Comparison of Various Tabular Method Algorithms on Grid World

CAP 6629 Reinforcement Learning Project 2

Fanchen Bao

1 Introduction

One of the goals in reinforcement learning is to have an agent reach a terminal state in a Markov Decision Process (MDP) finite episodic task that follows in the best way possible. An MDP finite episodic task is something that has a clear end goal, can be broken down into discrete stages, and the next stage is solely dependent on the state of the current stage, such as finding a route in a maze, playing a board game, etc. The definition of ”best way possible” is dependent on the task itself. For finding a route in a maze, the best way would be a successful route with least number of steps. For playing a board game, the best way would simply be to win. To generalize MDP finite episodic task, we can view it as a collection of all states that could possibly occur in the task. A reinforcement learning agent would begin with the starting state, visit certain number of intermediate states, and end up with the terminal state. Given that the starting and terminal states are known, the question becomes how to make the agent find the best intermediate states by itself. One way to guide the agent is to offer proper rewards during state transition. If a state transition is desired, e.g. a deterministic winning move in a game, it is rewarded handsomely. If not, it is rewarded less or even penalized. This way, as the agent moves across intermediate states, it accumulates knowledge of the overall value of each state. If from state A the terminal state can immediately be reached, the handsome reward offered means that state A has high value. Likewise, if from state B, the agent is stuck in place, any transition from state B onward would have low or no reward, thus making it a state of low value. If an agent is able to visit all intermediate states and experience all state transitions, it will know the value of all states. Then the task of going from the starting to the terminal state becomes trivial —from any state, the agent only needs to look around the next states it can access, pick the one with the highest value, move to that state, and repeat until the terminal state is reached.

It is apparent that the key of solving a MDP finite episodic task is to compute the values of each intermediate state. This is exactly the focus of reinforcement learning —finding algorithms that can make such computation possible. Tabular method is one approach in reinforcement learning to compute state values. While there are many algorithms in tabular method, they all rely on visiting the majority of the states at least once and tabulating the states as an array or matrix such that the value of any state can be easily stored, modified, and retrieved. This setup clearly indicates that tabular method is not suitable for tasks with a lot of states (unfortunately most real life application of reinforcement learning has to deal with astronomical number of states, which means tabular method is rarely used in practice), yet for a finite episodic task with manageable number of states, such as grid world, tabular method is the right tool for the task.

Grid world is essentially a maze, with well defined boundary and rules. Figure 1 illustrates a simple 10×10 grid world used in this report. The yellow square is the start point; the green square is the end point; and the brown squares are the blocks that do not allow traverse. An agent on this grid world can choose any of the four directions to go: up, down, left, right, except when such direction would lead to the agent going out of bound or hitting a block. Each legal step incurs a reward of -1. The task for the agent is to find the shortest path to go from the start to the end square.

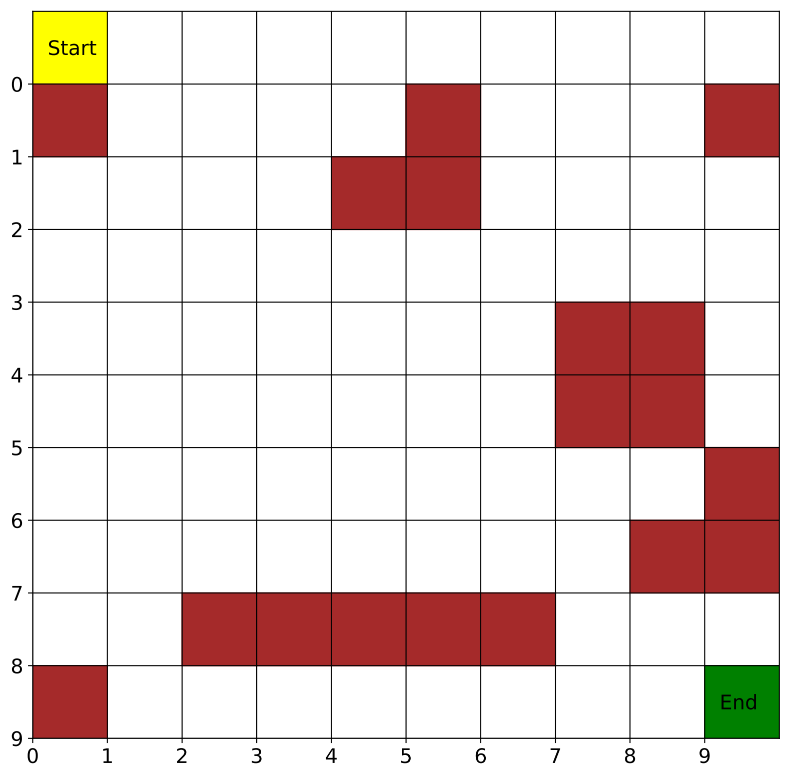


Figure 1: A Grid World with Obstacles

The aim of this report is to use the grid world shown in Figure (1) as a simple MDP finite episodic task, upon which various algorithms in tabular method can be implemented and compared. By doing this, we will have a better understanding of the performance of each algorithm and the gotchas in algorithm implementation. The algorithms we will tackle are listed as follows:

- Dynamic Programming
  - Iterative Policy Evaluation
  - Policy Iteration
  - Value Iteration
- Monte Carlo
  - First Visit Policy Evaluation
  - Exploring Starts
  - On-policy Control
- Temporal Difference
  - TD(0)
  - Sarsa with TD Control
  - Q-learning with TD Control
- Eligibility Tracing
  - TD(λ)

– Sarsa(λ)
– Watkins's Q(λ)

This report is organized in the following manner. Section 2 introduces mathematical notations and formulas that model an MDP finite episodic task. Section 3 provides pseudo-code for each tabular method algorithm listed above. Section 4 evaluates the performance of each algorithm, including a step-to-go curve, a residue mean squared (RMS) error curve, the state values, and a visualization of the path found in the grid world. Finally, section 5 provides insight into the algorithm comparison, the implementation gotchas, and concludes the report.

2 Formal Definition

Given a current state S_t at time step t of an MDP, and given that it will take in total T time steps to reach the terminal state from S_t, the value of S_t, or future reward, can be estimated as G_t as in Equation (1), where R_{t+1}, R_{t+2}, ... are rewards in future state transitions.

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{1}$$

However, Equation (1) becomes problematic if T is infinite. Thus, we need another expression for G_t that converges. Furthermore, we are always more interesting in the rewards of the state transition close to S_t than those further away. Thus, we can rewrite G_t as in Equation (2), where $0 \leq \gamma < 1$ and the time extends to infinity. Here, G_t is considered discounted future reward and γ the discount rate. With the addition of γ, G_t always converges and it places more attenuation towards the reward happening further away from S_t.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2}$$

Notice that Equation (2) reveals that there exists a recursive relation in G_t, where the current discounted future reward can be computed by the discounted future reward of the next time step plus the reward to transition to the next time step. This recursive relation means that it is possible to estimate G_t without having to go through all possible states. This idea is central to all on-line tabular algorithms.

Let π be a policy that determines the probability that each state transition, i.e. action, takes place for any state, we can write the state value function for policy π (recall that a state value is the representation of how valuable a state is for an agent to reach the terminal state) as Equation (3).

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right] \end{aligned} \tag{3}$$

Here, $\pi(a|s)$ is the probability that an action a is taken at state s based on policy π. $p(s', r \mid s, a)$ is the state transition probability, which describes the probability of transitioning from state s to next state s' via action a and receiving reward r. $r + \gamma v_\pi(s')$ is the estimated discounted future reward of the current state s based on one of its next state s' using the recursive relation described in Equation (2). Since $v_\pi(s)$ is an expected value, we need to sum up all the discounted future reward estimates, weighted by their probabilities. Equation (3) is also known as the *Bellman equation for* v_π.

While state value represents how valuable each state is with regards to reaching the terminal state, we can dig further into each state and consider how valuable each action is. If the action values of each state is known, we will know exactly what action to take at each state. This way of thinking focuses on the action value (note that each action value is associated with a state-action pair), and similar to state value, we can also write an action-value function for policy π as shown in Equation (4).

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \end{aligned} \tag{4}$$

Now that we know the state and action value function for policy π, the next step in reinforcement learning is to find the best such policy π among all policies. In other words, we want to find the optimal state-value function $v_*(s)$ and action-value function $q_*(s,a)$. The optimal action-value function can be written as Equation (5).

$$\begin{aligned} q_*(s,a) &= \max_\pi q_\pi(s,a) \\ &= \max_\pi \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right] \end{aligned} \tag{5}$$

Note that in Equation (7), we rewrite $q_*(s,a)$ in terms of $v_*(s)$. The intuition behind it is that the optimal action value of a state is the expected value of the optimal state value itself.

With $q_*(s,a)$ expressed in terms of $v_*(s)$, we can tackle the optimal state-value function more comfortably in Equation (6).

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s) \\ &= \max_a q_*(s,a) \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right] \end{aligned} \tag{6}$$

Similarly, we can also feed Equation (6) back to Equation (5) to get Equation (7).

$$q_*(s,a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{7}$$

Equations (6) and (7) are also called *Bellman optimality equations*. They allow the expression of $v_*(s)$ and $q_*(s,a)$ by themselves, which means they can be approximated iteratively in algorithm. In other words, the *Bellman optimality equations* bridge the gap between the concept of reinforcement learning and algorithmic implementation. Its importance is paramount.

3 Tabular Method

Tabular method is one approach to solve the *Bellman optimality equations*. It creates a table (array or matrix) to hold state values of all states, or action values of all state-action pairs. The various algorithms listed in Section 1 are different implementations of tabular method. Their main goal is to converge an arbitrarily initialized $V(s)$ to the optimal state value function $v_*(s)$, or similarly $Q(s,a)$ to $q_*(s,a)$. Once a converged $V(s)$ or $Q(s,a)$ is obtained, obtaining the optimal policy is trivial. Below, we will provide pseudo-code for each algorithm and provide clarification if necessary. All pseudo-codes are adapted from [1].

3.1 Dynamic Programming

Given a current state s, Dynamic Programming visits all of its next state s' to establish the most accurate state or action value for s. Since all next states need to be visited, Dynamic Programming is very computation-intensive when the state space becomes big.

3.1.1 Iterative Policy Evaluation

This is the most pure Dynamic Programming. For each state, we examine all possible next states to obtain all possible estimated discounted future rewards, and compute the expected state value based on their weighted average. The pseudo-code is provided in Algorithm 1.

Algorithm 1 DP - Iterative Policy Evaluation

```
 1: $V(s) \leftarrow 0$ for all $s \in \mathcal{S}$        ▷ Set state values of all states to zero
 2: ep $\leftarrow 0$        ▷ keep track of the number of episodes experienced
 3: max_eps        ▷ Maximum number of episodes allowed
 4:
 5: repeat
 6:     for each $s \in \mathcal{S}$ do:
 7:         $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$        ▷ Evaluate policy for each state
 8:     end for
 9:     ep $\leftarrow$ ep $+ 1$
10: until ep = max_eps
11: return $V(s)$        ▷ After sufficient number of episodes, $V(s)$ converges to $v_*(s)$
```

3.1.2 Policy Iteration

This algorithm contains two parts. The first part is policy evaluation, which estimates state values based on the current policy. The second part is policy improvement, which updates the policy by choosing the action that can lead to the highest discounted future reward. The benefit of doing this is that each policy evaluation builds on top of what has been learned previously, which can reduce the number of episodes needed to converge $V(s)$ to $v_*(s)$. The pseudo-code is provided in Algorithm 2.

3.1.3 Value Iteration

This algorithm always updates state value with the best estimated discounted future reward. In a sense, it mixes policy improvement into policy evaluation, and thus does not need to improve the policy separately. Each episode, the algorithm moves each state value towards the maximum. Hence it approaches the optimal state value function. The pseudo-code is provided in Algorithm 3.

Algorithm 2 DP - Policy Iteration

```
 1: $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ for all $s \in \mathcal{S}$    ▷ Initial value for $V(s)$ and $\pi(s)$ can be arbitrary
 2: max_eps    ▷ Maximum number of episodes allowed
 3:
 4: ep ← 0    ▷ Begin policy evaluation
 5: repeat
 6:     for each $s \in \mathcal{S}$ do:
 7:         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \, [r + \gamma V(s')]$    ▷ Estimate state value based on the current policy
 8:     end for
 9:     ep ← ep + 1
10: until ep = max_eps
11:
12: policy_stable ← $true$    ▷ Begin policy improvement
13: for each $s \in \mathcal{S}$ do:
14:     $a \leftarrow \pi(s)$
15:     $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \, [r + \gamma V(s')]$    ▷ Obtain the best action
16:     if $a \neq \pi(s)$ then    ▷ Policy stable when current policy already optimal
17:         policy_stable ← $false$
18:     end if
19: end for
20: if policy_stable then
21:     return $V(s), \pi(s)$    ▷ $V(s)$ converges to $v_*(s)$, $\pi(s)$ is also optimal
22: else
23:     Go to line 4    ▷ Need more policy evaluation
24: end if
```

Algorithm 3 DP - Value Iteration

```
 1: $V(s) \in \mathbb{R}$ for all $s \in \mathcal{S}$    ▷ Initial value for $V(s)$ can be arbitrary
 2: ep ← 0    ▷ keep track of the number of episodes experienced
 3: max_eps    ▷ Maximum number of episodes allowed
 4:
 5: repeat
 6:     for each $s \in \mathcal{S}$ do:
 7:         $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) \, [r + \gamma V(s')]$    ▷ Compute state value using the best estimated discounted future reward
 8:     end for
 9:     ep ← ep + 1
10: until ep = max_eps
11: return $V(s)$    ▷ After sufficient number of episodes, $V(s)$ converges to $v_*(s)$
```

3.2 Monte Carlo

In contrast to Dynamic Programming, Monte Carlo method does not visit all next states of a current state. Instead, it visits only one next state and continuously moves from one state to the next until the terminal state is reached. By doing so, Monte Carlo is able to provide the true discounted future rewards for any state involved in an episode. When sufficient number of episodes are executed, the average of the discounted future rewards received for a particular state converges to the optimal state value. A hard requirement for Monte Carlo is that there must exist a terminal state and it must be reachable. This limits its usage. Furthermore, although Monte Carlo does not fan out in visiting the next state, it could still be computation-intensive if it takes a long time for each episode to reach the terminal state.

3.2.1 First-visit Policy Evaluation

This is the most basic Monte Carlo algorithm. Given a start state, a random episode that ends in the terminal state is generated. For each first encountered state, its true discounted future rewards for the current episode is computed and stored in a separate table (i.e. if the same state is encountered again later in an episode, its discounted future reward is not counted). Repeat such random episodes many times, the average of the discounted future rewards for each state becomes the state value. The pseudo-code is provided in Algorithm 4.

Algorithm 4 MC - First-visit Policy Evaluation

```
1: $V(s) \in \mathbb{R}$ for all $s \in \mathcal{S}$          ▷ Initial state value can be arbitrary
2: Returns($s$) ← empty list for all $s \in \mathcal{S}$ ▷ Record the discounted future reward assigned to a
   state for different episodes
3: ep ← 0                         ▷ keep track of the number of episodes experienced
4: max_eps                                ▷ Maximum number of episodes allowed
5:
6: repeat
7:     Generate an episode from start to terminal state
8:     for each $s \in$ episode do:
9:         $G$ ← discounted future reward for the first encounter of $s$
10:        Append $G$ to Returns($s$)
11:        $V(s)$ ← average(Returns($s$))
12:    end for
13:    ep ← ep + 1
14: until ep = max_eps
15: return $V(s)$               ▷ After sufficient number of episodes, $V(s)$ converges to $v_*(s)$
```

3.2.2 Exploring Starts

Instead of always starting from the start state, exploring starts make any state as a start state and continue towards the terminal state. The benefit of doing this is that it is faster to visit all states at least once, which in turn improves the state value estimation more quickly. Also exist in exploring starts is policy improvement and policy-based action choice. The involvement of policy reduces the use of random action, which potentially shortens the length of each episode. The combination of random start and policy-based episode allows exploring starts to converge $Q(s, a)$ to $q_*(s, a)$ faster. The pseudo-code is provided in Algorithm 5.

Algorithm 5 MC - Exploring Starts

```
1: for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$ do:
2:     $Q_(s, a) \leftarrow$ arbitrary                    ▷ Initialize arbitrary action values
3:     $\pi_(s) \leftarrow$ arbitrary                      ▷ Initialize arbitrary policy
4:     Returns$(s, a) \leftarrow$ empty    ▷ Record the discounted future reward assigned to a state-action pair for different episodes
5: end for
6: ep $\leftarrow 0$                      ▷ keep track of the number of episodes experienced
7: max_eps                             ▷ Maximum number of episodes allowed
8:
9: repeat
10:    Randomly choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$
11:    Generate an episode from $(S_0, A_0)$ to the terminal state following $\pi(s)$
12:    for each $(s, a) \in$ episode do:
13:        $G \leftarrow$ discounted future reward for the first encounter of $(s, a)$
14:        Append $G$ to Returns$(s, a)$
15:        $Q(s, a) \leftarrow$ average(Returns$(s, a)$)
16:    end for
17:    for each $(s, a) \in$ episode do:                 ▷ Update policy
18:        $\pi(s) \leftarrow \arg\max_a Q(s, a)$
19:    end for
20:    ep $\leftarrow$ ep $+ 1$
21: until ep = max_eps
22: return $V(s), \pi(s)$  ▷ After sufficient number of episodes, $V(s)$ converges to $v_*(s)$ and $\pi(s)$ is optimal
```

3.2.3 On-policy Control

In exploring starts, we must randomly choose starting point because the action is generated deterministically from the policy. If we can make action generation, i.e. the control of states, stochastic, then there is no need to use random start. On-policy control addresses this concern directly, by using the ϵ-greedy method to generate action based on the policy. Recall that ϵ-greedy method explore with probability of ϵ, and exploit with the remaining probability. The benefit of not using random starts is that each start can be fixed on the desired starting point, which makes each episode more representative of what the actual solution would look like and the accumulated discounted future reward more relevant to the optimal state value function. The pseudo-code is provided in Algorithm 6.

3.3 Temporal Difference

Temporal difference (TD) algorithm is akin to Monte Carlo in that it also only visits one next state at each state transition. However, it updates state value along the way, unlike Monte Carlo where an episode must terminate before state value can be updated. The benefit of doing this is that by the time an episode terminates, state values have already been updated. Moreover, since state value is updated while an episode is ongoing, it is likely that the episode itself would be influenced by the change in state value, leading to faster termination. These benefits speed up the learning process. A new parameter α is added to the TD algorithm, which signifies the proportion that a state value's old value and the estimated discounted future reward should contribute to creating a new state value.

Algorithm 6 MC - On-policy Control

```
1:  for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$ do:
2:      $Q_(s,a) \leftarrow$ arbitrary                    ▷ Initialize arbitrary action values
3:      $\pi_(a|s) \leftarrow$ arbitrary    ▷ Initialize arbitrary policy with the probabilities of each action
    sum up to 1
4:      Returns$(s,a) \leftarrow$ empty          ▷ Record the discounted future reward assigned to a
    state-action pair for different episodes
5:  end for
6:  ep $\leftarrow 0$                         ▷ keep track of the number of episodes experienced
7:  max_eps                                 ▷ Maximum number of episodes allowed
8:
9:  repeat
10:     Generate an episode from the start to the terminal state following $\pi(s)$
11:     for each $(s,a) \in$ episode do:
12:         $G \leftarrow$ discounted future reward for the first encounter of $(s,a)$
13:         Append $G$ to Returns$(s,a)$
14:         $Q(s,a) \leftarrow$ average(Returns$(s,a)$)
15:     end for
16:     for each $s \in$ episode do:                  ▷ Reassign probability of each action
17:         $A^* \leftarrow \arg\max_a Q(s,a)$
18:         for all $a \in \mathcal{A}(s)$ do
19:             $\pi(a|s) \leftarrow \begin{cases} 1-\epsilon+\frac{\epsilon}{|\mathcal{A}(s)|} & a = A^* \\ \frac{\epsilon}{|\mathcal{A}(s)|} & a \neq A^* \end{cases}$
20:         end for
21:     end for
22:     ep $\leftarrow$ ep $+ 1$
23: until ep = max_eps
24: return $Q(s,a), \pi(a|s)$    ▷ After sufficient number of episodes, $Q(s,a)$ converges to $q_*(s,a)$
    and $\pi(a|s)$ is optimal
```

3.3.1 TD(0)

This is the most basic form of TD, where value update depends on one-step look ahead, i.e. state value is estimated only by one next state value. Since no stochastic procedure is involved in policy, the action taken at each state must be random, instead of based on the current state values. This is quite an important point, because if action is chosen based on state values, an agent is likely stuck in some local maximum and unable to reach the terminal state. The pseudo-code is provided in Algorithm 7.

Algorithm 7 TD(0)

```
 1: $V(s) \in \mathbb{R}$ for all $s \in \mathcal{S}$          ▷ Initial state value can be arbitrary
 2: ep ← 0                          ▷ keep track of the number of episodes experienced
 3: max_eps                                  ▷ Maximum number of episodes allowed
 4:
 5: repeat
 6:     $S \leftarrow$ the start state
 7:     repeat
 8:         $A \leftarrow$ a random action at $S$
 9:         Take action $A$ and observe reward $R$ and next state $S'$
10:         $V(S) \leftarrow V(S) + \alpha\,[R + \gamma V(S') - V(S)]$
11:         $S \leftarrow S'$
12:     until $S$ is terminal
13:     ep ← ep + 1
14: until ep = max_eps
15: return $V(s)$            ▷ After sufficient number of episodes, $V(s)$ converges to $v_*(s)$
```

3.3.2 Sarsa with TD Control

This is the control version (i.e. estimating action value instead of state value) of TD(0). In addition to using action value, another major difference between Sarsa with TD control and TD(0) is that Sarsa employs the ϵ-greedy method to incorporate exploration in action generation. This way, action generation can be based on the best current action value (exploitation), while at the same time maintain the ability to jump out of any traps using exploration. The benefit of relying more on exploitation is to speed up action value convergence. The pseudo-code is provided in Algorithm 8.

3.3.3 Q-learning with TD Control

Q-learning with TD Control is almost the same as Sarsa with TD control, except that during action value update, the maximum of the next action values is used (in Sarsa with TD control, the action value corresponding to the generated action is used). This means in Q-learning with TD control, it is possible that the next action generated by *epsilon*-greedy method is NOT the owner of the next action value that is used to update the current action value. In other words, Q-learning with TD control guarantees using the best value to update action value, while at the same time also maintains the ability to explore via *epsilon*-greedy method. The benefit of doing this is that it combines the best practices of both action value update and action generation. The pseudo-code is provided in Algorithm 9.

