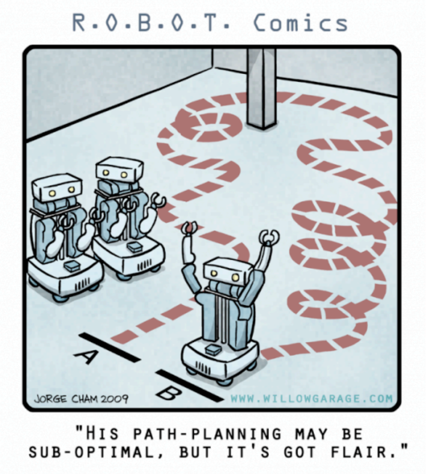
Sequential Decision Making Under Uncertainty

material from Marie desJardin, Lise Getoor, Jean-Claude Latombe, Daphne Koller, Stuart Russell, Dawn Song, Mark Hasegawa-Johnson, Svetlana Lazebnik, Pieter Abbeel, Dan Klein



"HIS PATH-PLANNING MAY BE SUB-OPTIMAL, BUT IT'S GOT FLAIR."

1

Bookkeeping

- Phase I (writeup and code) due **17th**
- Today: "Planning" under uncertainty (sequential decision making)
- Next lecture: Finding optimal policies; Reinforcement Learning (RL)

2

A poor assumption in the planning we've seen so far

- NO UNCERTAINTY!
  - Assumes the agent knows everything about the world and what can happen in it.
- Sources of Uncertainty
  - Agent may not know all states of the world.
  - Agent may not know what state of the world it is in.
  - Outcomes of actions may not be known

3

Decision Making Under Uncertainty

- Many environments have multiple possible outcomes
- Some of these outcomes may be good; others may be bad
- Some may be very likely; others unlikely
- What's a poor agent to do??
- To understand how to plan under uncertainty, we need to revisit **expected utilities**

4

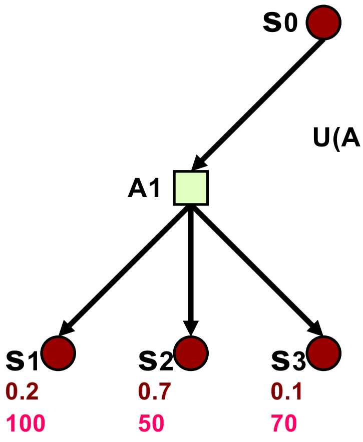
Review: Expected Utility

- Random variable X with n values $x_1,\dots,x_n$ and distribution $(p_1,\dots,p_n)$
  - E.g.: X is the state reached after doing an action A under uncertainty
- Function U of X
  - E.g., U is the utility of a state
- The expected utility of A is

$$EU[A] = \sum_{i=1,\dots,n} p(x_i|A)U(x_i)$$

5

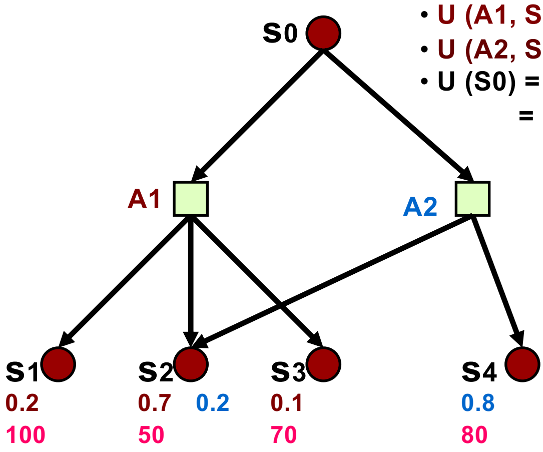
One State/One Action Example



U(A1, S0) = 100 x 0.2 + 50 x 0.7 + 70 x 0.1
= 20 + 35 + 7
= 62

6

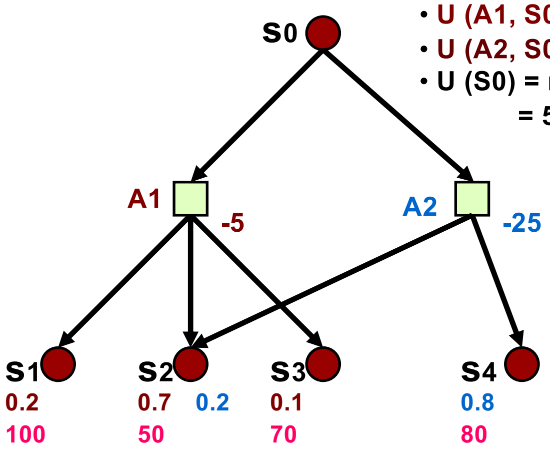
One State/Two Actions Example



- U (A1, S0) = 62
- U (A2, S0) = 74
- U (S0) = $\max_a$ {U(a,S0)}
  = 74

7

Introducing Action Costs



- U (A1, S0) = 62 – 5 = 57
- U (A2, S0) = 74 – 25 = 49
- U (S0) = $\max_a$ {U(a, S0)}
  = 57

8

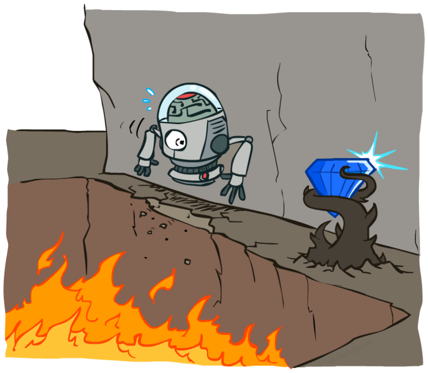
Review: MEU Principle

- A **rational agent** should choose the action that maximizes agent's expected utility
- This is the basis of the field of **decision theory**
- The MEU principle provides a **normative criterion** for rational choice of action
- So we know what to do when planning actions?

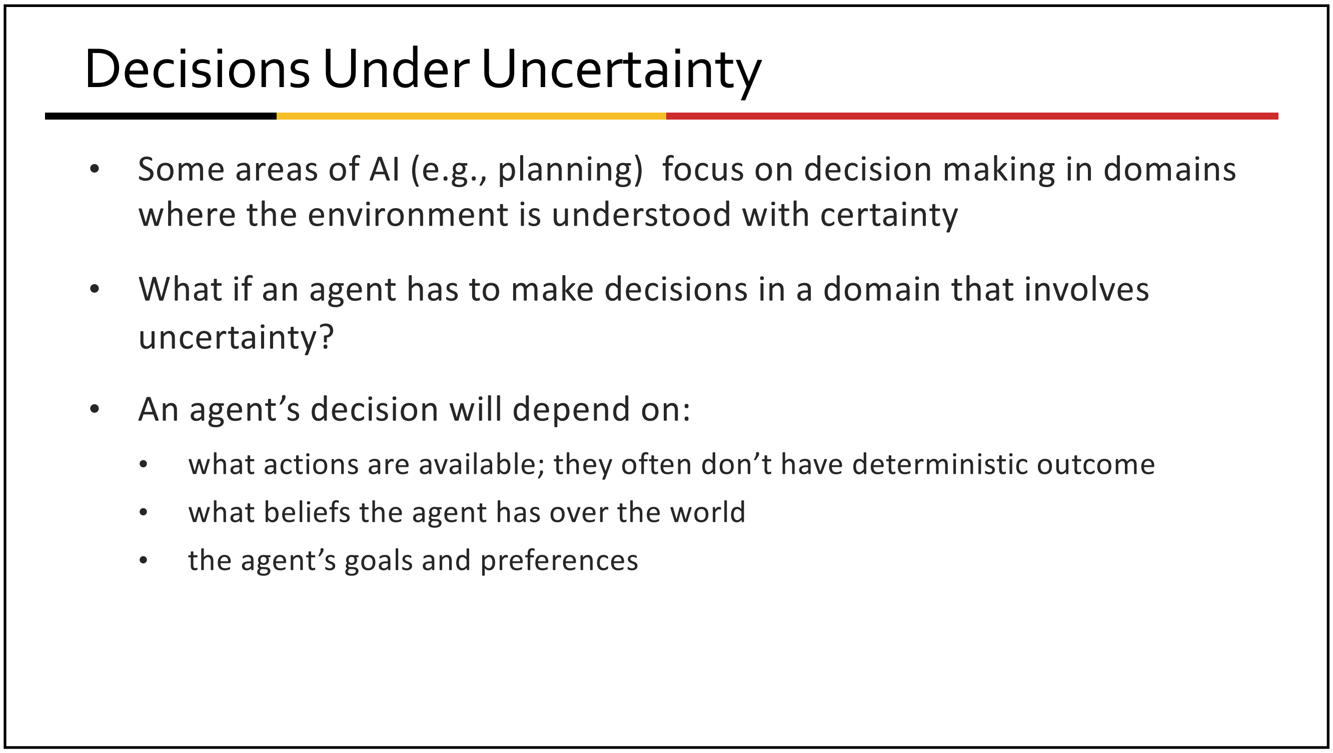
9

Sequential decisions under uncertainty

- So far, decision problem is one-shot—concerning only one action
- Sequential decision problem: agent's utility depends on a **sequence** of actions
- This is where we get into **planning**



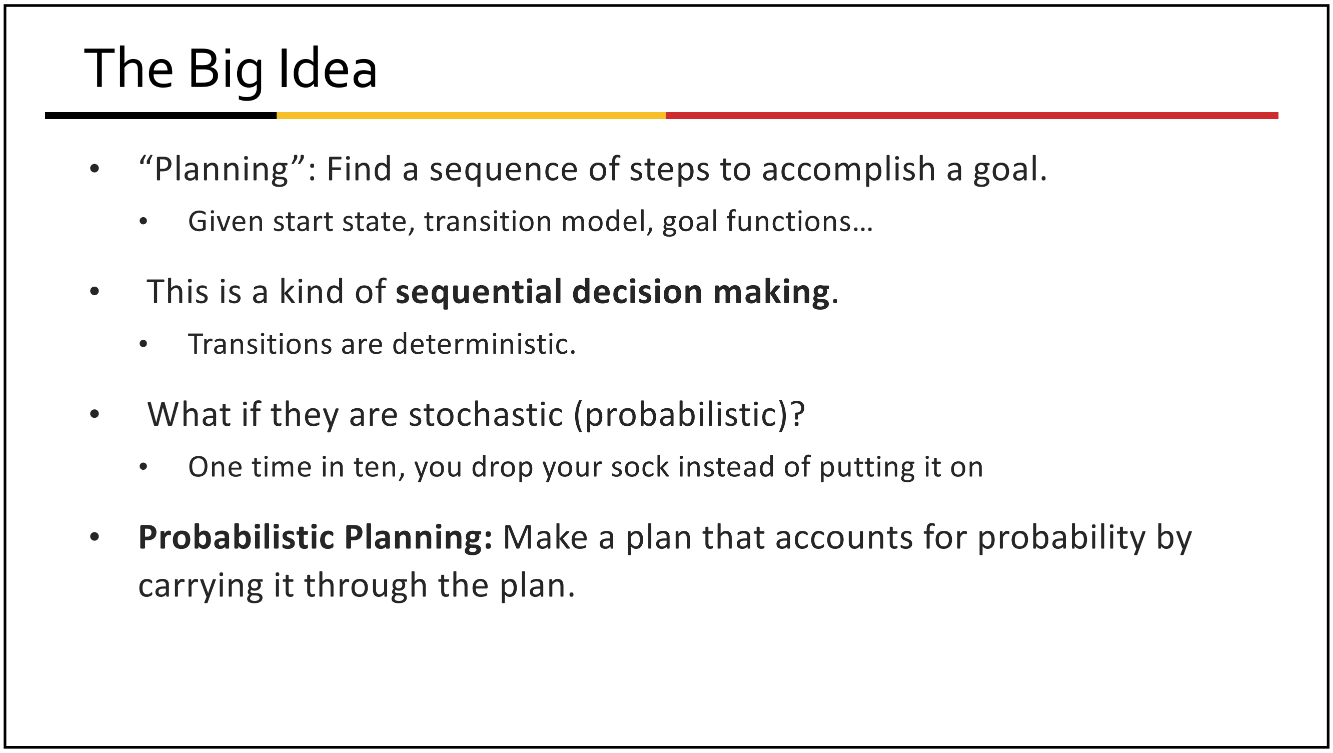
23



Decisions Under Uncertainty

- Some areas of AI (e.g., planning) focus on decision making in domains where the environment is understood with certainty
- What if an agent has to make decisions in a domain that involves uncertainty?
- An agent's decision will depend on:
  - what actions are available; they often don't have deterministic outcome
  - what beliefs the agent has over the world
  - the agent's goals and preferences

24



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 instead of putting it on
- **Probabilistic Planning:** Make a plan that accounts for probability by carrying it through the plan.

