Lecture III : Planning by Dynamic Programming

27/08/19

Dynamic Programming :

- dynamic : sequential / temporal component to the problem
- programming : optimising a "program" (a policy)
- helps solve complex problems
- breaks problem into subproblems
  - solves the subproblems
  - combine solutions of subproblems

DP is a general method of solving problems which have :
- optimal substructure (optimal solution can be decomposed into subproblems)
- overlapping sub problems (subproblems recur many times)

MDPs satisfy both of these properties
- Bellman equation gives the recursive decomposition
- Value function stores and reuses solutions

DP assumes full knowledge of the MDP
- it used for planning in an MDP
  - prediction : input : $MDP\langle S, A, P, R, \gamma \rangle$ and policy π
    output : value function v_π
  - control : input : $MDP\langle S, A, P, R, \gamma \rangle$
    output : optimal value function v_* and optimal policy π_*

DP solves problems like :

- scheduling algorithms
- string algorithms (eg : sequence alignment)
- graph algorithms (eg : shortest path)
- graphical models (eg : Viterbi algorithm)
- bioinformatics (eg : lattice models)

Policy Evaluation :

problem : evaluate a given policy π

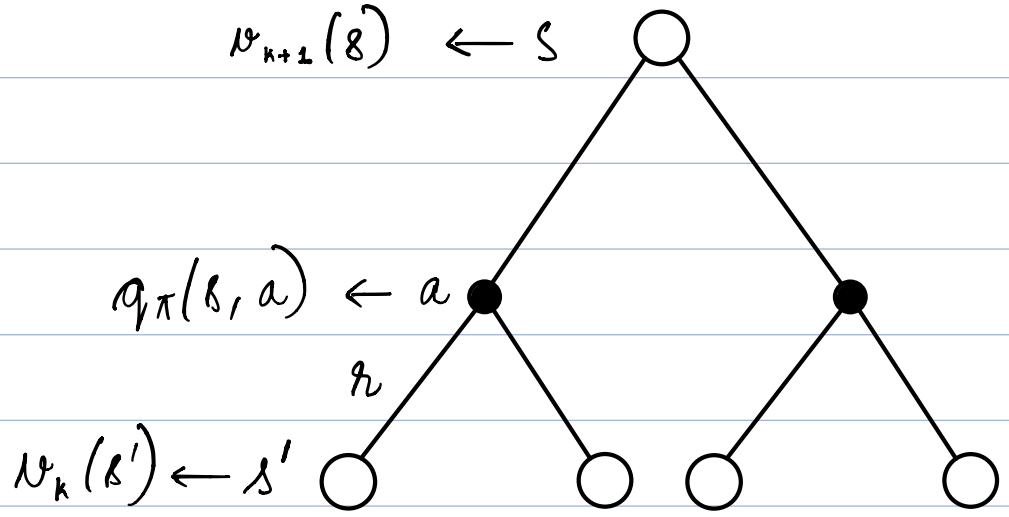
solution : iterative application of Bellman expectation backup

loop : $v_1 \longrightarrow v_2 \longrightarrow \cdots v_\pi$

synchonous backups :

- at each iteration $k+1$
- for all state $s \in S$
- update $v_{k+1}(s)$ from $v_k(s')$
- where s' is succesor of s

{ there is also asynchronous backup }

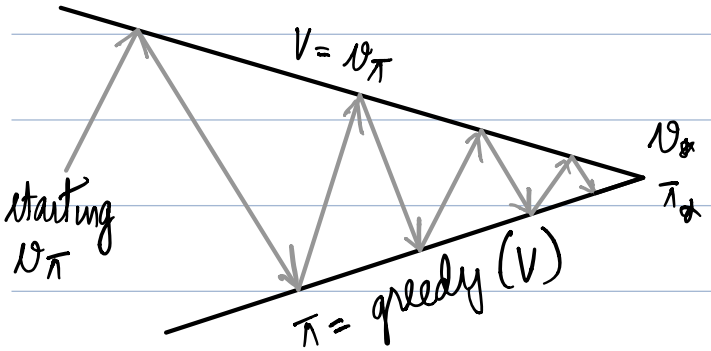


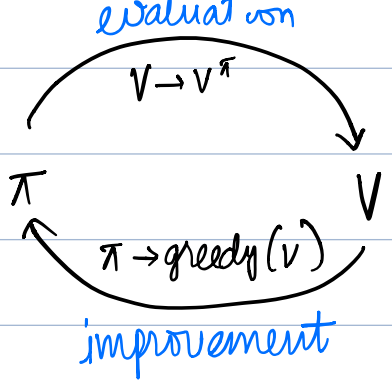
$$\therefore v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right)$$

$$\text{also,} \quad V_{k+1} = R^\pi + \gamma P^\pi V_k$$

Policy Iteration :

