Introduction to Reinforcement Learning

Lecture 1: Fundamentals of RL

Joshua B. Evans

jbe25@bath.ac.uk

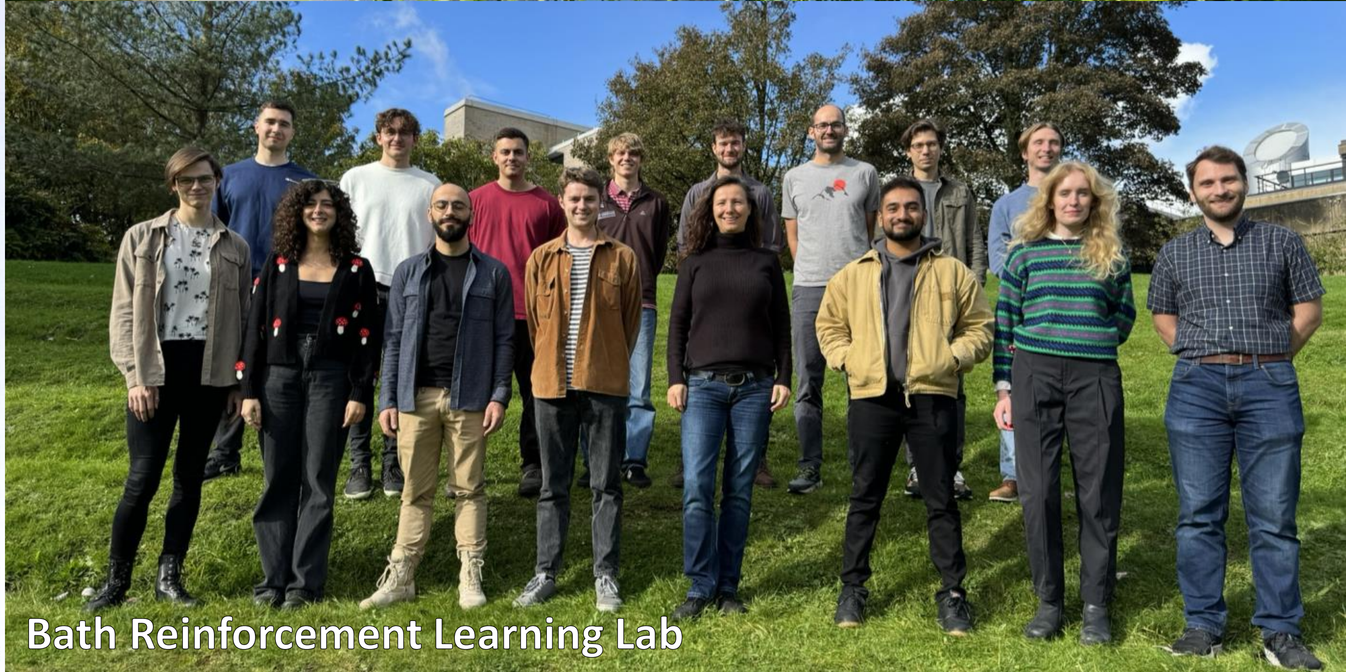
Bath Reinforcement Learning Laboratory

Department of Computer Science

UNIVERSITY OF BATH


Europe
Scale 1:4,900,000
Bath, UK
Greenland (DENMARK)
ICELAND
Reykjavík
North Atlantic Ocean
Norwegian Sea
North Sea
Barents Sea
NORWAY
SWEDEN
FINLAND
RUSSIA
ESTONIA
LATVIA
LITHUANIA
BELARUS
UKRAINE
POLAND
GERMANY
DENMARK
IRELAND
UNITED KINGDOM
NETHERLANDS
BELGIUM
FRANCE
SWITZERLAND
AUSTRIA
CZECH REPUBLIC
SLOVAKIA
HUNGARY
ROMANIA
MOLDOVA
BULGARIA
SERBIA AND MONTENEGRO
CROATIA
BOSNIA AND HERZEGOVINA
ALBANIA
MACEDONIA
GREECE
TURKEY
ITALY
SPAIN
PORTUGAL
MOROCCO
ALGERIA
TUNISIA
LIBYA
EGYPT
CYPRUS
MALTA
Black Sea
Mediterranean Sea
Bay of Biscay
Celtic Sea
English Channel
Baltic Sea
Adriatic Sea
Tyrrhenian Sea
Ionian Sea
Aegean Sea
London
Paris
Berlin
Madrid
Lisbon
Rome
Vienna
Prague
Warsaw
Budapest
Bucharest
Sofia
Athens
Moscow
Kiev
Minsk
Oslo
Stockholm
Helsinki
Copenhagen
Amsterdam
Dublin
Istanbul
Ankara


University of Bath


Bath Reinforcement Learning Lab

Today's Lecture

"Fundamentals of RL"

- What is RL?
- Key RL Theory
  - MDPs
  - Policies
  - Value Functions
- Key Solution Methods
  - Dynamic Programming
  - Monte Carlo
  - Temporal-Difference

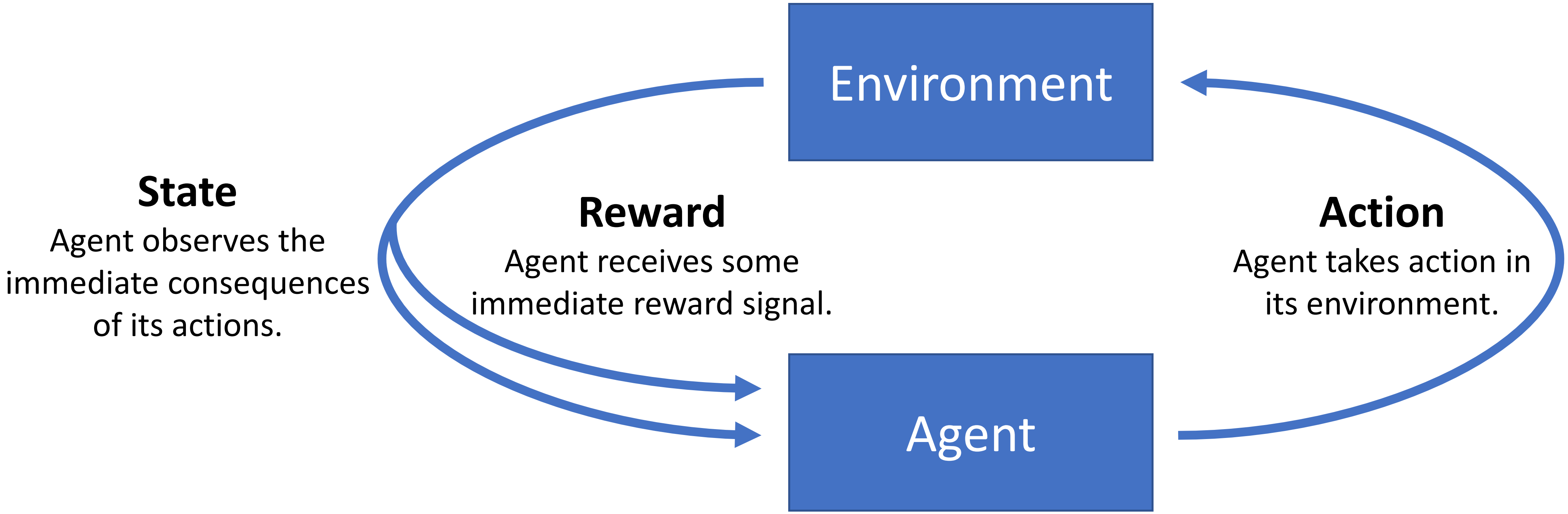
Tomorrow's Lecture

"Frontiers of RL"

- Generalisation & Scaling Up
- Deep RL
  - Value-Based: DQN
  - Policy-Based: REINFORCE
  - Actor-Critic: DDPG
- Research Topics
  - Offline RL
  - Inverse RL
  - Intrinsically-Motivated RL
  - Hierarchical RL

What is Reinforcement Learning?

Reinforcement Learning (RL) is a computational approach to **goal-directed learning from interaction**.

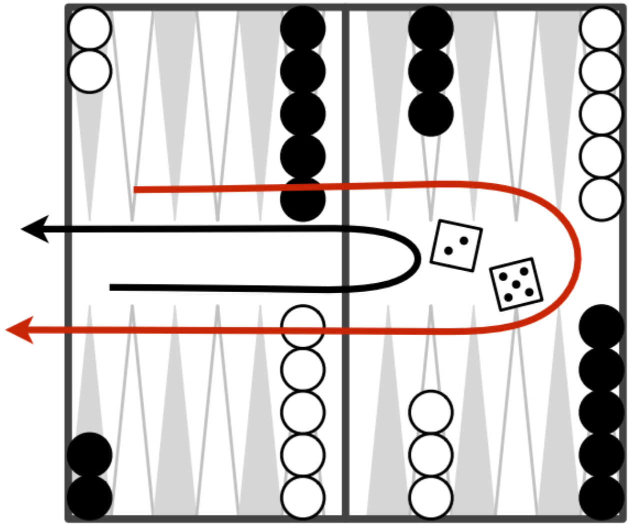


RL is **learning how to act:** how to **map states to actions** in order **to maximise long-term reward.**

What Can Reinforcement Learning Do?

- In RL, we aim to solve **sequential decision problems**.
  - Our agent must take a **sequence of actions** in order **to reach its goal**.
  - The overall **reward** it earns **depends on the whole sequence** of actions.