25

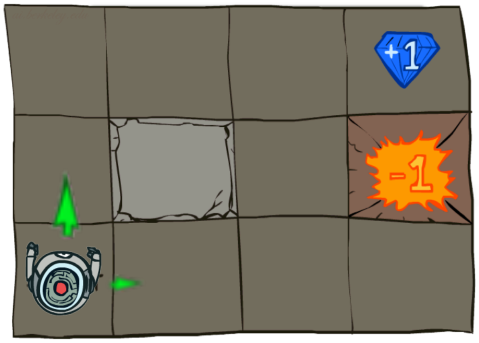
Decision Processes

- Often an agent needs to decide how to act in situations that involve sequences of decisions
  - The agent's utility depends upon the final state reached, and the sequence of actions taken to get there
- Would like to have an ongoing decision process. At any stage of the process:
  - The agent decides which action to perform
  - The new state of the world depends probabilistically upon the previous state as well as the action performed
  - The agent receives rewards or punishments at various points in the process
- **Aim: maximize the reward received**

26

Sequential Decision Problem Example

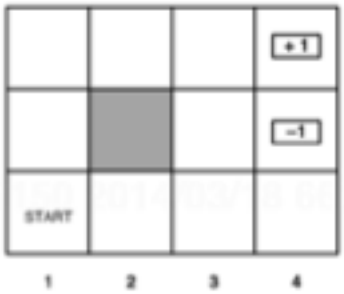
- Beginning at the start state, choose an action at each time step.
- Problem terminates when either goal state is reached.
- Possible actions are Up, Down, Left, and Right
- Assume that the environment is fully observable, i.e., the agent always knows where it is.



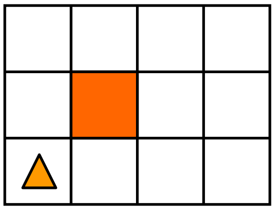
27

Sequential Decision Problem Example

- Deterministic Solution
- If the environment is deterministic and the objective is get the maximum reward, then the solution is easy:
- (Up, Up, Right, Right, Right)
- But that's assuming the agent always ends up where it thinks it is going
  - Robots do not in general do so ☹

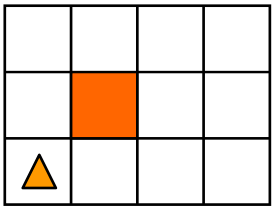


Simple Robot Navigation Problem



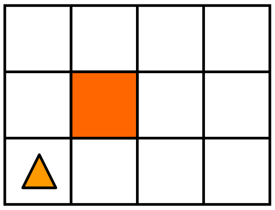
- In each state, the possible actions are U, D, R, and L

Probabilistic Transition Model



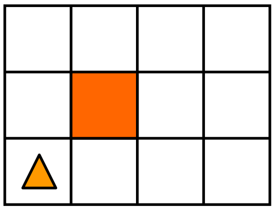
- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
  - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)

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)

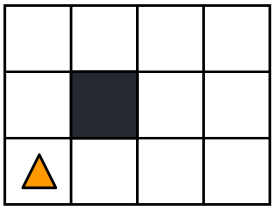
Probabilistic Transition Model

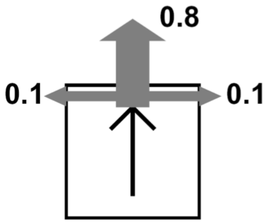


- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
  - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)
  - With probability 0.1, the robot moves right one square (if the robot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

32

Probabilistic Transition Model



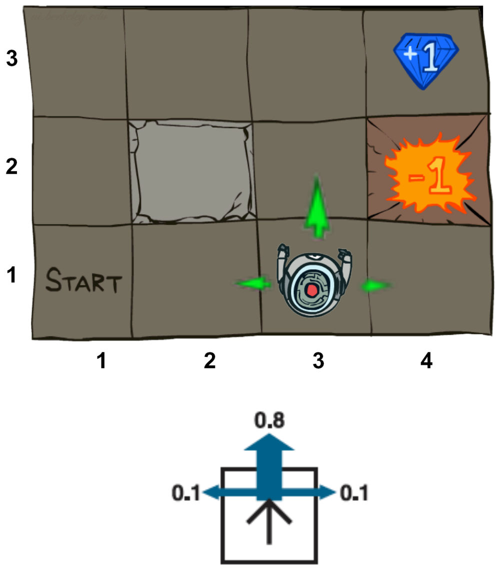


- In each state, the possible actions are U, D, R, and L
- The effect of U is as follows (transition model):
  - With probability 0.8, the robot moves up one square (if the robot is already in the top row, then it does not move)
  - With probability 0.1, the robot moves right one square (if the robot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

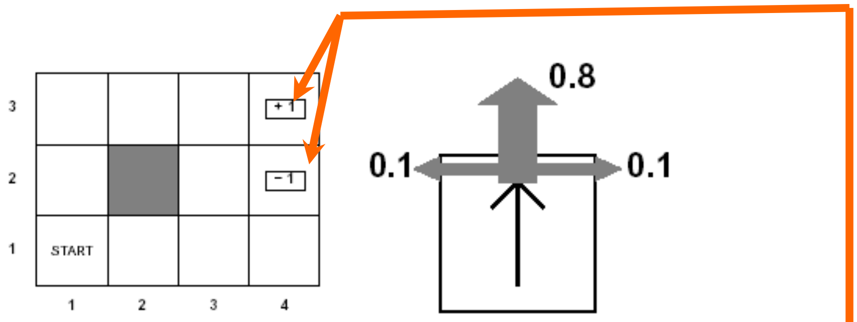
33

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, North takes the agent North (if there is no wall there)
  - 10% of the time, North → West; 10% East
  - If there is a wall in the direction the agent would have gone, the agent stays put
- The agent receives rewards each time step
  - Small “living” reward r each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of rewards

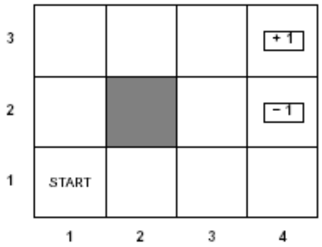


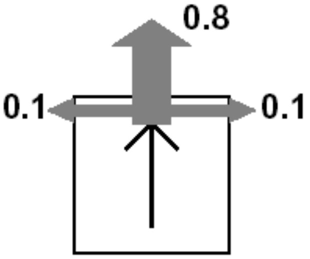
Example



$$R(s) = \begin{cases} -0.04 & \text{(small penalty) for nonterminal states} \\ \pm 1 & \text{for terminal states} \end{cases}$$

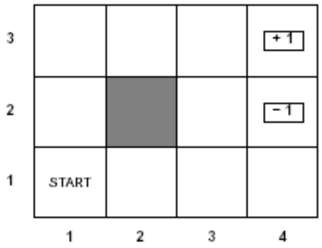
Example

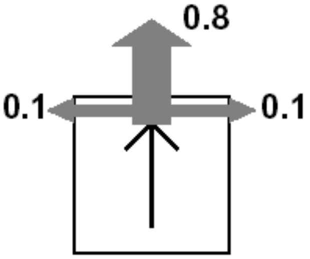




- Can the sequence [*Up, Up, Right, Right, Right*] take the agent to terminal state (4,3)?
- Can the sequence reach the goal in any other way?

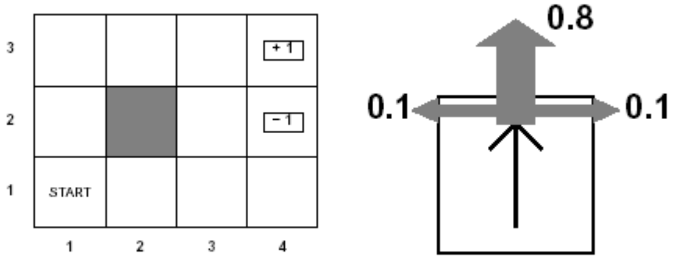
Example





- Can the sequence [*Up, Up, Right, Right, Right*] take the agent to terminal state (4,3)?
  - Yes, with probability 0.8^5=0.3278
- Can the sequence reach the goal in any other way?

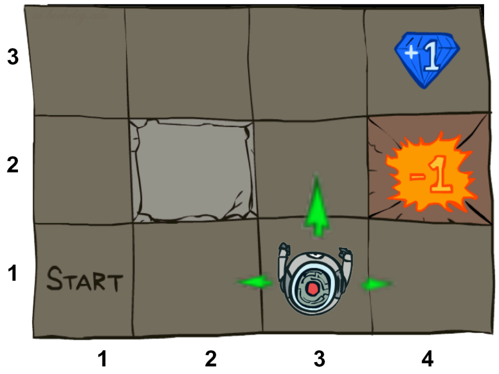
Example

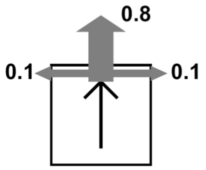


- Can the sequence [*Up, Up, Right, Right, Right*] take the agent to terminal state (4,3)?
  - Yes, with probability 0.8^5=0.3278
- Can the sequence reach the goal in any other way?
  - yes, going the other way around with probability 0.1^4x0.8 = 0.00008

Markov Decision Processes

- An MDP is defined by:
  - A **set of states** s ∈ S
  - A **set of actions** a ∈ 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"
  - A **reward function** R(s, a, s')
    - Sometimes just R(s) or R(s')
  - A **start state** (or distribution)
  - Maybe a **terminal state(s)**





Transition Model

- A transition model is a specification of the outcome probabilities for each action in each possible state.
- T(s,a,s') denotes the probability of reaching state s' if action a is done on state s.
- Make Markov Assumption, i.e., the probability of reaching state s' from s depends only on s and not on the history of earlier states.

40

Rewards and Utilities

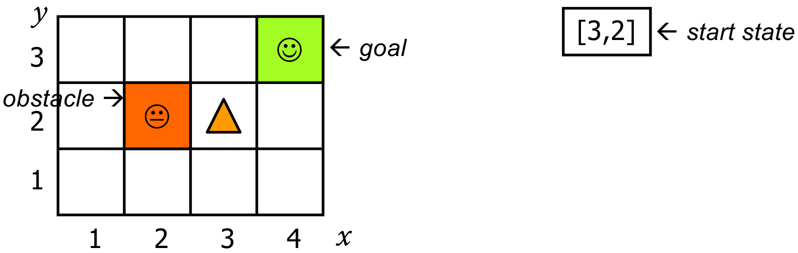
- A utility function must be specified for the agent in order to determined the value of an action.
- Because the problem is sequential, the utility function depends on a **sequence of states** (environment history).
- Rewards are assigned to states, i.e., R(s) returns the reward of the state.
- For this example, assume the following:
  - The reward for all states, except for the goal states, is -0.04.
  - The utility function is the sum of all the states visited.
    - E.g., if the agent reaches (4,3) in 10 steps, the total utility is 1 + (10 x -0.04) = 0.6.
  - The negative reward is an incentive to stop interacting as quickly as possible.

41

Markov Property

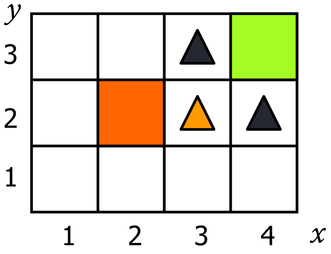
- We will focus on decision processes that can be represented as Markovians (as in Markov models)
  - Actions have probabilistic outcomes that depend only on the current state
  - Let s_t be the state at time t
  - $P(s_{t+1} | s_0 , a_0 ,\dots ,s_t , a_t) = P(s_{t+1} | s_t , a_t)$
- The transition properties 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).

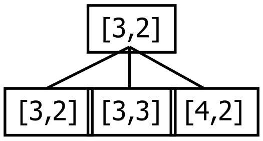
Sequence of Actions



- Planned sequence of actions: (U, R)

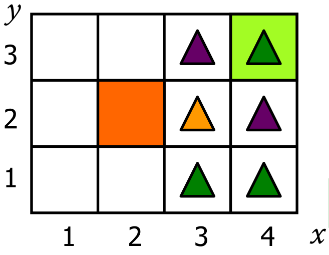
Sequence of Actions

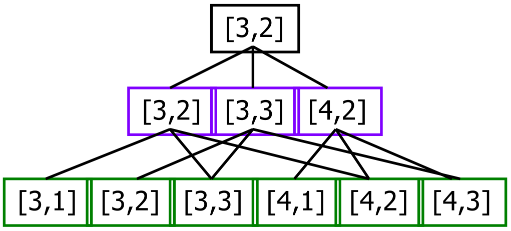




- Planned sequence of actions: (U, R)
- U is executed

Histories

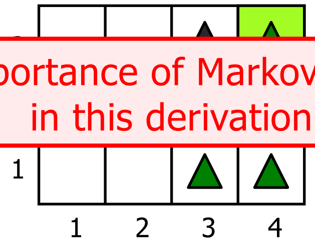




- Planned sequence of actions: (U, R)
- U has been executed
- R is executed

- 9 possible sequences of states – called histories
- 6 possible final states for the robot!

