Lecture 16: Markov Decision Processes. Policies and value functions.

- Markov decision processes
- Policies and value functions
- Dynamic programming algorithms for evaluating policies and optimizing policies
- Introduction to learning

Recall: Markov Decision Processes (MDPs)

- Finite set of *states* S (we will lift this later)
- Finite set of *actions* A
- $\gamma =$ *discount factor* for future rewards (between 0 and 1, usually close to 1). Two possible interpretations:
  - At each time step there is a $1 - \gamma$ chance that the agent dies, and does not receive rewards afterwards
  - Inflation rate: if you receive the same amount of money in a year, it will be worth less
- *Markov assumption:* s_{t+1} and r_{t+1} depend only on s_t and a_t but not on anything that happened before time t

Recall: Models for MDPs

- Because of the Markov property, an MDP can be completely described by:
  - *Reward function* $r : S \times A \to \mathbb{R}$
    $r_a(s)$ = the immediate reward if the agent is in state s and takes action a
    This is the *short-term utility* of the action
  - *Transition model* (dynamics): $T : S \times A \times S \to [0, 1]$
    $T_a(s, s')$ = probability of going from s to s' under action a

$$T_a(s, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$$

- These form the *model* of the environment

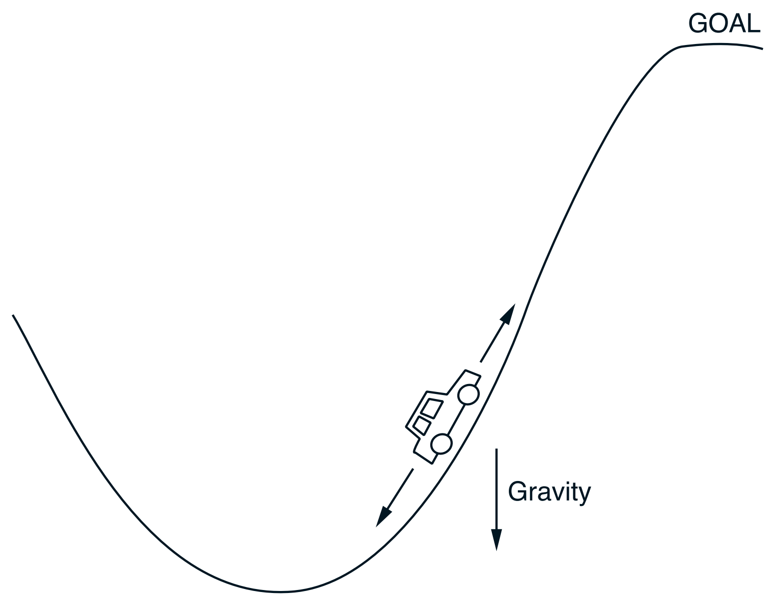
Recall: Discounted returns

- The *discounted return* R_t for a trajectory, starting from time step t, can be defined as:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=1}^{\infty} \gamma^{t+k-1} r_{t+k}$$

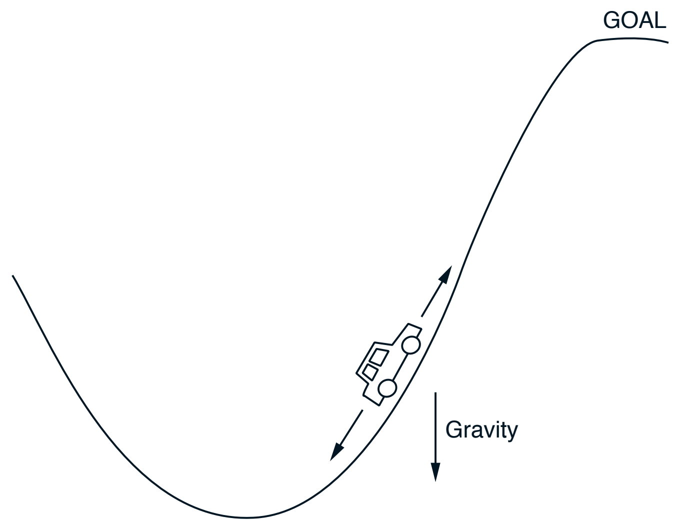
Discount factor $\gamma < 1$ ensures that the return is finite, assuming that rewards are bounded.