AlphaGo Zero (Silver et al., 2017)



TD-Gammon (Tesauro, 1992-1995)

DQN (Mnih et al., 2013-2015)

Image Credit: [CNET.com](), Özgür Şimşek, [DeepMind]()

What Can Reinforcement Learning Do?

- In RL, we aim to solve **sequential decision problems**.
  - Our agent must take a **sequence of actions** in order **to reach its goal**.
  - The overall **reward** it earns **depends on the whole sequence of actions**.

- The RL framework is very flexible, and can be applied to many different problems in many different ways.

- If a given problem requires our agent to make a sequence of decisions in order to reach some goal, we can probably make use of RL.

Many, many application areas!

Key Features of Reinforcement Learning

- Rewards can be **delayed**.
- **Short-term sacrifices** may lead to long-term gains.
- Trade-off between **exploration** and **exploitation**.
- It's **not supervised learning**.
  - We don't tell our agent which actions to choose.
  - Our agent learns through trial-and-error.
- It's **not unsupervised learning**.
  - Our agent isn't trying to find hidden structure in unlabelled data.
- RL is a separate branch of machine learning.

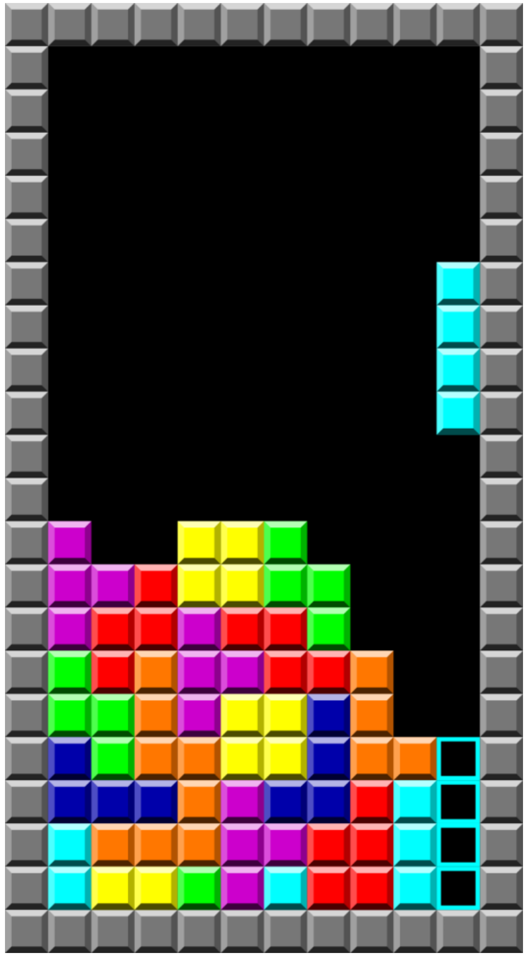
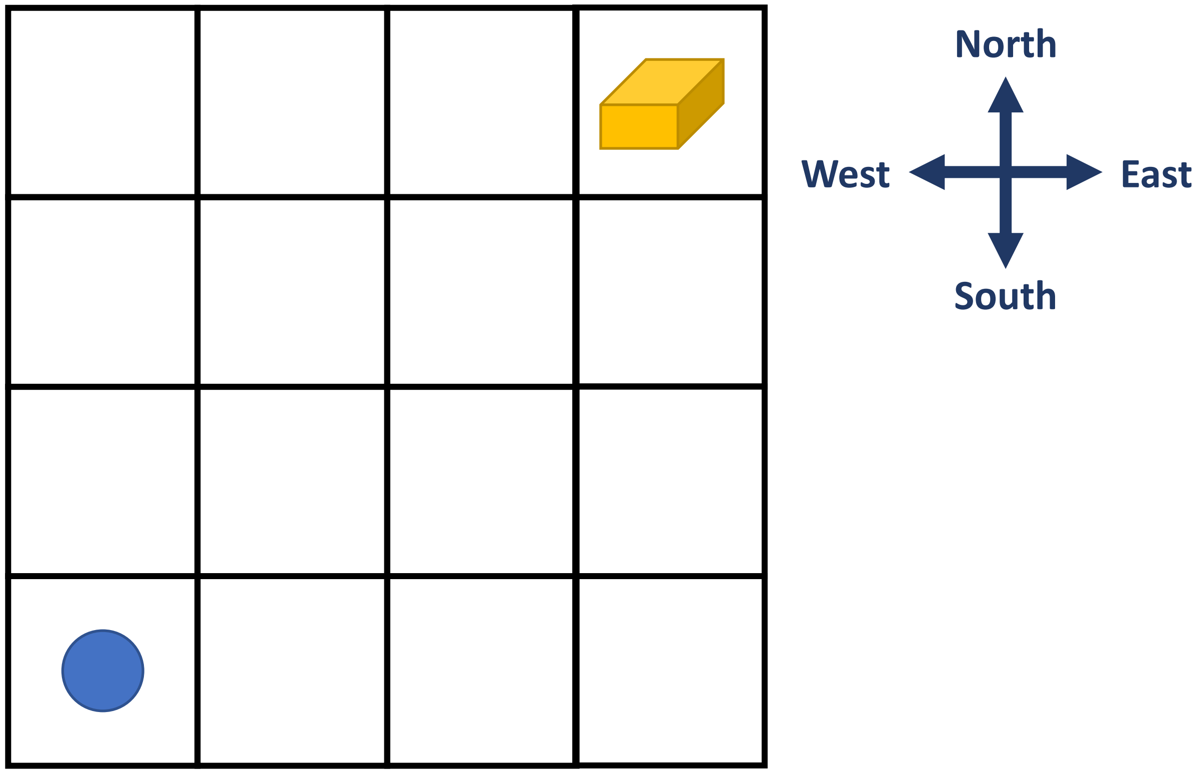


Image Credit: Wikimedia Commons

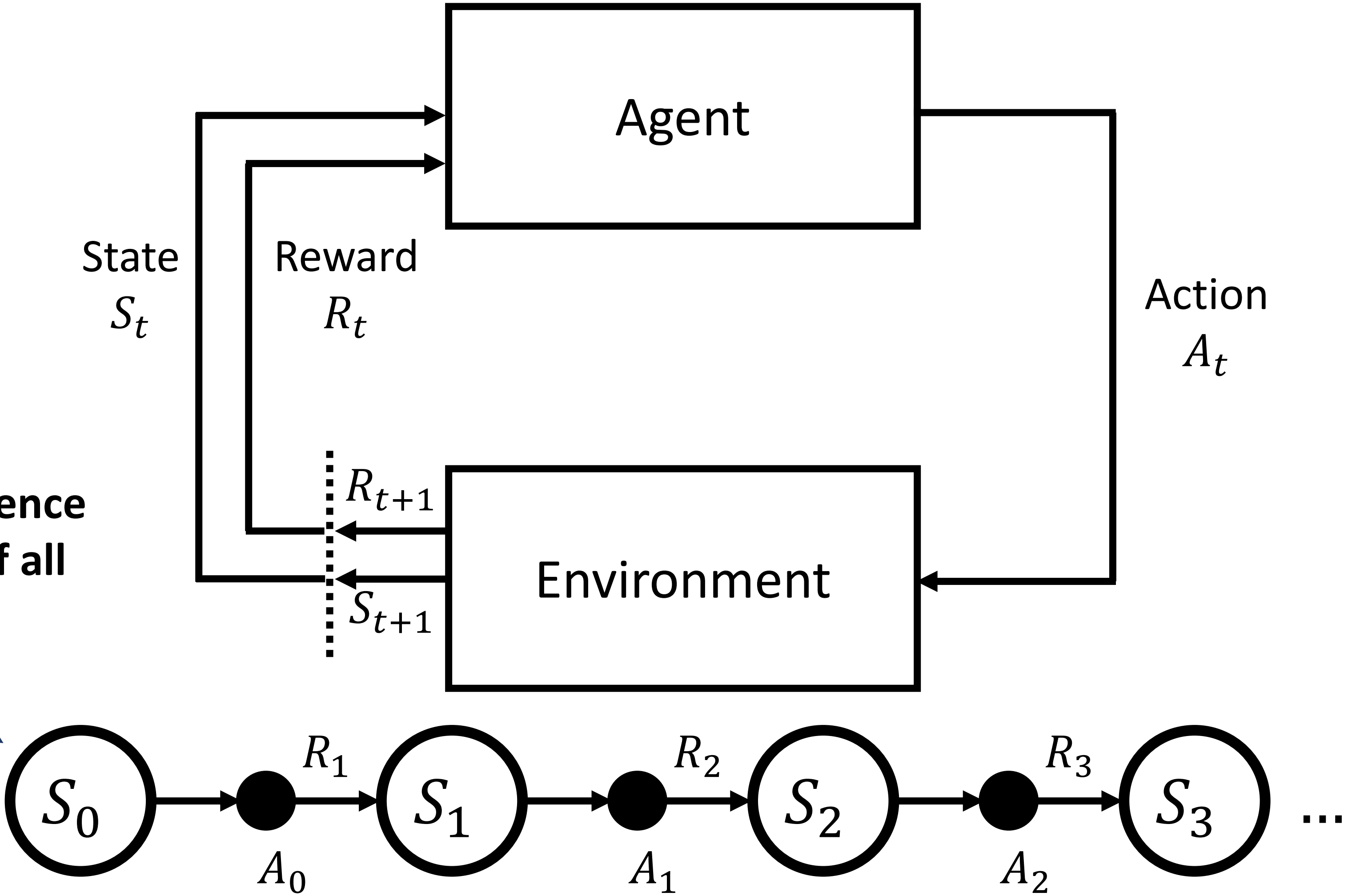
The Gold Gridworld



How can we formally represent the interaction between an agent and this environment?

