Mid Project: Tabular Reinforcement Learning in MiniGrid

Monte Carlo, SARSA, and Q-Learning on Two Sparse-Reward Missions

Author 1: Avital Fine (ID: 208253823)
Author 2: Daniel Katz (ID: 315114991)
Course: RL2026A   Semester: 2025

January 10, 2026

Abstract

This report studies tabular reinforcement learning in two MiniGrid tasks with sparse terminal rewards. We compare First-Visit Monte Carlo, SARSA(0), and Q-Learningunder the same exploration scheme (epsilon-greedy with decay), and evaluate convergence, sample efficiency, and policy quality. We report training curves (reward/success rate vs. episodes), and inference curves (average steps to solve). Video rollouts during training and after convergence are included in the submitted notebook.

1 Environments and MDP Analysis

We evaluate two grid worlds (size 10×10 with outer walls) implemented in the notebook:

Environment 1: `RandomEmptyEnv_10`**.** This environment is an empty room, and the goal of the agent is to reach the green goal square. Agent position at beginning is random. Direction of the agent at beginning is random. Goal position could be: $(8, 1)$ or $(1, 8)$ or $(8, 8)$. The episode ends upon reaching the goal or when the step limit is reached. **Reward:** The agent receives $+1$ upon successfully reaching the goal (termination). No other rewards are defined.

Environment 2: `RandomKeyMEnv_10`**.** A two-room layout separated by a vertical wall with a locked door. The key is placed on the left side; the agent must pick it up, open the door, and reach a goal on the right side. The goal location is sampled from a small set of fixed coordinates. The episode ends upon reaching the goal or step limit. **Reward:** We use a shaped reward function: $+1$ upon reaching the goal, $+0.5$ for picking up the key (once), $+0.5$ for opening the door (once), and a step penalty of -0.01.

1.1 Environment Type and Observability

Episodic: Yes. Each run is a episodic task that terminates when the goal is reached (`terminated`) or when the maximum number of steps is exceeded (`truncated`).

MDP: Yes. MiniGrid is Markovian: the next state and reward depend only on the current environment state (agent position/direction, grid contents such as walls/door/key/goal, and relevant internal flags). With our `KeyFlatObsWrapper`, the agent observes a global encoding of the entire grid (plus the agent's position and direction), so the task can be treated as an MDP from the agent's point of view.

Action space: Discrete. MiniGrid defines 7 actions: turn-left, turn-right, move-forward, pick-up, drop, toggle, and done. In our implementation we restrict actions only in `RandomKeyMEnv_10` by excluding `avoid_actions={drop, done}`, leaving 5 effective actions. For `RandomEmptyEnv_10` we keep the default action set (some actions are simply no-ops in this environment).

State space: Discrete and finite (but large). We learn a tabular $Q(s, a)$. In practice, the state s is the *flattened observation vector* returned by `env.reset() / env.step()`, after applying `KeyFlatObsWrapper` (Section 2).

Observability: Fully observable. With `KeyFlatObsWrapper`, the agent observes a global encoding of the entire grid (including agent direction and door state in the grid encoding), so we treat the problem as an MDP from the agent's point of view (no additional latent variables are required).

2 State Representation and Q-table Size

2.1 State definition used in the notebook

In the final notebook, the tabular state is taken directly from the environment observation returned by `env.reset()` and `env.step(action)`.

We wrap MiniGrid with `KeyFlatObsWrapper`, which encodes the full grid state as follows: the 10×10 grid is encoded using MiniGrid's 3-value cell representation (object, color, state), the agent cell is overwritten to include the agent's direction, the outer walls are cropped, and the remaining $8 \times 8 \times 3$ tensor is flattened into a vector of length $8 \cdot 8 \cdot 3 = 192$. This flattened vector is stored as a tuple and used as the key in the Q-table.

2.2 State space size estimation (reachable configurations)

Below we give simple *upper bounds* by multiplying together the main independent choices that can vary in our code.

