Lecture 4:
Reinforcement Learning: Planning

ECS7002P - Artificial Intelligence in Games

Diego Perez Liebana - diego.perez@qmul.ac.uk
Office: CS.301

https://gaigresearch.github.io/

Queen Mary University of London

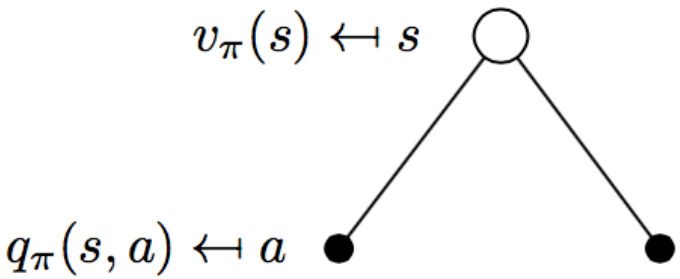
Outline

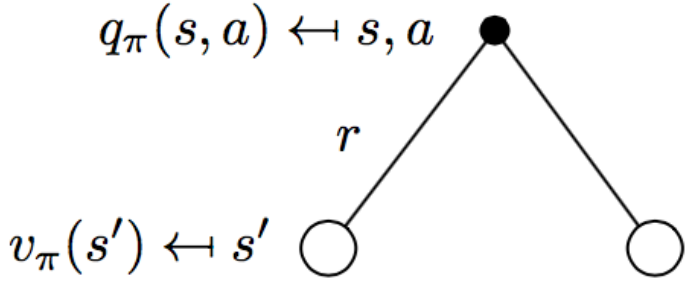
Dynamic Programming

Monte Carlo Methods

Temporal Difference Learning

Key reminders

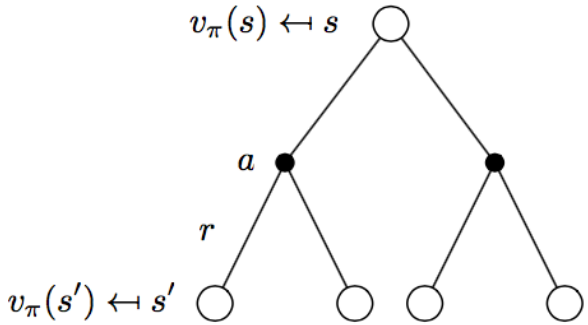


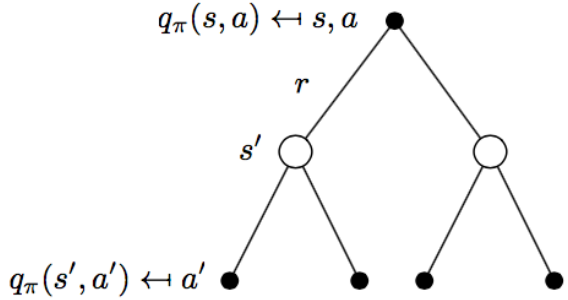


$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

Key reminders: Bellman Expectation Equation

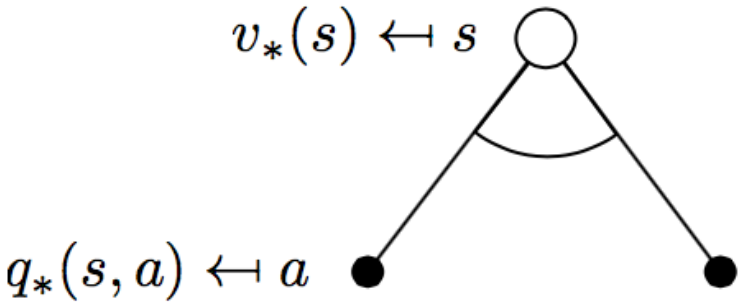


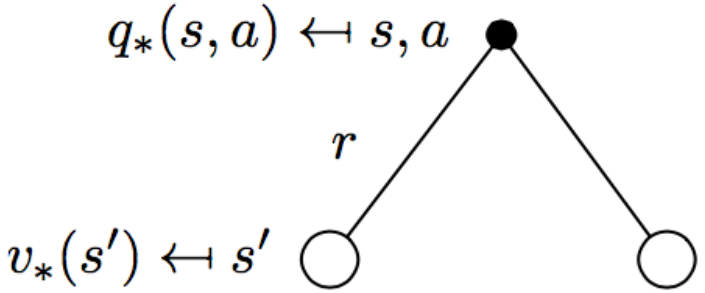


$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

Key reminders

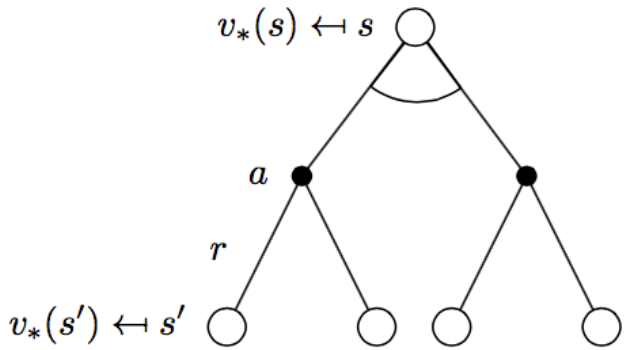


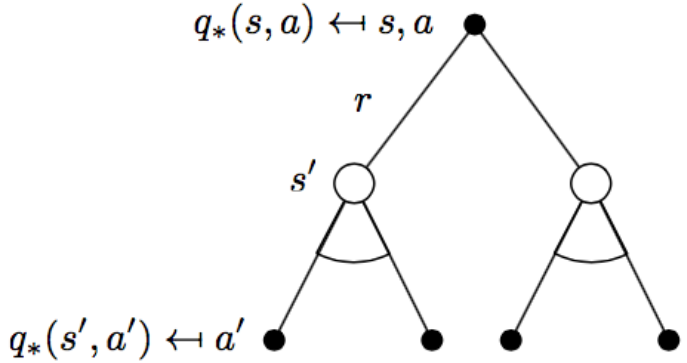


$$v_*(s) = \max_a q_*(s,a)$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Key reminders: Bellman Optimality Equation





$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

Reinforcement Learning: Planning

Dynamic Programming

Dynamic Programming (DP)

What is Dynamic Programming?

- Dynamic: sequential or temporal component of the problem.
- Programming: optimizing a program (policy).

Dynamic Programming (DP) solves problems by decomposing them into sub-problems that can be solved separately. DP works successfully in problems that have two properties:

- Optimal substructure:
  - Principle of optimality: the optimal solution can be decomposed into sub-problems.
  - In MDPs, this is satisfied by the *Bellman Optimality Equation*.
- Overlapping sub-problems:
  - Sub-problems may occur many times, and solutions can be cached and reused.
  - In MDPs, this is satisfied by information in the *value function* $v(s)$.

$\rightarrow$ DP can be used to solve MDPs . . . assuming **full knowledge** of the MDP! (this is, assuming we know $P^a_{ss'}$)

Iterative Policy Evaluation

Iterative Policy Evaluation is a DP algorithm that evaluates a given policy π (calculates the value of $v_\pi(s)$ for all states s). It performs a **prediction**: estimates how good is to follow a policy π in a given MDP.

We use the Bellman Expectation Equation. Remember:

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

where:

- $\pi(a \mid s)$ is the probability of taking action a in state s, under policy π.
- R_s^a is the reward obtained in state s after applying action a.
- γ is the discount factor.
- $p(s' \mid s, a)$ is the probability of transiting from state s to s' when applying action a. This is determined by nature (the environment of the MDP).

Iterative Policy Evaluation

Main idea: start with a set of values $v(s)$ (initialized at random, or $= 0$) for every $s \in S$ and apply the Bellman Expectation equation iteratively.

Iterative Policy Evaluation uses *synchronous* back-ups. This is, on each iteration, all states of the MDP are considered for updating $v(s)$ ($\forall s \in S$).

