4.0 : Intro

DP (dynamic programming) = a collection of algorithms used to compute the optimal policies given a perfect MDP.

The key idea is to use value functions to organize the search for good policies.

- Dynamic : sequential or temporal component to the problem.
- Programming : optimizing a "program", i.e. a policy.

DP is used to solve problems that have these 2 properties:

- optimal structure : we can break down the problem into subproblems, and finding the optimal for each subproblem will find the optimal for the overall problem.
- overlapping subproblems : the subproblems recur many times.

Luckily, MDPs have these properties:

- Bellman equation gives recursive decomposition
- Value functions store and reuses solutions.

DP can be used in MDPs for:

- prediction : Input MDP and policy π, output value function v_π
- control : input MDP, output optimal value functions v_* and optimal policy π_*.

- Assumes full knowledge of the MDP (transition probability, etc).

DP uses the Bellman Eqs into update rules for improving approximations of the desired value functions.

Bootstrapping : making estimates from other estimates.

4.1 : Policy Evaluation (Prediction)

Inputs :
- MDP
- policy π.

Output :
- state-value function v_π.

You're given a policy π, and want to determine how good it is. This algorithm does NOT change the policy.

To do this, we turn the Bellman Eq into an iterative update. we assume we know the transition probability p, in order to calculate v_π.

To compute v_π iteratively, we chose an arbitrary value for the initial state v_0, and each successive approximation is obtained by using the Bellman eq for v_π as an update rule. This algorithm is called iterative policy evaluation :

$$\underbrace{v_{k+1}(s)}_{\text{the value of state } s \text{ at iteration } k+1} \doteq \mathbb{E}_\pi\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s\right]$$

$$= \sum_a \underbrace{\pi(a|s)}_{\text{probability of taking action } a \text{ when in state } s} \sum_{s',r} \underbrace{p(s', r \mid s, a)}_{\text{transition probability: chance of reaching state } s' \text{ and getting reward } r, \text{ when taking action } a \text{ when in state } s.} \underbrace{\left[r + \gamma v_k(s')\right]}_{\text{immediate reward + discounted future value.}}$$

Sum over all actions

Sum over all possible "next-state" and rewards

This is guaranteed to terminate if $\gamma < 1$ or policy is episodic and has terminal state.

Notation : Here the subscript k in v_k denotes the iteration number for the current computation of v.

Explanation of the algorithm:

- At each iteration $k+1$
- For all states $s \in S$ (sweep over all states)
- update $v_{k+1}(s)$ from $v_k(s')$
  - where s' is a successor state of s

$v_k = v_\pi$ is a fixed point for this update rule. The sequence of v_k can be shown to converge to v_π as $k \to \infty$.

To produce each approximation v_{k+1} from v_k, we apply the same operation to each state s: we replace the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along with the one-step transitions possible under the policy being evaluated. This is called an expected update.

- Each iteration updates the value of every state once to produce v_{k+1}.
- updates are called expected updates bc they rely on an expectation over all possible next states (rather than a sample next state).

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$: // loop thru each state
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$ ← next state
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

4.2: Policy Improvement

- Previously: Given policy π, we determined the associated value function V_π using policy evaluation.
- For some state s we would like to know if we should select action $a \neq \pi(s)$, a better action than the one currently advised by our policy.
- One way to do this is to select a, and thereafter following the existing policy π:

$$q_\pi(s,a) \doteq E[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a]$$
$$= \sum_{s',r} p(s', r \mid s, a)[r + \gamma V_\pi(s')]$$

- If this quantity is greater than $V_\pi(s)$, then it's better to take action a then follow π, rather than follow π all the time.
- This is a special case of the policy improvement theorem.

Policy Improvement Theorem:

Let π and π' be two deterministic policies s.t. $\forall s \in S$:

$$q_\pi(s, \underbrace{\pi'(s)}_{\text{new action}}) \geq V_\pi(s)$$

Then the policy π' is as good or better than π, meaning the expected return is greater or equal for all states:

$$V_{\pi'}(s) \geq V_\pi(s).$$

Proof is in the book.

