The advantage of using eligibility traces

Table of contents

Introduction

In this first computational exercise session, you will learn how eligibility traces can lead to more efficient training. As you have discussed in the lecture, standard temporal-difference methods, such as Q-Learning or Sarsa, leverage bootstrapping and update Q-values based on the consistency relation derived by the Bellman equation, i.e.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{1}$$

for Q-Learning and

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2}$$

for Sarsa. These kind of updates do not take into account the history of training before time t and therefore, their performance slows down for refined discretization schemes of a given environment.

Eligibility traces are one of the basic mechanisms of reinforcement learning. They are implemented by defining a shadow variable $e(s, a)$ for each state-action pair (s, a) and they can be combined with almost any temporal-difference (TD) method, such as Q-learning or Sarsa, to obtain a more general method that learns more quickly and more efficiently. When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end and one-step TD methods at the other.

An eligibility trace is a temporary record of the occurrence of an event, such as the visiting of a state or the taking of an action. The trace marks the memory parameters associated with the event as eligible for undergoing learning changes. When a TD error occurs, only the eligible states or actions are assigned credit or blame for the error. This helps propagating the information back from the rewarded states to the initial states in a faster and more robust way.

For the sake of convenience, you find below the pseudocode for Sarsa(λ) with eligibility traces.

$\text{For } i = 1, \cdots, n$
    $\text{Set } e(s, a) = 0 \, \forall (s, a)$
    $\text{While current episode is not ended}$
        $\text{Rescale all traces } e(s, a) = \lambda e(s, a) \, \forall (s, a)$
        $\text{Choose } a_t \sim \pi(\cdot | s_t), \text{ observe } r_t \text{ and } s_{t+1}$
        $e(s_t, a_t) = e(s_t, a_t) + 1$
        $\text{Update } Q(s, a) = Q(s, a) + \alpha * (r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) e(s, a) \, \forall (s$
        $t \leftarrow t + 1$
    End while
End for

Imports and examples

Please import the **`TMaze`** environment from `environment1.py`.

In [1]:

```
import matplotlib.pyplot as plt

# Load the autoreload extension
%load_ext autoreload
%autoreload 2

# Import the BinaryTreeMaze environment
from environments.environment1 import TMaze
```

The environment used in this exercise session is a T Maze. You can find a sketch of the environment in `environment1.py` or by running the cell below. Starting from the bottom of the Maze, the agent's goal is basically to:

1. Learn quickly to arrive at the bifurcation of the T Maze.
2. Once there, learn the direction (left or right) giving the highest reward.

The possible actions that the agent can take from a generic state are "u" (up), "d" (down), "l" (left) or "r" (right). Infeasible moves, such as going down, left or right from the initial state, are forbidden. Reaching the goal state on the left gives a +1 reward, while the reward for reaching the right end-state of the bifurcation is +2. All other actions have a reward equal to 0.

Because we are interested in different discretizations of the maze to test the advantage of eligibility traces, to initialize the environment you should, in general, type

```
env = TMaze(a, b)
```

where a is the number of steps required from the bifurcation to the two rewarded states and b is the number of steps the agent must move up to arrive at the bifurcation from the initial state. Thus, the state representation follows this convention:

1. The initial state is the origin $(0, 0)$.
2. The bifurcation state is labelled as $(0, b)$.
3. The two rewarded states are in positions $(-a, b)$ (reward=1) and (a, b) (reward=2) respectively.

Additionally, the environment has the following methods:

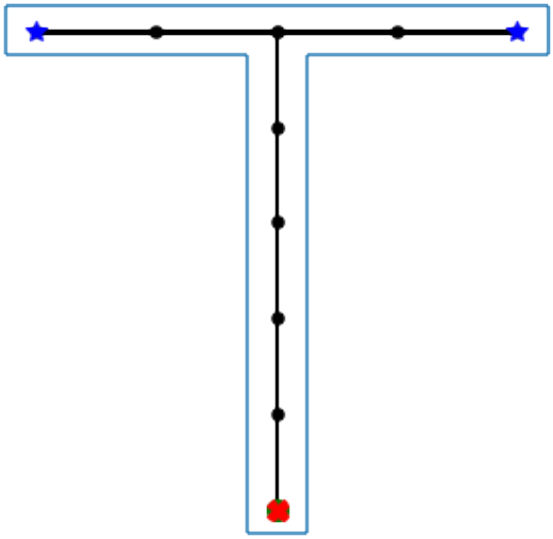
- `end` : Attribute of the class that becomes true when the environment is in one of the goal states.
- `get_state()` : Returns the current state.
- `reset()` : Reset the environment, i.e. set the state back to the origin $(0, 0)$ and set the accumulated reward to 0.
- `get_initial_state()` : Returns the starting point of every training episode. This statically returns $(0, 0)$.
- `get_num_actions()` : Returns the number of maximum available actions. This statically returns `4` for this environment.
- `get_num_states()` : Returns the number of possible states. This corresponds to

$2a + b + 1$ where a and b are defined above.

- `get_direct_path_len()` : Returns the length of the direct path from the starting state to the rewarded states, namely $a + b$.
- `available()` : Returns a list of available actions from the current state, i.e. a subset of `["u", "d", "l", "r"]` .
- `do_action(action)` : Takes the action `action` in the current state. It returns a tuple `(state, reward)` corresponding to the new state after the action is taken and the reward obtained by the agent along the transition.
- `reward()` : Returns the reward for getting to the current state. This is a reward of 1 if the agent reaches the goal state $(-a, b)$, a reward of 2 if the agent reaches the goal state (a, b), a reward of 0 otherwise.
- `encode_action(action)` : Maps the strings of the possible actions `["u", "d", "l", "r"]` to the integers `[0, 1, 2, 3]` .
- `inverse_encoding(action)` : Inverse map of `encode_action(action)` .
- `neighbours()` : Returns a list of the neighbours states of the current state of the environment.
- `render(Q=None)` : Prints the current game state. If no set of Q-values is passed (run the cell below for an example), this functions simply plots the environment: the starting state is marked by a green circle, the two goal states are represented by a blue star and the current position of the agent by a red cross. If a set of Q-values, it shows a heatmap of the Q-values passed in input (see examples below after your implementation of the algorithms).
- `render(Q=None)` : Prints the current game state. If no set of Q-values is passed (run the cell below for an example), this functions simply plots the environment: the starting state is marked by a green circle, the two goal states are represented by a blue star and the current position of the agent by a red cross. If a set of Q-values, it shows a heatmap of the Q-values passed in input (see examples below after your implementation of the algorithms).

In [2]:

```
# Initialize the environment with default parameters
env = TMaze(2, 5)

# Render the maze and show the plot
env.render()
plt.show()
```



Additionally to carrying out a comparison between TD algorithms with and without traces, you will also compare two different policies for action selection during training:

1. ϵ**-greedy policy**: given a state s and a set of Q-values Q, the ϵ-greedy policy chooses:

- with probability $1-\epsilon$ the action $a^\star = \text{argmax}_a Q(s,a)$, where ties are broken randomly;
- with probability ϵ a random action, where ties are broken randomly.

1. $\mathbf{Softmax\ policy}$: given a state s and a set of Q-values Q, it chooses the actions sampling from the probability distribution $\pi(\cdot|s)$ defined as

$$\pi(a|s) = \frac{\exp(\beta Q(s,a))}{\sum_{a'}\exp(\beta Q(s,a'))}, \tag{3}$$

where $\beta > 0$ is the so-called scaling parameter.

