Dynamic Programming

Reading

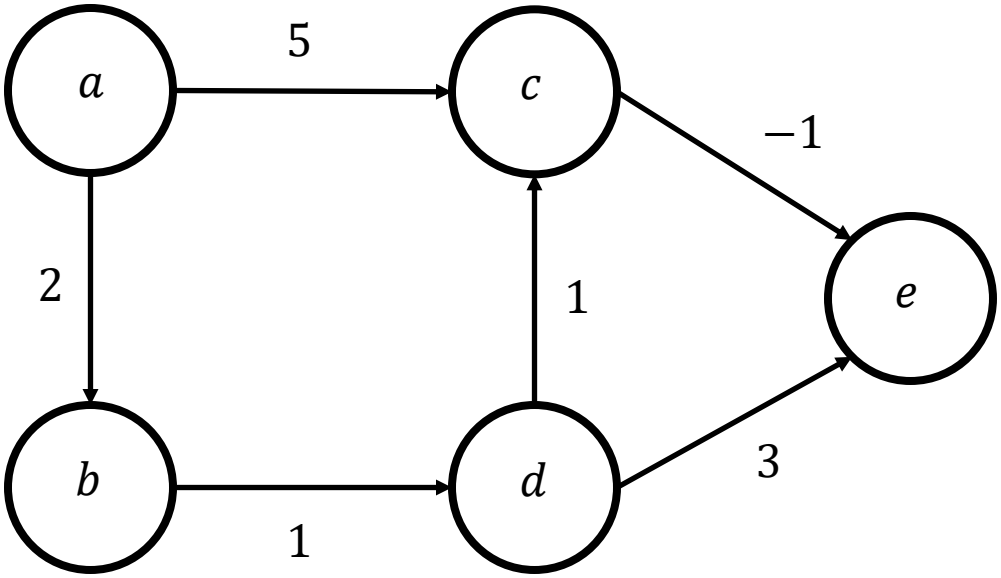
- Sutton, Richard S., and Barto, Andrew G. Reinforcement learning: An introduction. MIT press, 2018.
  - http://www.incompleteideas.net/book/the-book-2nd.html
  - Chapter 4
- Puterman, Martin L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
  - Chapters 4-6
- David Silver lecture on Dynamic Programming
  - https://www.youtube.com/watch?v=Nd1-UUMVfz4

Overview of Dynamical Programming

- A classical algorithm for computing solutions to problems that can be separated into subproblems with known solutions
  - E.g., all shortest paths
  - Essentially, store all subproblem solutions in a table and reuse them when necessary
- If we have a finite (state and action) MDP, we can find the optimal policy by incremental search
  - For example, in a finite-horizon setting:
    - Find the optimal policy for 1 step, then 2 steps, etc.
    - Actually done backwards in time
- Polynomial complexity in the number of states
  - Number of states can be large of course

Dynamic Programming: Shortest Paths

- A famous application of dynamic programming
  - Compute shortest paths from a node to all other nodes in a graph
  - E.g., all shortest paths from a to other nodes