There are 2 ways of doing this (policy improvement):

1) policy iteration,
2) value iteration

Policy Improvement:

So far, we saw how to evaluate a change in policy in a single state on a particular action. The natural extension to this is to consider all states and all actions. In the greedy policy π', we select at each state the action that appears best according to $q_\pi(s, a)$:

$$\pi'(s) \doteq \arg\max_a q_\pi(s, a) \quad \text{// returns the action that maximize } q_\pi$$

$$= \arg\max_a \mathbb{E}[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a]$$

$$= \arg\max_a \sum_{s', r} p(s', r \mid s, a)[r + \gamma V_\pi(s')]$$

- we take the action that looks best in the short term (one step lookahead) according to V_π.
- If the new policy π' is equally good but not better than π, then we have reached the optimal policy.

4.3: Policy Iteration

Purpose is to find the optimal policy (maximizes rewards).

1) Take policy π_n (meaning the policy at iteration n)
2) Compute V_{π_n}
3) Use V_{π_n} to compute better policy π_{n+1}
4) Repeat until convergence.

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

Perks:

- Each policy is guaranteed to be a strict improvement of the previous one,
- This process converges to an optimal policy and optimal value function in a finite number of steps (for finite MDPs).

Implementation:

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

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

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

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

Policy iteration for action values:

1. **Initialization**

   $Q(s, a) \in \mathbb{R}$ and $\pi(s) \in A(s)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$

2. **Policy Evaluation**

   Loop:
   $\Delta \leftarrow 0$ Loop for each (s, a) pair:
   $q \leftarrow Q(s, a)$
   $Q(s, a) \leftarrow \sum_{s', r} p(s', r|s, a)[r + \gamma \sum_{a'} Q(s', a')]$
   $\Delta \leftarrow \mathbf{max}(\Delta, |q - Q(s, a)|)$
   until $\Delta < \theta$ some threshold

3. **Policy Iteration**

   is_stable $\leftarrow$ true for each (s, a) pair:
   $p \leftarrow \pi(s, a)$ $\pi(s, a) \leftarrow \mathbf{argmax}_{s,a} Q(s, a)$ if $p \neq \pi(s, a)$ then
   is_stable $\leftarrow$ false
   if is_stable then stop else goto 2.

4.4: Value Iteration

TLDR: we're trying to find optimal policy using policy iteration with one sweep of policy evaluation.

Optimal Policy (to build intuition about why value iteration works):
Any optimal policy can be subdivided into 2 components:
- An optimal first action A_*
- Followed by an optimal policy from successor state s'

Here we break down a definition of optimal policy in terms of finding an optimal policy from all the states that we can end up in.

Principle of optimality applied to policies: A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, iff:
- for any state s' reachable from s
- π achieves the optimal value from state s'

We're going to use this to build value iteration:
- If we know the solution for subproblems $v_*(s')$
- Then the solution $v_*(s)$ can be found by one-step lookahead:

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')]$$

- The idea of value iteration is to apply these updates iteratively.
- Start w/ final rewards and work backwards.

We know the optimal solution from the leaves and we back this up in the tree by maxing out of all the things we can do.

Value iteration:

- Every iteration of policy iteration involves policy evaluation to compute V.
- Convergence to V_π occurs only in the limit.
- value iteration: policy evaluation is stopped after one sweep.
- Simple update combining policy improvement and truncated policy evaluation:

$$V_{k+1}(s) \doteq \max_a \mathbb{E}\left[R_{t+1} + \gamma V_k(S_{t+1}) \mid S_t = s, A_t = a\right]$$
$$= \max_a \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma V_k(s')\right]$$

- we turned the Bellman Optimality Eq into an update rule.
- value iteration update is identical to policy evaluation update except here we max over all actions.
- Each intermediate construct of V doesn't correspond to any policy.

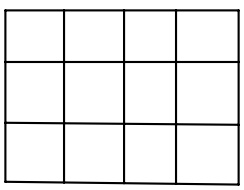
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$ //Bellman Optimality Eq for V_*.
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

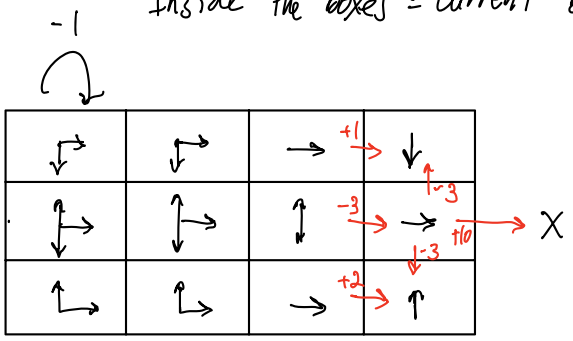
Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

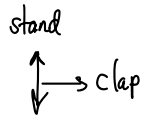
//very similar to iterative policy evaluation.



