DYNAMIC PROGRAMMING

By: Dimple Bohra

CONTENTS

**Reference: Chapter 4 from Sutton and Barto

DYNAMIC PROGRAMMING

- Dynamic Programming is a mathematical optimization approach typically used to improvise recursive algorithms.
- It basically involves simplifying a large problem into smaller sub-problems.
- There are two properties that a problem must exhibit to be solved using dynamic programming:

1. Overlapping Subproblems
2. Optimal Substructure

DYNAMIC PROGRAMMING

- The term dynamic programming (DP) in RL refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP).
- Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically.

DYNAMIC PROGRAMMING

- Assume that the environment is a finite MDP.
- That is, we assume that its state, action, and reward sets, S, $A(s)$, and R, for $s \in S$, are finite, and that its dynamics are given by a set of probabilities $p(s', r|s, a)$, for all $s \in S$, $a \in A(s)$, $r \in R$, and $s' \in S+$ ($S+$ is a terminal state if the problem is episodic).
- The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies
- we can easily obtain optimal policies once we have found the optimal value functions, $v*$ or $q*$, which satisfy the Bellman Optimality Equations

DYNAMIC PROGRAMMING

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r | s, a) \Big[r + \gamma v_*(s') \Big] \end{aligned}$$

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \big[r + \gamma v_\pi(s') \big]$$

$$\downarrow$$

$$v_{k+1}(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \big[r + \gamma v_k(s') \big]$$

BELLMAN UPDATE RULE

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$

In this bellman expectation equation:

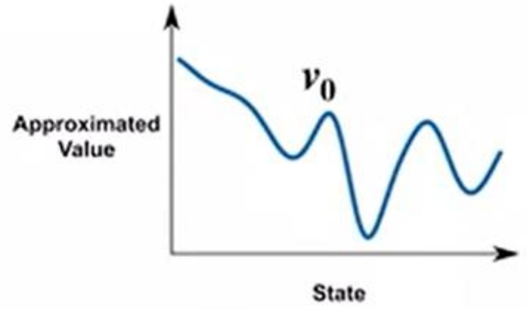
- Given a state ***s*** and an action ***a***, calculate the next state ***s'***.
- Retrieve the state-transition probability distribution ***p***, for the transition from state s to state ***s'*** while taking an action ***a***. Remember that this is model-based RL, the transition probabilities are known.
- Collect the reward ***r***.
- Retrieve the state value ***𝓥(s')*** for the new state ***s'*** from the ***current*** value table.
- Calculate **p * (r + γ 𝓥(s'))**.
- Loop through each action and each possible new state as per policy **π**, and accumulate (mean). The result is the *expected cumulative discounted reward* for the state ***s***.
- Update the ***𝓥(s)*** with the newly calculated value.

ITERATIVE POLICY EVALUATION

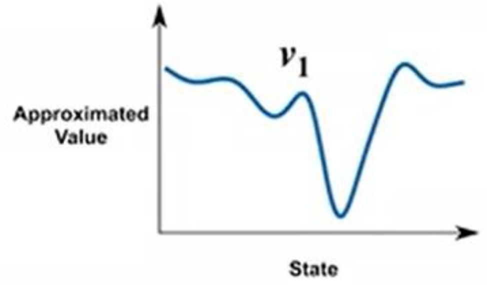
- Iterative Policy Evaluation applies the expected updates for each state, iteratively, until the convergence to the true value function is met.
- Let's see visually how this procedure works. We begin with an arbitrary initialization for our approximate value function, let's call this v_0. Each iteration then produces a better approximation by using the update rule shown at the top of the slide. Each iteration applies these updates to every state, S, in the state space, which we call a sweep. Applying this update repeatedly leads to a better and better approximation to the state value function v Pi. If this update leaves the value function approximation unchanged, that is, if v_k plus 1 equals v_k for all states, then v_k equals v Pi, and we have found the value function.
- This is because v Pi is the unique solution to the Bellman equation. The only way the update could leave v_k unchanged is if v_k already obeys the Bellman equation. In fact, it can be proven that for any choice of v_0, v_k will converge to v Pi in the limit as k approaches infinity.

ITERATIVE POLICY EVALUATION

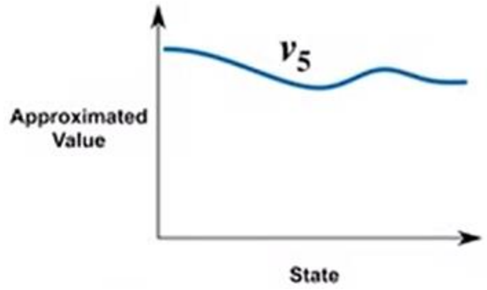
$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$



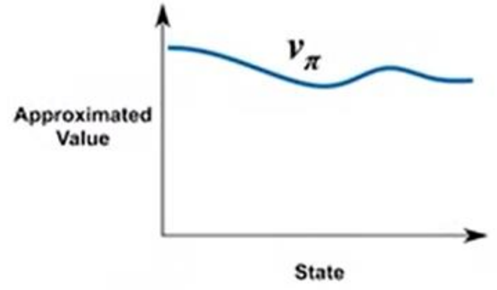
$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$



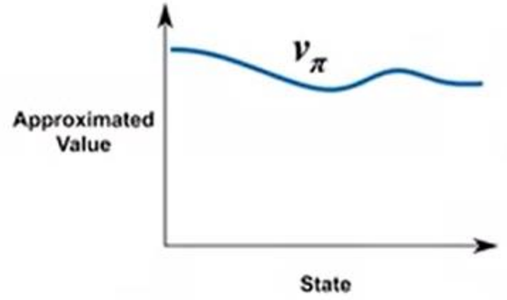
$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$



$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$



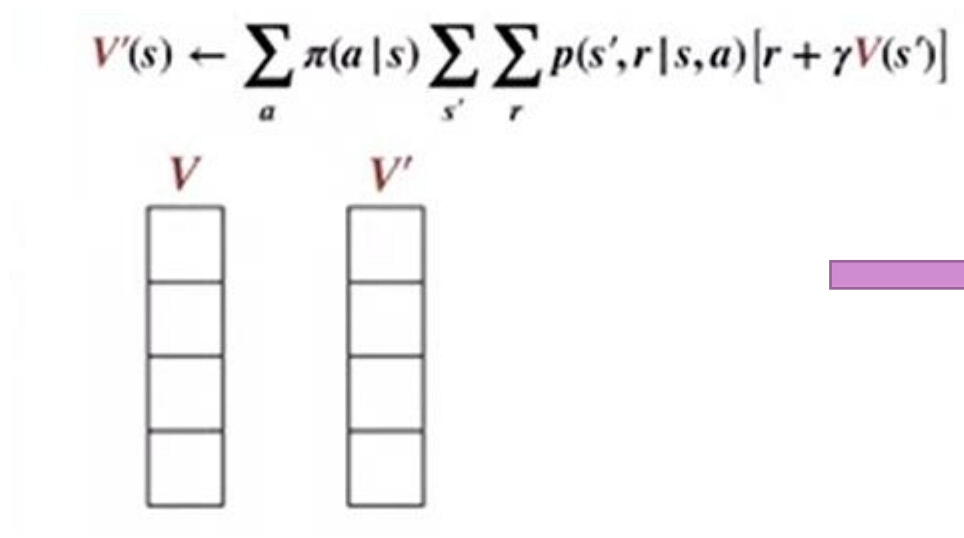
$$v_k(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$



