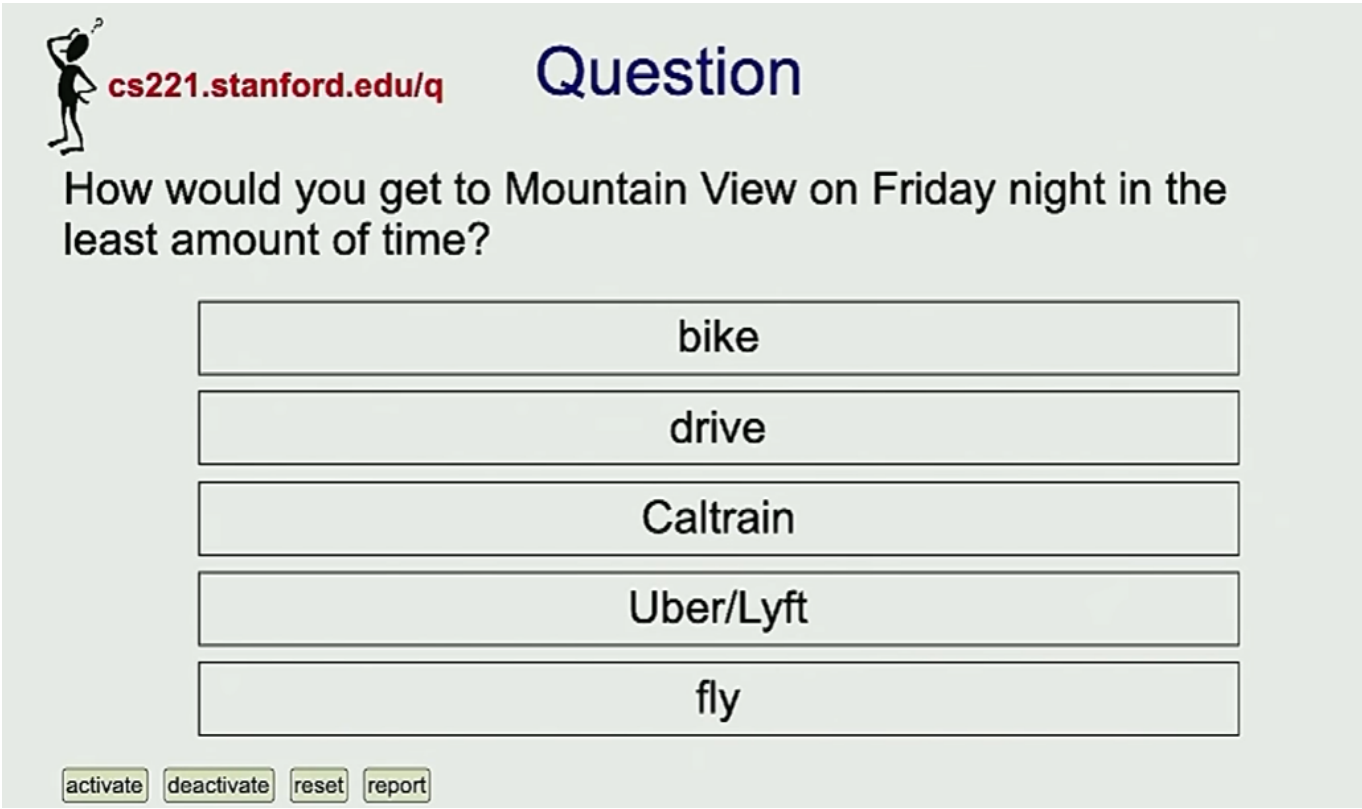
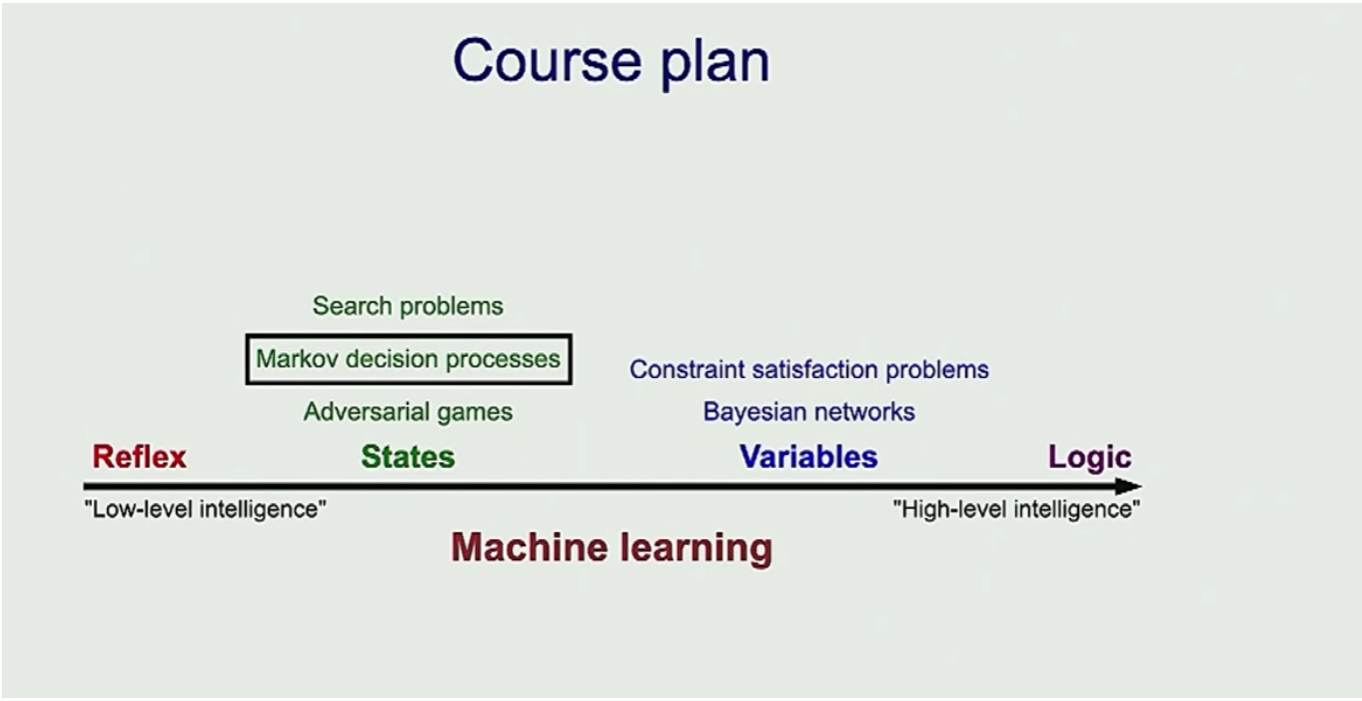
6. Markov Decision Processes 1 - Value Iteration

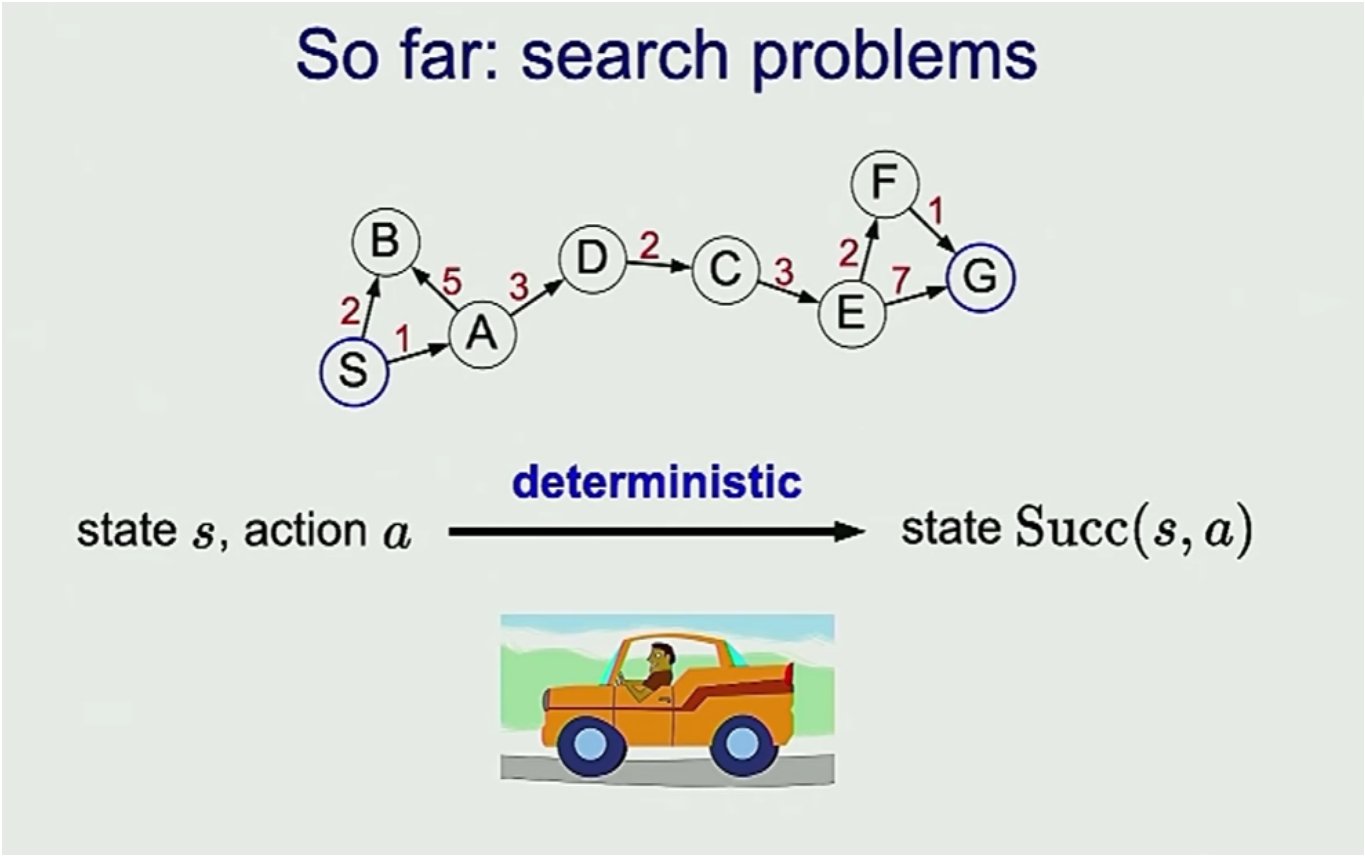
Intro



There are lot of uncertainties

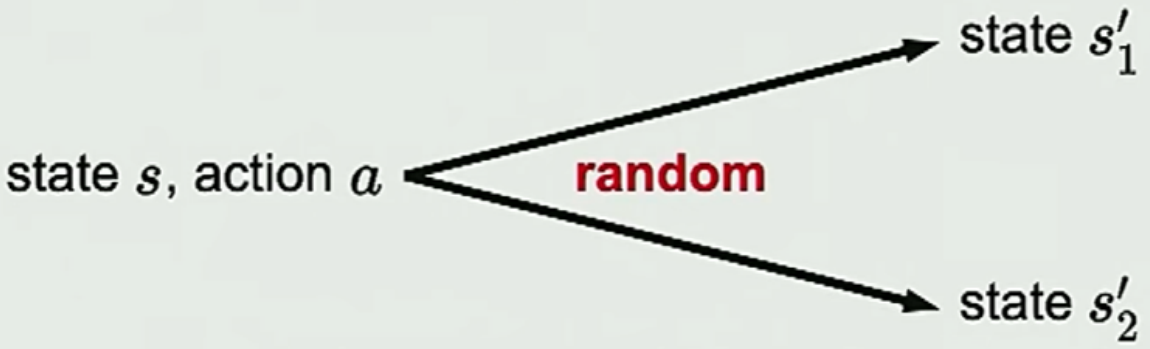


We talked about search problems, where everything was deterministic The solution was a sequence of actions



And now we are talking about this next class of state-based functions, Markov Decision Processes The idea is that, we take actions, but we may not end up where we expected to, because there's a nature around you and there is going to be uncertainty and stuff that u did not expect

Uncertainty in the real world

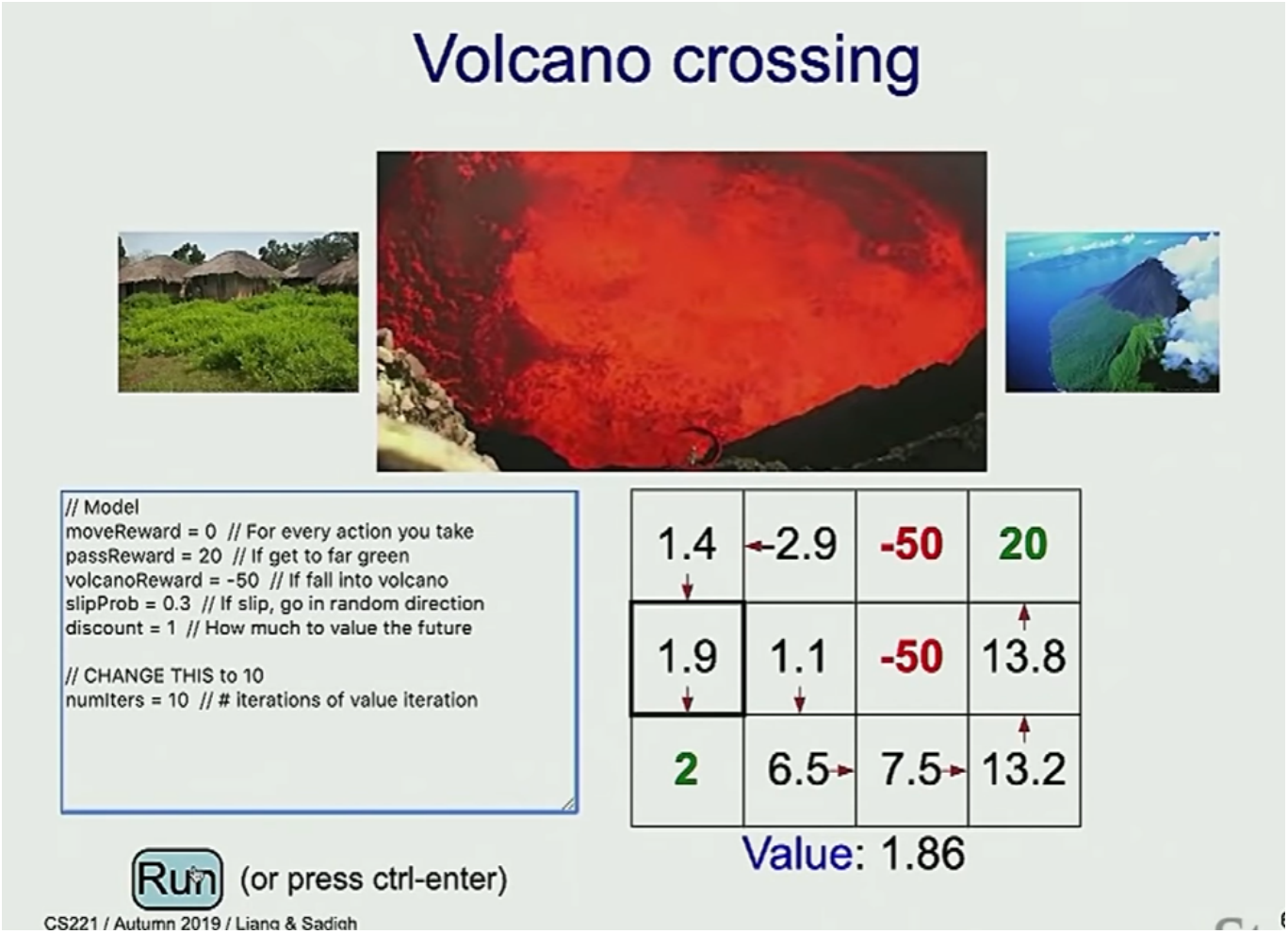


Applications

Robotics: decide where to move, but actuators can fail, hit unseen obstacles, etc.