Implement utilities for action selection

In [3]:
```
from RL_algorithms.algorithms import epsilon_greedy, softmax_
```

Exercise 0: One step horizon

Start by considering a one step horizon reward scheme, i.e. the highest rewarded states are immediately reached after one step from the beginning. To do this, set $a = 1$ and $b = 0$ in the initialization of the environment. Let us import the Sarsa algorithm implementation.

In [4]:
```
from RL_algorithms.algorithms import sarsa
```

Before running any further experiment, answer the following questions:

1. For a learning rate $\alpha = 1$ and starting Q-values equal to zero, convergence to the exact Q-values can be retained in 2 steps if the agent chooses action "left" at the first episode and "right" at the second episode.
2. Let us denote by q^l and q^r the state-action values for the left and right actions, respectively. From the Sarsa update, we have the recurrence relations

$$q^r_{t+1} = q^r_t + \alpha(2 - q^r_t) = (1-\alpha)q^r_t + 2\alpha, \quad q^l_{t+1} = q^l_t + \alpha(1 - q^l_t) = (1-\alpha)q^l_t + \alpha$$

with $q^l_0 = q^r_0 = 0$. The above recurrence relations can be solved explicitly; this yields

$$q^r_t = 2\alpha \left(\frac{(1-\alpha)^t - 1}{(1-\alpha) - 1} \right) = 2(1 - (1-\alpha)^t), \quad q^l_t = \alpha \left(\frac{(1-\alpha)^t - 1}{(1-\alpha) - 1} \right) = 1 - (1$$

Thus, we simply need $1 - (1-\alpha)^t \geq 0.95$. Hence $(1-\alpha)^t \leq 0.05$, or equivalently $t \geq \left(\frac{\log(0.05)}{\log(1-\alpha)} \right)$.

1. For $\epsilon = 0$ (i.e. greedy policy) the agent will pick the same action in all the episodes after the first one, so one Q-value will remain to zero and convergence won't be retained.

Thus, the requested couple is $\epsilon = 0$ and $\alpha = 1$.

1. Using a soft-max policy makes sure that action probabilities are never zero, so we discard the above scenario in which with probability 1 always the same action is taken and the agent does not explore an entire part of the state space (left or right). An exploratory policy which guarantees that the agent does not remain stuck in a suboptimal policy (always receiving a reward equal to one) can be achieved by taking small values of β. High values of β yield a policy which approaches a deterministic one.

In [5]:

```
env = TMaze(1, 0)
alpha = 0.5  # alpha = 0.5 means at least 5 episodes for each action

###### GREEDY POLICY #########
print("============ GREEDY POLICY ==============")
Q, stats = sarsa(env, num_episodes=5, epsilon_exploration=0, alpha=alpha)
print("5 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=10, epsilon_exploration=0, alpha=alpha)
print("10 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=15, epsilon_exploration=0, alpha=alpha)
print("15 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=20, epsilon_exploration=0, alpha=alpha)
print("20 episodes ---> ", Q[(0, 0)])

###### 0.5 GREEDY POLICY #########
print("============ 0.5 GREEDY POLICY ==========")
Q, stats = sarsa(env, num_episodes=5, epsilon_exploration=0.5, alpha=alpha)
print("5 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=10, epsilon_exploration=0.5, alpha=alpha
print("10 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=15, epsilon_exploration=0.5, alpha=alpha
print("15 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=20, epsilon_exploration=0.5, alpha=alpha
print("20 episodes ---> ", Q[(0, 0)])

###### EXPLORATORY SOFTMAX POLICY #########
print("============ EXPLORATORY SOFTMAX POLICY ==============")
Q, stats = sarsa(env, num_episodes=5, action_policy='softmax_', epsilon_exp
print("5 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=10, action_policy='softmax_', epsilon_ex
print("10 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=15, action_policy='softmax_', epsilon_ex
print("15 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=20, action_policy='softmax_', epsilon_ex
print("20 episodes ---> ", Q[(0, 0)])

###### EXPLOITATORY SOFTMAX POLICY #########
print("============ EXPLOITATORY SOFTMAX POLICY ==============")
Q, stats = sarsa(env, num_episodes=5, action_policy='softmax_', epsilon_exp
print("5 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=10, action_policy='softmax_', epsilon_ex
print("10 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=15, action_policy='softmax_', epsilon_ex
print("15 episodes ---> ", Q[(0, 0)])
Q, stats = sarsa(env, num_episodes=20, action_policy='softmax_', epsilon_ex
print("20 episodes ---> ", Q[(0, 0)])
```

```
============ GREEDY POLICY ==============
5 episodes --->  [0.     0.     0.     1.9375]
10 episodes --->  [0.         0.         0.99902344 0.        ]
15 episodes --->  [0.         0.         0.99996948 0.        ]
20 episodes --->  [0.         0.         0.99999905 0.        ]
============ 0.5 GREEDY POLICY ==========
5 episodes --->  [0.   0.   0.75 1.75]
10 episodes --->  [0.        0.        0.75      1.9921875]
15 episodes --->  [0.         0.         0.99951172 1.875     ]
20 episodes --->  [0.         0.         0.875      1.99998474]
============ EXPLORATORY SOFTMAX POLICY ==============
5 episodes --->  [0.    0.    0.5   1.875]
10 episodes --->  [0.        0.        0.75      1.9921875]
15 episodes --->  [0.         0.         0.5        1.99987793]
20 episodes --->  [0.         0.         0.9375     1.99996948]
============ EXPLOITATORY SOFTMAX POLICY ==============
5 episodes --->  [0.     0.     0.     1.9375]
10 episodes --->  [0.         0.         0.         1.99804688]
15 episodes --->  [0.         0.         0.99996948 0.        ]
20 episodes --->  [0.         0.         0.99999905 0.        ]
```

Exercise 1: Implementation of TD algorithms

Implementation of Q-Learning(λ), Sarsa(λ) and n-step Sarsa

When implementing eligibility traces, the standard version of the algorithms are recovered for $\lambda = 0$. On the other hand, note that $\lambda = 1$ gives Monte Carlo methods for state-action value estimation.

In [6]:
```
from RL_algorithms.algorithms import q_learning, sarsa, n_step_sarsa
env = TMaze(2,5)
```

