Mid Project: Tabular Reinforcement Learning in MiniGrid

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

Author 1 Name (ID: `XXXXXXXXX`)
Author 2 Name (ID: `XXXXXXXXX`)
Course: RL2026A Semester: 2025

December 27, 2025

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`**.** An empty room (no internal obstacles) with a randomized agent start position and direction. The goal location is sampled from a small set of fixed coordinates. The episode ends upon reaching the goal or when the step limit is reached.

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.

1.1 Environment Type and Observability

Episodic: Yes. Each run is a finite-horizon episode 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. (If the "carrying key" information is not explicitly present in the flattened grid when the key is picked up, it can be appended as a small extra flag to make the representation strictly Markov.)

Action space: Discrete. MiniGrid provides a small finite set of actions (typically 7): turn-left, turn-right, move-forward, pick-up, drop, toggle (e.g., open door), and done.

State space: Discrete and finite (but large). We learn a tabular $Q(s, a)$. In practice, we use a discretized state representation (a state tuple) rather than the full raw grid vector.

Observability:

- **Default MiniGrid observation:** partially observable (agent-centric view).
- **With `KeyFlatObsWrapper`:** the observation is (almost) fully observable because it encodes the entire grid globally; only inventory-like information (e.g., holding the key) may require adding a small flag if not encoded.
- **State tuple used by our tabular agents:** $(x_{agent}, y_{agent}, dir, x_{goal}, y_{goal})$. This is sufficient to be Markov in `RandomEmptyEnv_10`, but for `RandomKeyMEnv_10`it should be extended with at least `hasKey` and `doorOpen` to avoid aliasing different situations into the same state.

2 State Representation and Q-table Size

2.1 State definition used in the notebook

Our agents map the environment to a discrete tuple:

$$s = (x_{agent}, y_{agent}, dir, x_{goal}, y_{goal})$$

with $dir \in \{0, 1, 2, 3\}$.

In a 10×10 MiniGrid with outer walls, the agent typically occupies interior coordinates $x, y \in \{1, \ldots, 8\}$ (8 possible values each), giving 64 positions.

Let $|\mathcal{A}|$ be the number of discrete actions (MiniGrid default is often $|\mathcal{A}| = 7$). If the goal is sampled from G possible locations, then the *theoretical* number of states is:

$$|\mathcal{S}| \approx 64 \times 4 \times G$$

and the theoretical Q-table size is $|\mathcal{S}| \times |\mathcal{A}|$. In practice, we store the Q-table as a dictionary keyed by visited states, so memory grows with visited states.

2.2 State size estimates (fill $|\mathcal{A}|$ if different)

Environment	Interior positions	Directions	Goal choices G	Theoretical $\lvert\mathcal{S}\rvert$
`RandomEmptyEnv_10`	64	4	3	$64 \cdot 4 \cdot 3 = 768$
`RandomKeyMEnv_10`	64	4	2	$64 \cdot 4 \cdot 2 = 512$

Table 1: Theoretical number of discrete states under the state tuple used in the notebook.

Important note for `RandomKeyMEnv_10`. The above count ignores key possession and door status. A more Markov state for `RandomKeyMEnv_10`should include:

$$s' = (x, y, dir, x_g, y_g, \texttt{hasKey}, \texttt{doorOpen})$$

which would multiply $|\mathcal{S}|$ by up to 4 (two binary flags). If key position is included when not carried, the state space becomes larger.

3 Algorithms

We compare three tabular algorithms under epsilon-greedy exploration.

3.1 First-Visit Monte Carlo

Type: On-policy, episodic method.
We generate a complete episode, compute returns G_t, and update $Q(s,a)$ (first-visit) toward the observed return. Using a constant step-size α yields an incremental Monte Carlo update:

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

Strengths: no bootstrap bias; stable in episodic settings.
Weaknesses: high variance; slow on sparse-reward tasks because reward arrives only at episode end.

3.2 SARSA(0)

Type: On-policy TD control.
Updates bootstrapped target using the next action chosen by the current behavior policy:

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

Strengths: typically safer/conservative with epsilon-greedy; can learn stable policies in stochastic settings.
Weaknesses: can converge slower than off-policy methods to the optimal greedy policy.

3.3 Q-Learning

Type: Off-policy TD control.
Bootstraps toward the greedy next-state value:

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

Strengths: often more sample-efficient; learns the greedy policy while exploring.
Weaknesses: can be unstable with function approximation; in tabular settings can still suffer from overestimation in noisy tasks.

4 Experimental Setup

4.1 Reward and termination

Both environments use a **sparse terminal reward**: $r = 1$ only when reaching the goal, otherwise $r = 0$. Episodes terminate on success or when the maximum step limit is reached.