Resource allocation: decide what to produce, don't know the customer demand for various products

Agriculture: decide what to plant, but don't know weather and thus crop yield



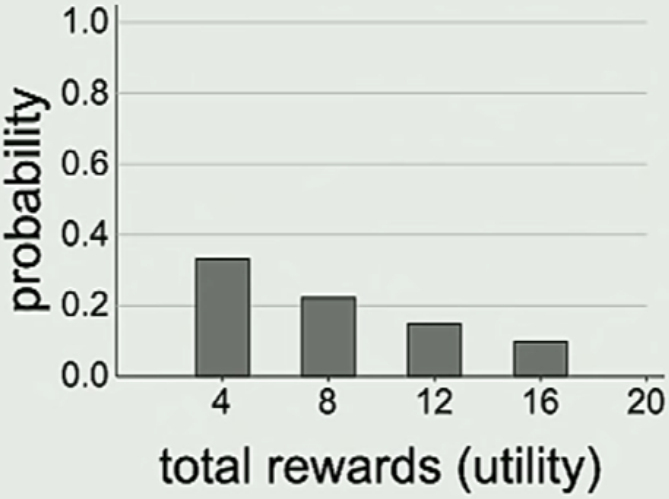
The dice game


Dice game
Example: dice game
For each round $r = 1, 2, \ldots$
You choose stay or quit.
If quit, you get $10 and we end the game.
If stay, you get $4 and then I roll a 6-sided dice.
If the dice results in 1 or 2, we end the game.
Otherwise, continue to the next round.
Start
Stay
Quit
Dice: 2
Rewards: 8

Rewards

Rewards

If follow policy "stay":

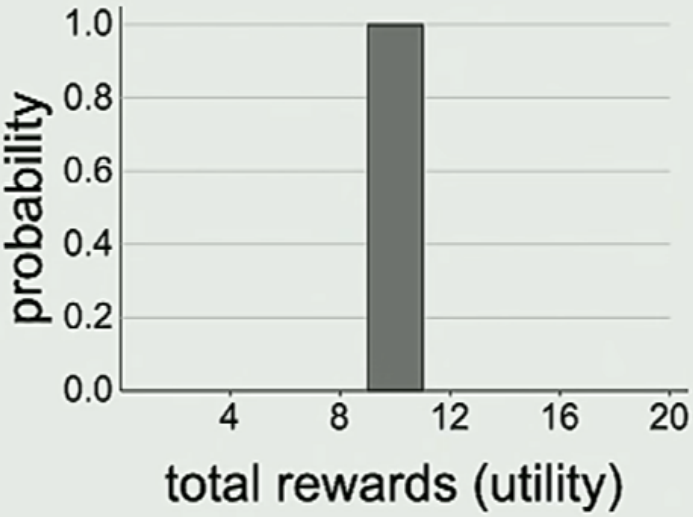


Expected utility:

$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \cdots = 12$$

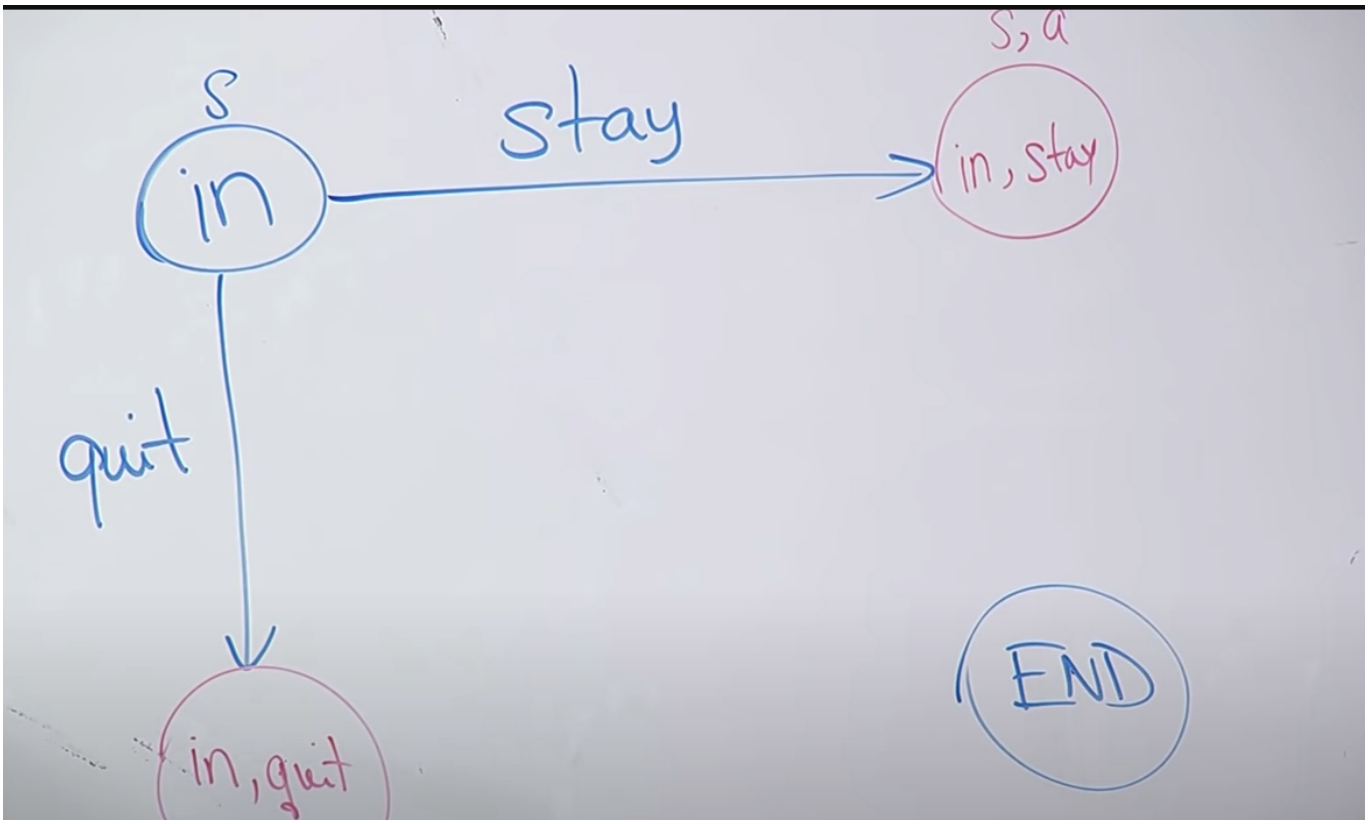
Rewards

If follow policy "quit":



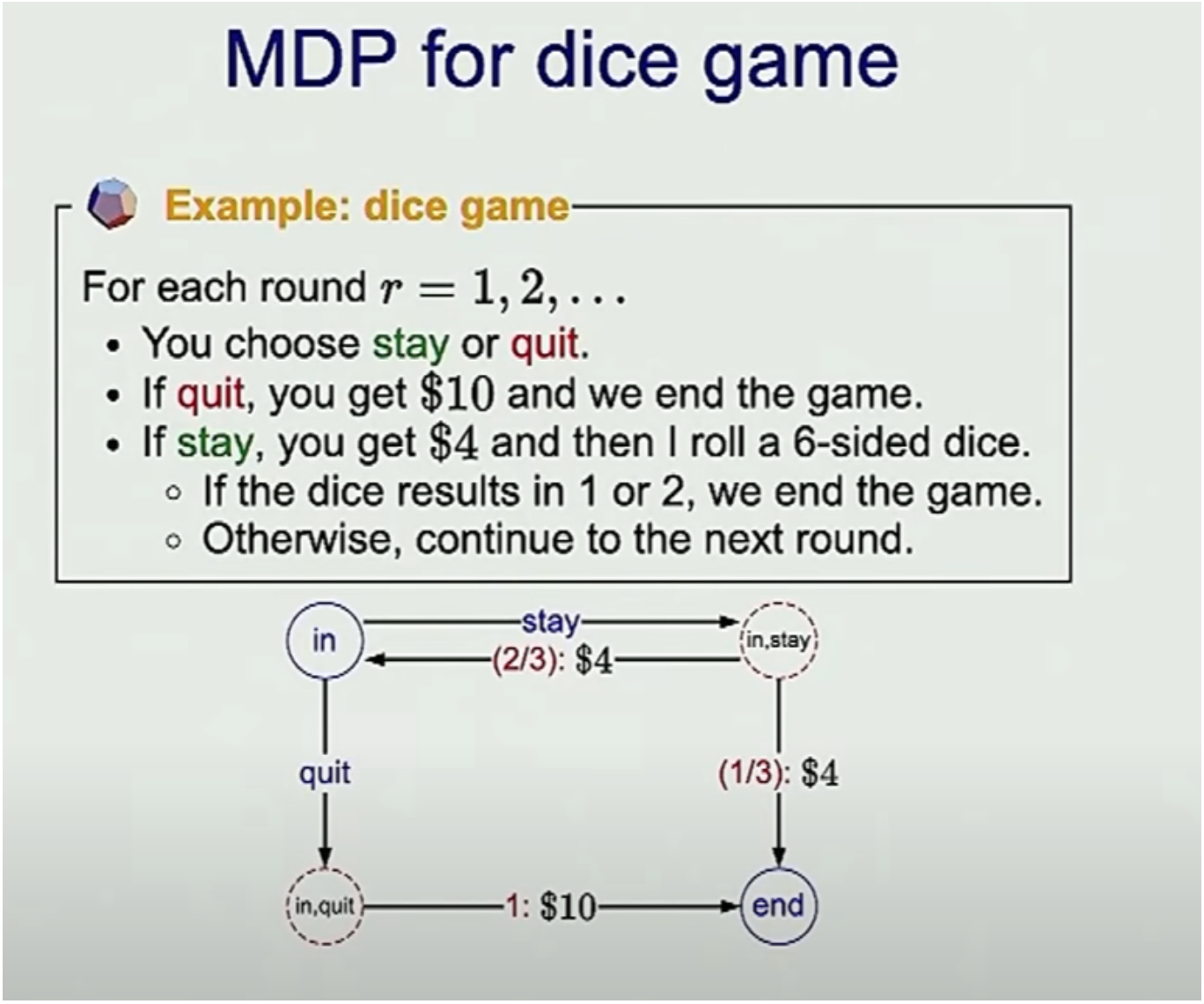
Expected utility:

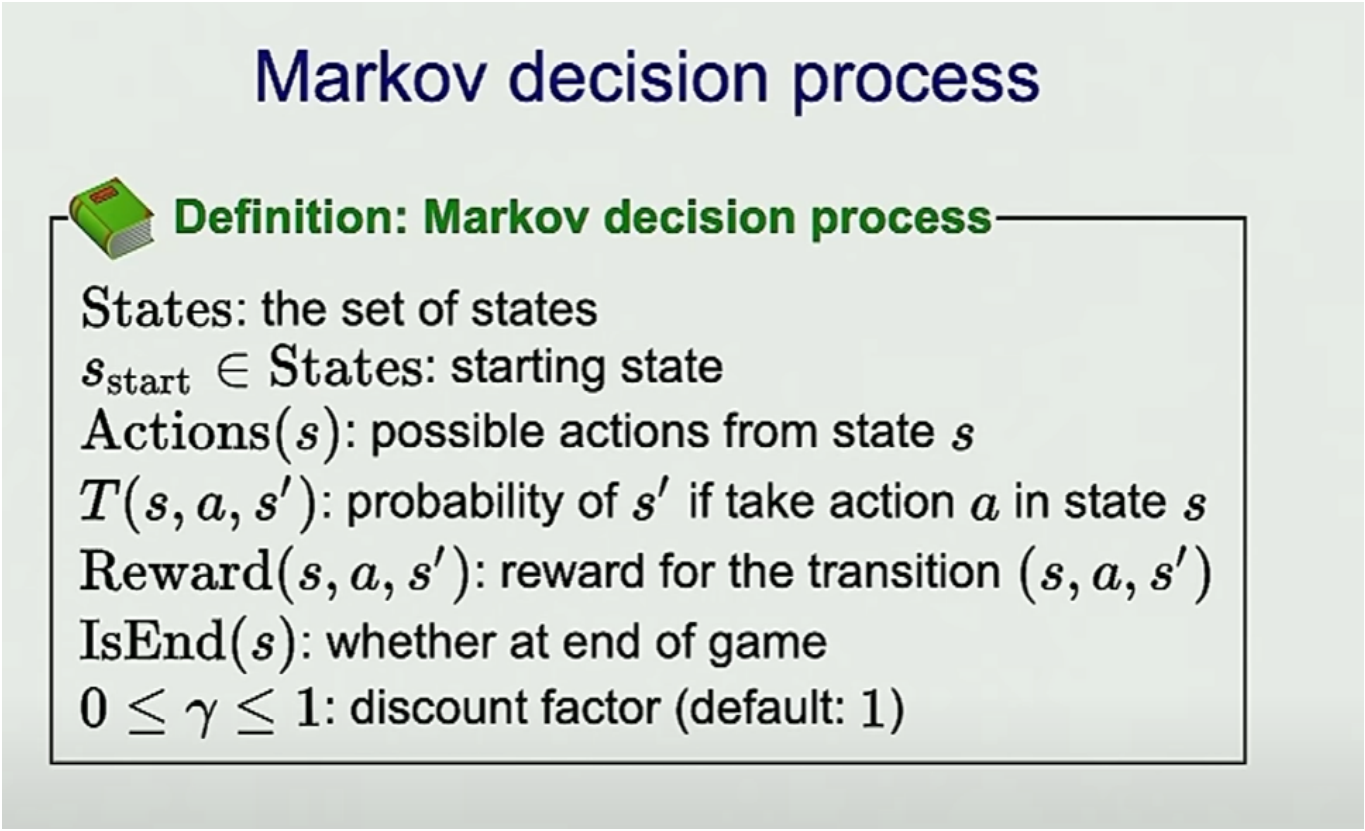
$$1(10) = 10$$



State: You can either in or out(end of the gane) Blue: States Red: "Chance Node", state with actions, flagging

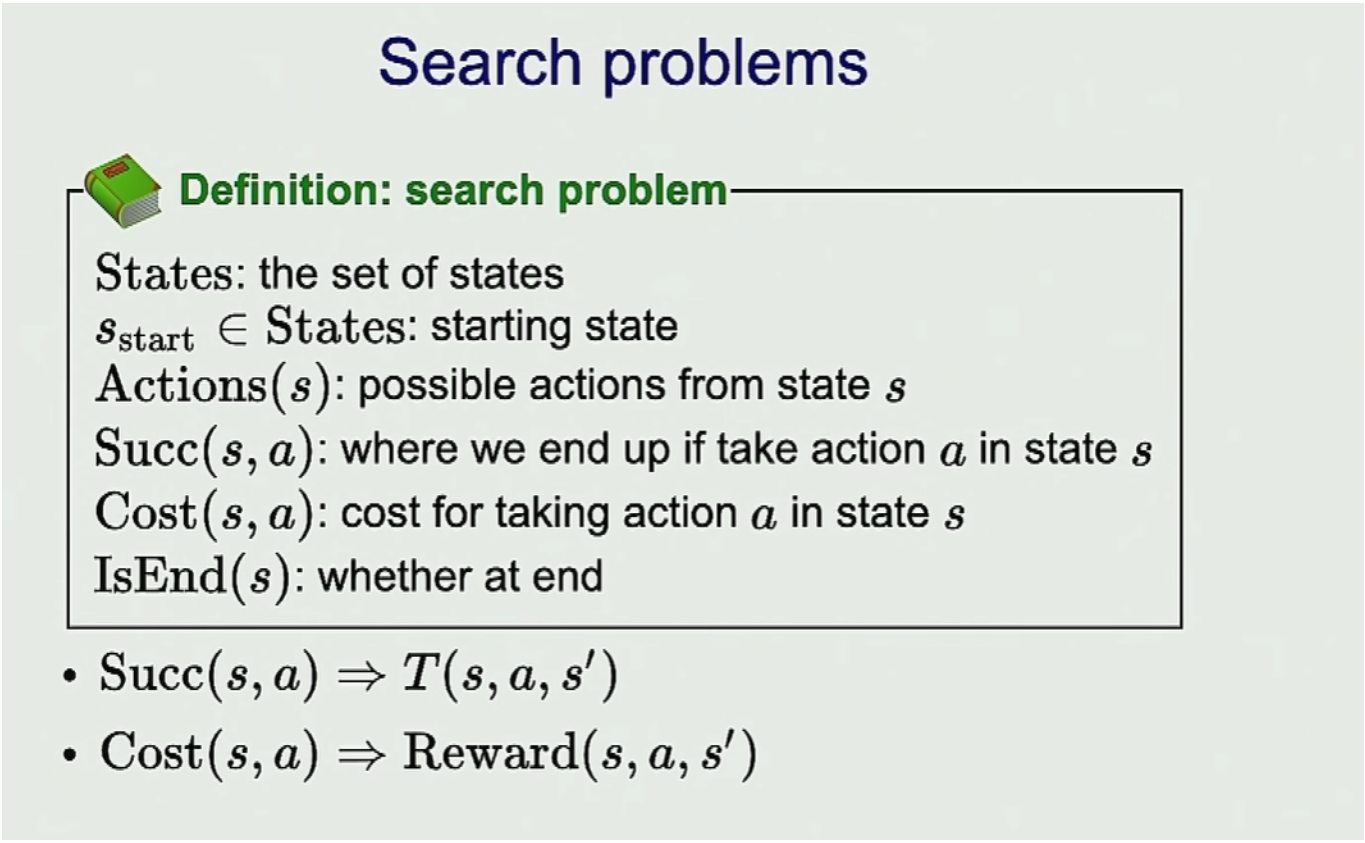
the state that ur in when taking that action From the "Chance Node", we are going to take probobilities