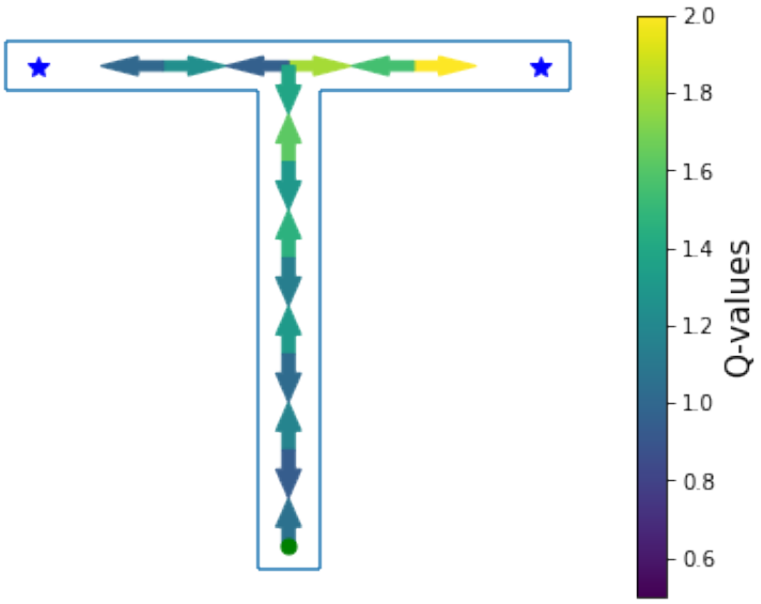
Test your algorithms

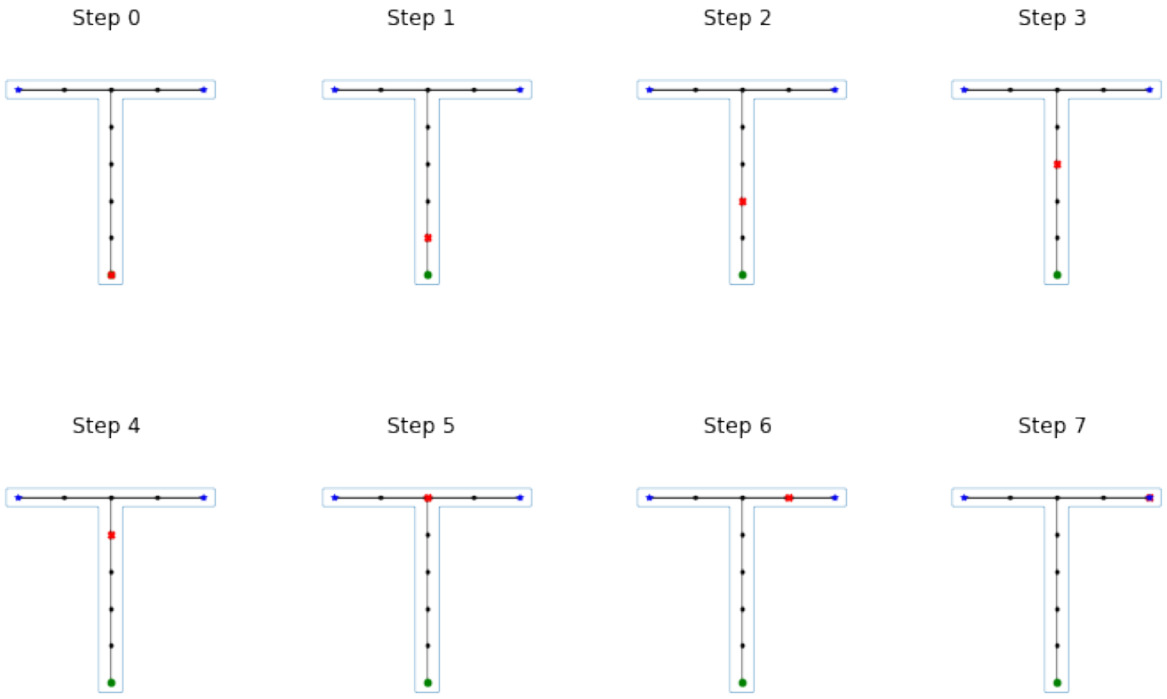
In [7]:
```
from RL_algorithms.utils import play
```

1. In 200 episodes the Q-values still have in general to converge. This can be seen for example by looking at the Sarsa(0) heatmap, in particular the Q-value for "correcting the direction at the branch" after moving left (i.e. action "right" in the first state on the left branch of the T Maze) has still a very small value, because it is rarely explored during training. These are the most difficult values to be updated if there is not enough exploration, because the route that takes the agent from this state directly to the high reward is very unlikely to be followed.
2. On the other hand, the policy has converged in all cases and the agent, by playing greedily with the learned Q-values, is able to reach the high reward directly in the minimum number of steps.
3. Given the large number of episodes played, there is no difference to be appreciated between the standard algorithms and the trace-decay algorithms.

In [8]:

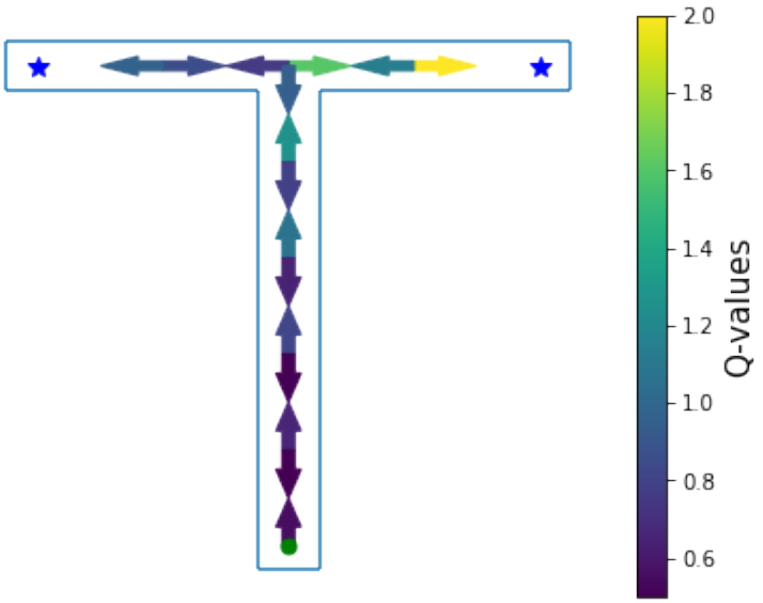
```
# Q-Learning(0)
Q, stats = q_learning(env, gamma=0.9, num_episodes=200, epsilon_exploratior
env.render(Q)
play(env, Q)
```

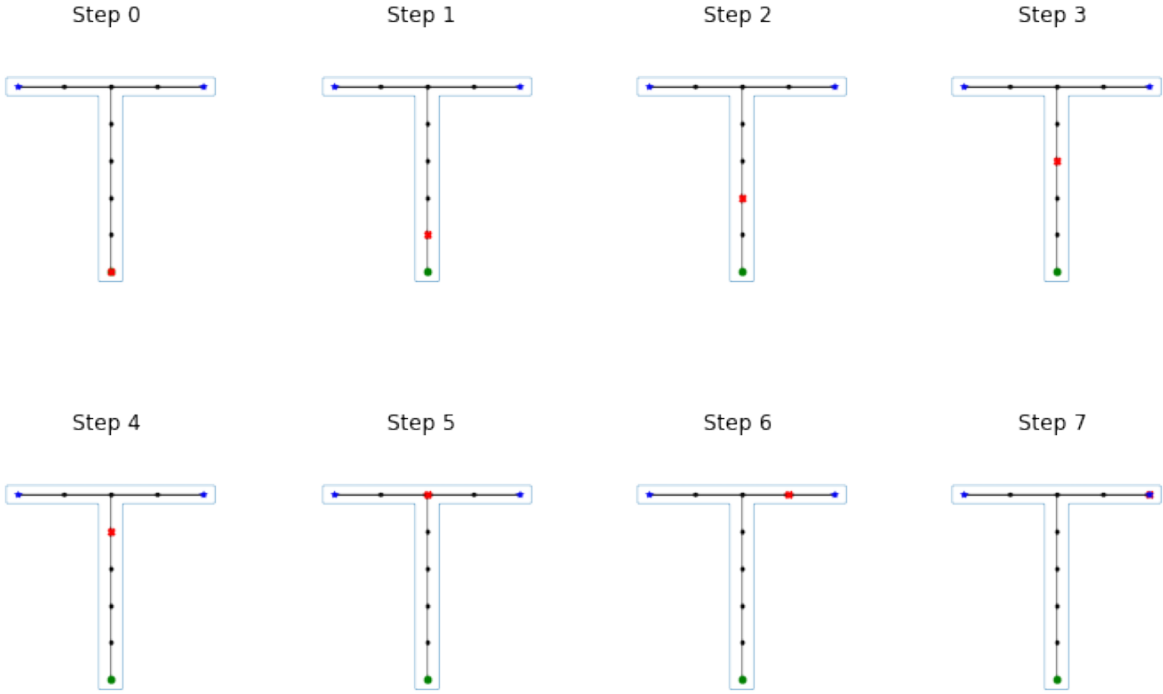




In [9]:

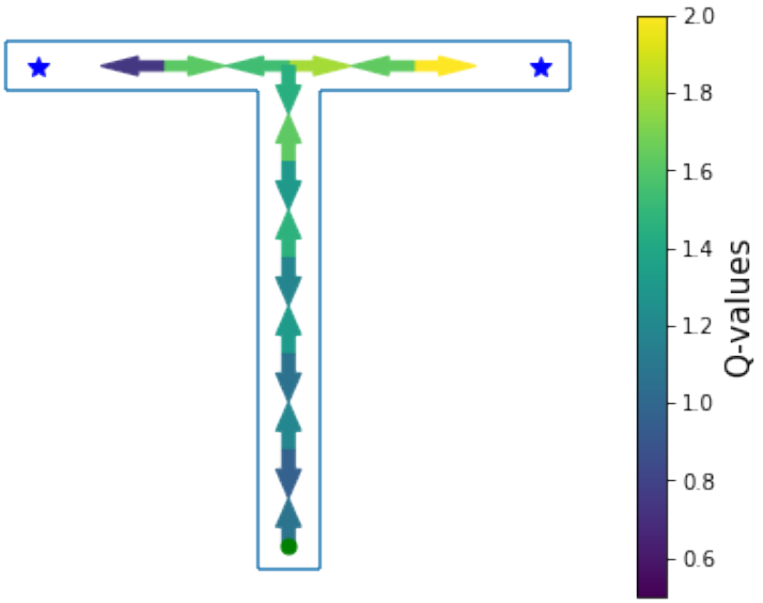
```
# Sarsa(0)
Q, stats = sarsa(env, gamma=0.9, num_episodes=200, epsilon_exploration=0.5,
env.render(Q)
play(env, Q)
```

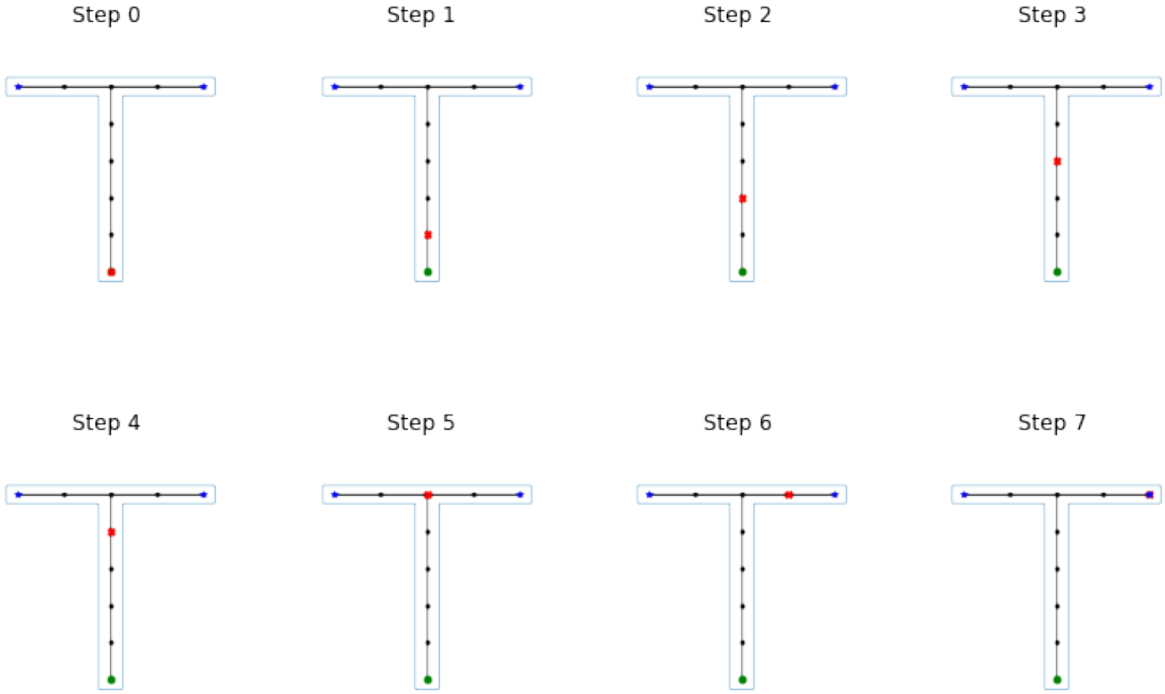




In [10]:

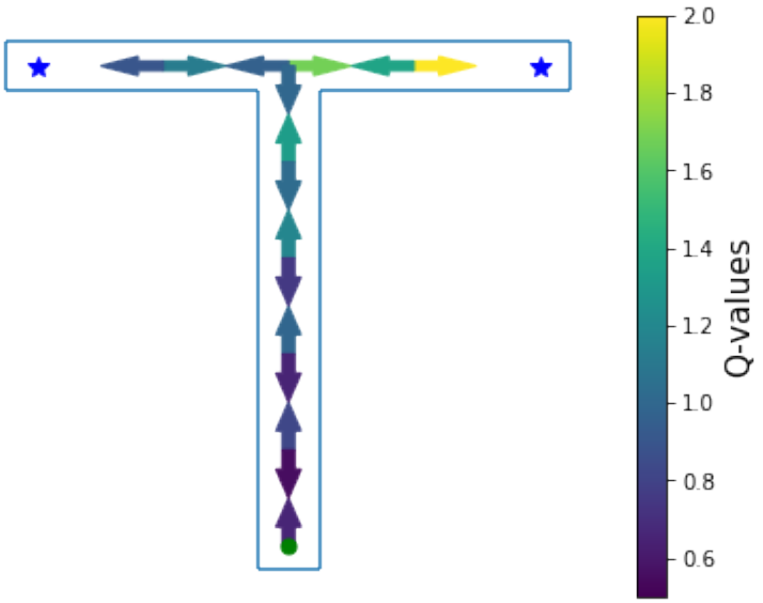
```
# Q-Learning(\lambda)
Q, stats = q_learning(env, gamma=0.9, num_episodes=200, epsilon_exploratior
env.render(Q)
play(env, Q)
```