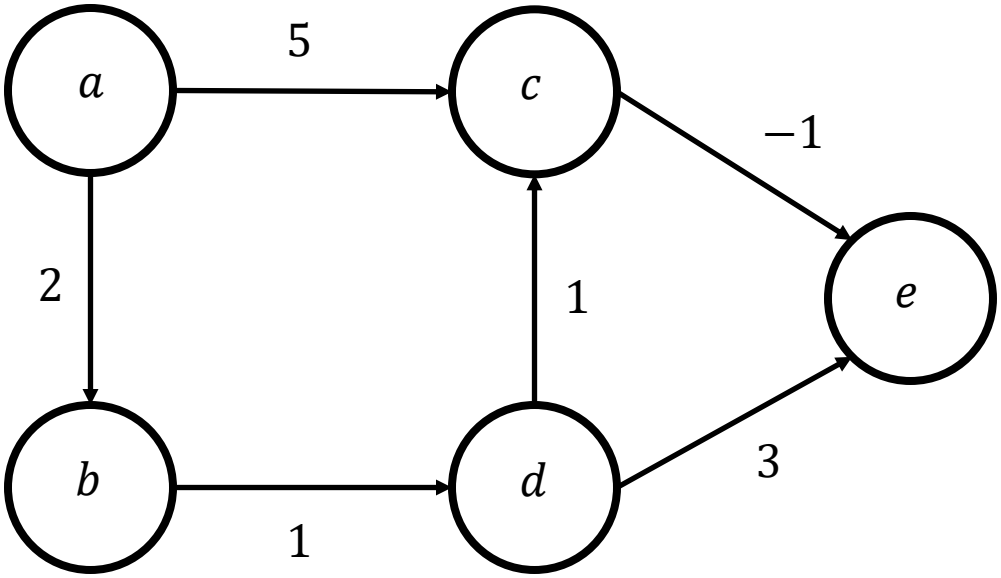


1 step

a	0		
b	2		
c	5		
d	∞		
e	∞		

Dynamic Programming: Shortest Paths

- A famous application of dynamic programming
  - Compute shortest paths from a node to all other nodes in a graph
  - E.g., all shortest paths from a to other nodes

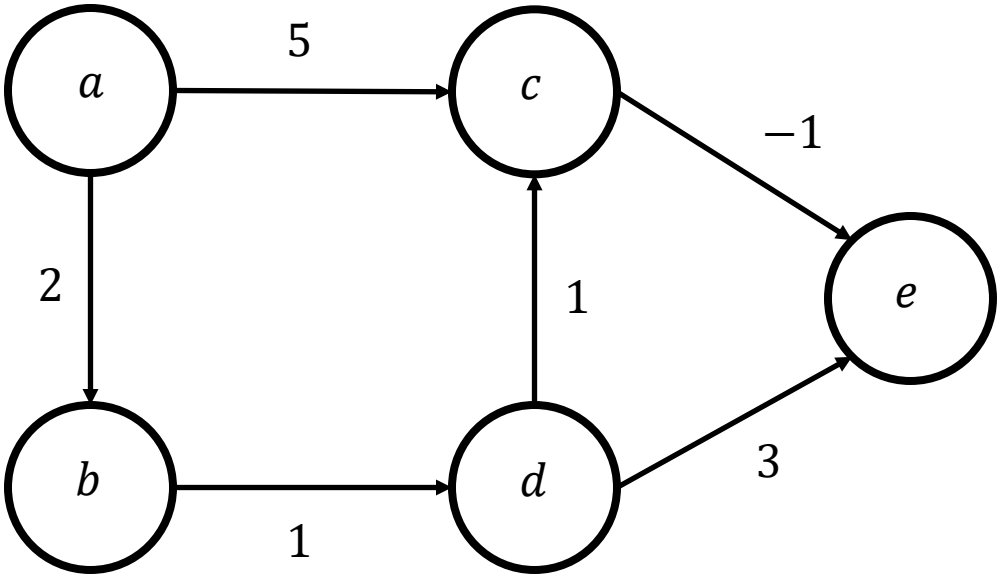


	1 step	2 steps	
a	0	0	
b	2	2	
c	5	5	
d	∞	3	
e	∞	4	

Loop through all 1-step nodes and see if you can reach other nodes in 1 step at lower cost