Probability of Reaching the Goal

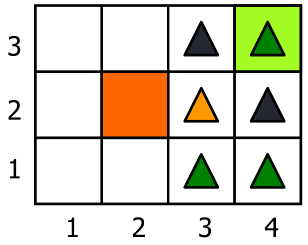


Note importance of Markov property in this derivation

- **P**([4,3] | (U,R).[3,2]) =
  P([4,3] | R.[3,3]) x **P**([3,3] | U.[3,2])
  + **P**([4,3] | R.[4,2]) x **P**([4,2] | U.[3,2])
- **P**([4,3] | R.[3,3]) = 0.8 •**P**([3,3] | U.[3,2]) = 0.8
- **P**([4,3] | R.[4,2]) = 0.1 •**P**([4,2] | U.[3,2]) = 0.1
- **P**([4,3] | (U,R).[3,2]) = 0.8 x 0.8 + 0.1 x 0.1 = 0.65

47

Probability of Reaching the Goal



- Core idea: multiply backward probabilities of each step taken from end state reached
- But we still need to consider different ways of reaching a state
  - Going all the way around the obstacle would be “worse”

48

Utility Function

3				+1
2		🟧		-1
1				
	1	2	3	4

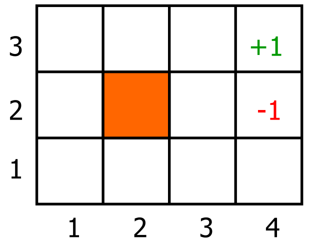
- [4,3] provides power supply
- [4,2] is a sand area from which the robot cannot escape

Utility Function

3				+1
2		🟧		-1
1				
	1	2	3	4

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

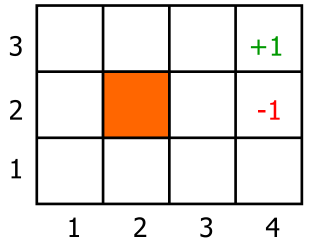
Utility Function



- [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

51

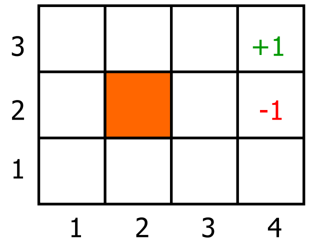
Utility Function



- [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!

52

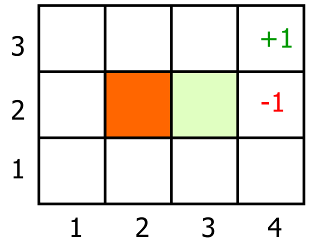
Utility of a History



- [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.

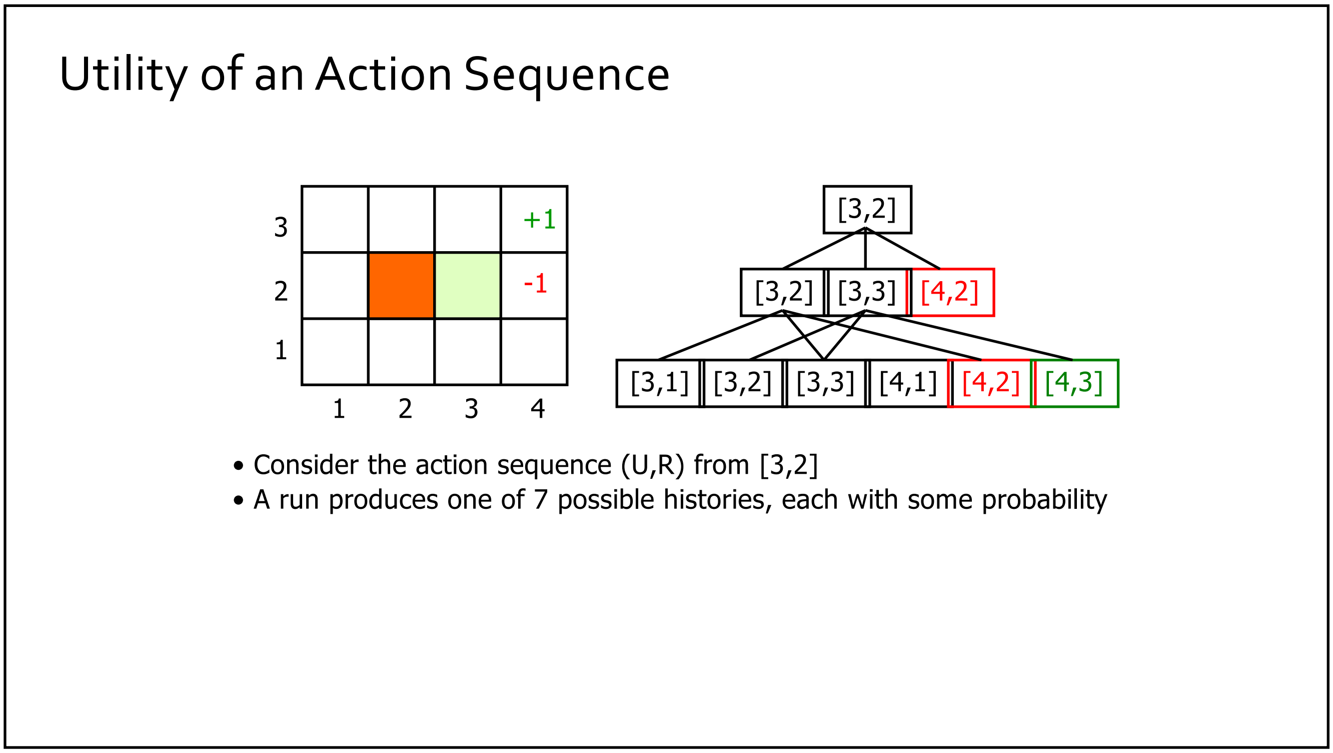
53

Utility of an Action Sequence



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

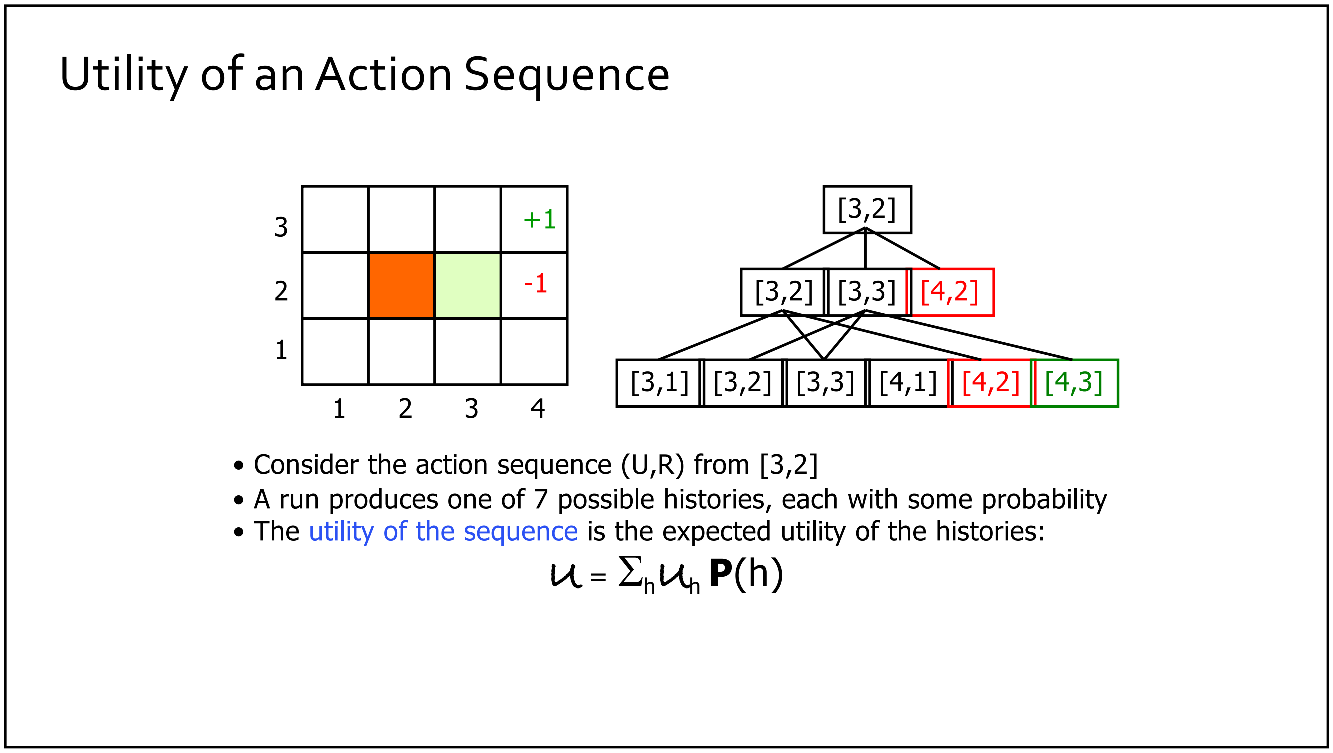
54



Utility of an Action Sequence

3				+1
2		🟧	🟩	-1
1				
	1	2	3	4

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



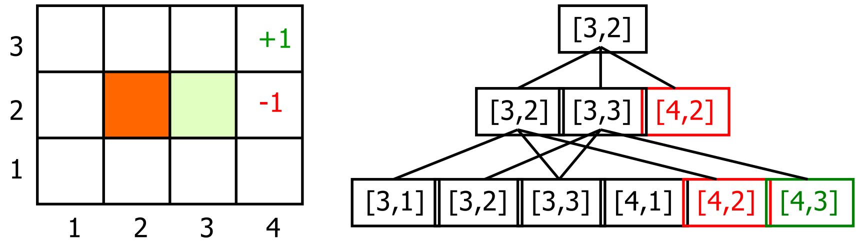
Utility of an Action Sequence

3				+1
2		🟧	🟩	-1
1				
	1	2	3	4

- 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



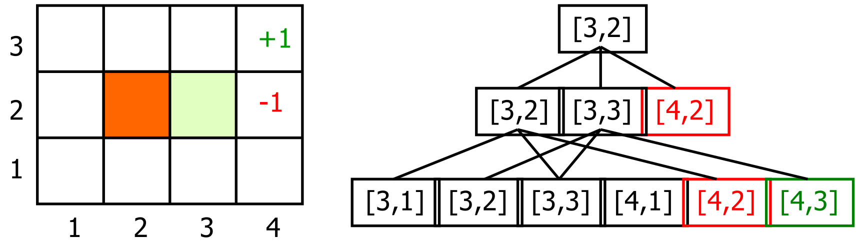
- Consider the action sequence (U,R) from [3,2]
- A run produces one of 7 possible histories, each with some probability
- The utility of the sequence is the expected utility of the histories:

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

- The optimal sequence is the one with maximal utility

57

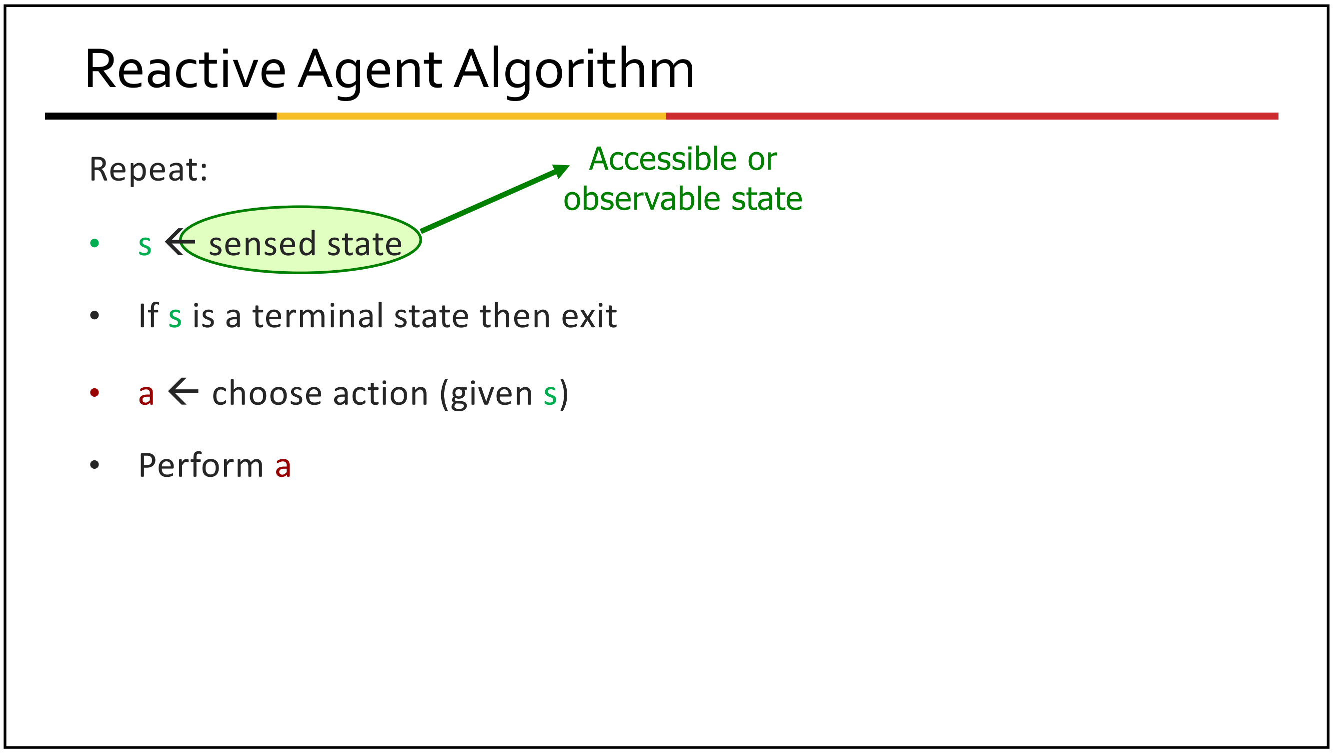
Optimal Action Sequence



- 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!