Example: Mountain-Car



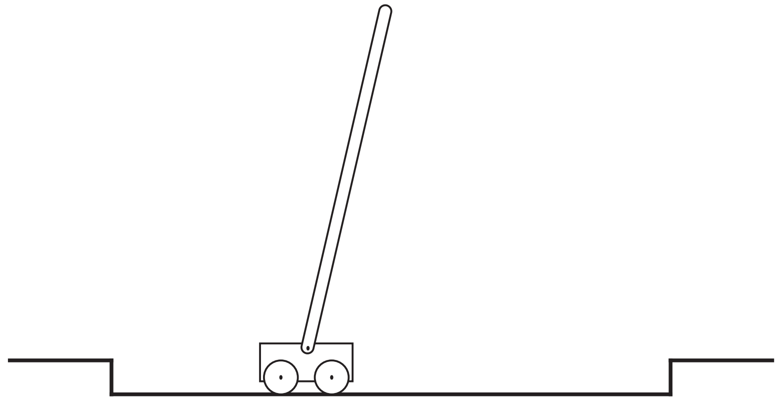
- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- We want the car to get to the top of the hill as quickly as possible
- How do we define the rewards? What is the return?

Example: Mountain-Car



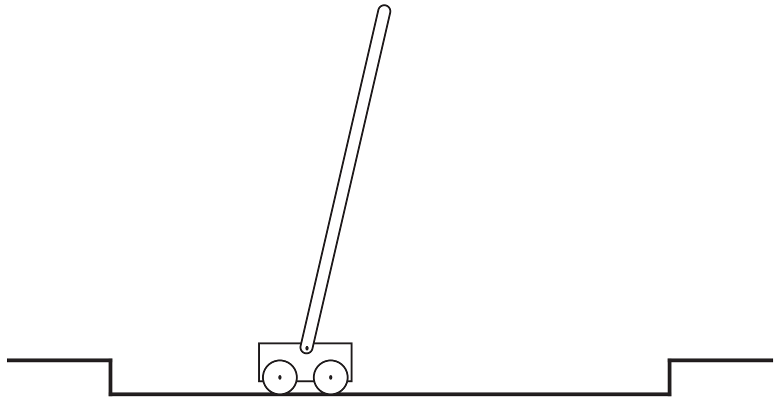
- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- Two reward formulations:
  1. reward $= -1$ for every time step, until car reaches the top
  2. reward $= 1$ at the top, 0 otherwise $\gamma < 1$
- In both cases, the return is maximized by minimizing the number of steps to the top of the hill

Example: Pole Balancing



- We can push the cart along the track
- The goal is to avoid failure: pole falling beyond a given angle, or cart hitting the end of the track
- What are the states, actions, rewards and return?

Example: Pole Balancing



- States are described by 4 variables: angle and angular velocity of the pole relative to the cart, position and speed of cart along the track
- We can think of 3 possible actions: push left, push right, do nothing
- Episodic task formulation: reward $= +1$ for each step before failure
  $\Rightarrow$ return $=$ number of steps before failure
- Continuing task formulation: reward $=$ -1 upon failure, 0 otherwise, $\gamma < 1$
  $\Rightarrow$ return $= -\gamma^k$ if there are k steps before failure

Formulating Problems as MDPs

- The *rewards are quite "objective"* (unlike, e.g., heuristics), they are intended to capture the goal for the problem
- Often there are several ways to formulate a sequential decision problem as an MDP
- It is important that the state is defined in such a way that the Markov property holds
- Sometimes we may start with a more informative or lenient reward structure in the beginning, then change it to reflect the real task
- In psychology/animal learning, this is called *shaping*

Formulating Games as MDPs

- Suppose you played a game against a fixed opponent (possibly stochastic), which acts only based on the current board
- We can formulate this problem as an MDP by *making the opponent part of the environment*
- The states are all possible board positions for your player
- The actions are the legal moves in each state where it is your player's turn
- If we do not care about the length of the game, then $\gamma = 1$
- Rewards can be $+1$ for winning, -1 for losing, 0 for a tie (and 0 throughout the game)
- But it would be hard to define the transition probabilities!
- Later we will talk about how to learn such information from data/experimentation