`RandomEmptyEnv_10` **(RandomEmptyEnv_10).**

- Agent position: interior is $8 \times 8 \Rightarrow 64$ possibilities.
- Agent direction: 4 possibilities.
- Goal position: chosen from 3 fixed corners $\Rightarrow$ 3 possibilities.

So an upper bound is:

$$|\mathcal{S}_{\text{empty}}| \leq 64 \cdot 4 \cdot 3 = 768.$$

MiniGrid defines 7 actions. In `RandomEmptyEnv_10` only {left, right, forward} affect navigation, while {pick-up, drop, toggle, done} are effectively no-ops; however, since we do not filter them in this environment, the Q-table can still store values for all 7 actions. Thus, an upper bound on Q-entries is $768 \cdot 7 = 5376$.

`RandomKeyMEnv_10` **(RandomKeyMEnv_10).** Here the observation can additionally change due to *key location/carrying*, *door location*, *door state*, and *goal location*. Each number in the product below comes directly from our environment generation:

- **Agent position:** interior $8 \times 8 \Rightarrow 64$.
- **Agent direction:** 4.
- **Key location:** in our code the key is placed in the left room with $\mathtt{x} \in \{1, 2\}$ (2 columns) and $\mathtt{y} \in \{2, 3, 4, 5, 6, 7, 8\}$ (7 rows), so $2 \cdot 7 = 14$ possible key tiles. Additionally, after pickup the key disappears from the grid and is *carried* $\Rightarrow +1$ extra case. Total: $14 + 1 = 15$.

- **Door location:** the door row is sampled from $\{1, \ldots, 8\}$ (any interior row), while the door column is fixed at the partition wall $\Rightarrow$ 8 possibilities.
- **Goal location:** chosen from $(8, 1)$ or $(8, 8)$ $\Rightarrow$ 2 possibilities.
- **Door state:** the door can be in up to 3 discrete states in MiniGrid encoding (locked / closed / open) $\Rightarrow$ 3 possibilities.

Putting it together:

$$|\mathcal{S}_{\text{key}}| \lesssim (64) \cdot (4) \cdot (15) \cdot (8) \cdot (2) \cdot (3).$$

So:

$$|\mathcal{S}_{\text{key}}| \lesssim 184{,}320.$$

In practice, we exclude two actions, `drop` and `done`, which are never useful in our setups. Thus, the effective action set has $|\mathcal{A}| = 5$ actions. The resulting upper bound on Q-table entries is therefore:

$$|\mathcal{S}_{\text{key}}| \cdot |\mathcal{A}| \lesssim 184{,}320 \cdot 5 = 921{,}600.$$

Important note (why this is only an upper bound). Not every combination above is actually reachable (e.g., when the agent stands on the key tile, the wrapper overwrites that cell with the agent encoding), and our Q-table is stored *sparsely* (a dictionary): we only create (s, a) entries for states that are visited during training.

3 Algorithms

We compare three tabular reinforcement learning algorithms under the same ϵ-greedy exploration strategy. All methods operate on the discrete, fully observable state representation described in Section 2 and are evaluated on two MiniGrid missions with sparse terminal rewards.

3.1 First-Visit Monte Carlo

Type: On-policy, episodic Monte Carlo control.

Monte Carlo learning updates action values only after observing a complete episode. For each first visit of a state–action pair (s, a), the return G_t is computed and used to update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big(G_t - Q(s, a)\big).$$

3.2 SARSA(0)

Type: On-policy temporal-difference (TD) control.

SARSA performs incremental updates at every step using the next action selected by the current behavior policy:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma Q(s', a') - Q(s, a)\Big).$$

3.3 Q-Learning

Type: Off-policy temporal-difference (TD) control.

Q-learning updates toward the greedy value of the next state, independent of the behavior policy:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(r + \gamma \max_{a'} Q(s', a') - Q(s, a)\Big).$$

3.4 Hyperparameters and initializations

We tuned α, γ, and the ϵ-greedy decay schedule separately for each environment and algorithm, and we report only the best-performing configuration in the notebook outputs and figures.