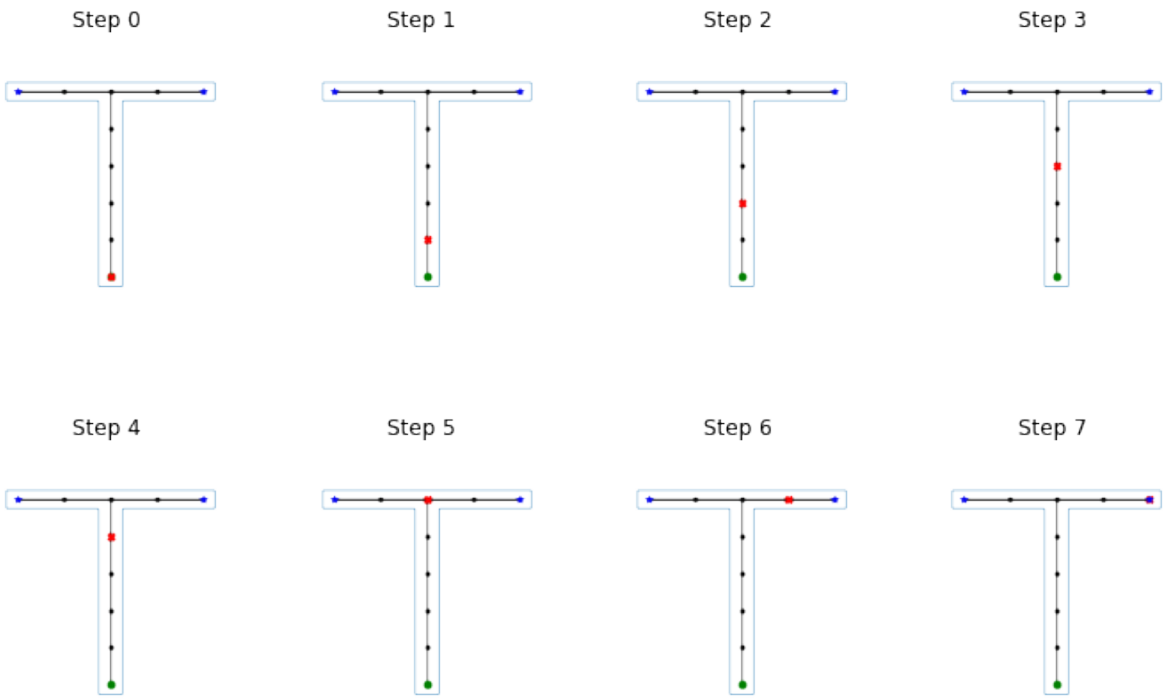




In [11]:

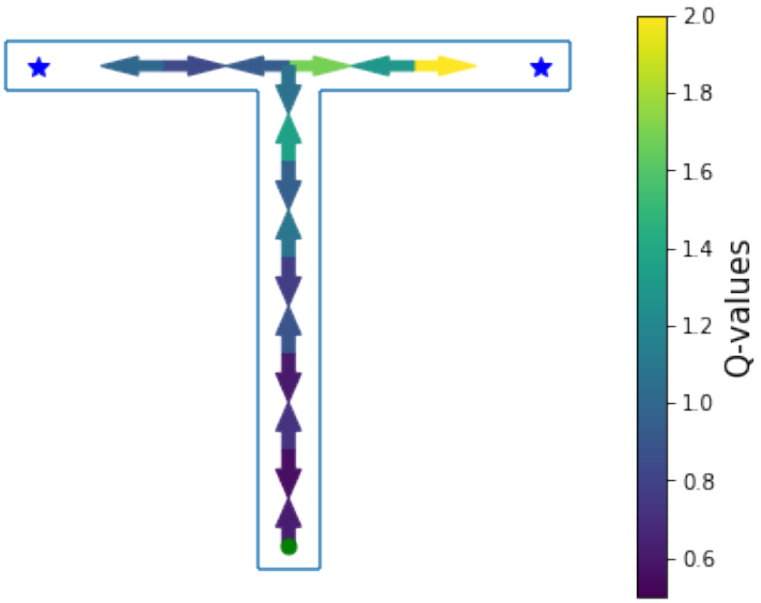
```
# Sarsa(\lambda)
Q, stats = sarsa(env, gamma=0.9, num_episodes=200, epsilon_exploration=0.5,
env.render(Q)
play(env, Q)
```

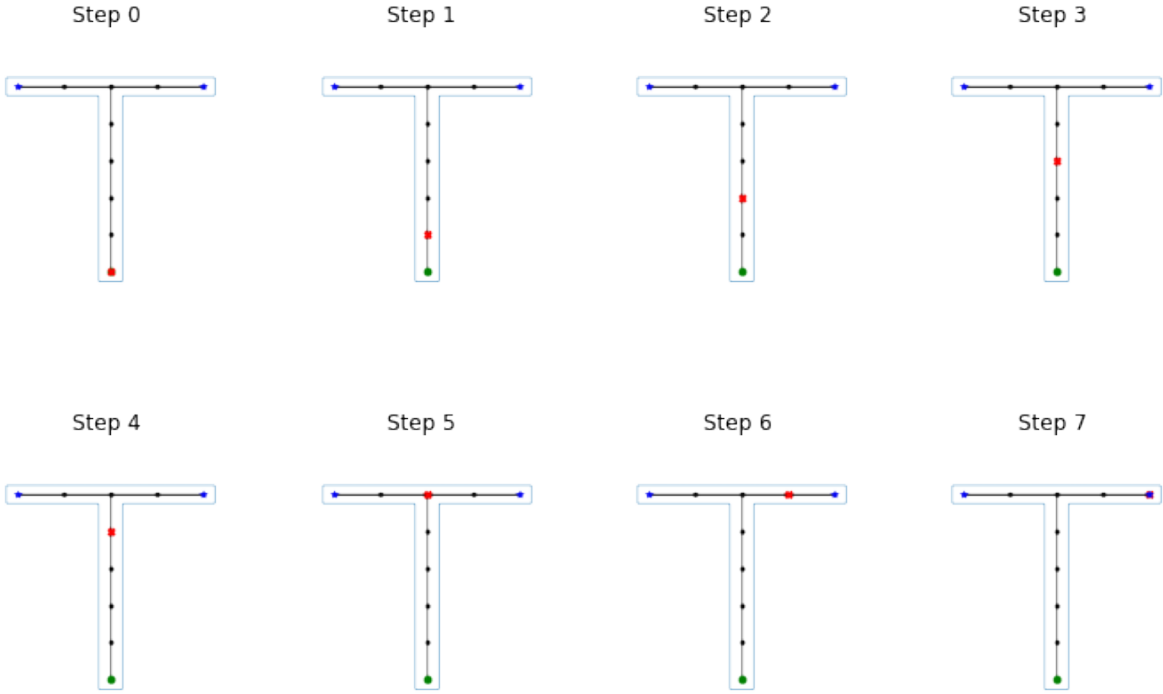




In [12]:

```
# n-step Sarsa
Q, stats = n_step_sarsa(env, gamma=0.9, num_episodes=200, epsilon_explorati
env.render(Q)
play(env, Q)
```