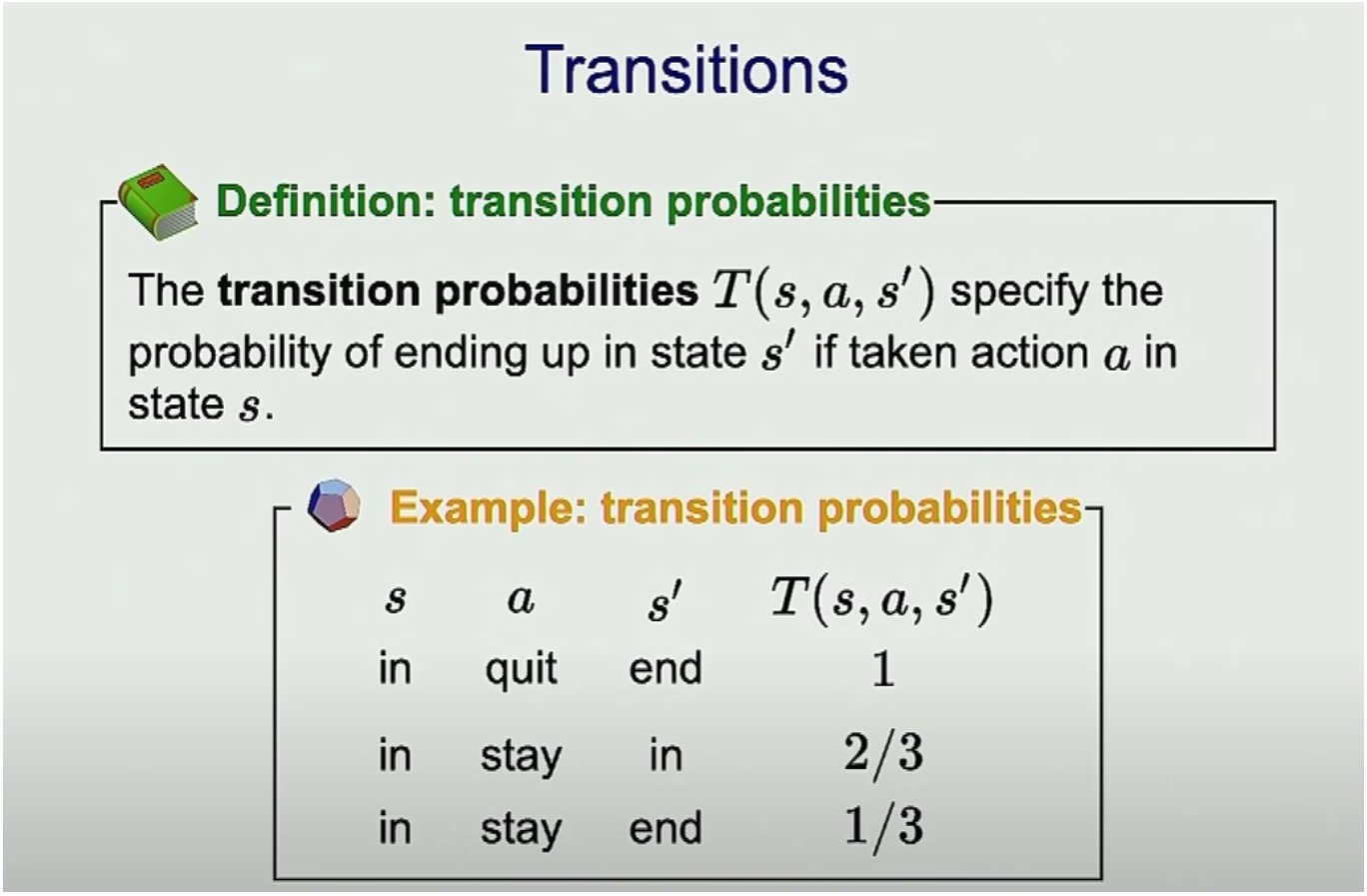


Discount factor: How much you value the future

Different from search

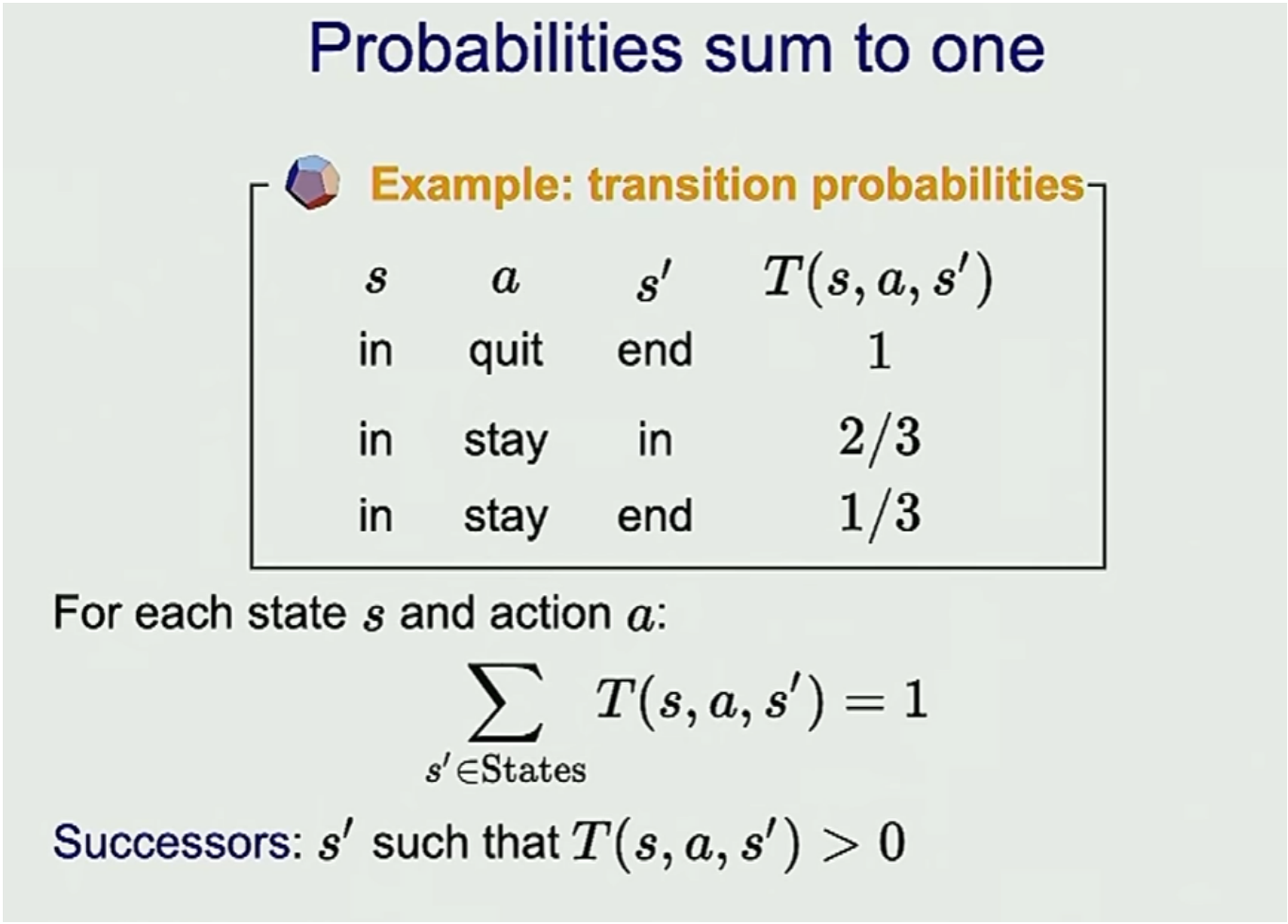


About the goal: In search problems, we want to minimize the cost In Markov Decision Processes, we wanna maximize the reward, similar to real world view 😃

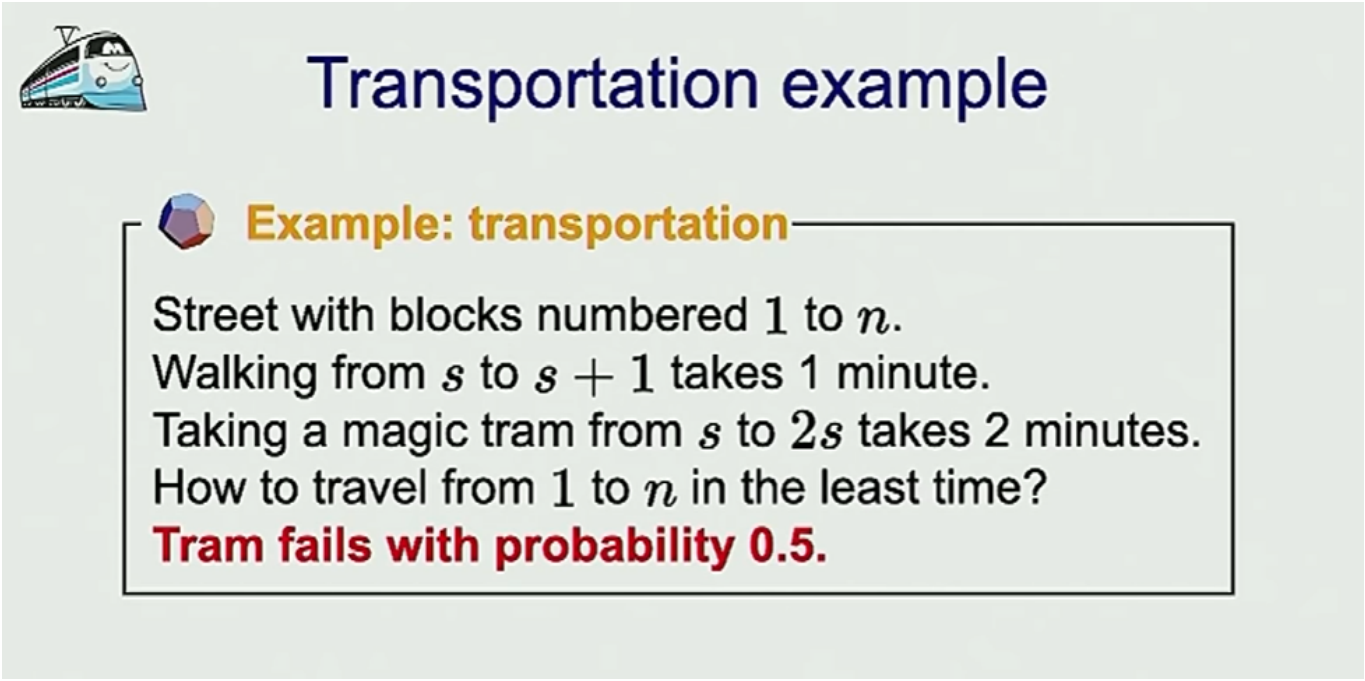


What sum up to 1?

The different S' you are going to end up at, when you choose the same action



Tram Problem Example



If the tram fails, we just gonna wast 2 minutes reward for walk: -1 reward for tram: -2

tram.py (~/Desktop/CS221-Spring2018/spring2018/semilive/mdp1) - VIM

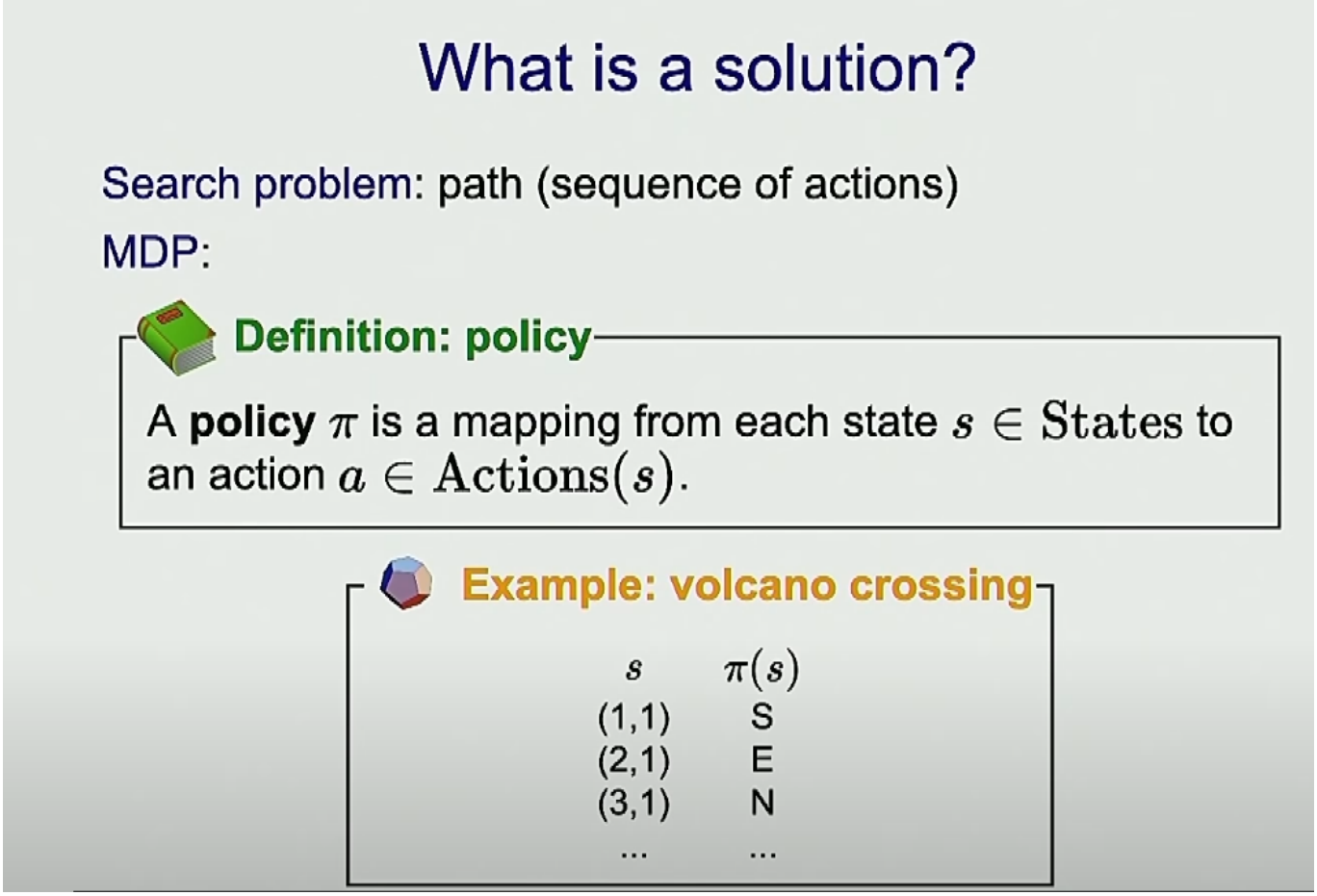
```
12      def isEnd(self, state):
13          return state == self.N
14      def actions(self, state):
15          # return list of valid actions
16          result = []
17          if state+1<=self.N:
18              result.append('walk')
19          if state*2<=self.N:
20              result.append('tram')
21          return result
22      def succProbReward(self, state, action):
23          # return list of (newState, prob, reward) triples
24          # state = s, action = a, newState = s'
25          # prob = T(s, a, s'), reward = Reward(s, a, s')
26          result = []
27          if action=='walk':
28              result.append((state+1, 1., -1.))
29          elif action=='tram':
30              result.append((state*2, 0.5, -2.))
31              result.append((state, 0.5, -2.))
32          return result
33      def discount(self):
34          return 1.
35      def states(self):
36          return range(1, self.N+1)
37
38  mdp = TransportationMDP(N=10)
39  print(mdp.actions(3))
```

Policy

The policy is the soltion to MDP problem(but not guerenteed to be the most optimistic solution).