58



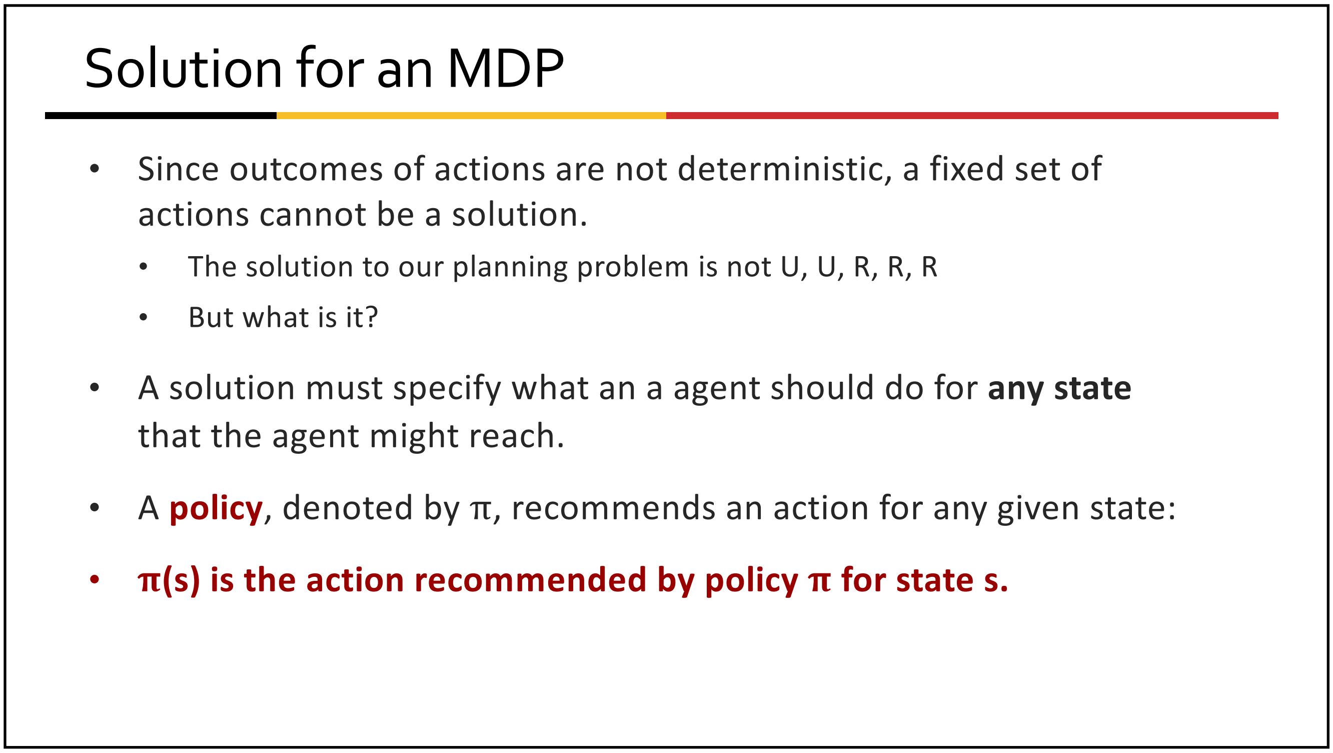
Reactive Agent Algorithm

Repeat:

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

Accessible or observable state

59



Solution for an MDP

- Since outcomes of actions are not deterministic, a fixed set of actions cannot be a solution.
  - The solution to our planning problem is not U, U, R, R, R
  - But what is it?
- A solution must specify what an a agent should do for **any state** that the agent might reach.
- A **policy**, denoted by π, recommends an action for any given state:
- **π(s) is the action recommended by policy π for state s.**

60

Policy (Reactive/Closed-Loop Strategy)

3	→	→	→	+1
2	↑	■	↑	-1
1	↑	←	←	←
	1	2	3	4

- 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

- An **optimal** policy is a policy that yields the highest expected utility.
- Optimal policy is denoted by π*.
- Once a π* is computed for a problem, then the agent, once identifying the state (s) that it is in, consults π*(s) for the next action to execute.

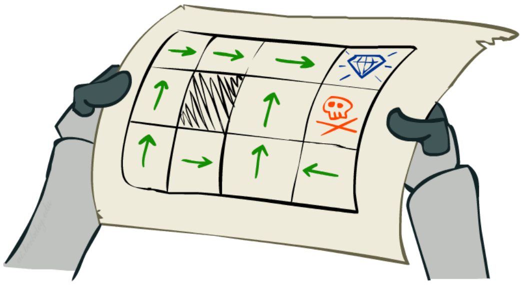
Reactive Agent Algorithm

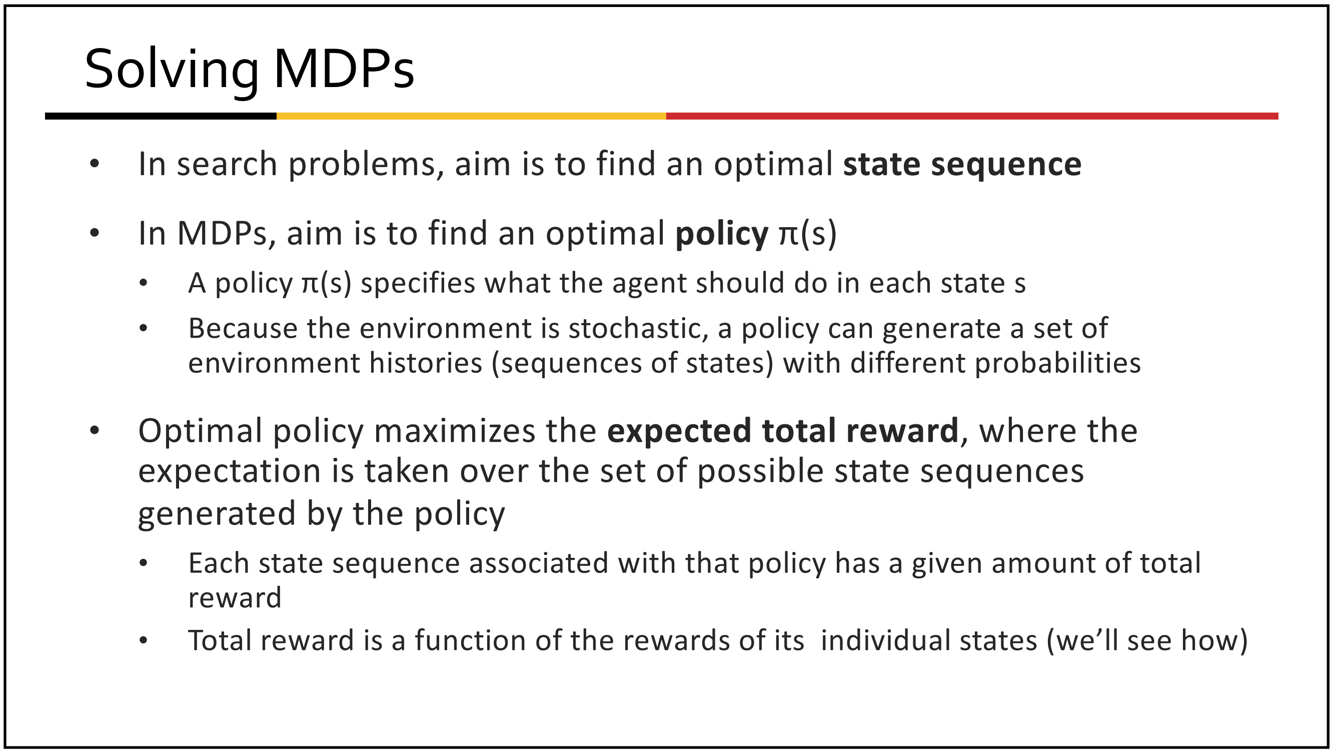
Repeat:

- s ← sensed state
- If s is terminal then exit
- a ← Π(s)
- Perform a

Policies

- A policy π gives an action for each state, $\pi: S \rightarrow A$
- 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*** $\pi^*: S \rightarrow A$
  - An optimal policy maximizes expected utility
  - An explicit policy defines a reflex agent

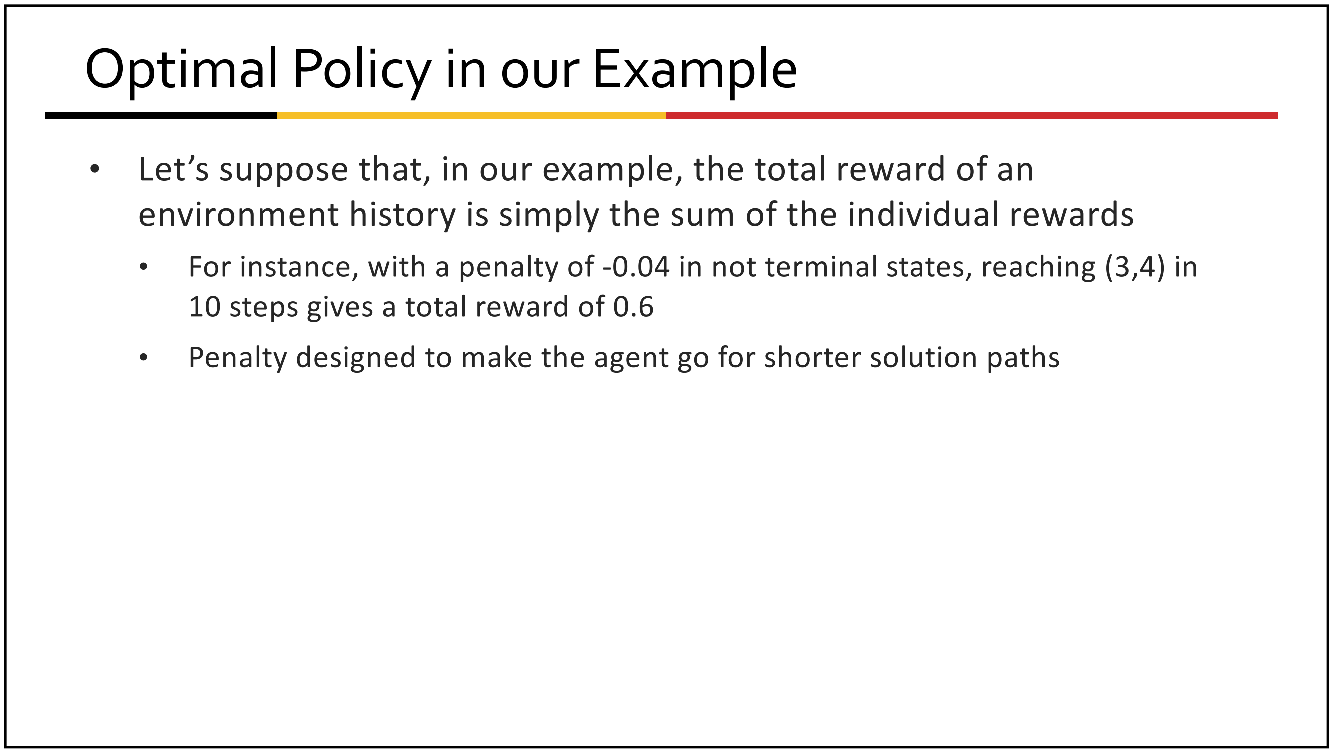




Solving MDPs

- In search problems, aim is to find an optimal **state sequence**
- In MDPs, aim is to find an optimal **policy** π(s)
  - A policy π(s) specifies what the agent should do in each state s
  - Because the environment is stochastic, a policy can generate a set of environment histories (sequences of states) with different probabilities
- Optimal policy maximizes the **expected total reward**, where the expectation is taken over the set of possible state sequences generated by the policy
  - Each state sequence associated with that policy has a given amount of total reward
  - Total reward is a function of the rewards of its individual states (we'll see how)

65

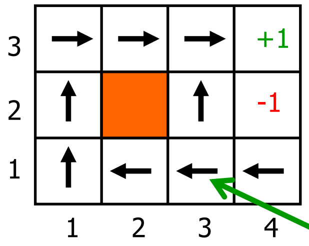


Optimal Policy in our Example

- Let's suppose that, in our example, the total reward of an environment history is simply the sum of the individual rewards
  - For instance, with a penalty of -0.04 in not terminal states, reaching (3,4) in 10 steps gives a total reward of 0.6
  - Penalty designed to make the agent go for shorter solution paths

