Assignment 2: Tabular RL

by: Johan Jiremalm

Describing the algorithms:

All algorithms are TDs.

Q-learning:

The algorithm achieves this optimization by iteratively updating the Q-values for each state-action pair using the Bellman optimality equation. The convergence point is when the Q-function reaches the optimal q_*. Notably, Q-Learning updates Q-values by selecting the maximum Q-value among all possible actions in the next state, which can lead to the overestimation of certain states.

The Bellman optimality equation for q* is expressed as:
$q_*(s, a) = E[R_{t+1} + \gamma max_{a'} q_*(s', a')]$

epsilon-greedy policy. This policy strikes a balance between exploration and exploitation by choosing the action with the highest Q-value with probability (1 - epsilon) and selecting a random action with probability epsilon.

The general update rule for Q-Learning can be summarized as follows:
Q(state, action) = Q(state, action) + learning_rate * (reward + discount_factor * max(Q(next_state, all_actions)) - Q(state, action))

By following this update rule, Q-Learning iteratively improves its estimates of the optimal Q-values for state-action pairs, leading to the convergence towards the optimal policy.

Lastly a basic outline for the Q-learning algorithm can be described as following:

1. Initialize the Q-values for all state-action pairs arbitrarily or to some predefined values.
2. Select an action ***a*** from the current state ***s*** based on the current Q-values using an exploration-exploitation strategy (e.g. epsilon-greedy).
3. Execute the selected action ***a*** and observe the reward r and the resulting next state s'.
4. Update the Q-value for the current state-action pair using the observed reward, discount factor, and the maximum Q-value in the next state:

- Q(state, action) = Q(state, action) + learning_rate * (reward + discount_factor * max(Q(next_state, all_actions)) - Q(state, action))

5. Update the current state to be the next state.
6. Repeat steps 2-5 until the learning process converges or a certain number of iterations is reached.

Double Q-learning

Double Q-learning is an extension of Q-learning that addresses the problem of overestimation bias. While similar to Q-learning, Double Q-learning introduces a crucial difference by maintaining two sets of Q-values, which separate the action selection from the value estimation process.

In Double Q-learning, the Q-values are updated iteratively, just like in Q-learning. However, the key distinction lies in how it utilizes the two sets of Q-values. One set of Q-values is used for selecting the action, while the other set is employed for estimating the reward.

By decoupling the action selection and reward estimation, Double Q-learning mitigates the issue of overestimation bias that can occur in traditional Q-learning. This bias arises when the maximum Q-value is consistently overestimated due to the same set of Q-values being used for both action selection and reward estimation.

Below is a basic outline for the double Q-learning algorithm:

1. Select an action ***a*** from the current state based on the combined set of Q-values Q1 + Q2
2. Execute the selected action ***a*** and observe the reward ***r*** and the resulting next state ***s'***.
3. With a respective 50% probability, update one of the Q-values for the current state-action pair using the observed reward, discount factor, and the estimated maximum Q-value:
    a. For Q1: Q1(state, action) = Q1(state, action) + learning_rate * (reward + discount_factor * Q2(s', argmax(Q1(s’,a))) - Q1(state, action))
    b. For Q2: Q2(state, action) = Q2(state, action) + learning_rate * (reward + discount_factor * Q1(s', argmax(Q2(s’,a))) - Q2(state, action))
4. Update the current state to be the next state.
5. Repeat steps 2-4 until the learning process converges or a certain number of iterations is reached.

SARSA

Sarsa is an on-policy reinforcement learning algorithm that aims to optimize the Q-values. It differs from Q-learning in several key aspects. Firstly, Sarsa is an on-policy learning algorithm, meaning that it learns and improves its policy while interacting with the environment.

The main distinction between Sarsa and Q-learning lies in how they estimate the value function. In Sarsa, rather than assigning the maximum value as in Q-learning, the value function is updated based on the average observed actions taken.
Q(state, action) = Q(state, action) + learning_rate * (reward + discount_factor * Q(next_state, next_action) - Q(state, action))

SARSA differs from Q-Learning in its approach to selecting the bootstrapped action at the next state. Q-Learning takes the maximum reward of the next state, regardless of the action it would actually choose. On the other hand, SARSA calculates the average reward based on its epsilon-greedy policy for selecting the next action.

In Figure 1, we can observe a grid world scenario. In this scenario, Q-Learning would choose the blue path, disregarding the risk of randomly walking into the red state. Conversely, SARSA might opt for the pink path to avoid the potential of reaching the red state.

The key distinction lies in their strategies for handling the exploration-exploitation trade-off. Q-Learning prioritizes the maximization of expected rewards, even if it means taking riskier paths, whereas SARSA considers the average rewards and tends to be more cautious in avoiding unfavorable outcomes.

These differences in action selection strategies make Q-Learning and SARSA suitable for different scenarios, depending on the desired balance between risk-taking and risk-aversion.

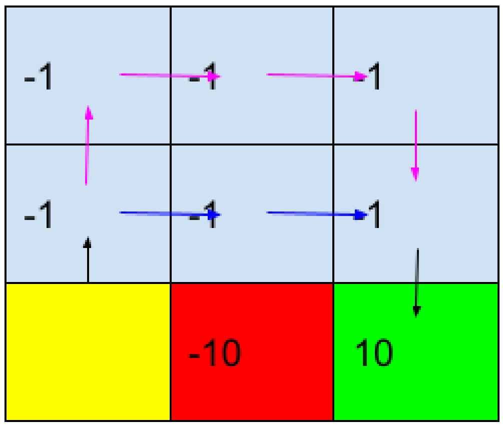


Figure 1: Grid world example. The yellow state represents the starting point, the green state is the goal, and the red state is a bad state that terminates the run.

A basic outline for the Sarsa algorithm can be described as the following:

1. Initialize the Q-values for all state-action pairs arbitrarily or to some predefined values.
2. Select an action a from the current state based on the current Q-values using an exploration-exploitation strategy (e.g. epsilon-greedy).
3. Execute the selected action a and observe the reward **r** and the resulting next state s'.
4. Select the next action a' from the next state s' based on the current Q-values using the exploration-exploitation strategy (commonly epsilon-greedy).
5. Update the Q-value for the current state-action pair using the observed reward, discount factor, and the maximum Q-value in the next state:
    - Q(state, action) = Q(state, action) + learning_rate * (reward + discount_factor *Q(next_state, next_action) - Q(state, action))
6. Update the current state to be the next state and the current action to be the next action..
7. Repeat steps 2-6 until the learning process converges or a certain number of iterations is reached.

Expected SARSA

In Expected Sarsa, unlike Sarsa and Q-learning, all possible actions of the next state are considered, taking into account the probability of each action being chosen according to the current policy.

Although Expected Sarsa introduces increased computational complexity compared to Sarsa, it addresses the issue of variance arising from the selection of the action in the next state. By taking the expected value over all possible actions, it reduces the variance and provides a more stable update to the Q-values.

The general update rule for Expected sarsa can be described as following:

Q(state, action) = Q(state, action) + learning_rate * (reward + discount_factor * expected_Q(next_state, all_actions) - Q(state, action))

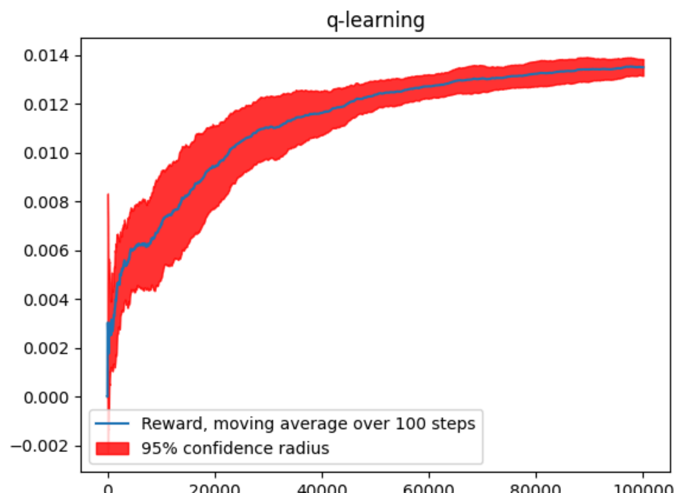
A basic outline for the Expected sarsa algorithm can be described as the following:

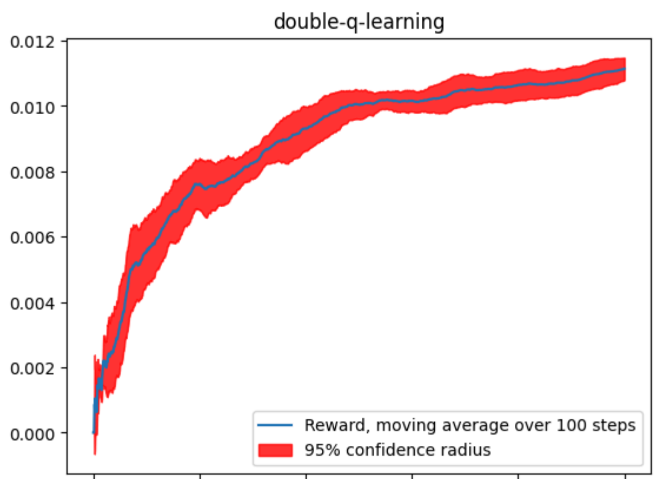
1. Initialize the Q-values for all state-action pairs arbitrarily or to some predefined values.
2. Select an action ***a*** from the current state based on the current Q-values using an exploration-exploitation strategy (e.g. epsilon-greedy).
3. Execute the selected action a and observe the reward r and the resulting next state ***s'***.
4. Estimate the expected value of the Q-value for the next state-action pairs using the current Q-values and the probabilities of selecting each action based on the current policy.
5. Update the Q-value for the current state-action pair using the observed reward, discount factor, and the estimated expected Q-value for the next state-action pairs:
    - Q(state, action) = Q(state, action) + learning_rate * (reward + discount_factor * expected_Q(next_state, all_actions) - Q(state, action))
6. Update the current state to be the next state.
7. Repeat steps 2-6 until the learning process converges or a certain number of iterations is reached.