So with a give state, there are many actions we can take, which means many policies, but they are not guaranteed to be optimistic, we have to evaluate those policies and find the best policy via policy evaluation
In summary : Policies are like recipes, and we need to taste-test them (evaluate) to find the most satisfying one!

Policies are action to take!

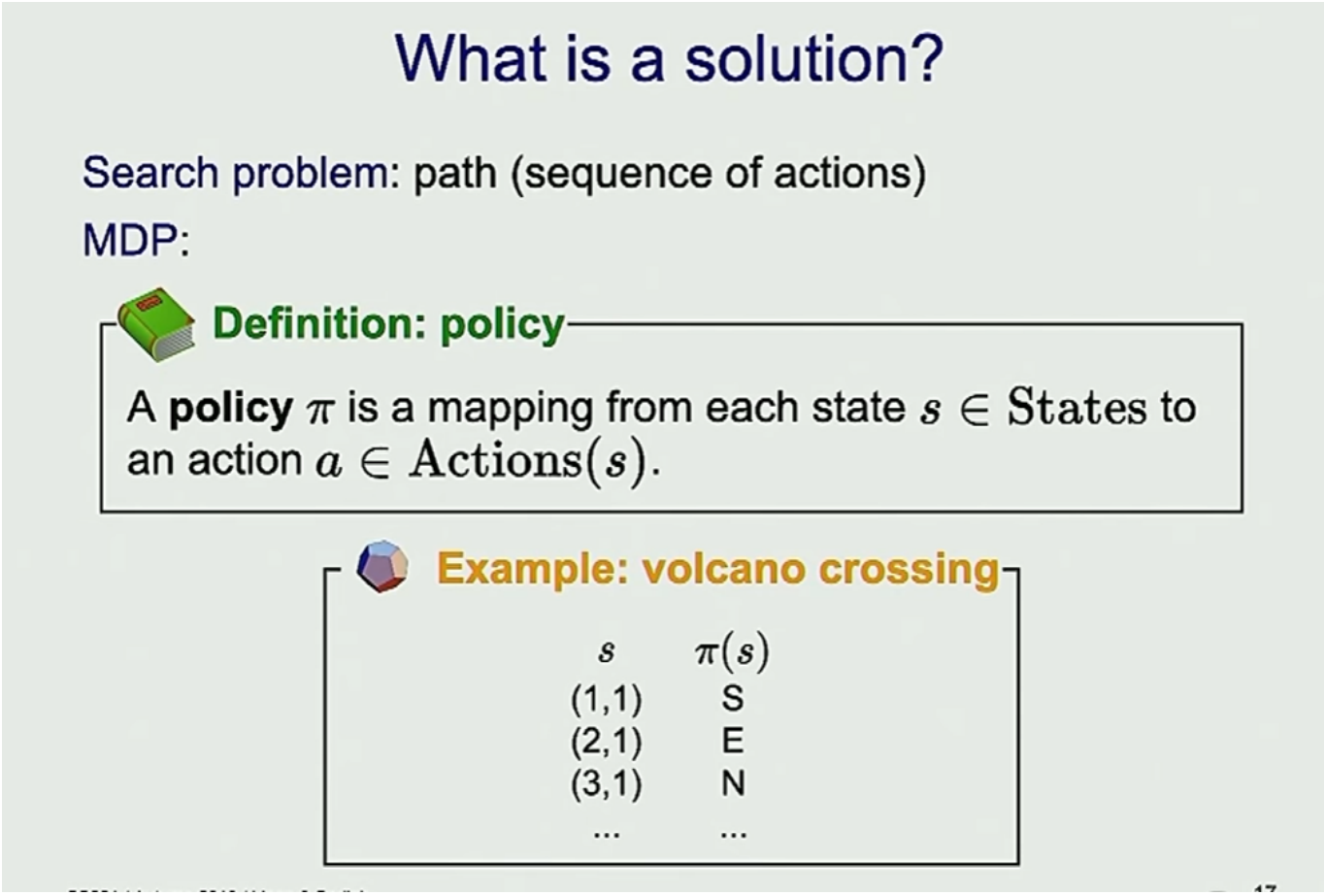


Find a solution

The solution for search problem was a sqeuence of paths, but that's when everything is deterministic But in MDP the way we are defining the solution is using the notion of policy

The policy is a path/action to take, but it is not gueranteed to be optimistic, we have to evaluate it

The policy is a function that takes any state and returns an action we can take? Pi(s) => Action

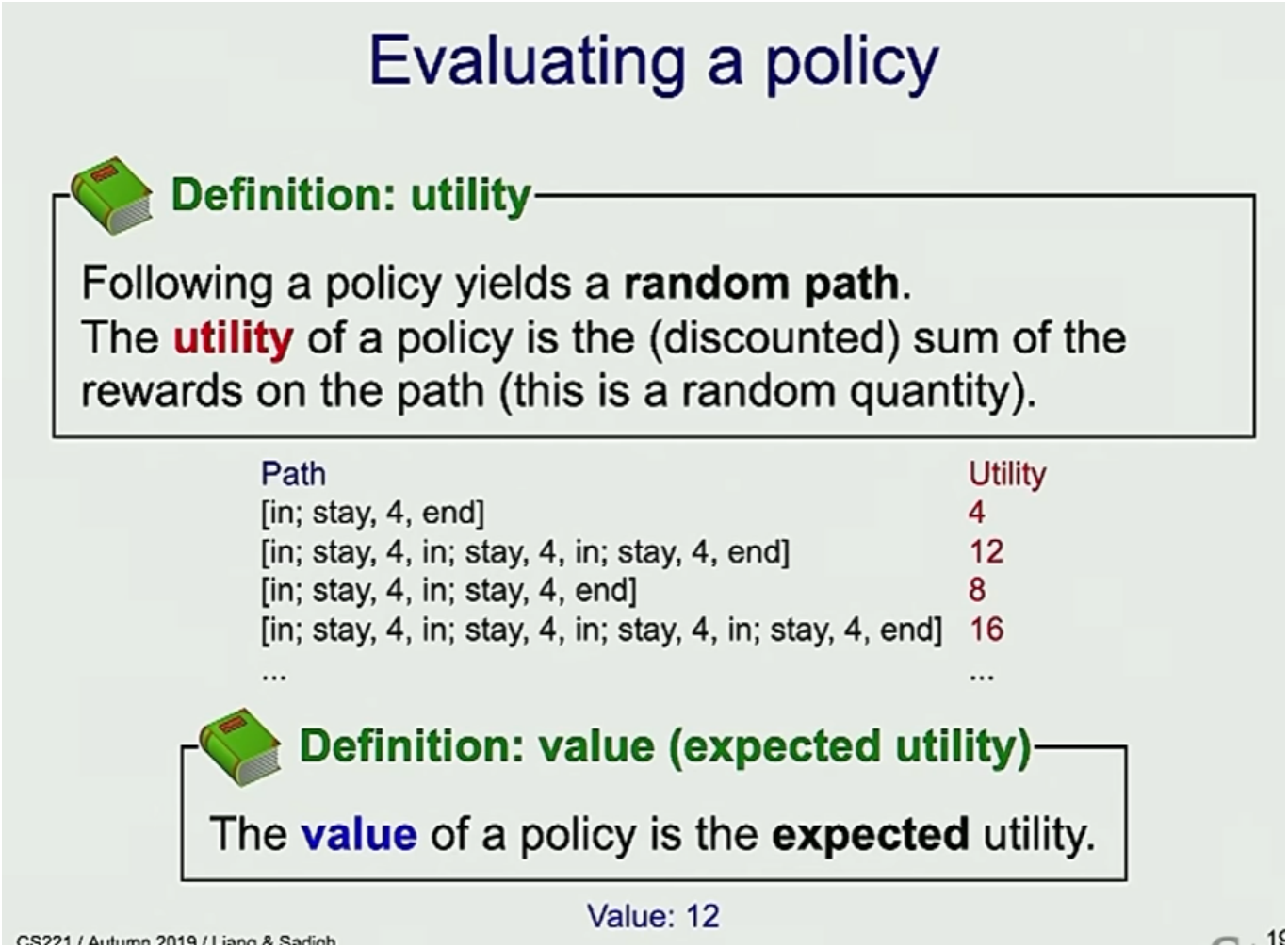


Policy Evaluation

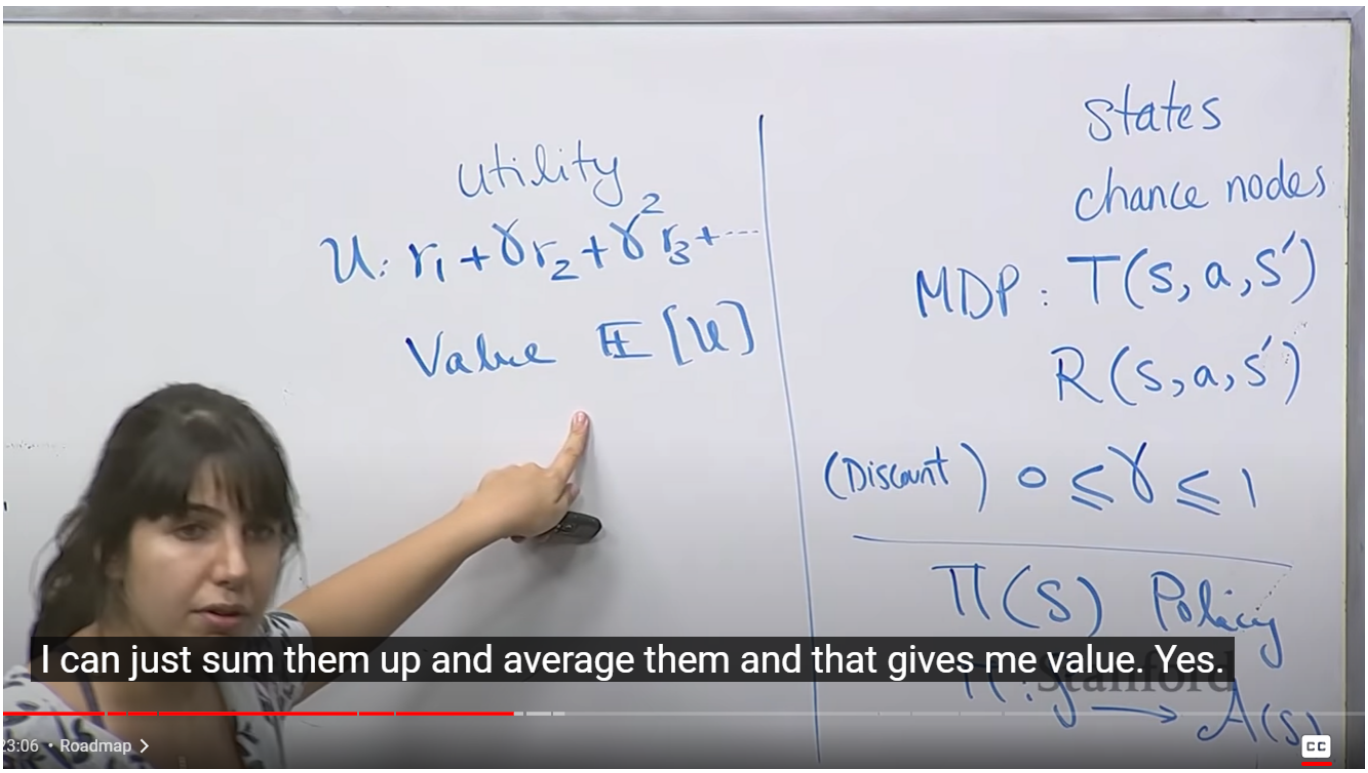
So ideally I would like to find the best policy to give me the best/right solution But first, we need to tell how good a policy is, and that is the `Policy Evaluation`

A policy from state S is denoted by pi(S)

Utility



The utility is just random variables and we cannot optimize on that Utility is a sum of rewards, we are summing them up with discount so Utility a **discounted sum of rewards **



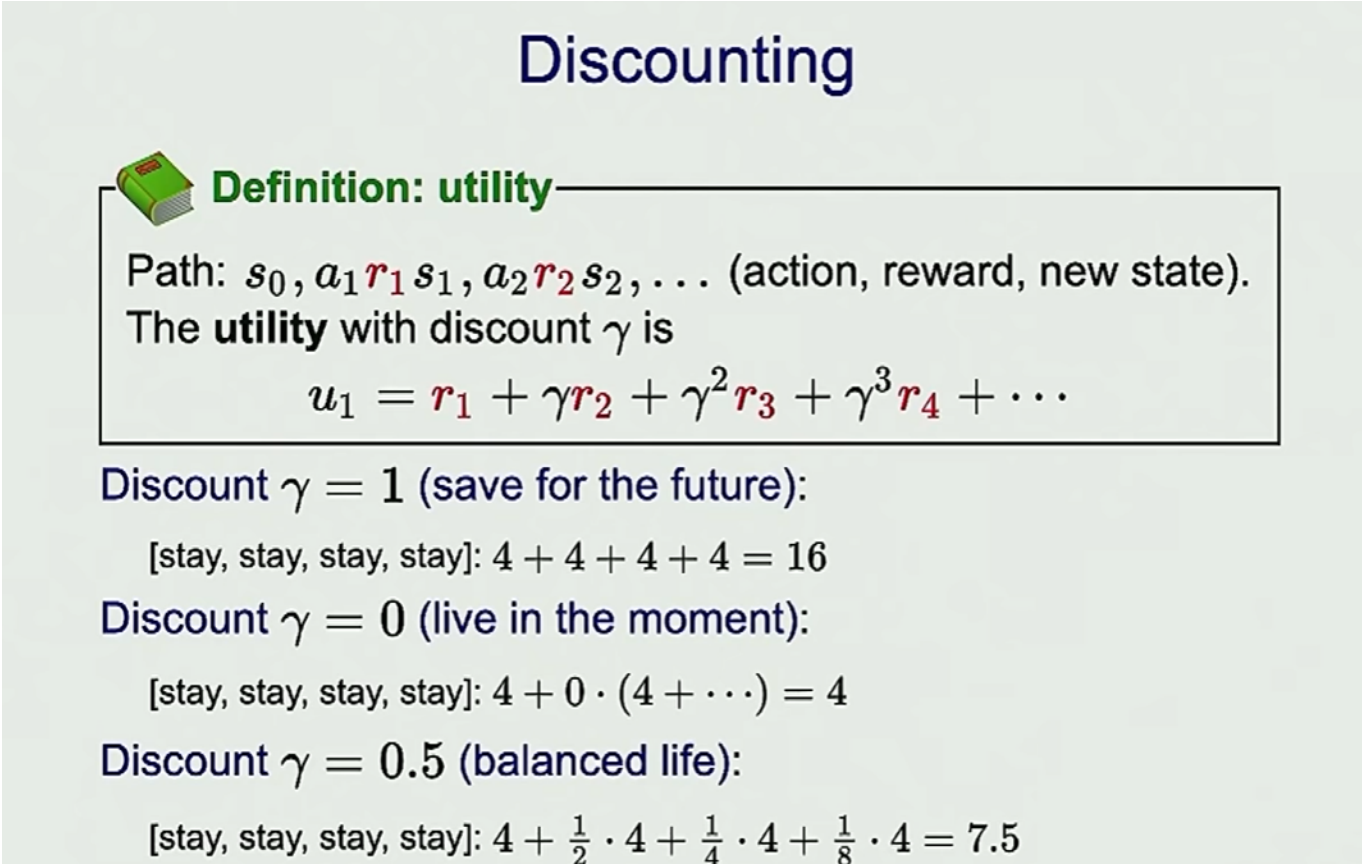
And the value is the expected utility

The value of a policy is the expected utility of that policy And that will be a number