3.4 Eligibility Tracing

While TD algorithm is faster than Monte Carlo or Dynamic Programming, its shortcoming is that each state or action value update only performs a one-step look-ahead. If a method allows

Algorithm 8 Sarsa with TD Control

```
1: $Q(s,a) \leftarrow$ arbitrary value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
2: Set all action values of the terminal state to 0.    ▷ Ensure that the state value of the terminal state is always the highest
3: ep $\leftarrow 0$    ▷ keep track of the number of episodes experienced
4: max_eps    ▷ Maximum number of episodes allowed
5:
6: **repeat**
7:     $S \leftarrow$ the start state
8:     Choose $A$ from $S$ using $Q$ via $\epsilon$-greedy method
9:     **repeat**
10:        Take action $A$ and observe reward $R$ and next state $S'$
11:        Choose $A'$ from $S'$ using $Q$ via $\epsilon$-greedy method
12:        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$    ▷ Use $Q(S',A')$ to update $Q(S,A)$
13:        $S \leftarrow S'$, $A \leftarrow A'$
14:    **until** $S$ is terminal
15:    ep $\leftarrow$ ep $+ 1$
16: **until** ep $=$ max_eps
17: **return** $Q(s,a)$    ▷ After sufficient number of episodes, $Q(s,a)$ converges to $q_*(s,a)$
```

Algorithm 9 Q-learning with TD Control

```
1: $Q(s,a) \leftarrow$ arbitrary value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
2: Set all action values of the terminal state to 0.    ▷ Ensure that the state value of the terminal state is always the highest
3: ep $\leftarrow 0$    ▷ keep track of the number of episodes experienced
4: max_eps    ▷ Maximum number of episodes allowed
5:
6: **repeat**
7:     $S \leftarrow$ the start state
8:     **repeat**
9:         Choose $A$ from $S$ using $Q$ via $\epsilon$-greedy method
10:        Take action $A$ and observe reward $R$ and next state $S'$
11:        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$    ▷ Use $\max_a Q(S',a)$ to update $Q(S,A)$
12:        $S \leftarrow S'$
13:    **until** $S$ is terminal
14:    ep $\leftarrow$ ep $+ 1$
15: **until** ep $=$ max_eps
16: **return** $Q(s,a)$    ▷ After sufficient number of episodes, $Q(s,a)$ converges to $q_*(s,a)$
```

the use of two-, three-, or arbitrarily many step look-ahead to update the current state or action value, then the update would be more accurate than relying only on a single step look-ahead. In turn, more accurate value update means the algorithm would converge to optimal faster. Fortunately, such forward view method exists and is realized using the backward view eligibility tracing method.

Briefly put, eligibility tracing is a weight value associated with each state. It determines how much of the error on the current state or action value estimate shall be propagated to all the other state. Apparently, the closer a state is to the current state or the more often a state is visited, the higher its eligibility tracing and the more influence the current error will have on it. All eligibility tracing attenuates as episode moves forward, but it gets a boost each time its associated state is visited. Eventually, only those nearby states and/or frequently visited states share the blame of the current state value's error.

The benefit of using eligibility tracing is that it allows more than one-step look-ahead when state or action values are updated. This speeds up the convergence of state or action values.

Finally, a new parameter is added: λ. In the forward view, λ determines the how much of the multi-step look-ahead result shall contribute to the current state or action value. In the backward view, λ serves as the attenuation tool for the eligibility tracing.

3.4.1 TD(λ)

TD(λ) is similar to TD(0), except that the former also includes a procedure to update all eligibility tracing. Another key difference is that TD(λ) uses eligibility tracing to control how much of the current error is propagated to all the other states. The pseudo-code is provided in Algorithm 10.

Algorithm 10 TD(λ)

```
1: $V(s) \leftarrow$ arbitrary value for all $s \in \mathcal{S}$
2: ep $\leftarrow 0$                         ▷ keep track of the number of episodes experienced
3: max_eps                                   ▷ Maximum number of episodes allowed
4:
5: repeat
6:     $E(s) \leftarrow 0$ for all $s \in \mathcal{S}$
7:     $S \leftarrow$ the start state
8:     repeat
9:         $A \leftarrow$ a random action at $S$
10:        Take action $A$ and observe reward $R$ and next state $S'$
11:        $\delta \leftarrow R + \gamma V(S') - V(S)$              ▷ current error
12:        $E(S) \leftarrow E(S) + 1$    ▷ Accumulating traces; it boosts eligibility tracing of the state being visited to the highest.
13:        for all $s \in \mathcal{S}$ do
14:            $V(s) \leftarrow V(s) + \alpha\delta E(s)$   ▷ State value of all states are updated, but the amount of update is attenuated by $E(s)$
15:            $E(s) \leftarrow \gamma\lambda E(s)$
16:        end for
17:        $S \leftarrow S'$
18:    until $S$ is terminal
19:    ep $\leftarrow$ ep $+ 1$
20: until ep = max_eps
21: return $V(s)$          ▷ After sufficient number of episodes, $V(s)$ converges to $v_*(s)$
```

3.4.2 Sarsa(λ)

Sarsa(λ) is similar to TD(λ), except the target of update is action value instead of state value. The pseudo-code is provided in Algorithm 11.

Algorithm 11 Sarsa(λ)

```
 1: $Q(s,a) \leftarrow$ arbitrary value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
 2: ep $\leftarrow 0$                                ▷ keep track of the number of episodes experienced
 3: max_eps                                         ▷ Maximum number of episodes allowed
 4:
 5: repeat
 6:     $E(s,a) \leftarrow 0$ for all $s \in \mathcal{S}$
 7:     $S \leftarrow$ the start state
 8:     $A \leftarrow$ random action at $S$
 9:     repeat
10:         Take action $A$ and observe reward $R$ and next state $S'$
11:         Choose $A'$ from $S'$ using $Q$ via $\epsilon$=greedy method
12:         $\delta \leftarrow R + \gamma Q(S',A') - Q(S,A)$                    ▷ current error
13:         $E(S,A) \leftarrow E(S,A) + 1$                                   ▷ Accumulating traces
14:         for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$ do
15:             $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$  ▷ Action values updated with attenuation $E(s,a)$
16:             $E(s,a) \leftarrow \gamma\lambda E(s,a)$
17:         end for
18:         $S \leftarrow S'$
19:         $A \leftarrow A'$
20:     until $S$ is terminal
21:     ep $\leftarrow$ ep + 1
22: until ep = max_eps
23: return $Q(s,a)$              ▷ After sufficient number of episodes, $Q(s,a)$ converges to $q_*(s,a)$
```

3.4.3 Watkins's Q(λ)

Watkins's Q(λ) is the application of multi-step look-ahead and eligibility tracing on Q-learning, where each action value update depends on the best next action value. The key point in Watkins's Q(λ) is how eligibility tracing is modified after each state-action pair visit. Note that by using the maximum next action value, it is guaranteed that the error thus generated is also maximum. If the current action taken is indeed the greedy action, then the maximum error is justified to be propagated to all states. In this situation, the eligibility tracing is modified in the same way as in Sarsa(λ). However, if the current action taken is not greedy, then it is not responsible for the maximum error. In this situation, eligibility tracing of all states is set to 0, such that future error wouldn't be blamed due to the current non-greedy action. The pseudo-code is provided in Algorithm 12.

4 Evaluation

All algorithms are evaluated on the grid world shown in Figure 1. To standardize the training procedure, all algorithms use the same parameters listed below.

- Reward $= -1$ (all state transition incurs a reward of -1)
- $\gamma = 0.9$ (discount rate)
- $\alpha = 0.1$ (step size or learning rate)
- $\epsilon = 0.2$ (for ϵ-greedy method)

Algorithm 12 Watkins's Q(λ)

```
 1: $Q(s,a) \leftarrow$ arbitrary value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
 2: ep $\leftarrow 0$                          ▷ keep track of the number of episodes experienced
 3: max_eps                                   ▷ Maximum number of episodes allowed
 4:
 5: repeat
 6:     $E(s,a) \leftarrow 0$ for all $s \in \mathcal{S}$
 7:     $S \leftarrow$ the start state
 8:     $A \leftarrow$ random action at $S$
 9:     repeat
10:         Take action $A$ and observe reward $R$ and next state $S'$
11:         Choose $A'$ from $S'$ using $Q$ via $\epsilon$=greedy method
12:         $A^* \leftarrow \arg\max_a Q(S',a)$
13:         if $Q(S',A') = Q(S',A^*)$ then            ▷ In case multiple actions are greedy
14:             $A^* \leftarrow A'$
15:         end if
16:         $\delta \leftarrow R + \gamma Q(S',A^*) - Q(S,A)$                ▷ current error
17:         $E(S,A) \leftarrow E(S,A) + 1$                          ▷ Accumulating traces
18:         for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$ do
19:             $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$  ▷ Action values updated with attenuation $E(s,a)$
20:             if $A' = A^*$ then
21:                 $E(s,a) \leftarrow \gamma\lambda E(s,a)$ ▷ Keep eligibility tracing alive if current action is greedy
22:             else
23:                 $E(s,a) \leftarrow 0$        ▷ If current action not greedy, not propagate future error
24:             end if
25:         end for
26:         $S \leftarrow S'$
27:         $A \leftarrow A'$
28:     until $S$ is terminal
29:     ep $\leftarrow$ ep $+ 1$
30: until ep = max_eps
31: return $Q(s,a)$          ▷ After sufficient number of episodes, $Q(s,a)$ converges to $q_*(s,a)$
```

- $\lambda = 0.8$ (for eligibility tracing)

Four figures are generated for each evaluation. The step-to-go curve (subplot A) shows the expected number of steps needed to reach the terminal state from the start at each episode. The number is computed by following the state values of each episode and always moving the agent towards the next state with the highest value. It is possible that with state values not yet converged, adjacent state values would form a local trap, e.g. state A points to state B but state B points back to state A. When this happens, we rely on a maximum step restriction to break out of the infinite loop. The restriction is set to 1,000 steps, which means if traversing state values leads to more than 1,000 steps, we consider these state values unable to help an agent reach the terminal state, or in other words, the state values haven't converged to optimal yet. On the step-to-go curve, the corresponding episode will then be assigned 1,000 steps. On the curve, there is also a green horizontal line, indicating the optimal number of steps for the grid world (optimal steps is 18).

The RMS error curve (subplot B) shows the mean squared error when comparing the state values of the current episode to the previous one. As the number of episode increases, the RMS error decreases, indicating that the state values are converging.

The path plot (subplot C) demonstrates the optimal path (shown in red) generated by the algorithm after training. The yellow cell is the start, green cell the terminal, and brown cells the blocks.

The state value heat-map (subplot D) illustrates the converged values for each state. The darker the color the more negative the state value. The complete black cells correspond to the blocks in the grid world.

4.1 Dynamic Programming

Table 1 summarizes the comparison of the three Dynamic Programming algorithms. Value Iteration is the best performing algorithm among the three, as it identifies the optimal solution and converges RMS error with the fewest number of episodes. In contrast, Iterative Policy Evaluation performs the worst.

All three algorithms experience tough time during the first few episodes, as the agent is not able to end at the terminal state using the state values generated thus far. But performance improves fairly quickly. And once the agent is able to terminate an episode, the performance does not degrade anymore and gradually converges to the optimal number of steps.

All three algorithms find the optimal steps before their error converges.

4.1.1 Implementation Gotcha

One implementation gotcha is that in Dynamic Programming, bootstrapping does not work. In other words, when a state value is updated by the next state value, the next state value cannot be obtained from the same episode. Instead, a separate copy of the state values of the previous episode must be used. This is because Dynamic Programming does not update state value as the state moves towards the terminal. Instead, it simply iterates through all states in an arbitrary manner (e.g. iterate the cells in the grid world row by row), which means the next state might have nothing to do with reaching the terminal state. If we use bootstrapping, wrong rewards will be accumulated and the state values will not converge. Therefore, using two sets of state values is necessary to avoid bootstrapping.

4.2 Monte Carlo

Table 2 summarizes the comparison of the three Monte Carlo algorithms. The number with stars on it means they are estimates due to the nature of Monte Carlo implementation. To be specific, Monte Carlo has a hard requirement that each episode must end at the terminal

Table 1: Dynamic Programming Algorithm Comparison Summary

ALGORITHM	**Iterative Policy Evaluation**	**Policy Iteration**	**Value Iteration**
TOTAL EPISODES	30	30	30
1ST EPISODE OPTIMAL	27	23	16
1ST EPISODE NO ERROR	Not Converged	26	18
FIGURE REFERENCE	2	3	4

state to be considered valid. However, during learning, it is not uncommon that some episodes fail to end at the terminal state within the 1,000-step restriction. Since these failed episodes provide no useful information to achieving the goal, they are NOT counted when generating the step-to-go and RMS error curves. In other words, the stared numbers in Table 2 are likely underestimation of the true values.

That said, using the stared values in "1ST EPISODE OPTIMAL" as guidance, we can still see that On-policy Control performs the best, whereas Exploring Starts the worst. This is not surprising as On-policy Control actively adjust its strategy per episode, yet First Visit Policy Evaluation relies completely on random action. The bad performance of Exploring Starts is unexpected, as it should perform better than First Visit Policy Evaluation, given that more states are likely to be visited earlier in training. However, the presence of blocks is a curse to Exploring Starts, because it is also more likely for a random start state to get stuck near obstacles, thus wasting an episode. Had the grid world not contained that many blocks, Exploring Starts should be able to perform better than First Visit Policy Evaluation.

Another thing worth noting is that both Exploring Starts and On-policy Control experience fluctuation even after optimal step has been found. For Exploring Starts in particular, the fluctuation happens even after the algorithm has "converged" to the optimal solution for almost 50 episodes. This indicates the fragility of Monte Carlo method when optimal solution is first encountered, and calls for extended episodes to confirm that the state values have indeed been converged.

Finally, just as Dynamic Programming, all three Monte Carlo algorithms experience non-ending episodes at the beginning of the training. And except for Exploring Starts, which understandably has very unstable RMS error (each episode is drastically different from the previous one due to random starting point), the other two algorithms find the optimal solution before the error converges.

4.2.1 Implementation Gotcha

There are two gotchas in the implementation of Monte Carlo. First, the restriction of 1,000 steps maximum for each episode is in place for Exploring Starts and On-policy Control. This is necessary because the action generation is not random enough to get the agent out of a local trap. In contrast, the restriction is not needed on First Visit Policy Evaluation, because all of its actions are randomly generated, and hence has no risk of the agent getting stuck.

Second, once the 1,000-step restriction is met in an episode, the episode is discarded completely. This is because an episode not ending in a terminal state is not helpful to Monte Carlo, as it might assign rewards not related to reaching the goal to state values. At best, this would delay arrival at the optimal solution. At worst, it can get the agent stuck in a local trap.

Table 2: Monte Carlo Algorithm Comparison Summary

ALGORITHM	**First Visit Policy Evaluation**	**Exploring Starts**	**On-policy Control**
TOTAL EPISODES	100	600	100
1ST EPISODE OPTIMAL	37*	411*	5*
1ST EPISODE NO ERROR	Not Converged	6*	89*
FIGURE REFERENCE	5	6	7

4.3 Temporal Difference

Table 3 summarizes the comparison of the three Temporal Difference algorithms. Sarsa with TD Control and Q-learning with TD Control both perform much better than TD(0). However, Sarsa with TD Control has much more fluctuation, so Q-learning with TD Control shall be considered the best performing algorithm. TD(0) performs the worst because all of its action generation is random, whereas the other two methods rely on prior information of state or action values to generate better actions.

4.3.1 Implementation Gotcha

There is no gotcha when implementing Temporal Difference. The only thing worth noting is that a 1,000-step restriction per episode is not necessary, because all Temporal Difference algorithms can generate stochastic actions to avoid the agent getting stuck in a local trap. For TD(0), random action is default. For Sarsa and Q-learning with TD Control, the ϵ-greedy method guarantees certain level of exploration.

Table 3: Temporal Difference Algorithm Comparison Summary

ALGORITHM	**TD(0)**	**Sarsa with TD Control**	**Q-learning with TD Control**
TOTAL EPISODES	100	200	100
1ST EPISODE OPTIMAL	46	5	7
1ST EPISODE NO ERROR	Not Converged	Not Converged	Not Converged
FIGURE REFERENCE	8	9	10

4.4 Eligibility Tracing

Table 4 summarizes the comparison of the three Eligibility Tracing algorithms. Similar to Temporal Difference, Sarsa(λ) and Watkins's Q(λ) perform better than TD(λ) due to the use of the updated state or action values to generate the next action. Between Sarsa(λ) and Watkins's Q(λ), there is little difference.

4.4.1 Implementation Gotcha

There is no implementation gotcha in Eligibility Tracing. And similar to Temporal Difference, the 1,000-step restriction per episode is not needed.

Table 4: Eligibility Tracing Algorithm Comparison Summary

ALGORITHM	**TD(λ)**	**Sarsa(λ)**	**Watkins's Q(λ)**
TOTAL EPISODES	100	100	100
1ST EPISODE OPTIMAL	25	7	7
1ST EPISODE NO ERROR	Not Converged	Not Converged	Not Converged
FIGURE REFERENCE	11	12	13

5 Discussion And Conclusion

In this report, we have implemented and compared 12 tabular method algorithms to solve the *Bellman Optimality Equation* for state or action values. Comparison of the algorithms within each group has been discussed in Section 4, so here we will say a few more words regarding the comparison across different groups. It is apparent that Monte Carlo has the least stable performance, due to its hard requirement on each episode ending at the terminal state. The fragility of its algorithms can be seen in the fluctuations of the step-to-go curves. Dynamic Programming has decent performs and never gets an agent stuck in a local trap. This is mainly due to the manageable size of the grid world. Had we used a 1000×1000 grid world, Dynamic Programming is not likely to find the optimal solution in tolerable period of time. The best performance is achieved in Temporal Difference and Eligibility Tracing. Both are based on the same concept, and Temporal Difference is just a special case in Eligibility Tracing where we only perform a 1-step look-ahead. The advantage of involving Control in the learning process is obvious in Temporal Difference and Eligibility Tracing, as the Control algorithms outperforms their purely random counterparts.

It is worth mentioning again that most of the algorithms identify the optimal solution way before the error has converged. This means when using tabular method in real application, it is not necessary to wait for convergence of the error. Once an optimal solution has been found, the learning can be terminated. That said, the fluctuation in step-to-go curves cautions that it is good practice not relying on the first few instances of seemingly optimal solution, but wait a bit longer until convergence can be confirmed. However, how long a wait is sufficient to confirm that optimal solution has been converged is an open question to us.

Finally, when fluctuation happens to step-to-go curves, it always goes to the extremes —either optimal or stuck in a local trap. This suggests that before convergence, the state or action values are quite fragile. Small disturbance could turn state or action values from optimal into failed.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning, An Introduction.* Cambridge, Massachusetts: The MIT Press, 2018.