Run and plot rewards

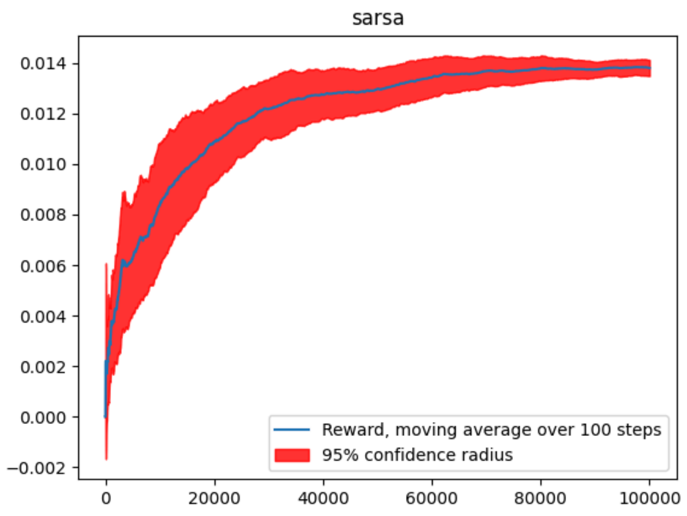
All experiments shown here were run with zero initialization on FrozenLake-v1 4x4, averaged over 5 runs. We also ran RiverSwim, however we don't show these results here as the graphs look similar but with shorter convergence times (as the game is simpler) - this goes for the rest of the report also. The x-axis represents steps, and y-axis reward/step. We chose this metric to track as it reveals quicker policies. Tracking reward/episode is also an option, which would be better to visualize how often the policy reaches the goal.

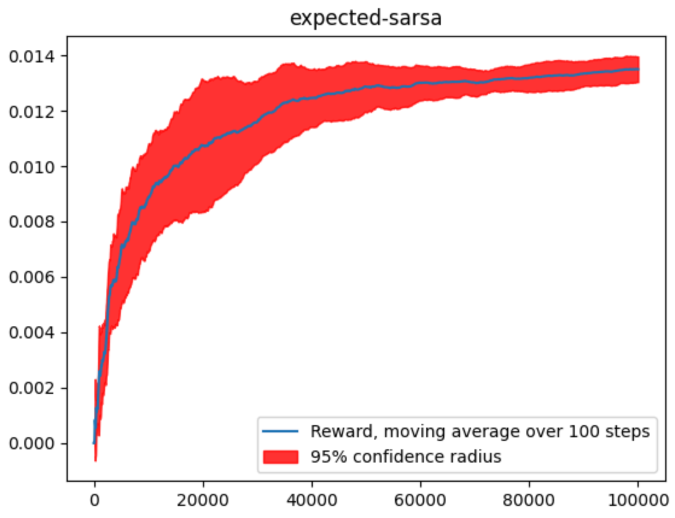
Q-learning & Double-Q-Learning:





Sarsa & Expected Sarsa:

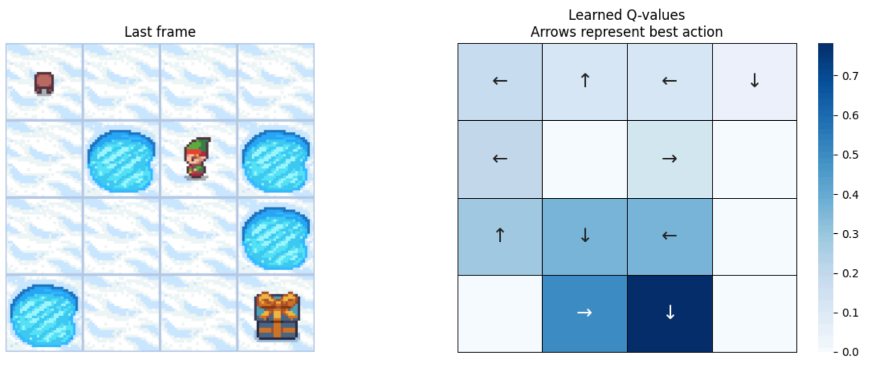




Visualize Q-values and greedy policy

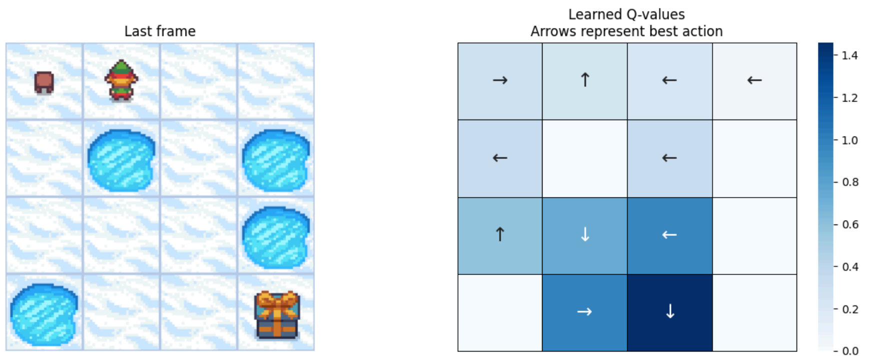
All experiments shown here were run with zero initialization on FrozenLake-v1 4x4, averaged over 5 runs. The image of the last frame is there to show what the map looks like to contextualize the learned greedy policy. The results of RiverSwim can be accessed in the "Results" folder in the GitHub repository.

Q-learning

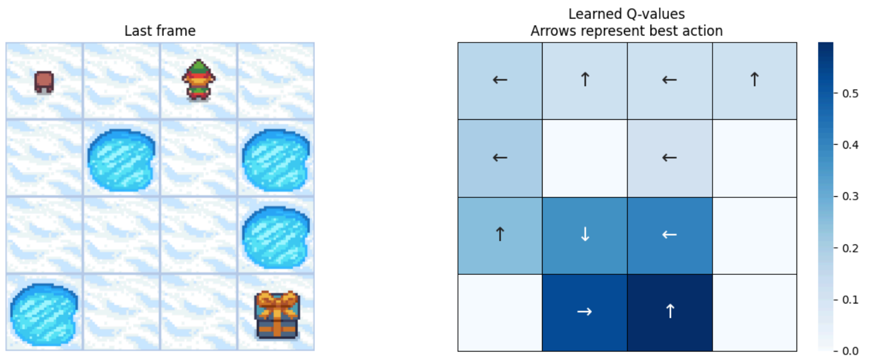


Double Q-learning

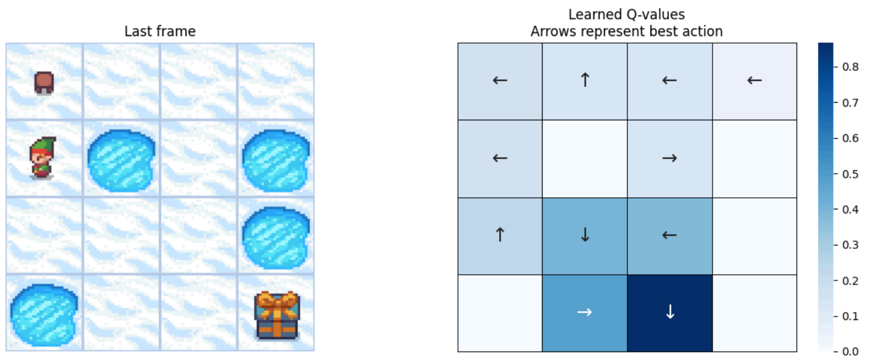
It should be noted that the Q-values displayed are the sum of both the Q-tables, this was chosen since the sum is the basis for action selection.



Sarsa



Expected sarsa



Determining the optimal policy for the slippery environment is challenging because the agent only executes its intended action one-third of the time and otherwize moves in either perpendicular direction two thirds of the time. For instance, at frame 6 (between the two lakes), the optimal move is to either move left or right, as there then only is a one-third chance of falling into the lake and a two-thirds chance of slipping either up or down, thus avoiding the lake.

However, the Expected-Sarsa, Q-Learning and double Q-learning algorithms exhibit a sub-optimal policy in state 3 (top-right corner), where the ideal action is to move upwards. One possible explanation is the infrequent exploration of state 3, indicated by color in the table. This scarcity of exploration arises because state 2 consistently favors moving left as the best action.