So everytime we run under some policy, we gonnd get different form of untility (becasue of the probobility) But the value won't change Because it is the expectation of all untilities(statictically)

Discounting

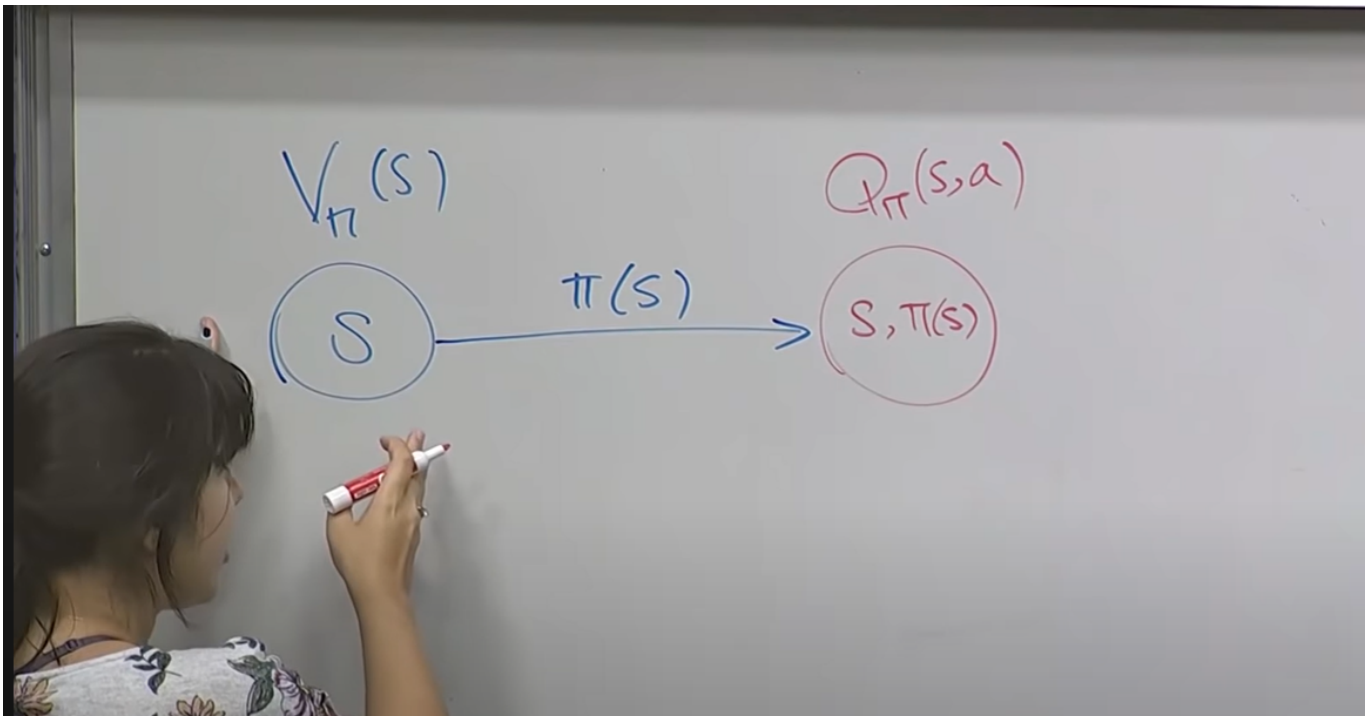
The idea of this discount factor is that: I might care about the future differently from how much I care about it now



So if gamma is 1, then the value in the future is the same as it is now IF gamma is 0, then we don't care about the future at all, we live at the moment(greedy) So in reliaty we'll be somewhere in between

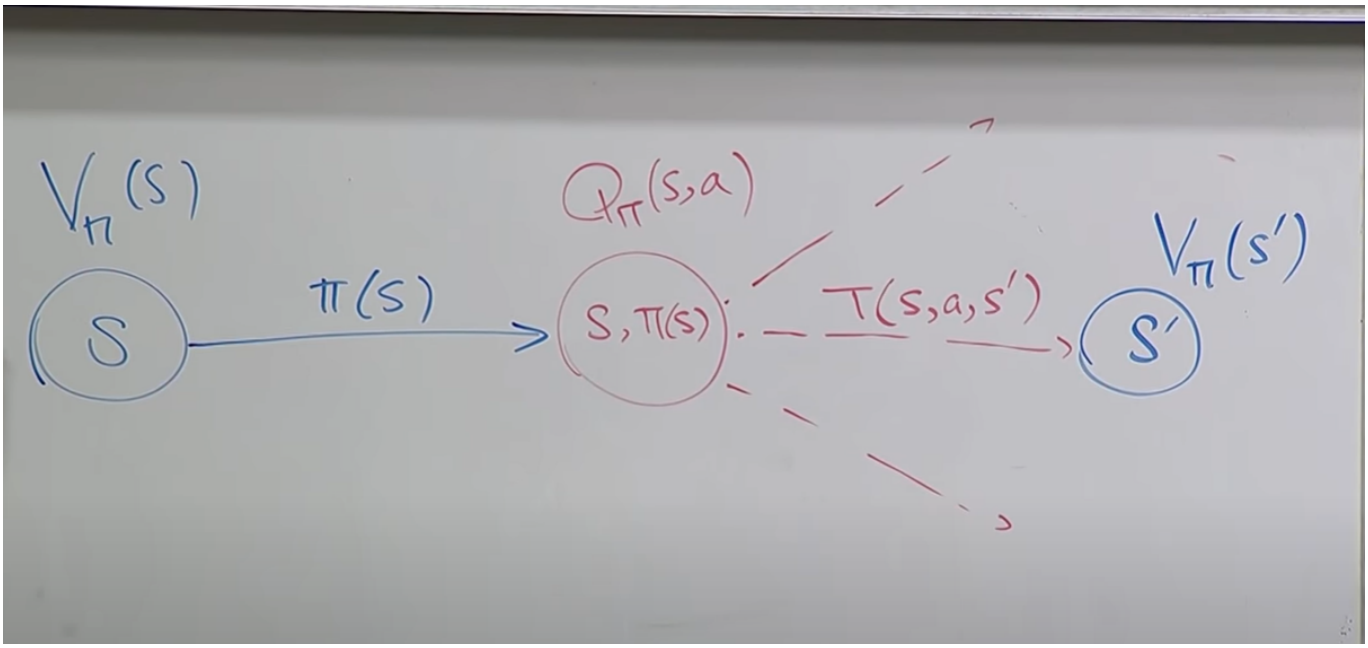
Gamma is not really a hyperprameter that we need to tune, we shall choose the the one that fit for the problem

Policy Evaluation



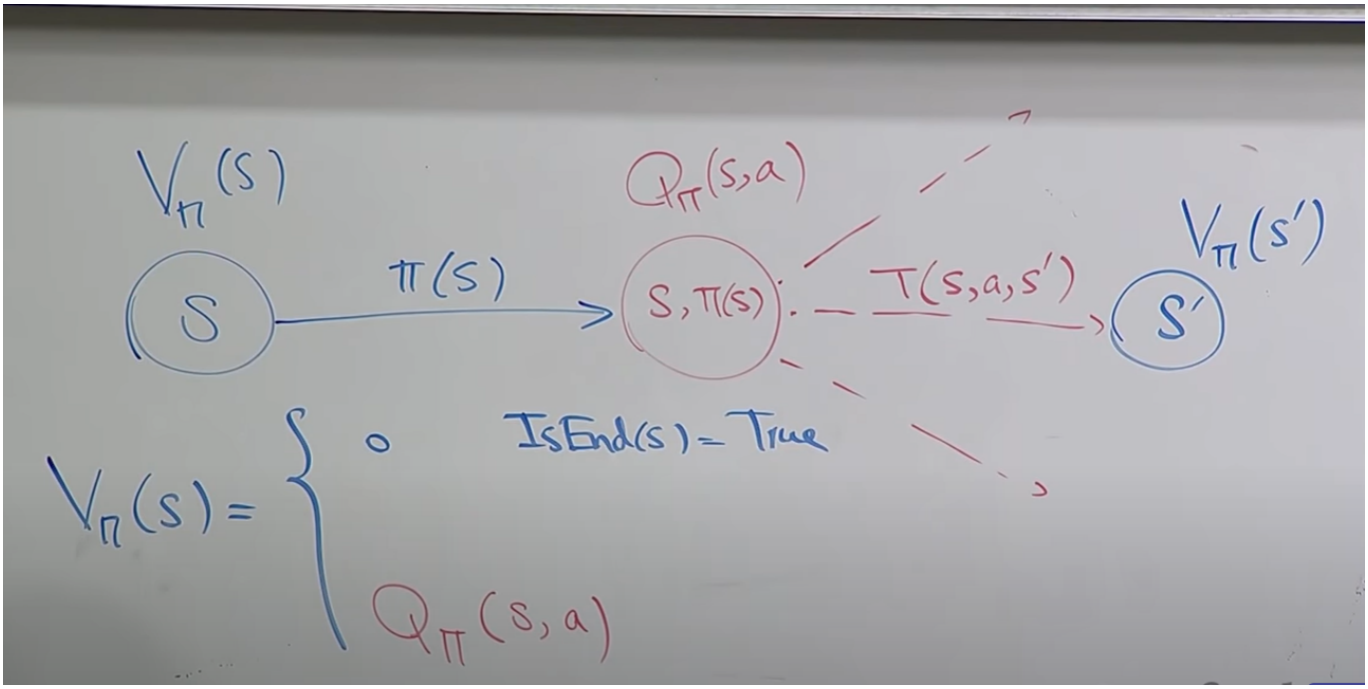
Value is the expected utility from actual state S, IF we take policy pi(s). When we take policy pi(s), we will endup at a chance node Qpi(S,a), where the state is in between and the probility is gonna come to play and decide the actual state you are gonna reach

The Q value is expected utilities from the chance node

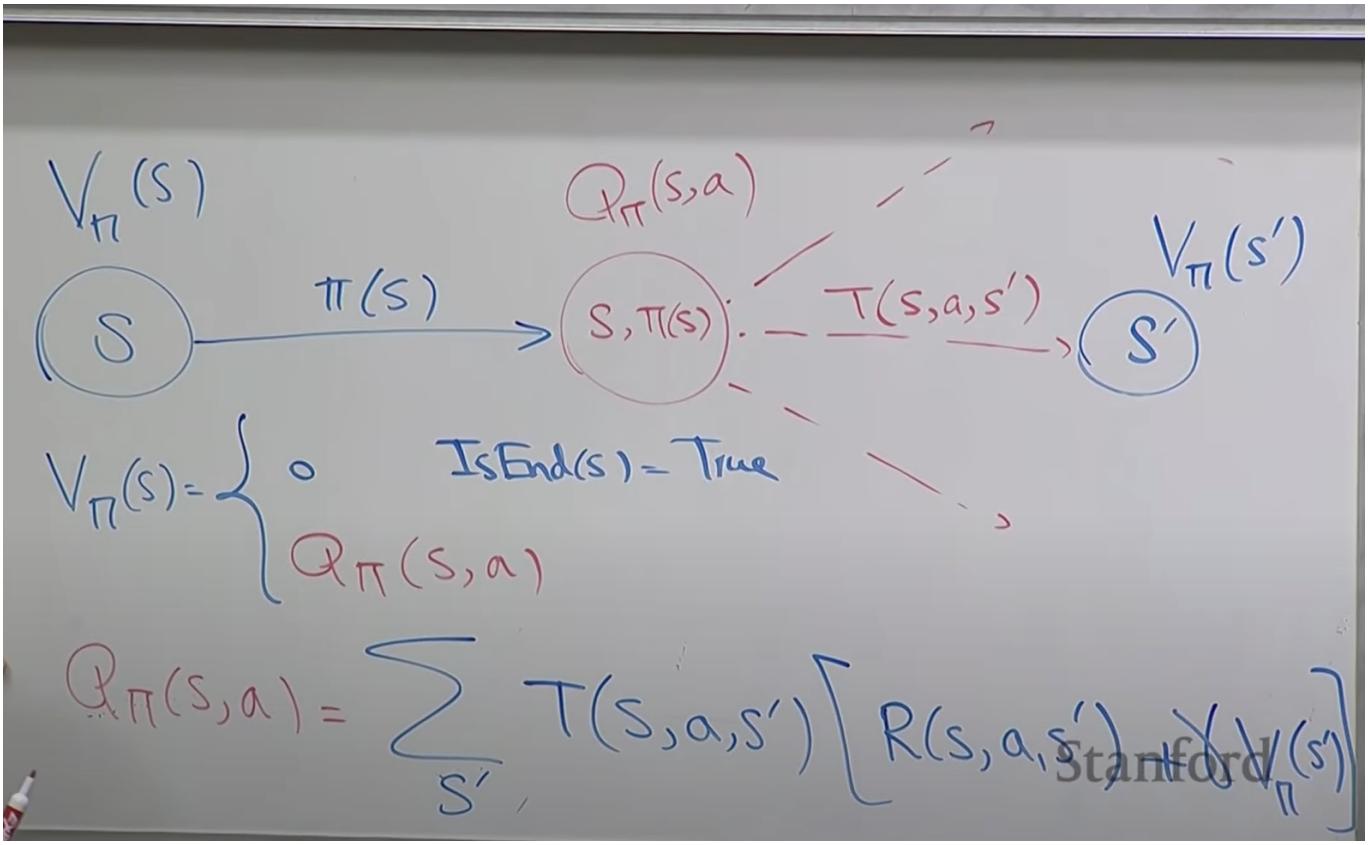


After the nature/probility is played, you are in a new state S', the value of the new state is `Vpi(S')`

So, what is Vpi(s) equal to



- Gonna be 0 if I am in a end state
- OR Qpi becase i already take action a



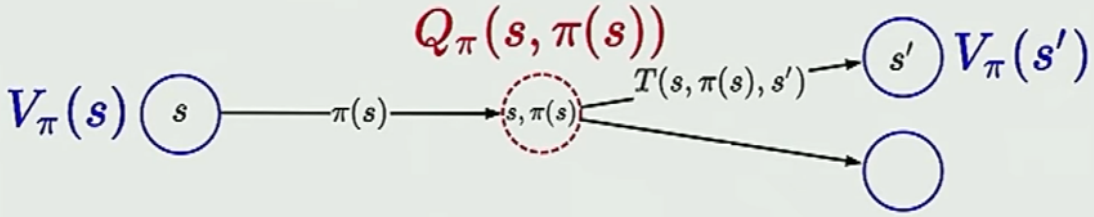
Qpi(S,a) is the sum of probility of state S' that we could endup at * the utility of this state

```
Transition(probility) * [immidiate reward of taking this action to this state +
discounted value(value of new state/future)]
```

Qpi(S,a) is the sum of expectations where all the places that I can end up at

Policy evaluation

Plan: define recurrences relating value and Q-value

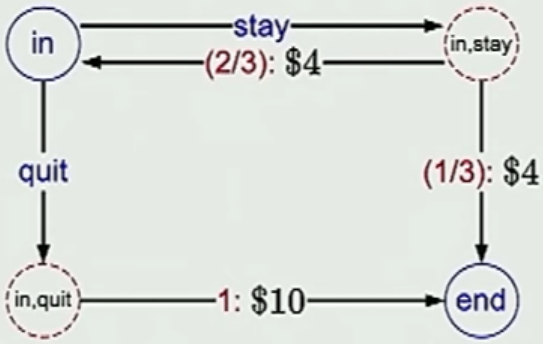


$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

Let's say somebody comes in and tell me the policy is to stay

Dice game



(assume $\gamma = 1$)

Let π be the "stay" policy: $\pi(\text{in}) = \text{stay}$.