The algorithm works as follows:

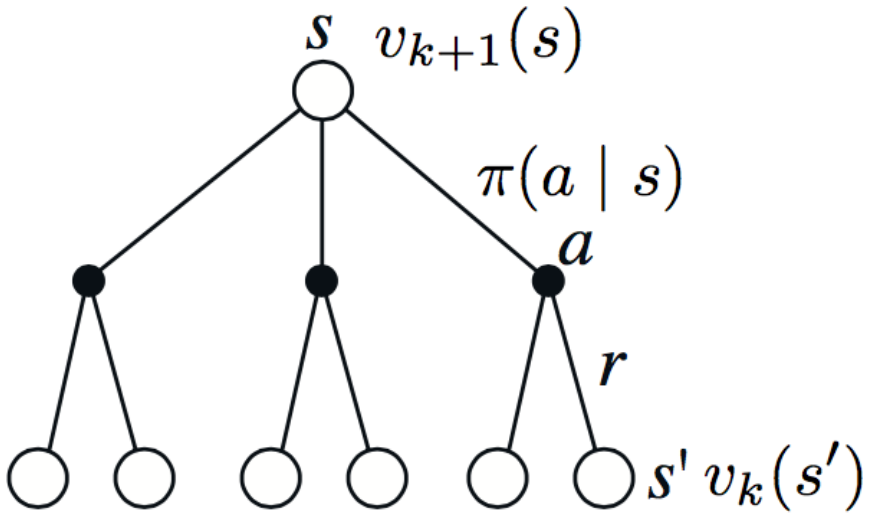
1. For all states $s \in S$, initialize $v(s)$ to a random value.
2. For each iteration k:
   2.1 For all states $s \in S$, update $v_{k+1}(s)$ from $v_k(s')$, where s' are all successors of s, using Bellman Expectation Equation:

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \Big(R_s^a + \gamma \sum_{s'} p(s' \mid s, a) v_k(s') \Big)$$

Convergence to the *true value* of v_π is guaranteed, as long as $\gamma < 1$, for all states s under policy π when $k \to \infty$.

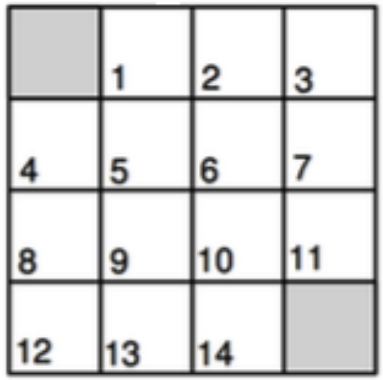
Iterative Policy Evaluation

Iterative Policy Evaluation performs a **full backup**: replaces the old value of $v(s)$ with the new value obtained from the Bellman Expectation Equation. This is moving one step further to the future.



$$v_{k+1}(s) = \sum_a \pi(a \mid s) \left(R_s^a + \gamma \sum_{s'} p(s' \mid s, a) v_k(s') \right)$$

Example: the Small Gridworld



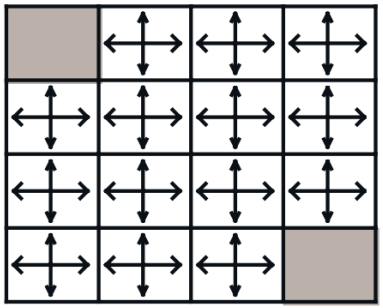
- States:
  - Non-terminal: $1, 2, \ldots, 14$.
  - Terminal: Grey cells.
- Actions:
  - Available: *Up*, *Down*, *Left*, *Right*.
  - Actions that would take the agent off grid, leave the state unchanged.
- Reward is -1 when exiting all states.
- Undiscounted task ($\gamma = 1$).
- Policy:
  - Equi-probable random policy for all $s \in S$.
  - $\pi(a \mid s) = \frac{1}{4}$
  - **We are evaluating the random policy.**
  - **Calculate** $v_\pi(s)$ for all $s \in S$?

Iterative Policy Evaluation for the Small Gridworld

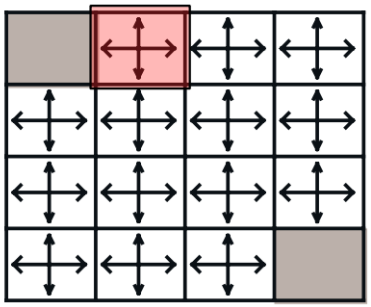
- $k = 0$; For all states $s \in S$, initialize $v(s)$ to 0 or a random value:

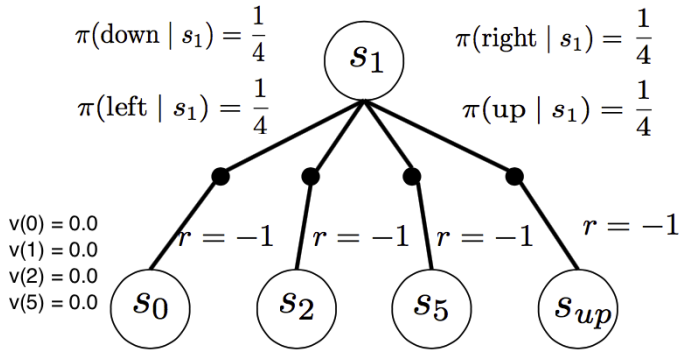
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Random policy:



- $k = 1$; For all states $s \in S$, update $v_{k+1}(s)$ from $v_k(s')$, where s' are all successors of s, using Bellman Expectation Equation.





$$v_1(s_1) = \sum_a \pi(a \mid s)\left(R_s^a + \gamma \sum_{s'} p(s' \mid s, a) v_k(s')\right)$$

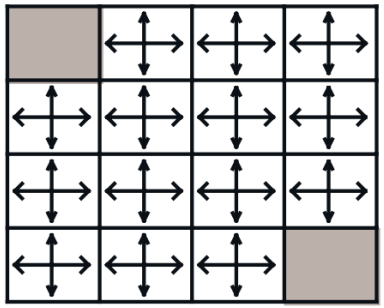
$$= \frac{1}{4}(-1+0) + \frac{1}{4}(-1+0) + \frac{1}{4}(-1+0) + \frac{1}{4}(-1+0) = -1$$

Iterative Policy Evaluation for the Small Gridworld

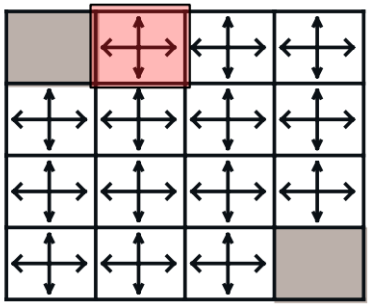
- $k = 1$:

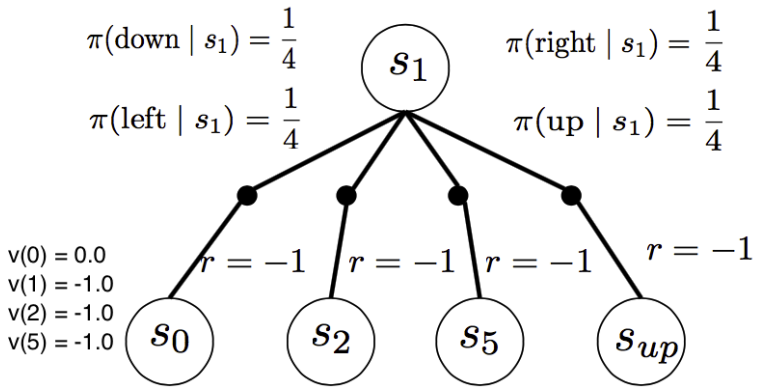
0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

Random policy:



- $k = 2$:





$$v_1(s_1) = \sum_a \pi(a \mid s)\left(R_s^a + \gamma \sum_{s'} p(s' \mid s, a) v_k(s')\right)$$

$$= \frac{1}{4}(-1+0) + \frac{1}{4}(-1-1) + \frac{1}{4}(-1-1) + \frac{1}{4}(-1-1) = -1.75$$

Iterative Policy Evaluation for the Small Gridworld

- For all iterations

$k = 0$

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

Q? When we reach $k = 10$, is it sensible to keep using a random policy? Can't we do better?