- Learning rate: $\alpha \in \{0.1, 0.2\}$
- Discount factor: $\gamma \in \{0.9, 0.99\}$
- Epsilon decay: $\epsilon_{decay} \in \{0.999, 0.9995\}$
- Initial exploration: $\epsilon_0 = 1.0$, $\epsilon_{min} = 0.1$
- Initialization: zero initialization vs. optimistic initialization ($Q_0 > 0$)

After evaluating all combinations, we selected the parameters that yielded the best average reward and success rate. It is important to note that the accompanying notebook includes only the final, best-performing hyperparameter configurations. This was done to maintain clarity and focus on the most successful results, avoiding the clutter of hundreds of suboptimal experimental runs. The best hyperparameters per environment and algorithm are summarized in Table 3.

4 Results

We report results separately for each environment. All plots correspond to the best hyperparameter configuration found for each algorithm.

4.1 `RandomEmptyEnv_10` (RandomEmptyEnv10)

4.1.1 Hyperparameters and Training

Extensive grid search revealed the following optimal hyperparameters: $\gamma = 0.99$, $\alpha = 0.2$, $\epsilon_0 = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.995$.

Reward Structure: The agent receives a reward of +1 upon termination (reaching the goal).

Figure 1 compares the training performance.

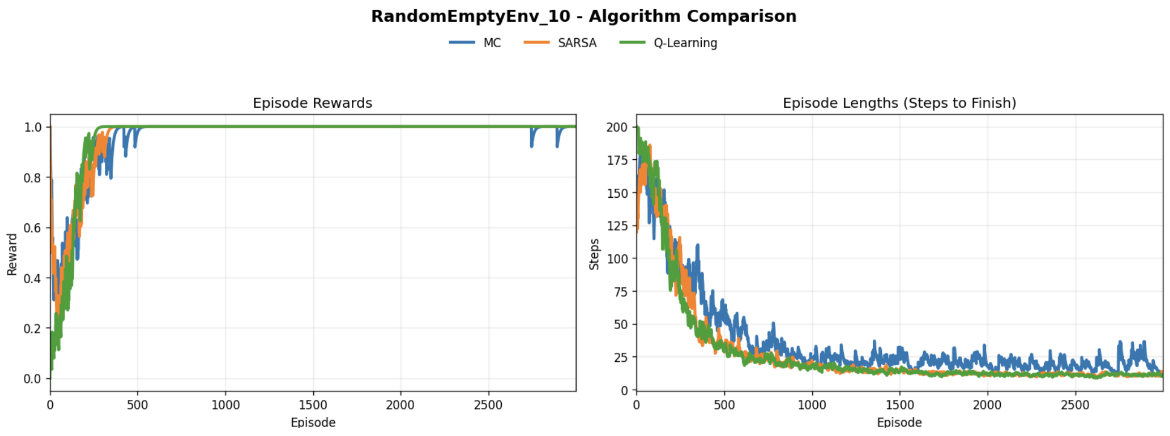


Figure 1: `RandomEmptyEnv_10`: Training reward comparison. All algorithms successfully learn the task.

4.1.2 Inference Evaluation

The trained policies were evaluated over 100 episodes. The results are summarized below:

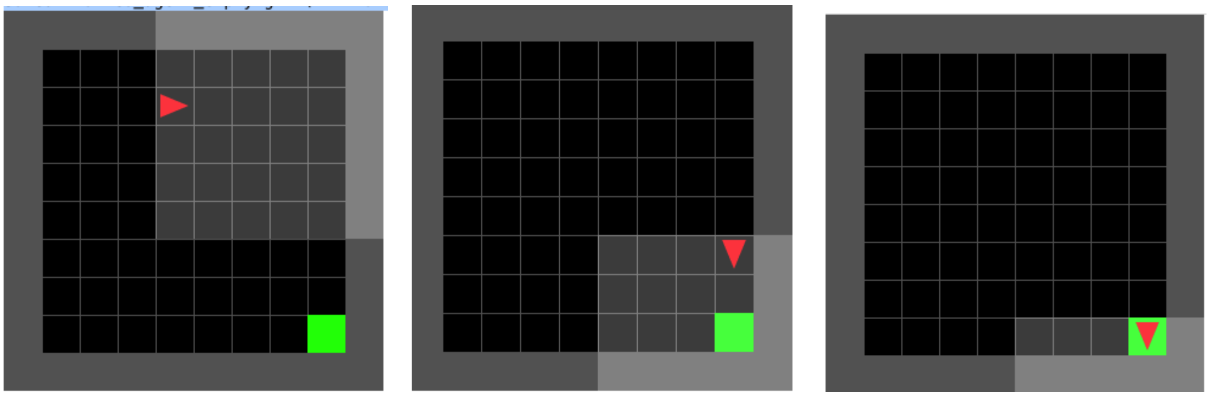
Algorithm	Avg Reward	Avg Steps	Success Rate
Monte Carlo	0.860	36.8	86.0%
SARSA	1.000	9.9	100.0%
Q-Learning	1.000	9.2	100.0%

Table 1: Performance metrics on `RandomEmptyEnv_10` (100 evaluation episodes).

Both SARSA and Q-Learning achieve 100% success with near-optimal step counts. Monte Carlo is less stable with 86.0% success.

4.1.3 Agent Visualization

We recorded the behavior of the trained Q-Learning agent:



(a) Start (b) Mid-trajectory (c) Goal Reached

Figure 2: Visualizing the trained Q-Learning agent on `RandomEmptyEnv_10`. The agent efficiently navigates to the green goal square.

4.2 `RandomKeyMEnv_10`

4.2.1 Hyperparameters and Training

For this environment, we used the following configuration: $N_{episodes} = 10000$, $\gamma = 0.99$, $\alpha = 0.1$, $\epsilon_0 = 1.0$, $\epsilon_{min} = 0.1$, $\epsilon_{decay} = 0.999$ (decay every 5 episodes). The maximum steps per episode was set to 1000.

Reward Structure: The agent receives +1 upon termination (reaching the goal), +0.5 for picking up the key (can be done only once), +0.5 for opening the door (can be done only once), and a penalty of -0.01 for each step.

Figure 3 compares the training performance.

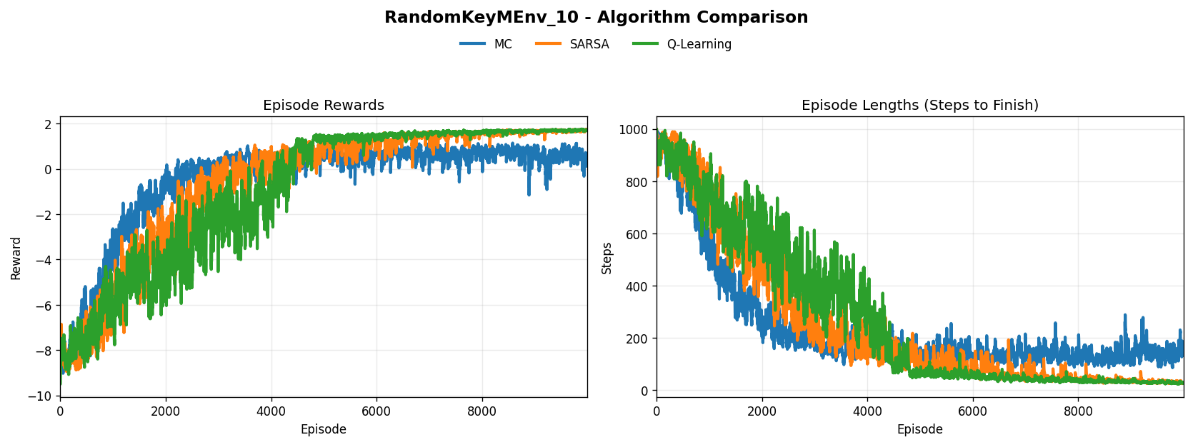


Figure 3: `RandomKeyMEnv_10`: Training reward comparison.