66

Optimal Policy



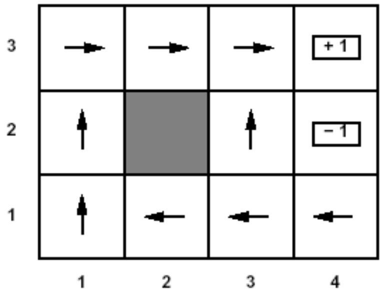
Note that [3,2] is a "dangerous" state that the optimal policy tries to avoid

- A policy π is a complet
- The optimal policy π* i
  history (sequence of s
  with maximal ***expected*** utility

67

Rewards and Optimal Policy

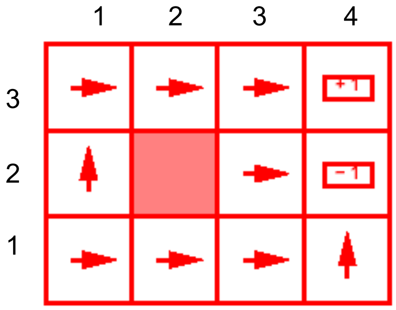
- Optimal Policy when penalty in non-terminal states is -0.04
- Note that here the cost of taking steps is small compared to the cost of ending into (4,2)
  - Thus, the optimal policy for state (3,1) is to take the long way around the obstacle rather then risking to fall into (4,2) by taking the shorter way that passes next to it
  - But the optimal policy may change if the reward in the non-terminal states (let's call it r) changes



68

Rewards and Optimal Policy

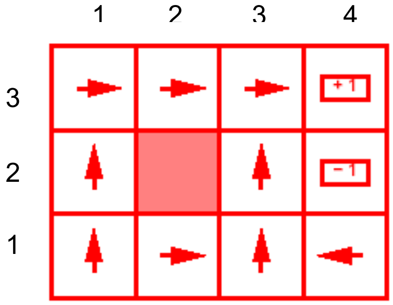
- Optimal Policy when r < -1.6284
- Why is the agent heading straight into (4,2) from its surrounding states?
- The cost of taking a step is so high that the agent heads straight into the nearest terminal state, even if this is (4,2) (reward -1)



r = [− ∞ : −1.6284]

Rewards and Optimal Policy

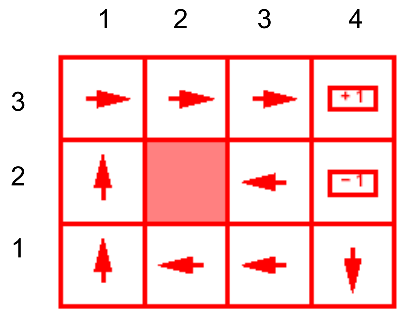
- Optimal Policy when -0.427 < r < -0.085
- The cost of taking a step is high enough to make the agent take the shortcut to (4,3) from (3,1)



r = [−0.4278 : −0.0850]

Rewards and Optimal Policy

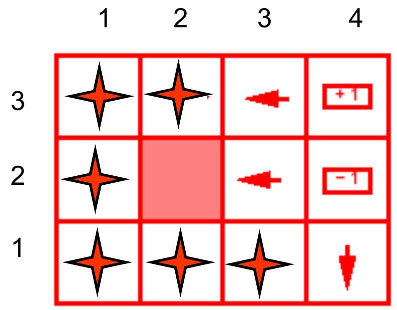
- Optimal Policy when -0.0218 < r < 0
- Why is the agent heading straight into the obstacle from (3,2)?
- Staying longer in the grid is not penalized as much as before. The agent is willing to take longer routes to avoid (4,2)
- This is true even when it means banging against the obstacle a few times when moving from (3,2)



r = [−0.0218 : 0.0000]

Rewards and Optimal Policy

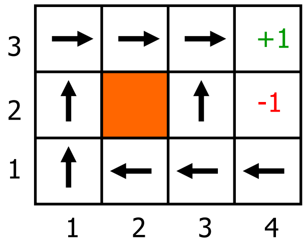
- Optimal Policy when r > 0
- What happens when the agent is rewarded for every step it takes?
- It is basically rewarded for sticking around
- The only actions that matter are the ones in states that are adjacent to the terminal states: take the agent away from them



r = [−0.0218 : 0.0000]

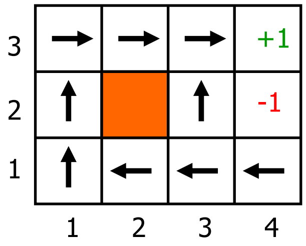
state where every action belongs to an optimal policy

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 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*?

Computing the optimal policy π*

- Additive utility
- State utilities
- Action sequences
- The Bellman equation
- Value iteration
- Policy iteration

75

Additive Utility

- History H = $(s_0,s_1,\ldots,s_n)$
- The utility of H is additive iff:

$$\mathcal{U}(s_0,s_1,\ldots,s_n) = \mathcal{R}(0) + u(s_1,\ldots,s_n) = \Sigma\ \mathcal{R}(i)$$

Reward

- The reward accumulates as you step through states.

76

Additive Utility

- History H = $(s_0,s_1,\ldots,s_n)$
- The utility of H is additive iff:

  $\mathcal{U}(s_0,s_1,\ldots,s_n) = \mathcal{R}(0) + \mathcal{U}(s_1,\ldots,s_n) = \Sigma\, \mathcal{R}(i)$
- Robot navigation example:
  - $\mathcal{R}(n) = +1$ if $s_n = [4,3]$
  - $\mathcal{R}(n) = -1$ if $s_n = [4,2]$
  - $\mathcal{R}(i) = -1/25$ if $i = 0, \ldots, n-1$

→	→	→	+1
↑	🟧	↑	-1
↑	←	←	←

Defining the optimal policy

- Given a policy π, we can define the **expected utility** over all possible state sequences produced by following that policy:

$$U^{\pi}(s_0) = \sum_{\substack{\text{state sequences}\\ \text{starting from } s_0}} P(\text{sequence})\,U(\text{sequence})$$

- The optimal policy should maximize this utility
- **But how to define the utility of a state sequence?**
  - Sum of rewards of individual states
  - Problem: infinite state sequences

Utilities of state sequences

- Normally, we would define the utility of a state sequence as the sum of the rewards of the individual states
- **Problem**: infinite state sequences
- **Solution**: discount the individual state rewards by a factor γ between 0 and 1:

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

$$= \sum_{t=0}^{\infty} \gamma^t R(s_t) \le \frac{R_{\max}}{1-\gamma} \qquad (0 < \gamma < 1)$$

- Sooner rewards "count" more than later rewards
- Makes sure the total utility stays bounded
- Helps algorithms converge

Sum of discounted rewards

- To define the utility of a state sequence, discount the individual state rewards by a factor γ between 0 and 1:

$$U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

1
Worth Now
γ
Worth Next Step
γ^2
Worth In Two Steps

- When γ = 1 this is just additive utility

Utilities of states

- Expected utility obtained by policy π starting in state s:

$$U^{\pi}(s) = \sum_{\substack{state\ sequences \\ starting\ from\ s}} P\big(sequence|s, a = \pi(s)\big)U(sequence)$$

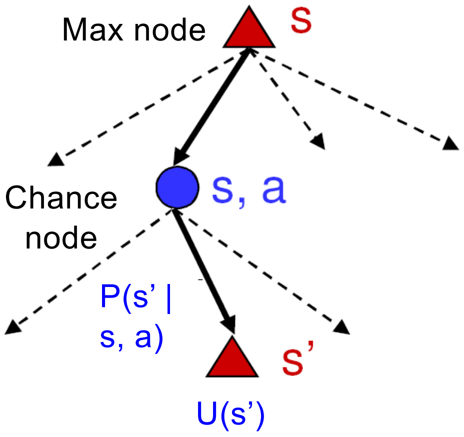
- The "true" utility of a state, denoted U(s), is the **best possible** expected sum of discounted rewards
  - if the agent executes the **best possible policy** starting in state s
- Reminiscent of minimax values of states

Defining State Utility

Problem:

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

Finding the utilities of states

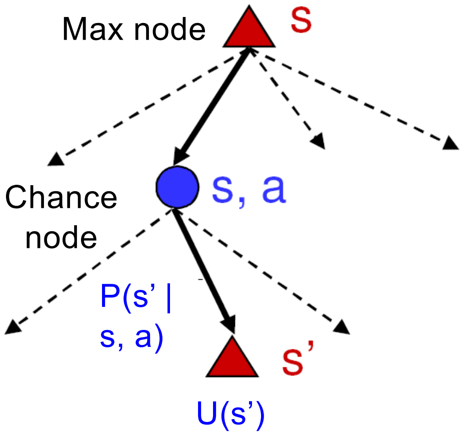


- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?
- How do we choose the optimal action?

- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

83

Finding the utilities of states



- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?

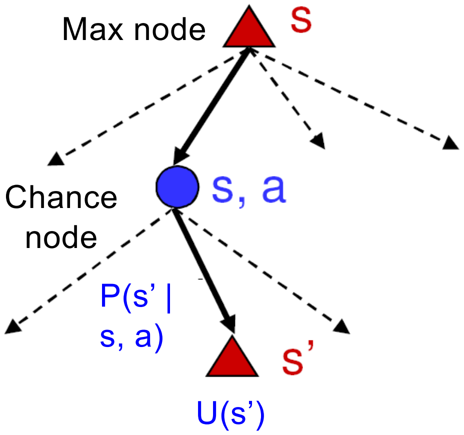
$$\sum_{s'} P(s'|s,a)U(s')$$

- How do we choose the optimal action?

- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

84

Finding the utilities of states



- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?

$$\sum_{s'} P(s'|s,a)U(s')$$

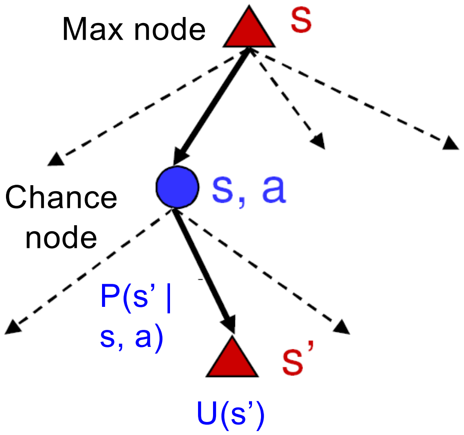
- How do we choose the optimal action?

$$\pi^*(s) = \arg\max_{a\in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

85

Finding the utilities of states



- If state s' has utility U(s'), then what is the expected utility of taking action **a** in state **s**?

$$\sum_{s'} P(s'|s,a)U(s')$$

- How do we choose the optimal action?

$$\pi^*(s) = \arg\max_{a\in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

- What is the recursive expression for **U(s)** in terms of the utilities of its successor states?

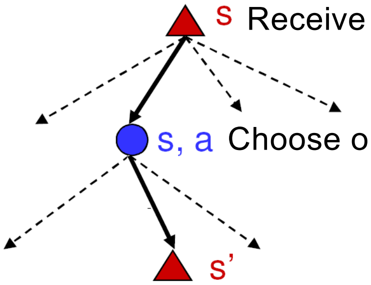
$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$$

86

The Bellman equation

- Recursive relationship between the utilities of successive states:

$$\boxed{U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')}$$



s Receive reward R(s)

s, a Choose optimal action a

s’

End up here with P(s’ | s, a)
Get utility U(s’)
(discounted by γ)

87

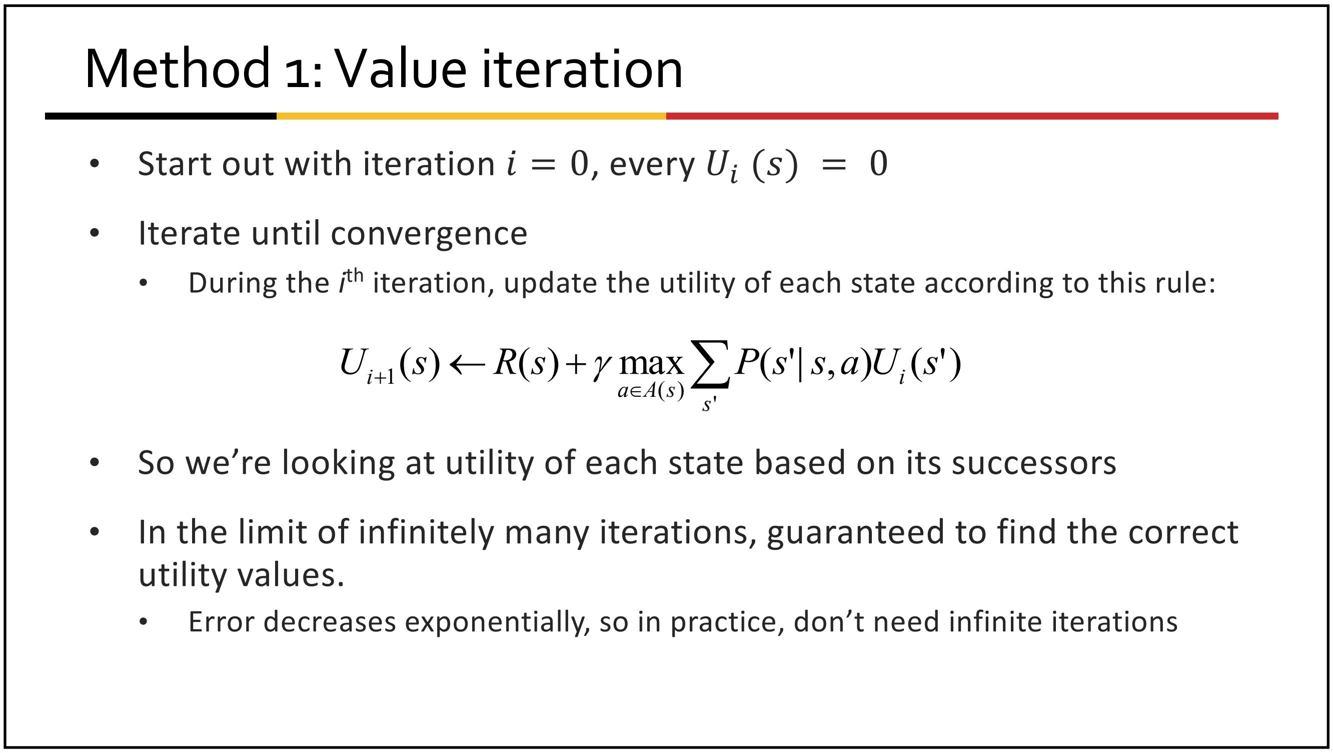
The Bellman equation

- Recursive relationship between the utilities of successive states:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