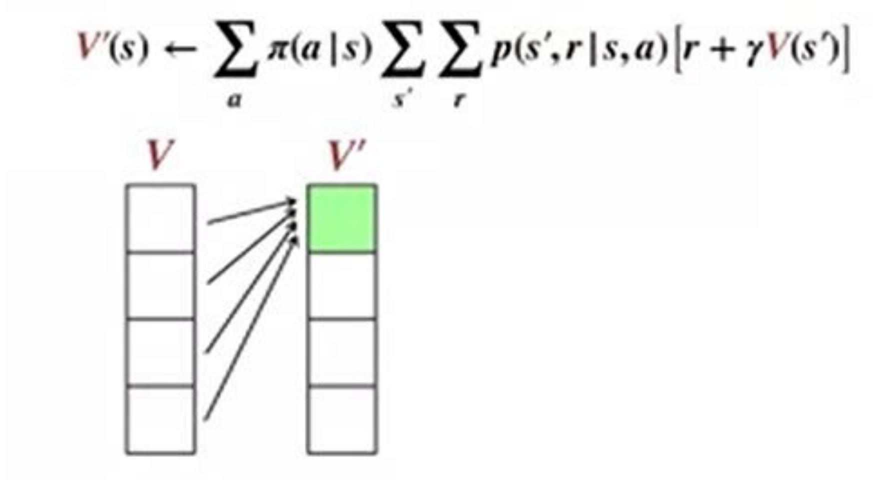
For any v_0

$$\lim_{k\to\infty} v_k = v_\pi$$

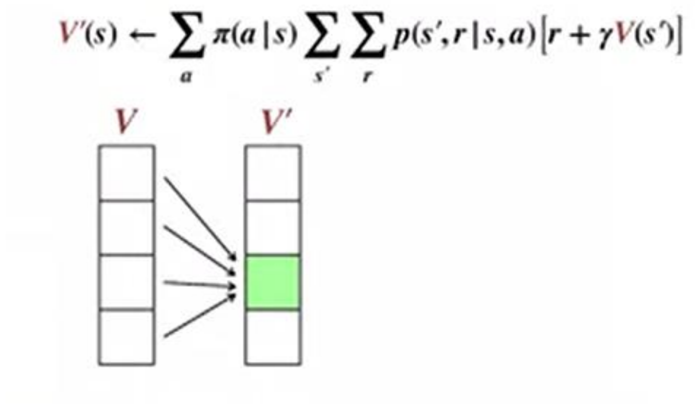
ITERATIVE POLICY EVALUATION



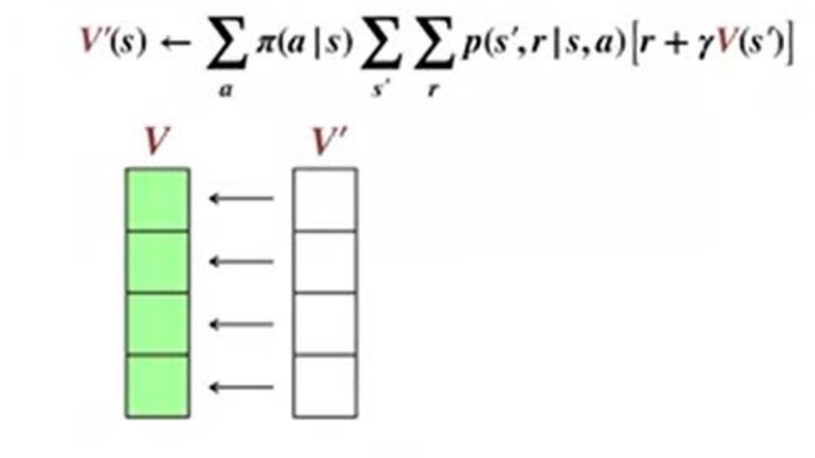
$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma V(s')\right]$$



$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma V(s')\right]$$

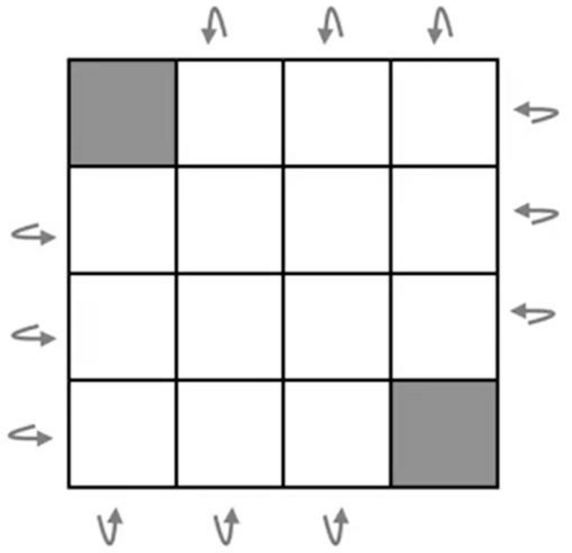


$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma V(s')\right]$$



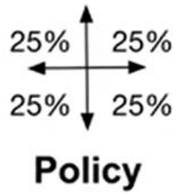
$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma V(s')\right]$$

ITERATIVE POLICY EVALUATION



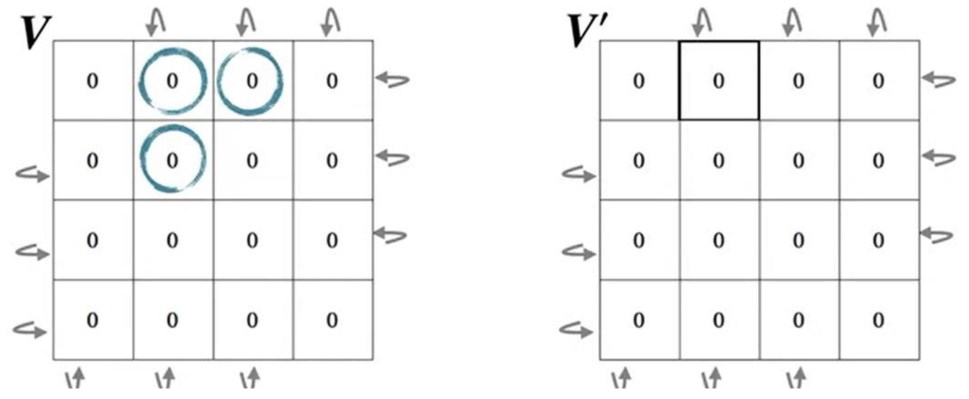
$$R = -1$$

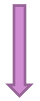
$$\gamma = 1$$



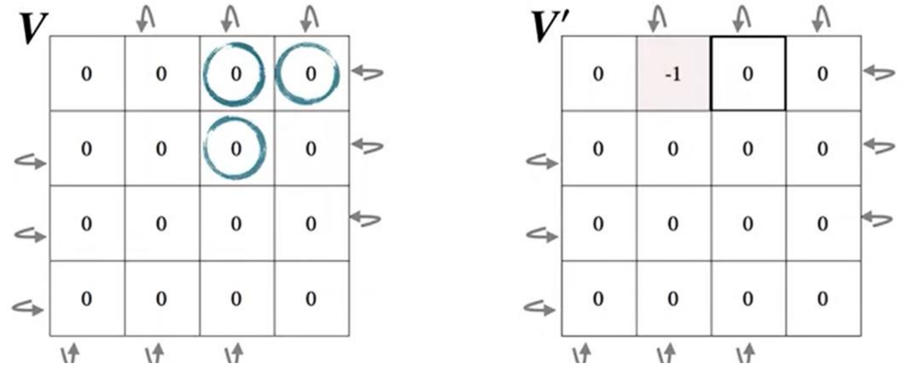
$$v_{k+1}(s) \leftarrow \sum_{a} \pi(a \mid s) \sum_{s'} \sum_{r} p(s', r \mid s, a)\left[r + \gamma v_k(s')\right]$$

$$0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) = -1$$

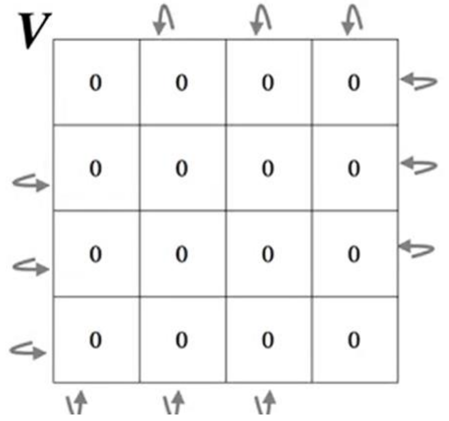


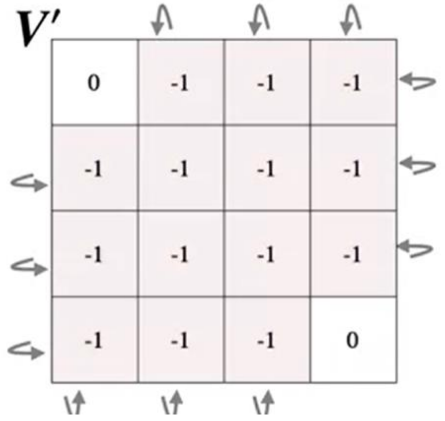


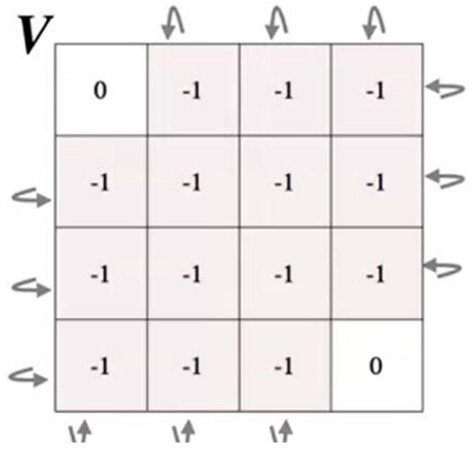
$$0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) + 0.25 * (-1 + 0) = -1$$