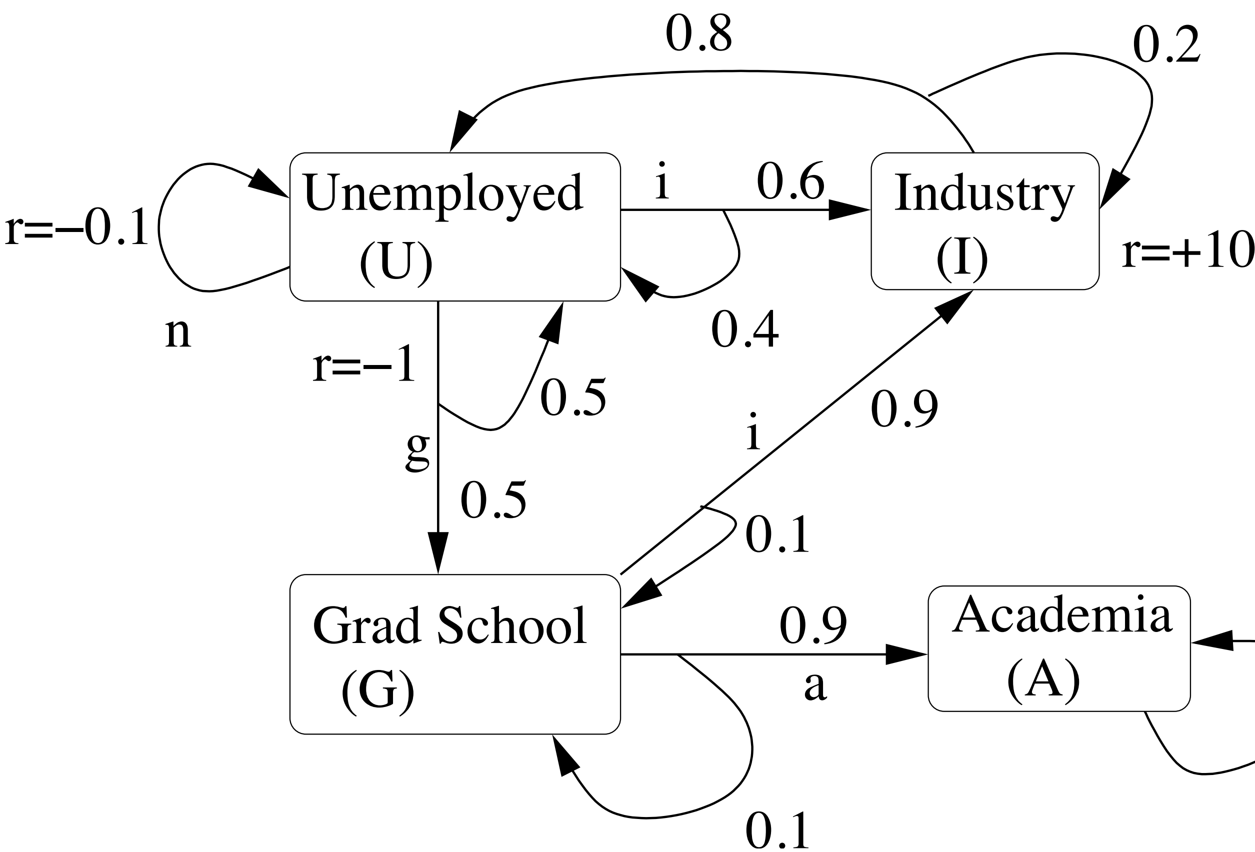
Policies

- The goal of the agent is to find a way of behaving, called a *policy* (plan or strategy) that maximizes the expected value of the return, $E[R_t], \forall t$
- A *policy* is a way of choosing actions based on the state:
  - *Stochastic policy*: in a given state, the agent can "roll a die" and choose different actions

$$\pi : S \times A \to [0, 1], \;\; \pi(s, a) = P(a_t = a | s_t = s)$$

  - *Deterministic policy*: in each state the agent chooses a unique action $\pi : S \to A, \;\; \pi(s) = a$

Example: Career Options



n=Do Nothing
i = Apply to industry
g = Apply to grad school
a = Apply to academia

What is the best policy?