Furthermore, it is worth noting that the optimal policy favors the left path over the right path. This preference arises due to state 6 on the right path, where there is a one-third chance of falling into either of the lakes with the optimal policy. Conversely, such a risky state does not exist on the left path. Consequently, the optimal policy dictates attempting a leftward move in state 0 (top left corner), a behavior consistently exhibited by all the algorithms. This is caught by all algorithms except double Q-learning.

State 14, which is the state before the goal, poses an interesting challenge when determining the optimal policy. To analyze the best action in this state, it is helpful to consider the quality of the surrounding states. The ultimate goal is to reach the goal state, which guarantees a victory and has the highest value. The current state, being closest to the goal and without immediate dangers, is the second-best option.

However, deciding on the optimal action beyond these two states becomes more intricate. State 13 (moving left) is considered the third-best option because choosing state 10 (moving up) runs the risk of encountering the previously discussed hazardous state 6. Therefore, the optimal action in state 14 is to move down. This choice leads to the optimal state one-third of the time, the second most optimal state one-third of the time, and the third most optimal state one-third of the time.

This intricate detail is captured by all algorithms except for Sarsa, which suggests the sub-optimal action of moving up. However, interestingly, in the case of Sarsa, state 13 is explored relatively more compared to most other algorithms. This higher exploration rate indicates that the selected action in state 13 does not align with its preferred action during a significant portion of the training process.

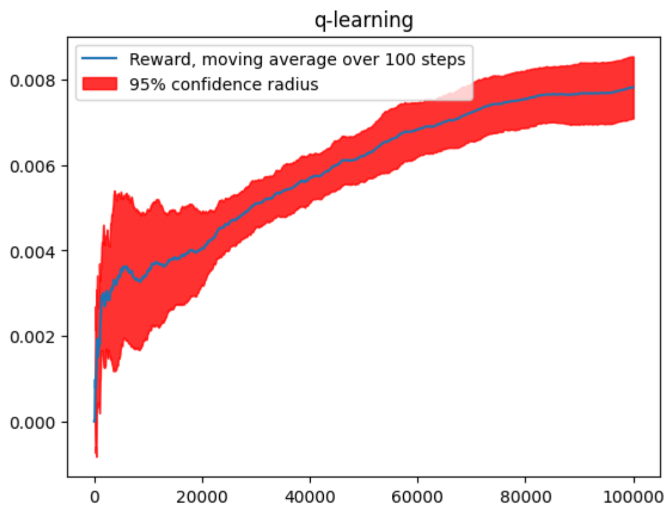
Initialization of Q-values

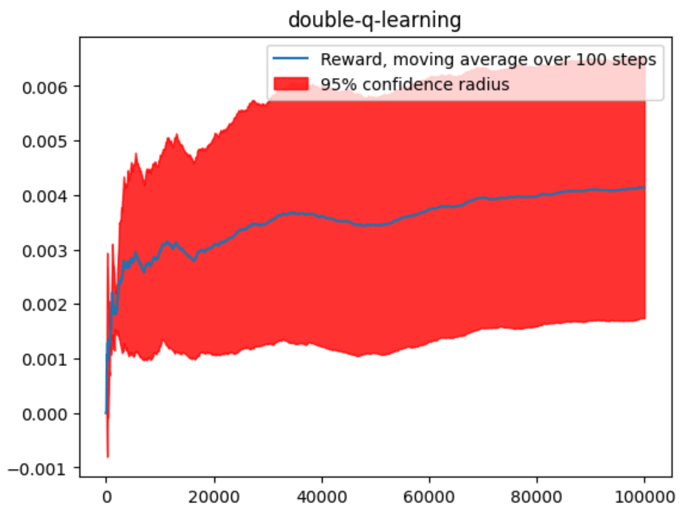
For RiverSwim and FrozenLake, we experimented with zero initialization and random initialization. For FrozenLake we also ran a heuristic initialization, which biased the initial Q-table to move down and to the right (where the goal is relative to the starting position).

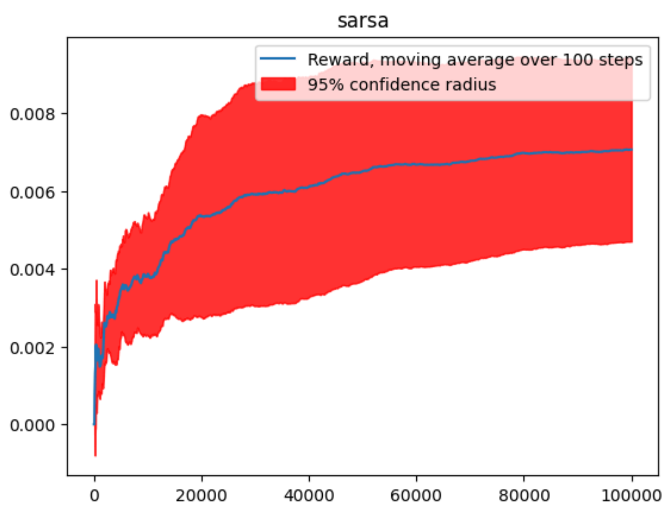
Zero

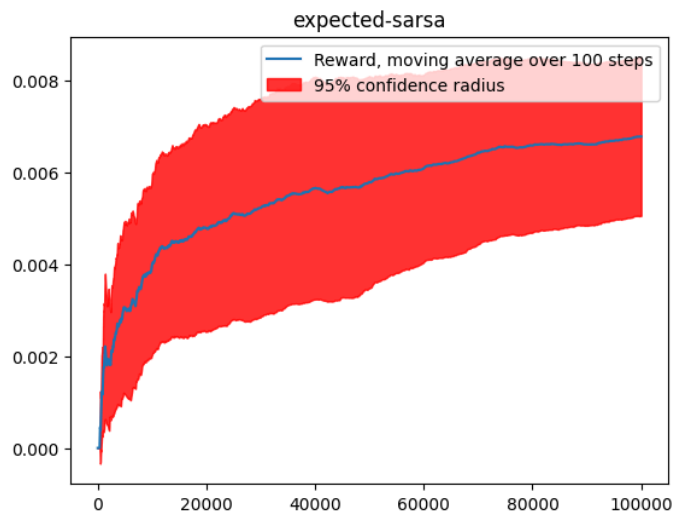
Results can be seen in “Run and plot rewards” above.

Random





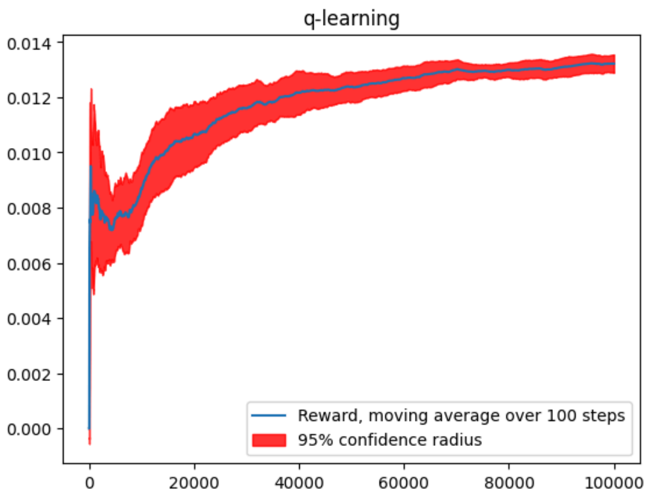


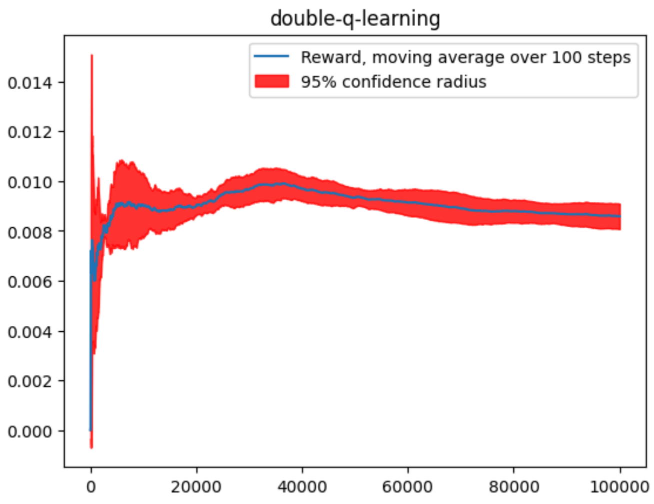


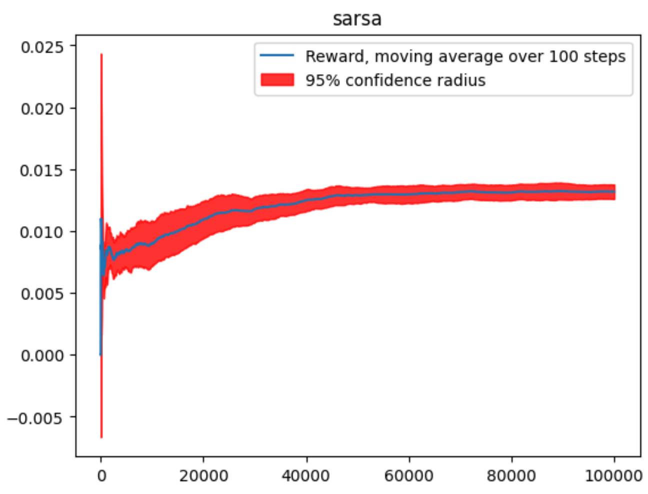
Random initialization hindered every learner in its convergence-time, and even ability to learn at all for sarsa, expected-sarsa and especially double-Q-learning. For every learner, the variance between runs increased drastically, which we would expect as some Q-priors may bias the learned towards a more or less performant solution.