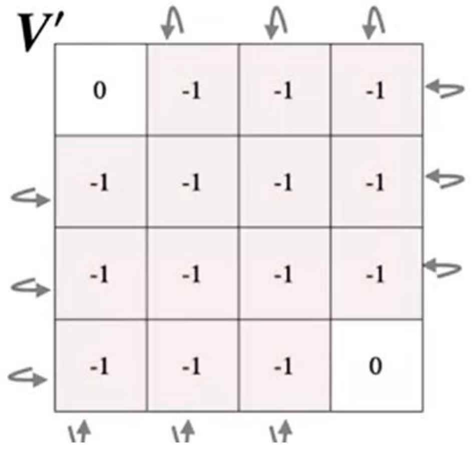


ITERATIVE POLICY EVALUATION

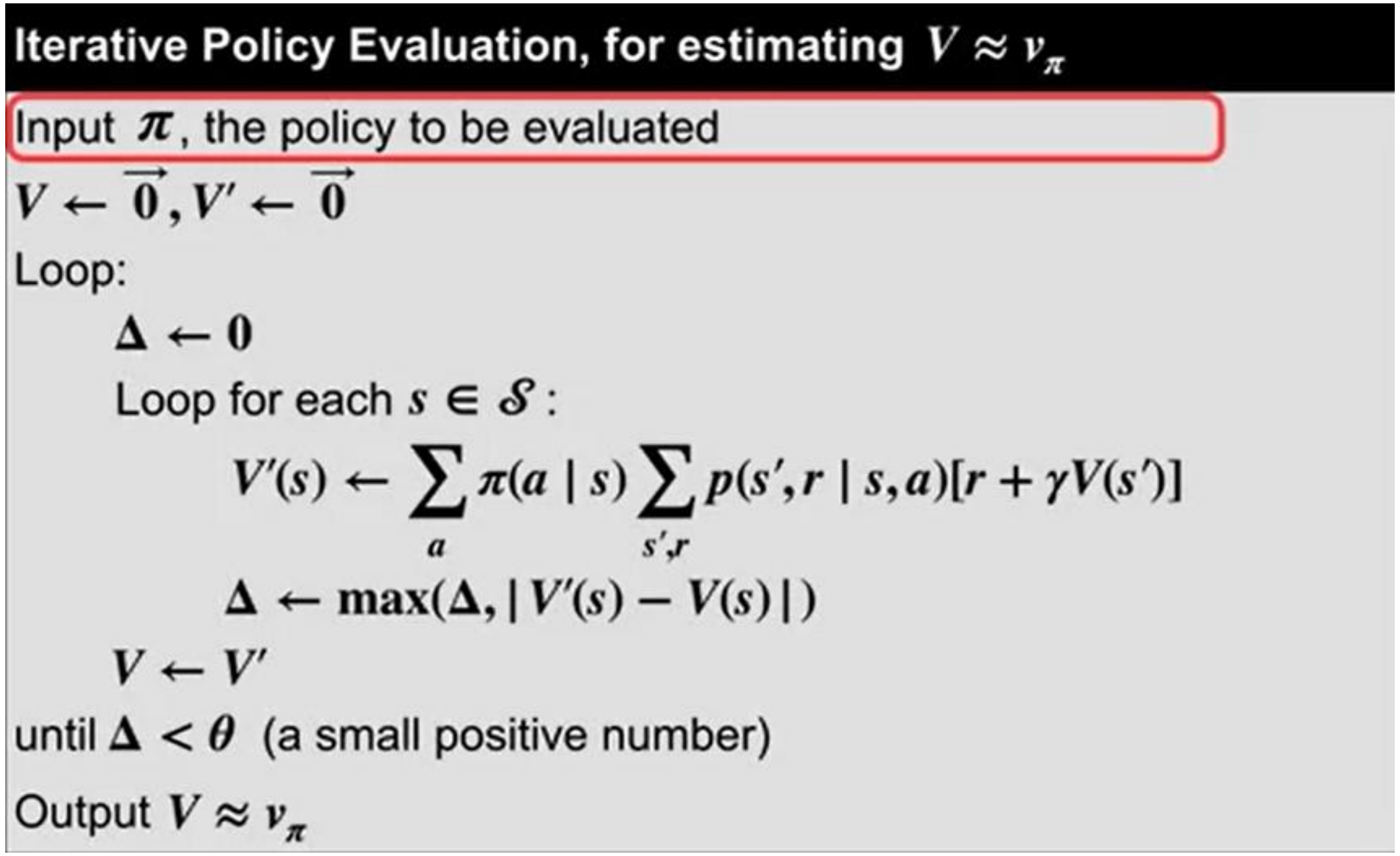








ITERATIVE POLICY EVALUATION

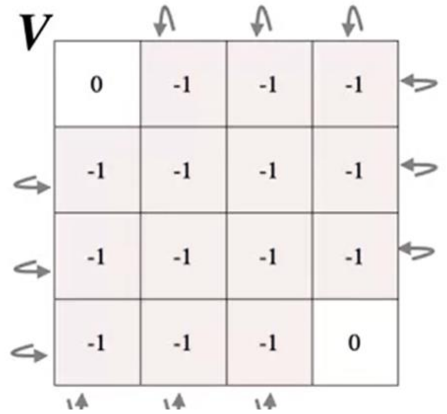


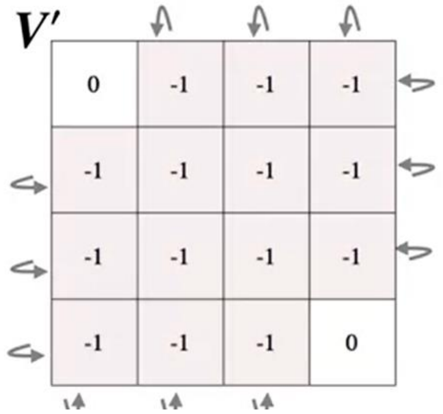
```
Iterative Policy Evaluation, for estimating $V \approx v_\pi$
Input $\pi$, the policy to be evaluated
$V \leftarrow \vec{0}, V' \leftarrow \vec{0}$
Loop:
    $\Delta \leftarrow 0$
    Loop for each $s \in \mathcal{S}$:
        $V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s',r \mid s,a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$
    $V \leftarrow V'$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$
```

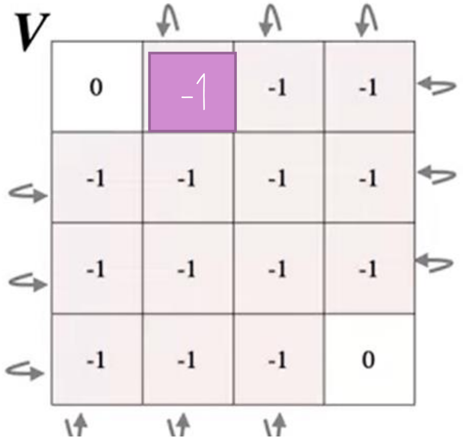
ITERATIVE POLICY EVALUATION

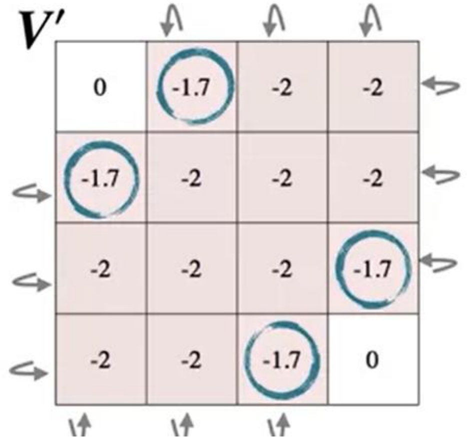
$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$

$\theta = 0.001 \ \Delta = 1.0$









Second iteration (k=2)

$V_{2(S_{0,1})} = 0.25*[-1+0] + 0.25*[-1+(-1)] + 0.25*[-1+(-1)] + 0.25*[-1+(-1)]$

$= -0.25 - 0.5 - 0.5 - 0.5$

$= -1.75 \approx -1.7$

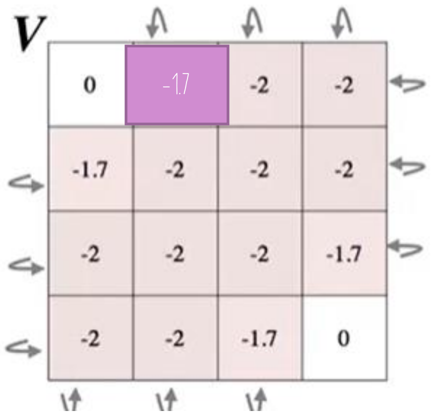
$V_{2(S_{0,2})} = 0.25*[-1+(-1)] + 0.25*[-1+(-1)] + 0.25*[-1+(-1)] + 0.25*[-1-1]$

$= -0.5 - 0.5 - 0.5 - 0.5$

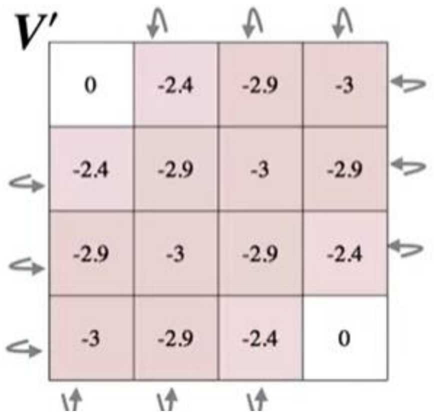
$= -2$

ITERATIVE POLICY EVALUATION

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_k(s')\right]$$



0	-1.7	-2	-2
-1.7	-2	-2	-2
-2	-2	-2	-1.7
-2	-2	-1.7	0



0	-2.4	-2.9	-3
-2.4	-2.9	-3	-2.9
-2.9	-3	-2.9	-2.4
-3	-2.9	-2.4	0

Third iteration (k=3)

$V3_{(S_{0,1})}$ = 0.25*[-1+0] + 0.25*[-1+(-1.7)] + 0.25*[-1+(-2)] + 0.25*[-1+(-2)]

= -0.25 -0.675-0.75-0.75

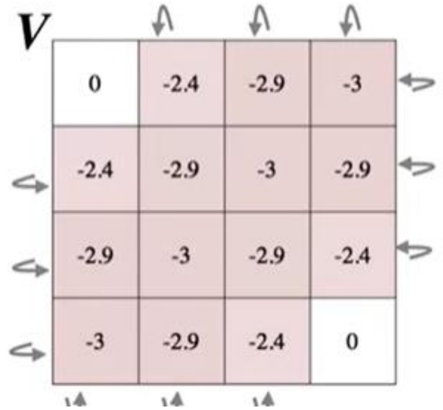
= -2.425 $\approx$ -2.4

$V3_{(S_{0,2})}$ = 0.25*[-1+(-1.7)] + 0.25*[-1+(-2)] + 0.25*[-1+(-2)] + 0.25*[-1+(-2)]