Policy Improvement

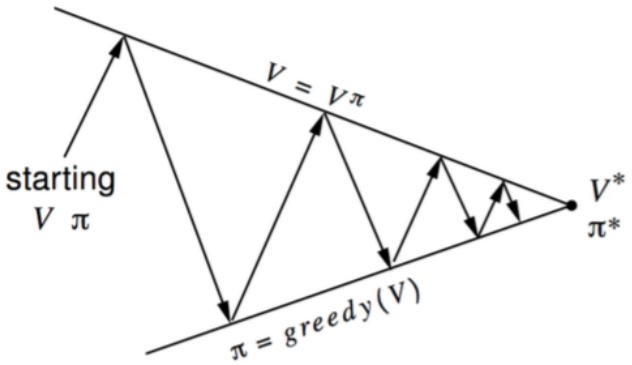
What if, instead of always using the random policy, after the first iteration we modify our policy using the values $v(s)$ we are calculating?

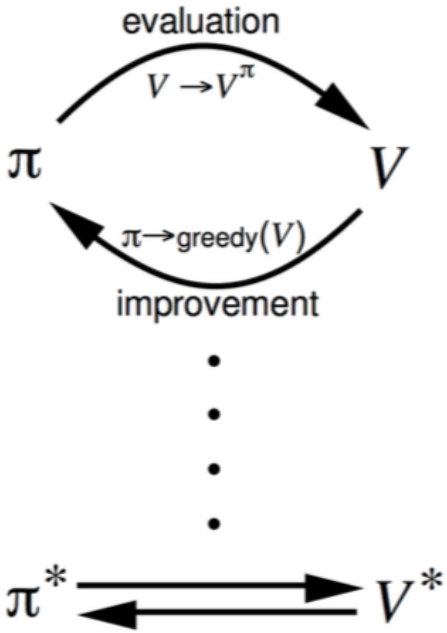
We can improve our policy by selecting the action that leads to the highest $v(s')$. This can be done with a **greedy** policy:

$$\pi'(s) = \arg\max_a q_\pi(s, a)$$

Policy improvement is the process of making a new policy that improves on an original policy, by acting greedily with respect to v_π.

Policy Iteration





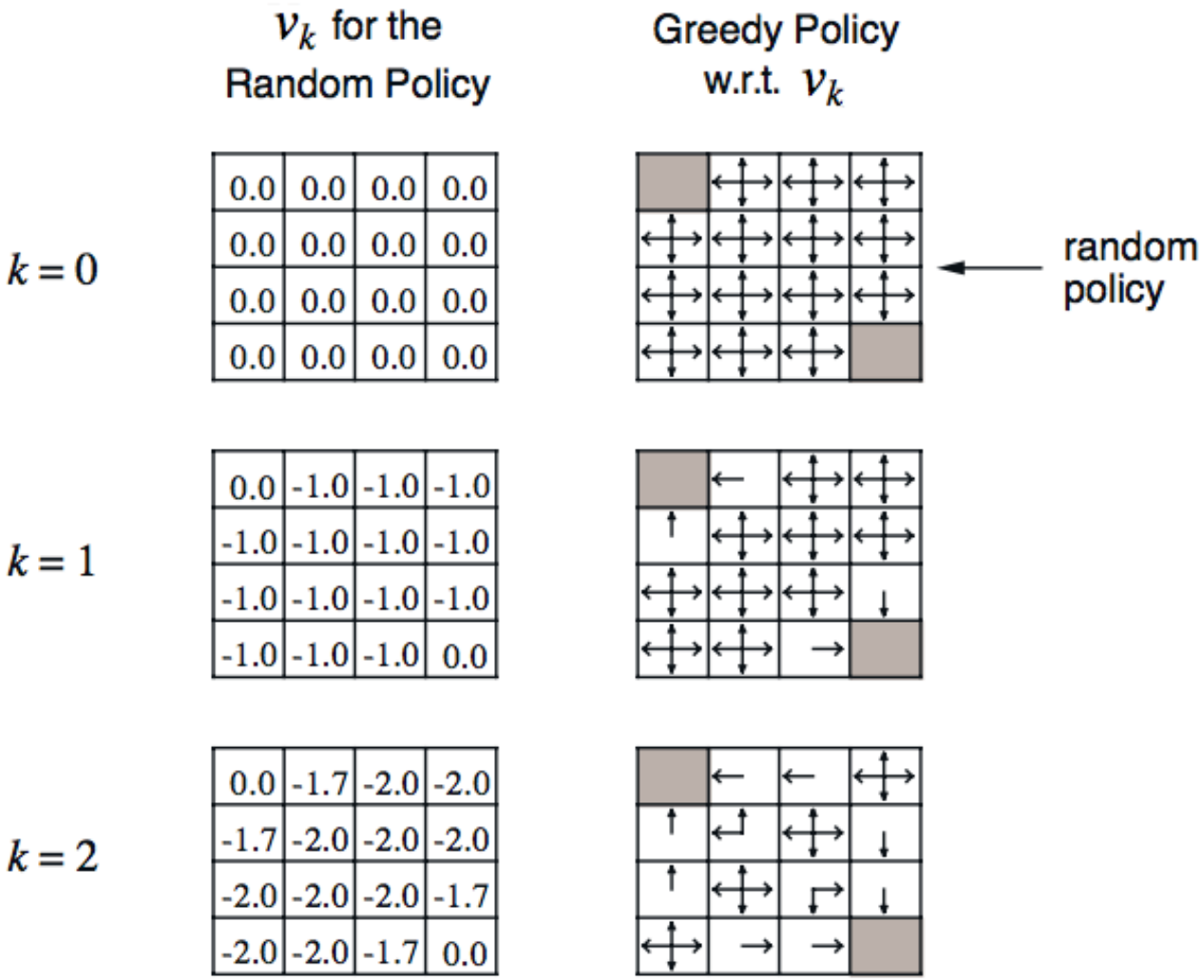
How to obtain π^*? We iterate through the following two steps:

1. **Policy evaluation:** Estimate v_π
   e.g. Iterative policy evaluation
2. **Policy improvement:** Generate $\pi' \geqslant \pi$
   e.g. Greedy policy improvement