Value Functions

- Because we want to find a policy which maximizes the expected return, it is a good idea to *estimate the expected return*
- Then we can *search* through the space of policies for a good policy
- *Value functions* represent the expected return, for every state, given a certain policy
- Computing value functions is an intermediate step towards computing good policies

State Value Function

- The *state value function of a policy* π is a function $V^\pi : S \rightarrow \mathbb{R}$
- The *value of state* s *under policy* π is the expected return if the agent starts from state s and picks actions according to policy π:

$$V^\pi(s) = E_\pi[R_t | s_t = s]$$

- For a finite state space, we can represent this as an array, with one entry for every state
- We will talk later about methods used for very large or continuous state spaces

Computing the value of policy π

- First, re-write the return a bit:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \\ &= r_{t+1} + \gamma \left(r_{t+2} + \gamma r_{t+3} + \cdots \right) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

- Based on this observation, V^π becomes:

$$V^\pi(s) = E_\pi[R_t | s_t = s] = E_\pi[r_{t+1} + \gamma R_{t+1} | s_t = s]$$

- Now we need to recall some properties of expectations...

Detour: Properties of expectations

- Expectation is *additive*: $E[X+Y] = E[X] + E[Y]$

  Proof: Suppose X and Y are discrete, taking values in $\mathcal{X}$ and $\mathcal{Y}$

$$
\begin{aligned}
E[X+Y] &= \sum_{x_i \in \mathcal{X}, y_i \in \mathcal{Y}} (x_i + y_i) p(x_i, y_i) \\
&= \sum_{x_i \in \mathcal{X}} x_i \sum_{y_i \in \mathcal{Y}} p(x_i, y_i) + \sum_{y_i \in \mathcal{Y}} y_i \sum_{x_i \in \mathcal{X}} p(x_i, y_i) \\
&= \sum_{x_i \in \mathcal{X}} x_i p(x_i) + \sum_{y_i \in \mathcal{Y}} y_i p(y_i) = E[X] + E[Y]
\end{aligned}
$$

- $E[cX] = cE[X]$ is $c \in \mathbb{R}$ is a constant

  Proof: $E[cX] = \sum_{x_i} c x_i p(x_i) = c \sum_{x_i} x_i p(x_i) = cE[X]$

Detour: Properties of expectations (2)

- The expectation of the product of random variables *is not* equal to the product of expectations, unless the variables are independent

$$E[XY] = \sum_{x_i \in \mathcal{X}, y_i \in \mathcal{Y}} x_i y_i p(x_i, y_i) = \sum_{x_i \in \mathcal{X}, y_i \in \mathcal{Y}} x_i y_i p(x_i | y_i) p(y_i)$$

- If X and Y are independent, then $p(x_i|y_i) = p(x_i)$, we can re-arrange the sums and products and get $E[X]E[Y]$ on the right-hand side
- But is X and Y are not independent, the right-hand side does not decompose!

Going back to value functions...

- We can re-write the value function as:

$$\begin{aligned} V^\pi(s) &= E_\pi[R_t|s_t = s] = E_\pi[r_{t+1} + \gamma R_{t+1}|s_t = s] \\ &= E_\pi[r_{t+1}] + \gamma E[R_{t+1}|s_t = s] \text{ (by linearity of expectation)} \\ &= \sum_{a \in A} \pi(s,a) r_a(s) + \gamma E[R_{t+1}|s_t = s] \text{ (by using definitions)} \end{aligned}$$

- The second term looks a lot like a value function, if we were to condition on s_{t+1} instead of s_t
- So we re-write as:

$$E[R_{t+1}|s_t = s] = \sum_{a \in A} \pi(s,a) \sum_{s' \in S} T_a(s,s') E[R_{t+1}|s_{t+1} = s']$$

- The last term is just $V^\pi(s')$

Bellman equations for policy evaluation

- By putting all the previous pieces together, we get:

$$V^\pi(s) = \sum_{a \in A} \pi(s,a) \left(r_a(s) + \gamma \sum_{s' \in S} T_a(s,s') V^\pi(s') \right)$$

- This is a *system of linear equations* (one for every state) whose unique solution is V^π.
- The uniqueness is ensured under mild technical conditions on the transitions p
- So if we want to find V^π, we could try to solve this system!