4.2 Discount factor γ

We use a high discount factor (e.g., $\gamma = 0.99$) because rewards are delayed and reaching the goal may require many steps, especially in `RandomKeyMEnv_10`. Lower γ values can prefer short-term behavior and may hinder learning with sparse rewards.

4.3 Exploration–Exploitation strategy

We use **epsilon-greedy** exploration:

$$a = \begin{cases} \text{random action} & \text{w.p. } \epsilon \\ \arg\max_a Q(s,a) & \text{w.p. } 1-\epsilon \end{cases}$$

with multiplicative decay $\epsilon \leftarrow \max(\epsilon_{min}, \epsilon \cdot \epsilon_{decay})$. We also discuss alternative strategies in the Discussion section.

4.4 Hyperparameters and initializations

We test:

- Learning rate $\alpha \in \{\dots\}$
- Epsilon decay $\epsilon_{decay} \in \{\dots\}$
- Discount $\gamma \in \{\dots\}$
- Initialization: zero initialization vs. optimistic initialization $Q_0 > 0$

(Replace the sets above with the exact grids used in the notebook.)

5 Results

This section must include:

- **Convergence graphs:** reward (or success rate) vs. episodes for each algorithm.
- **Steps-to-solve:** number of steps to finish the episode for the most successful run, and average steps curves.
- **Inference evaluation:** greedy policy performance (average steps and success rate) measured over multiple episodes.
- **Videos:** middle of training and after convergence (included in notebook).

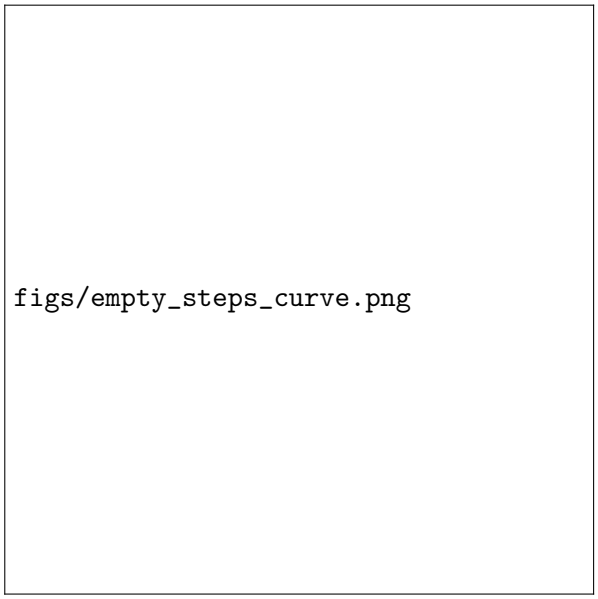
5.1 Training curves (reward / success rate)

(a) `RandomEmptyEnv_10`: reward (or success rate) vs. episodes.

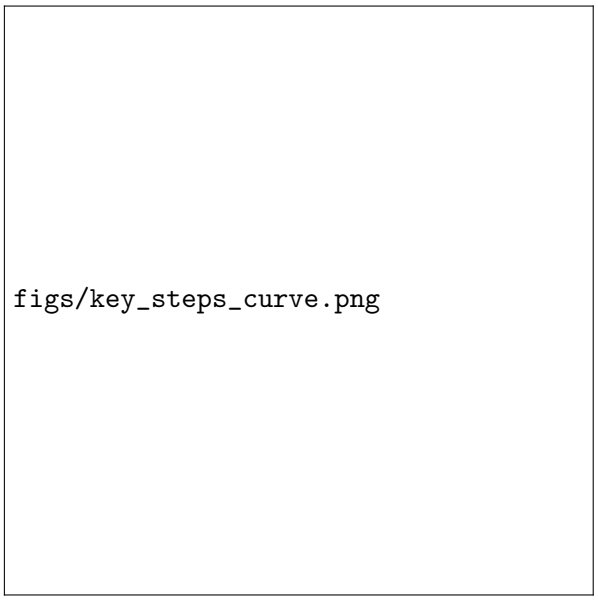
(b) `RandomKeyMEnv_10`: reward (or success rate) vs. episodes.

Figure 1: Training convergence curves comparing First-Visit Monte Carlo, SARSA(0), and Q-Learning.