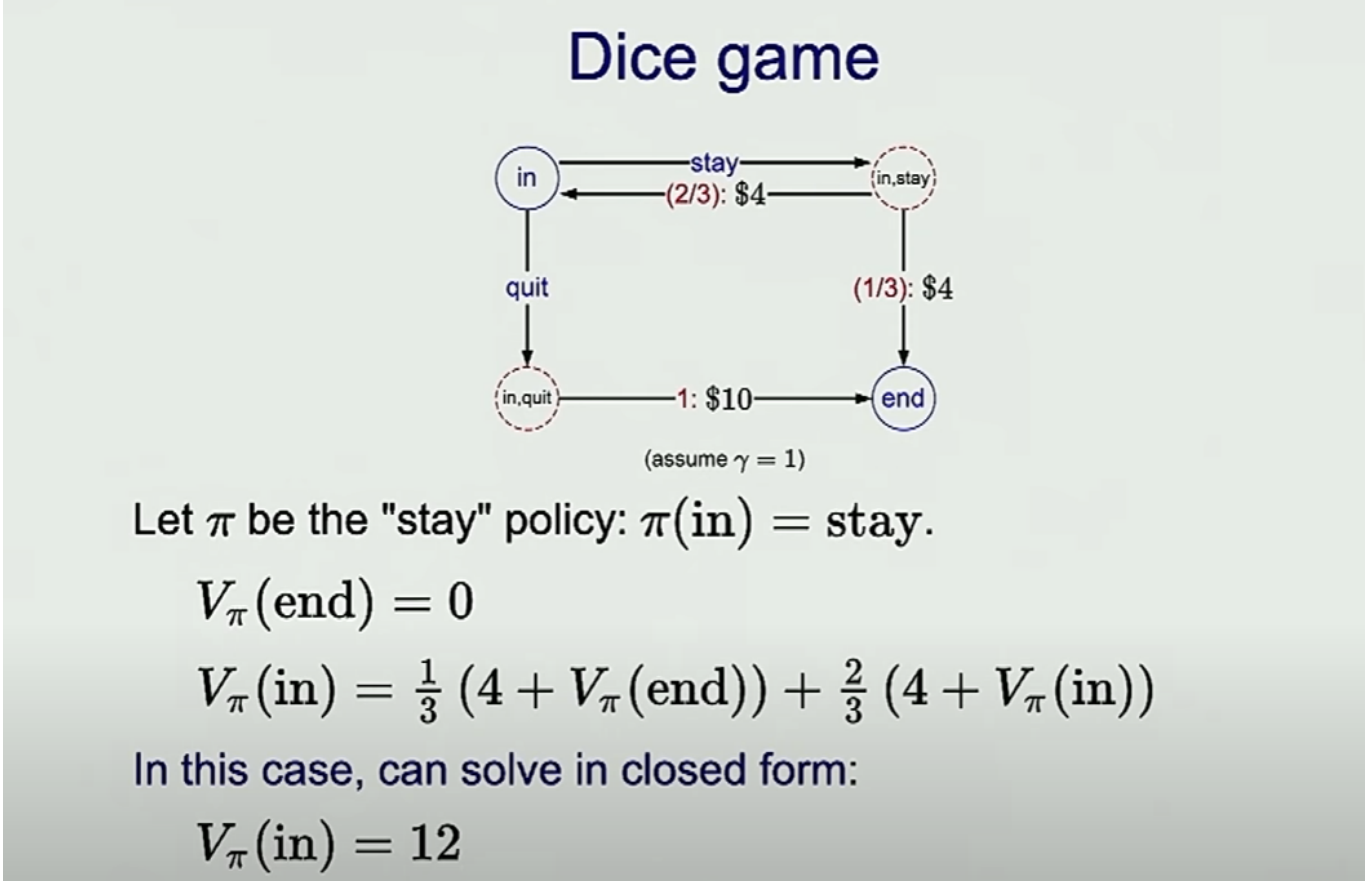
$$V_\pi(\text{end}) = 0$$

$$V_\pi(\text{in}) = \frac{1}{3}(4 + V_\pi(\text{end})) + \frac{2}{3}(4 + V_\pi(\text{in}))$$

In this case, can solve in closed form:

$$V_\pi(\text{in}) = \frac{1}{3}4 + \frac{2}{3}(4 + V_\pi(\text{in}))$$

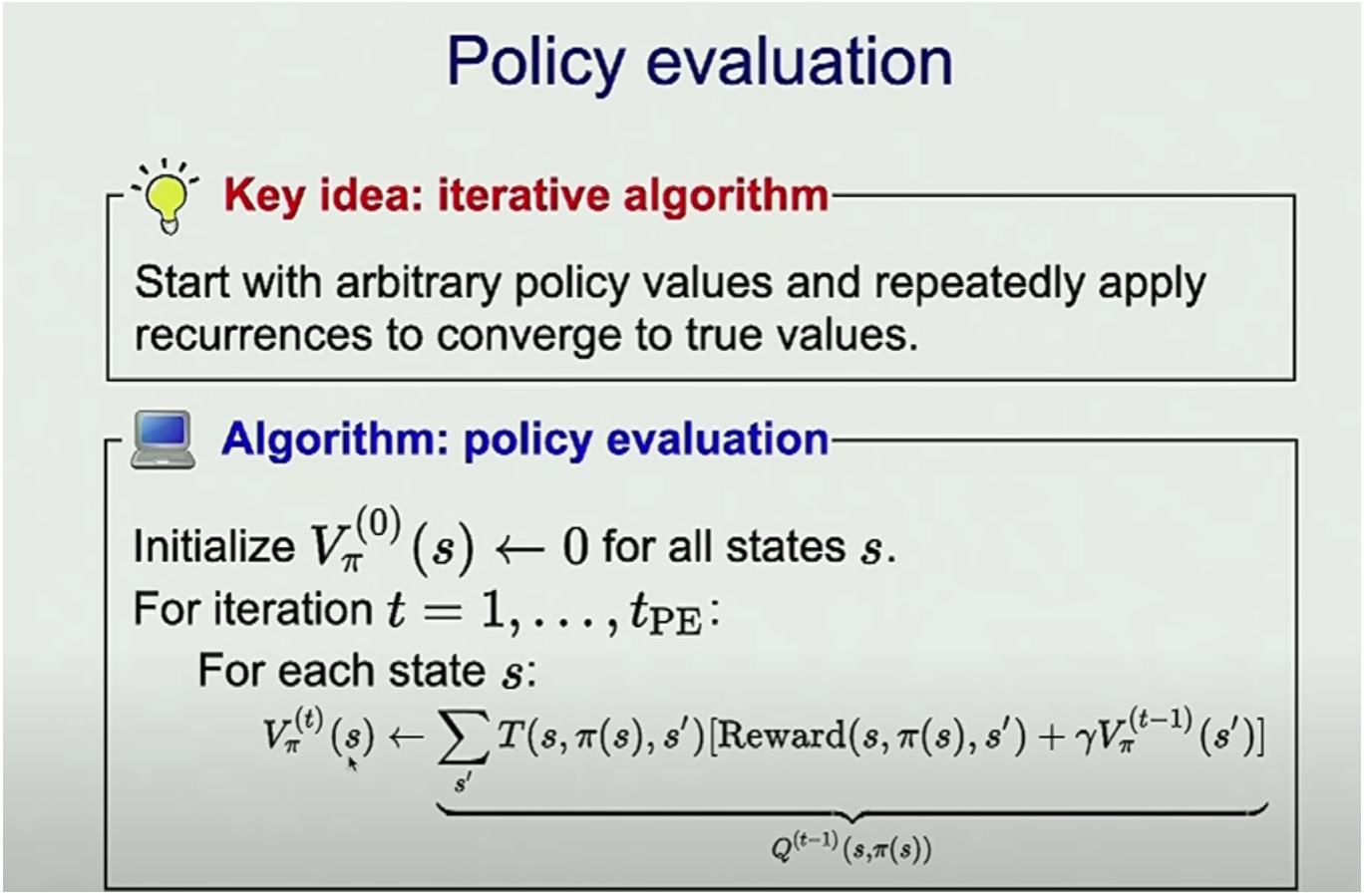
CS221 / Autumn 2019 / Liang & Sadigh



Itratively find the value

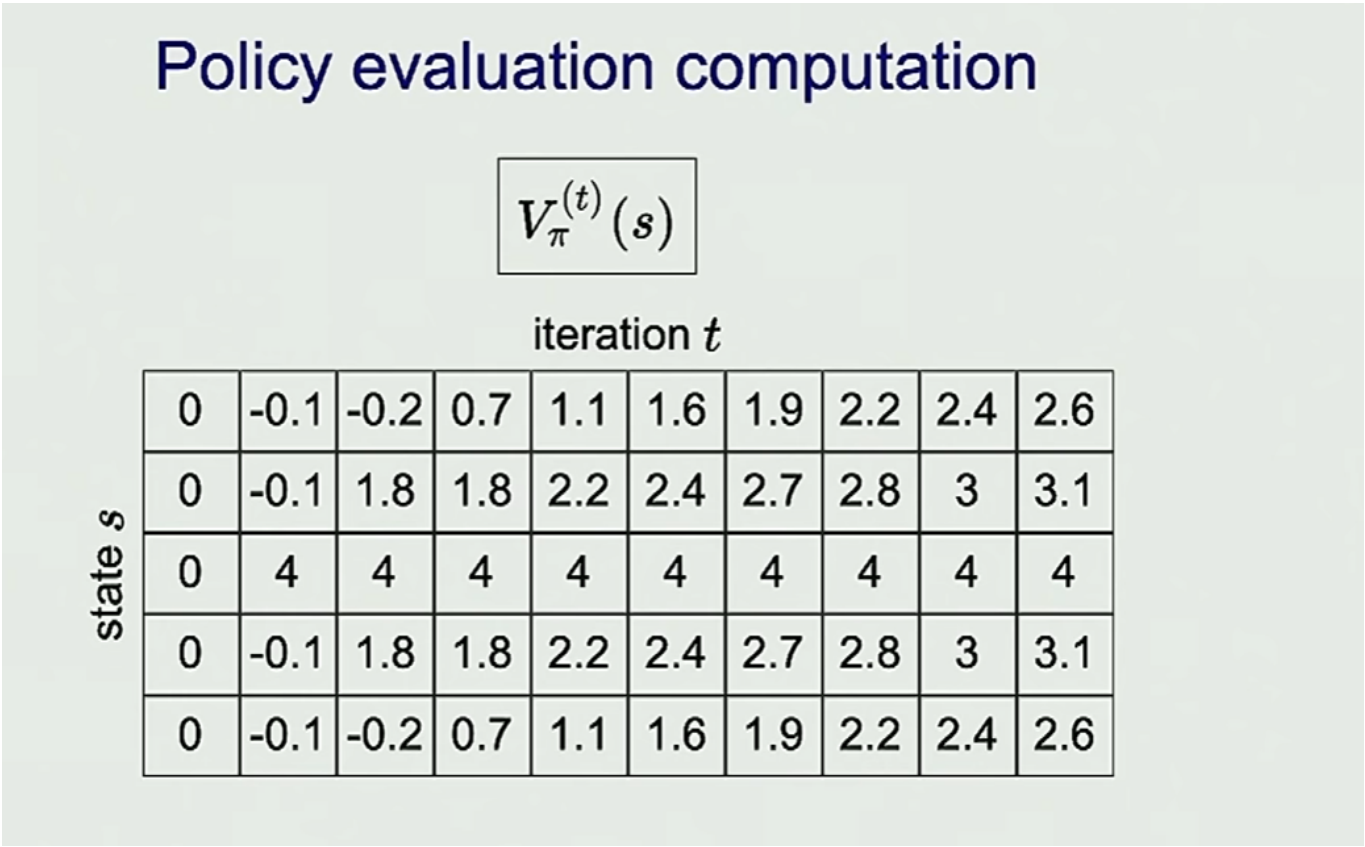
In the previous example we just solve the Vpi(S) from the equiation (By canceling the same state) However in the reality there can be much more and more states, and we cannot just solve from equation

In this case we can itratively try to find the value Vpi(S)



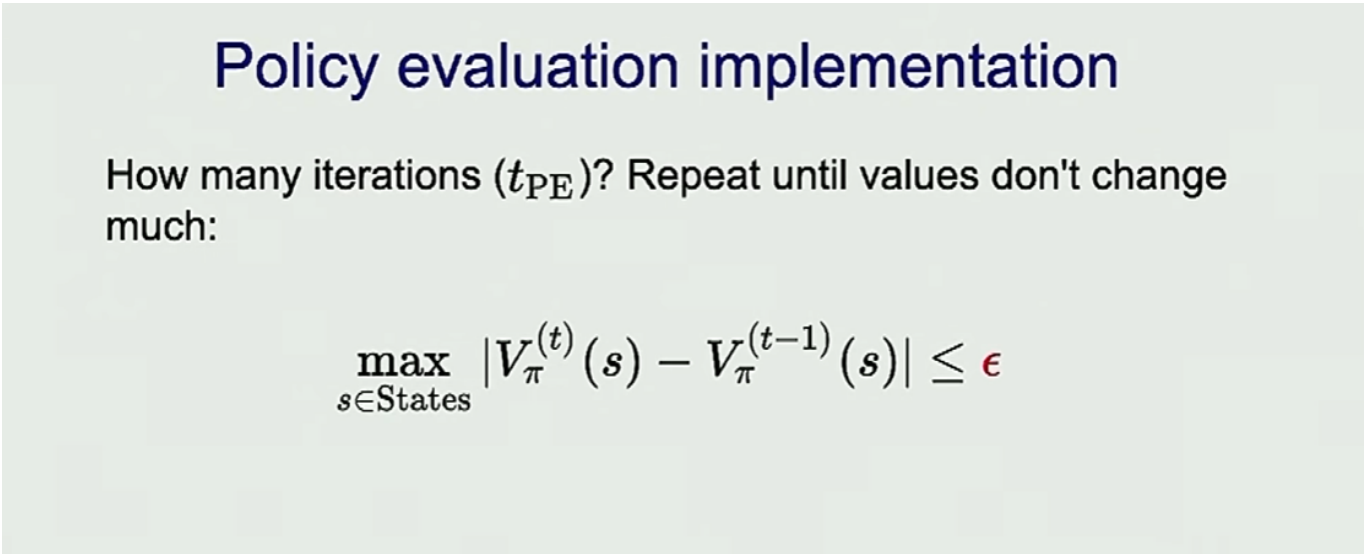
We initialize the values to some numbers close to 0 And we interate some number of times For every state, we will update the value with that formula, the same equation with the value of the previous step/itration ?