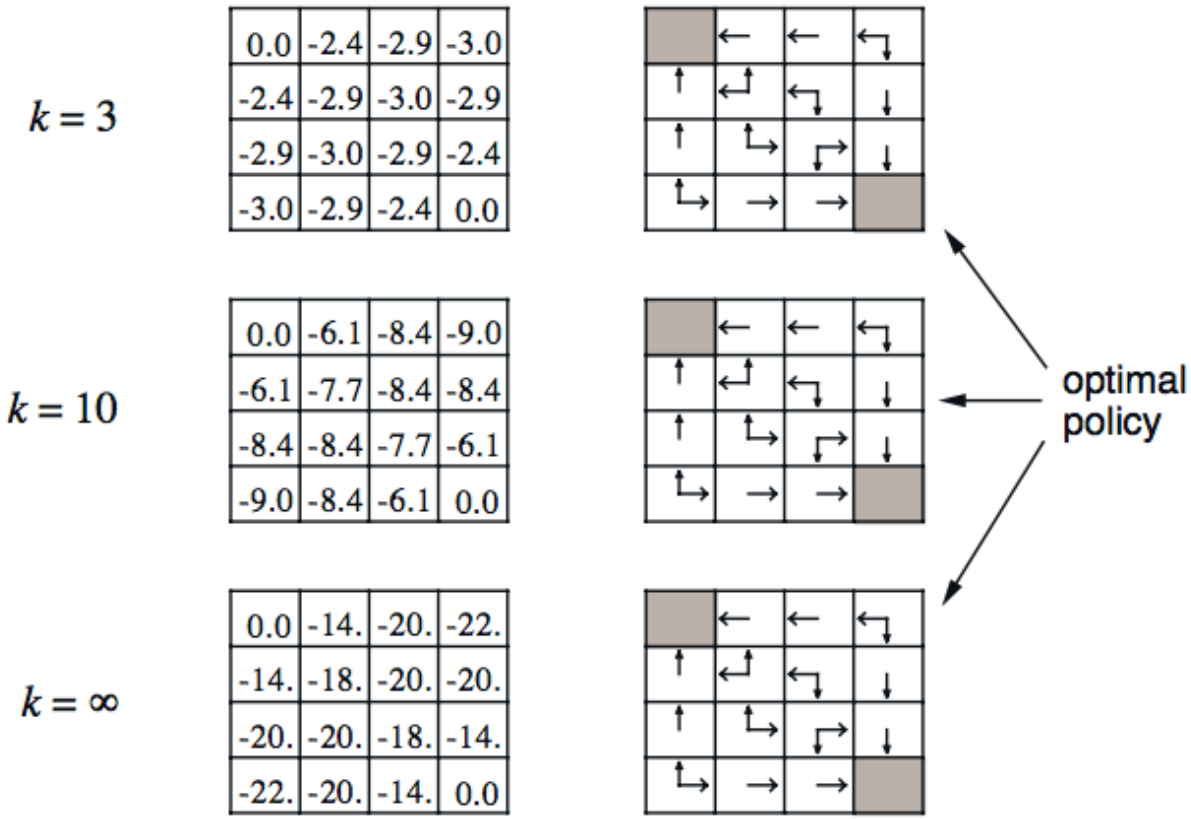
Process: we have a policy and evaluate how good is it (we complete Policy Evaluation). Then, change our policy to act better (greedily) according to v_π (Policy Improvement). We evaluate again how good this new policy is (another round of Policy Evaluation), to improve it again later (Policy Improvement again). Rinse and repeat.

Policy Improvement for the Small Gridworld

How does **Policy Improvement** work? Let's see how does the greedy policy **would look like** at each step of Policy Evaluation:



Policy Improvement for the Small Gridworld



In the Small Gridworld example, $\pi' = \pi^*$ in $k = 3$, but in general many more iterations on these two steps are needed. However, policy iteration **always** converges to π^*.

Q? When do we stop?

Policy Improvement

Q? When do we stop? When the improvements in the policy stop.

We are improving our current policy by acting greedily:

$$\pi'(s) = \arg\max_a q_\pi(s, a)$$

By definition, this implies that following our new policy π' from s is never worse than following our previous policy π from s:

$$q_\pi(s, \pi'(s)) \geqslant q_\pi(s, \pi(s))$$

Applying the definition of acting greedily:

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

and knowing that these two are equivalent:

$$q_\pi(s, \pi(s)) = v_\pi(s)$$

we can conclude that:

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

Policy Improvement

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

(this means we are improving - or at least not getting worse!)

Convergence: (when do we stop) if for all states $s \in S$, there is no improvement:

- $\pi'(s) = \pi(s)$, same actions are picked from every state before and after the iteration.
- $\rightarrow q_\pi(s, \pi'(s)) = q_\pi(s, \pi(s))$
- (see previous slide) $\rightarrow max_{a \in A} q_\pi(s, a) = v_\pi(s)$

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

- Bellman Optimality Equation is satisfied, so $v_\pi(s) = v_*(s)$
- $\rightarrow \pi$ is an optimal policy.

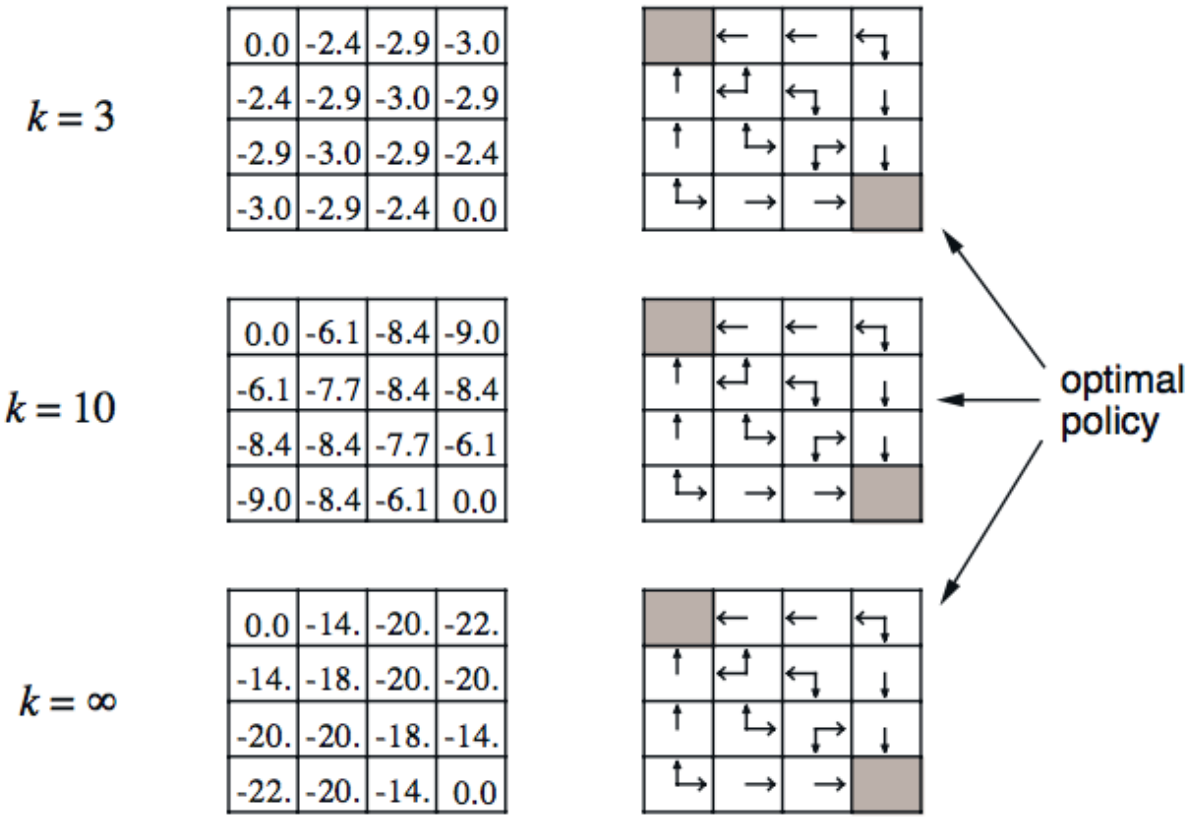
When improvement stops, we have reached the **optimal policy** $\pi*$.

Policy Iteration

```
1: procedure PolicyIteration
2:     v(s) ∈ ℝ and π(s, a) at random ∀s ∈ S                ▷ Initialize v(s), π(s, a)
3:     while not policyStable do
4:         PolicyEvaluation()
5:         policyStable ← PolicyImprovement()
6:     end while
7:     return V and π
8: end procedure
9:
10: procedure PolicyEvaluation
11:     repeat
12:         Δ ← 0                                           ▷ Largest change in v(s)
13:         for all s ∈ S do
14:             v_old ← v(s)
15:             v(s) ← Σ_a π(a | s)(R^a_s + γ Σ_s' p(s' | s, a)v_k(s'))
16:             Δ ← max(Δ, | v_old − v(s) |)                ▷ Keep the largest change in v(s)
17:         end for
18:     until Δ < Θ                                        ▷ Δ is still a small positive number
19: end procedure
20:
21: procedure PolicyImprovement
22:     for all s ∈ S do
23:         a ← π(s)
24:         π(s) ← arg max_a (R^a_s + γ Σ_s' p(s' | s, a)v_k(s'))
25:         if a ≠ π(s) then
26:             return False
27:         end if
28:     end for
29:     return True
30: end procedure
```

Modified Policy Iteration

Does policy evaluation need to converge to v_π?