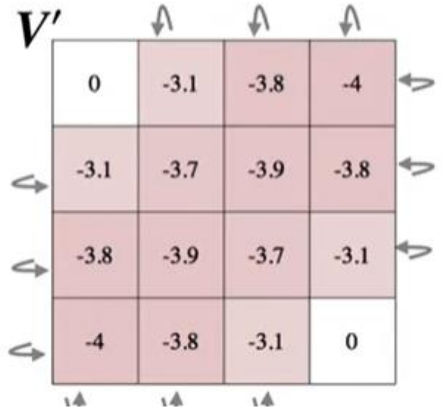
= -0.675-0.75-0.75-0.75

= -2.925 $\approx$ -2.9

fourth iteration (k=4)

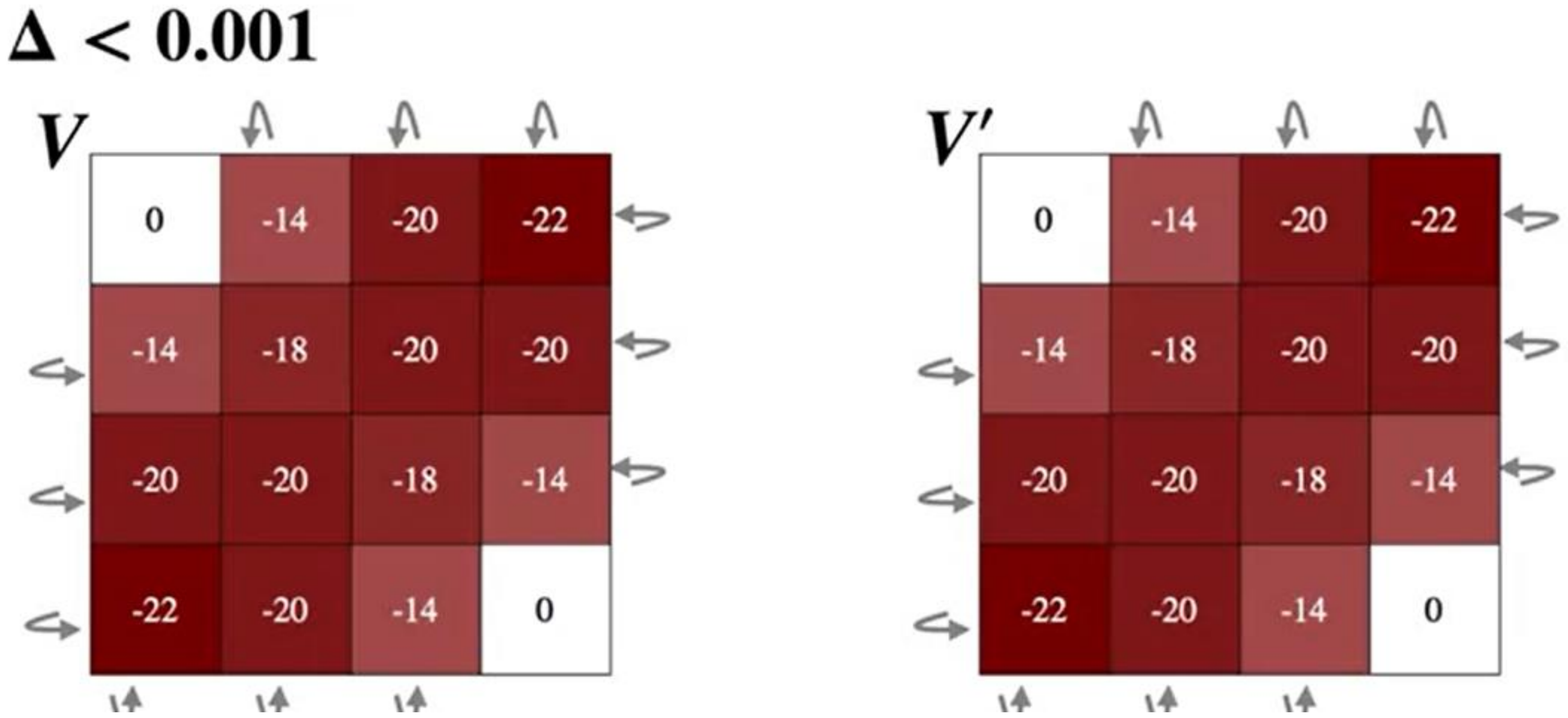


0	-2.4	-2.9	-3
-2.4	-2.9	-3	-2.9
-2.9	-3	-2.9	-2.4
-3	-2.9	-2.4	0



0	-3.1	-3.8	-4
-3.1	-3.7	-3.9	-3.8
-3.8	-3.9	-3.7	-3.1
-4	-3.8	-3.1	0

GRID WORLD USING DYNAMIC PROGRAMMING



$\Delta < 0.001$

V

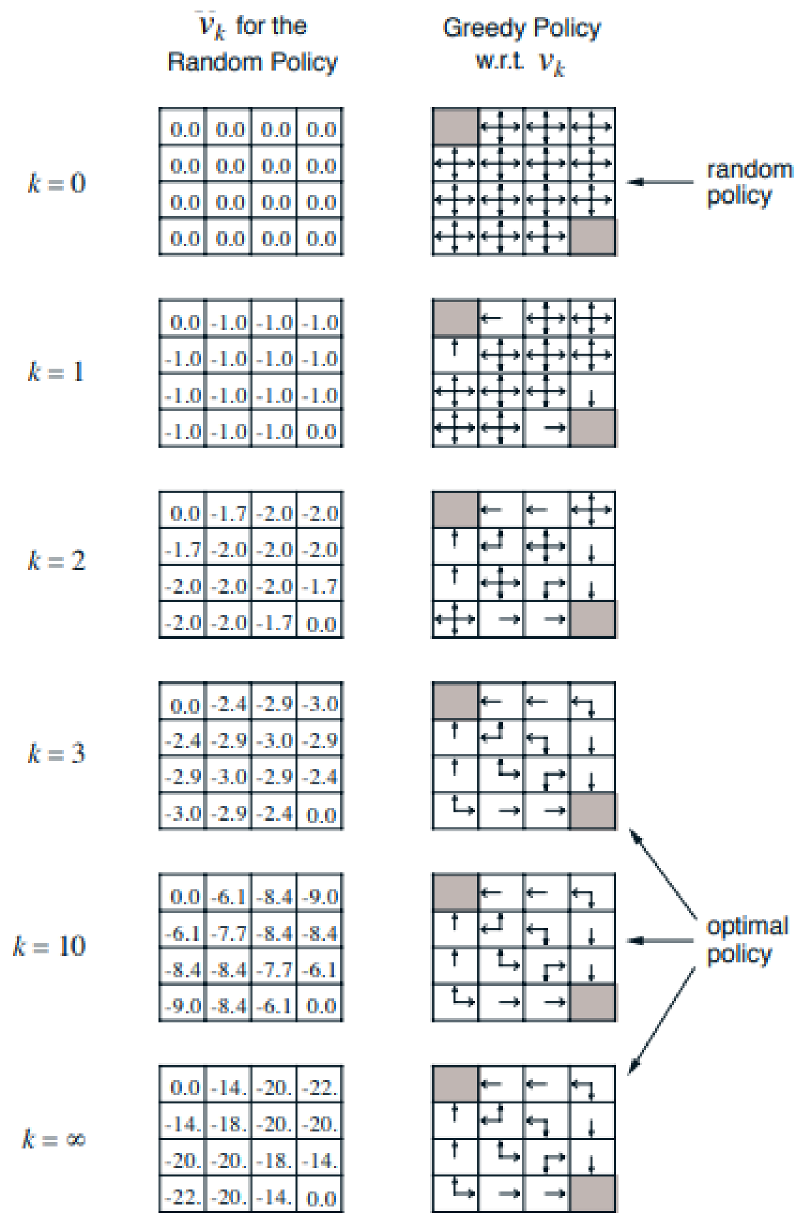
0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0

V'

0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0

GRID WORLD PROBLEM

- The left column is the sequence of approximations of the state-value function for the random policy (all actions equal).
- The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum).
- The last policy is guaranteed only to be an improvement over the random policy, but in this case policy in third iteration and all policies after the third iteration, are optimal.