Using this representation, how can we train an agent to maximise long-term reward?

The Agent-Environment Interaction



Our agent's **experience forms the basis of all learning** in RL.

$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3 \ldots$$

Figure adapted Sutton & Barto (2018)

Episodic & Continuing Tasks

Episodic Tasks

- Interaction naturally splits into **episodes.**
  - Example: games of chess.
- The environment resets when our agent reaches a **terminal state** at time-step T.

Continuing Tasks

- Interaction **continues forever** with no clear breaks.
  - Example: a mars rover exploring its environment.
- There are no terminal states or final time-steps.

Learning a Policy

- Our agent should learn a **policy**, a function that determines which action it should take in each state.
  - A **policy** $\pi_t(s, a)$ returns the **probability** of selecting action a in state s at time-step t.
- The agent should learn a policy that maximises the **total discounted return** – the discounted sum of all future rewards.

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

For **episodic** tasks, we can use $\gamma = 1.0$.

For **continuing** tasks, we must use $\gamma < 1.0$.

What is a State?

- The **state** at time-step t should contain whatever relevant information is available to our agent about its environment at time-step t.
  - It could be very simple (e.g., a pair of coordinates on a grid).
  - It could be more complex (e.g., pixel-inputs from a camera).

- Importantly, it should **summarise all past information** relevant to our agent's decision-making process.
  - Specifically, it should possess the **Markov property**.

$$P(S_{t+1}, R_{t+1} | \boxed{S_0, A_0, R_1, \ldots, R_t,} \boxed{S_t, A_t}) = P(S_{t+1} R_{t+1} | S_t, A_t)$$

These should not give us additional information.

Given these…

Markov Decision Processes (MDPs)

- If a sequential decision problem has the Markov property, then it is a **Markov Decision Process** (**MDP**).
- To define an MDP, we need:
  - A **set of states**: S
  - A **set of actions** available in each state: $A(s), \quad s \in S$
  - A **transition function**: $p(s'|s, a), \quad s \in S, \ s' \in S, \ a \in A(s)$
  - A **reward function**: $r(s, a, s'), \quad s \in S, \ s' \in S, \ a \in A(s)$
  - An **initial state distribution**: $h(s), \quad s \in S$
  - A **discount factor**: $0 \leq \gamma \leq 1$
- We will often combine p and r into $p(s', r|s, a)$.
- If S and $A(s)$ are finite, then it is a **finite MDP**.

Markov Decision Processes (MDPs)

$S = \{1,2,3, \ldots, 15,16\}$

$A(s) = \{N, S, E, W\} \quad \forall s \in S$

$+10$ reward for transitioning to state 16,
-1 reward otherwise.

$$h(s) = \begin{cases} 1.0 & \text{if } s = 1 \\ 0.0 & \text{otherwise} \end{cases}$$

$\gamma = 1.0$

- To define an MDP, we need:
  - A **set of states**: S
  - A **set of actions** available in each state: $A(s), \quad s \in S$
  - A **transition function**: $p(s'|s, a), \quad s \in S, \; s' \in S, \; a \in A(s)$
  - A **reward function**: $r(s, a, s'), \quad s \in S, \; s' \in S, \; a \in A(s)$
  - An **initial state distribution**: $h(s), \quad s \in S$
  - A **discount factor**: $0 \leq \gamma \leq 1$

What Does an RL Algorithm Do?

It should tell us how to use **experience** generated by an agent to **modify its policy** in order to **maximise the discounted return**.

Dynamic Programming

Monte Carlo Methods

Temporal Difference Methods

Value Functions

- The **value of a state** is the return that our agent can expect to earn if it starts in a given state and then follows its policy thereafter.

- The **value of taking an action in a state** is the return that our agent can expect to earn if it starts in a given state, takes a given action, and then follows its policy thereafter.

- Note that values are defined with respect to a specific **policy.**
  - The value of being in a given state or taking a given action might be very different depending on what policy our agent is using!

Value Functions

- **State-Value Function**

$$v_\pi(s) \doteq E_\pi\{G_t|S_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right\}$$

- **Action-Value Function**

$$q_\pi(s, a) \doteq E_\pi\{G_t|S_t = s, A_t = a\}$$
$$= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right\}$$

$E_\pi\{\cdot\}$ denotes the expected value under a given policy π.

Comparing Policies

- We can use value functions to compare policies.

- Policy π is as good as or better than policy π' if π has at least as high a state-value as π' in every state.

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S$$

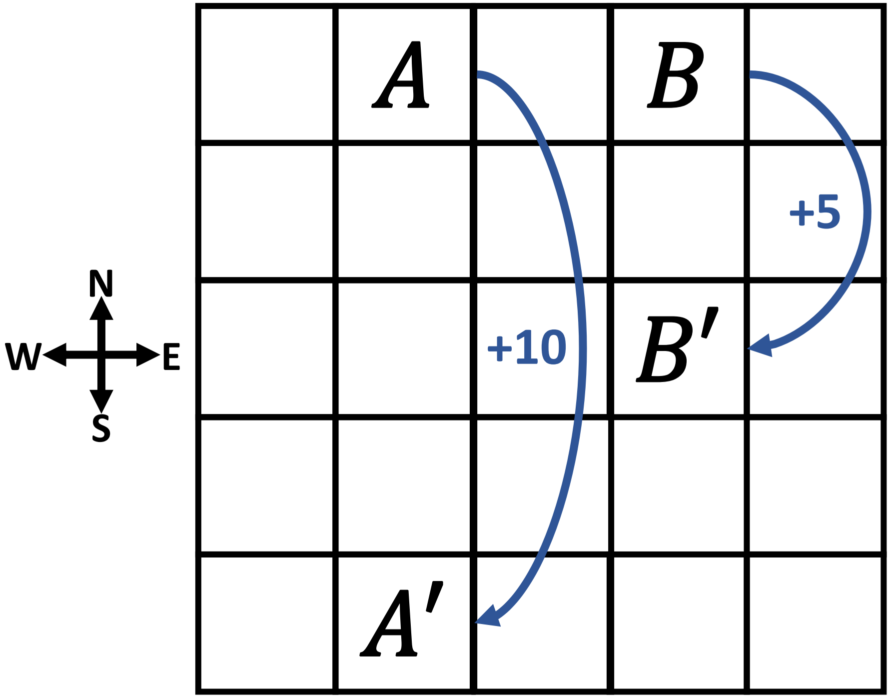
Optimal Policies

- There is always at least one policy that is better than or equal to all other policies. This is the **optimal policy**, denoted π_*.

- Optimal policies share the same optimal state-value function:
$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in S$$

- Optimal policies also share the same optimal action-value function:
$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in S, \forall a \in A(s)$$

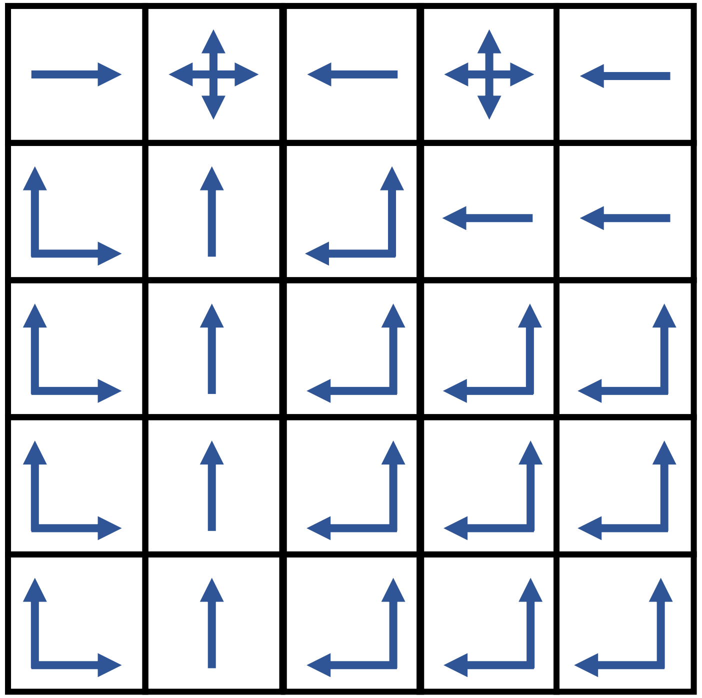
From Value Functions to Policies



Gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

v_*



π_*

The optimal policy π_* chooses actions that maximise $r + v_*(s')$.

Values computed using $\gamma = 0.9$.