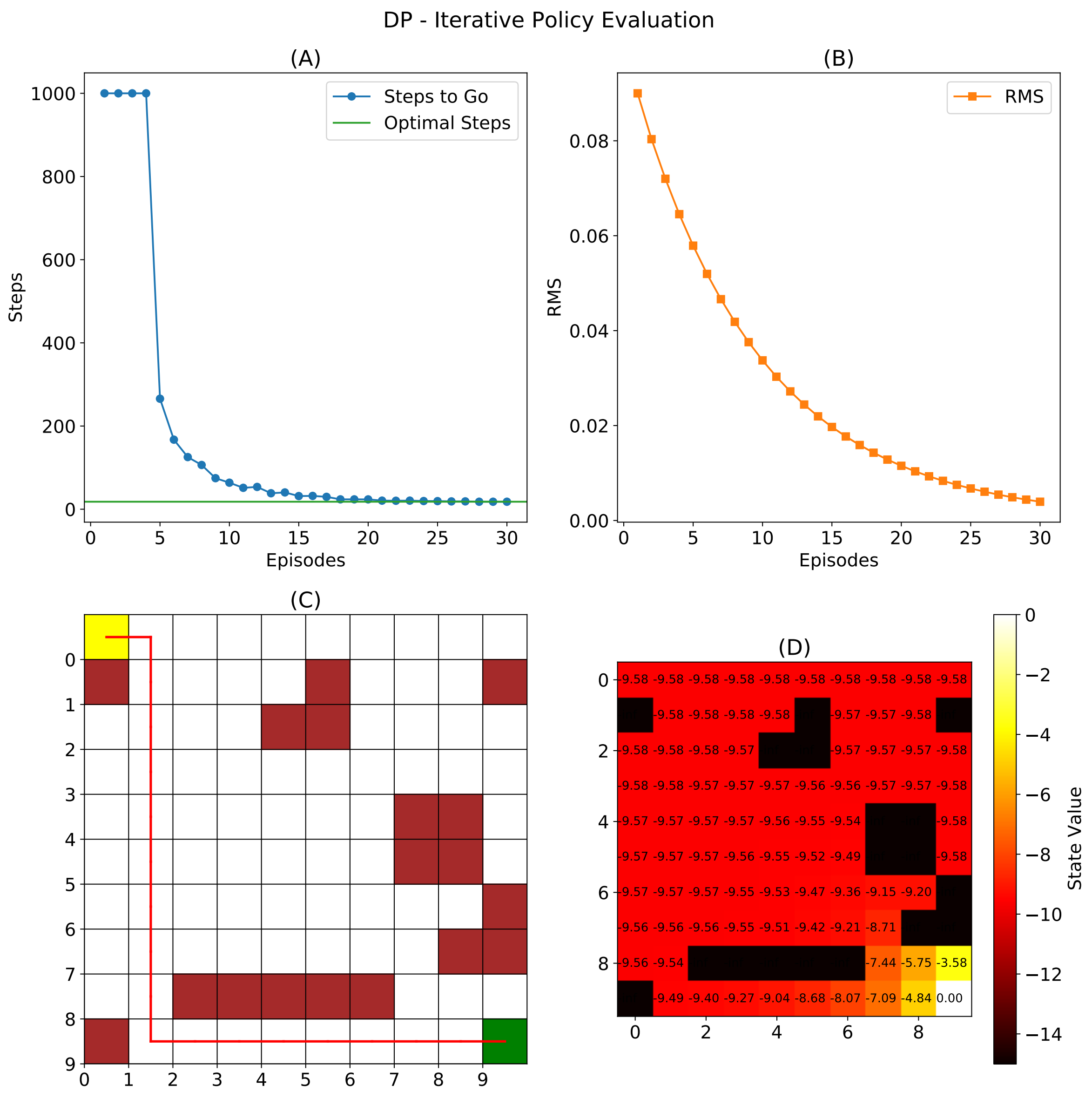


Figure 2: Dynamic Programming Iterative Policy Evaluation

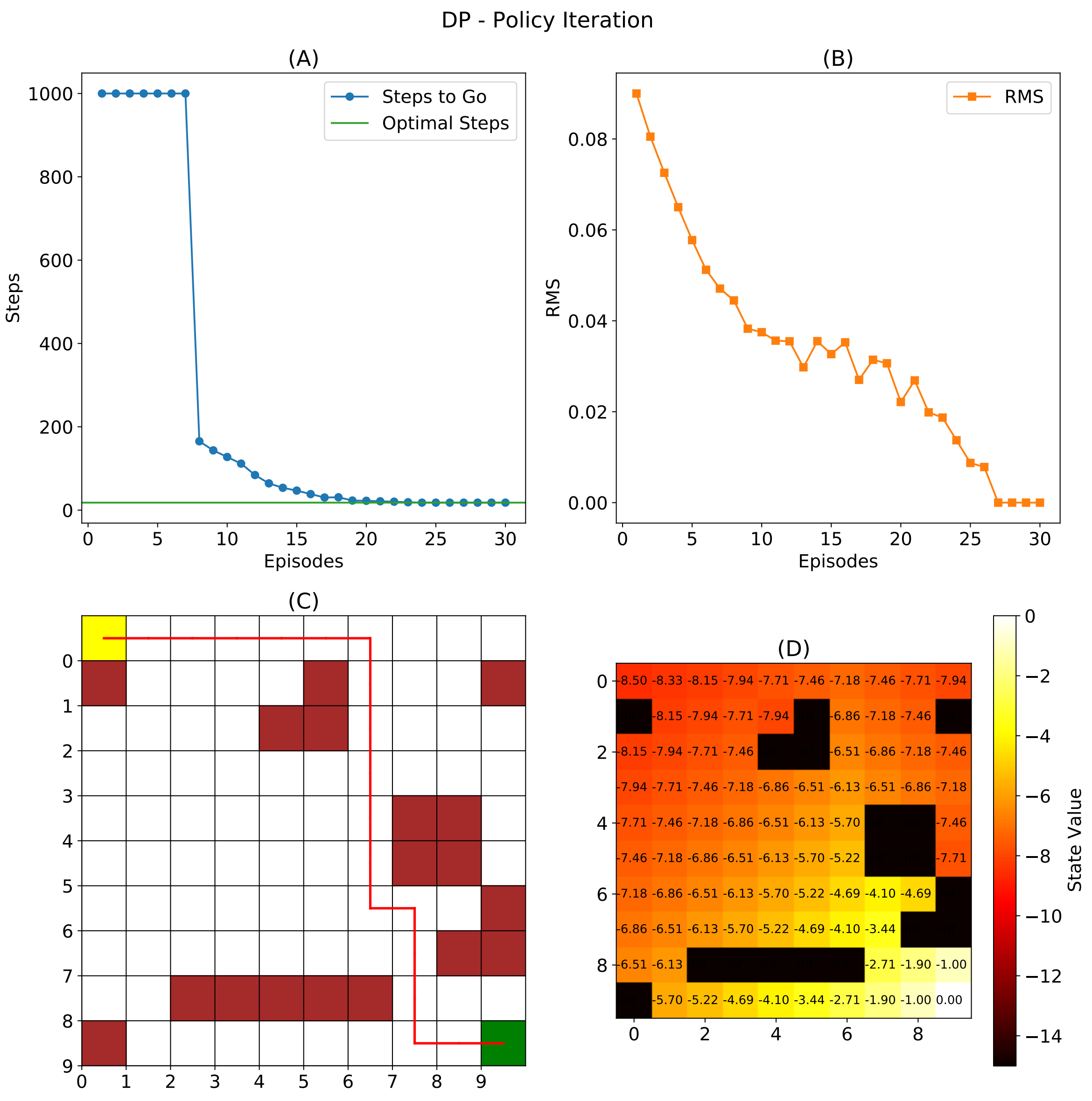


Figure 3: Dynamic Programming Policy Iteration

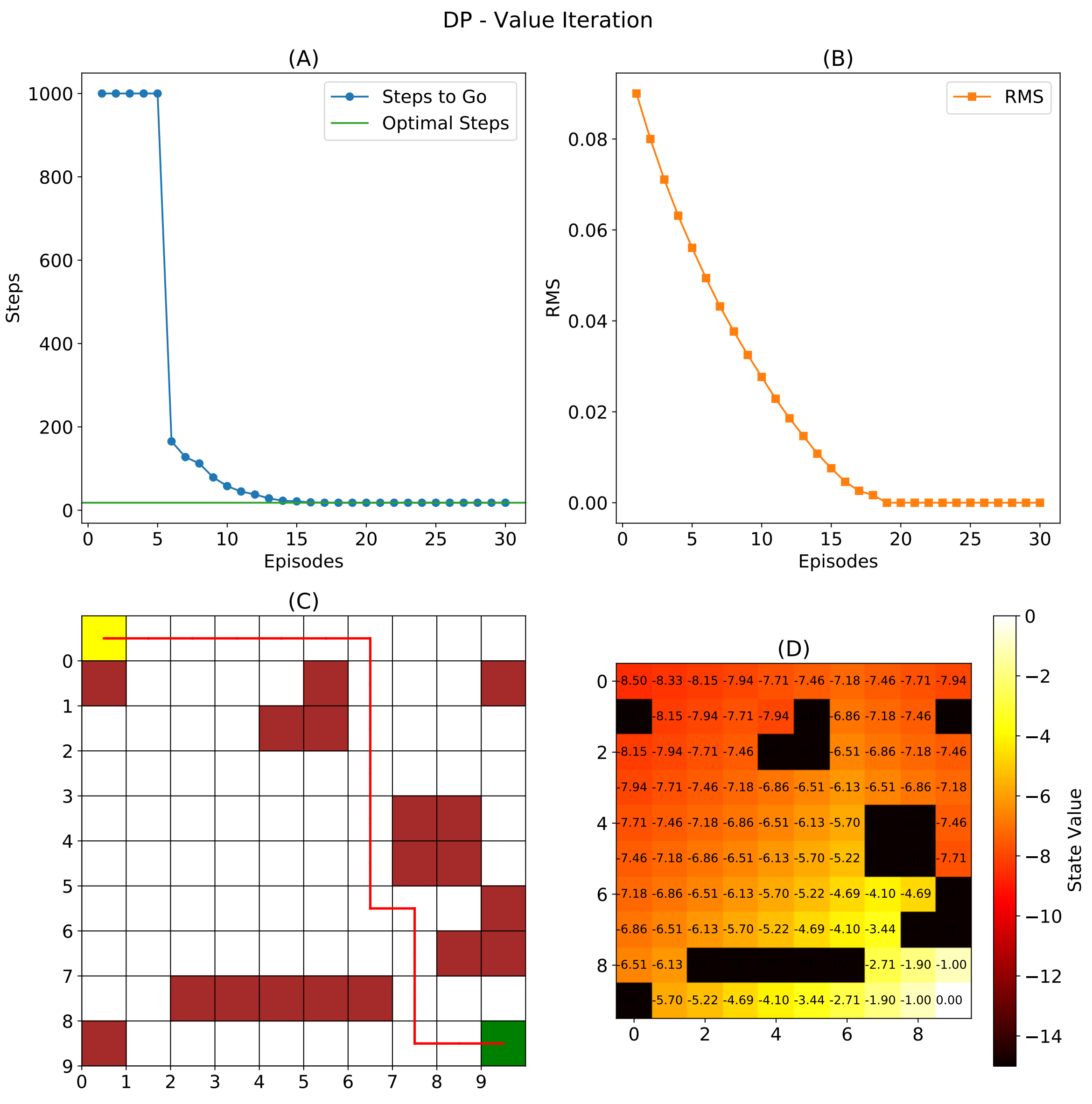


Figure 4: Dynamic Programming Value Iteration

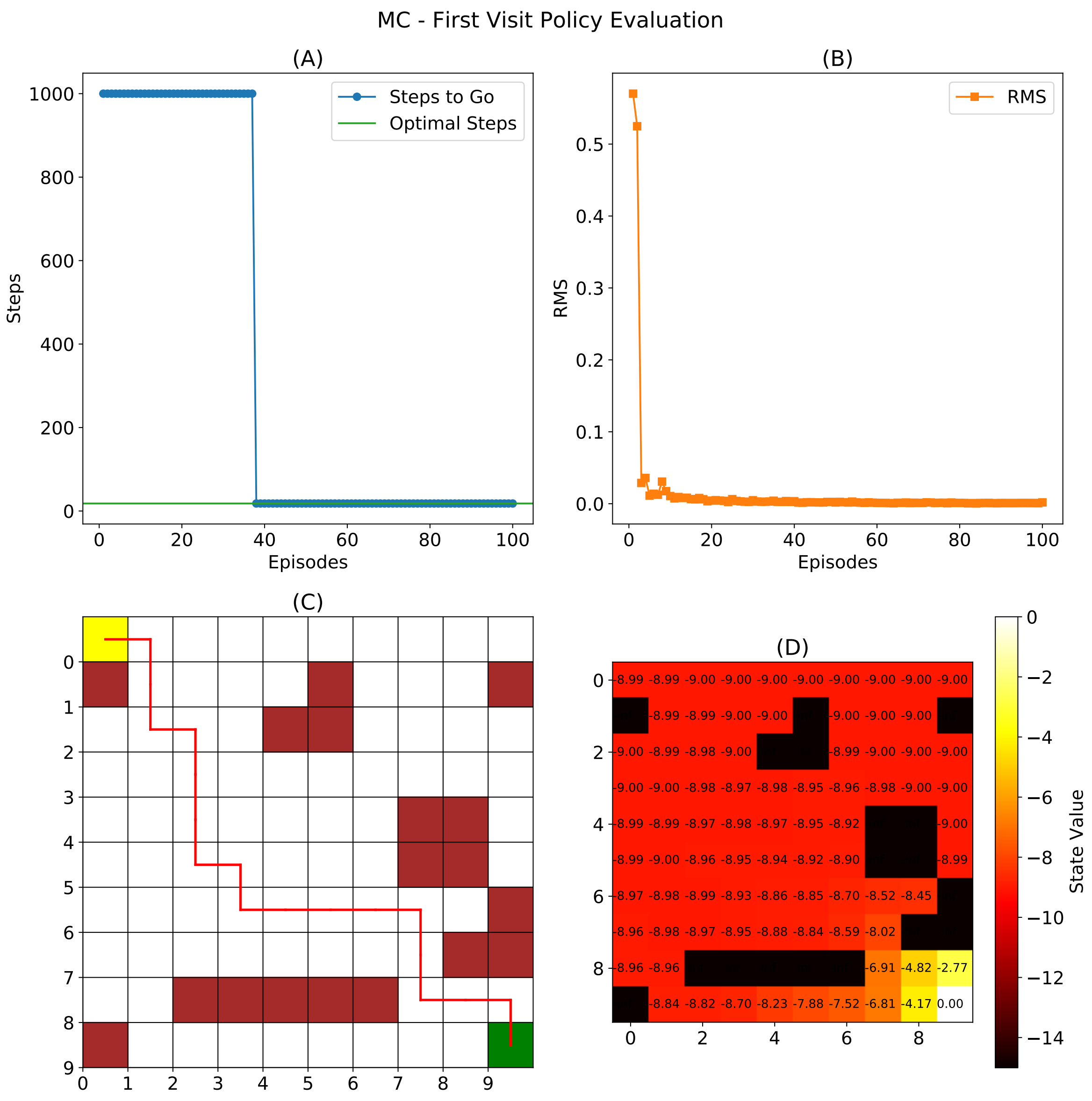


Figure 5: Monte Carlo First Visit Policy Evaluation

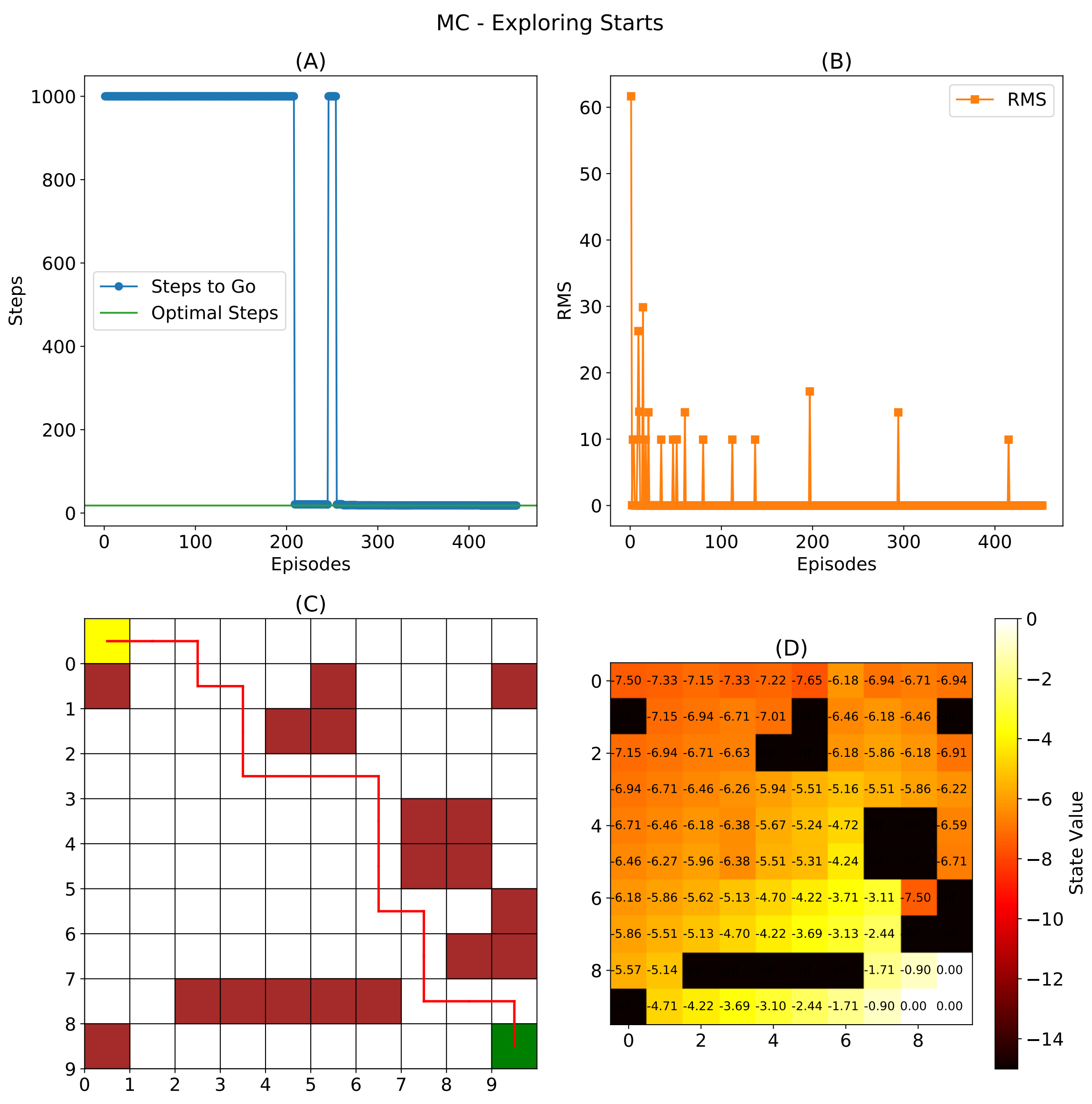


Figure 6: Monte Carlo Exploring Starts

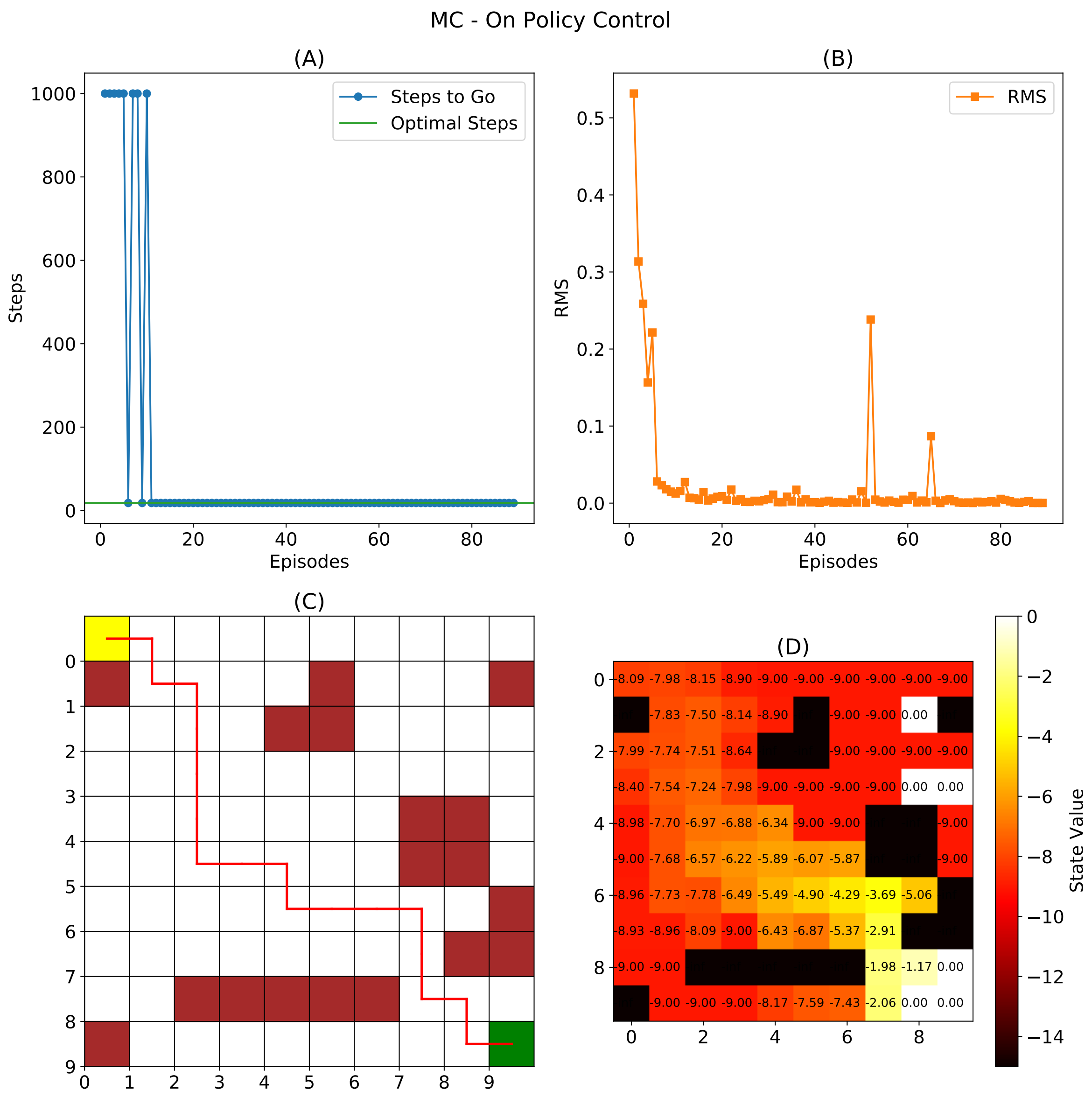


Figure 7: Monte Carlo On Policy Control

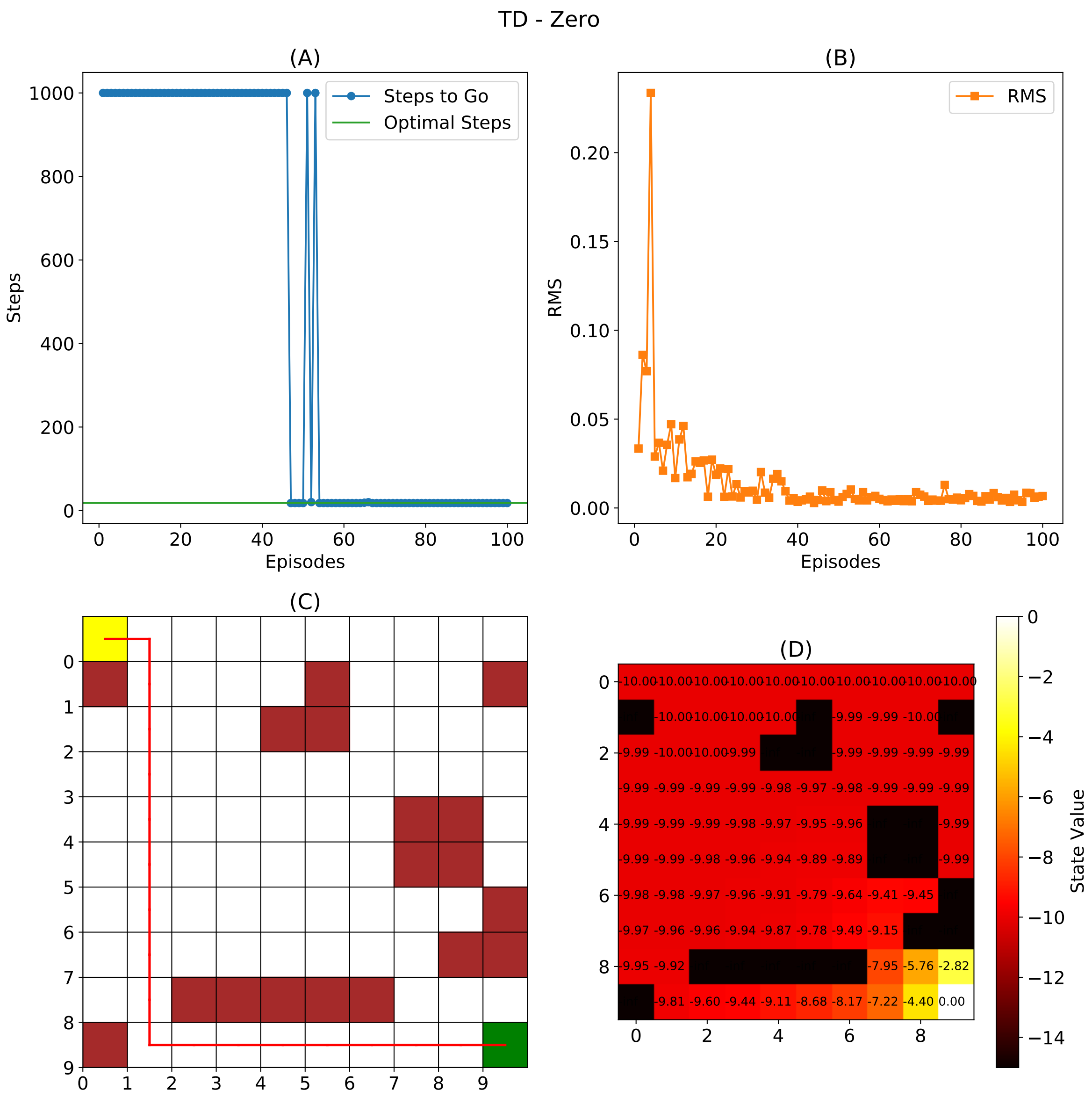

TD - Zero
(A)
Steps to Go
Optimal Steps
Steps
Episodes
(B)
RMS
Episodes
(C)
(D)
State Value