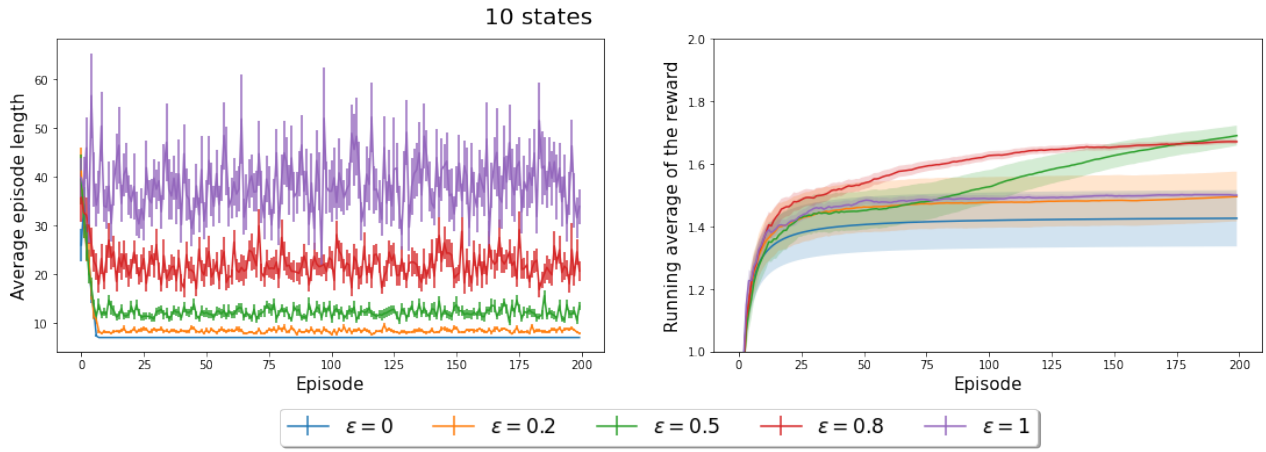
Exercise 2: Exploration-Exploitation dilemma

1. Assume $\epsilon = 0$, i.e. the agent always chooses the best action available according to the current estimates of the Q-values. The problem with this kind of approach, given that the Q-values are all initialized to zero, is that one of the goal states (and possibly other states/actions too) remains unexplored during training. At the first episode the agent will update the Q-values to one goal state and the Q-value of all state-action pairs leading to the other goal state remain zero, so with a greedy policy, the agent will never try these other actions/states in the next episodes.
2. A greedy policy performs badly, as just explained above. On the other hand, taking ϵ too large may slow down learning because "good/promising paths" are not explored preferentially. Intermediate values should provide the best trade-off between exploration and exploitation. As a side note, a common strategy which you may try and implement is to decay ϵ over time, as to favour exploration during the first episodes and exploitation at the end of training.

The numerical results are showing, coherently with the discussion above, that $\epsilon = 0.5$ is the best exploration rate.

In [13]:

```
from RL_algorithms.utils import *

# Hyperparameters
gamma = 0.9
alpha = 0.1
trace_decay = 0.8
num_avg = 30

params = {
    'gamma' : gamma,
    'alpha' : alpha,
    'action_policy': "epsilon_greedy"
}

Q_LEARNING_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\epsilon = 0$',
    'params': {**params, 'epsilon_exploration': 0}
}

Q_LEARNING_20_PERCENT_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\epsilon = 0.2$',
    'params': {**params, 'epsilon_exploration': 0.2}
}

Q_LEARNING_50_PERCENT_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\epsilon = 0.5$',
    'params': {**params, 'epsilon_exploration': 0.5}
}

Q_LEARNING_80_PERCENT_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\epsilon = 0.8$',
    'params': {**params, 'epsilon_exploration': 0.8}
}

Q_LEARNING_RANDOM = {
    'algo_name': 'q_learning',
    'name': r'$\epsilon = 1$',
    'params': {**params, 'epsilon_exploration': 1}
}


algorithms = [Q_LEARNING_GREEDY, Q_LEARNING_20_PERCENT_GREEDY, Q_LEARNING_5
              Q_LEARNING_80_PERCENT_GREEDY, Q_LEARNING_RANDOM]

env = TMaze(2, 5)
compare_episodes_lengths_and_rewards(env=env, algos=algorithms, num_avg=nur
                                     show_std=True, additional_params=[{"nu
```

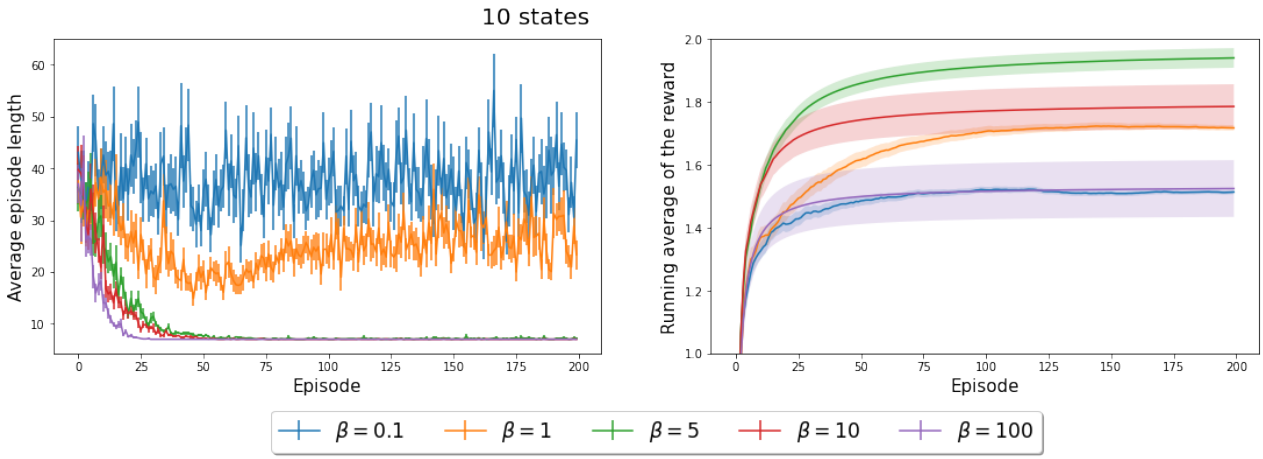


1. Small values of β should guarantee a better exploration during training. Very high values of β (e.g. 50 or 100) should yield an almost greedy policy.
2. Similarly to the discussion above for the exploration rate ϵ with a greedy policy, intermediate values of β should provide the best trade-off.

The best value of β according to the numerical experiments is indeed $\beta = 5$.

In [14]:

```
from RL_algorithms.utils import *

# Hyperparameters
gamma = 0.9
alpha = 0.1
num_avg = 30

params = {
    'gamma' : gamma,
    'alpha' : alpha,
    'action_policy': 'softmax_'
}

Q_LEARNING_NON_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\beta = 0.1$',
    'params': {**params, 'epsilon_exploration': 0.1}
}

Q_LEARNING_1_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\beta = 1$',
    'params': {**params, 'epsilon_exploration': 1}
}

Q_LEARNING_5_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\beta = 5$',
    'params': {**params, 'epsilon_exploration': 5}
}

Q_LEARNING_10_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\beta = 10$',
    'params': {**params, 'epsilon_exploration': 10}
}

Q_LEARNING_100_GREEDY = {
    'algo_name': 'q_learning',
    'name': r'$\beta = 100$',
    'params': {**params, 'epsilon_exploration': 100}
}



algorithms = [Q_LEARNING_NON_GREEDY, Q_LEARNING_1_GREEDY, Q_LEARNING_5_GREE

env = TMaze(2, 5)
compare_episodes_lengths_and_rewards(env=env, algos=algorithms, num_avg=nun
                                     show_std=True, additional_params=[{"nu
```



Exercise 3: Comparison of RL algorithms for different discretization schemes

As stated in the introduction, the main goal of this computational exercise session is to understand how the coupling of TD algorithms with eligibility traces can provide benefits for the learning of an agent with respect to the case $\lambda = 0$.