Figure adapted from Sutton & Barto (2018)

Policy Evaluation & Improvement

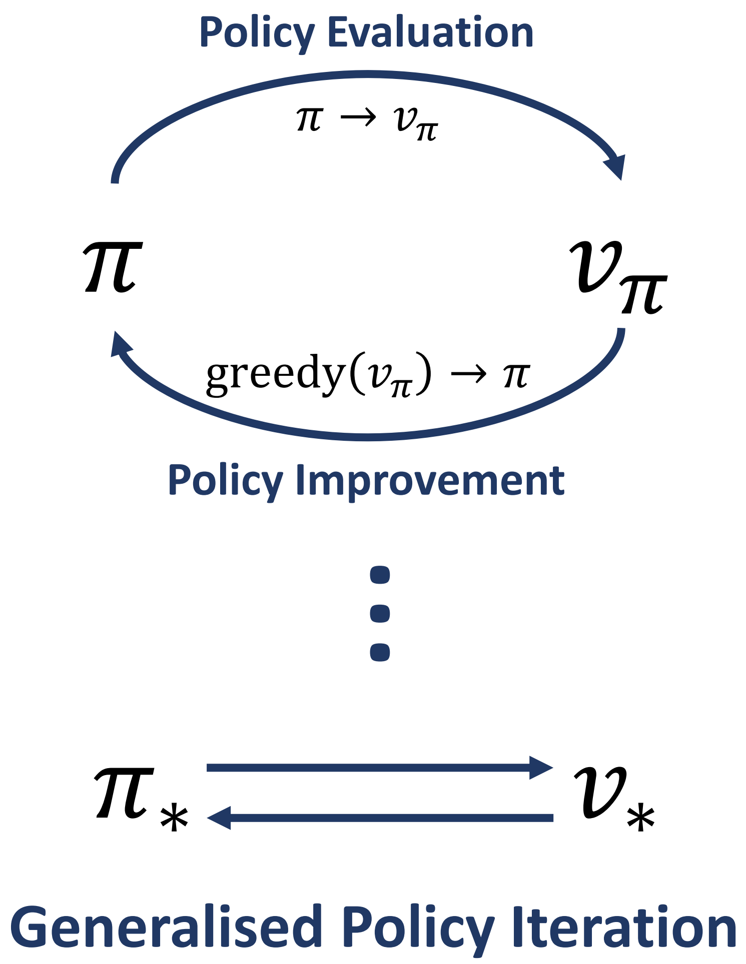
- **Policy Evaluation**: Finding the value function v_π for a given policy π.

$$\pi \rightarrow v_\pi$$

- **Policy Improvement**: Acting greedily with respect to a value function v_π to yield a new policy, π'.

$$\text{greedy}(v_\pi) \rightarrow \pi'$$

- The **policy improvement theorem** guarantees that $\pi' \geq \pi$.

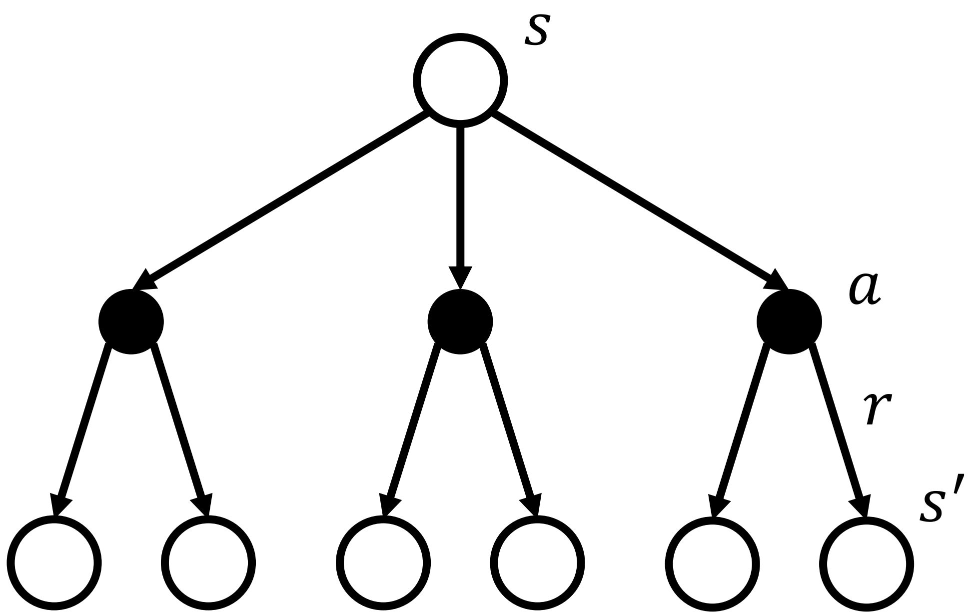


Everything here applies to action-value functions q_π too!

Dynamic Programming Methods

Bootstrapping
Basing one estimate on another.

The estimate of $v_\pi(s)$ is based on an estimate of $v_\pi(s')$.



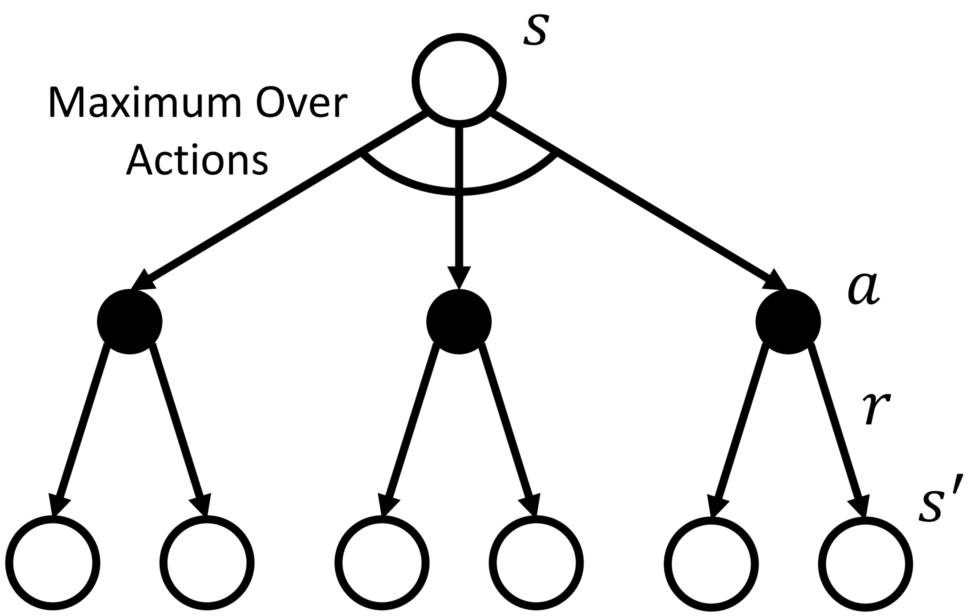
$$v_\pi(s) = \sum_a \pi(s,a) \sum_{s',r} p(s',r|s,a)\,[r + \gamma v_\pi(s')]$$

Value of a State

Value of its Successors

Bellman Equation for v_π

Dynamic Programming Methods



Problem: to solve this directly, we need full knowledge of $p(s', r|s, a)$.

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

Value of a State

Value of its Successors

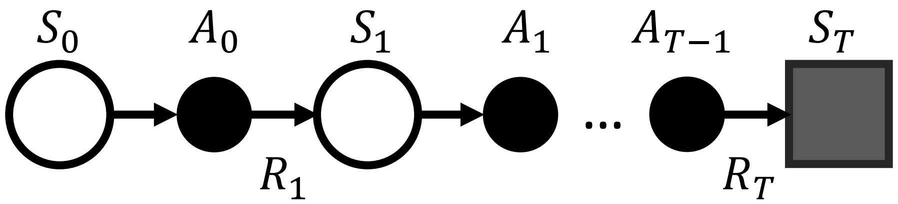
Bellman Equation for v_π

Monte Carlo Methods

- **Monte Carlo** methods learn directly from our agent’s experience.
- How would a Monte Carlo method estimate the value of a state S_0?
  - **Sample many episodes of experience** starting from S_0 following policy π.

$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3 \dots, R_T$$

  - Estimate $v_\pi(S_0)$ by **averaging the returns** our agent observes after visiting S_0, computed across all the sample trajectories.
- **Sample returns** may vary between episodes, but our answer will converge upon the true v_π if we average across enough episodes.



What information do we use to estimate the value of this state?