Figure 8: TD(0)

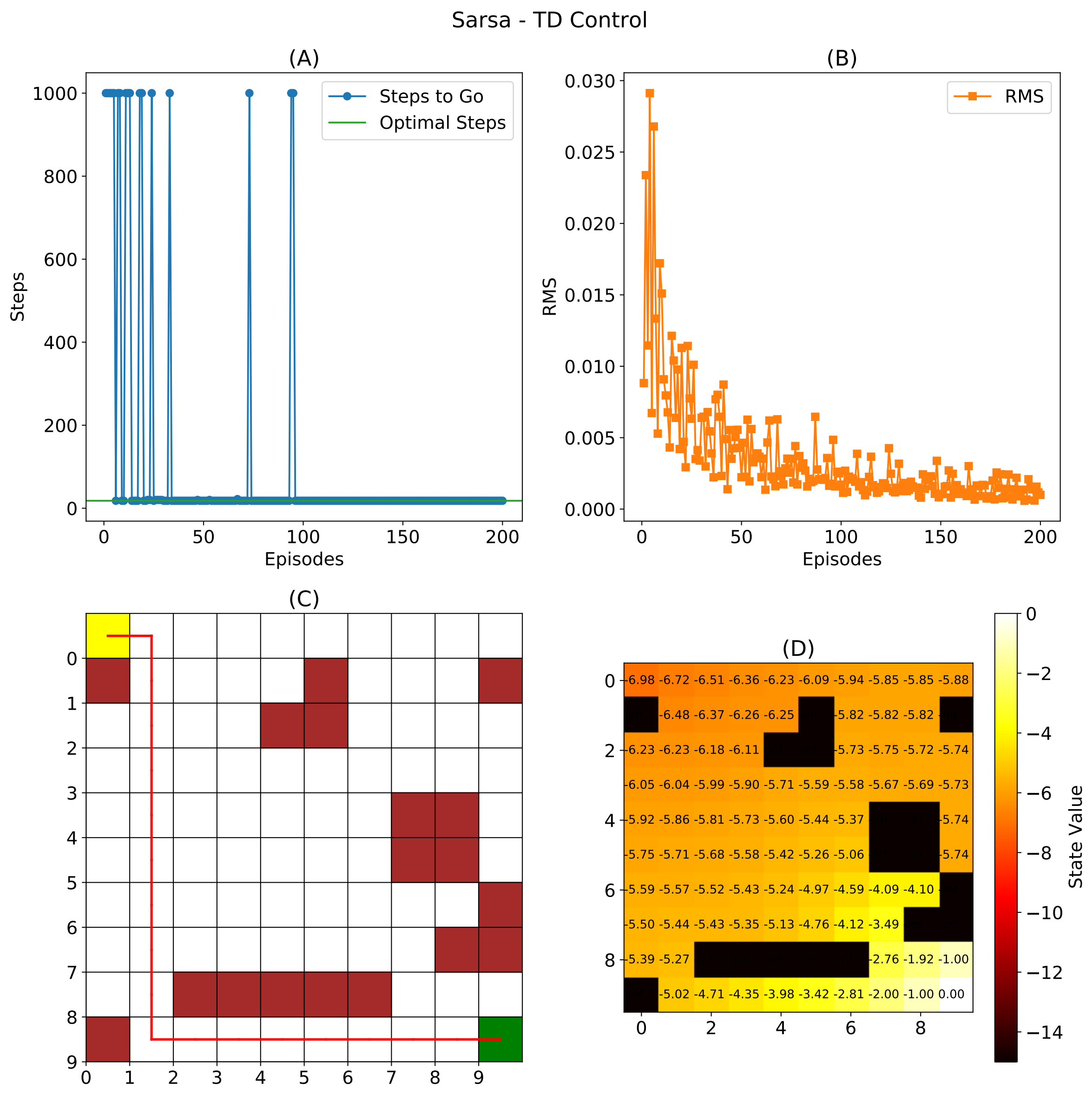


Figure 9: Sarsa - TD Control

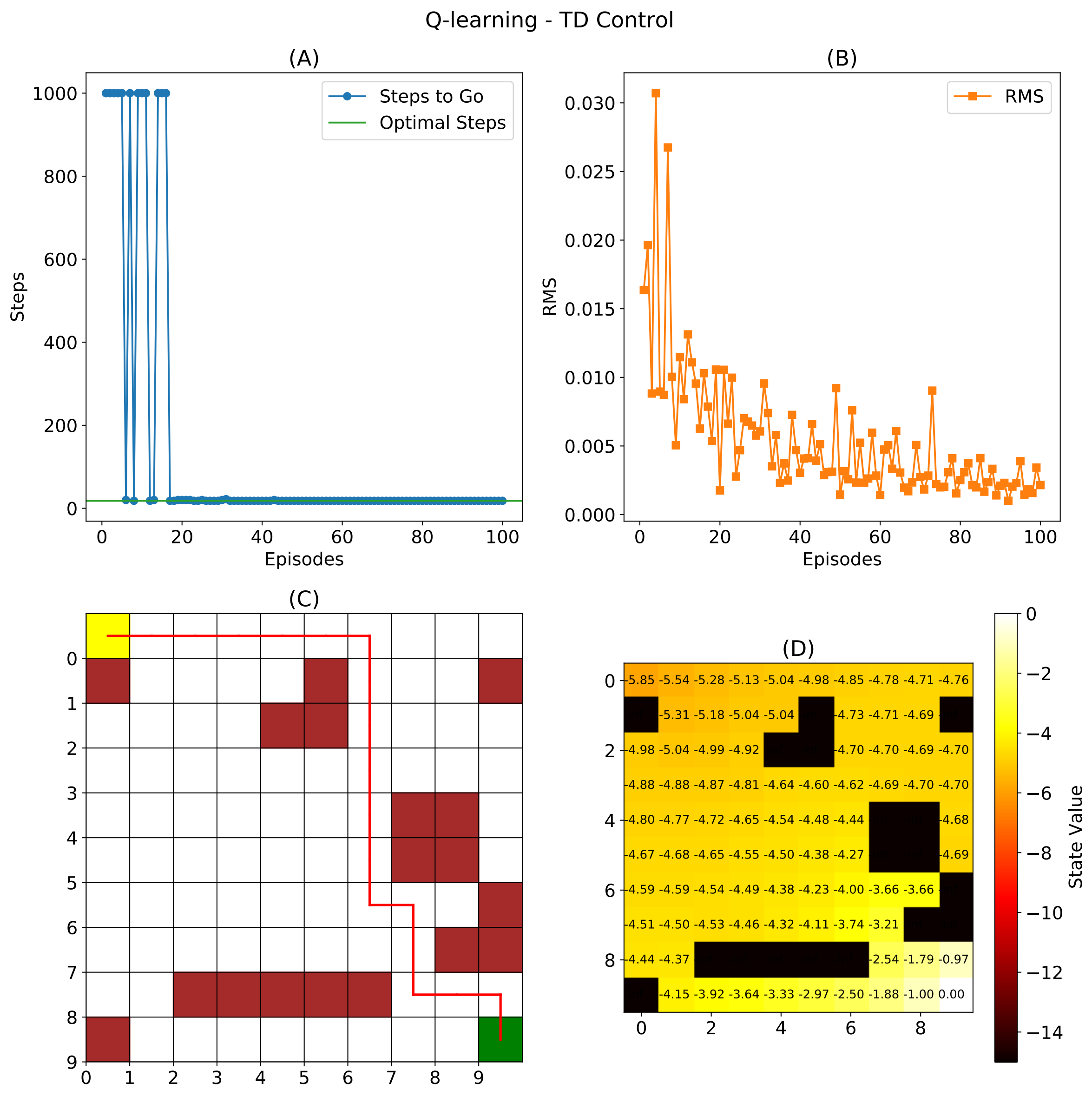


Figure 10: Monte Carlo On Policy Control

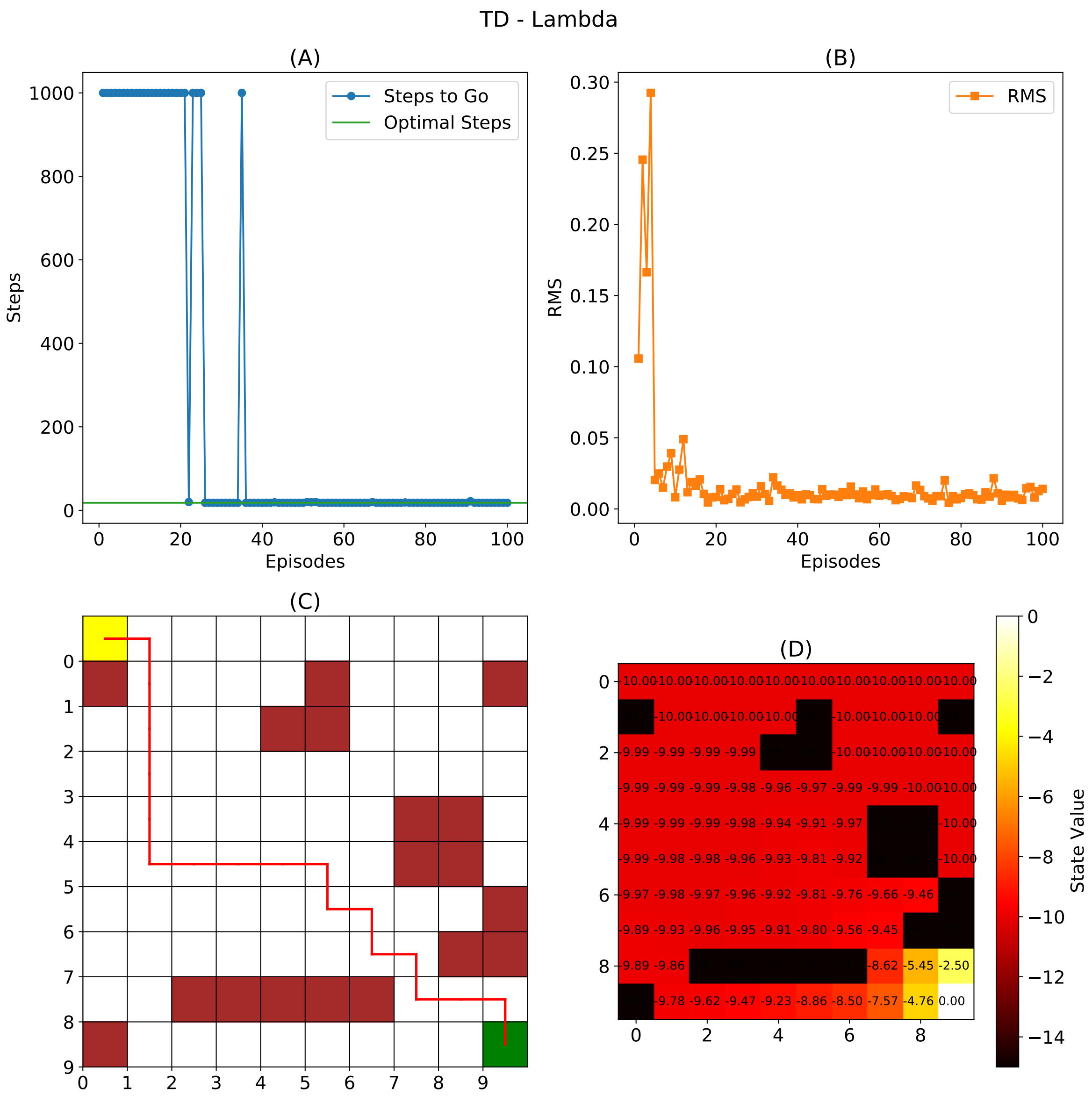


Figure 11: TD(λ)

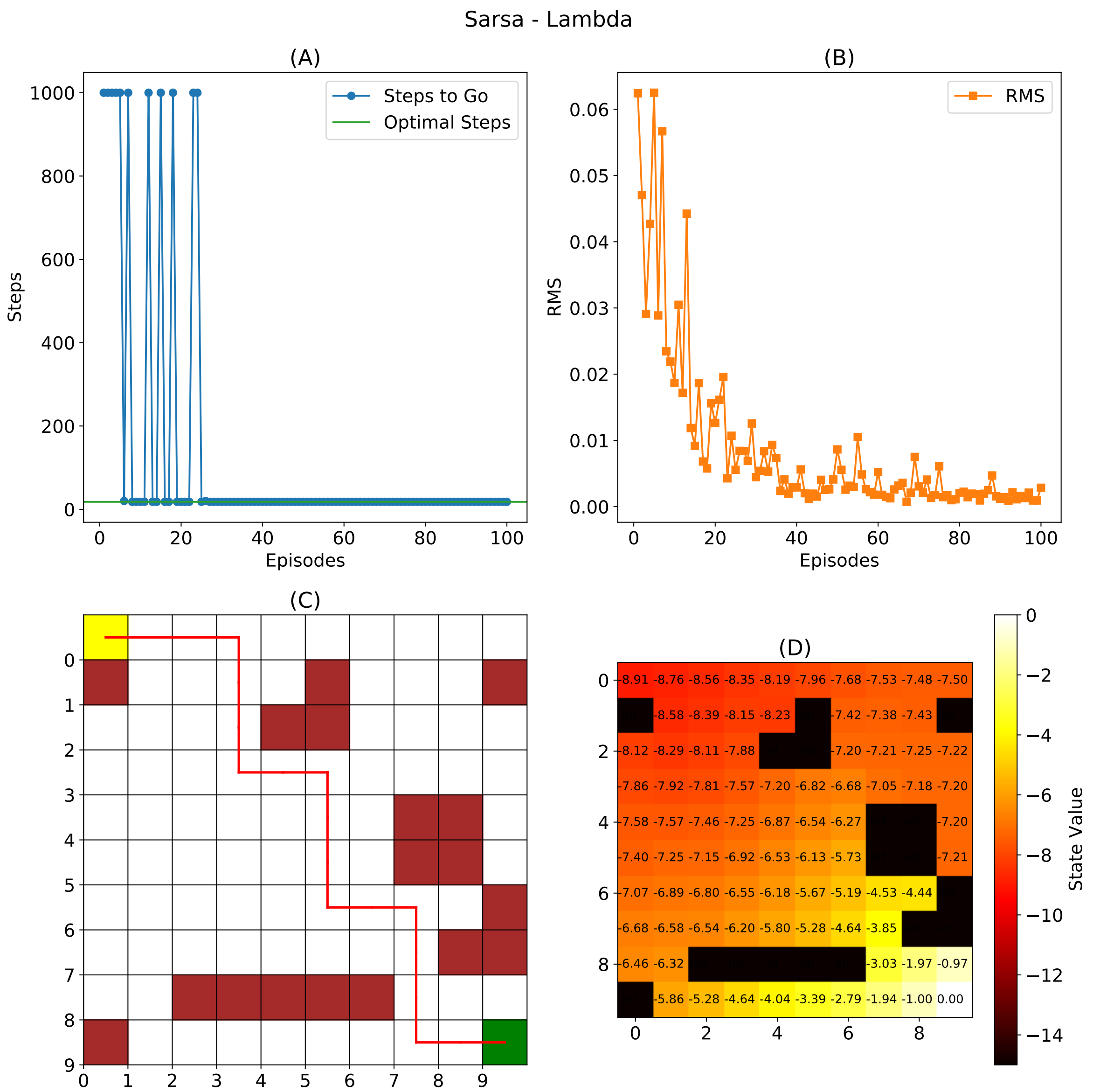


Figure 12: Sarsa(λ)

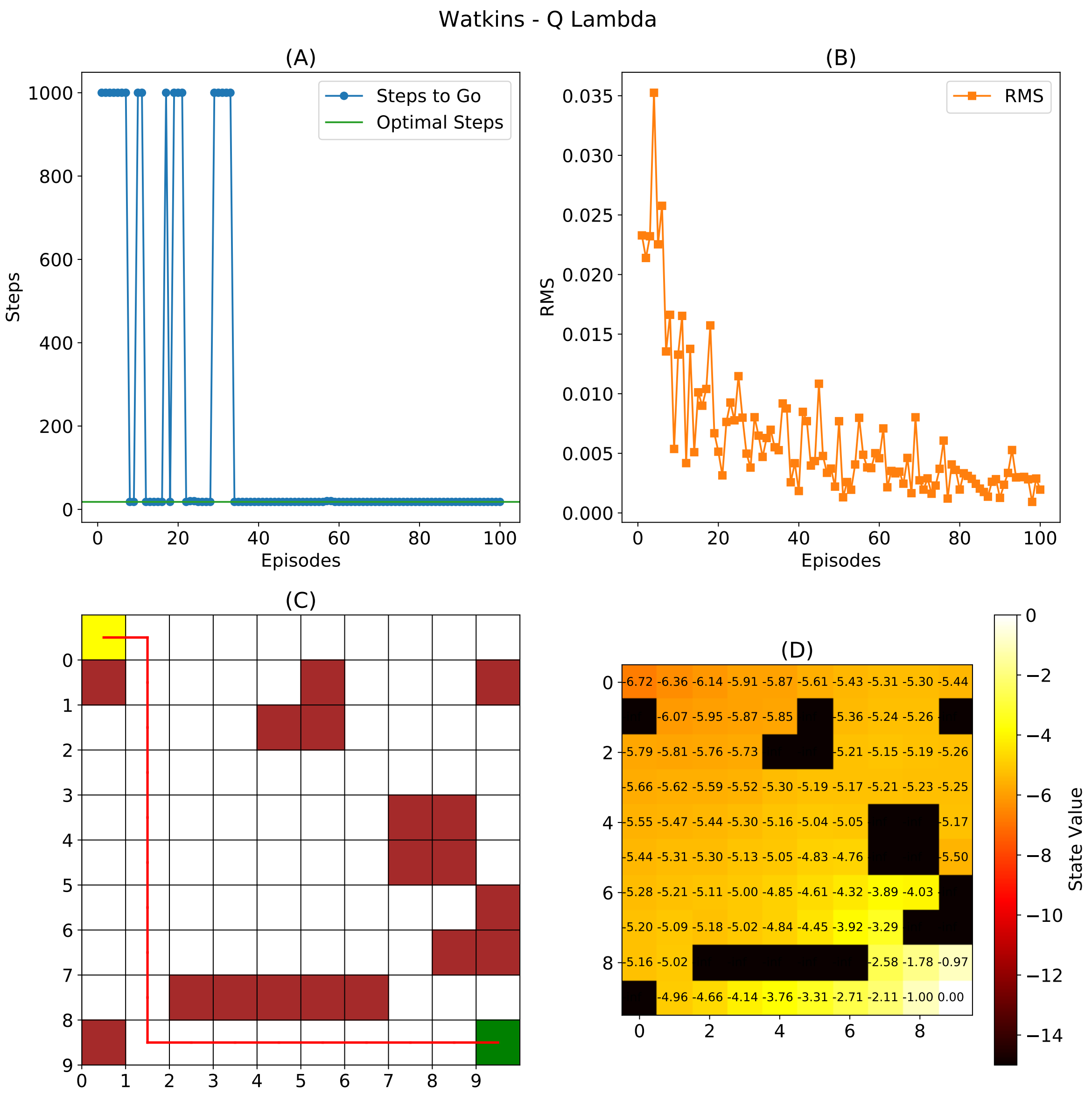


Figure 13: Watkins's Q(λ)

p2_Bao

March 10, 2021

[1089]:
```python
# Ensure that the root directory is in Python's path. This is to make importing
# custom library easier.
from pathlib import Path
import sys
root = Path('.').absolute().parent.parent
if str(root) not in sys.path:
    sys.path.append(str(root))

# built-in packages
from collections import defaultdict
import json
from typing import List, Dict, Tuple, Callable
import math

# external packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import scipy
import dill

# matplotlib param
# https://stackoverflow.com/a/55188780/9723036
# SMALL_SIZE = 10
MEDIUM_SIZE = 15
BIGGER_SIZE = 17
TEXT_BOTTOM = 0.9

plt.rc('font', size=MEDIUM_SIZE)          # controls default text sizes
# plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=MEDIUM_SIZE)    # legend fontsize
# plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title
```

```
np.set_printoptions(formatter={'float': lambda x: "{0:0.10f}".format(x)})
```

1 Initialize Parameters

[1136]:
```
ACTIONS = [[-1, 0], [0, 1], [1, 0], [0, -1]]  # N, W, S, E
M = 10  # number of rows
N = 10  # number of columns
NUM_BLK = 15  # number of blocks
GAMMA = 0.9  # discount rate
ALPHA = 0.1  # learning rate
OPTIMAL_STEPS = 18  # empirically acquired
EPSILON = 0.2  # for epsilon-greedy method
LAMBDA = 0.8  # for eligibility tracing
```