Can't we stop after k iterations of iterative policy evaluation?

- $k = 3$ was okay for the small gridworld!

Value Iteration

Why not update policy **every** iteration ($k = 1$)? This is **Value Iteration**.

1: **procedure** ValueIteration
2: $v(s) \in \mathbb{R}$ at random $\forall s \in S$ ▷ Note: $\pi(s)$ is not random at start
3: **repeat**
4: $\Delta \leftarrow 0$ ▷ Largest change in $v(s)$
5: **for all** $s \in S$ **do**
6: $v_{old} \leftarrow v(s)$
7: $v(s) \leftarrow \max_{a \in A} R_s^a + \gamma \sum_{s'} p(s' \mid s, a) v_k(s')$
8: $\Delta \leftarrow max(\Delta, \mid v_{old} - v(s) \mid)$ ▷ Keep the largest change in $v(s)$
9: **end for**
10: **until** $\Delta < \Theta$ ▷ Δ is still a small positive number
11: **return** V and π
12: **end procedure**

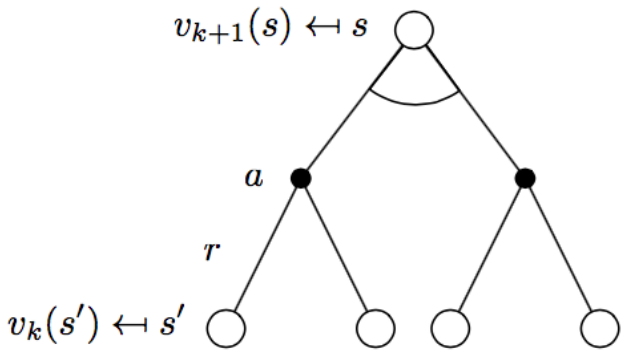
We **always** use: $\pi(s) = \arg\max_a q_\pi(s, a)$

Reinforcement Learning: Planning

Monte Carlo Methods

Using Samples

Dynamic Programming uses full backups. This is, for each state, all successor states and actions are considered, assuming full knowledge of the MDP (this is, assuming we know $P^a_{ss'}$):



This is not effective for large sized problems, as it suffers Bellman's *curse of dimensionality*: the number of states $s \in S$ grows exponentially with the number of state parameters.

Instead, **Monte Carlo** (MC) methods use sampling: sample sequences of states, actions and rewards from actual or simulated interaction with an environment. MC methods require only experience, they **do not** assume complete knowledge of the MDP (we **don't** necessarily know $P^a_{ss'}$). This is **model-free prediction**.

Monte Carlo Policy Evaluation

In MC, we learn the value of v_π from episodes of experience (interaction of the agent with the environment) using policy π. The return of a single sample is, as seen before:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

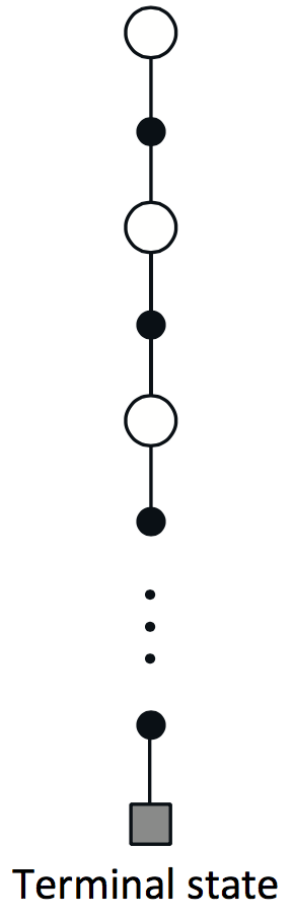
And the value function still is:

$$v_\pi(s) = \mathbb{E}_\pi(G_t \mid S_t = s)$$

But note:

- In DP, we **compute** the value function as the expected return, with full knowledge of the MDP.
- In MC, we **learn** the value function, as empirical mean, sampling from the MDP. MC uses the *average of returns*, which is the most obvious/simplest way. As more experience is observed (more returns are calculated), the average converges to the expected value.

MC only works for episodic (always terminate) tasks.



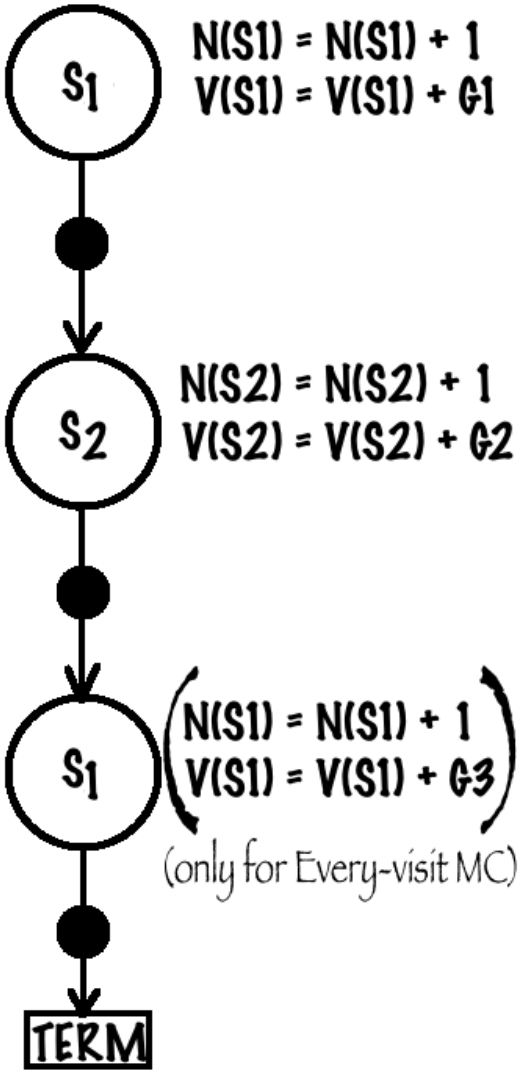
Monte Carlo Policy Evaluation

procedure FIRSTVISITMCPOLICYEVALUATION(π)
  repeat
    Generate an episode using π (EP_π)
    for all $s \in EP_\pi$ **do**
      $N(s) \leftarrow N(s) + 1$ ▷ Increment visit counter
      $S(s) \leftarrow S(s) + G_t$ ▷ Increment accumulated return
      $v_{old} \leftarrow v(s)$
      $v(s) = S(s)/N(s)$ ▷ Update $v(s)$ by mean return
      $\Delta \leftarrow max(\Delta, \mid v_{old} - v(s) \mid)$ ▷ Keep the largest change in $v(s)$
    end for
  until $\Delta < \Theta$ ▷ Δ is still a small positive number
  return V_π ▷ $v(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$
end procedure

Note that we need to wait until the end of the episode to be able to update $N(s), v(s)$. There are two variants:

- **First-Visit** MC Policy Evaluation: Update $N(s)$ and $v(s)$ only from the **first** time s was found in an episode.
- **Every-Visit** MC Policy Evaluation: Update $N(s)$ and $v(s)$ **every** time s is found in the episode.

First versus Every-Visit MC



The value of the Return G_t is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

For this example:

$$G_1 = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}$$

$$G_2 = R_{t+2} + \gamma R_{t+3}$$

$$G_3 = R_{t+3}$$

Incremental Monte Carlo Updates

An alternative way of computing a mean, incrementally (μ_{k-1} : previous mean):

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) = \frac{1}{k}(x_k + (k-1)\mu_{k-1})$$

$$= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

Therefore, we can update $v(s)$ incrementally after episode. For each state s_t with return G_t:

$$N(s_t) \leftarrow N(s_t) + 1$$

$$v(s_t) \leftarrow v(s_t) + \frac{1}{N(s_t)}(G_t - v(s_t))$$

$(G_t - v(s_t))$ is a type of *error* (what happened, minus what I expect to happen!).