- For N states, we get N equations in N unknowns
    - Solving them solves the MDP
    - The “max” means that there is no closed-form solution. Need to use an iterative solution method, which might not converge to the globally optimum solution.
    - Two solution methods: value iteration and policy iteration

88



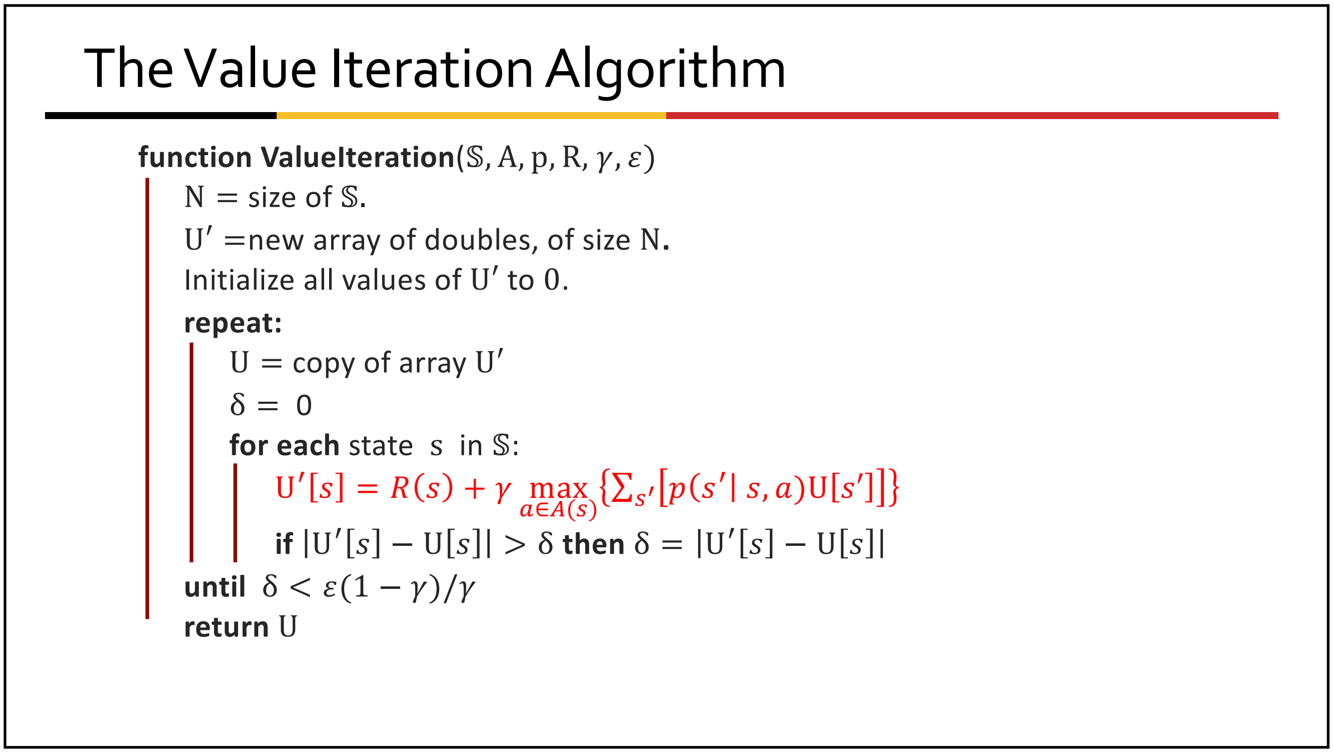
Method 1: Value iteration

- Start out with iteration $i = 0$, every $U_i\ (s)\ =\ 0$
- Iterate until convergence
  - During the i^{th} iteration, update the utility of each state according to this rule:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$

- So we're looking at utility of each state based on its successors
- In the limit of infinitely many iterations, guaranteed to find the correct utility values.
  - Error decreases exponentially, so in practice, don't need infinite iterations

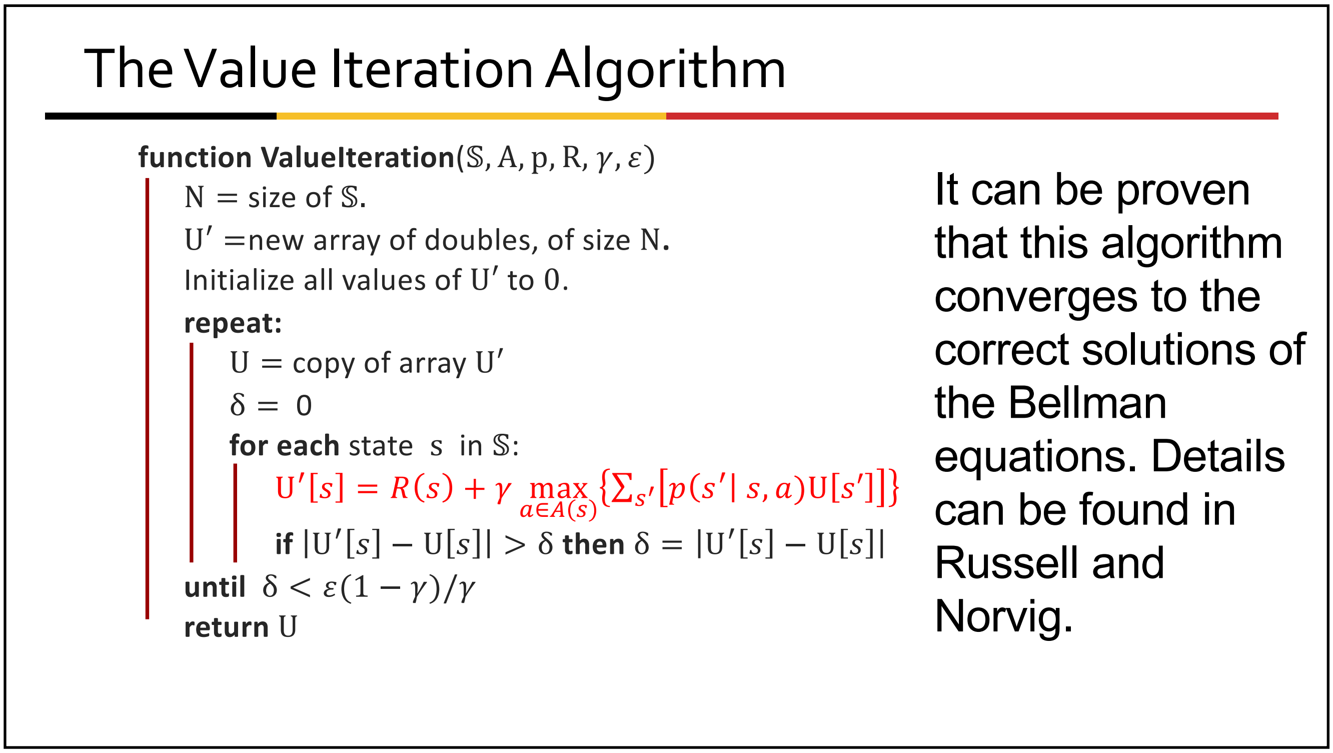
89



The Value Iteration Algorithm

```
function ValueIteration($\mathbb{S}$, $\mathrm{A}$, $\mathrm{p}$, $\mathrm{R}$, $\gamma$, $\varepsilon$)
    $\mathrm{N}$ = size of $\mathbb{S}$.
    $\mathrm{U}'$ =new array of doubles, of size $\mathrm{N}$.
    Initialize all values of $\mathrm{U}'$ to $0$.
    repeat:
        $\mathrm{U}$ = copy of array $\mathrm{U}'$
        $\delta = 0$
        for each state $s$ in $\mathbb{S}$:
            $\mathrm{U}'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' \mid s, a)\mathrm{U}[s']]\}$
            if $|\mathrm{U}'[s] - \mathrm{U}[s]| > \delta$ then $\delta = |\mathrm{U}'[s] - \mathrm{U}[s]|$
    until $\delta < \varepsilon(1-\gamma)/\gamma$
    return $\mathrm{U}$
```

90

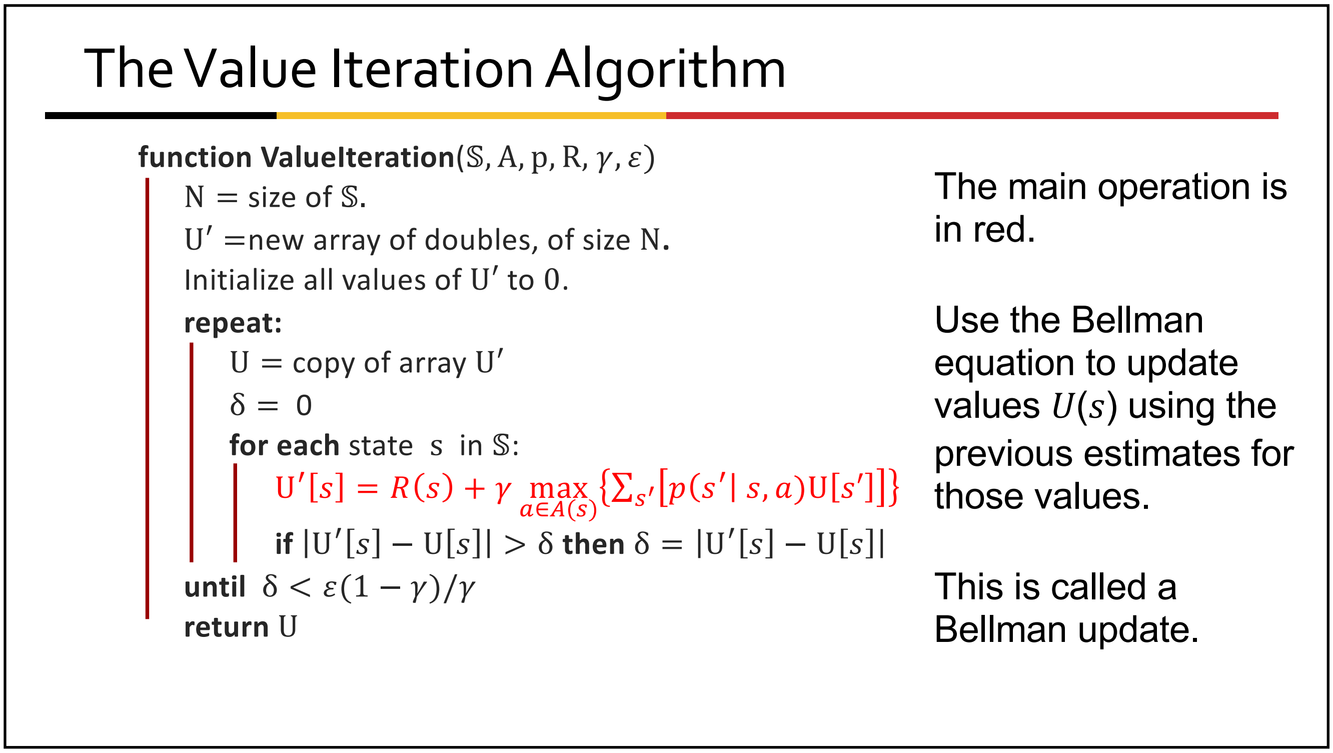


The Value Iteration Algorithm

```
function ValueIteration($\mathbb{S}$, A, p, R, $\gamma$, $\varepsilon$)
    N = size of $\mathbb{S}$.
    U' =new array of doubles, of size N.
    Initialize all values of U' to 0.
    repeat:
        U = copy of array U'
        $\delta = 0$
        for each state s in $\mathbb{S}$:
            $U'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' | s, a) U[s']]\}$
            if $|U'[s] - U[s]| > \delta$ then $\delta = |U'[s] - U[s]|$
    until $\delta < \varepsilon(1-\gamma)/\gamma$
    return U
```

It can be proven that this algorithm converges to the correct solutions of the Bellman equations. Details can be found in Russell and Norvig.