4.2.2 Inference Evaluation

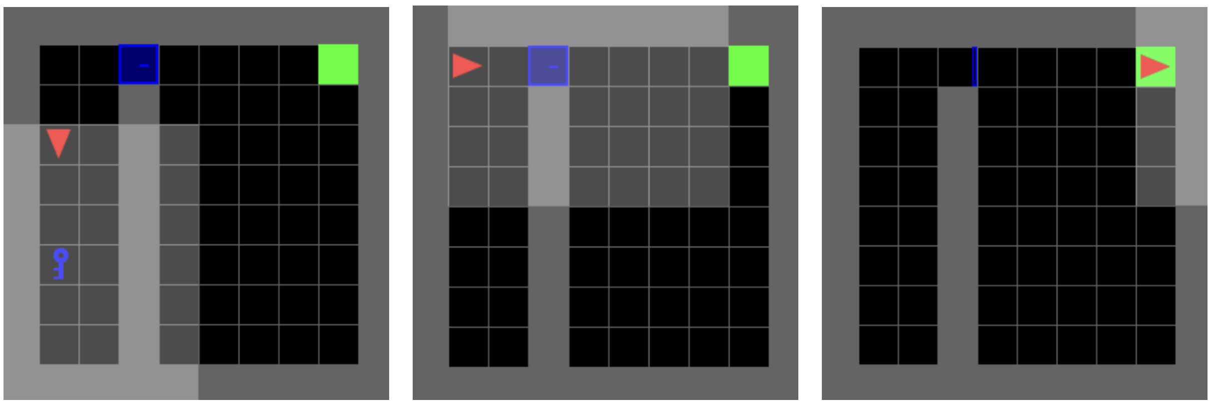
The trained policies were evaluated over 100 episodes. The results are summarized below:

Algorithm	Avg Reward	Avg Steps	Success Rate
Monte Carlo	−2.342	266.7	12.0%
SARSA	0.449	99.1	72.0%
Q-Learning	1.496	38.4	94.0%

Table 2: Performance metrics on `RandomKeyMEnv_10` (100 evaluation episodes).

4.2.3 Agent Visualization

We recorded the behavior of the trained agent:



(a) Pick up Key (b) Unlock Door (c) Goal Reached

Figure 4: Visualizing the interactions in `RandomKeyMEnv_10`.

4.3 Best hyperparameter summary

Env	Algo	α	γ	ϵ_{decay}	Inference avg steps
`RandomEmptyEnv_10`	First-Visit Monte Carlo	0.2	0.99	0.995	36.8
`RandomEmptyEnv_10`	SARSA(0)	0.2	0.99	0.995	9.9
`RandomEmptyEnv_10`	Q-Learning	0.2	0.99	0.995	9.2
`RandomKeyMEnv_10`	First-Visit Monte Carlo	0.1	0.99	0.999	266.7
`RandomKeyMEnv_10`	SARSA(0)	0.1	0.99	0.999	99.1
`RandomKeyMEnv_10`	Q-Learning	0.1	0.99	0.999	38.4

Table 3: Best hyperparameters and inference performance per environment and algorithm.

5 Discussion: Advantages and Disadvantages

We discuss the three algorithms in the context of our two missions and the observed results (Empty: MC 86%, SARSA/Q 100%; Key: MC 12%, SARSA 72%, Q-Learning 94%).

5.1 Monte Carlo (First-Visit Monte Carlo)

What worked well. In `RandomEmptyEnv_10`, First-Visit Monte Carlolearns a reasonable policy (86% success), since episodes are short and successful trajectories are discovered relatively early, allowing returns to update many visited states at once.

What failed / why. In `RandomKeyMEnv_10`, First-Visit Monte Carloperforms poorly (12% success, long episodes) because learning occurs only after an episode ends. When success trajectories are rare, most episodes provide weak or misleading learning signals, and the variance of returns is high (especially with long horizons and step penalty). Even with shaped rewards, the algorithm still needs many successful full trajectories to stabilize value estimates.