2 Shared Functions

[1092]:
```
def initialize_grid(num_rows: int, num_cols: int, num_blocks: int, random_state:
↪ int = 42):
    """Initialize a grid world, with obstacles.

    :param num_rows: Number of rows in the grid world.
    :param num_cols: Number of columns in the grid world.
    :param num_blocks: Number of RANDOM blocks that agent cannot cross. The␣
↪final grid world
        could include more blocks as additional  a manually
    :param random_state: For reproducing random values.
    :return: Grid for state values, action values, and the coordinates for␣
↪blocks.
    """
    # Create a grid
    grid = np.array([[0] * num_cols for _ in range(num_rows)])
    V = np.array([[0.0] * num_cols for _ in range(num_rows)])  # initial state␣
↪values
    Q = np.array([[[0.0] * len(ACTIONS) for _ in range(num_cols)] for _ in␣
↪range(num_rows)])  # initial action values
    # randomly generate blocks
    rng = np.random.default_rng(random_state)
    block_i = rng.integers(low=0, high=num_rows, size=num_blocks)
    block_j = rng.integers(low=0, high=num_cols, size=num_blocks)

    ####################################################
    # hardcodede additional blocks, such that we can   #
    # limit the possible number of ways to reach the   #
    # terminal state                                   #
    ####################################################
    block_i = np.append(block_i, [8, 8, 8])
    block_j = np.append(block_j, [4, 5, 6])

    # Update grid and state values by identifying the blocks
    grid[block_i, block_j] = 1
    V[block_i, block_j] = -1000.0

    # Update state action values to set the illegal actions to a big negative␣
↪value
    for i in range(num_rows):
        for j in range(num_cols):
            if i == num_rows - 1 and j == num_cols - 1:  # not touching the␣
↪terminal state
                continue
            if grid[i, j] == 1:  # the blocks
                Q[i, j] = np.full(4, -1000.0)
                continue
            for k, (di, dj) in enumerate(ACTIONS):
                ni, nj = i + di, j + dj
                if not (0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0):
                    Q[i, j, k] = -1000.0

    # initialize random policy
    pi = np.array([[-1] * M for _ in range(N)])
    for i in range(M):
        for j in range(N):
            if (i == M - 1 and j == N - 1) or grid[i][j]:
                continue  # skip terminal state and blocks
            pot_actions = []
            for idx, (di, dj) in enumerate(ACTIONS):
                ni, nj = i + di, j + dj
                if 0 <= ni < M and 0 <= nj < N and grid[ni, nj] == 0:
                    pot_actions.append(idx)
            if pot_actions:
                pi[i, j] = rng.choice(pot_actions)

    return np.copy(V), np.copy(Q), grid, np.copy(pi)


def get_next_state_vals(i: int, j: int, V, grid, fv: Callable = lambda v: v):
    """Given the current state i, j, return its next state values.

    To be more specific, we return the state values of all possible states
    that a next action can lead to, based on the given V. The order of
    the next state values is the same as ACTIONS.

    :param i: Row index.
    :parma j: Column index.
    :param V: State values.
    :param grid: Grid.
    :param fv: A lambda function that takes state value as argument and returns␣
↪a
        new value used for next state values. Default to a lambda that returns␣
↪the
        input state value (i.e., output and input is the same)
    :return: A numpy array of shape (4, ) that contains the next state
        values. If an action leads to a invalid state, such as out of bound
        or hitting a block, its value is set to negative infinite.
    """
    next_state_values = []  # store all Next State ValueS
    for di, dj in ACTIONS:
        ni, nj = i + di, j + dj
        if 0 <= ni < M and 0 <= nj < N and grid[ni, nj] == 0:
            next_state_values.append(fv(V[ni, nj]))
        else:
            next_state_values.append(-math.inf)
    return np.array(next_state_values)


def find_best_actions(next_state_values) -> List[int]:
    """Find the best action for an agent based on the given next state values

    Note that there could be multiple best actions if multiple next state values
    are maximum.

    :param next_state_values: A list of state values corresponding to the next
        state achieved from taking the actions in ACTIONS. The order of this
        list of state values correspond to the order in ACTIONS.
    :return: A list of indices correspinding to the best action to take.
    """
    max_V = max(next_state_values)
    sorted_indices = np.argsort(np.array(next_state_values))
    # find all the potential action indices. This is to handle the situation
    # where multiple actions lead to the same next state value. When that
    # happens, we randomly pick an action to go.
    return [idx for idx in sorted_indices if np.isclose(next_state_values[idx],␣
↪max_V)]


def steps_to_go(V, grid, random_state: int = 42) -> int:
    """Compute on average the number of steps needed to reach terminal state␣
↪from start.
```

```
    Tie breaks on state value is a random choice. We also have a high bound on␣
↪the number
    of steps to take. If the path search results in steps larger than the high␣
↪bound, it
    is very likely that the state values are incomplete and contain loop. In␣
↪that case,
    we exist the path search and directly assign the high bound as the number␣
↪of steps.

    :param V: State values of the grid world.
    :param grid: The grid world.
    :param random_state: For reproducing random values.
    :return: Average number of steps to reach terminal state.
    """
    max_step_allowed = 1000  # if steps count go beyond this value, we␣
↪terminate the search
    m, n = V.shape
    rng = np.random.default_rng(random_state)
    total_steps = 0
    total_iteration = 50  # walk the agent 50 times, take the average steps at␣
↪the end
    for _ in range(total_iteration):
        i, j = 0, 0
        cur_steps = 0
        deterministic = True  # flag to indicate whether a deterministic path␣
↪can be found
        while (i != m - 1 or j != n - 1) and cur_steps < max_step_allowed:
            nsvs = get_next_state_vals(i, j, V, grid)
            best_acts = find_best_actions(nsvs)
            if len(best_acts) > 1:
                deterministic = False
            next_idx = rng.choice(best_acts)
            i, j = i + ACTIONS[next_idx][0], j + ACTIONS[next_idx][1]
            cur_steps += 1
        if deterministic:  # already deterministic, we can end already
            return cur_steps
        if cur_steps == max_step_allowed:  # If V hasn't converged, return high␣
↪bound directy
            return cur_steps
        total_steps += cur_steps
    return total_steps / total_iteration


def update_policy(V, pi, grid):
    """Update the current policy pi given new state values V
    :param V: Current state values.
    :param pi: previous policy.
    :param grid: The grid world.
    """
    m, n = pi.shape
    for i in range(m):
        for j in range(n):
            if pi[i, j] >= 0:
                nsvs = get_next_state_vals(i, j, V, grid)
                if nsvs[pi[i, j]] < max(nsvs):  # current policy is not optimal
                    pi[i, j] = np.argmax(nsvs)


def get_epsilon_greedy_action(Q, i: int, j: int, grid) -> int:
    """Use epsilon-greedy method to generate an action based on state, action␣
 ↪pair.

    With epsilon probablity, we exploit. With 1 - epsilon probability, we␣
 ↪explore.

    :param Q: state, active values.
    :param i: Row index of the state.
    :param j: Column index of the state.
    :param grid: The grid world.
    """
    while True:  # choose a legal action based on epsilon-greedy
        method = rng.choice(['explore', 'exploit'], p=[EPSILON, 1 - EPSILON])
        if method == 'explore':
            act = rng.choice(len(ACTIONS))
        else:
            best_acts = find_best_actions(Q[i, j])
            act = rng.choice(best_acts)
        di, dj = ACTIONS[act]
        ni, nj = i + di, j + dj
        if 0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0:
            return act  # valid action
```

3 Plot Related

[1123]:
```
def plot_step_to_go(ax, steps, optimal_steps: int, title: str):
    """Plot step-to-go vs. episode

    :param ax: An axis object of matplotlib.
    :param steps: Number of expected steps to reach the target for each episode.
    :param optimal_steps: The optimal number of steps to reach the target.
    :param title: Title for the step-to-go plot
    """
    ax.plot(np.arange(1, len(steps) + 1), steps, marker='o', color='C0',␣
↪label='Steps to Go')
    ax.axhline(optimal_steps, color='C2', label='Optimal Steps')
    ax.set_xlabel('Episodes')
    ax.set_ylabel('Steps')
    ax.set_title(title)
    ax.legend()


def plot_rms_error(ax, rms, title: str):
    """Plot RMS error vs. episode

    :param ax: An axis object of matplotlib.
    :param rms: The rms error for each episode.
    :param title: Title for the step-to-go plot
    """
    ax.plot(np.arange(1, len(rms) + 1), rms, marker='s', color='C1',␣
↪label='RMS')
    ax.set_xlabel('Episodes')
    ax.set_ylabel('RMS')
    ax.set_title(title)
    ax.legend()


def plot_state_value_heatmap(ax, V, fig, title: str, color_vmin: int = -15,␣
↪color_vmax: int = 0):
    """Plot state value as a heatmap.

    :param ax: An axis object of matplotlib.
    :param V: The final estimated state values.
    :param fig: The fig parameter, acquired when calling plt.subplots().
    :param title: Title for the step-to-go plot.
    :param color_vmin: Min value for the color map indicator, default to -15.
    :param color_vmax: Max value for the color map indicator, default to 0.
    """
    hm = ax.imshow(
        V,
        cmap='hot',
        interpolation='nearest',
        vmin=-15,  # colorbar min
        vmax=0,  # colorbar max
    )
    cbar = fig.colorbar(hm)
    cbar.set_label('State Value')
    # add V value to each heatmap cell
    m, n = V.shape
    for i in range(m):
        for j in range(n):
            val = '-inf' if V[i, j] <= -1000.0 else f'{V[i, j]:.2f}'
            ax.annotate(val, xy=(j - 0.5, i + 0.1), fontsize=10)
    ax.set_title(title)


def plot_path(ax, pi, grid, title: str):
    """Plot a route from start to end on the grid world.

    Yellow is start; green is end; brown is obstacle.

    :param ax: An axis object of matplotlib.
    :parma pi: The final policy.
    :param grid: The grid world.
    :param title: Title for the step-to-go plot
    """
    m, n = pi.shape
    ax.grid(True)
    ax.set_xticks(np.arange(n))
    ax.set_xlim(left=0, right=n)
    ax.set_yticks(np.arange(m))
    ax.set_ylim(top=m - 1, bottom=-1)
    ax.invert_yaxis()  # invert y axis such that 0 is at the top
    i = j = 0
    while i != m - 1 or j != n - 1:
        di, dj = ACTIONS[pi[i, j]]
        ni, nj = i + di, j + dj
        ax.plot([j + 0.5, nj + 0.5], [i - 0.5, ni - 0.5], color='red', lw=2)
        i, j = ni, nj
    # label start, end and blocks
    ax.add_patch(Rectangle((0, -1), 1, 1, fill=True, color='yellow'))
    ax.add_patch(Rectangle((n - 1, m - 2), 1, 1, fill=True, color='green'))
    for i, j in zip(*np.where(grid == 1)):
        ax.add_patch(Rectangle((j, i - 1), 1, 1, fill=True, color='brown'))
    ax.set_title(title)


def plot(steps, rms, V, pi, optimal_steps: int, title: str):
    """Plot the training data and results.

    This function plots four graphs.
    1. Step-to-go vs. episode
    2. RMS error vs. episode
    3. State value as a heatmap
    4. One route found by the reinforcement learning to reach destination.
    :param steps: Number of expected steps to reach the target for each episode.
    :param rms: The rms error for each episode.
    :param V: The final estimated state values.
    :parma pi: The final policy
    :param optimal_steps: The optimal number of steps to reach the target.
    :param title: The title for the plot, also serving as the filename of the␣
↪saved
        plot file.
    """
    plt.rc('grid', linestyle="-", color='black')
    fig, axes = plt.subplots(2, 2, figsize=(13, 13))
    axes = axes.flatten()

    plot_step_to_go(axes[0], steps, optimal_steps, '(A)')
    plot_rms_error(axes[1], rms, '(B)')
    plot_path(axes[2], pi, grid, '(C)')
    plot_state_value_heatmap(axes[3], V, fig, '(D)')

    fig.suptitle(title)
    plt.tight_layout()
    filename = ''.join(title.split())
    plt.savefig(f'{filename}.pdf')
```

4 Dynamic Programming

4.1 Iterative Policy Evaluation

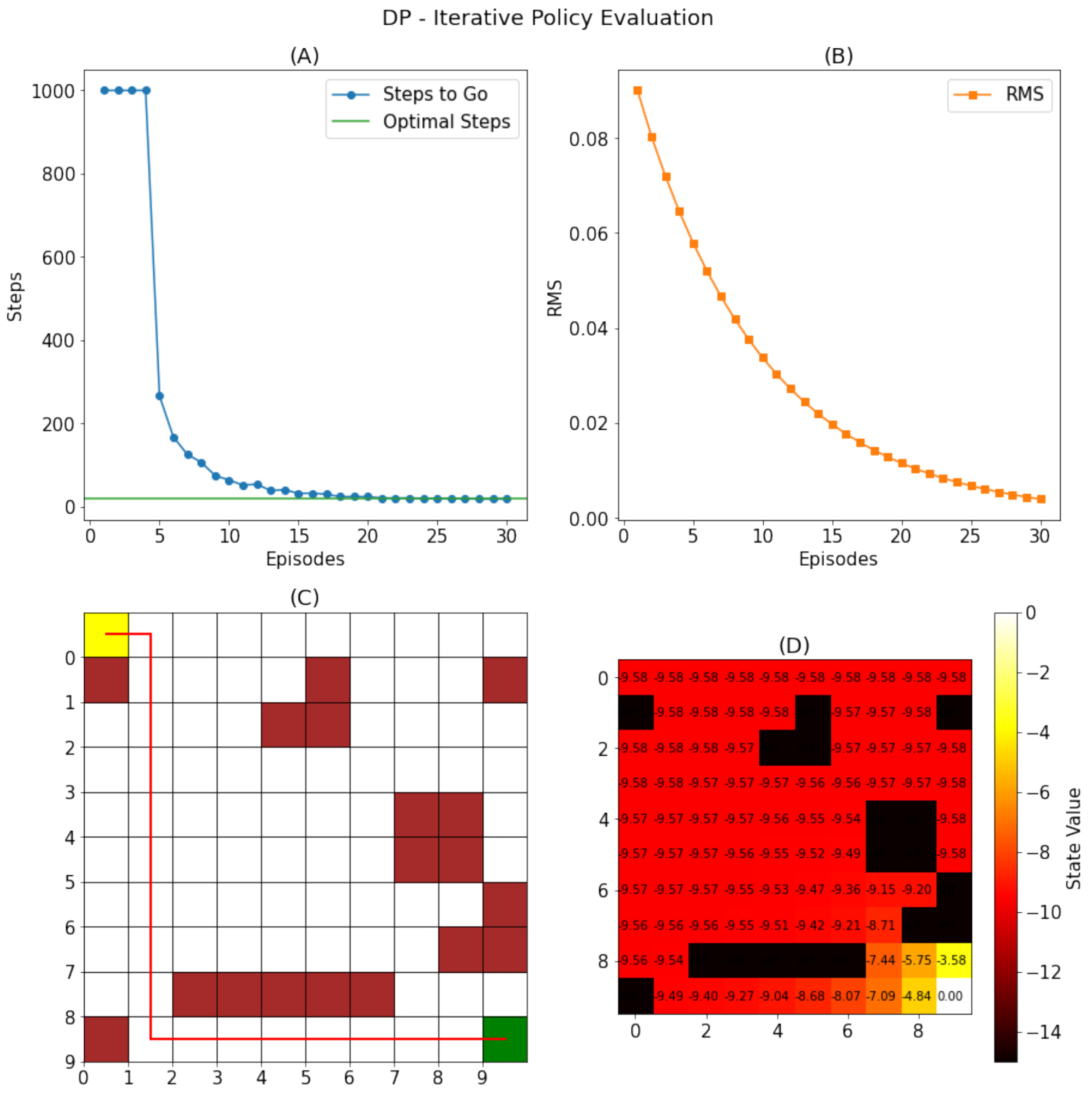
- Random action

[1094]:
```
dp_it_pol_eval_V, _, grid, dp_it_pol_eval_pi = initialize_grid(M, N, NUM_BLK,␣
↪random_state=10)
V_pre = np.copy(dp_it_pol_eval_V)
dp_it_pol_eval_steps = []
dp_it_pol_eval_rms = []
for _ in range(30):
    # Iterate through ALL states
    for i in range(M):
        for j in range(N):
            if (i == M - 1 and j == N - 1) or grid[i][j]:
                continue  # skip terminal state and blocks
            num_action_taken = 0
            total = 0
            for di, dj in ACTIONS:
                ni, nj = i + di, j + dj
                if 0 <= ni < M and 0 <= nj < N and grid[ni, nj] == 0:
                    num_action_taken += 1
                    total += (-1 + GAMMA * V_pre[ni, nj])
            dp_it_pol_eval_V[i, j] = total / num_action_taken
    # Compute error metrics
    dp_it_pol_eval_steps.append(steps_to_go(dp_it_pol_eval_V, grid))
    dp_it_pol_eval_rms.append(np.linalg.norm(dp_it_pol_eval_V - V_pre) / (M *␣
↪N))
    V_pre = np.copy(dp_it_pol_eval_V)

update_policy(dp_it_pol_eval_V, dp_it_pol_eval_pi, grid)
dp_it_pol_eval_steps = np.array(dp_it_pol_eval_steps)
dp_it_pol_eval_rms = np.array(dp_it_pol_eval_rms)

# Pickle everything
with open('dp_it_pol_eval_steps.pickle', 'wb') as f_obj:
    dill.dump(dp_it_pol_eval_steps, f_obj)
with open('dp_it_pol_eval_rms.pickle', 'wb') as f_obj:
    dill.dump(dp_it_pol_eval_rms, f_obj)
with open('dp_it_pol_eval_V.pickle', 'wb') as f_obj:
    dill.dump(dp_it_pol_eval_V, f_obj)
with open('dp_it_pol_eval_pi.pickle', 'wb') as f_obj:
    dill.dump(dp_it_pol_eval_pi, f_obj)
```

[1124]:
```
with open('dp_it_pol_eval_steps.pickle', 'rb') as f_obj:
    dp_it_pol_eval_steps = dill.load(f_obj)
with open('dp_it_pol_eval_rms.pickle', 'rb') as f_obj:
    dp_it_pol_eval_rms = dill.load(f_obj)
with open('dp_it_pol_eval_V.pickle', 'rb') as f_obj:
    dp_it_pol_eval_V = dill.load(f_obj)
with open('dp_it_pol_eval_pi.pickle', 'rb') as f_obj:
    dp_it_pol_eval_pi = dill.load(f_obj)

plot(
    dp_it_pol_eval_steps,
    dp_it_pol_eval_rms,
    dp_it_pol_eval_V,
    dp_it_pol_eval_pi,
    OPTIMAL_STEPS,
    'DP - Iterative Policy Evaluation',
)
print('First episode reach optimal steps', np.where(dp_it_pol_eval_steps ==␣
↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(dp_it_pol_eval_rms, 0.
↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(dp_it_pol_eval_rms)} episodes')
```

```
First episode reach optimal steps 27
```

```
RMS not converged yet after 30 episodes
```



4.2 Policy Iteration

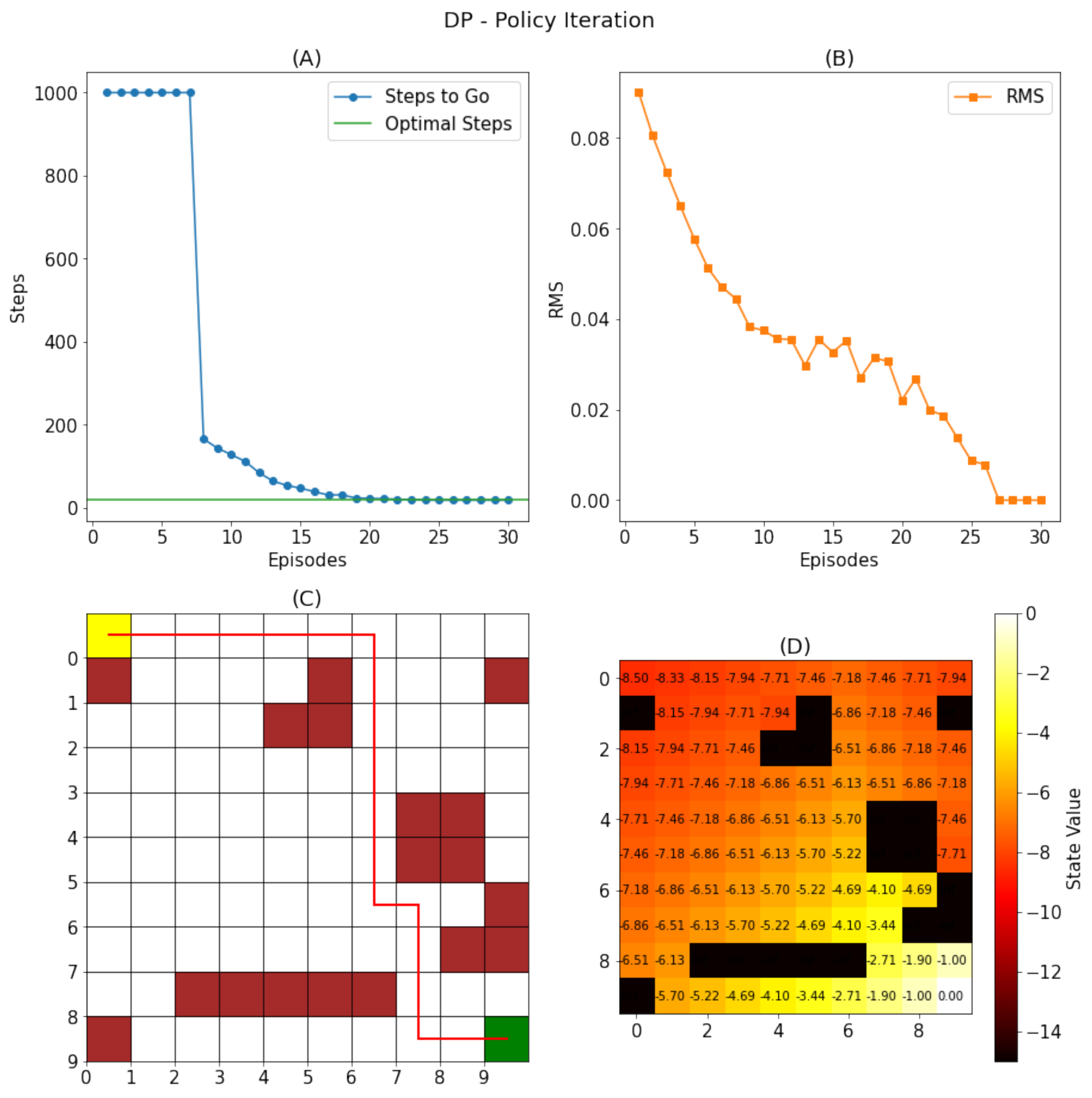
- Action controlled by policy

[1096]:
```
dp_pol_it_V, _, grid, dp_pol_it_pi = initialize_grid(M, N, NUM_BLK,␣
 ↪random_state=10)
rng = np.random.default_rng(42)
V_pre = np.copy(dp_pol_it_V)
dp_pol_it_steps = []
dp_pol_it_rms = []
for _ in range(30):
    # Policy Evaluation
    for i in range(M):
        for j in range(N):
            if (i == M - 1 and j == N - 1) or grid[i][j]:
                continue  # skip terminal state and blocks
            di, dj = ACTIONS[dp_pol_it_pi[i, j]]
            dp_pol_it_V[i, j] = -1 + GAMMA * V_pre[i + di, j + dj]
    # Compute error metrics
    dp_pol_it_steps.append(steps_to_go(dp_pol_it_V, grid))
    dp_pol_it_rms.append(np.linalg.norm(dp_pol_it_V - V_pre) / (M * N))
    # Policy Improvement
    policy_stable = True
    for i in range(M):
        for j in range(N):
            nsvs = get_next_state_vals(i, j, V_pre, grid, fv=lambda v: -1 +␣
↪GAMMA * v)
            best_acts = find_best_actions(nsvs)
            A = rng.choice(best_acts)
            if A != dp_pol_it_pi[i, j]:
                policy_stable = False
                dp_pol_it_pi[i, j] = A
    if policy_stable:
        break
    V_pre = np.copy(dp_pol_it_V)

dp_pol_it_steps = np.array(dp_pol_it_steps)
dp_pol_it_rms = np.array(dp_pol_it_rms)

# Pickle everything
with open('dp_pol_it_steps.pickle', 'wb') as f_obj:
    dill.dump(dp_pol_it_steps, f_obj)
with open('dp_pol_it_rms.pickle', 'wb') as f_obj:
    dill.dump(dp_pol_it_rms, f_obj)
with open('dp_pol_it_V.pickle', 'wb') as f_obj:
    dill.dump(dp_pol_it_V, f_obj)
with open('dp_pol_it_pi.pickle', 'wb') as f_obj:
    dill.dump(dp_pol_it_pi, f_obj)
```