5.2 Training curves (steps per episode)



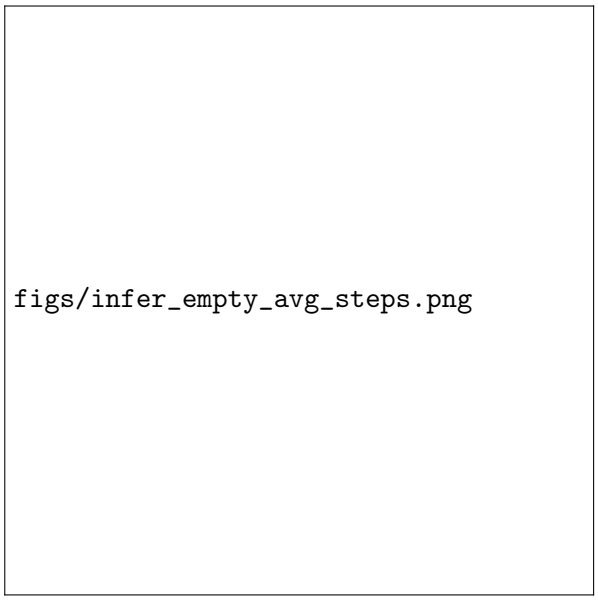
(a) `RandomEmptyEnv_10`: steps/episode vs. episodes.



(b) `RandomKeyMEnv_10`: steps/episode vs. episodes.

Figure 2: Training efficiency: average steps per episode during learning.

5.3 Inference evaluation (greedy policy)



(a) `RandomEmptyEnv_10`: inference average steps.

(b) `RandomKeyMEnv_10`: inference average steps.

Figure 3: Inference stage: greedy policy evaluated over N episodes (report N).

5.4 Best parameters cell (required)

In the notebook, include a dedicated cell named `BEST_PARAMS` that clearly reports:

- Best hyperparameters per environment and algorithm: α, γ, ϵ_0, ϵ_{decay}, ϵ_{min}

- Best training run summary: episodes to reach consistent success, final success rate, and typical steps to solve
- Best inference summary: average steps and success rate for greedy policy

In this report, summarize the chosen best setting in Table 2.

Env	Algo	α	γ	ϵ_{decay}	Inference avg steps
`RandomEmptyEnv_10`	First-Visit Monte Carlo	...	...	...	...
`RandomEmptyEnv_10`	SARSA(0)	...	...	...	...
`RandomEmptyEnv_10`	Q-Learning	...	...	...	...
`RandomKeyMEnv_10`	First-Visit Monte Carlo	...	...	...	...
`RandomKeyMEnv_10`	SARSA(0)	...	...	...	...
`RandomKeyMEnv_10`	Q-Learning	...	...	...	...

Table 2: Best hyperparameters and inference performance (fill from notebook).

6 Discussion

6.1 Advantages and disadvantages on our missions

Sparse reward impact. Both tasks provide reward only at success, which makes exploration the main difficulty. First-Visit Monte Carlooften learns slower because it must complete successful trajectories to propagate credit information to earlier states. TD methods (SARSA(0), Q-Learning) can propagate value estimates earlier via bootstrapping, typically improving sample efficiency.

On-policy vs off-policy. SARSA(0)learns values consistent with the behavior policy (epsilon-greedy), which can yield safer behavior during learning. Q-Learninglearns a greedy target while exploring, often converging faster to an optimal greedy policy, but can be more sensitive to exploration settings.

Environment differences. `RandomEmptyEnv_10`is mostly navigation, so the reduced state (x, y, dir, x_g, y_g) is close to Markov and learning is comparatively easy. `RandomKeyMEnv_10`requires reasoning about *key possession* and *door state*; if these are excluded from the state representation, learning becomes harder because distinct situations collapse into the same tabular state.

6.2 Exploration–Exploitation tradeoff

We used epsilon-greedy with decay. Alternative or complementary approaches:

- **Optimistic initialization:** start $Q(s, a)$ above 0 to encourage exploration early.
- **Exploring starts:** force random starting states/actions (when allowed) to improve coverage.
- **Different decay schedules:** linear decay, piecewise decay, or keeping a non-trivial ϵ_{min}.
- **Count-based exploration (tabular):** add bonuses for rarely visited (s, a).

6.3 Strengths and limitations of our approach

Good points. Tabular methods are simple, interpretable, and fast to run. They are a strong baseline for small discrete tasks and allow clear comparisons between MC and TD control.

Bad points. State design is critical; an incomplete state (especially in `RandomKeyMEnv_10`) breaks the Markov property and can prevent full convergence. Also, tabular approaches do not scale to larger grids or richer observations without function approximation.

7 Conclusion

Summarize which algorithm performed best in each environment and why (based on your plots and inference metrics), and propose improvements (better state, exploration bonuses, or function approximation).

Reproducibility Checklist

- Provide Colab notebook link: `<paste link>`
- Report random seeds used (if any): `<paste seeds>`
- Mention training episodes and evaluation episodes: `<numbers>`
- List best hyperparameters (Table 2)