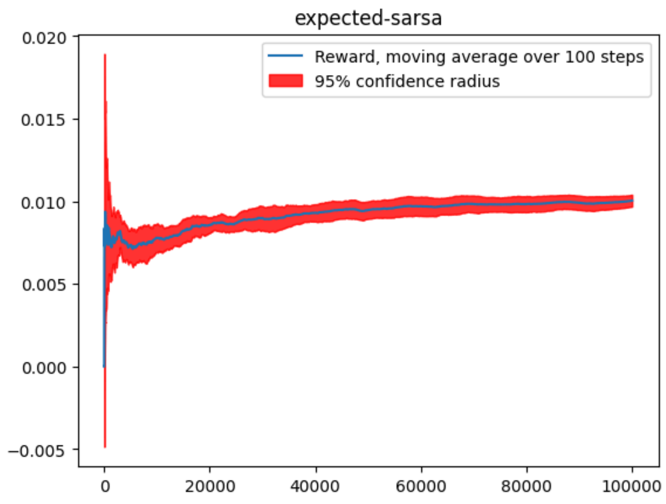
Heuristic

Our heuristic, initialized with a bias down and to the right, proved effective to give an initial boost in performance, but overall the convergence time did not decrease in the slippery environment. Exception to this is double-Q-learning, which decreased its final policy convergence from ~0.011→0.009 relative to zero initialization. Also, expected-sarsa seems to converge to a worse policy than with zero initialization (0.013→0.010) - probably owing to a decrease in exploration. In the non-slip environment a slight (~1-2x) increase in convergence time for all learners is noted (not shown here, but in github repo ./Results/).









Optimistic:

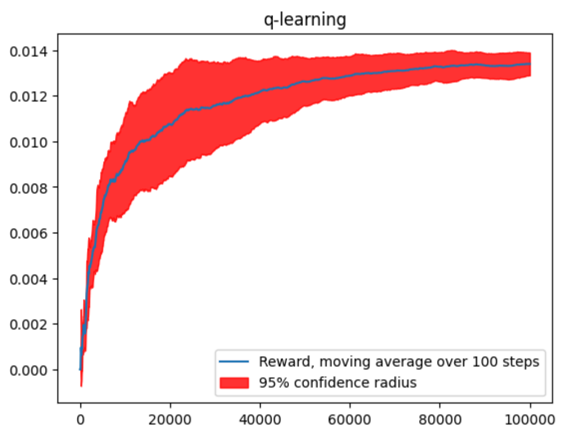
Optimistic initialization is a technique used in reinforcement learning to stimulate exploration by assigning initially high Q-values to all actions. This positive bias encourages the agent to consider actions that may not be optimal according to its current knowledge but could potentially lead to higher rewards.

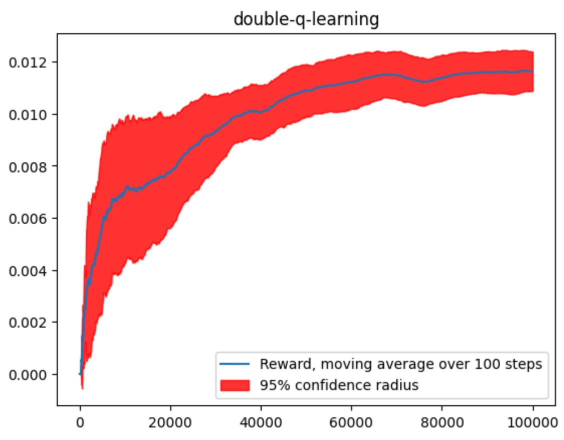
By providing optimistic Q-values, the agent is motivated to explore different actions and states in the early stages of learning. This is because the initially high Q-values (set to 0.5 in our case. Though we also tested 1, which provided no change) make these actions appear more promising in terms of potential rewards. The aim is to ensure that the agent does not prematurely converge to a suboptimal policy but continues to explore the environment thoroughly.

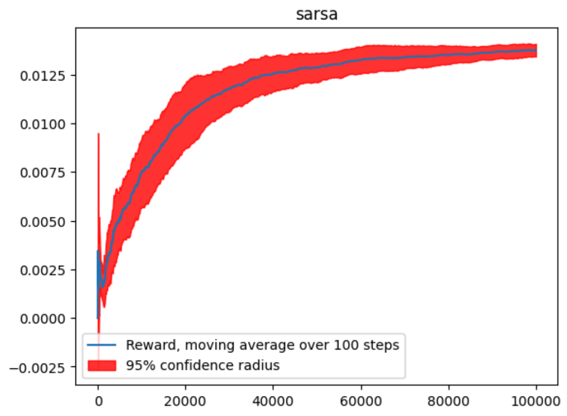
However, as the agent gains more experience and improves its policy, it becomes more confident in its actions. This increased confidence can lead to a gradual reduction in the exploration rate as the agent starts to rely more on its learned policy and exploit its current knowledge to maximize rewards.

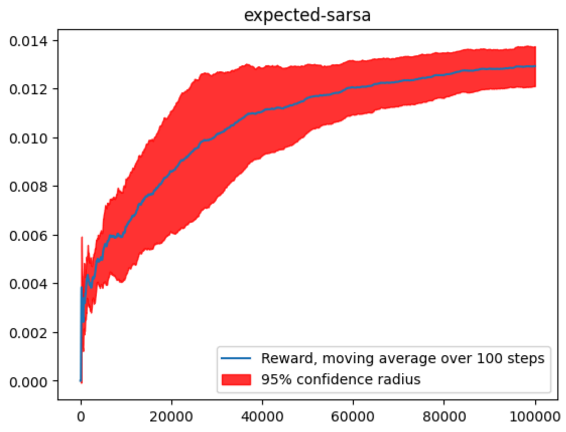
Interestingly, in our testing, we found that all algorithms, except for Q-learning, exhibited slower convergence and achieved lower total rewards when using optimistic initialization compared to the baseline zero initialization. This suggests that the optimistic initialization approach did not provide significant benefits in terms of performance for these algorithms.

It's worth noting that Q-learning, in particular, appeared similar to the baseline case, indicating that it may be less affected by the optimistic initialization technique. The reasons behind the suboptimal performance of the other algorithms with optimistic initialization would require further investigation.









Optimistic initialization with all state-action pairs initialized to 0.5, though we also tested 1, which provided no change.

Discussion:

What are your conclusions? How do the agents perform? Discuss if the behavior is expected based on what you have learned in the course.

In general, Q-learning seems to converge most robustly, with sarsa being close but with higher variance on specifically random initialization.

With zero initialization, it is notable that double Q-learning converged slower than the other algorithms. It did however also have the tightest confidence radius. This behavior is not surprising given the design of double Q-learning in that it attempts to be more careful and avoid over-estimation.

In the case of random initialization, Q-learning had by far the tightest confidence radius, this is not surprising. Double Q-learning is by its nature more reluctant to make drastic changes to the Q-values and so when the initial ones are random it will take longer for them to tend to their true values. The behavior also makes sense for the Sarsa-algorithms, since they rely on their previous actions, and those actions are based upon initially random Q-values.

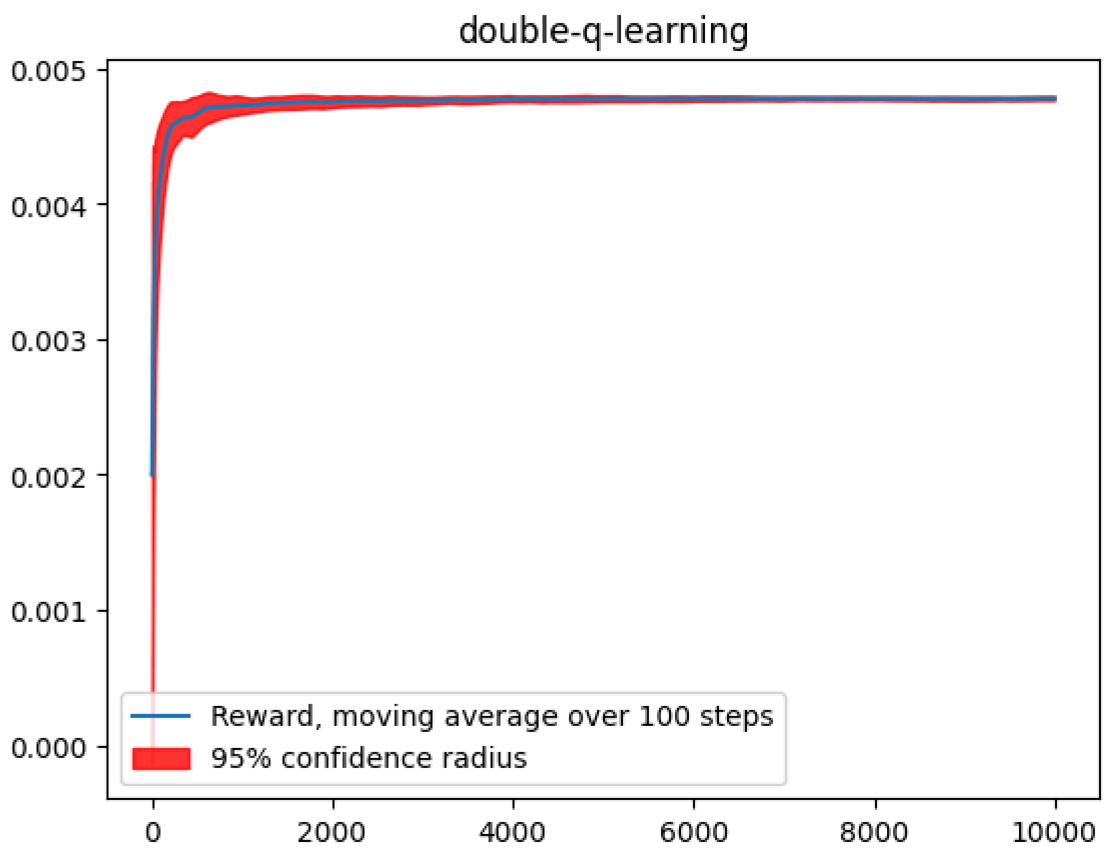
When the environment was optimistically initialized, there was a stark difference in the confidence radius of SARSA and Expected SARSA. It is possible that the most important adjustment to be made for the optimistic Q-values is to reduce the values of state-action-pairs which originate from or might take the agent into a lake. Since Expected SARSA considers the actions available from the following states, it might be more easily tempted into a lake since it perceives there to be good value in the actions leaving the lake.