91



The Value Iteration Algorithm

```
function ValueIteration($\mathbb{S}$, A, p, R, $\gamma$, $\varepsilon$)
    N = size of $\mathbb{S}$.
    U' =new array of doubles, of size N.
    Initialize all values of U' to 0.
    repeat:
        U = copy of array U'
        $\delta = 0$
        for each state s in $\mathbb{S}$:
            $U'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' | s, a) U[s']]\}$
            if $|U'[s] - U[s]| > \delta$ then $\delta = |U'[s] - U[s]|$
    until $\delta < \varepsilon(1-\gamma)/\gamma$
    return U
```

The main operation is in red.

Use the Bellman equation to update values $U(s)$ using the previous estimates for those values.

This is called a Bellman update.

92

The Value Iteration Algorithm

```
function ValueIteration($\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, \varepsilon$)
    $\mathrm{N}$ = size of $\mathbb{S}$.
    $\mathrm{U}'$ =new array of doubles, of size $\mathrm{N}$.
    Initialize all values of $\mathrm{U}'$ to $0$.
    repeat:
        $\mathrm{U}$ = copy of array $\mathrm{U}'$
        $\delta = 0$
        for each state $s$ in $\mathbb{S}$:
            $\mathrm{U}'[s] = R(s) + \gamma \max_{a \in A(s)} \{\sum_{s'} [p(s' | s, a) \mathrm{U}[s']]\}$
            if $|\mathrm{U}'[s] - \mathrm{U}[s]| > \delta$ then $\delta = |\mathrm{U}'[s] - \mathrm{U}[s]|$
    until $\delta < \varepsilon(1-\gamma)/\gamma$
    return $\mathrm{U}$
```

So, the value iteration algorithm is:

Initialize utilities of states to zero values.

Repeatedly update utilities of states using Bellman updates, until the estimated values converge.

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- We initialize all utility values to 0.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0	0	0	0
2	0	▨	0	0
1	0	0	0	0
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after one round of updates:
  - The current estimate for each state s is $R(s)$.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	-0.04	-0.04	-0.04	+1
2	-0.04	▨	-0.04	-1
1	-0.04	-0.04	-0.04	-0.04
	1	2	3	4

Utility Values

95

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after two rounds of updates:
  - Information about the +1 reward reached state (3,3).

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	-0.08	-0.08	0.67	+1
2	-0.08	▨	-0.08	-1
1	-0.08	-0.08	-0.08	-0.08
	1	2	3	4

Utility Values

96

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after three rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	-0.11	0.43	0.73	+1
2	-0.11	▨	0.35	-1
1	-0.11	-0.11	-0.11	-0.11
	1	2	3	4

Utility Values

97

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after four rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.25	0.57	0.78	+1
2	-0.14	▨	0.43	-1
1	-0.14	-0.14	0.19	-0.14
	1	2	3	4

Utility Values

98

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after five rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2		■		-1
1	START			
	1	2	3	4

3	0.38	0.62	0.79	+1
2	0.12	■	0.47	-1
1	-0.16	0.07	0.24	-0.01
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after six rounds of updates:
  - Information about the +1 reward has reached all states.

3				+1
2		■		-1
1	START			
	1	2	3	4

3	0.45	0.64	0.79	+1
2	0.25	■	0.48	-1
1	0.04	0.15	0.30	0.05
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after seven rounds of updates:
  - Values keep getting updated.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.48	0.65	0.79	+1
2	0.33	▨	0.48	-1
1	0.16	0.21	0.32	0.09
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after eight rounds of updates:
  - Values continue changing.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.50	0.65	0.80	+1
2	0.37	▨	0.49	-1
1	0.23	0.23	0.34	0.11
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
    - $R(s) = -0.04$ if s is a non-terminal state.
    - $\gamma = 0.9$
- This is the result after 13 rounds of updates:
    - Values don't change much anymore after this round.

3				+1
2		▨		-1
1	START			
	1	2	3	4

3	0.51	0.65	0.80	+1
2	0.40	▨	0.49	-1
1	0.30	0.25	0.34	0.13
	1	2	3	4

Utility Values

Computing the Optimal Policy

- The value iteration algorithm computes $\mathrm{U}(s)$ for every state s.
- Once we have computed all values $\mathrm{U}(s)$, we can get the optimal policy π^* using this equation:
- $\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \{\sum_{s'}[p(s' \mid s, a) U(s')]\}$
- Thus, $\pi^*(s)$ identifies the action that leads to the highest expected utility for the next state, as measured over all possible outcomes of that action.
- This approach is called one-step look-ahead.

Approach 2: Policy Iteration

- There is a more efficient algorithm for computing optimal policies
- Remember that, if we know the utility of each state, we can compute the optimal policy π^* using:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ \sum_{s'} [p(s' \mid s, a) U(s')] \right\}$$

- However, to get the right $\pi^*(s)$, we don't need to know the utilities very accurately.
- We just need to know the utilities accurately enough so that, for each state s, argmax chooses the right action.

105

Method 2: Policy Iteration

- Start with some initial policy π_0 and alternate between the following steps:
  - **Policy Evaluation:** calculate the utility of every state under the assumption that the given policy is fixed and unchanging.
  - **Policy Improvement:** calculate a new policy π_{i+1} based on the updated utilities.
- Kind of like gradient descent:
  - Policy evaluation: Find ways in which the current policy is suboptimal
  - Policy improvement: Fix those problems
- Unlike Value Iteration, this is guaranteed to converge in a finite number of steps, as long as the state space and action set are both finite.

106

The Policy Iteration Algorithm

- This alternative algorithm for computing optimal policies is called the **policy iteration algorithm**.
- It is an iterative algorithm.
- Initialization:
  - Initiate some policy π_0 with random choices for the best action at each state.
- Main loop:
  - **Policy evaluation**: given the current policy π_i, calculate utility values $U^{\pi_i}(\mathrm{s})$, corresponding to the utility of each state s if the agent follows policy $\boldsymbol{\pi_i}$.
  - **Policy improvement**: Given current utility values $U^{\pi_i}(\mathrm{s})$, use one-step look-ahead to compute new policy π_{i+1}.

107

The Policy Evaluation Step

- Task: calculate utility values $U^{\pi_i}(\mathrm{s})$, corresponding to the assumption that the agent follows policy $\boldsymbol{\pi_i}$.
- When the policy was not known, we used the Bellman equation:

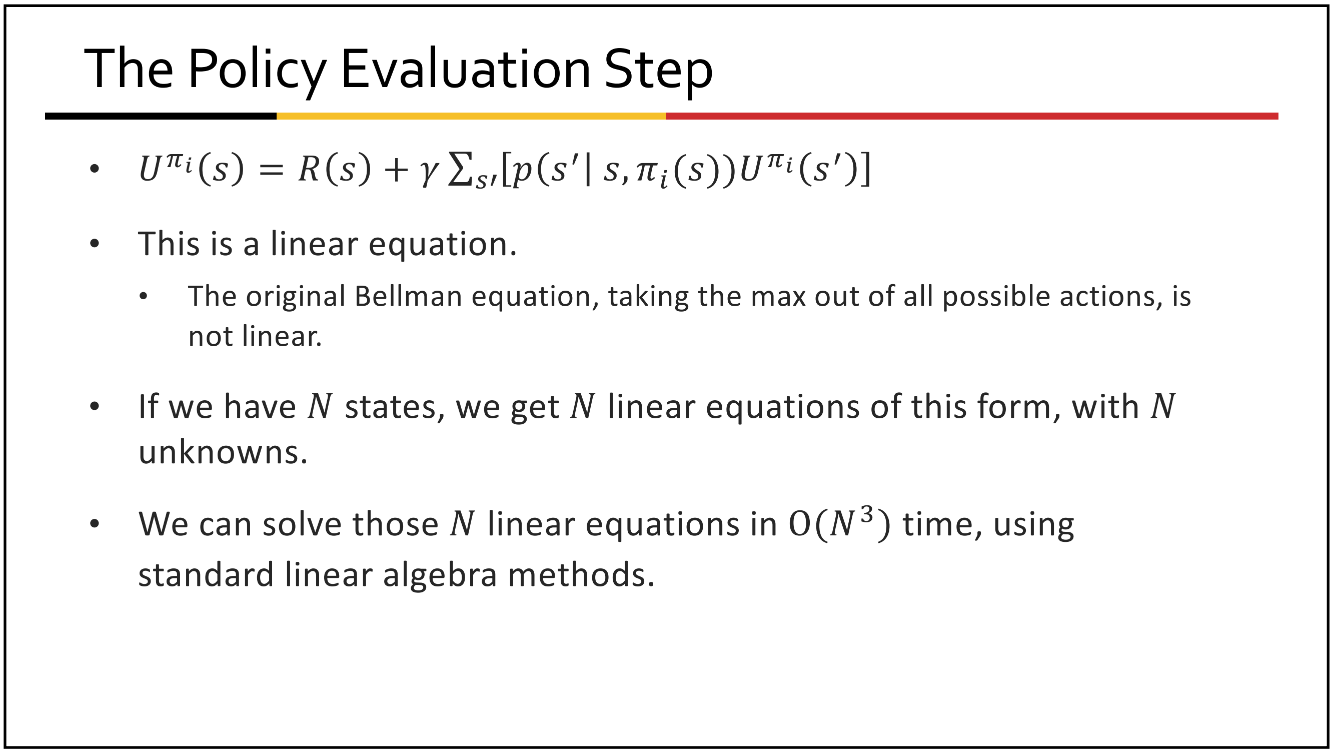
$$\mathrm{U}(s) = R(s) + \gamma \max_{a \in A(s)} \left\{ \sum_{s'} [p(s' | \, s, a) U(s')] \right\}$$

- Now that the policy π_i is specified, we can instead use a simplified version of the Bellman equation:

$$U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} [p(s' | \, s, \pi_i(s)) U^{\pi_i}(s')]$$

- Key difference: now $\pi_i(s)$ specifies the action for each state s, so we do not need to look for the max over all possible actions.

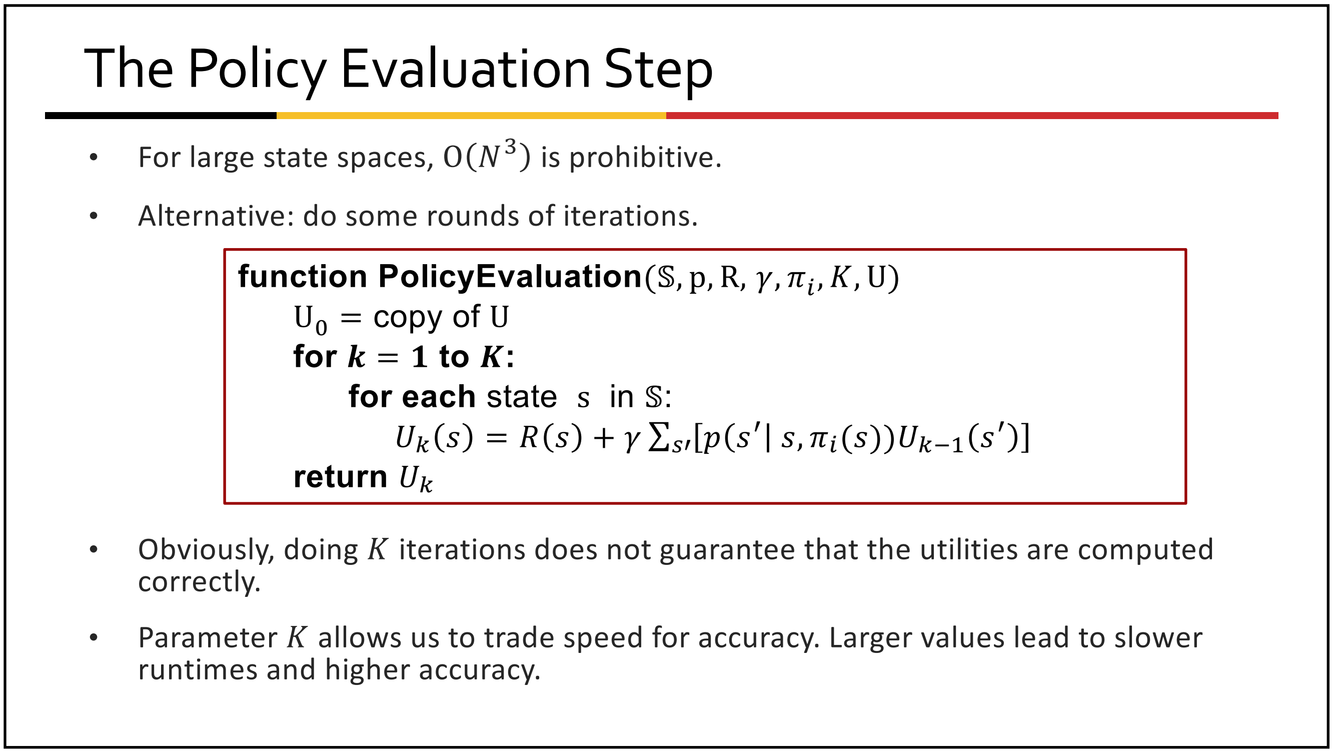
108



The Policy Evaluation Step

- $U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} [p(s' \mid s, \pi_i(s)) U^{\pi_i}(s')]$
- This is a linear equation.
  - The original Bellman equation, taking the max out of all possible actions, is not linear.
- If we have N states, we get N linear equations of this form, with N unknowns.
- We can solve those N linear equations in $O(N^3)$ time, using standard linear algebra methods.