Dynamic Programming: Shortest Paths

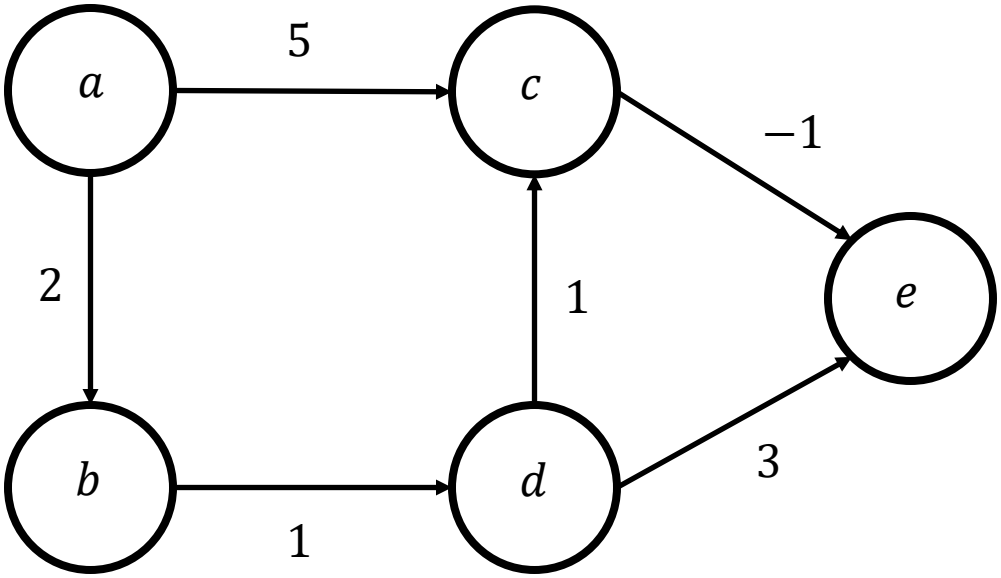
- A famous application of dynamic programming
  - Compute shortest paths from a node to all other nodes in a graph
  - E.g., all shortest paths from a to other nodes



	1 step	2 steps	3 steps
a	0	0	0
b	2	2	2
c	5	5	4
d	∞	3	3
e	∞	4	4

Dynamic Programming: Shortest Paths

- A famous application of dynamic programming
  - Compute shortest paths from a node to all other nodes in a graph
  - E.g., all shortest paths from a to other nodes



