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

January 3, 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`. 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 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 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: The environment becomes *fully observable*. The wrapper exposes a global, flattened representation of the grid, and we additionally extract all relevant latent variables (agent position and direction, goal position, key position, door position, whether the agent is carrying the key, and whether the door is open).

2 State Representation and Q-table Size

2.1 State definition used in the notebook

Our tabular agents map each observation to a discrete state tuple extracted from the unwrapped MiniGrid environment:

$$s = (x_a, y_a, d_a, x_g, y_g, x_k, y_k, x_d, y_d, \texttt{hasKey}, \texttt{doorOpen})$$

Each component of the state tuple has the following meaning:

- x_a, y_a – the agent's grid coordinates.
- d_a – the agent's orientation (direction), with $d_a \in \{0, 1, 2, 3\}$ corresponding to the four cardinal directions.
- x_g, y_g – the goal position coordinates.
- x_k, y_k – the key position coordinates (or $(-1, -1)$ if no key exists or the key is not present).
- x_d, y_d – the door position coordinates (or $(-1, -1)$ if no door exists).
- `hasKey` – a binary flag indicating whether the agent is currently carrying the key.
- `doorOpen` – a binary flag indicating whether the door is open.

This representation is fully observable and satisfies the Markov property.

Environment-dependent components. Not all state components contribute to the effective state-space complexity:

- In `RandomEmptyEnv_10` (RandomEmptyEnv10), there is no key or door. The variables (x_k, y_k) and (x_d, y_d) are fixed sentinel values $(-1, -1)$, and $\texttt{hasKey} = \texttt{doorOpen} = 0$. These components are constant and therefore do *not* increase the number of distinct states.
- In `RandomKeyMEnv_10`, all components are active and affect the transition dynamics.

2.2 State space size estimation

We consider a 10×10 MiniGrid with outer walls, so the agent occupies interior coordinates $x, y \in \{1, \dots, 8\}$, yielding 64 positions, and 4 orientations.

`RandomEmptyEnv_10`. The goal is sampled from $G = 3$ possible positions. Since all other variables are constant:

$$|\mathcal{S}_{\text{Empty}}| = 64 \times 4 \times 3 = 768$$

RandomKeyMEnv_10. In the key–door environment:

- the goal has $G = 2$ possible locations,
- the door is always located in column 3 ⇒ 8 possible positions,
- the key is located on the left side of the wall (columns 1 and 2), giving $2 \times 8 = 16$ possible positions,
- hasKey $\in \{0, 1\}$,
- doorOpen $\in \{0, 1\}$.

The resulting theoretical state-space size is:

$$|\mathcal{S}_{\text{Key}}| = 64 \times 4 \times 2 \times 8 \times 16 \times 2 \times 2 = 262{,}144$$

2.3 Summary of theoretical state sizes

Environment	Agent pos.	Orientation	Additional factors	$\lvert\mathcal{S}\rvert$
RandomEmptyEnv_10	64	4	3 goal positions	768
RandomKeyMEnv_10	64	4	$2 \cdot 8 \cdot 16 \cdot 2 \cdot 2$	262,144

Table 1: Theoretical number of discrete states under the full state representation used by the tabular agents.

Let $|\mathcal{A}|$ be the number of discrete actions : envEmpty $|\mathcal{A}| = 3$, envKey $|\mathcal{A}| = 7$.

Q-table size. The Q-table size is $|\mathcal{S}| \times |\mathcal{A}|$. In practice, Q-values are stored in a dictionary indexed by visited states, so memory usage scales with the number of states actually encountered during training rather than the full theoretical state space.

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 X 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:

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

Advantages in our MiniGrid missions:

- No bootstrapping bias: updates rely solely on actual observed returns.
- Conceptually simple and stable in strictly episodic environments.

Disadvantages in our MiniGrid missions:

- Very high variance due to sparse terminal rewards.
- Credit assignment is difficult: all actions in a long episode receive the same delayed reward.
- Particularly slow in `RandomKeyMEnv_10`, where episodes are long and successful trajectories are rare early in training.

As a result, Monte Carlo learning converges slowly and requires many episodes to propagate useful information back to early states.

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:

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

Advantages in our MiniGrid missions:

- Lower variance than Monte Carlo due to step-wise bootstrapping.
- Learns policies that account for exploration, leading to more conservative behavior.
- More stable learning than MC in sparse-reward settings.

Disadvantages in our MiniGrid missions:

- Convergence to the optimal greedy policy can be slower than off-policy methods.
- Continued exploration (nonzero ϵ) affects the learned value function.

In `RandomEmptyEnv_10`, SARSA learns efficiently due to the smaller state space. In `RandomKeyMEnv_10`, its conservative updates help stability but slow down optimal path learning.

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:

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

Advantages in our MiniGrid missions:

- More sample-efficient than MC and SARSA.
- Faster propagation of terminal rewards through the state space.
- Performs well in large discrete state spaces such as `RandomKeyMEnv_10`.

Disadvantages in our MiniGrid missions:

- Can overestimate action values due to the max operator.
- Sensitive to learning rate and exploration schedule.

Overall, Q-learning shows the fastest convergence in both environments, especially in the key–door task, where efficient reward propagation is crucial.

3.4 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.)

4 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).

4.1 Training curves (reward / success rate)

`figs/empty_reward_curve.png`

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

`figs/key_reward_curve.png`

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

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

4.2 Training curves (steps per episode)

figs/empty_steps_curve.png

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

figs/key_steps_curve.png

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

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

4.3 Inference evaluation (greedy policy)

figs/infer_empty_avg_steps.png

(a) RandomEmptyEnv_10: inference average steps.

figs/infer_key_avg_steps.png

(b) RandomKeyMEnv_10: inference average steps.

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

4.4 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).

5 Discussion

5.1 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_10is mostly navigation, so the reduced state (x, y, dir, x_g, y_g) is close to Markov and learning is comparatively easy. RandomKeyMEnv_10requires 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.

5.2 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).

5.3 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.

6 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)