109



The Policy Evaluation Step

- For large state spaces, $O(N^3)$ is prohibitive.
- Alternative: do some rounds of iterations.

```
function PolicyEvaluation($\mathbb{S}$, $\mathrm{p}$, $\mathrm{R}$, $\gamma$, $\pi_i$, $K$, $\mathrm{U}$)
    $\mathrm{U}_0$ = copy of $\mathrm{U}$
    for $k = 1$ to $K$:
        for each state $s$ in $\mathbb{S}$:
            $U_k(s) = R(s) + \gamma \sum_{s'} [p(s' \mid s, \pi_i(s)) U_{k-1}(s')]$
    return $U_k$
```

- Obviously, doing K iterations does not guarantee that the utilities are computed correctly.
- Parameter K allows us to trade speed for accuracy. Larger values lead to slower runtimes and higher accuracy.

110

The Policy Evaluation Step

- For large state spaces, $\mathrm{O}(N^3)$ is prohibitive.
- Alternative: do some rounds of iterations.

```
function PolicyEvaluation($\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi_i, K, \mathrm{U}$)
    $\mathrm{U}_0$ = copy of $\mathrm{U}$
    for $\boldsymbol{k} = \mathbf{1}$ to $\boldsymbol{K}$:
        for each state $s$ in $\mathbb{S}$:
            $U_k(s) = R(s) + \gamma \sum_{s'}[p(s' \mid s, \pi_i(s)) U_{k-1}(s')]$
    return $U_k$
```

- The PolicyEvaluation function takes as argument a current estimate U.

111

Policy Iteration

- Pick a policy π at random
- Repeat:
  - Compute the utility of each state for π

    $\mathcal{U}_{t+1}(i) \leftarrow \mathcal{R}(i) + \sum_k \mathcal{P}(k \mid \pi(i).i)\ \mathcal{U}_t(k)$
  - Compute the policy π’ given these utilities

    $\pi'(i) = \arg\max_a \sum_k \mathcal{P}(k \mid a.i)\ \mathcal{U}(k)$
  - If π’ = π then return π

112

Policy Iteration: Convergence

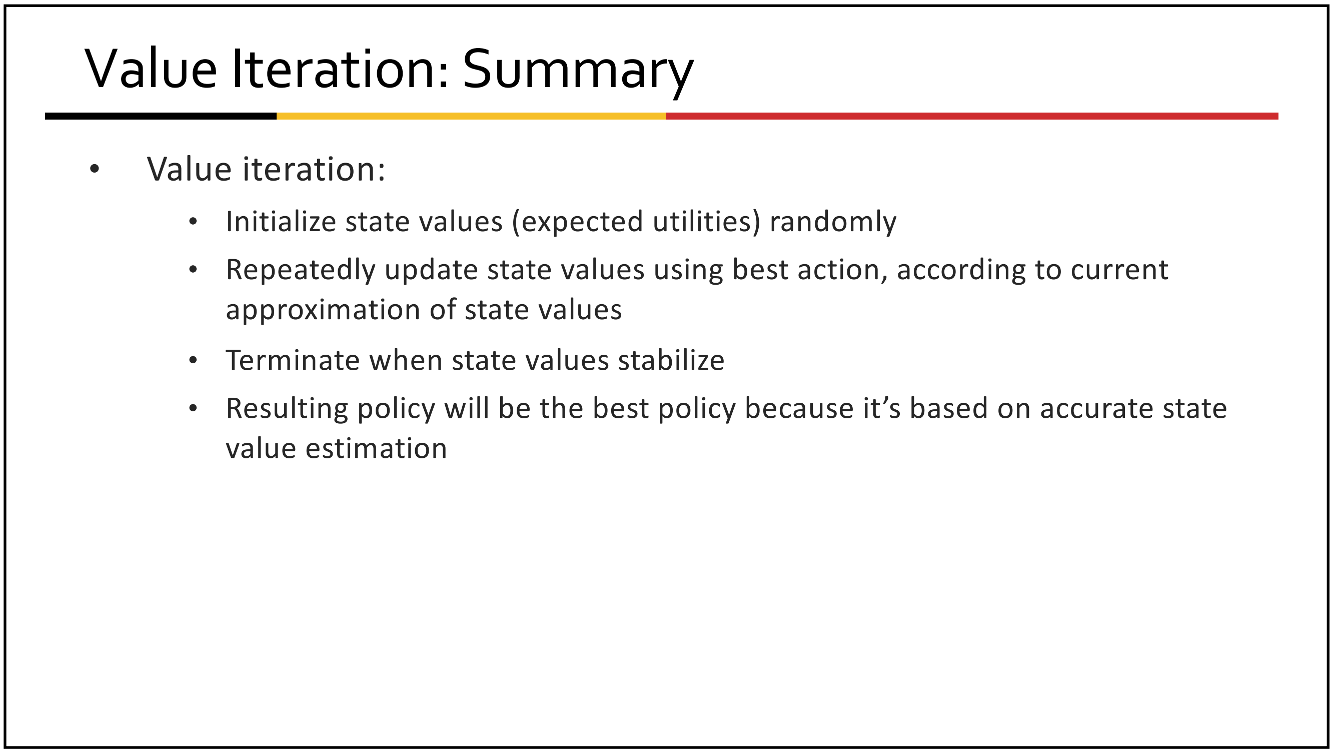
- Convergence assured in a finite number of iterations
  - Since finite number of policies and each step improves value, then must converge to optimal
- Gives exact value of optimal policy

113

Policy Iteration Complexity

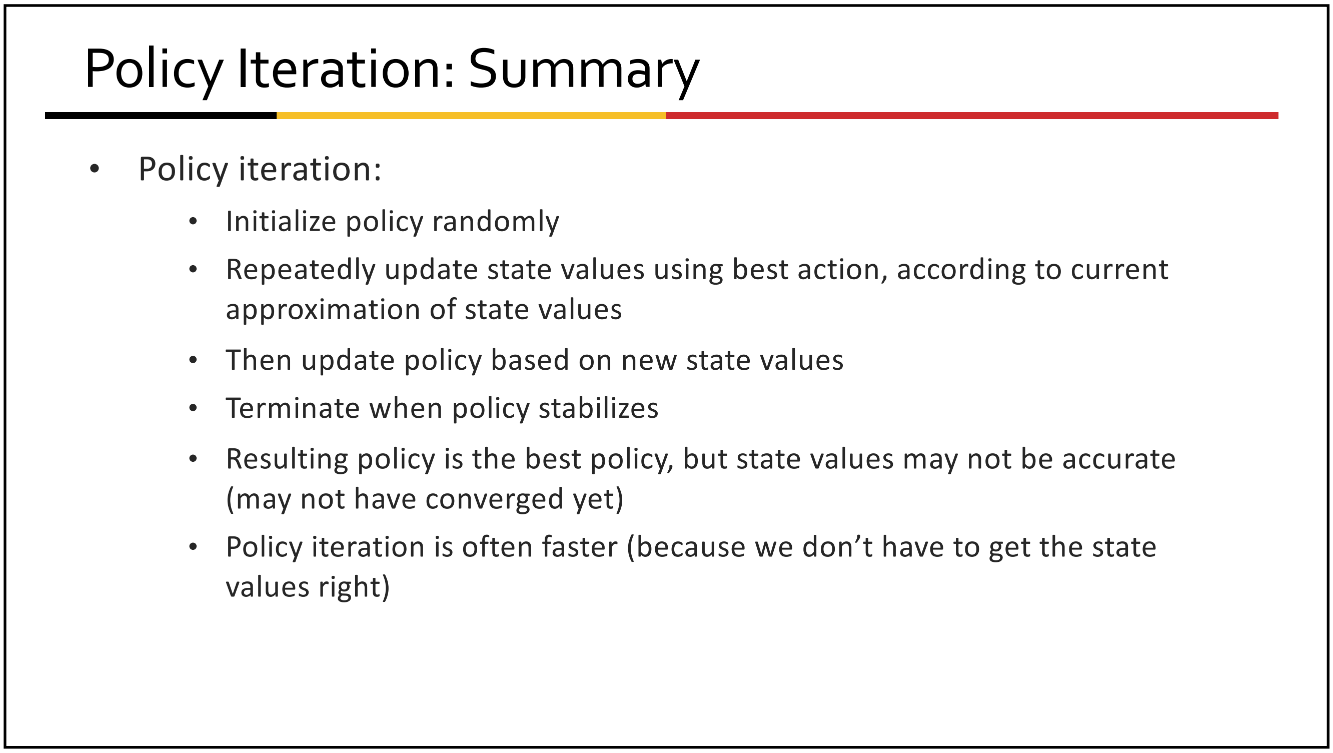
- Each iteration runs in polynomial time in the number of states and actions
- There are at most |A|n policies and PI never repeats a policy
  - So at most an exponential number of iterations
  - Not a very good complexity bound
- Empirically O(n) iterations are required – often it seems like O(1)
- Recent polynomial bounds.

114



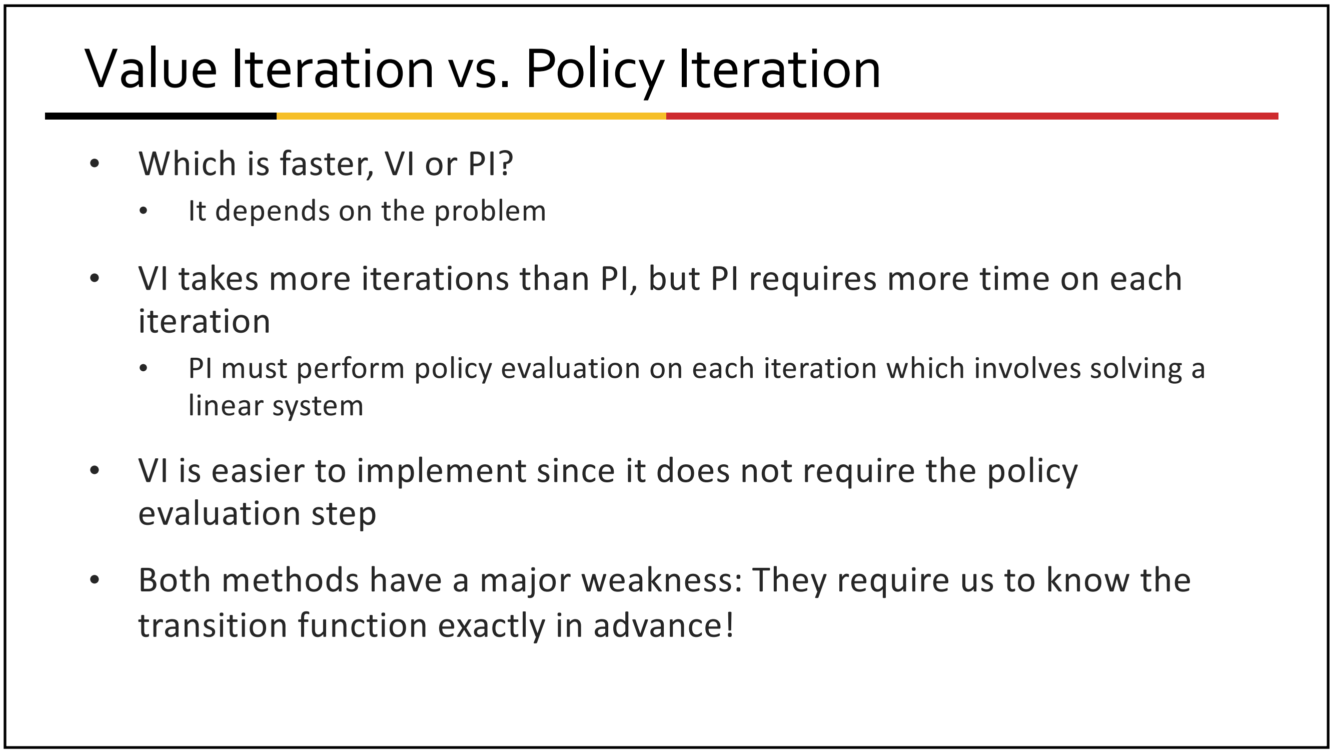
Value Iteration: Summary

- Value iteration:
  - 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

- Policy iteration:
  - 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)



Value Iteration vs. Policy Iteration

- Which is faster, VI or PI?
  - It depends on the problem
- VI takes more iterations than PI, but PI requires more time on each iteration
  - PI must perform policy evaluation on each iteration which involves solving a linear system
- VI is easier to implement since it does not require the policy evaluation step
- Both methods have a major weakness: They require us to know the transition function exactly in advance!