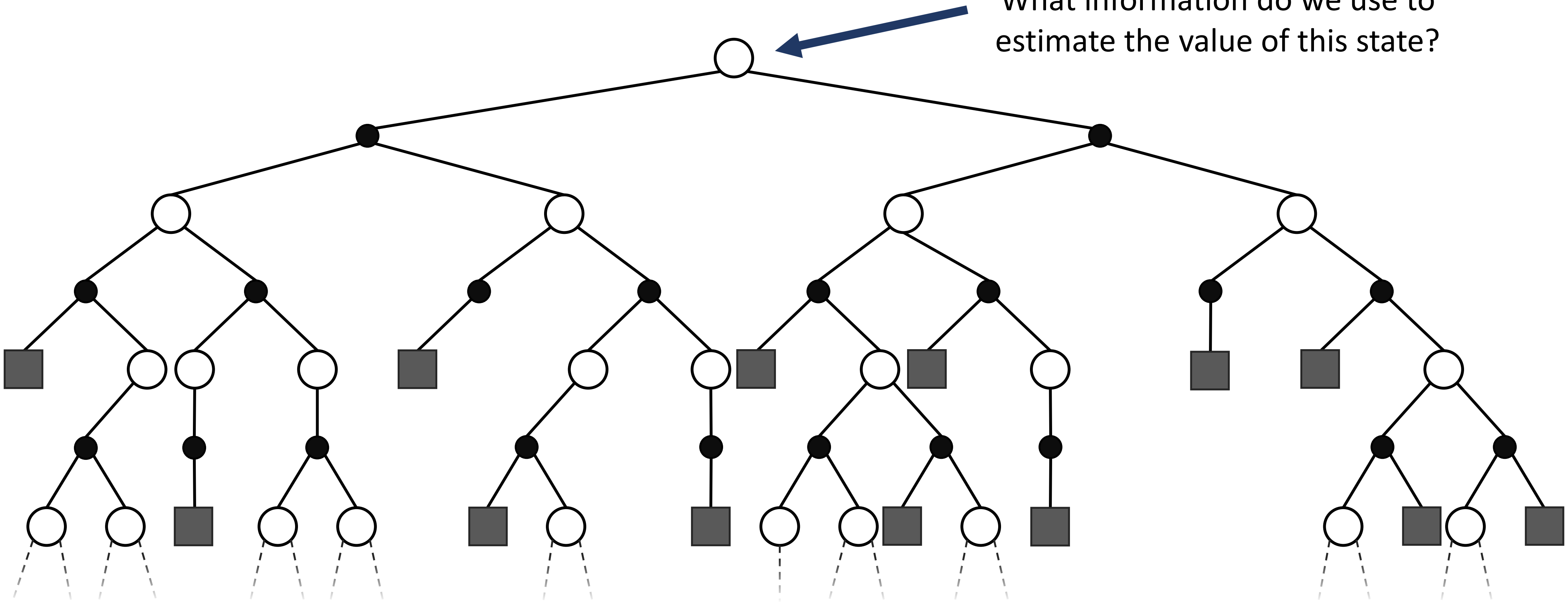
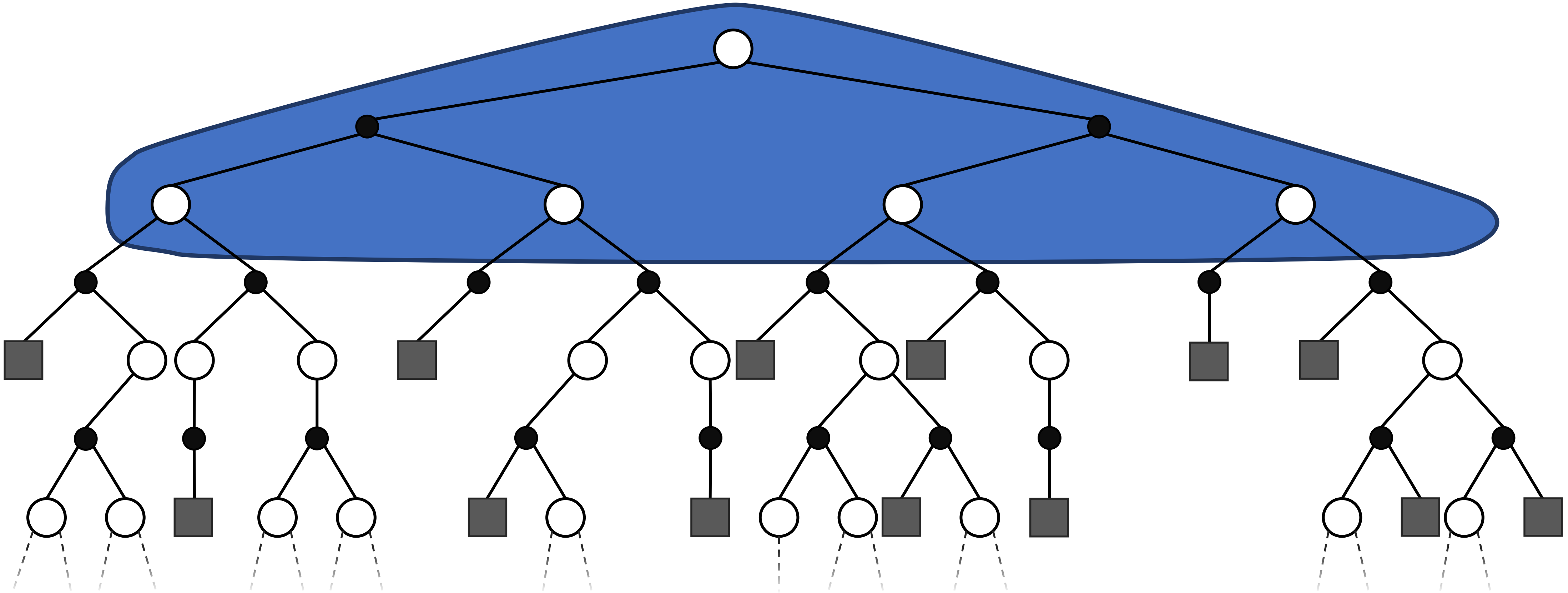


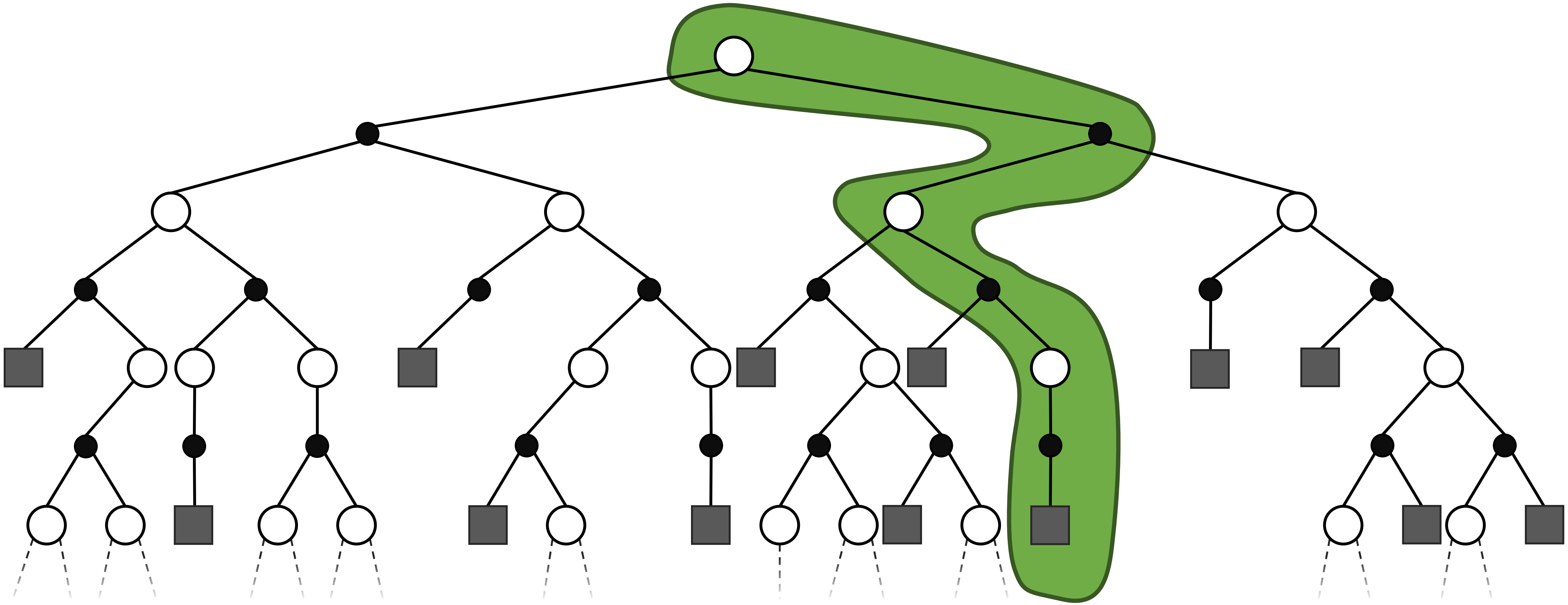
Figure adapted from Özgür Şimşek



Dynamic Programming Methods

We update the value of a state based on all the outcomes (i.e., immediate rewards r and next states s' that can be reached from it.
Requires **full knowledge** $p(s', r \mid s, a)$ of the environment.

Figure adapted from Özgür Şimşek



Monte Carlo Methods

We update the value of a state based on full sample returns generated by our agent after starting in that state.

Requires **full episodes** of experience, so can only be used with **episodic tasks**.