This allows us to compute new values based on the previous values that I've had I start with close to 0, and update values of all states and keep going itrative updating, and converged to true value

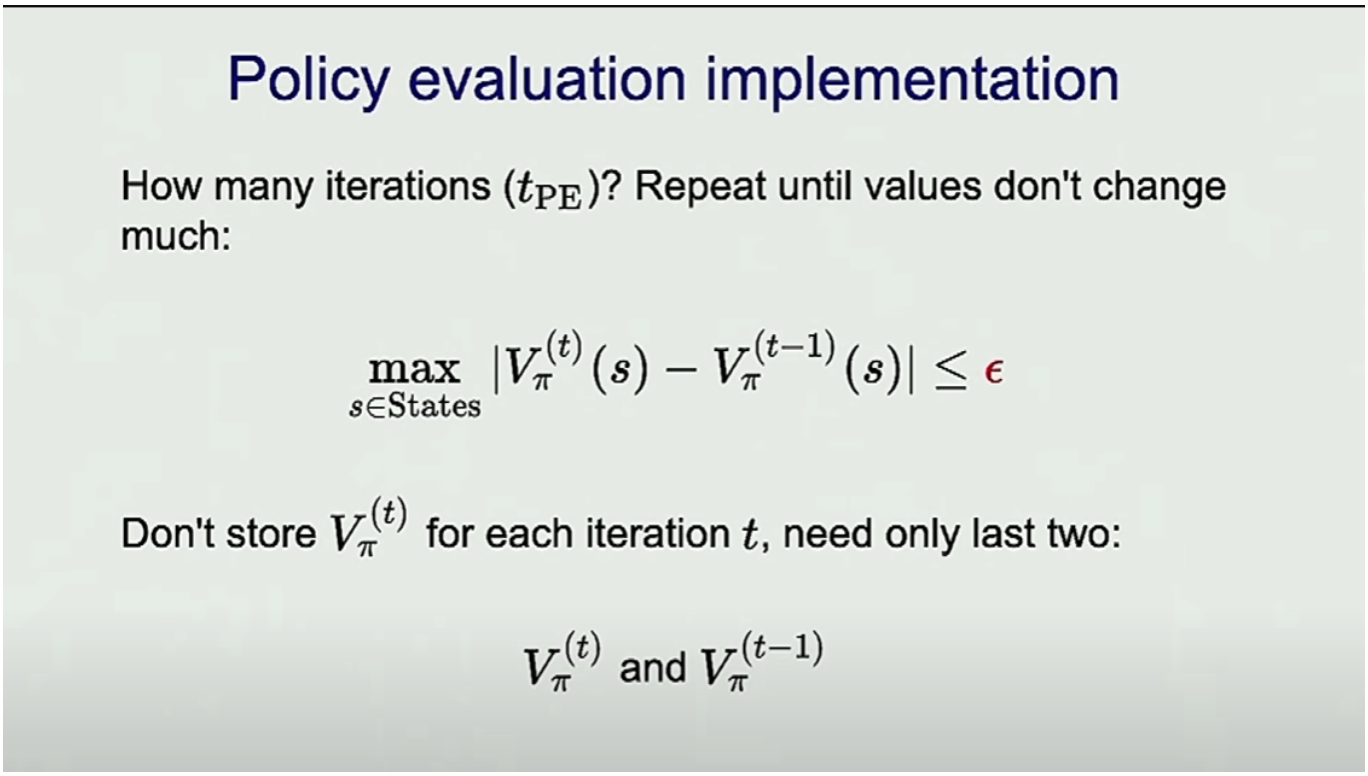


How long we should run this?

We can keep track of the difference between the value of previous time step and this time step If that is below some threshold, then we can call it done and believe the right value has been found

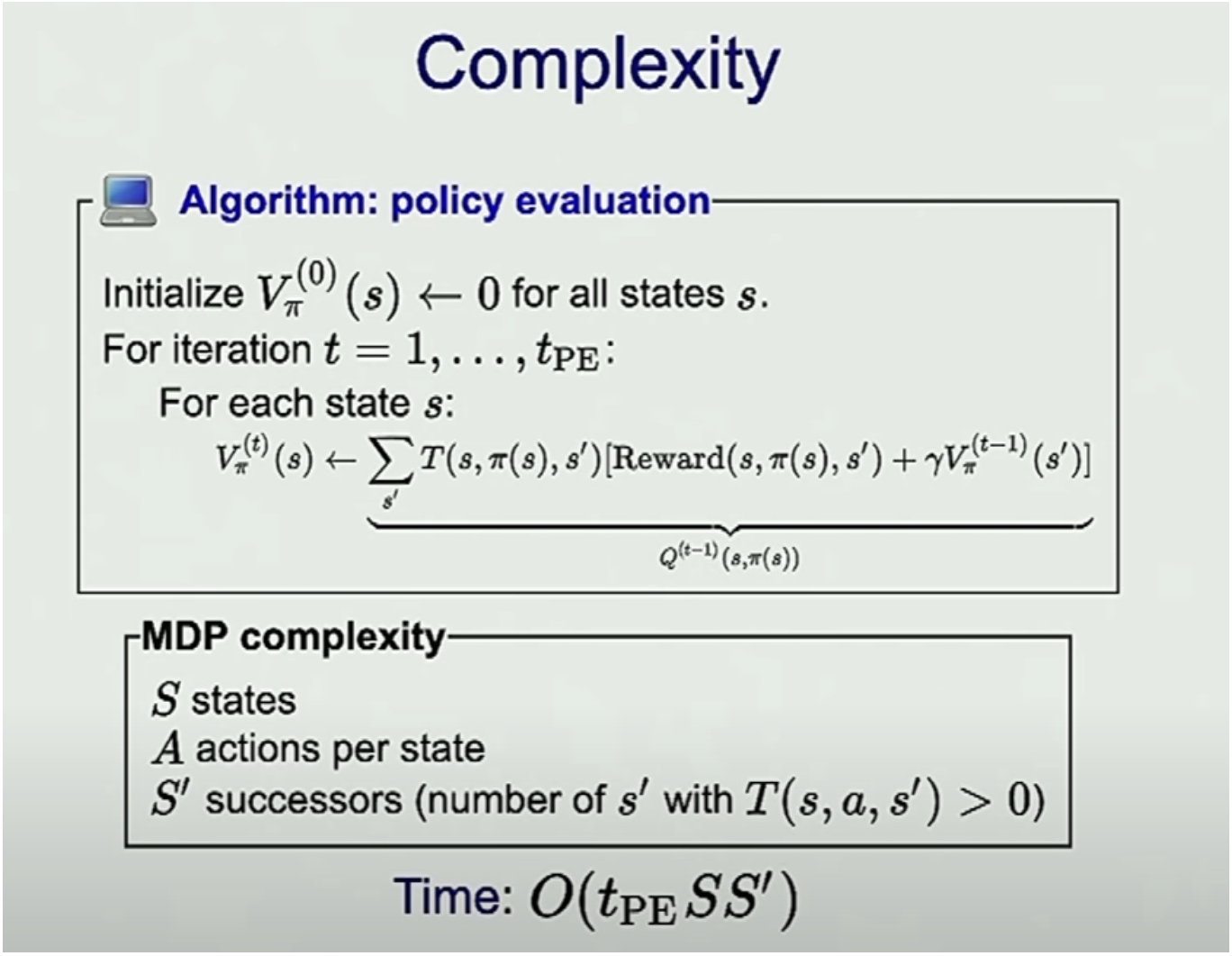


We are looking for the diff between iteration t and t-1 and taking max for all possible states, becasue i want the value to be close for all states



Only store the last two columns of the table to reduce space complexity

Complexity



Worst case: S' = S

Value Itration

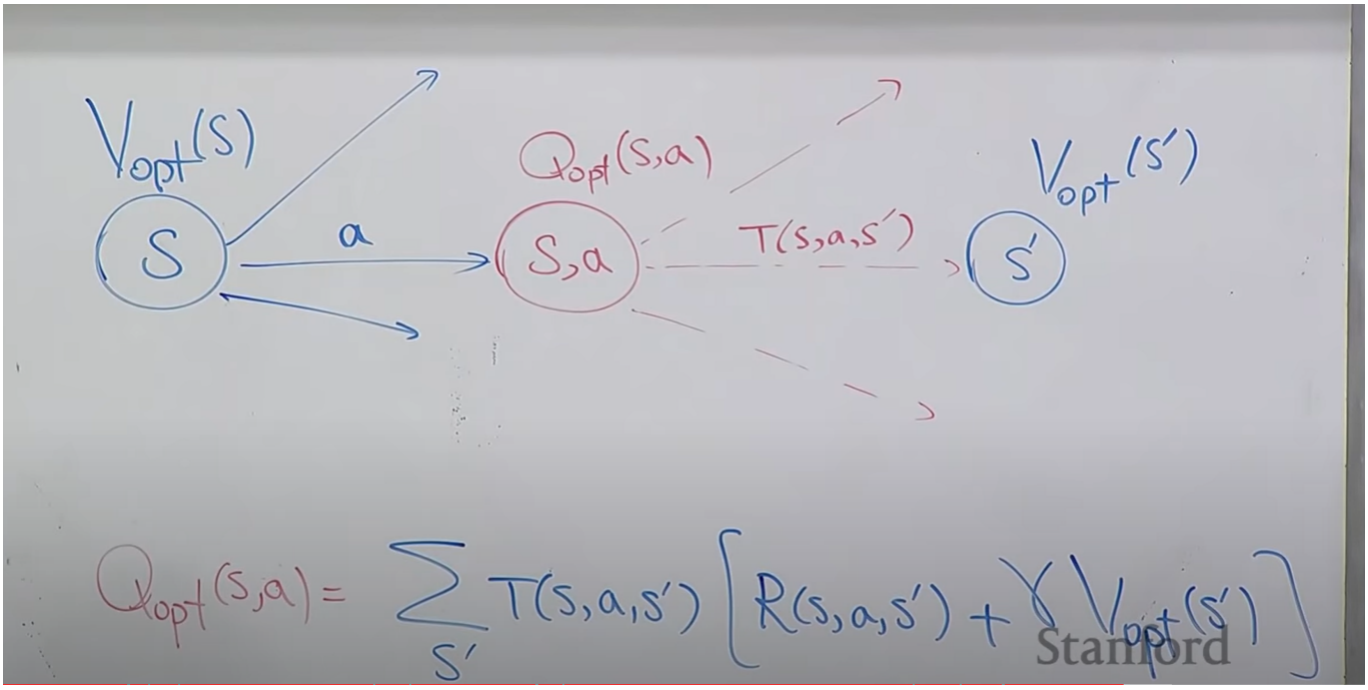
To find the right policy

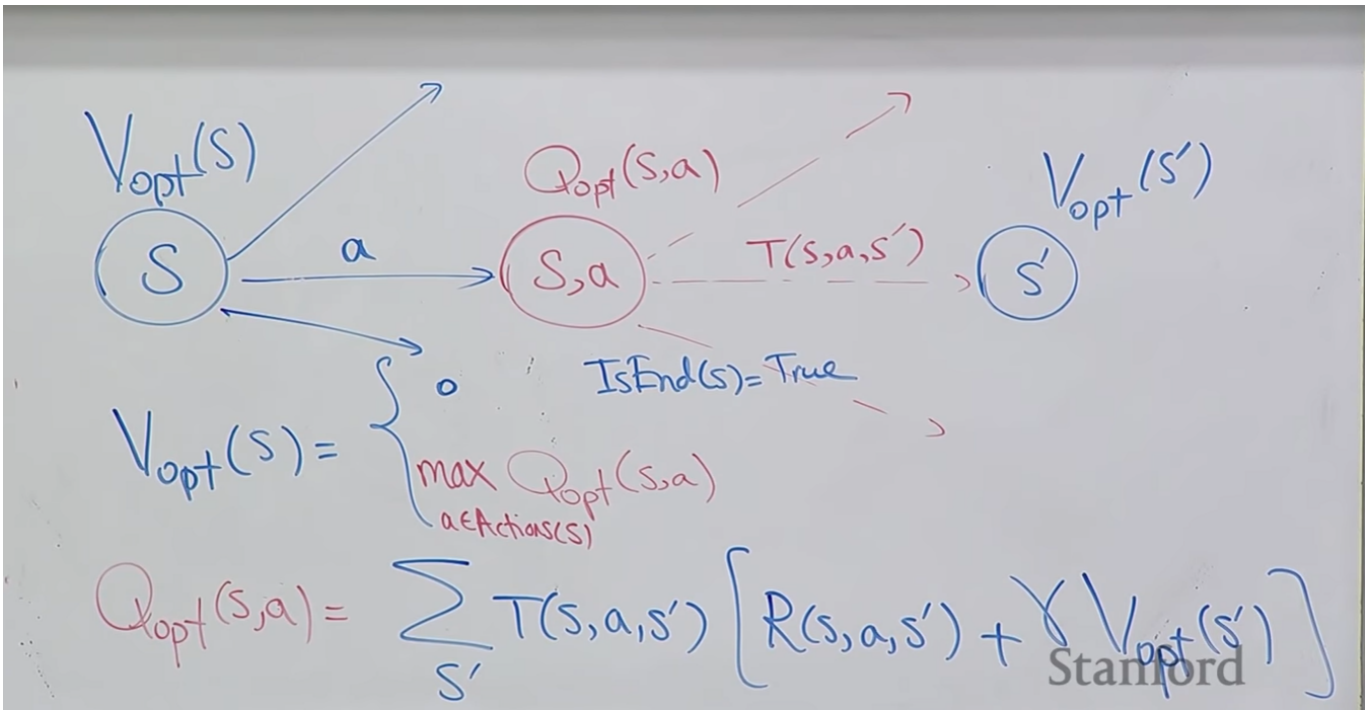
Optimal value and policy

Goal: try to get directly at maximum expected utility

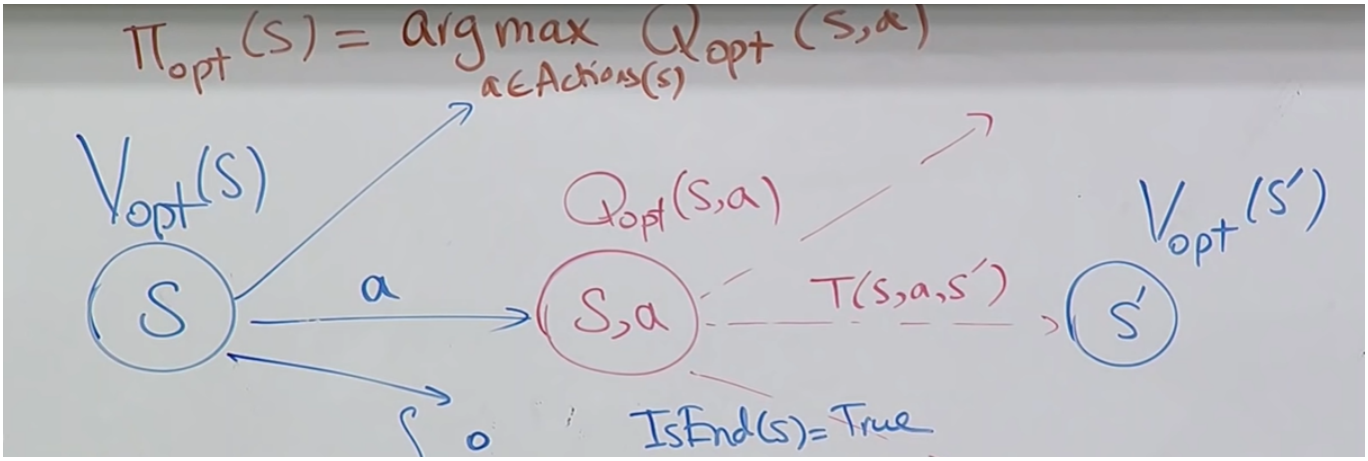
Definition: optimal value

The **optimal value** $V_{\text{opt}}(s)$ is the maximum value attained by any policy.





Just pick the maximun of Qopt

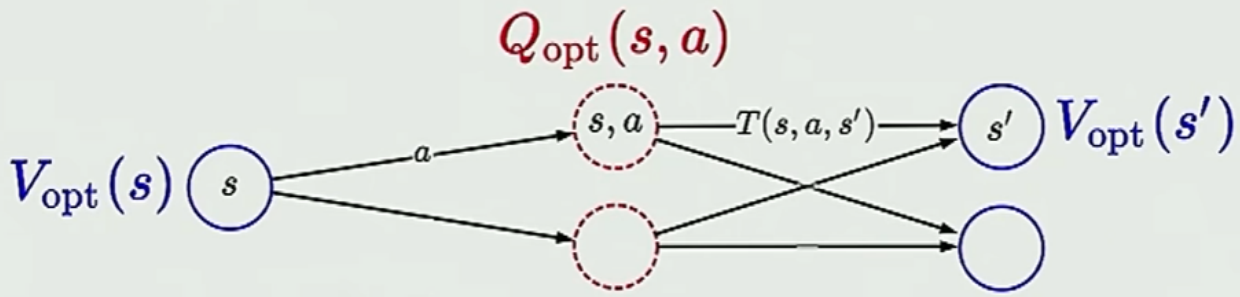


what is Argmax ?

> maxf(x) is the maximum value (if it exists) of f(x) as x varies through some domain, while argmaxf(x) is the value of x at which this maximum is attained(达到).