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

POLICY IMPROVEMENT

- We saw how dynamic programming can be used to iteratively evaluate a policy.
- This was the first step towards the control task, or the goal is to improve a policy.
- We need to understand the policy improvement theorem, and how it can be used to construct improved policies, and use the value function for a policy to produce a better policy

POLICY IMPROVEMENT

- We have determined the value function vπ for an arbitrary deterministic policy π.
- For some state s we would like to know whether or not we should change the policy to deterministically choose an action a ≠ π(s).
- We know how good it is to follow the current policy from s, that is vπ(s)- but would it be better or worse to change to the new policy?
- One way to answer this question is to consider selecting a in s and thereafter following the existing policy, π.
- The value of this way of behaving is :

$$
\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big].
\end{aligned}
$$

POLICY IMPROVEMENT

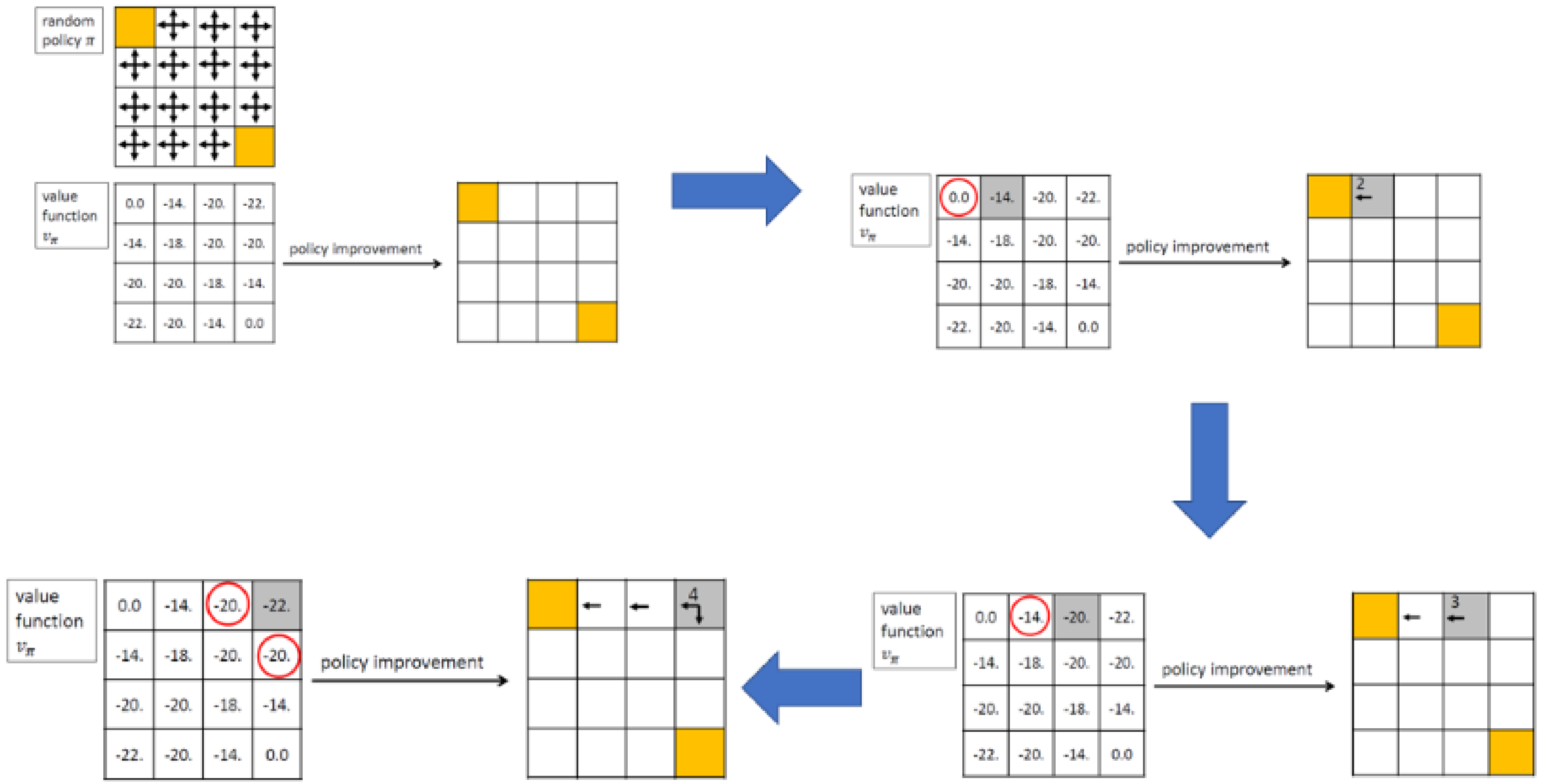
- If this happens to be greater than the value function $v_\pi(s)$, it implies that the new policy π' would be better to take.
- We do this iteratively for all states to find the best policy.
- In this case, the agent would be following a greedy policy in the sense that it is looking only one step ahead.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \end{aligned}$$

POLICY IMPROVEMENT

- Let's see example of grid world.
- Using $v_{\pi,}$ the value function obtained for random policy π, we can improve upon π by following the path of highest value.
- We start with an arbitrary policy, and for each state one step look-ahead is done to find the action leading to the state with the highest value.
- This is done successively for each state.

POLICY IMPROVEMENT



0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

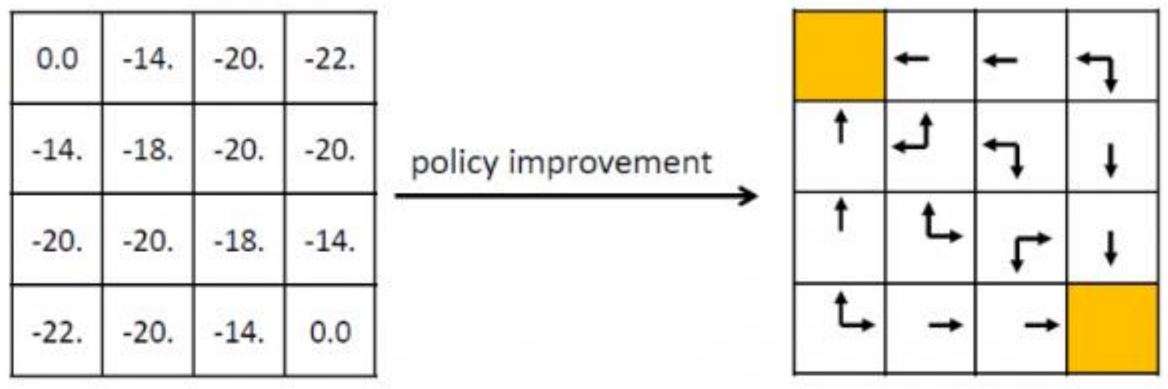
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

POLICY IMPROVEMENT

Overall, after the policy improvement step using Vπ, we get the new policy π':



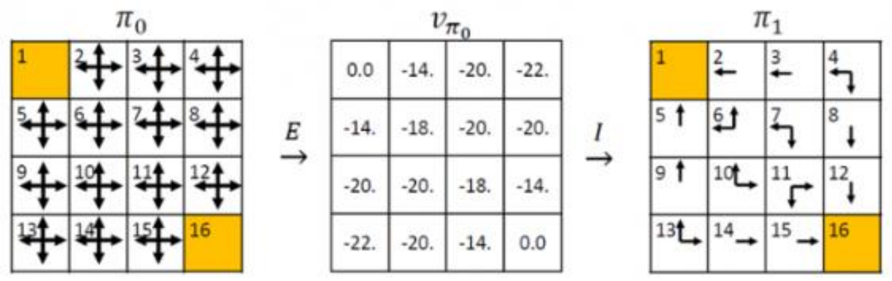
0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Looking at the new policy, it is clear that it's much better than the random policy. However, we should calculate Vπ' using the policy evaluation technique

POLICY ITERATION

- Policy Iteration: Policy Evaluation + Policy Improvement
- Once the policy has been improved using v_π to yield a better policy π', we can then compute v_π' to improve it further to π''. Repeated iterations are done to converge approximately to the true value function for a given policy π (policy evaluation). Improving the policy as described in the policy improvement section is called policy iteration.