Figure adapted from Özgür Şimşek

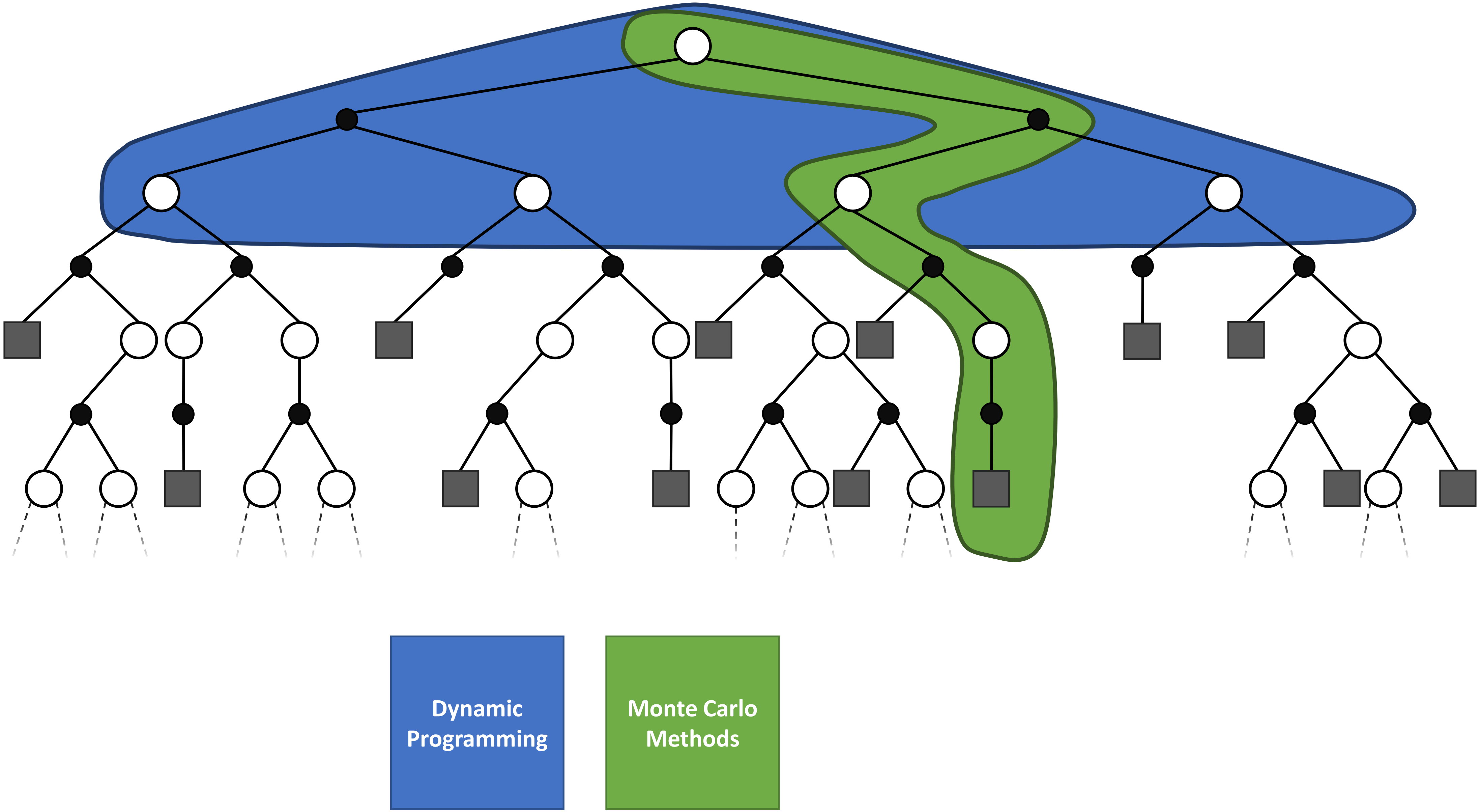


Figure adapted from Özgür Şimşek

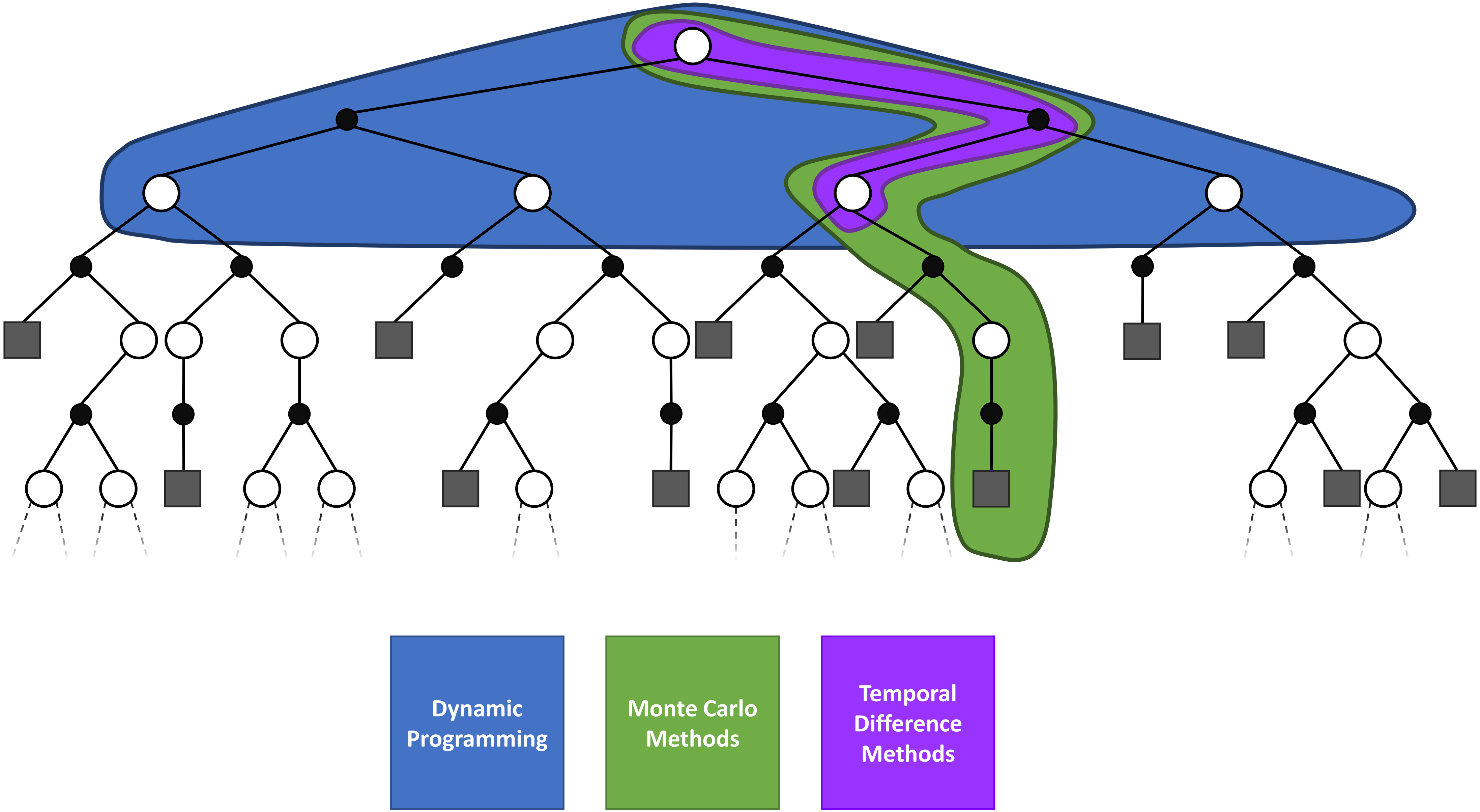
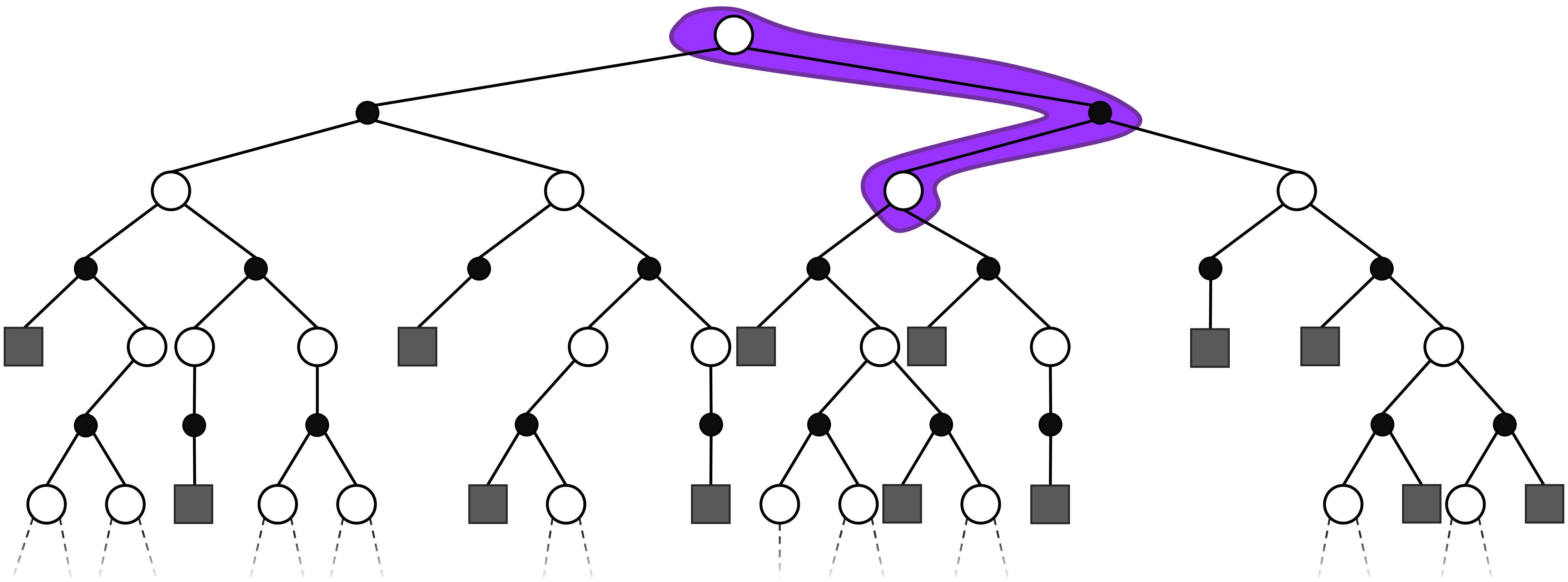