	1 step	2 steps	3 steps	4 steps
a	0	0	0	0
b	2	2	2	2
c	5	5	4	4
d	∞	3	3	3
e	∞	4	4	3

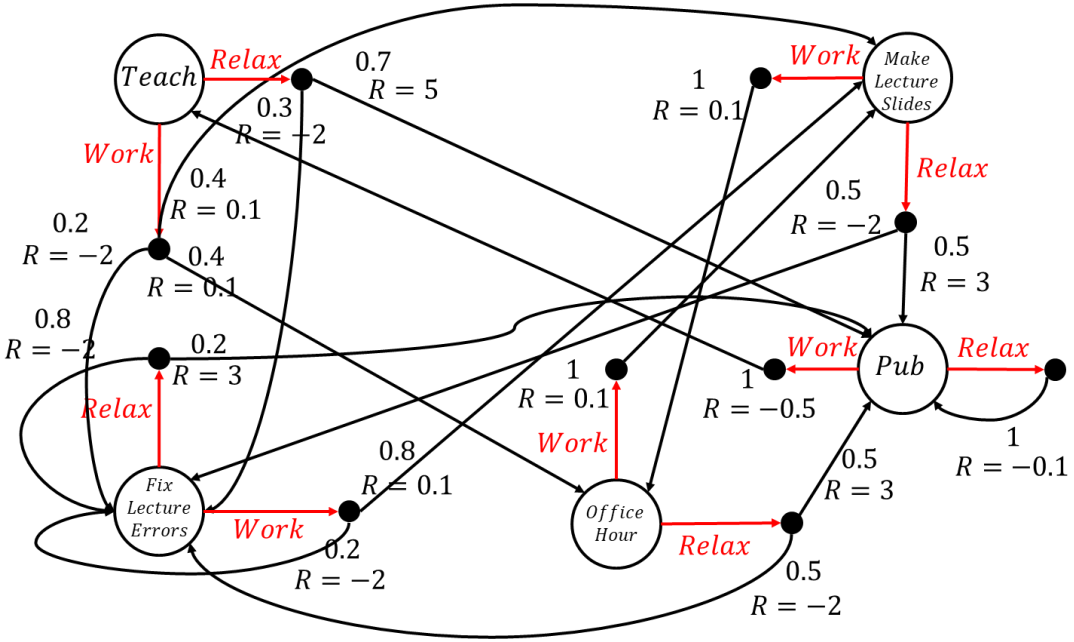
Cost to e through c is updated to 3

Dynamic Programming: Shortest Paths

- Repeat until no more improvements can be made
  - For each node n, go through all edges (n, n')
    - If $cost(a, n') > cost(a, n) + cost(n, n')$
    - Set $cost(a, n') = cost(a, n) + cost(n, n')$
- What is the worst-case complexity of the algorithm?
  - Complexity is $O(n^3)$, where n is the number of nodes
    - Each loop requires $O(n^2)$ operations
      - For each node, go through all other nodes and see if a shorter path exists
    - A total of $n - 1$ loops
      - Longest path to any node is $n - 1$ steps
- Turns out the same algorithm can be applied to MDPs
  - Find the optimal policy from any node

Workday Example

- Suppose $T = 2$
- What is $v_*^{T-1}(Pub)$?
  $q_*^{T-1}(Pub, Work) = -0.5$
  $q_*^{T-1}(Pub, Relax) = -0.1$
    - $Relax$ is better
- What is $v_*^{T-1}(OH)$?
  $q_*^{T-1}(OH, Relax) = 0.5$
- What is $v_*^{T-1}(FLE)$?
  $q_*^{T-1}(FLE, Work) = -0.32$
- Similarly, $q_*^{T-1}(MLS, Relax) = 0.5$
- Similarly, $q_*^{T-1}(Teach, Relax) = 2.9$

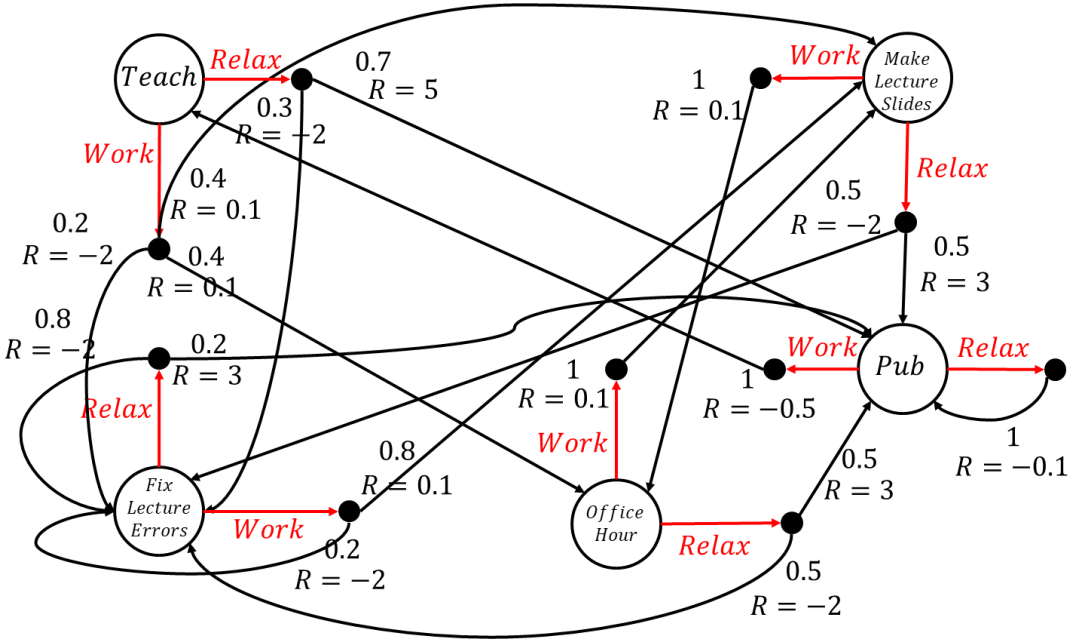


Workday Example

- Suppose $\gamma = 0.9$
- What is $v_*^0(Pub)$?