[1125]:
```
with open('dp_pol_it_steps.pickle', 'rb') as f_obj:
    dp_pol_it_steps = dill.load(f_obj)
with open('dp_pol_it_rms.pickle', 'rb') as f_obj:
    dp_pol_it_rms = dill.load(f_obj)
with open('dp_pol_it_V.pickle', 'rb') as f_obj:
    dp_pol_it_V = dill.load(f_obj)
with open('dp_pol_it_pi.pickle', 'rb') as f_obj:
    dp_pol_it_pi = dill.load(f_obj)
```

```
plot(
    dp_pol_it_steps,
    dp_pol_it_rms,
    dp_pol_it_V,
    dp_pol_it_pi,
    OPTIMAL_STEPS,
    'DP - Policy Iteration',
)
print('First episode reach optimal steps', np.where(dp_pol_it_steps ==␣
 ↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(dp_pol_it_rms, 0.
 ↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(dp_pol_it_rms)} episodes')
```

```
First episode reach optimal steps 23
First episode converge 26
```



4.3 Value Iteration

- Action controlled by policy

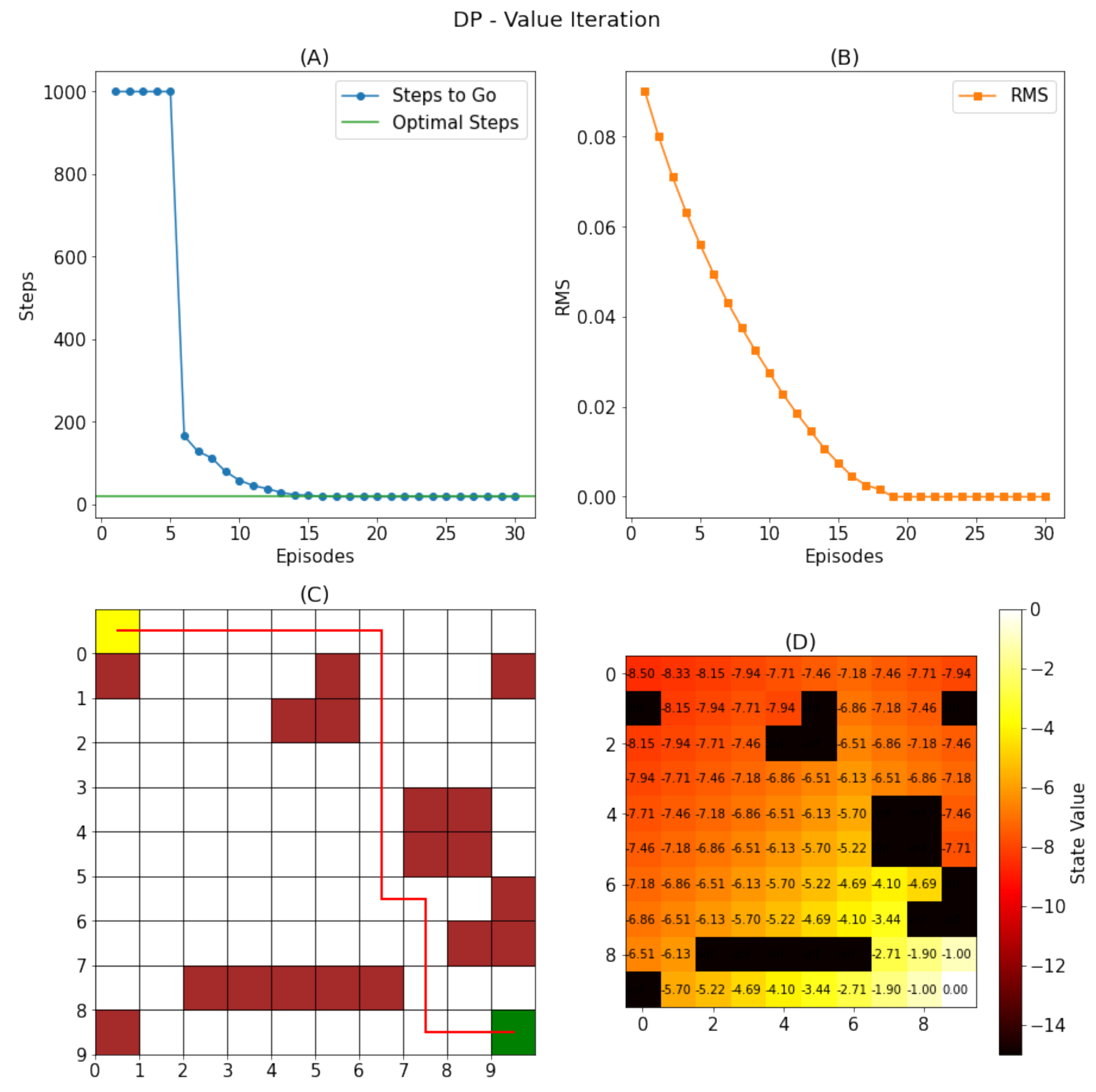
```
[1098]: dp_val_it_V, _, grid, dp_val_it_pi = initialize_grid(M, N, NUM_BLK,␣
         ↪random_state=10)
        rng = np.random.default_rng(42)
        V_pre = np.copy(dp_val_it_V)
        dp_val_it_steps = []
        dp_val_it_rms = []
        for _ in range(30):
            for i in range(M):
                for j in range(N):
```

```
            if (i == M - 1 and j == N - 1) or grid[i][j]:
                continue  # skip terminal state and blocks
            nsvs = get_next_state_vals(i, j, V_pre, grid, fv=lambda v: -1 +␣
↪GAMMA * v)
            dp_val_it_V[i, j] = max(nsvs)  # assign best next state value
    # Compute error metrics
    dp_val_it_steps.append(steps_to_go(dp_val_it_V, grid))
    dp_val_it_rms.append(np.linalg.norm(dp_val_it_V - V_pre) / (M * N))
    # update V_pre
    V_pre = np.copy(dp_val_it_V)

update_policy(dp_val_it_V, dp_val_it_pi, grid)
dp_val_it_steps = np.array(dp_val_it_steps)
dp_val_it_rms = np.array(dp_val_it_rms)

# Pickle everything
with open('dp_val_it_steps.pickle', 'wb') as f_obj:
    dill.dump(dp_val_it_steps, f_obj)
with open('dp_val_it_rms.pickle', 'wb') as f_obj:
    dill.dump(dp_val_it_rms, f_obj)
with open('dp_val_it_V.pickle', 'wb') as f_obj:
    dill.dump(dp_val_it_V, f_obj)
with open('dp_val_it_pi.pickle', 'wb') as f_obj:
    dill.dump(dp_val_it_pi, f_obj)
```

[1126]:
```
with open('dp_val_it_steps.pickle', 'rb') as f_obj:
    dp_val_it_steps = dill.load(f_obj)
with open('dp_val_it_rms.pickle', 'rb') as f_obj:
    dp_val_it_rms = dill.load(f_obj)
with open('dp_val_it_V.pickle', 'rb') as f_obj:
    dp_val_it_V = dill.load(f_obj)
with open('dp_val_it_pi.pickle', 'rb') as f_obj:
    dp_val_it_pi = dill.load(f_obj)

plot(
    dp_val_it_steps,
    dp_val_it_rms,
    dp_val_it_V,
    dp_val_it_pi,
    OPTIMAL_STEPS,
    'DP - Value Iteration',
)
print('First episode reach optimal steps', np.where(dp_val_it_steps ==␣
↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(dp_val_it_rms, 0.
↪0))[0][0])
```

```
except IndexError:
    print(f'RMS not converged yet after {len(dp_val_it_rms)} episodes')
```

```
First episode reach optimal steps 16
First episode converge 18
```



5 Monte Carlo

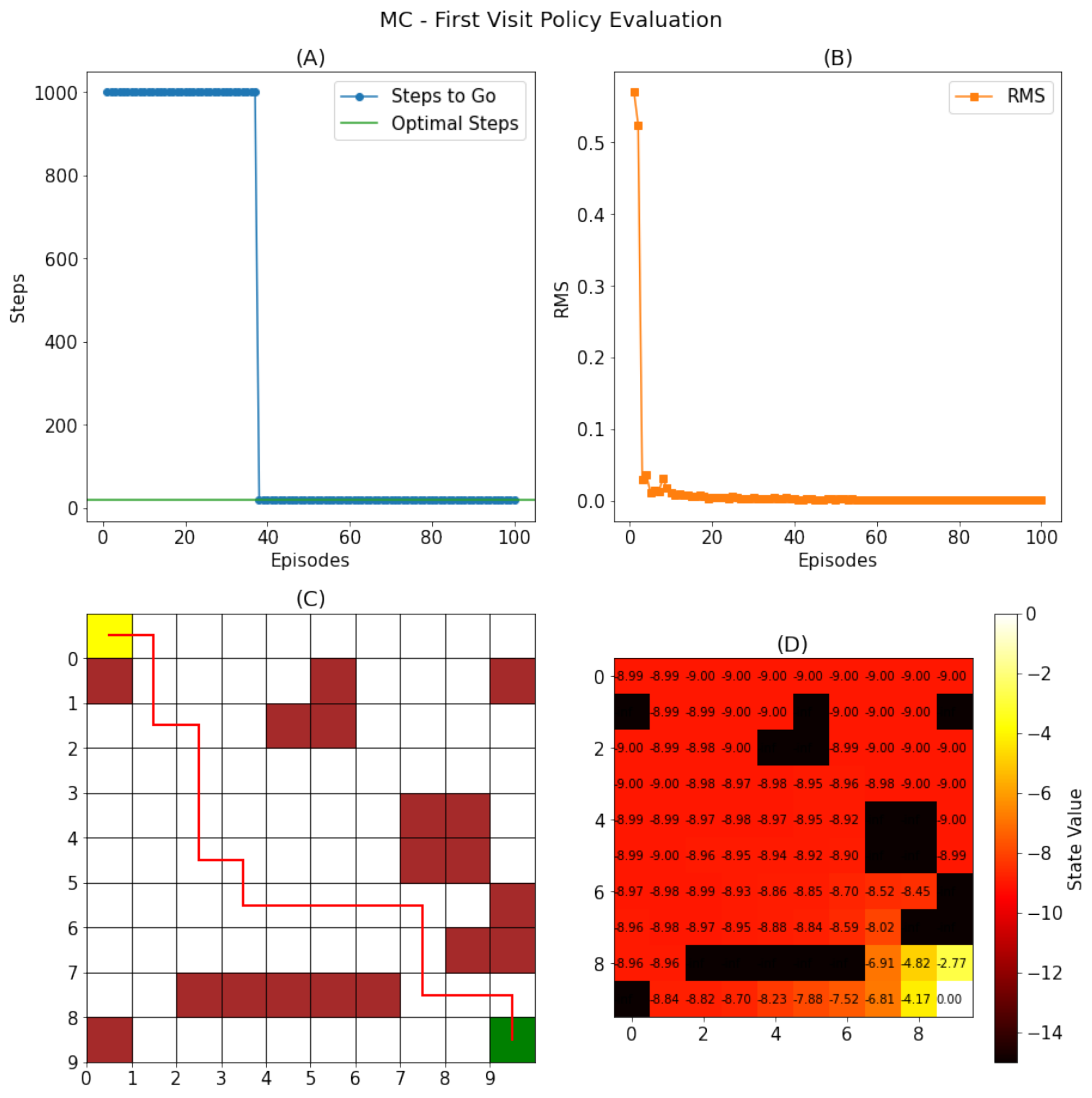
5.1 First-visit Policy Evalutation

- Action completely random
- No need to set 1,000-step restriction per episode.

```
[1120]: mc_fst_vst_pol_eval_V, _, grid, mc_fst_vst_pol_eval_pi = initialize_grid(M, N,␣
         ↪NUM_BLK, random_state=10)
        rng = np.random.default_rng(42)
        V_pre = np.copy(mc_fst_vst_pol_eval_V)
        mc_fst_vst_pol_eval_steps = []
        mc_fst_vst_pol_eval_rms = []
        # store all the returns encountered at each state
        returns = [[[] for _ in range(n)] for _ in range(m)]
        for _ in range(100):
            states = []
            i, j = 0, 0
            while i != m - 1 or j != n - 1:  # One episode
                states.append((i, j))
                while True:
                    di, dj = rng.choice(ACTIONS)  # take a random action
                    ni, nj = i + di, j + dj
                    if 0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0:
                        i, j = ni, nj
                        break
            seen = set()
            for k, state in enumerate(states):
                if state not in seen:  # only count first visit
                    seen.add(state)
                    G = -(GAMMA - GAMMA**(len(states) - k)) / (1 - GAMMA)
                    returns[state[0]][state[1]].append(G)
                    mc_fst_vst_pol_eval_V[state[0], state[1]] = np.
         ↪mean(returns[state[0]][state[1]])
            # Compute error metrics
            mc_fst_vst_pol_eval_steps.append(steps_to_go(mc_fst_vst_pol_eval_V, grid))
            mc_fst_vst_pol_eval_rms.append(np.linalg.norm(mc_fst_vst_pol_eval_V -␣
         ↪V_pre) / (M * N))
            # update V_pre
            V_pre = np.copy(mc_fst_vst_pol_eval_V)

        update_policy(mc_fst_vst_pol_eval_V, mc_fst_vst_pol_eval_pi, grid)
        mc_fst_vst_pol_eval_steps = np.array(mc_fst_vst_pol_eval_steps)
        mc_fst_vst_pol_eval_rms = np.array(mc_fst_vst_pol_eval_rms)

        # Pickle everything
        with open('mc_fst_vst_pol_eval_steps.pickle', 'wb') as f_obj:
            dill.dump(mc_fst_vst_pol_eval_steps, f_obj)
        with open('mc_fst_vst_pol_eval_rms.pickle', 'wb') as f_obj:
            dill.dump(mc_fst_vst_pol_eval_rms, f_obj)
        with open('mc_fst_vst_pol_eval_V.pickle', 'wb') as f_obj:
            dill.dump(mc_fst_vst_pol_eval_V, f_obj)
        with open('mc_fst_vst_pol_eval_pi.pickle', 'wb') as f_obj:
            dill.dump(mc_fst_vst_pol_eval_pi, f_obj)
```

```
[1127]: with open('mc_fst_vst_pol_eval_steps.pickle', 'rb') as f_obj:
            mc_fst_vst_pol_eval_steps = dill.load(f_obj)
        with open('mc_fst_vst_pol_eval_rms.pickle', 'rb') as f_obj:
            mc_fst_vst_pol_eval_rms = dill.load(f_obj)
        with open('mc_fst_vst_pol_eval_V.pickle', 'rb') as f_obj:
            mc_fst_vst_pol_eval_V = dill.load(f_obj)
        with open('mc_fst_vst_pol_eval_pi.pickle', 'rb') as f_obj:
            mc_fst_vst_pol_eval_pi = dill.load(f_obj)

        plot(
            mc_fst_vst_pol_eval_steps,
            mc_fst_vst_pol_eval_rms,
            mc_fst_vst_pol_eval_V,
            mc_fst_vst_pol_eval_pi,
            OPTIMAL_STEPS,
            'MC - First Visit Policy Evaluation',
        )
        print('First episode reach optimal steps', np.where(mc_fst_vst_pol_eval_steps␣
         ↪== OPTIMAL_STEPS)[0][0])
        try:
            print('First episode converge', np.where(np.
         ↪isclose(mc_fst_vst_pol_eval_rms, 0.0))[0][0])
        except IndexError:
            print(f'RMS not converged yet after {len(mc_fst_vst_pol_eval_rms)}␣
         ↪episodes')
```

```
First episode reach optimal steps 37
RMS not converged yet after 100 episodes
```



5.2 Exploring Starts

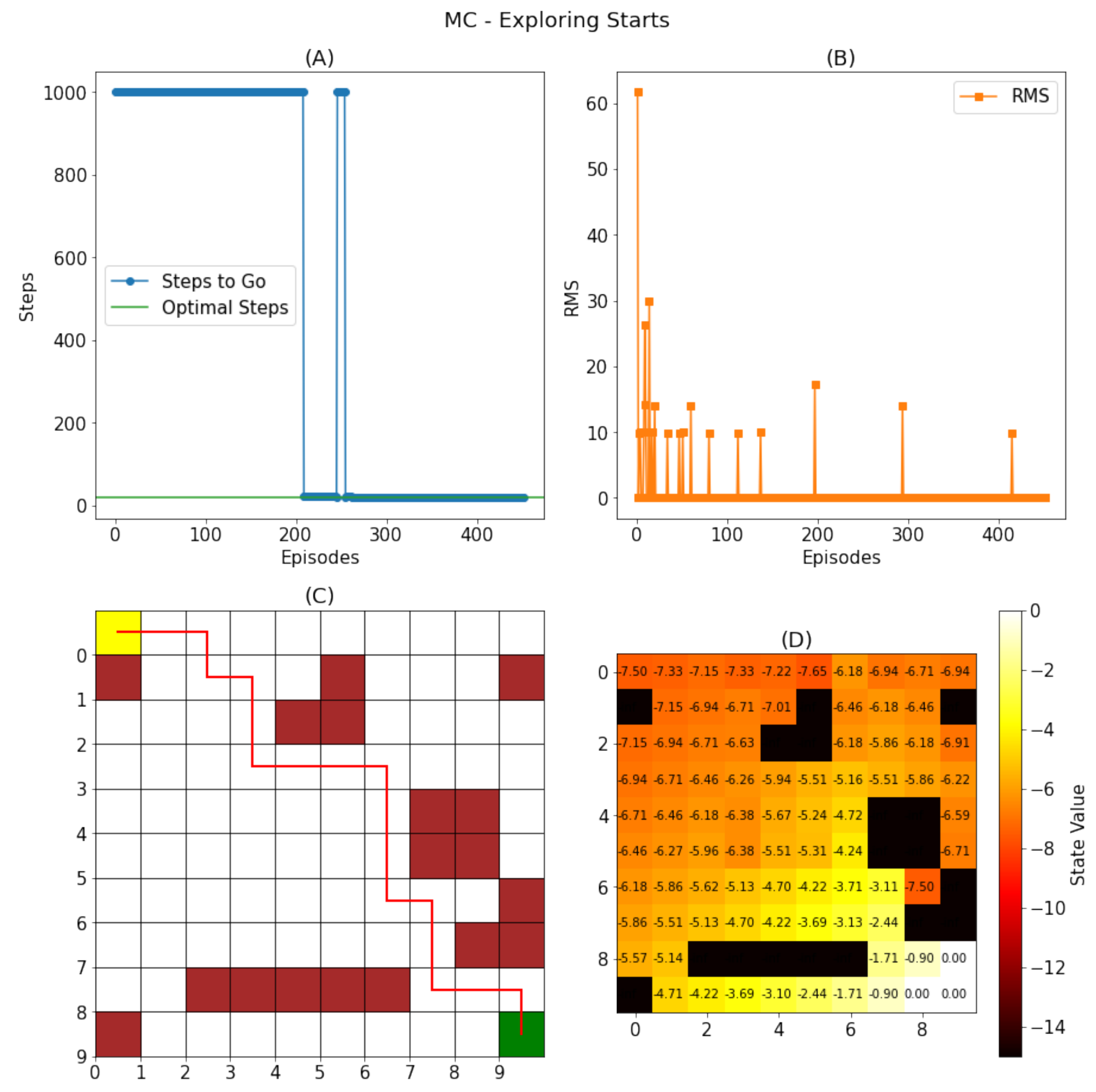
- Action follows policy
- Set 1,000-step restriction per episode in case an episode fails to terminate
- Failed episode discarded, i.e. not used to update state or action values.
- Initial action value set to a big negative value.