5.2 SARSA (SARSA(0))

What worked well. SARSA improves substantially over MC on `RandomKeyMEnv_10` (72% success). Because it bootstraps at every step, it can learn incrementally even before consistently reaching the final goal.

Tradeoff / limitation. SARSA is *on-policy*: it updates using the next action actually taken under ϵ-greedy exploration. This tends to learn more conservative policies that account for exploratory mistakes, which can increase robustness, but may slow convergence and lead to suboptimal paths compared to an off-policy greedy target.

5.3 Q-Learning (Q-Learning)

What worked well. Q-Learning achieved the best performance, especially on `RandomKeyMEnv_10` (94% success, 38.4 average steps). Being *off-policy*, it updates toward $\max_{a'} Q(s', a')$, which accelerates value propagation toward the optimal policy even while exploration remains high. This is particularly beneficial in multi-stage tasks (key $\rightarrow$ door $\rightarrow$ goal) where credit assignment is difficult.

Limitations. Q-Learning can suffer from maximization bias (overestimating action values due to the max operator), though in our small deterministic domains this did not prevent strong performance.

5.4 Environment-specific considerations and our approach

Reward shaping and step penalty. In `RandomKeyMEnv_10` we add intermediate rewards (+0.5 for picking the key once, +0.5 for opening the door once) and a per-step penalty (-0.01). This improves exploration by making partial progress informative and encourages shorter solutions during inference.

Action filtering. We exclude `drop` and `done` in `RandomKeyMEnv_10` (5 effective actions). This reduces wasted exploration and helps all methods concentrate on meaningful decisions.

State representation. Using the flattened full-grid observation (192-length tuple) makes the task fully observable and easy to implement, but it can increase the number of distinct visited states, slowing tabular learning. A more compact semantic state (agent position/direction + key/door/goal factors) could further improve sample efficiency.

6 Key Findings for MiniGrid

Algorithm choice matters most in multi-stage sparse tasks. On `RandomKeyMEnv_10`, Q-Learning (94%) significantly outperformed SARSA (72%) and Monte Carlo (12%), showing the advantage of off-policy TD updates with faster value propagation.

Credit assignment is the core difficulty in `RandomKeyMEnv_10`. The agent must solve a sequence of subgoals (get key $\rightarrow$ open door $\rightarrow$ reach goal). Monte Carlo struggles because successful full trajectories are rare; TD methods learn incrementally from partial progress.

Reward shaping helps exploration, but does not remove the need for efficient propagation. Intermediate rewards (+0.5 key, +0.5 door) and a step penalty (-0.01) make learning signals denser and encourage shorter solutions, but Q-Learning still benefits most from its greedy target updates.

Full-grid tabular states are simple but can be sample-inefficient. Our 192-length flattened grid encoding is fully observable, yet many distinct configurations may appear. This motivates either stronger TD methods (as seen here) or a more compact state encoding for better generalization.

Action filtering improves learning. Removing `drop` and `done` in `RandomKeyMEnv_10` reduces wasted exploration and improves convergence behavior.

7 Conclusion

This work compared three tabular reinforcement learning methods on two MiniGrid environments with sparse terminal rewards.

In the simple `RandomEmptyEnv_10` task, all algorithms were able to learn effective policies, with SARSA and Q-Learning achieving perfect success and near-optimal paths, while Monte Carlo showed higher variance and slightly lower stability.

In the more challenging `RandomKeyMEnv_10` environment, which requires solving a sequence of subgoals, Q-Learning proved to be the most robust and sample-efficient approach. Its off-policy updates enabled faster value propagation across the long horizon, leading to the highest success rate and shortest inference trajectories. SARSA achieved moderate performance, while Monte Carlo struggled due to high variance and the rarity of successful full episodes.

Overall, these results highlight the importance of algorithm choice, reward shaping, and exploration strategy when applying tabular reinforcement learning to multi-step, sparse-reward tasks with large state spaces.