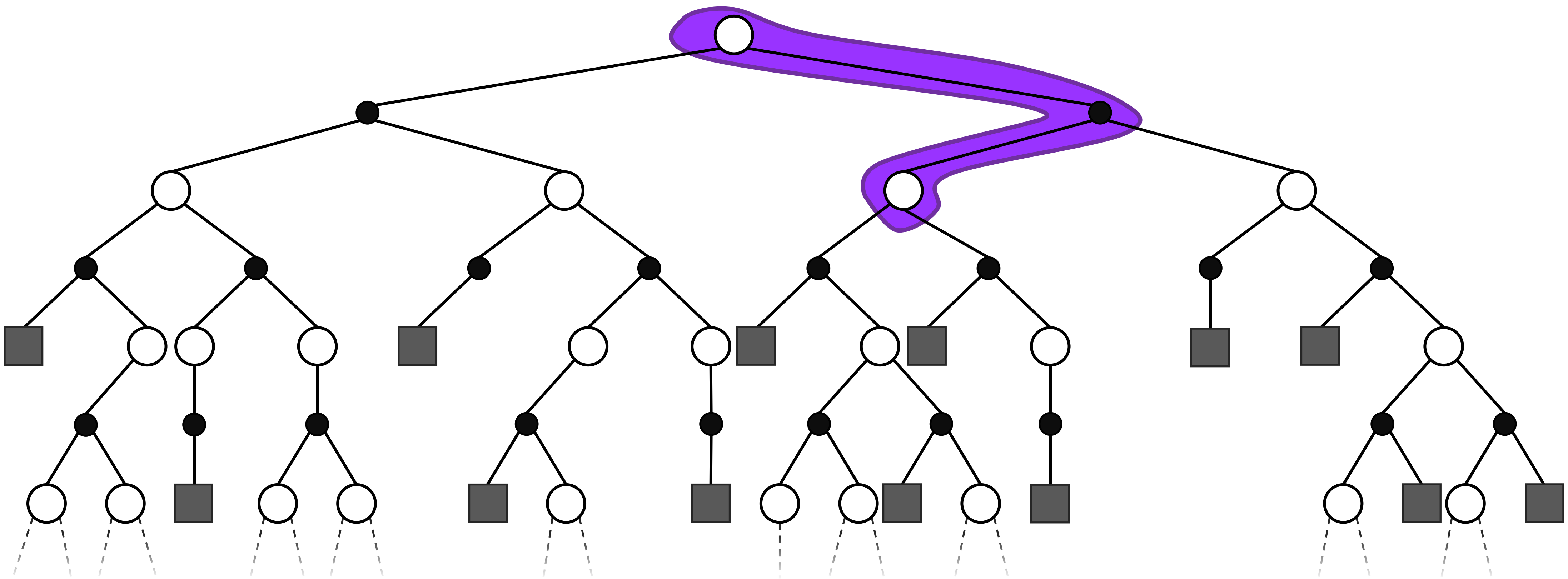
Figure adapted from Özgür Şimşek



Temporal-Difference Methods

We can update the value of a state using a single time-step of experience, based on the immediate reward earned and value of the next-state reached.

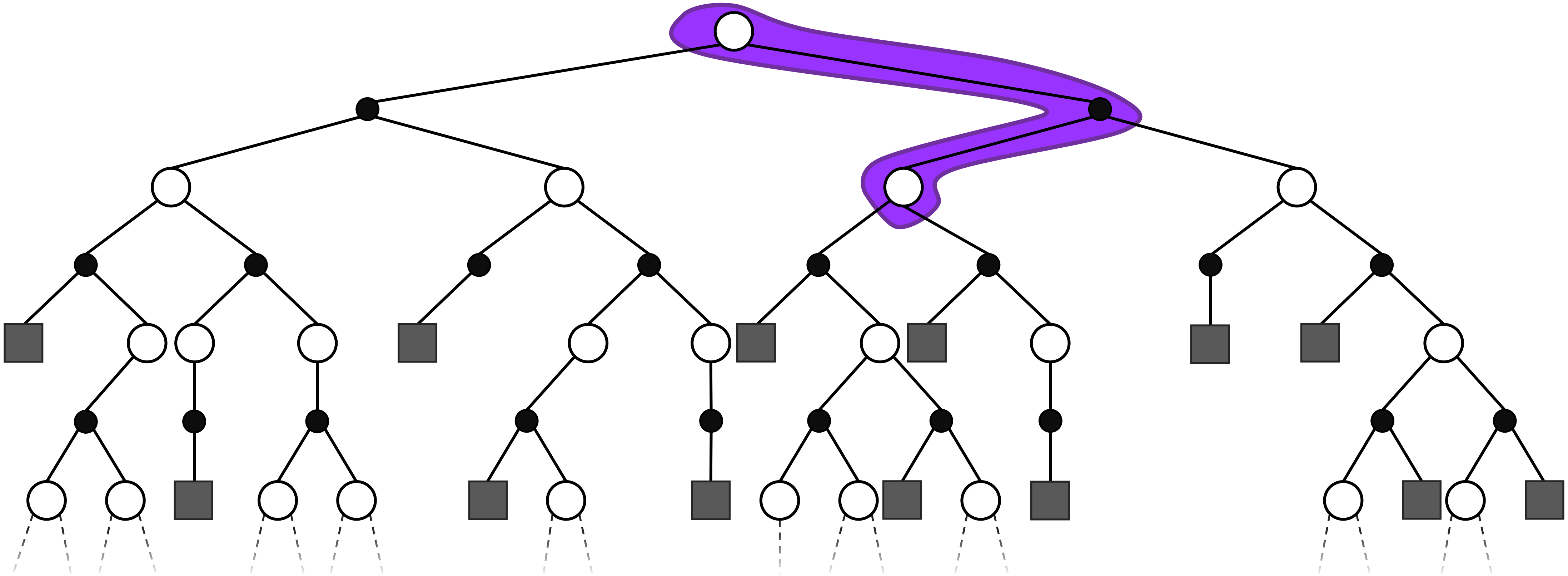
Figure adapted from Özgür Şimşek



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

Old Estimate **New Estimate (TD Target)**

Figure adapted from Özgür Şimşek



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

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

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

Figure adapted from Özgür Şimşek

Estimating Action-Values

- Updating **State-Value Estimates**

$$V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD Target}} - V(S_t)]$$

- Updating **Action-Value Estimates**

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{TD Target}} - Q(S_t, A_t)]$$

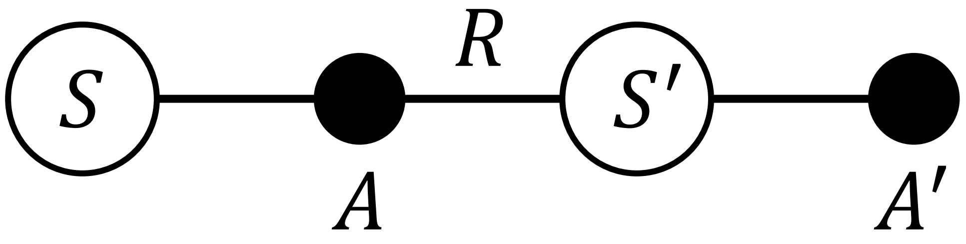
Exploration vs. Exploitation

- Our agent can't always do what it currently thinks is "best".
  - There might be better ways of doing things!
  - In other words, our agent needs to **explore**.
- To guarantee convergence, our agent needs to **maintain exploration**.
  - Given an infinite number of episodes, our agent should visit every state s and choose every action $a \in A(s)$ an infinite number of times.
- A simple solution is to use a **soft policy,** such as **ϵ-greedy**.
  - With probability $1 - \epsilon$, choose the **optimal** action.
  - With probability ϵ, choose a **random** action.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\textit{terminal}, \cdot) = 0$

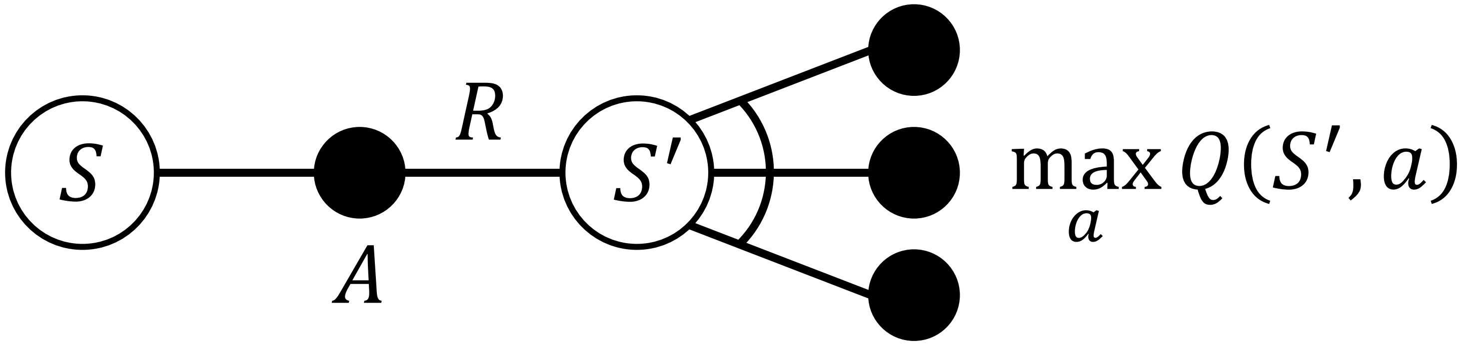
Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal



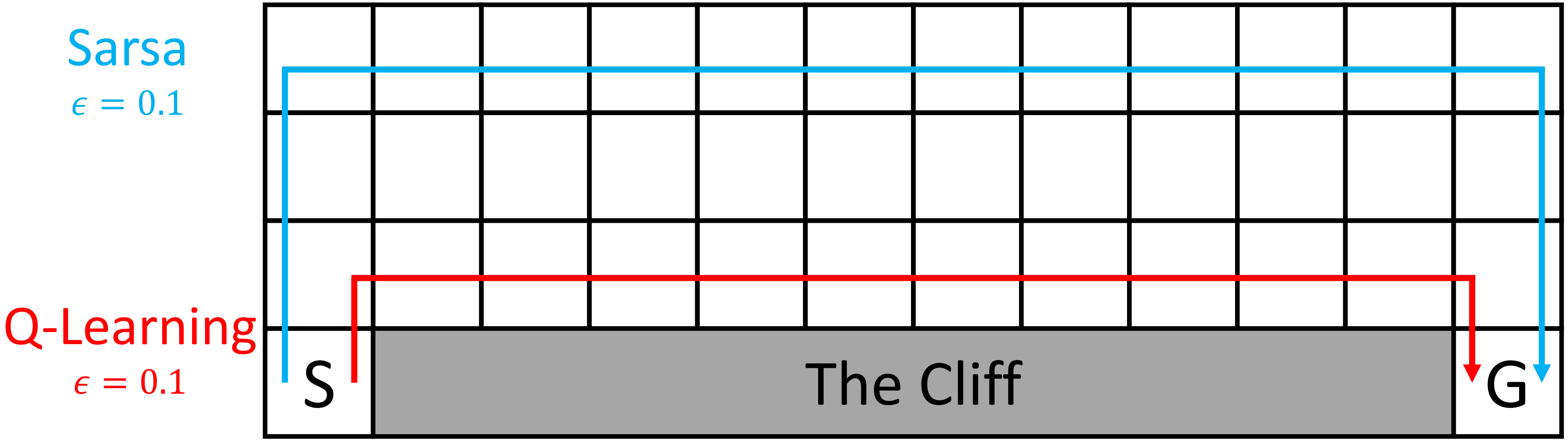
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

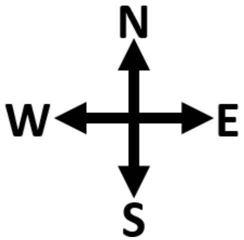
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal



Example: Cliff-Walking Problem





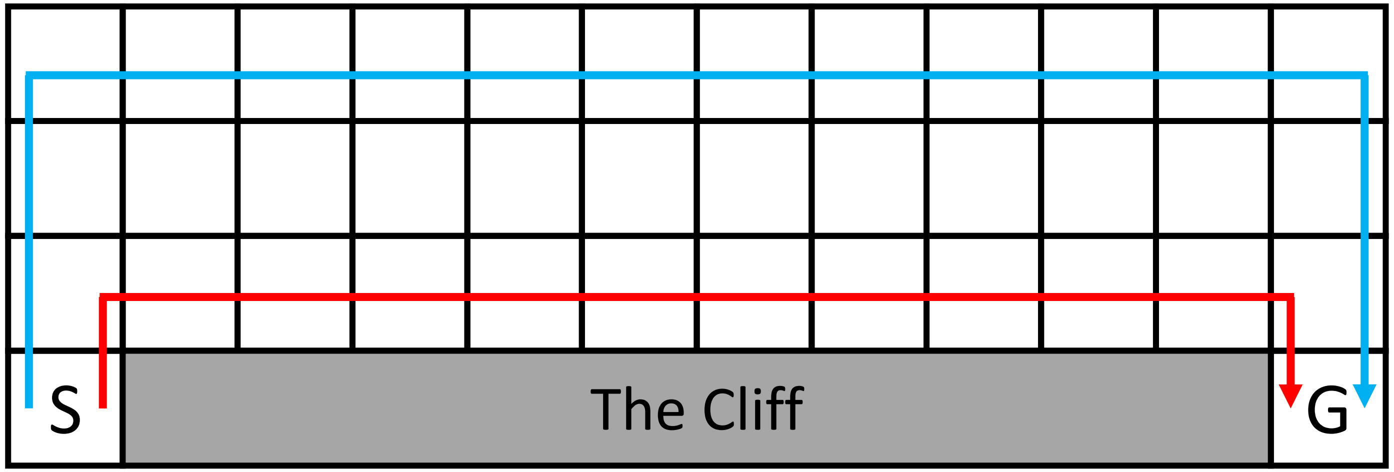
Deterministic Actions.
-1 per time-step.
-100 for falling off the cliff.

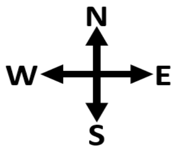
Figure from Sutton & Barto (2018)

Example: Cliff-Walking Problem

Sarsa
$\epsilon = 0.1$

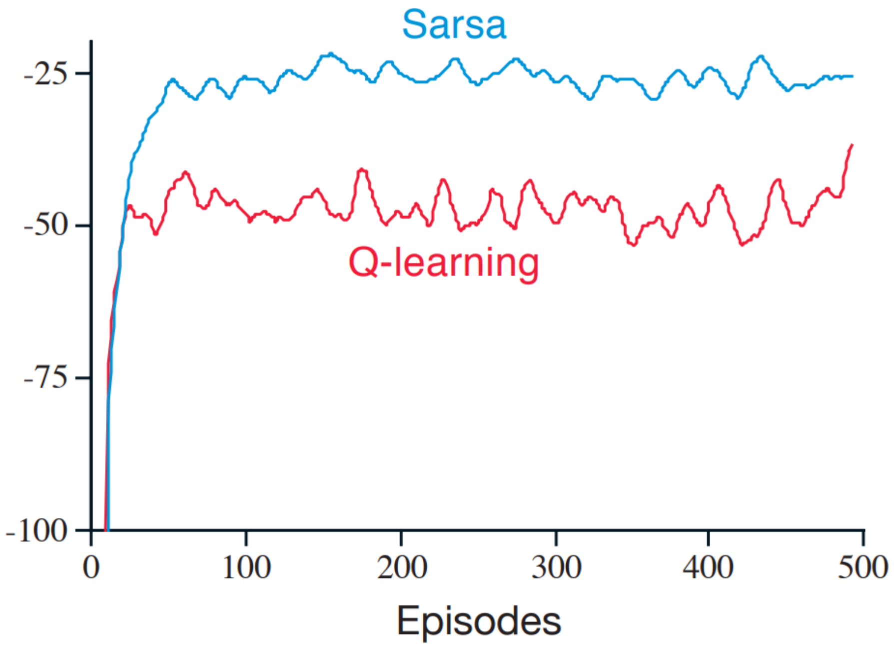
Q-Learning
$\epsilon = 0.1$





Deterministic Actions.
-1 per time-step.
-100 for falling off the cliff.

Sum of rewards per episode **during training**.



Figures from Sutton & Barto (2018)

Advantages of TD Learning

- TD methods **do not require a model of the environment**.
  - They can **learn using only our agent’s experience**, like MC methods.

- TD methods **can be fully incremental**.
  - They **bootstrap**, like DP methods.
  - We can learn **before** the end of an episode.
  - We can learn **without** reaching the end of an episode.
  - We can learn from **fragments of experience** shorter than full episodes.

In Today’s Lecture

MDP | Policy | Return | Value

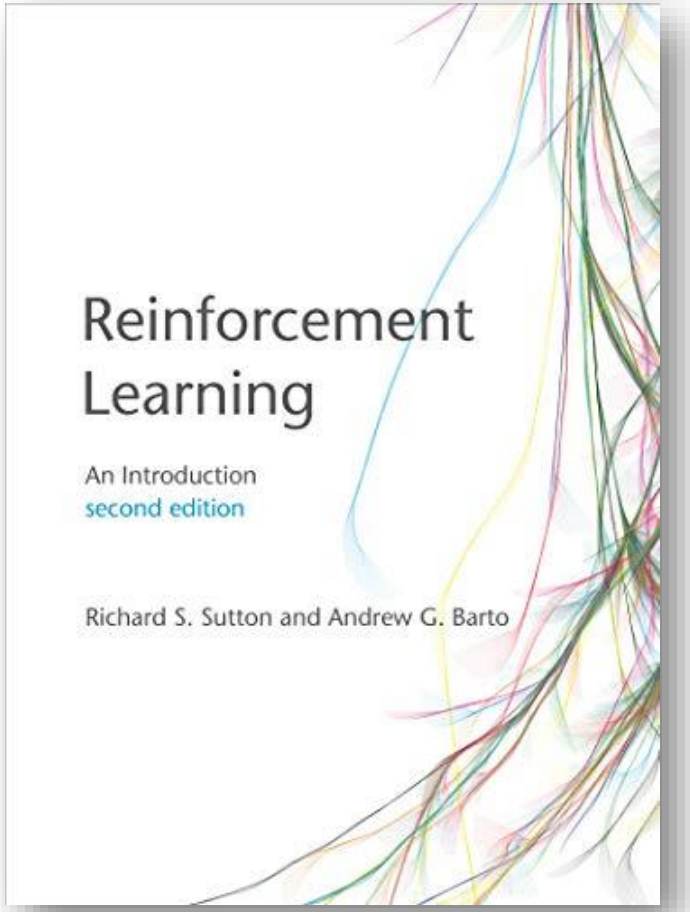
Dynamic Programming Methods | Monte Carlo Methods | Temporal-Difference Methods

Sarsa | Q-Learning

In Tomorrow’s Lecture

Applying these fundamental concepts to solve larger, more complex problems.

Active research topics and state-of-the-art in reinforcement learning.



Reinforcement Learning: An Introduction (Second Edition)
Richard S. Sutton & Andrew G. Barto

If you'd like to dive deeper into what was introduced today, I'd strongly recommend reading through Chapters 1, 3, 4, 5, 6.

Available for free at: http://incompleteideas.net/book/the-book.html