$q_*^0(Pub, Work) =$
$-0.5 + 0.9 * v_*^1(Teach)$
$= 2.11$

$q_*^0(Pub, Relax) =$
$-0.1 + 0.9 * v_*^1(Pub)$
$= -0.19$



- What is $v_*^0(FLE)$?

$q_*^0(FLE, Work) =$
$(0.2 * -2 + 0.8 * 0.1) +$
$0.2 * 0.9 * v_*^1(FLE) + 0.8 * 0.9 * v_*^1(MLS) =$
$= -0.0176$

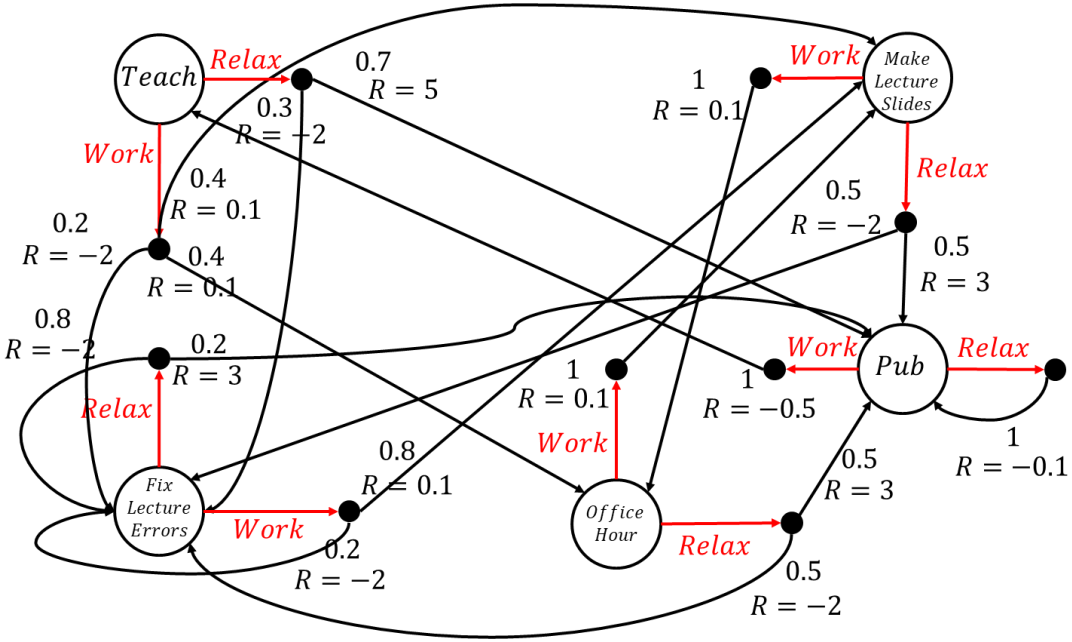
$q_*^0(FLE, Relax) = -1 - 0.9 * 0.276 = -1.25$

	$t = 0$	$t = 1$
$Teach$		2.9
OH		0.5
MLS		0.5
FLE		-0.32
Pub		-0.1

Value table

Workday Example

- Suppose $\gamma = 0.9$
- The other action values computed similarly



	$t = 0$	$t = 1$
Teach		2.9
OH		0.5
MLS		0.5
FLE	−0.0176	−0.32
Pub	2.11	−0.1

Value table

Finite-Horizon Dynamic Programming Summary

- Iterate backwards, starting from last decision step $T-1$
- First, compute $q_*^{T-1}(s,a)$ for each state/action pair
  - i.e., compute the one-step expected reward
  - Then set $v_*^{T-1}(s) = \max_a q_*^{T-1}(s,a)$
- For $t < T-1$, use Bellman equation to compute $q_*^t(s,a)$

$$q_*^t(s,a) = \mathbb{E}_*\left[R_{t+1} + \gamma v_*^{t+1}(S_{t+1}) \middle| S_t = s, A_t = a\right]$$