Iterative Policy Evaluation

- Main idea: turn Bellman equations into update rules.
  1. Start with some initial guess V_0
  2. During every iteration k, update the value function for all states:

$$V_{k+1}(s) \leftarrow \sum_{a \in A} \pi(s,a) \left(r_a(s) + \gamma \sum_{s' \in S} T_a(s,s') V_k(s') \right), \forall s$$

  3. Stop when the maximum change between two iterations is smaller than a desired threshold (the values stop changing)
- This is a *bootstrapping* algorithm: the value of one state is updated based on the current estimates of the values of successor states
- This is a dynamic programming algorithm
- If you have a linear system that is very big, using this approach avoids a big matrix inversion

Searching for a Good Policy

- We say that $\pi \geq \pi'$ if $V^{\pi}(s) \geq V^{\pi'}(s) \forall s \in S$
- This gives a partial ordering of policies: if one policy is better at one state but worse at another state, the two policies are incomparable
- Since we know how to compute values for policies, we can search through the space of policies
- Local search seems like a good fit.

Policy Improvement

$$V^\pi(s) = \sum_{a \in A} \pi(s,a) \left(r(s,a) + \gamma \sum_{s' \in S} T_a(s,s') V^\pi(s') \right)$$

- Suppose that there is some action a^*, such that:

$$r(s,a^*) + \gamma \sum_{s' \in S} p(s,a^*,s') V^\pi(s') > V^\pi(s)$$

- Then, if we set $\pi(s,a^*) \leftarrow 1$, the value of state s will increase
- This is because we replaced each element in the sum that defines $V^\pi(s)$ with a bigger value
- The values of states that can transition to s increase as well
- The values of all other states stay the same
- So the new policy using a^* is better than the initial policy π!

Policy iteration idea

- More generally, we can change the policy π to a new policy π', which is *greedy* with respect to the computed values V^π

$$\pi'(s) = \arg\max_{a \in A} \left(r(s,a) + \gamma \sum_{s' \in S} T_a(s,s') V^\pi(s') \right)$$

  Then $V^{\pi'}(s) \geq V^\pi(s), \forall s$

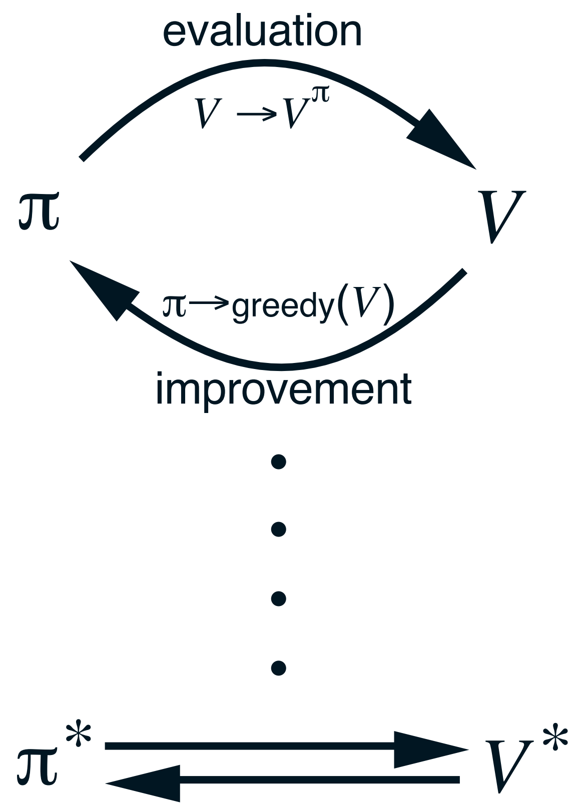
- This gives us a local search through the space of policies
- We stop when the values of two successive policies are identical

Policy Iteration Algorithm

1. Start with an initial policy π_0 (e.g., uniformly random)
2. Repeat:
   (a) Compute V^{π_i} using policy evaluation
   (b) Compute a new policy π_{i+1} that is greedy with respect to V^{π_i}

   until $V^{\pi_i} = V^{\pi_{i+1}}$

Generalized Policy Iteration



- In practice, we could run policy iteration incrementally
- Compute the value just to some approximation
- Make the policy greedy only at some states, not all states

Properties of policy iteration

- If the state and action sets are finite, there is a very large but finite number of deterministic policies
- Policy iteration is a greedy local search in this finite set
- We move to a new policy only if it provides a strict improvement
- So the algorithm *has to terminate*
- But if it is a greedy algorithm, can we guarantee an optimal solution?