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

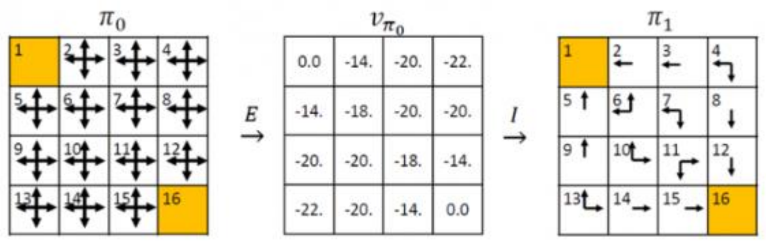


0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

E denotes policy evaluation and I denotes policy Iteration

POLICY IMPROVEMENT

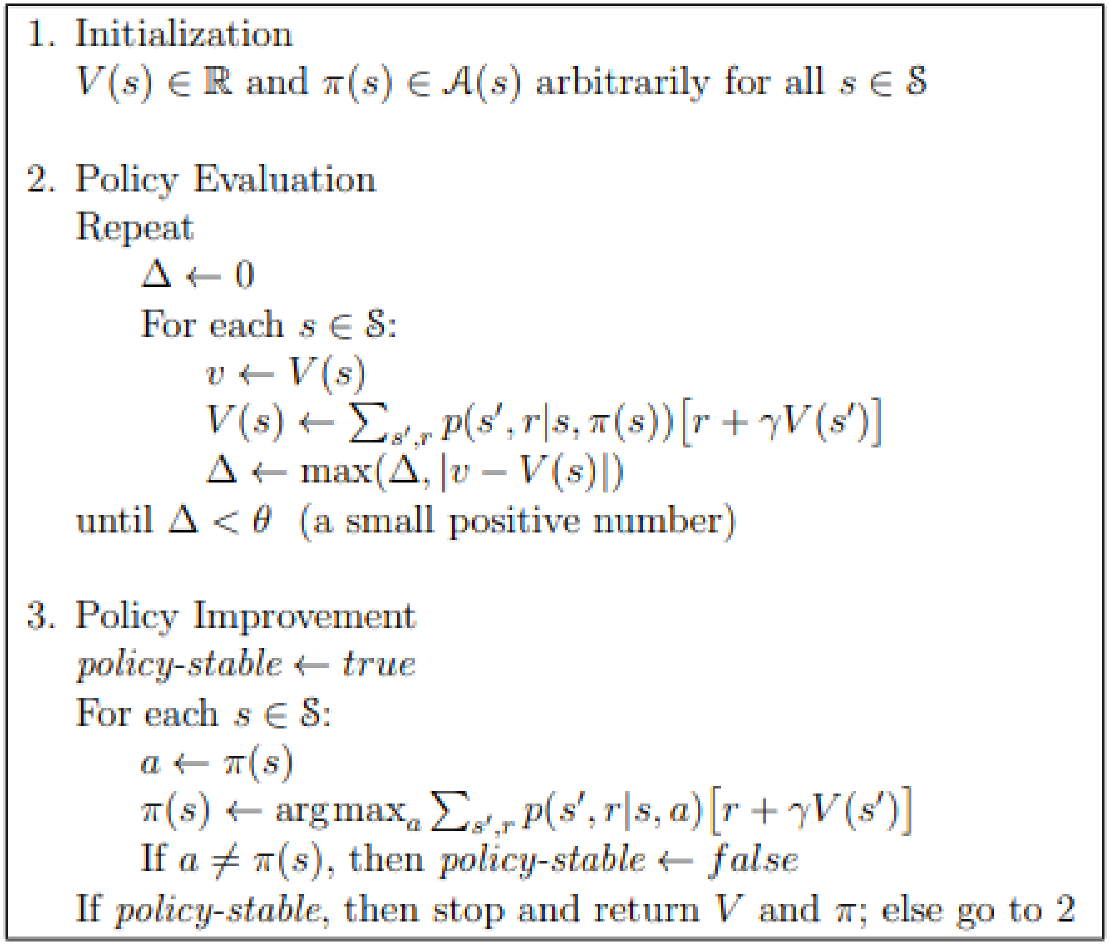
$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$



0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

- In this way, the new policy is sure to be an improvement over the previous one and given enough iterations, it will return the optimal policy.

Policy Iteration: Policy Evaluation + Policy Improvement



```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
        $\Delta \leftarrow 0$
        For each $s \in \mathcal{S}$:
             $v \leftarrow V(s)$
             $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
             $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
        $a \leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V$ and $\pi$; else go to 2
```

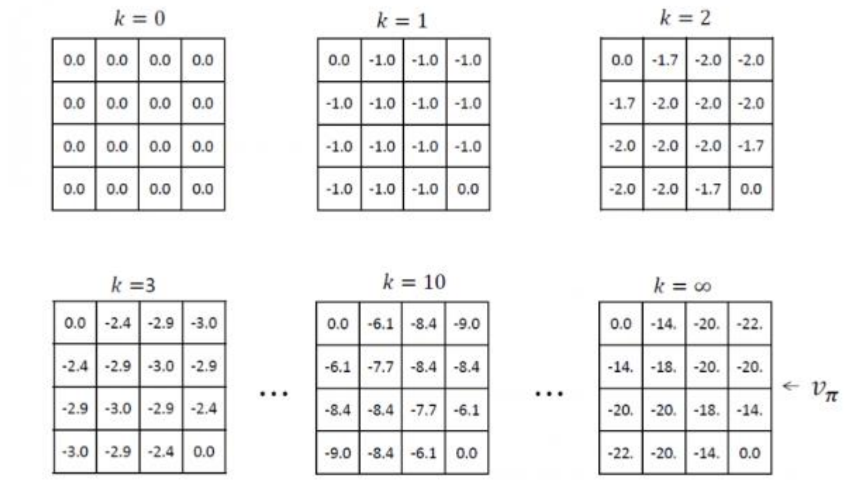
DRAWBACK OF POLICY ITERATION

- One drawback to policy iteration is that each of its iterations involves policy evaluation
- The policy evaluation may itself be a protracted iterative computation requiring multiple sweeps through all the states.

- If policy evaluation is done iteratively, then convergence exactly to $v\pi$ occurs only in the limit.

- Must we wait for exact convergence, or can we stop short of that?
- The solution for this is Value Iteration

VALUE ITERATION



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

…

$k = 10$

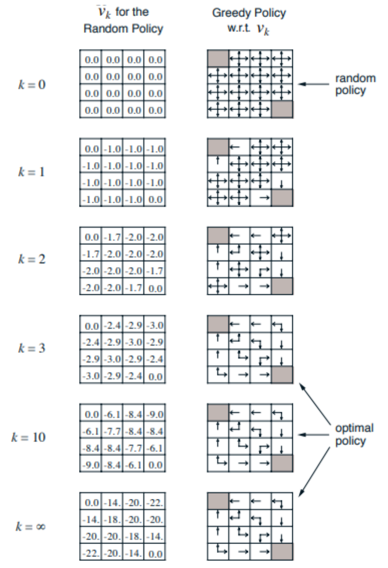
0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

…

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

$\leftarrow v_\pi$



- We can see here that at around k = 10, we were already in a position to find the optimal policy.
- So, instead of waiting for the policy evaluation step to converge exactly to the value function **V**π, we could stop earlier.
- We can also get the optimal policy with just 1 step of policy evaluation followed by updating the value function repeatedly (but this time with the updates derived from bellman optimality equation).

VALUE ITERATION

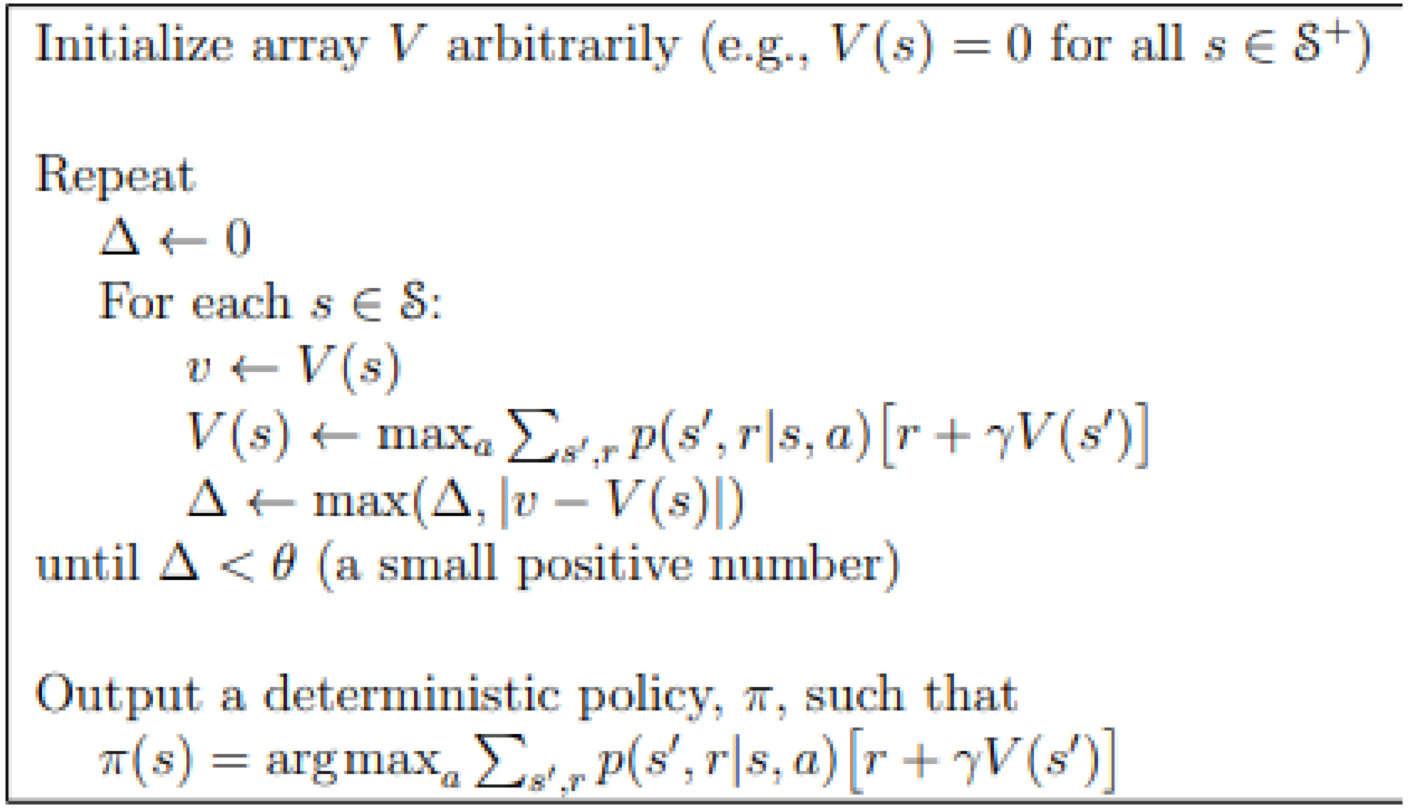
$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$

- Let's see how this is done as a simple backup operation:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big], \end{aligned}$$

- This is identical to the bellman update in policy evaluation, with the difference being that we are taking the maximum over all actions.
- Once the updates are small enough, we can take the value function obtained as final and estimate the optimal policy corresponding to that.