Example:

Inside the boxes = current best action(s).

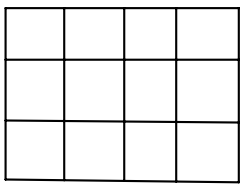




allowable actions for each box.

Policy Iteration

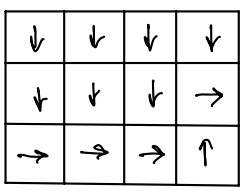
values



11 3/8	11.25	11	10
11.5	11.5	11.5	10
11 5/8	11.75	12	10

// lots of computation,

π_*



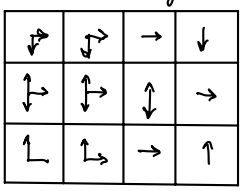
12	12	12	10
12	12	12	10
12	12	12	10

Value iteration

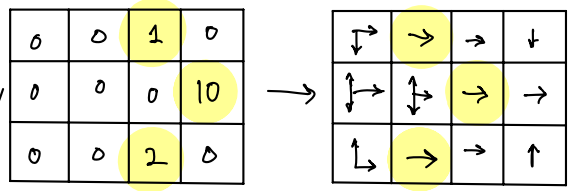
Best values

0	0	0	0
0	0	0	0
0	0	0	0

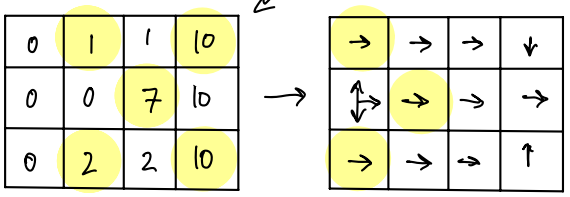
Best Policy



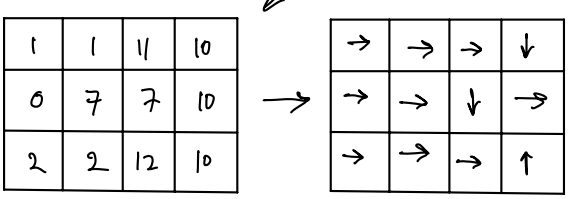
1 step lookahead, given previous



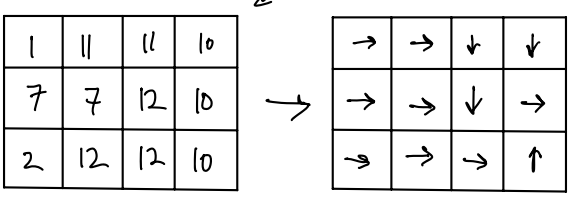
0	0	1	0
0	0	0	10
0	0	2	0



0	1	1	10
0	0	7	10
0	2	2	10

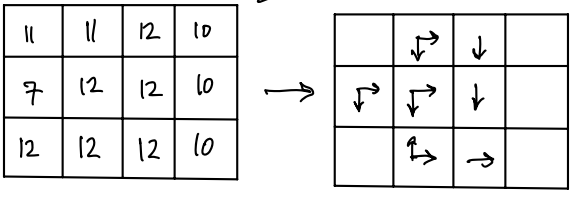


1	1	11	10
0	7	7	10
2	2	12	10



1	11	11	10
7	7	12	10
2	12	12	10

π_*

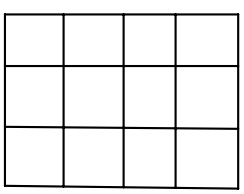


11	11	12	10
7	12	12	10
12	12	12	10

π_*

2 more iterations:

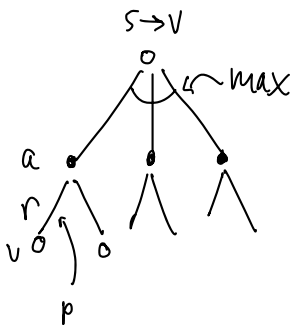
// multiple optimal policies

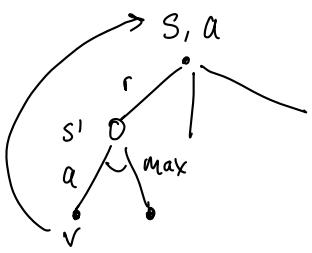


once you have U_*, do one step lookahead to obtain π_*.