  - Then set $v_*^t(s) = \max_a q_*^t(s,a)$
- What is the complexity of dynamic programming?

$$O\left(TS^2A\right)$$

  - where S is the number of states, A is the number of actions
  - for each state, loop through all actions and all other states

Dynamic Programming Assumptions

- So far dynamic programming only works for the case of
  - Finite horizon
  - Finite-state space
  - Finite-action space
- Finite-state and –action spaces hard to relax (for now)
- But we can modify algorithm for infinite horizon
- Policy iteration!

Policy Iteration

- The greedy policy improvement theorem suggests an algorithm for finding the optimal policy through iterating
  - Start from a policy, compute its value function, improve greedily for one state, repeat…

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

  - Terminate when you find optimal policy
- Is this guaranteed to terminate?
  - Yes, there are finitely many policies in a finite-state MDP
- Policy iteration is trickier in the finite-horizon case
  - Need to evaluate/improve the policy at each time t
  - Use finite-horizon dynamic programming in that case

Policy Iteration, Workday Example

- What are the optimal actions in the long run?
  - $\pi_*(Teach) = Relax$
  - $\pi_*(OH) = Relax$
  - $\pi_*(MLS) = Relax$
  - $\pi_*(FLE) = Work$
  - $\pi_*(Pub) = Work$
- Corresponding values are

$$v_*(s) = [9.18 \;\; 6.31 \;\; 6.31 \;\; 5.15 \;\; 7.76]$$

Policy Iteration Summary

- Start with a random policy π
- Repeat until you find the optimal policy:
  - Loop through all states
  - For each state s, loop through all actions
    - If you find an action a for which $q_\pi(s, a) > v_\pi(s)$
    - Modify π such that $\pi(s) = a$
  - Recalculate values v_π, for modified policy π'
    - Can do this step either after each action change or after a full loop over all states
  - If you did not change the policy at all, terminate
    - You found the optimal!

Value Iteration

- Policy iteration requires evaluating each new policy
  - i.e., need to compute $v_\pi(s)$ for all states
  - May take significant time
- Another approach is to use the Bellman optimality equation

$$v_*(s) = \max_a q_*(s,a)$$
$$= \max_a \left[\mathbb{E}_*[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]\right]$$

- The Bellman optimality equation suggests the recursion

$$v_{k+1}(s) = \max_a \left[\mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a]\right]$$

  - Starting from any v_0

Bellman Operator

- Consider the mapping

$$Lv = \max_a \left[\mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s, A_t = a]\right]$$

- This is known as the Bellman operator
- It is what enables the recursion

$$\begin{aligned} v_{k+1}(s) &= \max_a \left[\mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a]\right] \\ &= Lv_k \end{aligned}$$

- Essentially an application of the Policy Improvement Theorem over the state value
  - Except the v's may not be the values of any actual policy
    - Yet…

Value Iteration, cont'd

- The Bellman optimality equation suggests the recursion

$$v_{k+1}(s) = \max_a \left[\mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a]\right]$$

- The sequence is guaranteed to converge
  - Consider the Bellman operator

$$Lv = \max_a \left[\mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s, A_t = a]\right]$$

  - Bellman operator can be shown to be contractive
    - i.e., any 2 sequences get closer to each other after each iteration
  - The sequence v_k converges to a unique v_* for all v_0
- The unique v_* satisfies the Bellman optimality equation

$$v_*(s) = \max_a \left[\mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]\right]$$