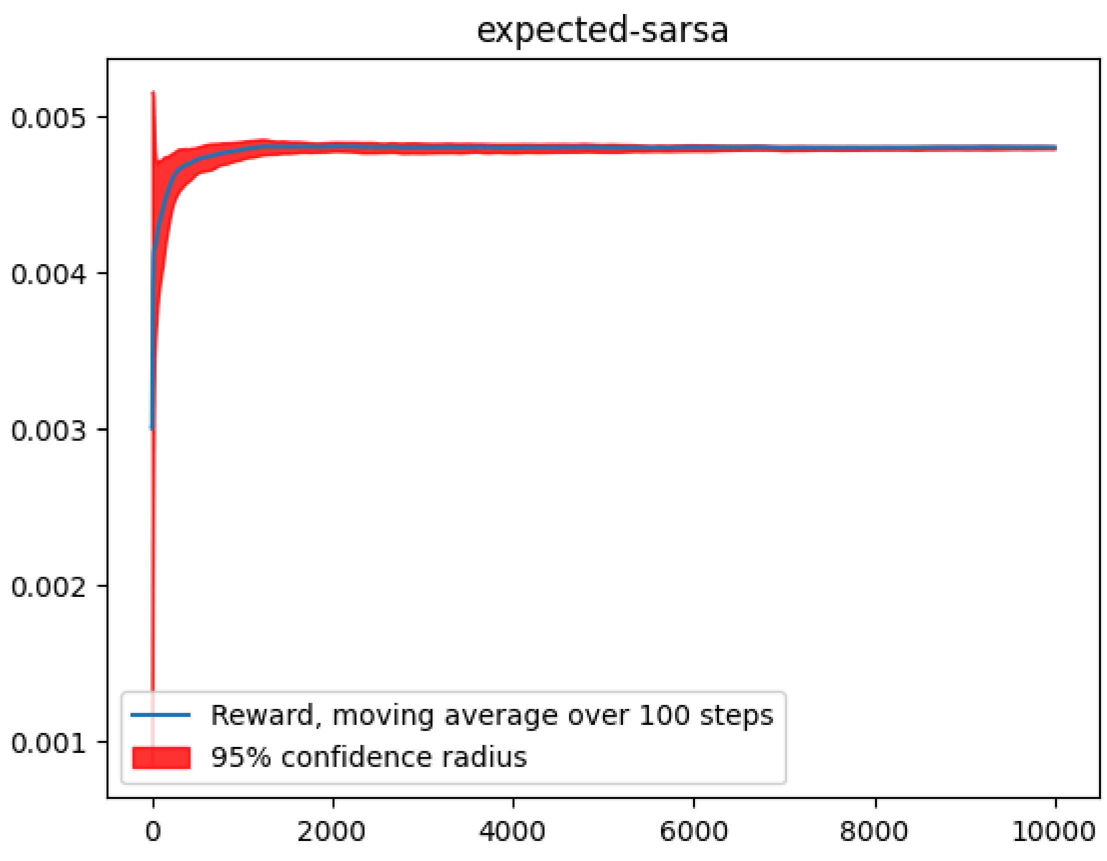
Discussion about the heuristic initialization can be found in section Initialization of Q-values#Heuristic.