- find the best policy for the MDP
- given a policy π
  - evaluate the policy
    $$v_\pi(s) = E[G_t \mid S_t = s]$$
  - improve policy by acting greedily wrt. v_π
    $$\pi' = \text{greedy}(v_\pi)$$
- policy iteration always converges to π_*





- let "a" be a deterministic policy, $a = \pi(s)$
- the policy is improved by acting greedily

$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}}\ q_\pi(s,a)$$

- this improves the value for one step from any state s

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

- if the improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$
- when the improvements stop, the equalities hold
  - ↳ Bellman optimality equation is satisfied
    - ↳ thus π must be optimal

Modified Policy Iteration:

- having a stopping criterion
- do k steps of policy evaluation and then improvement

Value Iteration:

optimal policy can be subdivided into two parts:
- an optimal first action A^*
- followed by an optimal policy from successor state s'

Principle of Optimality: $\pi(a|s)$ is optimal iff $\forall s' \in \text{successor}(s)$, $v_\pi(s) = v_*(s)$, where v_* is the optimal value function

Algorithm:

- Lets assume we know the solution to subproblems $v_*(s')$
- now using one step lookahead for $v_*(s)$

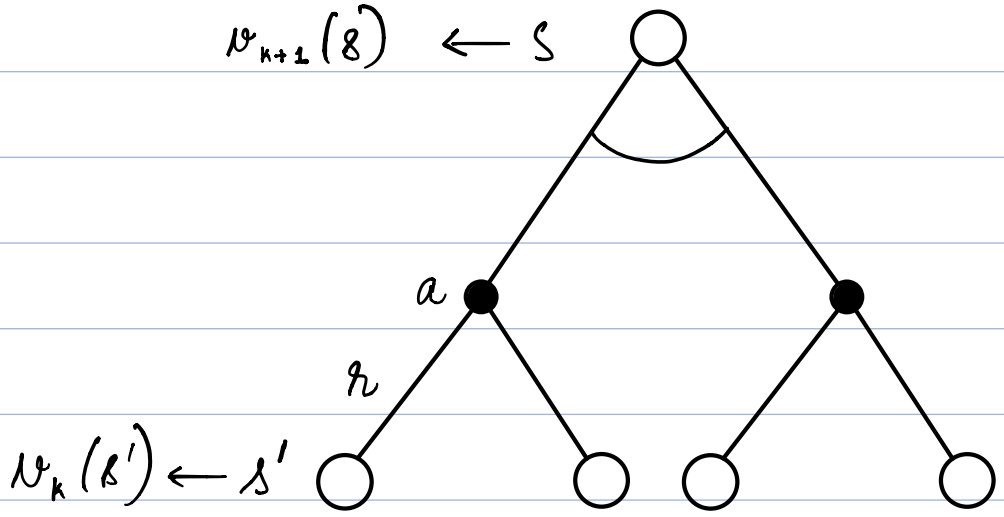
$$v_*(s) = \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$$

- value iteration applies this update rule iteratively
- intuition : start with final rewards and work backwards

So, summarizing value iteration:

- problem : find optimal policy π
- solution: iterative application of Bellman optimality backup
- using synchronous backups :
  - at each iteration $k+1$
  - for all states $s \in S$
  - update $v_{k+1}(s)$ from $v_k(s')$
- converges to v_*
- no explicit policy unlike policy iteration
- $v_1 \rightarrow v_2 \rightarrow \ldots \ldots v_*$
- intermediate value functions may not correspond to any policy

This is equivalent to doing modified policy iteration with $k = 1$.



$$\therefore v_{k+1}(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right)$$

$$\text{also, } V_{k+1} = \max_{a \in A} \left(R^a + \gamma P^a V_k \right)$$

Summary :

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

Complexity of algorithms based on state value functions
— $O(mn^2)$; m actions, n states per iteration

Complexity of algorithms based on action value functions
— $O(m^2n^2)$; m action, n states per iteration

Extensions to DP:

- Asynchronous DP backs up states individually in any order
- Can significantly reduce computation
- Guaranteed to converge
- in-place, prioritised sweeping, real time DP

- in place : use latest values always, more efficient
- prioritised sweeping :
  - use magnitude of Bellman error to guide state selection

$$\left| \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right) - v(s) \right|$$

  - use a priority queue
  - requires knowledge of reverse dynamics (preceding states)
- real-time DP:
  - use states that are relevant to agent
  - use agent's experience to guide selection of states
  - after each time-step S_t, A_t, R_{t+1}
  - backup the state S_t

$$v(S_t) = \max_{a \in A} \left(R_{S_t}^a + \gamma \sum_{s' \in S} P_{S_t s'}^a v(s') \right)$$

  - collect samples from a real trajectory, say from a simulation

- DP uses full-width backups
  - considers the whole branching factor
  - very computationally expensive
  - this also requires environment dynamics
  - to solve this, we sample a trajectory instead
    - sample backups
    - model is not required as we are sampling