VALUE ITERATION

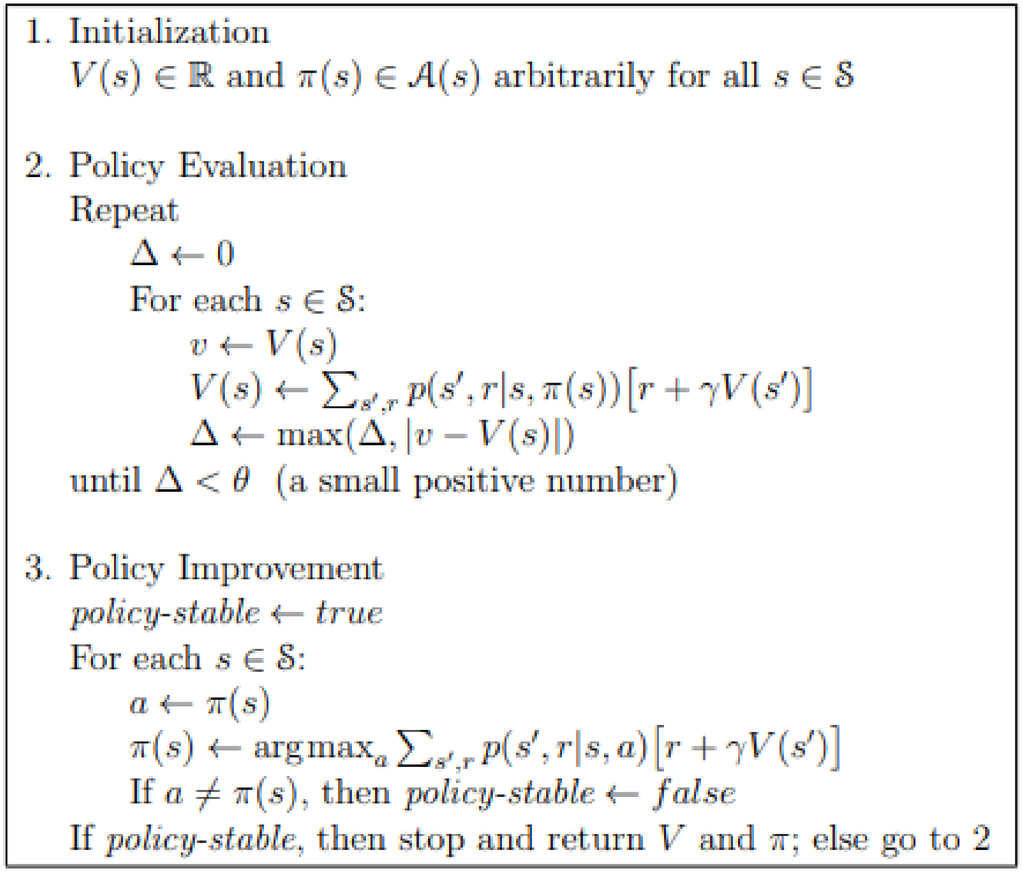


```
Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
```

Value Iteration Algorithm



```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
       $a \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
       If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V$ and $\pi$; else go to 2
```

Policy Iteration Algorithm

RELATION BETWEEN VALUE ITERATION AND POLICY EVALUATION

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

Bellman Estimation equation

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

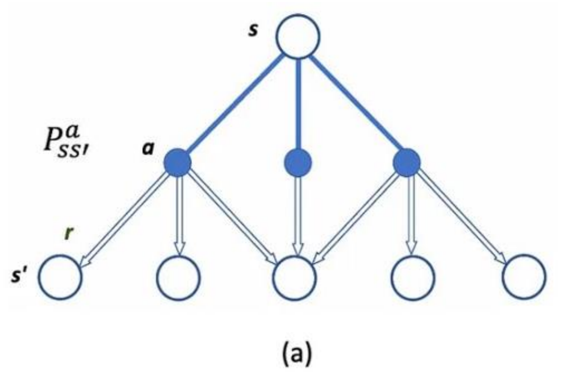
Bellman Optimality equation

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)[r + \gamma v_k(s')]$$

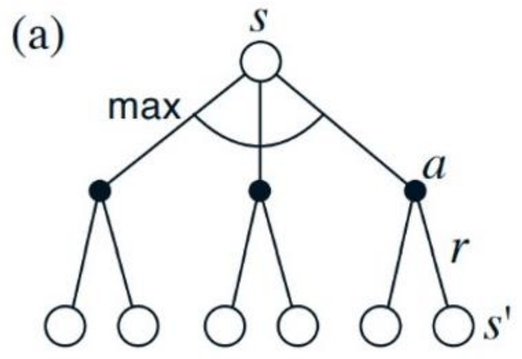
Bellman update equation for policy evaluation

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right], \end{aligned}$$

Bellman update equation for Value Iteration



Backup diagram for policy evaluation



Backup diagram for value iteration

These two are the natural backup operations for computing vπ and v∗.

GAMBLER'S PROBLEM SOLUTION

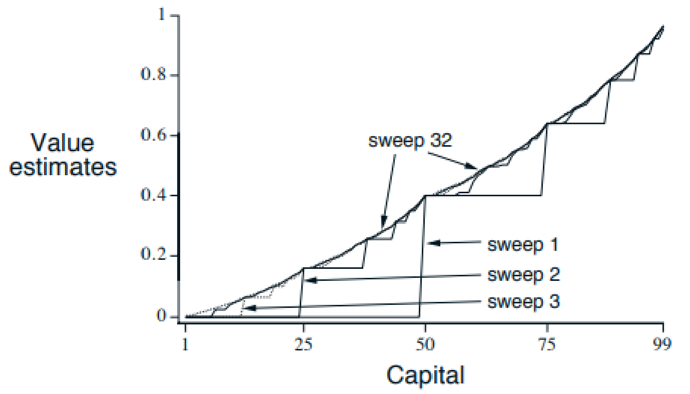
- Our agent is playing a game where they can place a bet on whether a coin flip will show heads.
- If it is heads, then they win the same amount they bet, otherwise, the opposite happens, and they lose all the money they bet.
- Let's assume that the coin lands on heads 40% of the time. The game continues over and over again until the player either has 0 capital (loss) or 100+ capital (win).

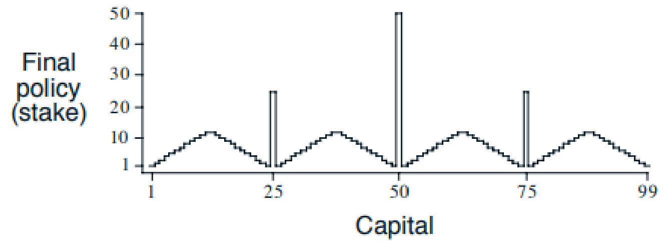
GAMBLER'S PROBLEM SOLUTION

Key points of the problem:

- Undiscounted, finite, episodic MDP: Therefore, gamma is 1, and there is a terminal state.
- The state is the gamblers capital, $s \in \{1, 2, \ldots, 99\}$
- The actions are stakes or bets, $a \in \{0, 1, \ldots, \min(s, 100 - s)\}$
- Policy is mapping of levels of capital to how much the agent should bet
- The optimal policy maximizes the probability of reaching the goal.
- Terminal states: 0 capital and 100+ capital.
- Reward of 0 at all states except the 100+ state.

GAMBLER'S PROBLEM SOLUTION





The solution to the gambler's problem for ph = 0.4. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy

KEY POINTS FOR DYNAMIC PROGRAMMING

- DP can only be used if the model of the environment is known.
- Has a very high computational expense, i.e., it does not scale well as the number of states increase to a large number.
- An alternative called asynchronous dynamic programming helps to resolve this issue to some extent.

ASYNCHRONOUS DYNAMIC PROGRAMMING

- A major drawback to the DP methods that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set.
- If the state set is very large, then even a single sweep can be prohibitively expensive.

ASYNCHRONOUS DYNAMIC PROGRAMMING

- Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set.
- These algorithms back up the values of states in any order whatsoever, using whatever values of other states happen to be available.
- The values of some states may be backed up several times before the values of others are backed up once.
- To converge correctly, however, an asynchronous algorithm must continue to backup the values of all the states: it can't ignore any state after some point in the computation.
- Asynchronous DP algorithms allow great flexibility in selecting states to which backup operations are applied

ASYNCHRONOUS DYNAMIC PROGRAMMING

- Avoiding sweeps does not necessarily mean that we can get away with less computation.
- It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy.
- We can try to take advantage of this flexibility by selecting the states to which we apply backups so as to improve the algorithm's rate of progress.
- We can try to order the backups to let value information propagate from state to state in an efficient way.
- Some states may not need their values backed up as often as others. We might even try to skip backing up some states entirely if they are not relevant to optimal behavior