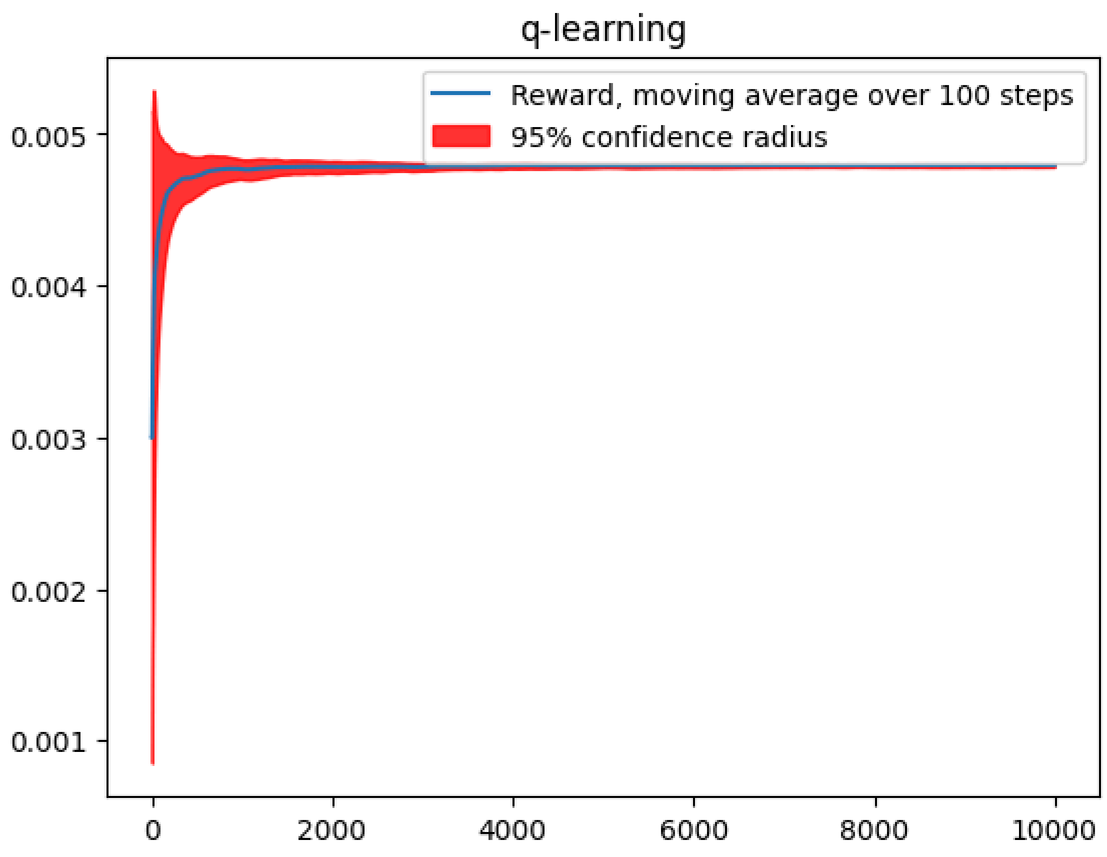
Appendix

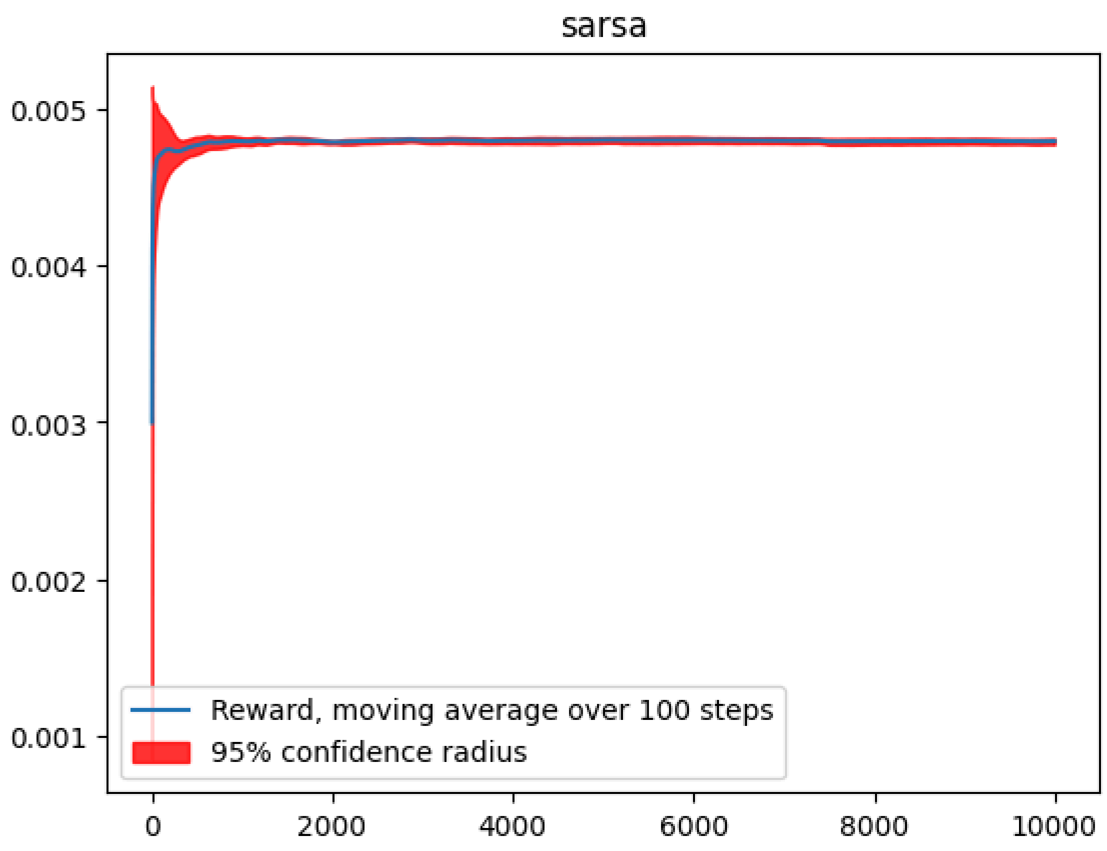
Riverswim

Zero initialization

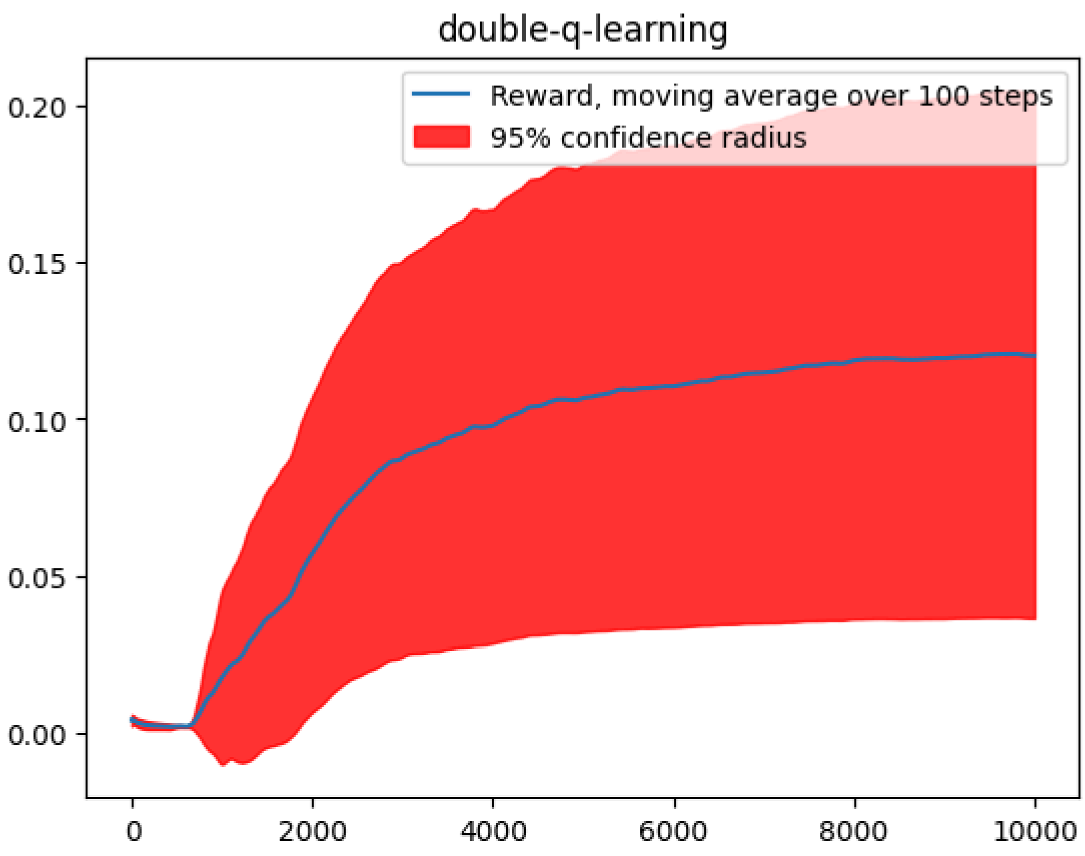

double-q-learning
0.005
0.004
0.003
0.002
0.001
0.000
0
2000
4000
6000
8000
10000
Reward, moving average over 100 steps
95% confidence radius

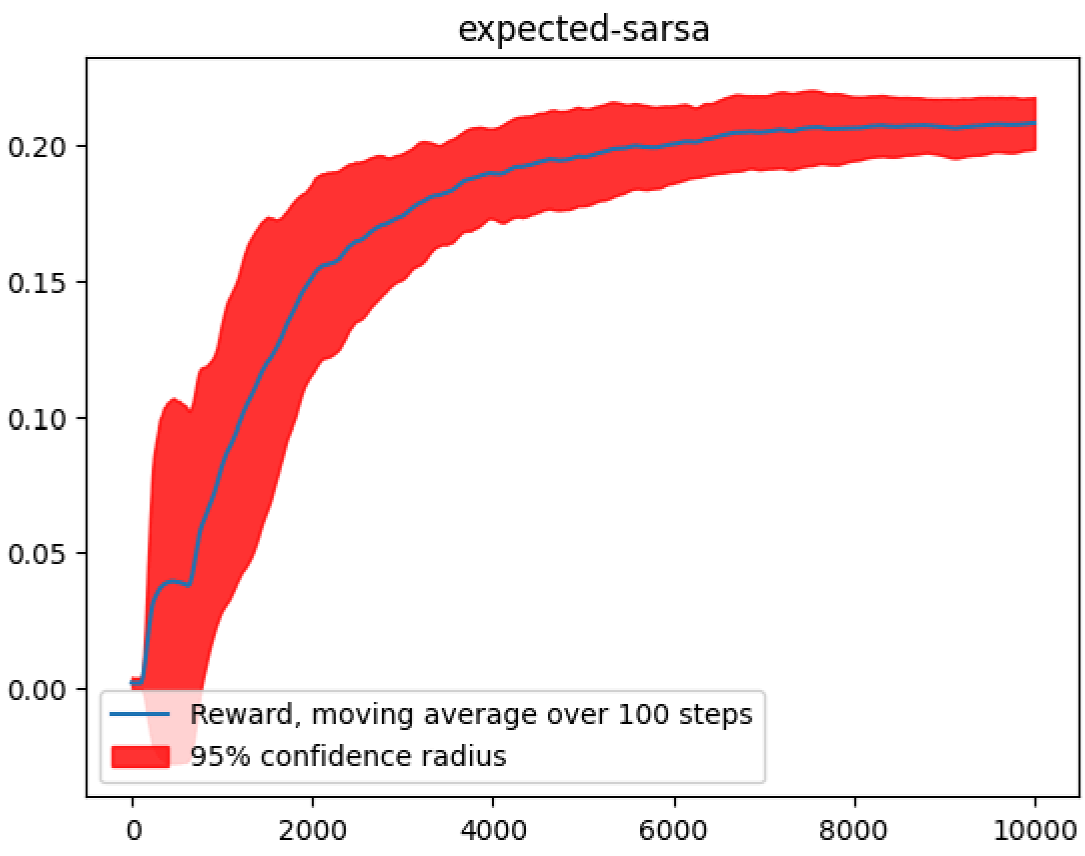

expected-sarsa
0.005
0.004
0.003
0.002
0.001
0
2000
4000
6000
8000
10000
Reward, moving average over 100 steps
95% confidence radius