  - So it is the value function corresponding to the optimal policy

Value Iteration Considerations

- Given a value function, the corresponding policy is

$$\pi(s) = argmax_a \, q_\pi(s,a)$$
$$= argmax_a \, \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a]$$
$$= \arg\max_a \left[R_e(s,a) + \sum_{s'} \gamma v_\pi(s') P(s,a,s') \right]$$

- Note that a v_k may not have the actual state values of the policy it represents
  - There are finitely many policies but infinitely many v_k
  - Ultimately, we don't care what the v_k are as long as the policy is optimal
  - Of course, when the v_k converge, the values will converge to the values of the optimal policy

Value Iteration Summary

- Start from an arbitrary v_0
- For each state s, update v_{k+1} as follows:

$$v_{k+1}(s) = \max_a \left[\mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \right]$$

- Iterate until $|v_{k+1} - v_k| < \epsilon$
  - Where ϵ is a hyperparameter
  - Can use your favorite norm above, e.g., L_∞
- Unlike policy iteration, no need to invert large matrices
  - Though may require many more iterations
  - Depending on MDP structure, value iteration may scale better than policy iteration

Workday Comparison

- The workday example has 5 states and 2 actions
  - How many policies are there in total?

$$2^5 = 32$$

  - Policy iteration likely to converge in several iterations
- Value iteration takes several dozen iterations to converge to true values
  - From an initial state of all 0's
  - Though you don't need to converge fully until you uncover the optimal policy
- Grid world has a bigger policy space

$$4^{25}$$

  - Optimal policy is very simple, so policy iteration still fast

Asynchronous Dynamic Programming

- Dynamic programming can't scale to very large state spaces
  - Will require an infeasible number of iterations
- How can we improve scalability?
  - Asynchronous dynamic programming!
  - i.e., only update some, hopefully "important", state values
- How do we choose which values to update?
  - Simplest approach is random sampling
    - May bring some improvements, but unlikely to help in general
    - At least guaranteed to converge to optimal values
  - Can focus on specific parts of the state space
    - E.g., work backwards from goal states, if possible
    - Or only update states whose values will change significantly

Asynchronous Dynamic Programming, cont'd

- Updating only states whose values will change significantly is known as prioritized sweeping
  - For example, use an agent to experience the MDP and thus guide which states to use DP for
  - This is getting closer to an actual RL setting where the Q-values are learned rather than calculated directly
- Any challenges with this approach?
  - If state space is large, hard to find/define important states
    - Ultimately, this is the exploration problem
  - Calculating expectations in value iteration may be very slow in a large state space
- Will discuss more effective approaches next!

Deterministic vs Stochastic Policy

- So far, we have mainly focused on deterministic policies
  - Both policy and value iteration will find the optimal *deterministic* policy
  - But could a stochastic policy be better?
  - Turns out the answer is no – it suffices to look at deterministic policies only
    - Of course, you might need a stochastic policy to explore – more next time!
- **Theorem**: Suppose there exists an optimal stochastic policy π_s for a given MDP. Then there also exists an optimal deterministic policy π_d.
  - Proof is fairly involved, so we will skip it

Markovian Policies vs History-Based Policies

- **Question**: can you come up with a better policy if you're given the full history of the execution so far?
- It turns out the answer is once again no
  - This should make sense intuitively since MDP transitions are only affected by the current state
- **Theorem**: Suppose there exists an optimal (stochastic) history-based policy π_h. Then there also exists an optimal Markovian deterministic policy π_d.
  - Proof is fairly involved, so we will skip it
- **Note**: Deterministic policies may not be sufficient in case of partially observed MDPs
  - E.g., a self-driving car with a single camera can only observe objects within the current image

Summary

- Dynamic programming is a powerful iterative algorithm
- Very popular in some fields of computer science and engineering
  - Widely used in control, in a similar way to RL
- Vanilla algorithm only works for finite-space MDPs
  - Overall iteration idea is still mainstream RL, however
- All algorithms discussed so far also need the user to know the MDP structure
  - Not realistic in many cases
  - Will relax this assumption next