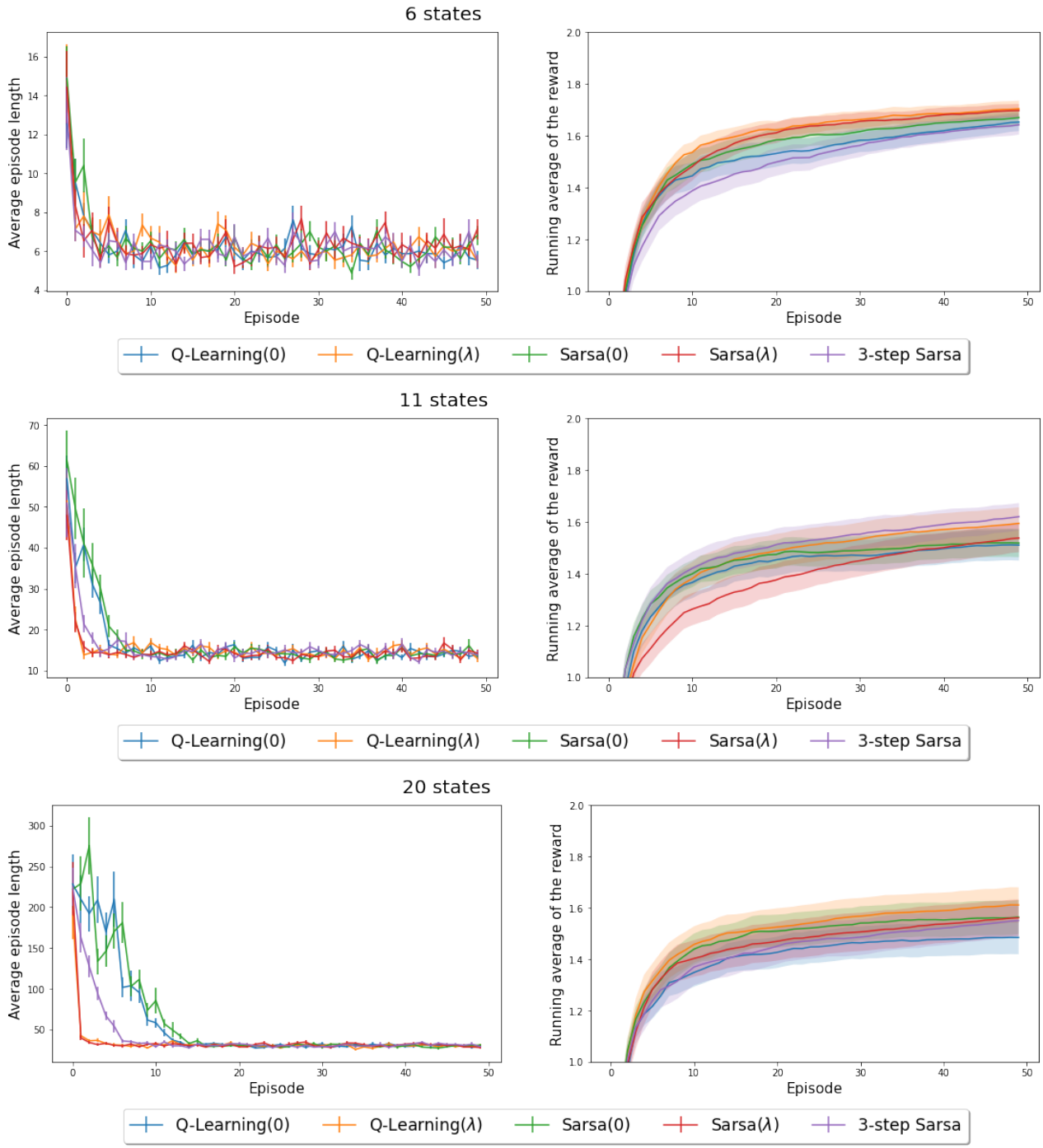
1. In the case $\lambda = 0$ on a T-Maze(a, b), at least $a + b$ episodes are needed to propagate back the reward information from the goal states to the origin: at every episode the information is propagated one step closer to the origin.
2. For $\lambda \neq 0$, this happens already at the end of the first episode: as soon as the agent reaches one of the goal states and gets a reward, all Q-values on the path of the episode are updated.

We now consider the best value of $\epsilon = 0.5$ for the experiments below with a ϵ-greedy policy.

1. The performance of Q-Learning and Sarsa without any eligibility traces is not independent of the discretization, because with a larger number of states more episodes are needed to backpropagate information to the origin.
2. This can be mitigated by Q-Learning(λ) and Sarsa(λ), as we discussed above. However, we cannot claim that the performance is independent of the discretization scheme, because to have the same speed of backpropagation we should maintain the quantity $\lambda^{(a+b)}$ constant, being a and b defined as above. You will discuss this later in the course, but could you explain already now why this quantity should be always the same when changing discretization scheme?

In [15]:

```
from RL_algorithms.utils import *

# Hyperparameters
epsilon_exploration = 0.5
gamma = 0.9
alpha = 0.1
trace_decay = 0.8
num_avg = 30

params = {
    'epsilon_exploration' : epsilon_exploration,
    'gamma' : gamma,
    'alpha' : alpha,
}

Q_LEARNING = {
    'algo_name': 'q_learning',
    'name': 'Q-Learning(0)',
    'params': params,
}

Q_LEARNING_LAMBDA = {
    'algo_name': 'q_learning',
    'name': 'Q-Learning($\lambda$)',
    'params': {**params, 'trace_decay': trace_decay}
}

SARSA = {
    'algo_name': 'sarsa',
    'name': 'Sarsa(0)',
    'params': params
}

SARSA_LAMBDA = {
    'algo_name': 'sarsa',
    'name': 'Sarsa($\lambda$)',
    'params': {**params, 'trace_decay': trace_decay}
}

THREE_STEP_SARSA = {
    'algo_name': 'n_step_sarsa',
    'name': '3-step Sarsa',
    'params': {**params, 'n': 3}
}

algorithms = [Q_LEARNING, Q_LEARNING_LAMBDA, SARSA, SARSA_LAMBDA, THREE_STE

couples = ((1, 3), (2, 6), (3, 13))
for j, couple in enumerate(couples):
    env = TMaze(*couple)
    compare_episodes_lengths_and_rewards(env=env, algos=algorithms, num_avg
                                         show_std=True, additional_params=[
```