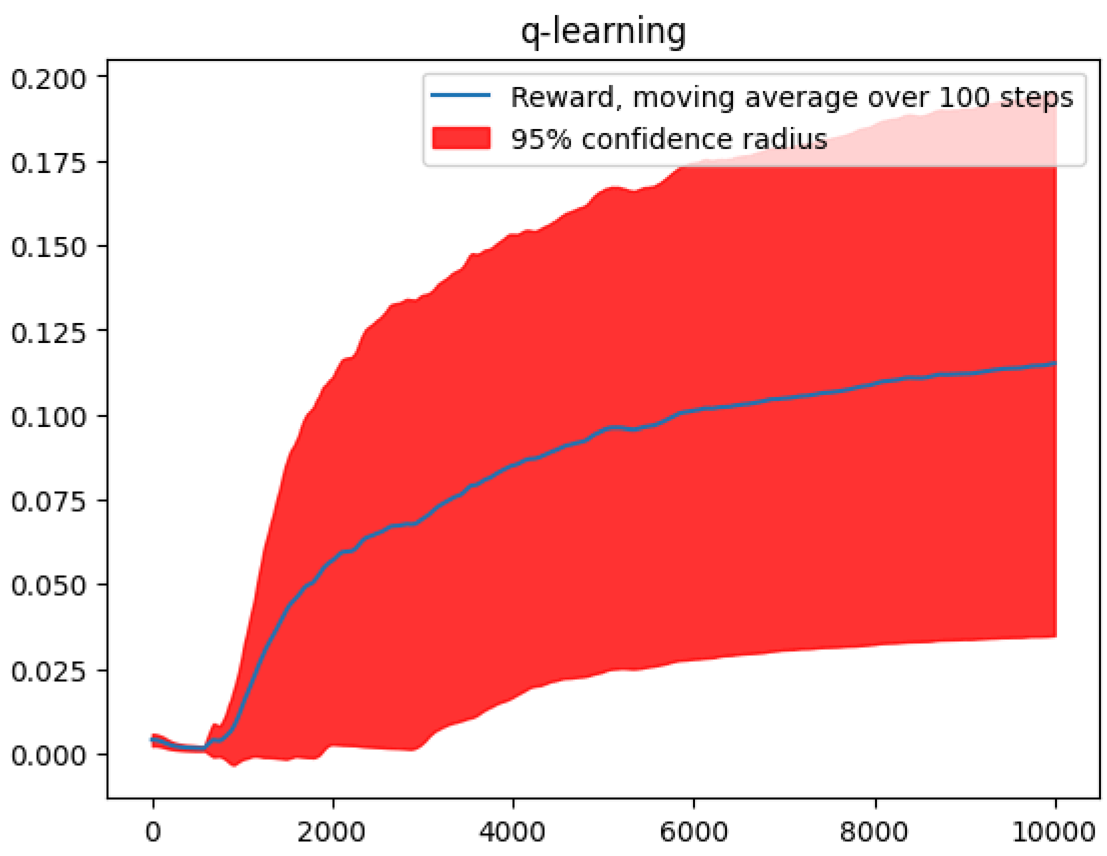

q-learning
Reward, moving average over 100 steps
95% confidence radius
0.005
0.004
0.003
0.002
0.001
0
2000
4000
6000
8000
10000

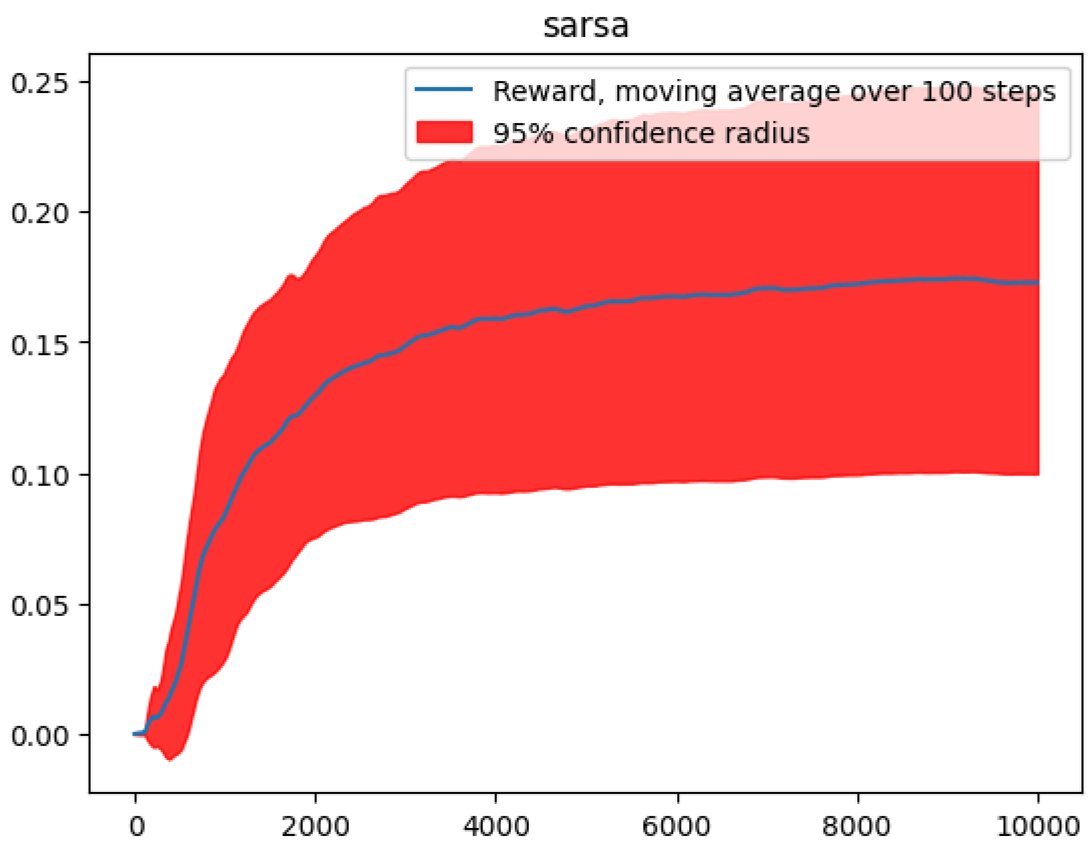

sarsa
Reward, moving average over 100 steps
95% confidence radius
0.005
0.004
0.003
0.002
0.001
0
2000
4000
6000
8000
10000

Random

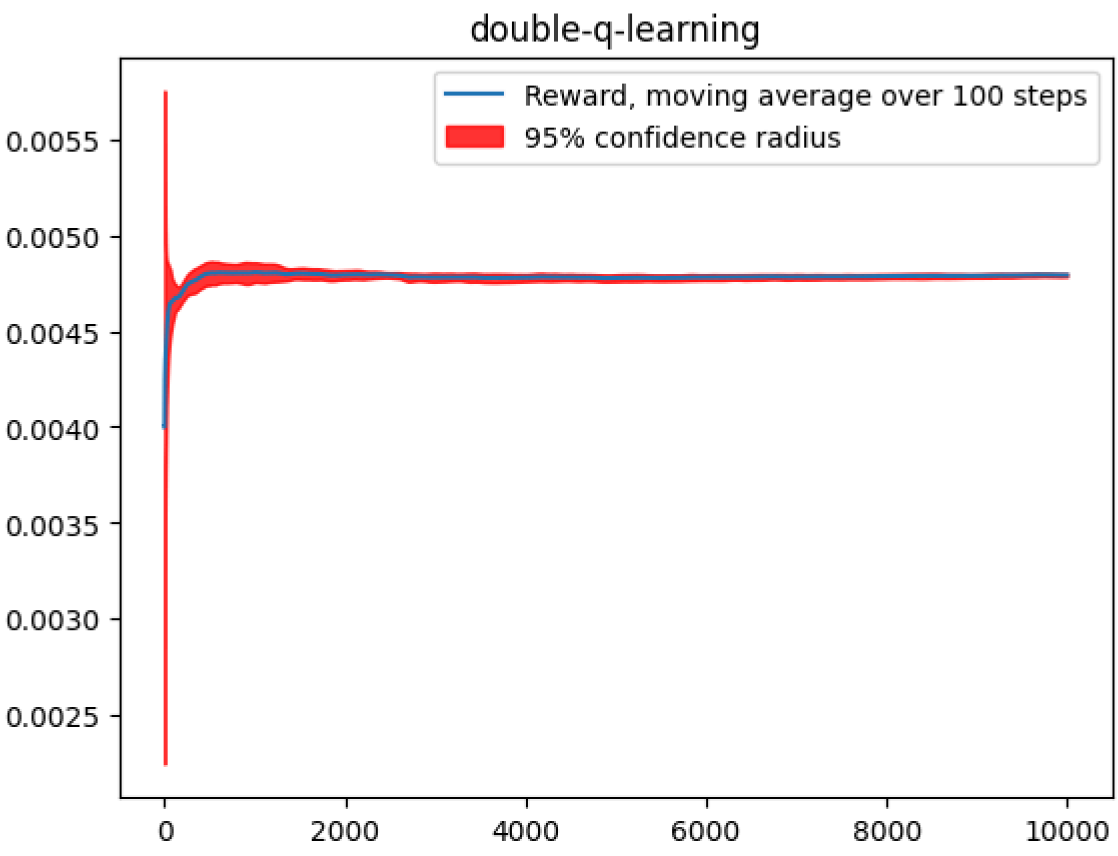

double-q-learning
Reward, moving average over 100 steps
95% confidence radius
0.20
0.15
0.10
0.05
0.00
0
2000
4000
6000
8000
10000