However, there could be more than one x value which gives rise to the maximum f(x) , in which case argmaxf(x) would be this set of values of x instead. 简而言之， 最大值和达到最大值时的自变量x值，自变量x可能不唯一

Optimal values and Q-values



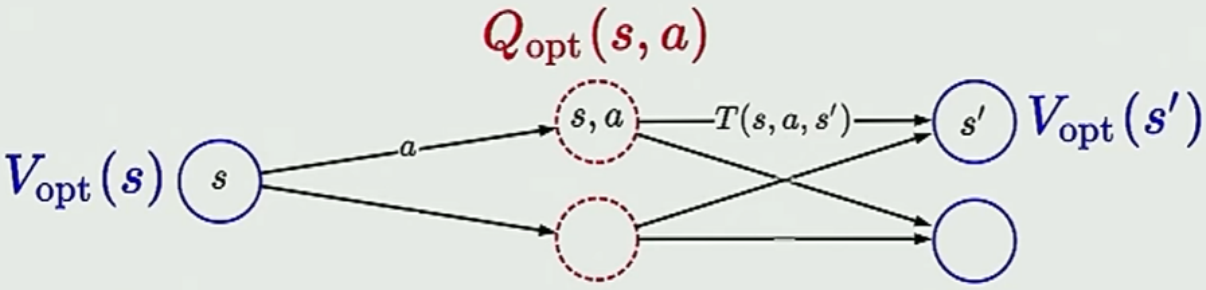
Optimal value if take action a in state s:

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')].$$

Optimal value from state s:

$$V_{\text{opt}}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s,a) & \text{otherwise.} \end{cases}$$

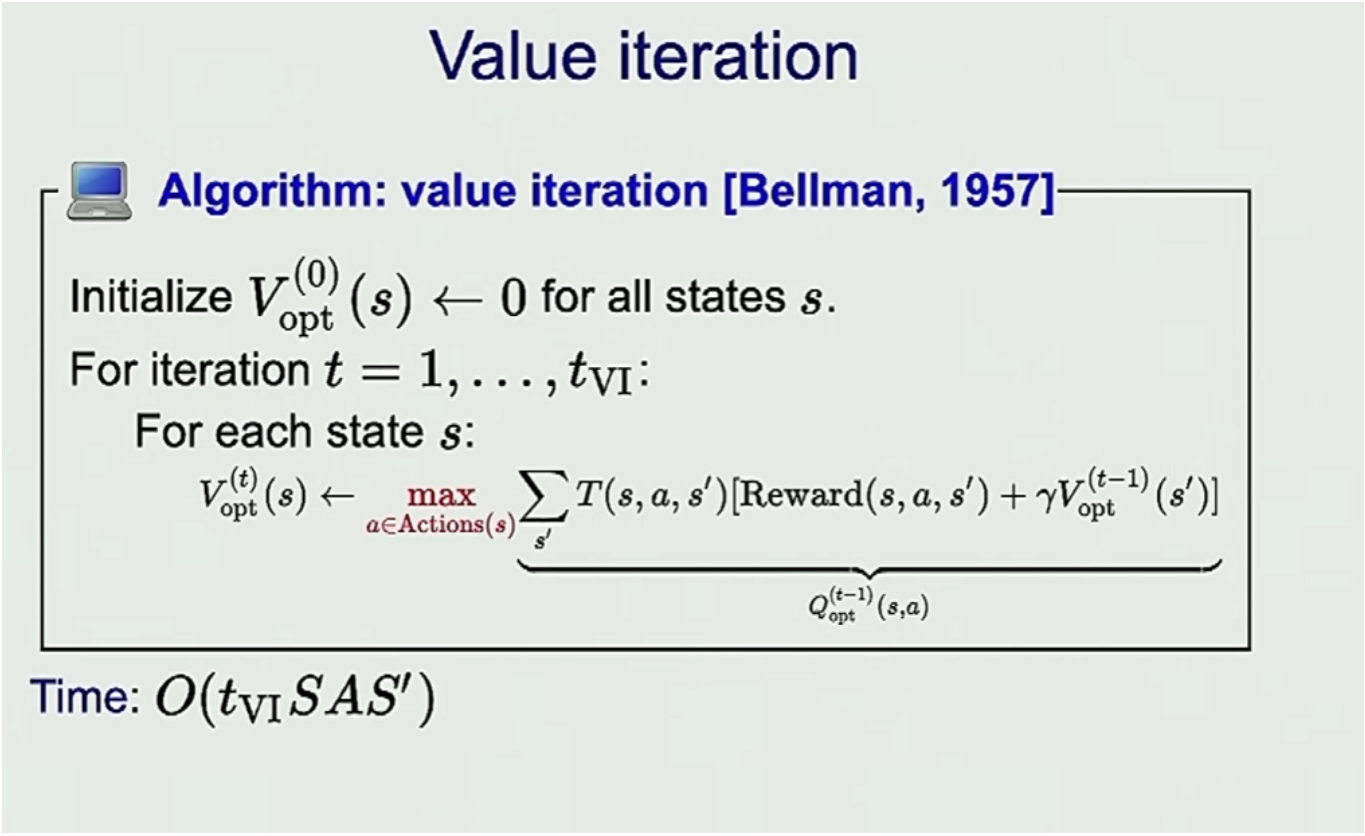
Optimal policies



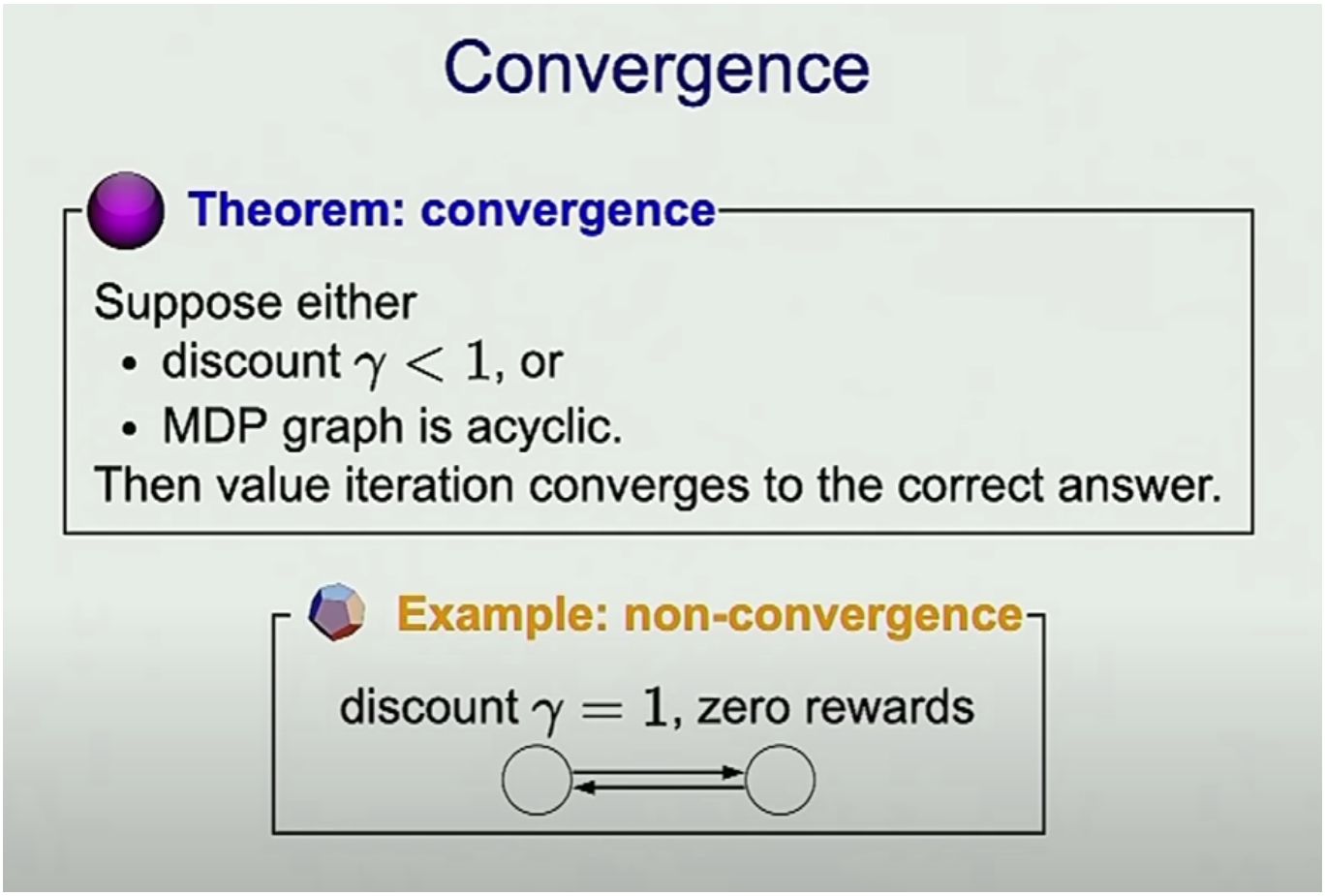
Given Q_{opt}, read off the optimal policy:

$$\pi_{\text{opt}}(s) = \arg \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s,a)$$

Complexity

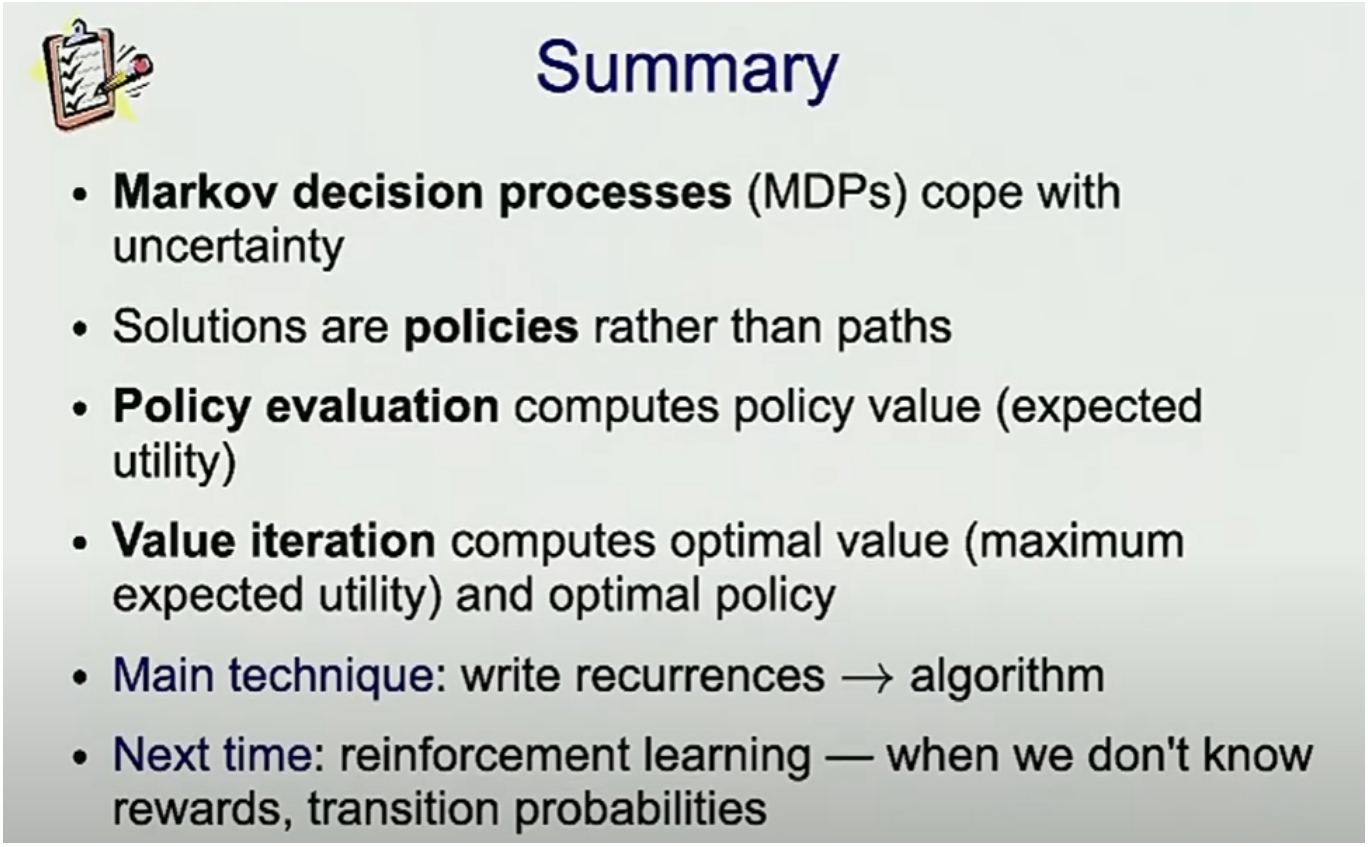


Convergence



If there's no cycles, ur just doing dynamic programming, it's gonna converge, ur gonna find the optimistic one

However if u have cycles, you should make ur discount rate < 1, otherwise it will never converge (Cuz you will just keep going between cycles)



Code

```
from enum import Enum
import os
import sys
from queue import PriorityQueue

sys.setrecursionlimit(100000)

Destination = 10

### Model (Search Problem)
class TransportationProblemMDP(object):

    WALK_REWARD = -1
    TRAM_REWARD = -2

    WALK = "walk"
    TRAM = "tram"

    TRAM_FAIL_PROB = 0.5
    def __init__(self, destination):
        # N number of blocks
        self.destination = destination

    def startState(self) -> int:
        return 1
```

```
    def isEnd(self, state):
        return state == self.destination

    def actions(self, state):
        '''
        Return a list of valid actions that we can take form the current state
        '''
        result = []
        if(state + 1 <= self.destination):
            result.append(self.WALK)

        if(state * 2 <= self.destination):
            result.append(self.TRAM)

        return result


    def succProbAndReward(self, state : int, action: str):
        '''
        Input:
            current state
            Action that we gonna take
        Return a list of (newState, prob, rewards) triples
        Meaning return the a list of:
            the new state we are gonna endup at,
            what probability of it,
            and the reward we get from it

        state = s, action = a, newState = s'
        prob = T(s, a, s'), reward = Reward(s, a, s')
        '''
        result = []
        if(action == self.WALK):
            # It is deterministic, so the prob is 1.
            result.append((state + 1, 1., self.WALK_REWARD))
        elif(action == self.TRAM):
            # There's a 50% chance that the tram could fail
            result.append((state, self.TRAM_FAIL_PROB, self.TRAM_REWARD))
            result.append((state * 2, 1. - self.TRAM_FAIL_PROB, self.TRAM_REWARD))

        return result

    def discountFactor(self):
        return 1.

    def states(self):
        return range(1, self.destination+1)

# Inference (Algorithms)
def valueIteration(mdp : TransportationProblemMDP):
    # Initialize
    valueDic = {} # state => Vopt[state]
    # Initialize all optimal value with 0
    for state in mdp.states():
```

```
        valueDic[state] = 0.

    def VQ(state, action):
        return sum(
            transProb * (reward + mdp.discountFactor() * valueDic[newState])
            for newState, transProb, reward in mdp.succProbAndReward(state,
action)
        )

    def isConverged(valueDic, newValueDic):
        '''
        We want the maximum value difference of all states to be smaller than the
treashold
        '''
        return max(abs(valueDic[state] - newValueDic[state])  for state in
mdp.states()) < 1e-10


    while True:
        # Compute the new values (new V) with old values (V)
        newValueDic = {}

        for state in mdp.states():
            if mdp.isEnd(state):
                newValueDic[state] = 0
                break

            newValueDic[state] = max(
                VQ(state, action)
                for action in mdp.actions(state)
            )

        # Check for convergence
        if (isConverged(valueDic, newValueDic)):
            break

        valueDic = newValueDic

        # Read out policy:
        pi = {}
        for state in mdp.states():
            if mdp.isEnd(state):
                pi[state] = "none"
            else:
                # Otherwise it's gonna be the argmax of Q values
                # Which is the input that can maximize Q values
                # And how we gonna find the argmax?
                #  - We just gonna try out different actions and see
                pi[state] = max(
                    (VQ(state, action), action) for action in mdp.actions(state)
                )[1]

        # Print stuff out
        os.system('clear')
```

```
        print('{:25} {:25} {:25}'.format('state', 'valueDic[state]',
'pi[state])'))

        for state in mdp.states():
            print('{:5} {:15} {:55}'.format(state, valueDic[state], pi[state]))
        input()


mdp = TransportationProblemMDP(destination=10)
# print(mdp.actions(3))
# print(mdp.succProbAndReward(3, "tram"))
valueIteration(mdp)
```