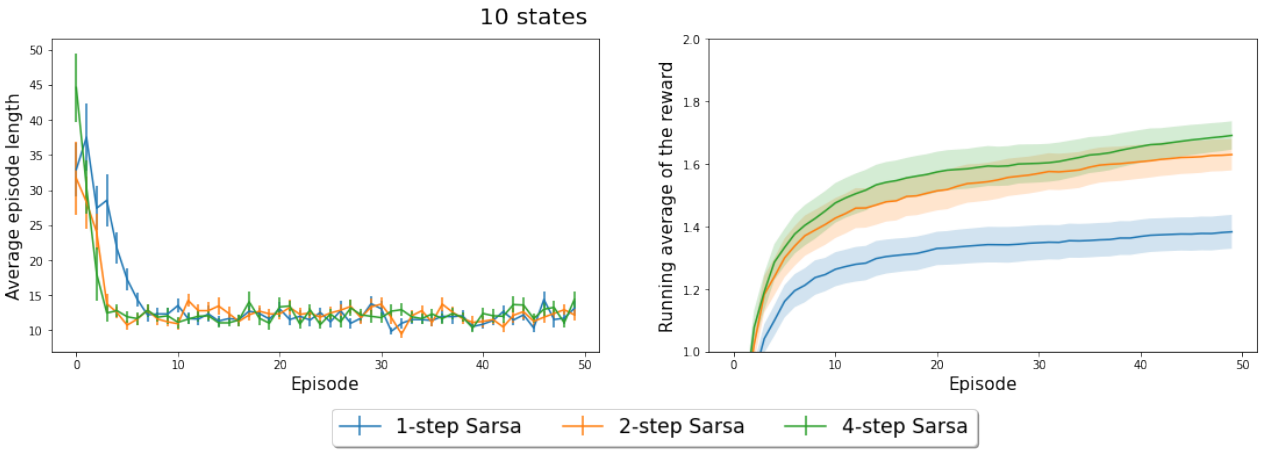
Exercise 4: Rescaling of the trace decay and step parameters

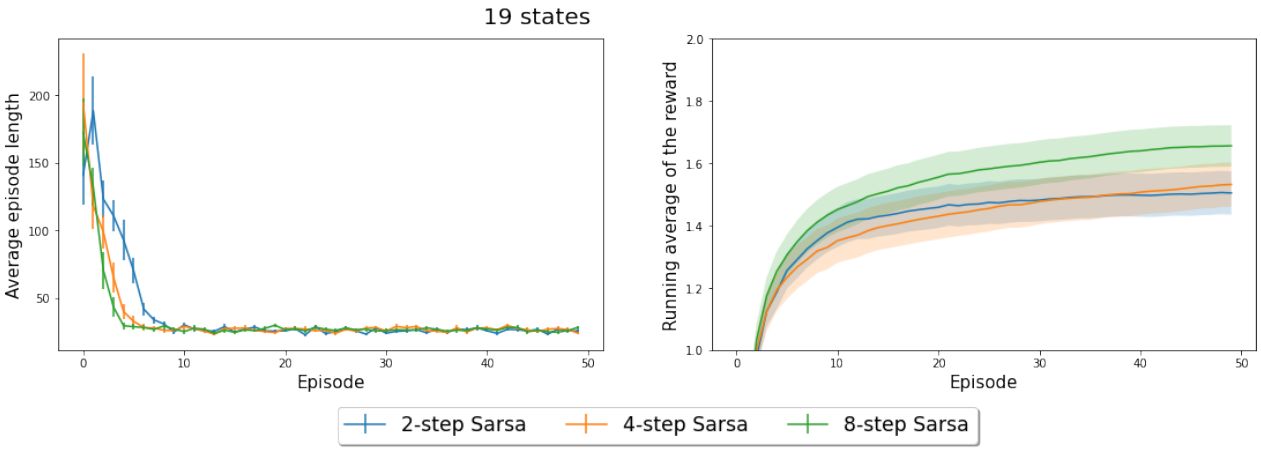
1. As we already claimed above, the relation between s and the number of steps needed to backpropagate information to the origin is linear. Thus, for $s \leftarrow 2s$, we also need $n \leftarrow 2n$.

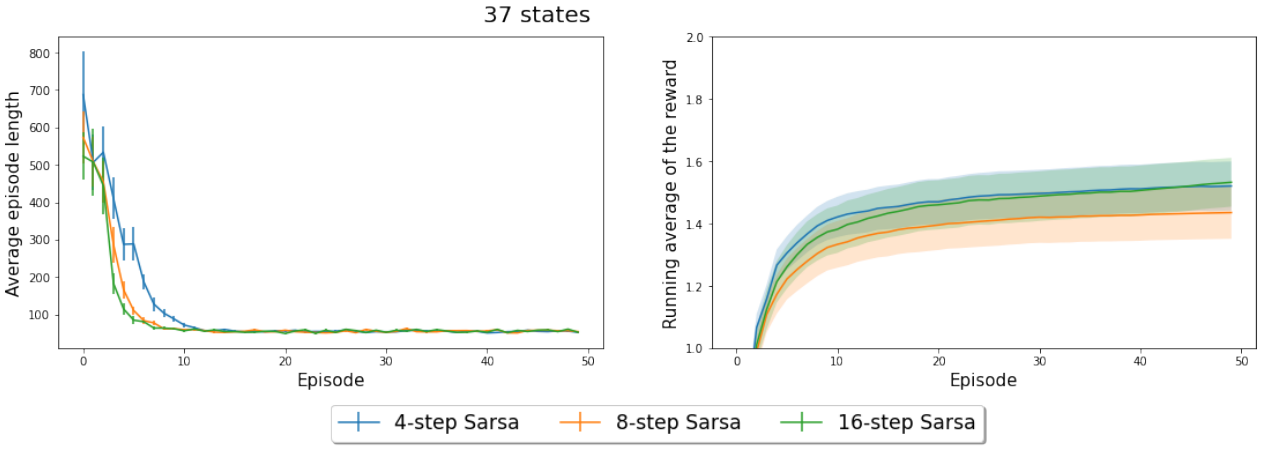
The numerical results support the claim. Look in particular at the episode lengths plot.

In [16]:

```
# Hyperparameters
epsilon_exploration = 0.5
gamma = 0.9
alpha = 0.1
num_avg = 30

params = {
    'epsilon_exploration' : epsilon_exploration,
    'gamma' : gamma,
    'alpha' : alpha,
    'num_episodes': 50
}

ONE_STEP_SARSA = {
    'algo_name': 'n_step_sarsa',
    'name': '1-step Sarsa',
    'params': {**params, 'n': 1}
}

TWO_STEP_SARSA = {
    'algo_name': 'n_step_sarsa',
    'name': '2-step Sarsa',
    'params': {**params, 'n': 2}
}

FOUR_STEP_SARSA = {
    'algo_name': 'n_step_sarsa',
    'name': '4-step Sarsa',
    'params': {**params, 'n': 4}
}

EIGHT_STEP_SARSA = {
    'algo_name': 'n_step_sarsa',
    'name': '8-step Sarsa',
    'params': {**params, 'n': 8}
}

SIXTEEN_STEP_SARSA = {
    'algo_name': 'n_step_sarsa',
    'name': '16-step Sarsa',
    'params': {**params, 'n': 16}
}

algorithms = {'0': [ONE_STEP_SARSA, TWO_STEP_SARSA, FOUR_STEP_SARSA],
              '1': [TWO_STEP_SARSA, FOUR_STEP_SARSA, EIGHT_STEP_SARSA],
              '2': [FOUR_STEP_SARSA, EIGHT_STEP_SARSA, SIXTEEN_STEP_SARSA]]

couples = ((2, 5), (4, 10), (8, 20))

for j, couple in enumerate(couples):
    env = TMaze(*couple)
    compare_episodes_lengths_and_rewards(env=env, algos=algorithms[str(j)],
                                         show_std=True, additional_params=[
```


10 states
Average episode length
Episode
Running average of the reward
Episode
1-step Sarsa
2-step Sarsa
4-step Sarsa


19 states
Average episode length
Episode
Running average of the reward
Episode
2-step Sarsa
4-step Sarsa
8-step Sarsa


37 states
Average episode length
Episode
Running average of the reward
Episode
4-step Sarsa
8-step Sarsa
16-step Sarsa