[1102]:
```
mc_epl_starts_V, mc_epl_starts_Q, grid, mc_epl_starts_pi = initialize_grid(M,␣
↪N, NUM_BLK, random_state=10)
rng = np.random.default_rng(42)
V_pre = np.copy(mc_epl_starts_V)
mc_epl_starts_steps = []
mc_epl_starts_rms = []
```

```
m, n = mc_epl_starts_V.shape
# store all the returns encountered at each state, action pair
returns = [[[[], [], [], []] for _ in range(n)] for _ in range(m)]
# Re-initialize Q values
for i in range(m):
    for j in range(n):
        if i == m - 1 and j == n - 1:
            continue
        mc_epl_starts_Q[i, j] = np.full(len(ACTIONS), -1000.0)

for _ in range(600):
    states = []
    while True:  # random start with state and action
        i, j, A = rng.integers(0, m), rng.integers(0, n), rng.integers(0,␣
↪len(ACTIONS))
        di, dj = ACTIONS[A]
        ni, nj = i + di, j + dj
        if 0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0 and grid[i, j] ==␣
↪0:
            break
    c = 0
    max_steps = 1000
    while (i != m - 1 or j != n - 1) and c < max_steps:  # One episode
        states.append((i, j, A))
        i, j = i + ACTIONS[A][0], j + ACTIONS[A][1]
        if any(returns[i][j]):  # the next state has been visited before
            A = mc_epl_starts_pi[i, j]
        else:  # the next state hasn't been visited, use random action
            while True:
                A = rng.integers(0, len(ACTIONS))
                ni, nj = i + ACTIONS[A][0], j + ACTIONS[A][1]
                if 0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0:
                    break
        c += 1
    if c < max_steps:  # Only update state values when an episode reaches␣
↪terminal state
        seen = set()
        for k, state in enumerate(states):
            if state not in seen:  # only count first visit
                seen.add(state)
                G = -(GAMMA - GAMMA**(len(states) - k)) / (1 - GAMMA)
                returns[state[0]][state[1]][state[2]].append(G)
                mc_epl_starts_Q[state[0], state[1], state[2]] = np.
↪mean(returns[state[0]][state[1]][state[2]])
        # update V and pi
        mc_epl_starts_V = np.max(mc_epl_starts_Q, axis=2)
        mc_epl_starts_V[m - 1, n - 1] = 0
        update_policy(mc_epl_starts_V, mc_epl_starts_pi, grid)
        # Compute error metrics
        mc_epl_starts_steps.append(steps_to_go(mc_epl_starts_V, grid))
        mc_epl_starts_rms.append(np.linalg.norm(mc_epl_starts_V - V_pre) / (M *␣
↪N))
        # update V_pre
        V_pre = np.copy(mc_epl_starts_V)

update_policy(mc_epl_starts_V, mc_epl_starts_pi, grid)
mc_epl_starts_steps = np.array(mc_epl_starts_steps)
mc_epl_starts_rms = np.array(mc_epl_starts_rms)

# Pickle everything
with open('mc_epl_starts_steps.pickle', 'wb') as f_obj:
    dill.dump(mc_epl_starts_steps, f_obj)
with open('mc_epl_starts_rms.pickle', 'wb') as f_obj:
    dill.dump(mc_epl_starts_rms, f_obj)
with open('mc_epl_starts_V.pickle', 'wb') as f_obj:
    dill.dump(mc_epl_starts_V, f_obj)
with open('mc_epl_starts_pi.pickle', 'wb') as f_obj:
    dill.dump(mc_epl_starts_pi, f_obj)
```

[1128]:
```
with open('mc_epl_starts_steps.pickle', 'rb') as f_obj:
    mc_epl_starts_steps = dill.load(f_obj)
with open('mc_epl_starts_rms.pickle', 'rb') as f_obj:
    mc_epl_starts_rms = dill.load(f_obj)
with open('mc_epl_starts_V.pickle', 'rb') as f_obj:
    mc_epl_starts_V = dill.load(f_obj)
with open('mc_epl_starts_pi.pickle', 'rb') as f_obj:
    mc_epl_starts_pi = dill.load(f_obj)

plot(
    mc_epl_starts_steps,
    mc_epl_starts_rms,
    mc_epl_starts_V,
    mc_epl_starts_pi,
    OPTIMAL_STEPS,
    'MC - Exploring Starts',
)
print('First episode reach optimal steps', np.where(mc_epl_starts_steps ==␣
↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(mc_epl_starts_rms, 0.
↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(mc_epl_starts_rms)} episodes')
```

```
First episode reach optimal steps 411
First episode converge 6
```



5.3 On-Policy MC Control

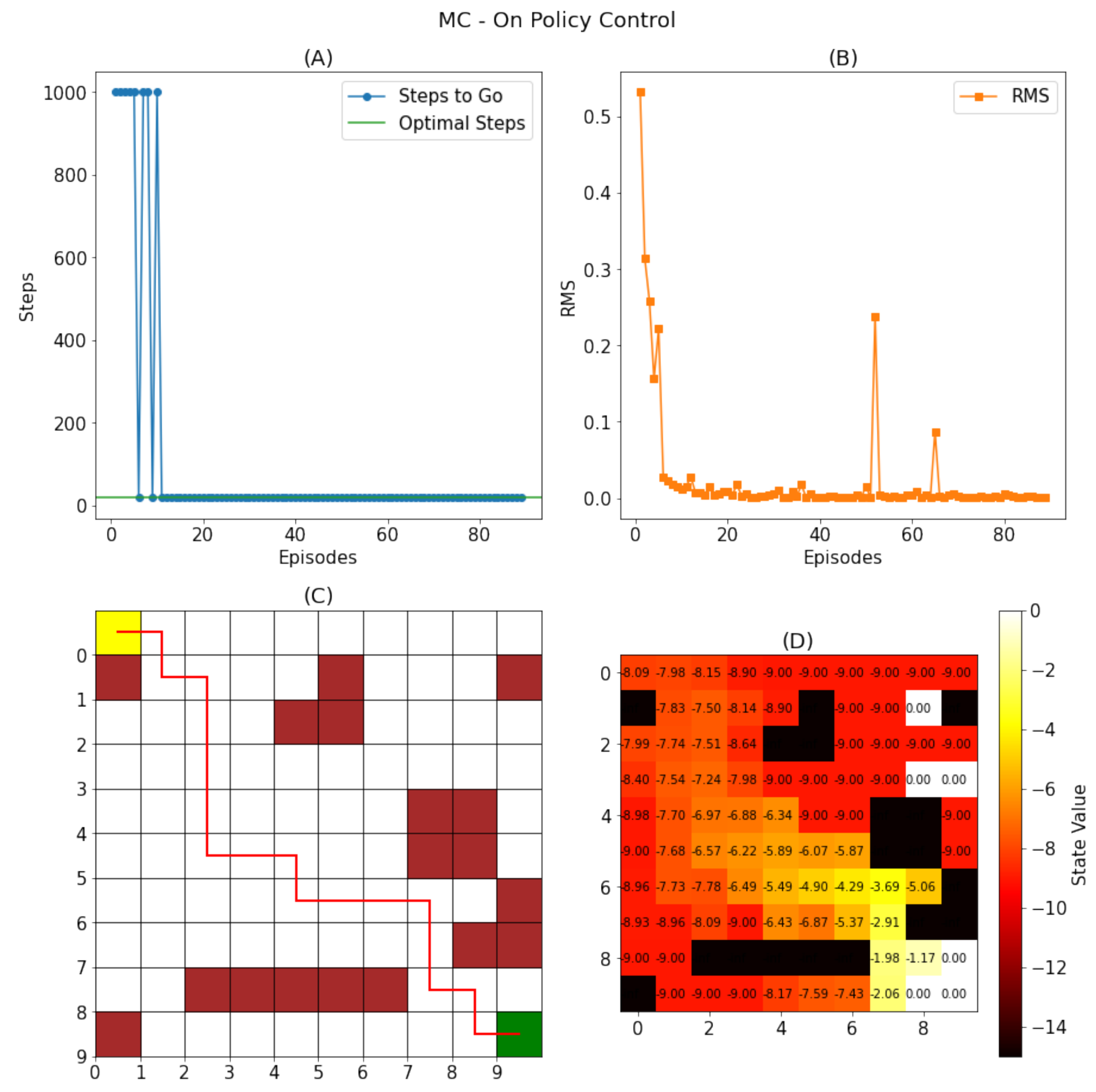
- Action follows policy
- Set 1,000-step restriction per episode in case an episode fails to terminate
- Failed episode discarded, i.e. not used to update state or action values.

[1104]:
```
mc_on_policy_V, mc_on_policy_Q, grid, mc_on_policy_pi = initialize_grid(M, N,␣
↪NUM_BLK, random_state=10)
rng = np.random.default_rng(42)
V_pre = np.copy(mc_on_policy_V)
```

```
mc_on_policy_steps = []
mc_on_policy_rms = []
m, n = mc_on_policy_V.shape
# store all the returns encountered at each state, action pair
returns = [[[[], [], [], []] for _ in range(n)] for _ in range(m)]

# set up epsilon-greedy policy. Initial policy as equal probability
# for each action
epsilon_pi = np.array([[[1 / len(ACTIONS)] * len(ACTIONS) for _ in range(n)]␣
 ↪for _ in range(m)])

for _ in range(100):
    states = []
    i, j = 0, 0
    c = 0
    max_steps = 1000
    while (i != m - 1 or j != n - 1) and c < max_steps:  # One episode
        while True:
            A = rng.choice(len(ACTIONS), p=epsilon_pi[i, j])
            di, dj = ACTIONS[A]
            ni, nj = i + di, j + dj
            if 0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0:
                break  # valid action
        states.append((i, j, A))
        i, j = ni, nj
        c += 1
    if c < max_steps:  # only update state-action values when an episode␣
 ↪succeeds
        seen = set()
        for k, state in enumerate(states):
            if state not in seen:  # only count first visit
                seen.add(state)
                G = -(GAMMA - GAMMA**(len(states) - k)) / (1 - GAMMA)
                returns[state[0]][state[1]][state[2]].append(G)
                mc_on_policy_Q[state[0], state[1], state[2]] = np.
 ↪mean(returns[state[0]][state[1]][state[2]])
        # update V and pi
        mc_on_policy_V = np.max(mc_on_policy_Q, axis=2)
        mc_on_policy_V[m - 1, n - 1] = 0
        update_policy(mc_on_policy_V, mc_on_policy_pi, grid)
        # update epsilon_pi
        for i, j, _ in states:
            A_star = mc_on_policy_pi[i, j]
            for k in range(len(ACTIONS)):
                if k == A_star:
                    epsilon_pi[i, j, k] = 1 - EPSILON + EPSILON / len(ACTIONS)
                else:
                    epsilon_pi[i, j, k] = EPSILON / len(ACTIONS)

        # Compute error metrics
        mc_on_policy_steps.append(steps_to_go(mc_on_policy_V, grid))
        mc_on_policy_rms.append(np.linalg.norm(mc_on_policy_V - V_pre) / (M *␣
↪N))
        # update V_pre
        V_pre = np.copy(mc_on_policy_V)

update_policy(mc_on_policy_V, mc_on_policy_pi, grid)
mc_on_policy_steps = np.array(mc_on_policy_steps)
mc_on_policy_rms = np.array(mc_on_policy_rms)

# Pickle everything
with open('mc_on_policy_steps.pickle', 'wb') as f_obj:
    dill.dump(mc_on_policy_steps, f_obj)
with open('mc_on_policy_rms.pickle', 'wb') as f_obj:
    dill.dump(mc_on_policy_rms, f_obj)
with open('mc_on_policy_V.pickle', 'wb') as f_obj:
    dill.dump(mc_on_policy_V, f_obj)
with open('mc_on_policy_pi.pickle', 'wb') as f_obj:
    dill.dump(mc_on_policy_pi, f_obj)
```

[1129]:
```
with open('mc_on_policy_steps.pickle', 'rb') as f_obj:
    mc_on_policy_steps = dill.load(f_obj)
with open('mc_on_policy_rms.pickle', 'rb') as f_obj:
    mc_on_policy_rms = dill.load(f_obj)
with open('mc_on_policy_V.pickle', 'rb') as f_obj:
    mc_on_policy_V = dill.load(f_obj)
with open('mc_on_policy_pi.pickle', 'rb') as f_obj:
    mc_on_policy_pi = dill.load(f_obj)

plot(
    mc_on_policy_steps,
    mc_on_policy_rms,
    mc_on_policy_V,
    mc_on_policy_pi,
    OPTIMAL_STEPS,
    'MC - On Policy Control',
)
print('First episode reach optimal steps', np.where(mc_on_policy_steps ==␣
↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(mc_on_policy_rms, 0.
↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(mc_on_policy_steps)} episodes')
```

```
First episode reach optimal steps 5
RMS not converged yet after 89 episodes
```



6 Temporal Difference Learning

6.1 TD(0)

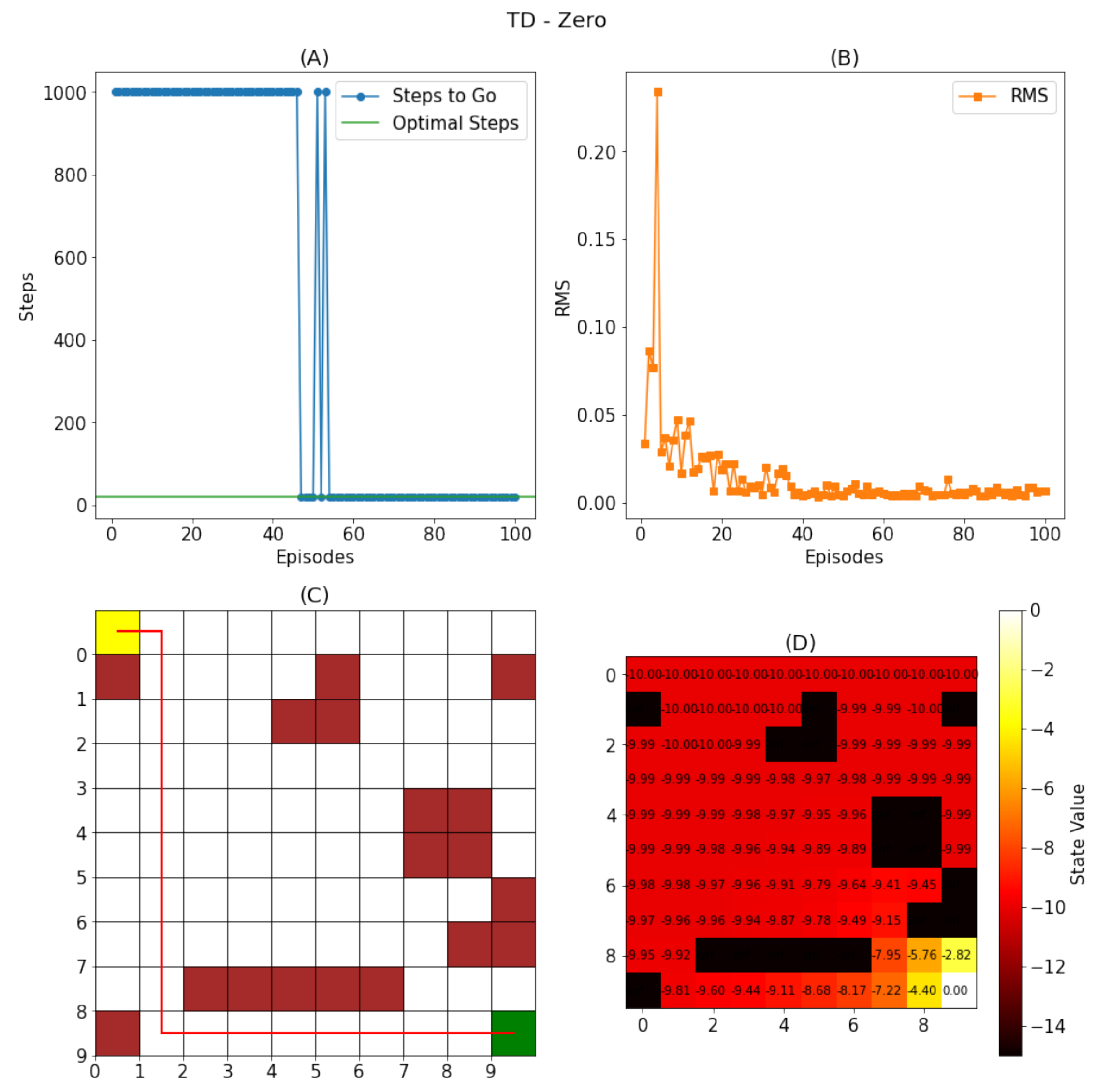
- Action completely random
- No step restriction on episode

```
[1139]: td_zero_V, _, grid, td_zero_pi = initialize_grid(M, N, NUM_BLK, random_state=10)
rng = np.random.default_rng(42)
V_pre = np.copy(td_zero_V)
```

```
td_zero_steps = []
td_zero_rms = []
m, n = td_zero_V.shape

for _ in range(100):
    i, j = 0, 0
    while i != m - 1 or j != n - 1:  # One episode
        while True:  # take random action
            di, dj = rng.choice(ACTIONS)
            ni, nj = i + di, j + dj
            if 0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0:
                break  # valid action
        td_zero_V[i, j] = td_zero_V[i, j] + ALPHA * (-1 + GAMMA * td_zero_V[ni,␣
 ↪nj] - td_zero_V[i, j])
        i, j = ni, nj

    # Compute error metrics
    td_zero_steps.append(steps_to_go(td_zero_V, grid))
    td_zero_rms.append(np.linalg.norm(td_zero_V - V_pre) / (M * N))
    # update V_pre
    V_pre = np.copy(td_zero_V)

update_policy(td_zero_V, td_zero_pi, grid)
td_zero_steps = np.array(td_zero_steps)
td_zero_rms = np.array(td_zero_rms)

# Pickle everything
with open('td_zero_steps.pickle', 'wb') as f_obj:
    dill.dump(td_zero_steps, f_obj)
with open('td_zero_rms.pickle', 'wb') as f_obj:
    dill.dump(td_zero_rms, f_obj)
with open('td_zero_V.pickle', 'wb') as f_obj:
    dill.dump(td_zero_V, f_obj)
with open('td_zero_pi.pickle', 'wb') as f_obj:
    dill.dump(td_zero_pi, f_obj)
```

[1140]:
```
with open('td_zero_steps.pickle', 'rb') as f_obj:
    td_zero_steps = dill.load(f_obj)
with open('td_zero_rms.pickle', 'rb') as f_obj:
    td_zero_rms = dill.load(f_obj)
with open('td_zero_V.pickle', 'rb') as f_obj:
    td_zero_V = dill.load(f_obj)
with open('td_zero_pi.pickle', 'rb') as f_obj:
    td_zero_pi = dill.load(f_obj)


plot(
    td_zero_steps,
    td_zero_rms,
    td_zero_V,
    td_zero_pi,
    OPTIMAL_STEPS,
    'TD - Zero',
)
print('First episode reach optimal steps', np.where(td_zero_steps ==␣
 ↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(td_zero_rms, 0.
 ↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(td_zero_steps)} episodes')
```

```
First episode reach optimal steps 46
RMS not converged yet after 100 episodes
```



6.2 Sarsa with TD Control

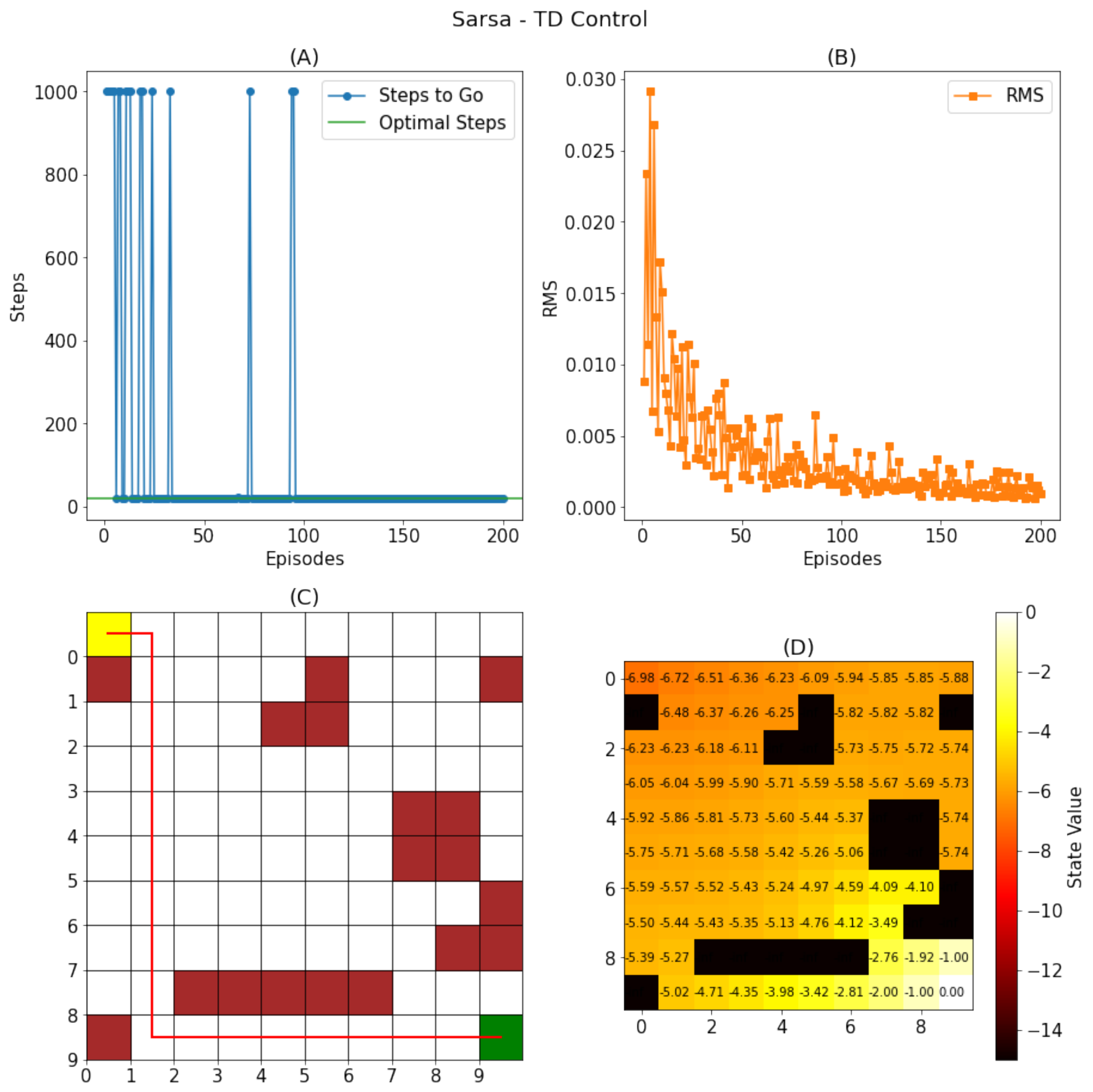
- Action follow policy using ϵ-greedy
- No step restriction on episode

```
[1141]: sarsa_td_V, sarsa_td_Q, grid, sarsa_td_pi = initialize_grid(M, N, NUM_BLK,␣
         ↪random_state=10)
        rng = np.random.default_rng(42)
        V_pre = np.copy(sarsa_td_V)
        sarsa_td_steps = []
        sarsa_td_rms = []
        m, n = sarsa_td_V.shape
```

```
for _ in range(200):
    i, j = 0, 0
    A = get_epsilon_greedy_action(sarsa_td_Q, i, j, grid)
    while i != m - 1 or j != n - 1:  # One episode
        ni, nj = i + ACTIONS[A][0], j + ACTIONS[A][1]
        A_prime = get_epsilon_greedy_action(sarsa_td_Q, ni, nj, grid)
        sarsa_td_Q[i, j, A] = sarsa_td_Q[i, j, A] + ALPHA * (-1 + GAMMA *␣
 ↪sarsa_td_Q[ni, nj, A_prime] - sarsa_td_Q[i, j, A])
        i, j = ni, nj
        A = A_prime

    # update V
    sarsa_td_V = np.max(sarsa_td_Q, axis=2)

    # Compute error metrics
    sarsa_td_steps.append(steps_to_go(sarsa_td_V, grid))
    sarsa_td_rms.append(np.linalg.norm(sarsa_td_V - V_pre) / (M * N))
    # update V_pre
    V_pre = np.copy(sarsa_td_V)

update_policy(sarsa_td_V, sarsa_td_pi, grid)
sarsa_td_steps = np.array(sarsa_td_steps)
sarsa_td_rms = np.array(sarsa_td_rms)

# Pickle everything
with open('sarsa_td_steps.pickle', 'wb') as f_obj:
    dill.dump(sarsa_td_steps, f_obj)
with open('sarsa_td_rms.pickle', 'wb') as f_obj:
    dill.dump(sarsa_td_rms, f_obj)
with open('sarsa_td_V.pickle', 'wb') as f_obj:
    dill.dump(sarsa_td_V, f_obj)
with open('sarsa_td_pi.pickle', 'wb') as f_obj:
    dill.dump(sarsa_td_pi, f_obj)
```