Incremental Monte Carlo Updates

$$N(s_t) \leftarrow N(s_t) + 1$$

$$v(s_t) \leftarrow v(s_t) + \frac{1}{N(s_t)}(G_t - v(s_t))$$

Also (for non-stationary problems), we may not want to remember all episodes (this happens when factoring by $\frac{1}{N(s_t)}$).

Instead, we can forget old episodes by factoring by a constant α (non-stationary environments).

Definition (Incremental MC: Constant-α MC method)

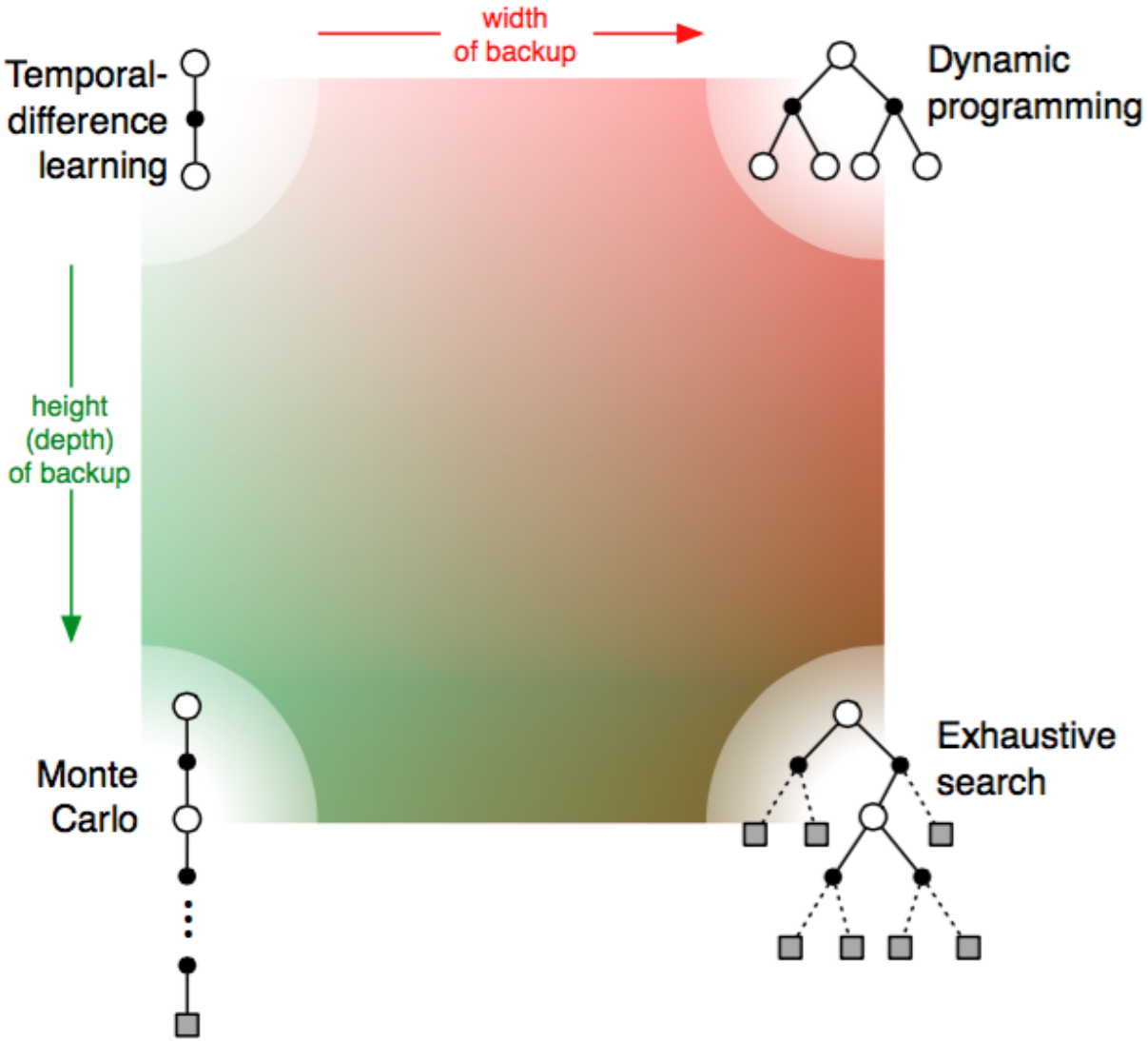
$$v(s_t) \leftarrow v(s_t) + \alpha(G_t - v(s_t))$$

We are correcting our estimate of $v(s_t)$ by moving the value a *little bit* (α) in the direction of the error.

Reinforcement Learning: Planning

Temporal Difference Learning

Unified Model of RL methods



Temporal Difference Learning (TDL)

Temporal Difference (TD) methods:

- Learn from episodes of experience.
- Model free: they **don't** need full knowledge of the MDP.
- Learns from incomplete episodes (*bootstrapping*), without waiting for a final outcome.

The simplest TDL algorithm is **TD(0)**:

The main idea is to substitute the target (G_t) used in incremental MC updates:

$$v(s_t) \leftarrow v(s_t) + \alpha(G_t - v(s_t))$$

for the target: $R_{t+1} + \gamma v(S_{t+1})$:

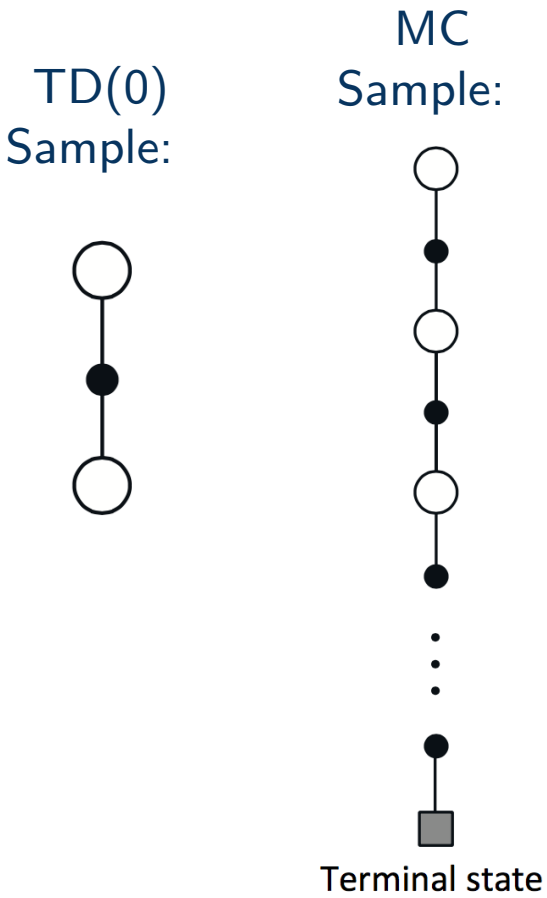
$$v(s_t) \leftarrow v(s_t) + \alpha(R_{t+1} + \gamma v(S_{t+1}) - v(s_t))$$

Temporal Difference Learning

Algorithm	Waits until	Updates towards (target)
MC	End of the episode	The actual return (G_t)
TD(0)	Next time step	The *estimated* return ($R_{t+1} + \gamma v(S_{t+1})$)

$$v(s_t) \leftarrow v(s_t) + \alpha(R_{t+1} + \gamma v(S_{t+1}) - v(s_t))$$

- $R_{t+1} + \gamma v(S_{t+1})$ is called *TD Target*.
- $\delta_t = R_{t+1} + \gamma v(S_{t+1}) - v(s_t)$ is called *TD Error*.