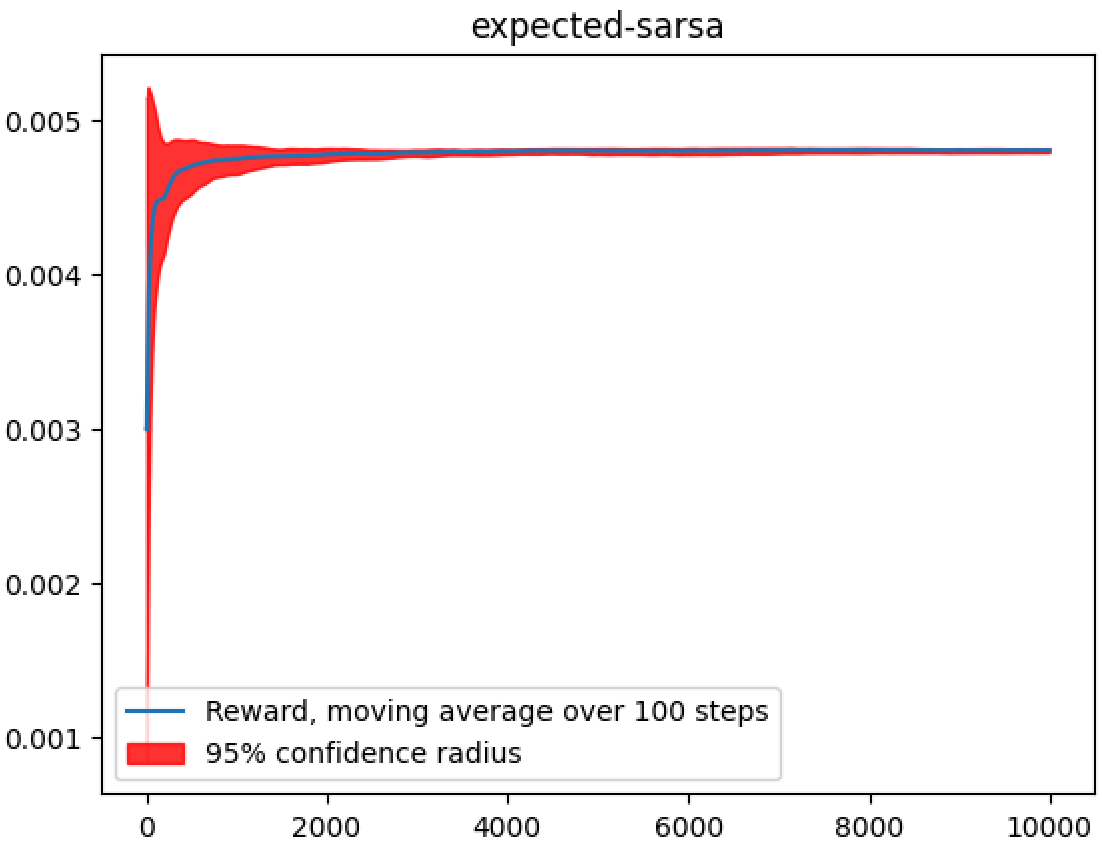

expected-sarsa
Reward, moving average over 100 steps
95% confidence radius
0.20
0.15
0.10
0.05
0.00
0
2000
4000
6000
8000
10000

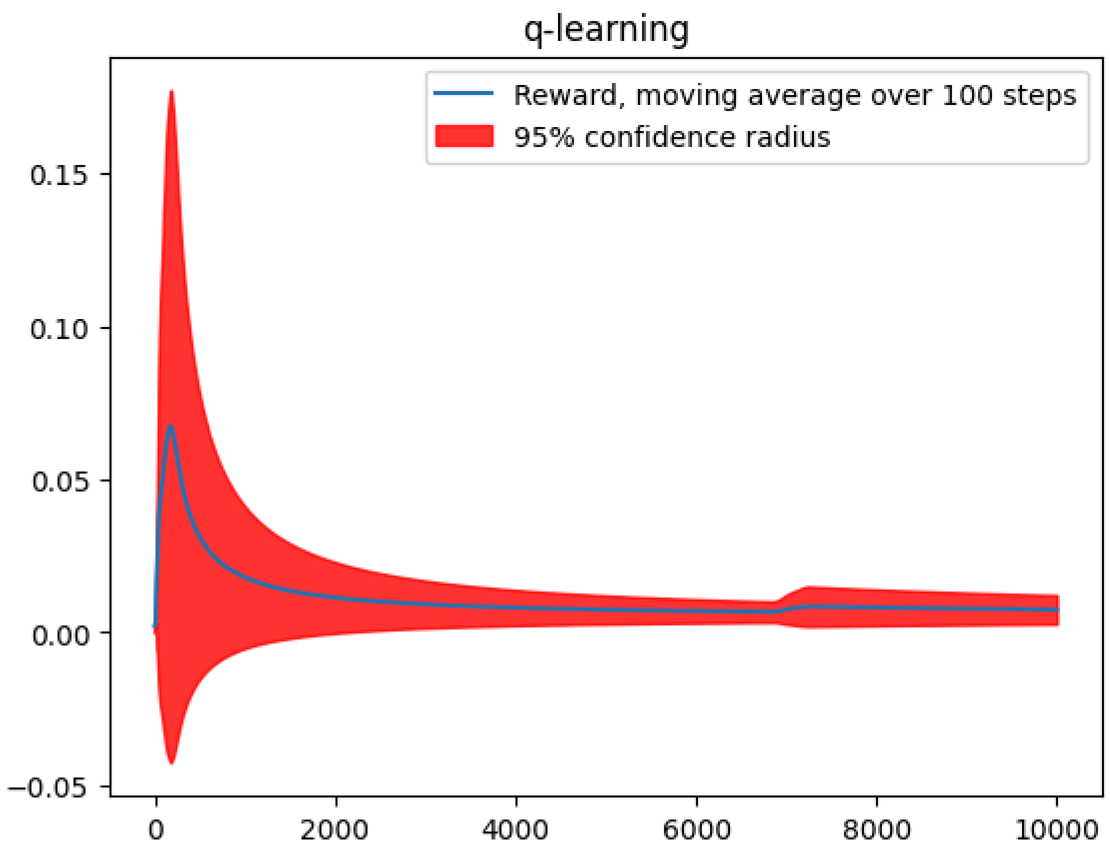

q-learning
Reward, moving average over 100 steps
95% confidence radius
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
2000
4000
6000
8000
10000

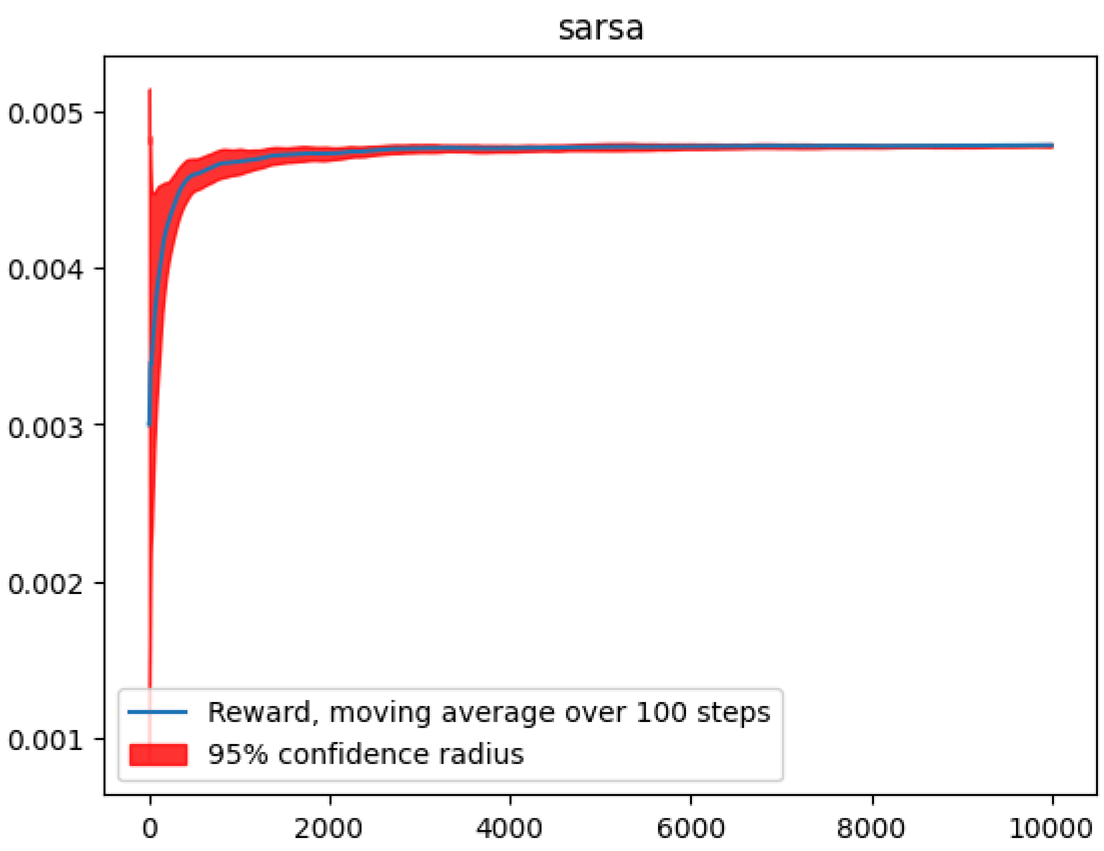

sarsa
Reward, moving average over 100 steps
95% confidence radius
0.25
0.20
0.15
0.10
0.05
0.00
0
2000
4000
6000
8000
10000

Optimistic


double-q-learning
Reward, moving average over 100 steps
95% confidence radius
0.0055
0.0050
0.0045
0.0040
0.0035
0.0030
0.0025
0
2000
4000
6000
8000
10000


expected-sarsa
Reward, moving average over 100 steps
95% confidence radius
0.005
0.004
0.003
0.002
0.001
0
2000
4000
6000
8000
10000


q-learning
Reward, moving average over 100 steps
95% confidence radius
0.15
0.10
0.05
0.00
−0.05
0
2000
4000
6000
8000
10000


sarsa
Reward, moving average over 100 steps
95% confidence radius
0.005
0.004
0.003
0.002
0.001
0
2000
4000
6000
8000
10000