Optimal Policies and Optimal Value Functions

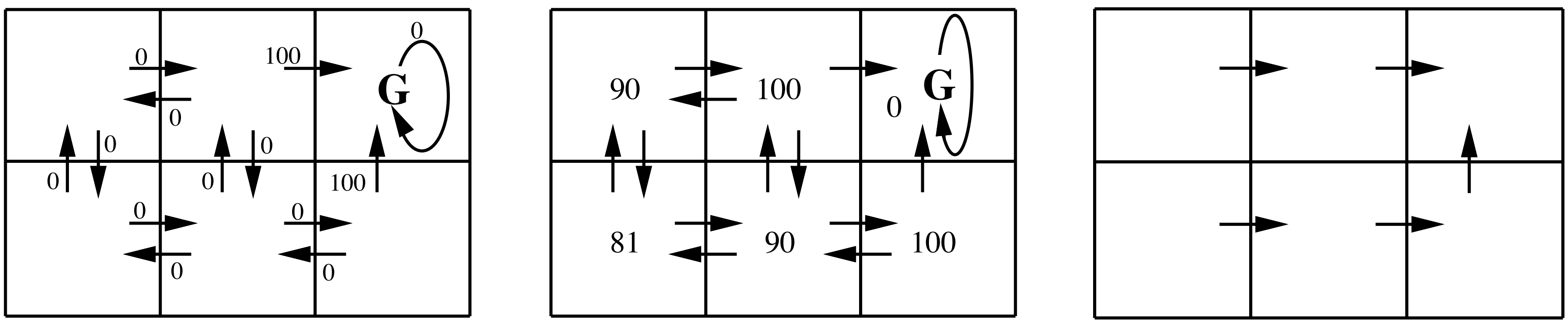
- Our goal is to find a policy that has maximum expected utility, i.e. maximum value
- Does policy iteration fulfill this goal?
- The *optimal value function* V^* is defined as the best value that can be achieved at any state:

$$V^*(s) = \max_\pi V^\pi(s)$$

- *In a finite MDP, there exists a unique optimal value function* (shown by Bellman, 1957)
- Any policy that achieves the optimal value function is called *optimal policy*
- There has to be at least one deterministic optimal policy

Illustration: A Gridworld

- Transitions are deterministic, as shown by arrows
- Discount factor $\gamma = 0.9$
- Optimal state values give information about the shortest path to the goal
- There are ties between optimal actions, so there is an infinite number of optimal policies
- One of the deterministic optimal policies is shown on right.



Reward values | $V^*(s)$ values | One optimal policy

Bellman Optimality Equation for V^*

- The value of a state under the optimal policy must be equal to the expected return for the best action in the state:

$$\begin{aligned} V^*(s) &= \max_a E\left[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right] \\ &= \max_a \left(r(s,a) + \gamma \sum_{s'} T_a(s,s') V^*(s') \right) \end{aligned}$$

  by an argument very similar to the policy evaluation case
- V^* is the *unique solution* of this system of non-linear equations (one equation for every state)
- The fact that there is a unique solution was proven by Bellman, and relies on the fact that $\gamma < 1$, and on an argument similar to the proof of convergence of policy iteration from last time

Why Optimal Value Functions are Useful

- Any policy that is greedy with respect to V^* is an optimal policy!
- If we know V^* and the model of the environment, *one step of look-ahead* will tell us what the optimal action is:

$$\pi^*(s) = \arg\max_a \left(r(s,a) + \gamma \sum_{s'} T_a(s,s') V^*(s') \right)$$

- This is in contrast to other algorithms we studied, for which finding an optimal solution required deep search!
- If the values are not computed perfectly, search might still help, though (e.g. in games)
- One way to compute optimal value functions is through policy iteration.