TD(0)

1: **procedure** TD_0(π)
2: Initialize v(s) arbitrarily. ▷ i.e. $v(s) = 0 \; \forall s \in S$
3: **repeat** ▷ for each episode
4: **for all** $s \in S$ **do** ▷ For all states
5: $A \leftarrow \pi(s)$ ▷ Take action following policy π
6: $R, s' \leftarrow A$ ▷ Determine immediate reward R and next state s'
7: $v(s) \leftarrow v(s) + \alpha(R + \gamma v(s') - v(s))$
8: $s \leftarrow s'$ ▷ Update variable s for next iteration
9: **end for**
10: **until** last episode ▷ All episodes
11: **end procedure**

Note: Where is the TD target $R_{t+1} + \gamma v(S_{t+1})$ coming from?

The Bellman Expectation Equation:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$$

TD(0) vs. MC

	TD(0)	MC
Learning	Can learn online after every step.	Waits until the end of the episode, when return is known.
Environment	Terminating and non-terminating environments.	Only works in terminating environments.
Bias	Biased: based on estimates and initialization of $v(s)$	Unbiased: works with the true value $G_t(s)$ (no dependence on $v(s)$).
Variance	Lower: less noise, depends on one step	Higher: depends on many decisions of the environment until the end of the episode.
Markov	Exploits Markov property: more efficient in Markov environments	Does not exploit Markov property: more effective in non-Markov environments.

Both MC and TD(0) converge: $V(s) \to v_\pi(s)$ as experience $\to \infty$.

TD(0) vs. MC - The Markov Property

Given the following 8 episodes (Markov States, $\gamma = 1$):

$$\begin{aligned}
&\mathrm{A}, 0, \mathrm{B}, 0 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 0
\end{aligned}$$

Q? Imagine **you** are the predictor. What's the value of $V(B)$?

TD(0) vs. MC - The Markov Property

Given the following 8 episodes (Markov States, $\gamma = 1$):

$$\begin{aligned}
&\mathrm{A}, 0, \mathrm{B}, 0 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 0
\end{aligned}$$

Q? Imagine **you** are the predictor. What's the value of $V(B)$?

6 of 8 (3/4, 75%) are 1. V(B) = 0.75.

TD(0) vs. MC - The Markov Property

Given the following 8 episodes (Markov States, $\gamma = 1$):

$$\begin{aligned}
&\mathrm{A}, 0, \mathrm{B}, 0 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 0
\end{aligned}$$

Q? And $V(A)$?

TD(0) vs. MC - The Markov Property

Given the following 8 episodes (Markov States, $\gamma = 1$):

$$\begin{aligned}
&\mathrm{A}, 0, \mathrm{B}, 0 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 0
\end{aligned}$$

Q? And $V(A)$?

- MC answer (Markov ignored, just experience). **V(A) = 0**

TD(0) vs. MC - The Markov Property

Given the following 8 episodes (Markov States, $\gamma = 1$):

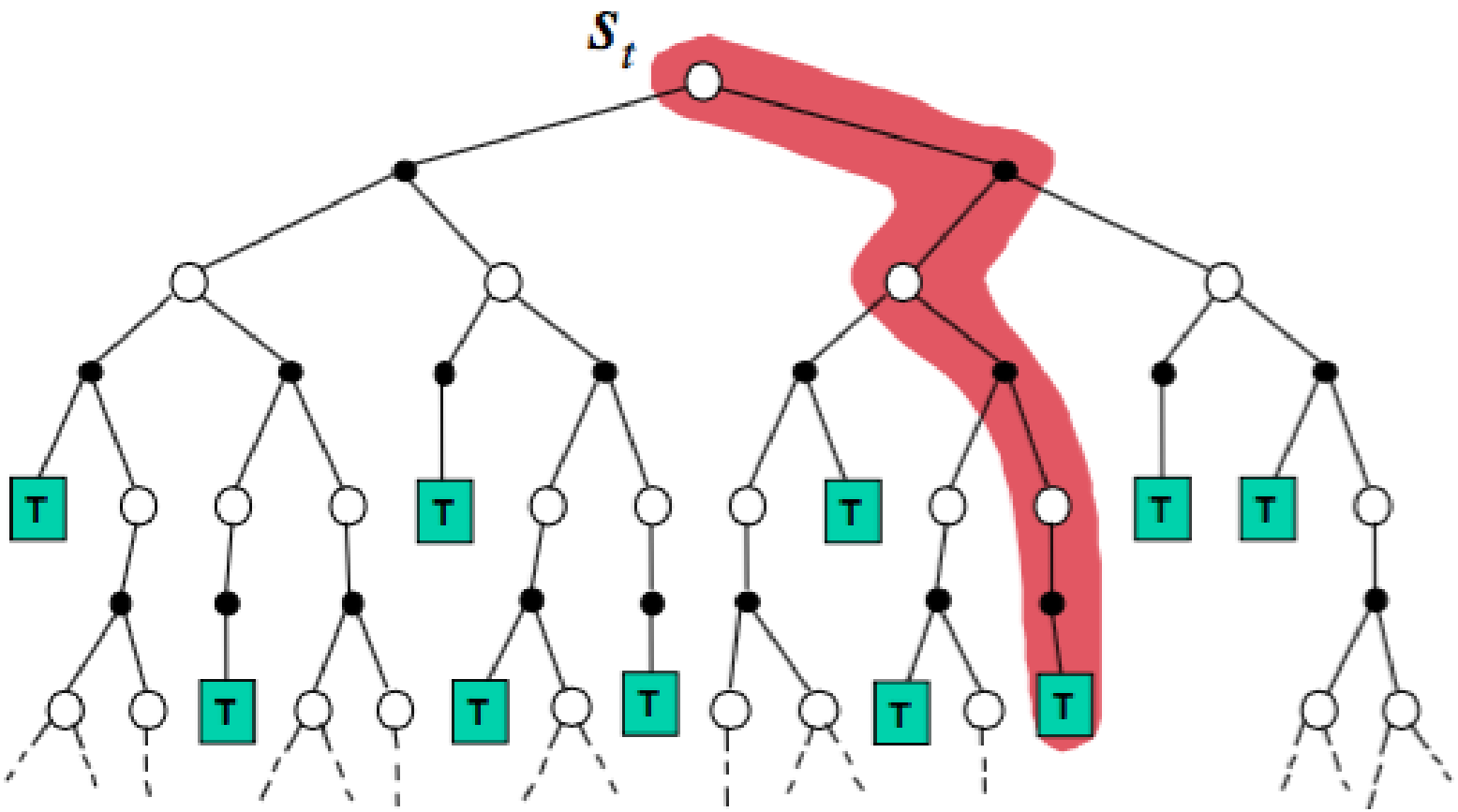
$$\begin{aligned}
&\mathrm{A}, 0, \mathrm{B}, 0 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 1 \\
&\mathrm{B}, 0
\end{aligned}$$

Q? And $V(A)$?