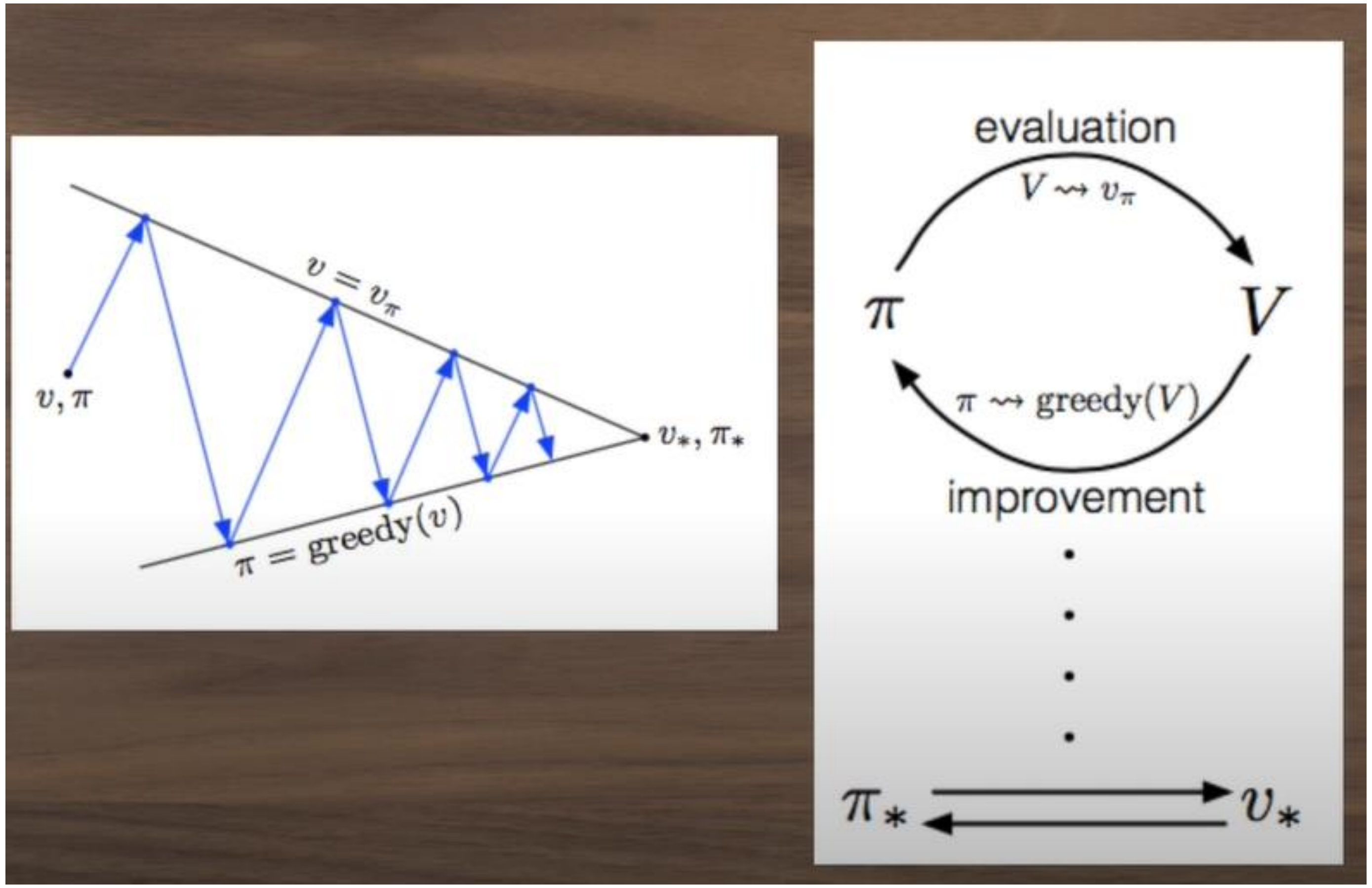
ASYNCHRONOUS DYNAMIC PROGRAMMING

- Asynchronous DP algorithms refers to the idea of updating the value estimates of only a subset of states rather than the entire set of states at every policy evaluation iteration.
- They simply update the values of states in any order, using the values of other states that are available.

ASYNCHRONOUS DYNAMIC PROGRAMMING

- Asynchronous algorithms also make it easier to intermix computation with real-time interaction.
- To solve a given MDP, we can run an iterative DP algorithm *at the same time that an agent is actually experiencing the MDP.*
- The agent's experience can be used to determine the states to which the DP algorithm applies its backups.
- At the same time, the latest value and policy information from the DP algorithm can guide the agent's decision-making.
- For example, we can apply backups to states as the agent visits them. This makes it possible to *focus* the DP algorithm's backups onto parts of the state set that are most relevant to the agent.

GENERALIZED POLICY ITERATION (GPI)

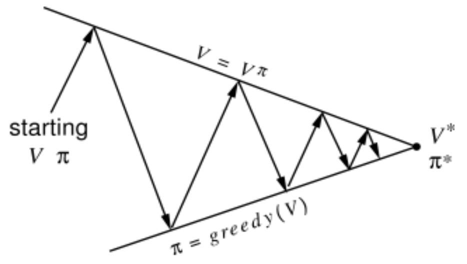


$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*,$$

GENERALIZED POLICY ITERATION (GPI)

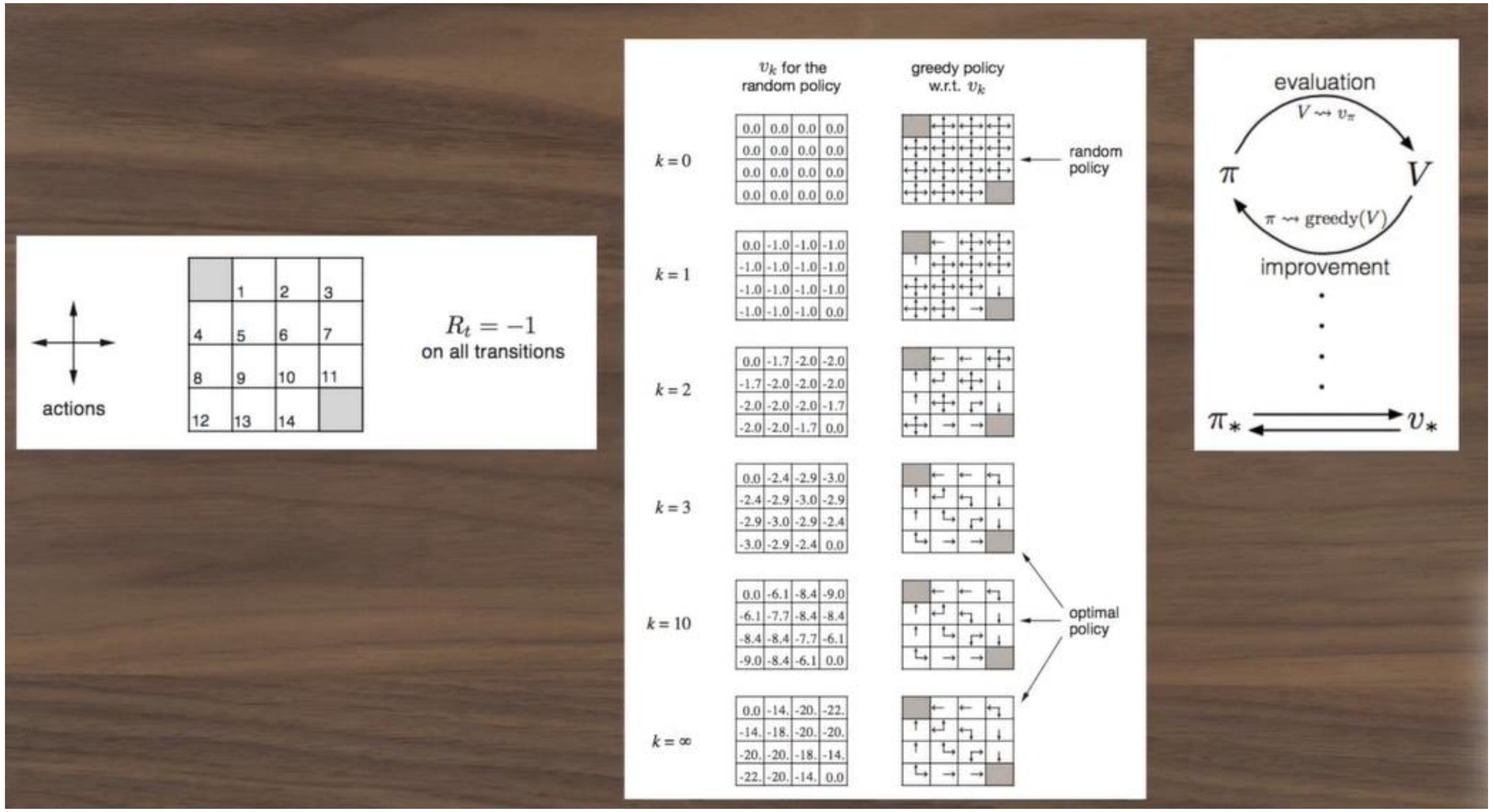
- GPI works as follows:
- We randomly initialize our value function estimates of every state and start with a random policy.
- We then evaluate the values of every state using this policy.
- And we update the policy by making greedy action choices, with respect to the value functions (i.e. taking the action that moves you to the state with the highest value). This will continue until it converges to the optimal policy and value function

GENERALIZED POLICY ITERATION (GPI)



- One might also think of the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals--for example, as two lines in two-dimensional space:
- Although the real geometry is much more complicated than this, the diagram suggests what happens in the real case.
- Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals.
- The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal.
- Inevitably, however, the joint process is brought closer to the overall goal of optimality.
- The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely.
- In GPI one could also take smaller, incomplete steps toward each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly.

GENERALIZED POLICY ITERATION (GPI) EXAMPLE



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

THANK YOU!!!