Computing Optimal Values: Value Iteration

- Main idea: Turn the Bellman optimality equation into an update rule (same as done in policy evaluation):

  1. Start with an arbitrary initial approximation V_0
  2. On each iteration, update the value function estimate:

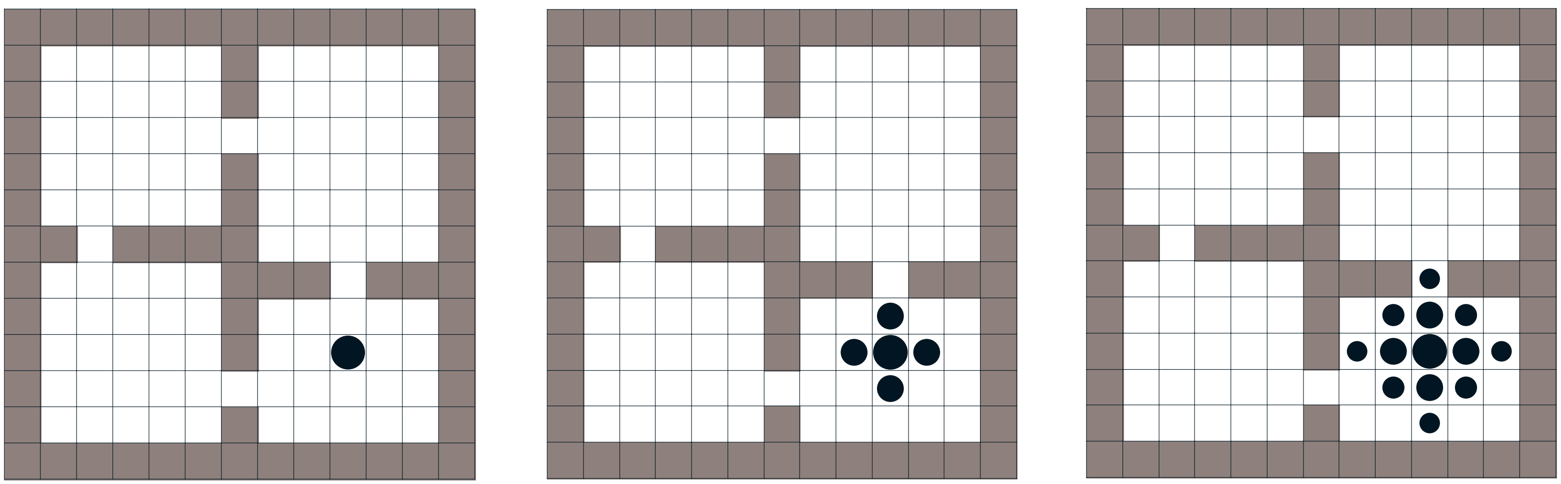
$$V_{k+1}(s) \leftarrow \max_a \left(r(s,a) + \gamma \sum_{s'} T_a(s,s') V_k(s') \right), \forall s$$

  3. Stop when the maximum value change between iterations is below a threshold

- The algorithm converges (in the limit) to the true V^* (almost identical proof to policy evaluation)

Illustration: Rooms Example

- Each square is a state; black squares are walls, initial circle (left) is the goal state
- Four actions, fail 30% of the time
- No rewards until the goal is reached, $\gamma = 0.9$.
- Circles indicate the magnitude of the value of the corresponding state (no circle means 0 value)
- Values propagate backwards from the goal



Iteration #1 Iteration #2 Iteration #3

A More Efficient Algorithm

- Instead of updating all states on every iteration, focus on *important states*
- Here, we can define important as *visited often*

  E.g., board positions that occur on every game, rather than just once in 100 games
- *Asynchronous dynamic programming*:
  - Generate trajectories through the MDP
  - Update states whenever they appear on such a trajectory
- This focuses the updates on states that are actually possible.

How Is Learning Tied with Dynamic Programming?

- Observe transitions in the environment, learn an *approximate model* $\hat{R}(s, a), \hat{T}_a(s, s')$
  - Use maximum likelihood to compute probabilities
  - Use supervised learning for the rewards
- Pretend the approximate model is correct and use it for any dynamic programming method
- This approach is called *model-based reinforcement learning*
- Many believers, especially in the robotics community