4.5: Asynchronous Dynamic Programming

- Major drawback of the methods we just saw: we always have to loop over the state space. If the space is huge, then we're screwed.
- Asynchronous DP algorithms update values of states in any order, using whatever values for other states are available at the time of computation.
- The order of the states that we will update will matter more (we might even try to skip useless states).
- Asynchronicity also makes it easier to intermix computation with interaction with the environment (which can guide the selection of states that we'll update, for example, we focus on states that we actually visit.)
- Prioritize the selection of which state to update based on the magnitude in the Bellman Eq error (more error: more priority).





4.6: Generalized Policy Iteration (GPI)

- Process 1: policy evaluation. Make the current value function consistent with the current policy.
- Process 2: policy improvement. Make the policy greedy wrt the current value function.
- In policy iteration, these two processes alternate,
- In value iteration, they don't really alternate, policy improvement only waits for one iteration of the policy evaluation.
- In asynchronous DP, the two processes are even more interweaved,
- Generalized policy iteration: let policy evaluation and policy improvement interact, independent of granularity. When they stabilize, we have reached optimality.

Why? The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy wrt the current value function. Thus, both processes stabilize only when a policy has been found that is greedy wrt its own evaluation function. This implies that the Bellman optimality Eq holds, and thus that the policy and the value function are optimal.

If improvement stops,

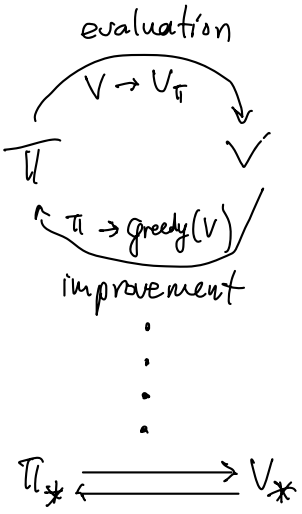
$$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

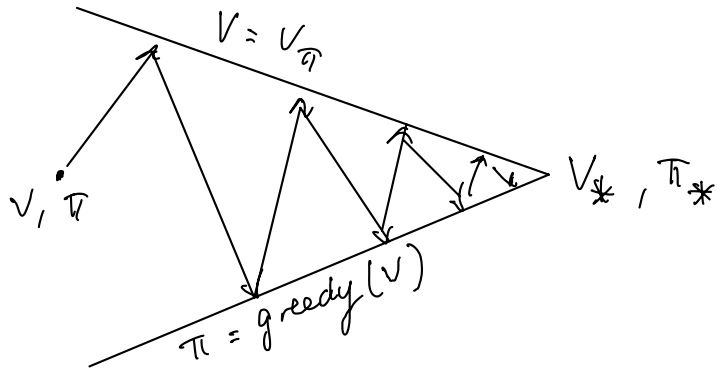
Then the Bellman Eq has been satisfied:

$$V_\pi(s) = \max_{a \in A} q_\pi(s,a)$$

Both processes are:

- cooperating towards the same goal: optimality.
- competing because making policy greedy wrt the value function typically makes the value function incorrect for the new policy.





4.7: Efficiency of Dynamic Programming

- If n and k are the number of states and actions, then there are k^n policies.
- DP methods have polynomial time, which is better than other methods.
- Policy iteration and value iteration can be used up to a fair # of states, and asynchronous DP even more.

4.8: Summary

- Policy evaluation: refers to the (typically) iterative computation of the value function given a policy.
- Policy improvement: refers to the computation of an improved policy given the value function of the current policy.
- Putting these two together gives policy iteration and value iteration.

- Policy Iteration: has two distinct phases that alternate:
  1) Policy Evaluation: completely evaluate the current policy by sweeping thru the entire state space, until value function converges.
  2) Policy Improvement: update the policy based on the converged value.

- Value Iteration: combines policy evaluation and improvement into a single step. It performs one update of the value function in each state, and then use it to implicitly improve/change the policy.

- DP methods sweep thru the state space, performing an expected update operation for each state.
- Expected updates: are Bellman Eqs turned into assignments to update the value of a state based on the values of all possible successor states, weighted by their probability of occurring.
- GPI: refers to the interweaving of policy evaluation and policy improvement to reach convergence.
- Asynchronous DP frees us from the complete state-space sweep.

Problem	Bellman equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative policy evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

Algorithms based on state-value functions $(v_\pi(s), v_*(s))$ have $O(mn^2)$, m actions and n states.

Action value functions $(q_\pi(s,a), q_*(s,a))$: $O(m^2n^2)$.