- MC answer (Markov ignored, just experience). **V(A) = 0**
- TD answer: relies in the fact that V(B) = 0.75 and we've only seen A $\rightarrow$ B once. So fully relies on V(B) independently on where it's coming from. **V(A) = 0.75**

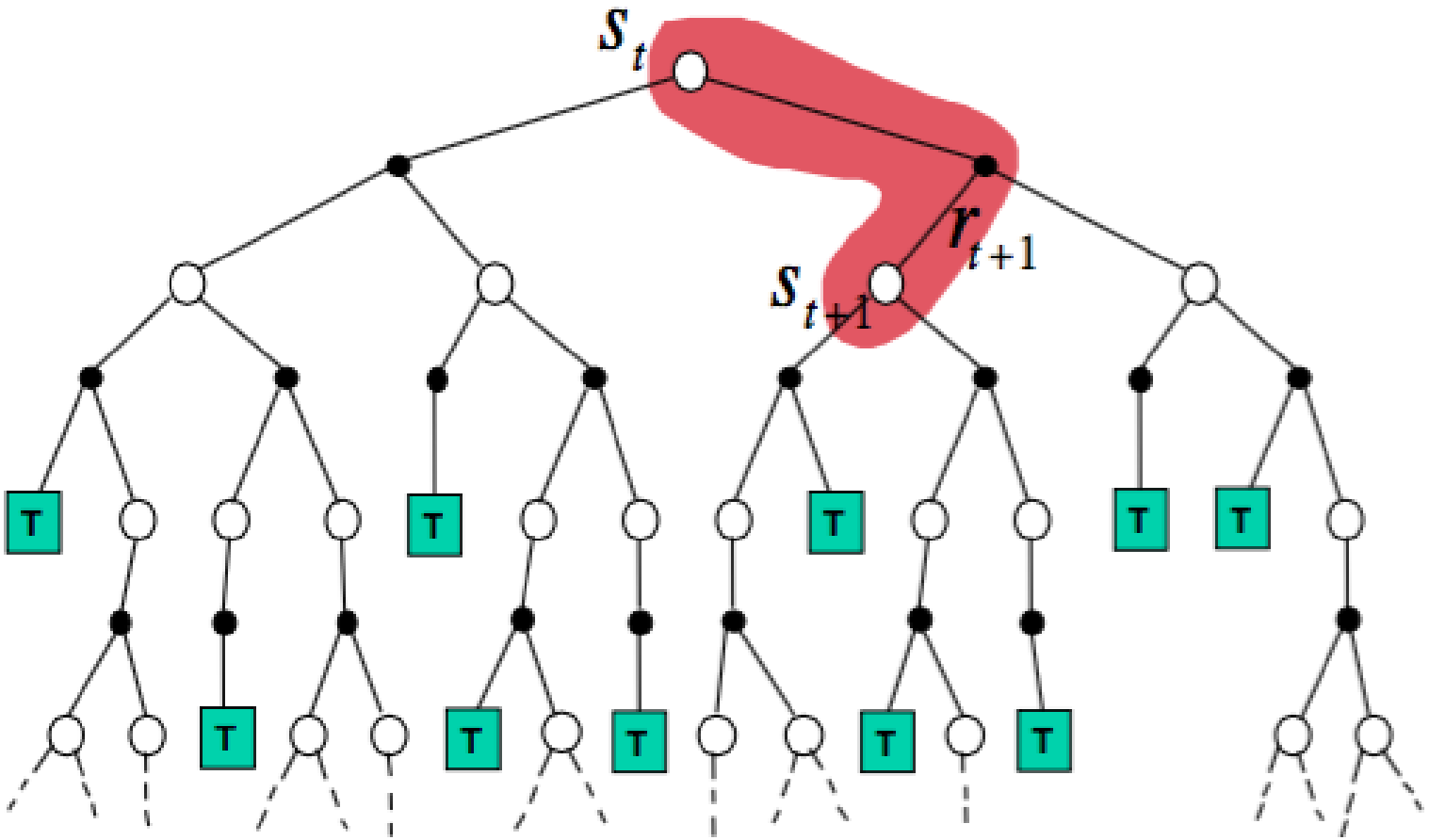
Monte Carlo: Back Up

$$V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$$



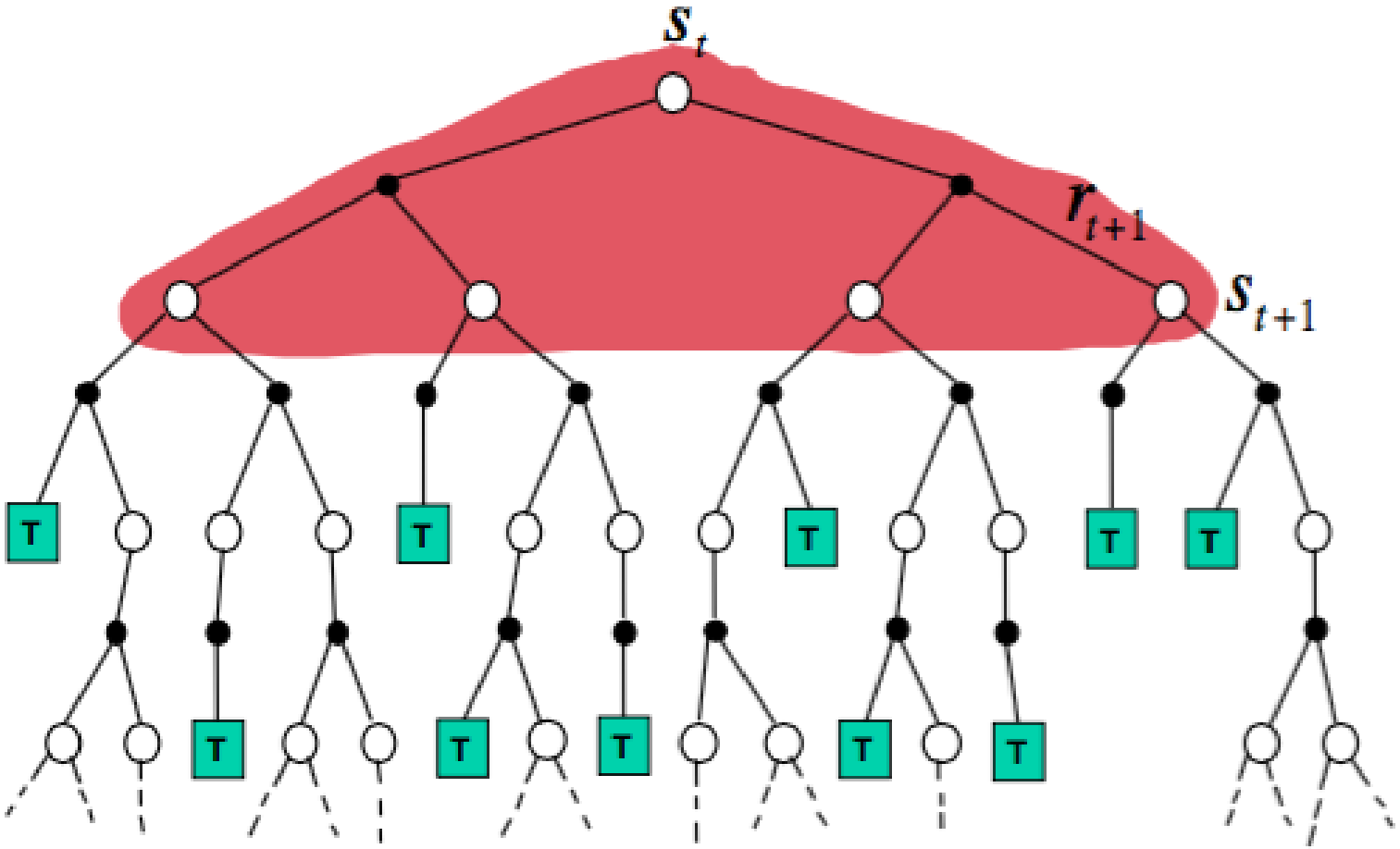
TD(0): Back Up

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$



Dynamic Programming: Back Up

$$V(S_t) \leftarrow \mathbb{E}_\pi \left[R_{t+1} + \gamma V(S_{t+1}) \right]$$



Acknowledgements

Additional Materials:

- *Reinforcement Learning: An Introduction*, by Andrew Barto and Richard S. Sutton (2017 Edition):

  `http://incompleteideas.net/book/bookdraft2017nov5.pdf`

- Prof. David Silver's course on Reinforcement Learning:

  `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

- Prof. Richard Sutton's lecture on Temporal Difference Learning (Montreal 2017)

  `http://videolectures.net/deeplearning2017_sutton_td_learning/`