[1142]:
```
with open('sarsa_td_steps.pickle', 'rb') as f_obj:
    sarsa_td_steps = dill.load(f_obj)
with open('sarsa_td_rms.pickle', 'rb') as f_obj:
    sarsa_td_rms = dill.load(f_obj)
with open('sarsa_td_V.pickle', 'rb') as f_obj:
    sarsa_td_V = dill.load(f_obj)
with open('sarsa_td_pi.pickle', 'rb') as f_obj:
    sarsa_td_pi = dill.load(f_obj)

plot(
    sarsa_td_steps,
    sarsa_td_rms,
    sarsa_td_V,
    sarsa_td_pi,
    OPTIMAL_STEPS,
    'Sarsa - TD Control',
)
print('First episode reach optimal steps', np.where(sarsa_td_steps ==␣
 ↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(sarsa_td_rms, 0.
 ↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(sarsa_td_steps)} episodes')
```

```
First episode reach optimal steps 5
RMS not converged yet after 200 episodes
```



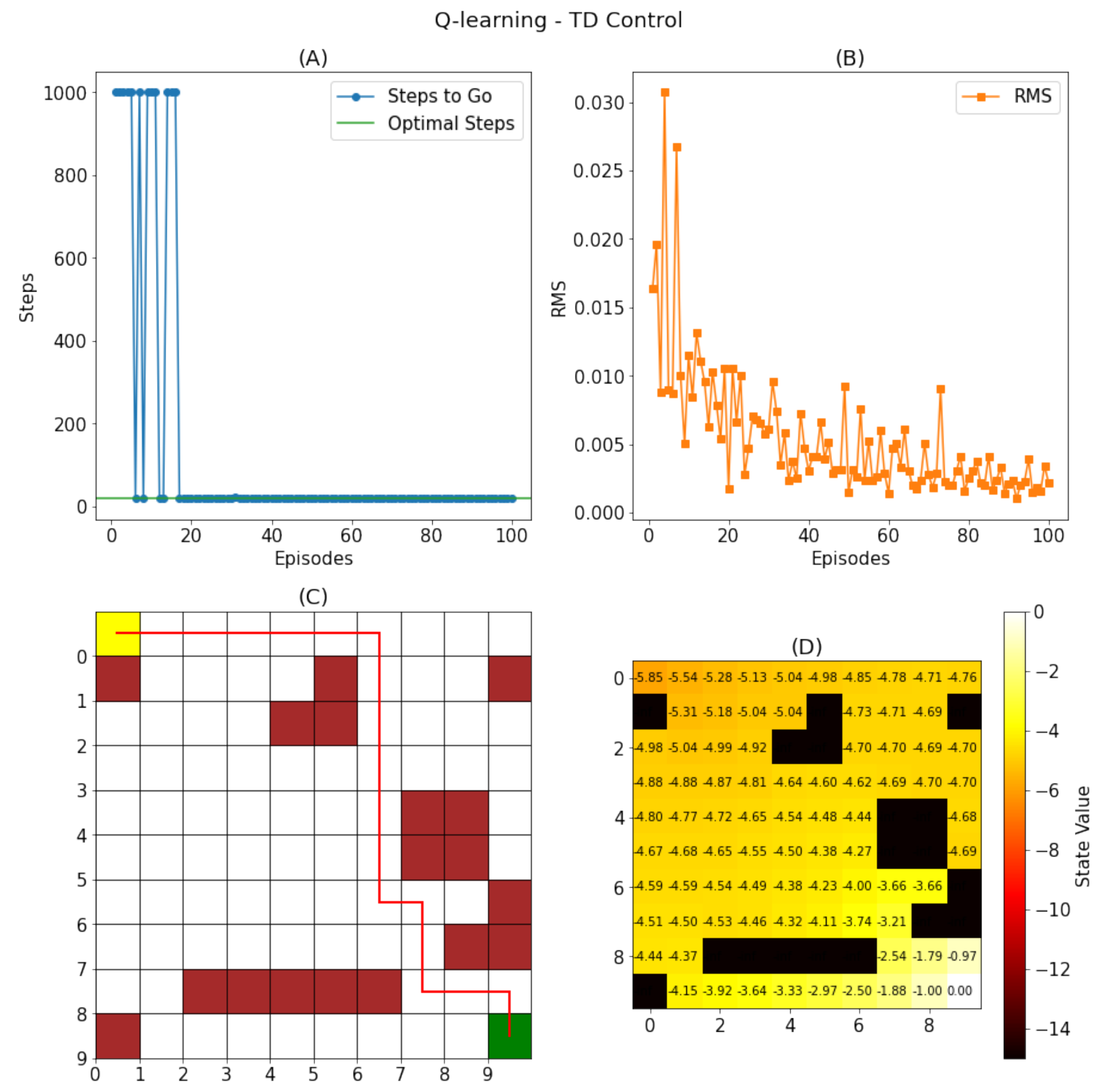
6.3 Q-learning with TD Control

- Action follow policy using ϵ-greedy
- No step restriction on episode

[1143]:
```
q_learn_td_V, q_learn_td_Q, grid, q_learn_td_pi = initialize_grid(M, N,␣
 ↪NUM_BLK, random_state=10)
rng = np.random.default_rng(42)
V_pre = np.copy(q_learn_td_V)
q_learn_td_steps = []
q_learn_td_rms = []
m, n = q_learn_td_V.shape

for _ in range(100):
    i, j = 0, 0
    while i != m - 1 or j != n - 1:  # One episode
        A = get_epsilon_greedy_action(q_learn_td_Q, i, j, grid)
        ni, nj = i + ACTIONS[A][0], j + ACTIONS[A][1]
        q_learn_td_Q[i, j, A] = q_learn_td_Q[i, j, A] + ALPHA * (-1 + GAMMA *␣
 ↪max(q_learn_td_Q[ni, nj]) - q_learn_td_Q[i, j, A])
        i, j = ni, nj

    # update V
    q_learn_td_V = np.max(q_learn_td_Q, axis=2)

    # Compute error metrics
    q_learn_td_steps.append(steps_to_go(q_learn_td_V, grid))
    q_learn_td_rms.append(np.linalg.norm(q_learn_td_V - V_pre) / (M * N))
    # update V_pre
    V_pre = np.copy(q_learn_td_V)

update_policy(q_learn_td_V, q_learn_td_pi, grid)
q_learn_td_steps = np.array(q_learn_td_steps)
q_learn_td_rms = np.array(q_learn_td_rms)

# Pickle everything
with open('q_learn_td_steps.pickle', 'wb') as f_obj:
    dill.dump(q_learn_td_steps, f_obj)
with open('q_learn_td_rms.pickle', 'wb') as f_obj:
    dill.dump(q_learn_td_rms, f_obj)
with open('q_learn_td_V.pickle', 'wb') as f_obj:
    dill.dump(q_learn_td_V, f_obj)
with open('q_learn_td_pi.pickle', 'wb') as f_obj:
    dill.dump(q_learn_td_pi, f_obj)
```

```
[1144]: with open('q_learn_td_steps.pickle', 'rb') as f_obj:
            q_learn_td_steps = dill.load(f_obj)
        with open('q_learn_td_rms.pickle', 'rb') as f_obj:
            q_learn_td_rms = dill.load(f_obj)
        with open('q_learn_td_V.pickle', 'rb') as f_obj:
            q_learn_td_V = dill.load(f_obj)
        with open('q_learn_td_pi.pickle', 'rb') as f_obj:
            q_learn_td_pi = dill.load(f_obj)

        plot(
            q_learn_td_steps,
            q_learn_td_rms,
            q_learn_td_V,
            q_learn_td_pi,
            OPTIMAL_STEPS,
            'Q-learning - TD Control',
        )
        print('First episode reach optimal steps', np.where(q_learn_td_steps ==␣
         ↪OPTIMAL_STEPS)[0][0])
        try:
            print('First episode converge', np.where(np.isclose(q_learn_td_rms, 0.
         ↪0))[0][0])
        except IndexError:
            print(f'RMS not converged yet after {len(q_learn_td_steps)} episodes')
```

```
First episode reach optimal steps 7
RMS not converged yet after 100 episodes
```



7 Eligibility Tracing

7.1 TD(λ)

- Action completely random
- No step restriction on episode
- Use accumulating traces

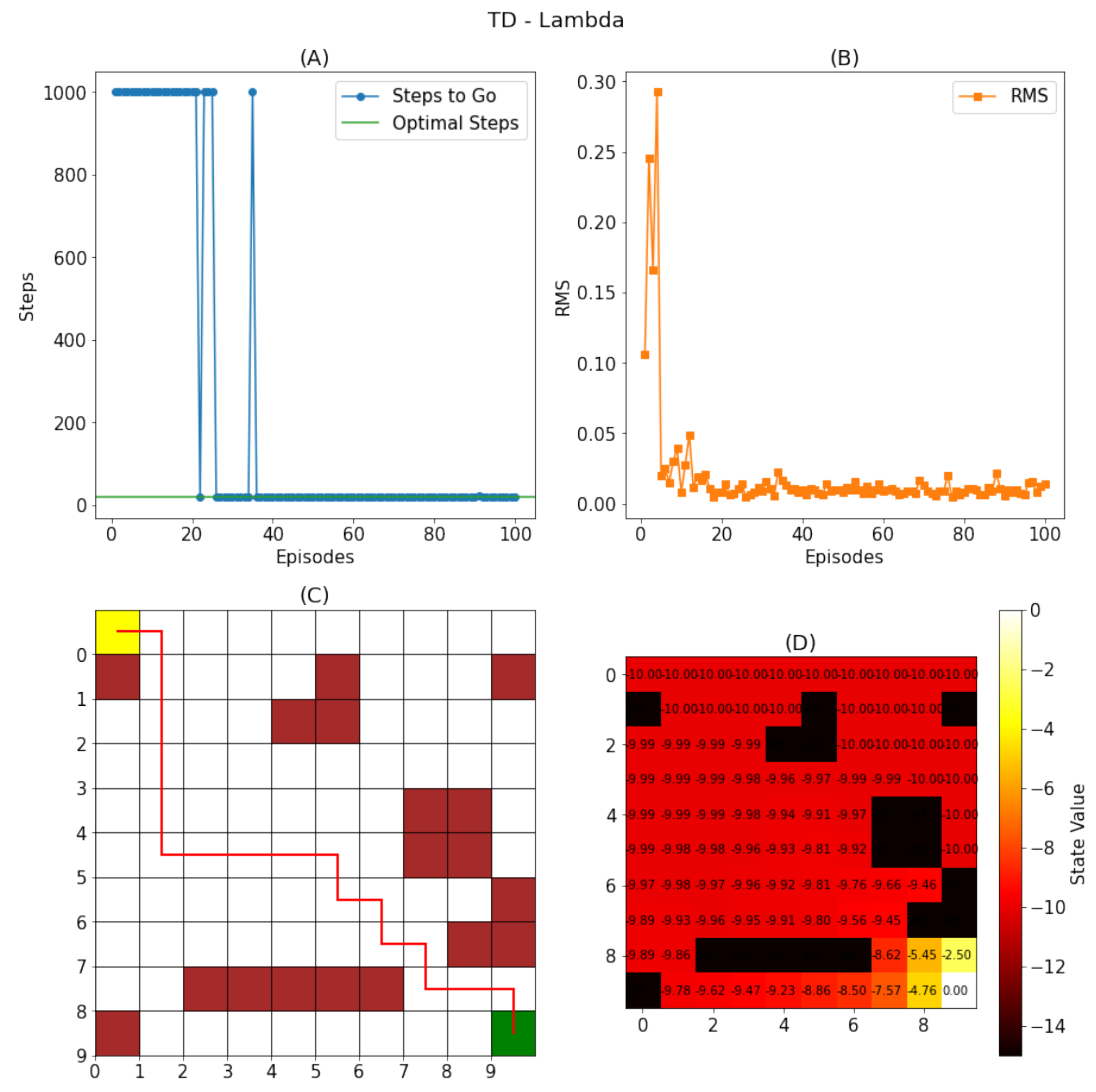
```
[1145]: td_lambda_V, _, grid, td_lambda_pi = initialize_grid(M, N, NUM_BLK,␣
         ↪random_state=10)
        rng = np.random.default_rng(42)
        V_pre = np.copy(td_lambda_V)
        td_lambda_steps = []
```

```
td_lambda_rms = []
m, n = td_lambda_V.shape

for _ in range(100):
    # Initialize eligibility tracing
    E = np.array([[0.0] * n for _ in range(m)])
    i, j = 0, 0
    while i != m - 1 or j != n - 1:  # One episode
        while True:  # take random action
            di, dj = rng.choice(ACTIONS)
            ni, nj = i + di, j + dj
            if 0 <= ni < m and 0 <= nj < n and grid[ni, nj] == 0:
                break  # valid action
        delta = -1 + GAMMA * td_lambda_V[ni, nj] - td_lambda_V[i, j]
        E[i, j] += 1
        for p in range(m):
            for q in range(n):
                # value update based on eligibility tracing
                td_lambda_V[p, q] += ALPHA * delta * E[p, q]
                # Attenuation of eligibility tracing
                E[p, q] *= LAMBDA * GAMMA
        i, j = ni, nj

    # Compute error metrics
    td_lambda_steps.append(steps_to_go(td_lambda_V, grid))
    td_lambda_rms.append(np.linalg.norm(td_lambda_V - V_pre) / (M * N))
    # update V_pre
    V_pre = np.copy(td_lambda_V)

update_policy(td_lambda_V, td_lambda_pi, grid)
td_lambda_steps = np.array(td_lambda_steps)
td_lambda_rms = np.array(td_lambda_rms)

# Pickle everything
with open('td_lambda_steps.pickle', 'wb') as f_obj:
    dill.dump(td_lambda_steps, f_obj)
with open('td_lambda_rms.pickle', 'wb') as f_obj:
    dill.dump(td_lambda_rms, f_obj)
with open('td_lambda_V.pickle', 'wb') as f_obj:
    dill.dump(td_lambda_V, f_obj)
with open('td_lambda_pi.pickle', 'wb') as f_obj:
    dill.dump(td_lambda_pi, f_obj)
```

[1151]:
```
with open('td_lambda_steps.pickle', 'rb') as f_obj:
    td_lambda_steps = dill.load(f_obj)
with open('td_lambda_rms.pickle', 'rb') as f_obj:
    td_lambda_rms = dill.load(f_obj)
```

```
with open('td_lambda_V.pickle', 'rb') as f_obj:
    td_lambda_V = dill.load(f_obj)
with open('td_lambda_pi.pickle', 'rb') as f_obj:
    td_lambda_pi = dill.load(f_obj)

plot(
    td_lambda_steps,
    td_lambda_rms,
    td_lambda_V,
    td_lambda_pi,
    OPTIMAL_STEPS,
    'TD - Lambda',
)
print('First episode reach optimal steps', np.where(td_lambda_steps ==␣
 ↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(td_lambda_rms, 0.
 ↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(td_lambda_steps)} episodes')
```

```
First episode reach optimal steps 25
RMS not converged yet after 100 episodes
```



7.2 Sarsa(λ)

- Action follow policy using ϵ-greedy
- No step restriction on episode
- Use accumulating traces

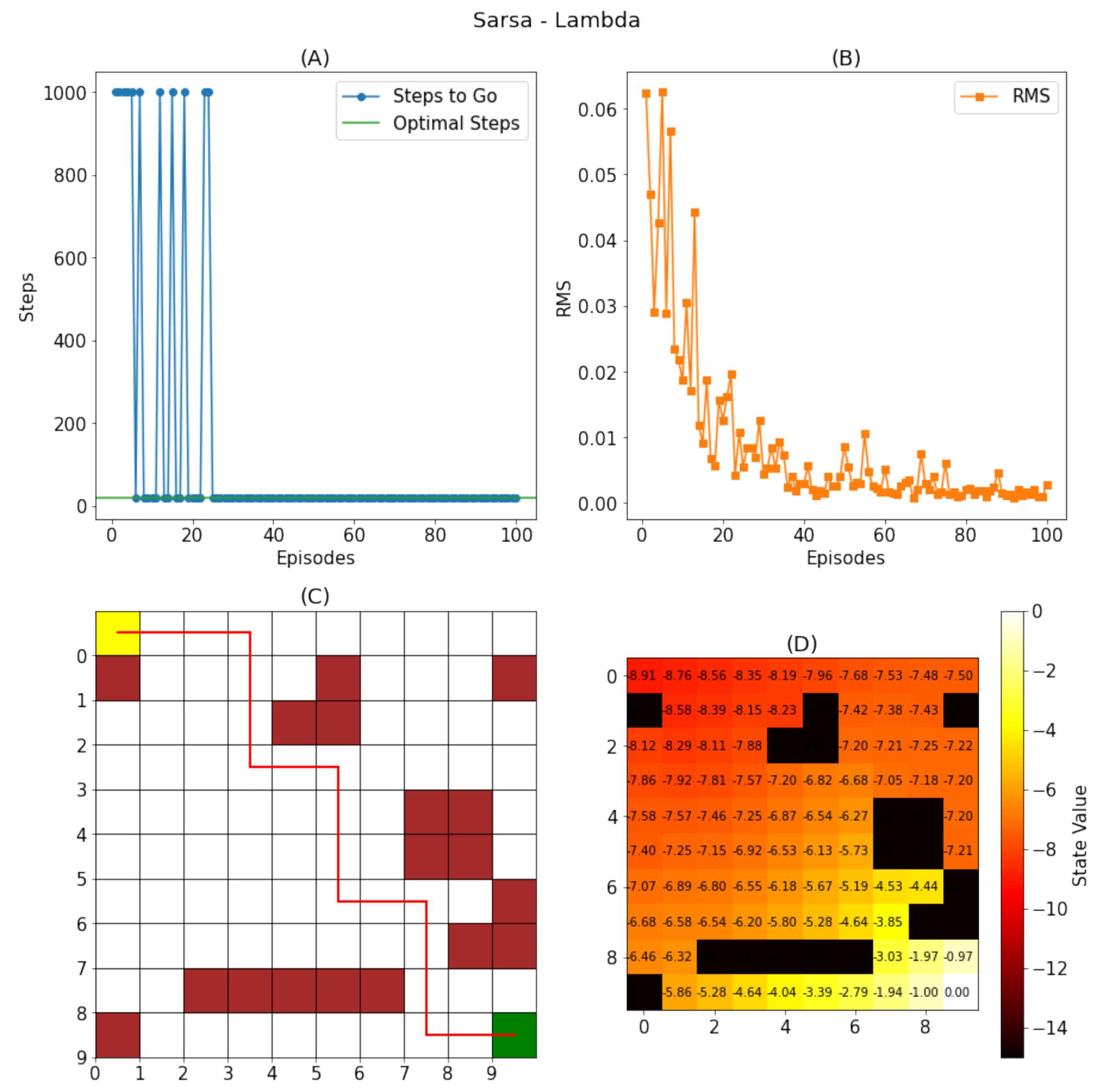
[1147]:
```
sarsa_lambda_V, sarsa_lambda_Q, grid, sarsa_lambda_pi = initialize_grid(M, N,␣
↪NUM_BLK, random_state=10)
rng = np.random.default_rng(42)
V_pre = np.copy(sarsa_lambda_V)
sarsa_lambda_steps = []
sarsa_lambda_rms = []
m, n = sarsa_lambda_V.shape
```

```
for _ in range(100):
    # Initialize eligibility tracing
    E = np.array([[[0.0] * len(ACTIONS) for _ in range(n)] for _ in range(m)])
    i, j = 0, 0
    A = get_epsilon_greedy_action(sarsa_lambda_Q, i, j, grid)
    while i != m - 1 or j != n - 1:  # One episode
        ni, nj = i + ACTIONS[A][0], j + ACTIONS[A][1]
        A_prime = get_epsilon_greedy_action(sarsa_lambda_Q, ni, nj, grid)
        delta = -1 + GAMMA * sarsa_lambda_Q[ni, nj, A_prime] -␣
 ↪sarsa_lambda_Q[i, j, A]
        E[i, j, A] += 1  # accumulating traces
        for p in range(m):
            for q in range(n):
                for k in range(len(ACTIONS)):
                    sarsa_lambda_Q[p, q, k] += ALPHA * delta * E[p, q, k]
                    E[p, q, k] *= LAMBDA * GAMMA  # attenuation
        i, j = ni, nj
        A = A_prime

    # update V
    sarsa_lambda_V = np.max(sarsa_lambda_Q, axis=2)

    # Compute error metrics
    sarsa_lambda_steps.append(steps_to_go(sarsa_lambda_V, grid))
    sarsa_lambda_rms.append(np.linalg.norm(sarsa_lambda_V - V_pre) / (M * N))
    # update V_pre
    V_pre = np.copy(sarsa_lambda_V)

update_policy(sarsa_lambda_V, sarsa_lambda_pi, grid)
sarsa_lambda_steps = np.array(sarsa_lambda_steps)
sarsa_lambda_rms = np.array(sarsa_lambda_rms)

# Pickle everything
with open('sarsa_lambda_steps.pickle', 'wb') as f_obj:
    dill.dump(sarsa_lambda_steps, f_obj)
with open('sarsa_lambda_rms.pickle', 'wb') as f_obj:
    dill.dump(sarsa_lambda_rms, f_obj)
with open('sarsa_lambda_V.pickle', 'wb') as f_obj:
    dill.dump(sarsa_lambda_V, f_obj)
with open('sarsa_lambda_pi.pickle', 'wb') as f_obj:
    dill.dump(sarsa_lambda_pi, f_obj)
```

[1148]:
```
with open('sarsa_lambda_steps.pickle', 'rb') as f_obj:
    sarsa_lambda_steps = dill.load(f_obj)
with open('sarsa_lambda_rms.pickle', 'rb') as f_obj:
    sarsa_lambda_rms = dill.load(f_obj)
```

```python
with open('sarsa_lambda_V.pickle', 'rb') as f_obj:
    sarsa_lambda_V = dill.load(f_obj)
with open('sarsa_lambda_pi.pickle', 'rb') as f_obj:
    sarsa_lambda_pi = dill.load(f_obj)

plot(
    sarsa_lambda_steps,
    sarsa_lambda_rms,
    sarsa_lambda_V,
    sarsa_lambda_pi,
    OPTIMAL_STEPS,
    'Sarsa - Lambda',
)
print('First episode reach optimal steps', np.where(sarsa_lambda_steps ==␣
 ↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(sarsa_lambda_rms, 0.
 ↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(sarsa_lambda_steps)} episodes')
```

```
First episode reach optimal steps 7
RMS not converged yet after 100 episodes
```



7.3 Watkin's Q(λ)

- Action follow policy using ϵ-greedy
- No step restriction on episode
- Use accumulating traces

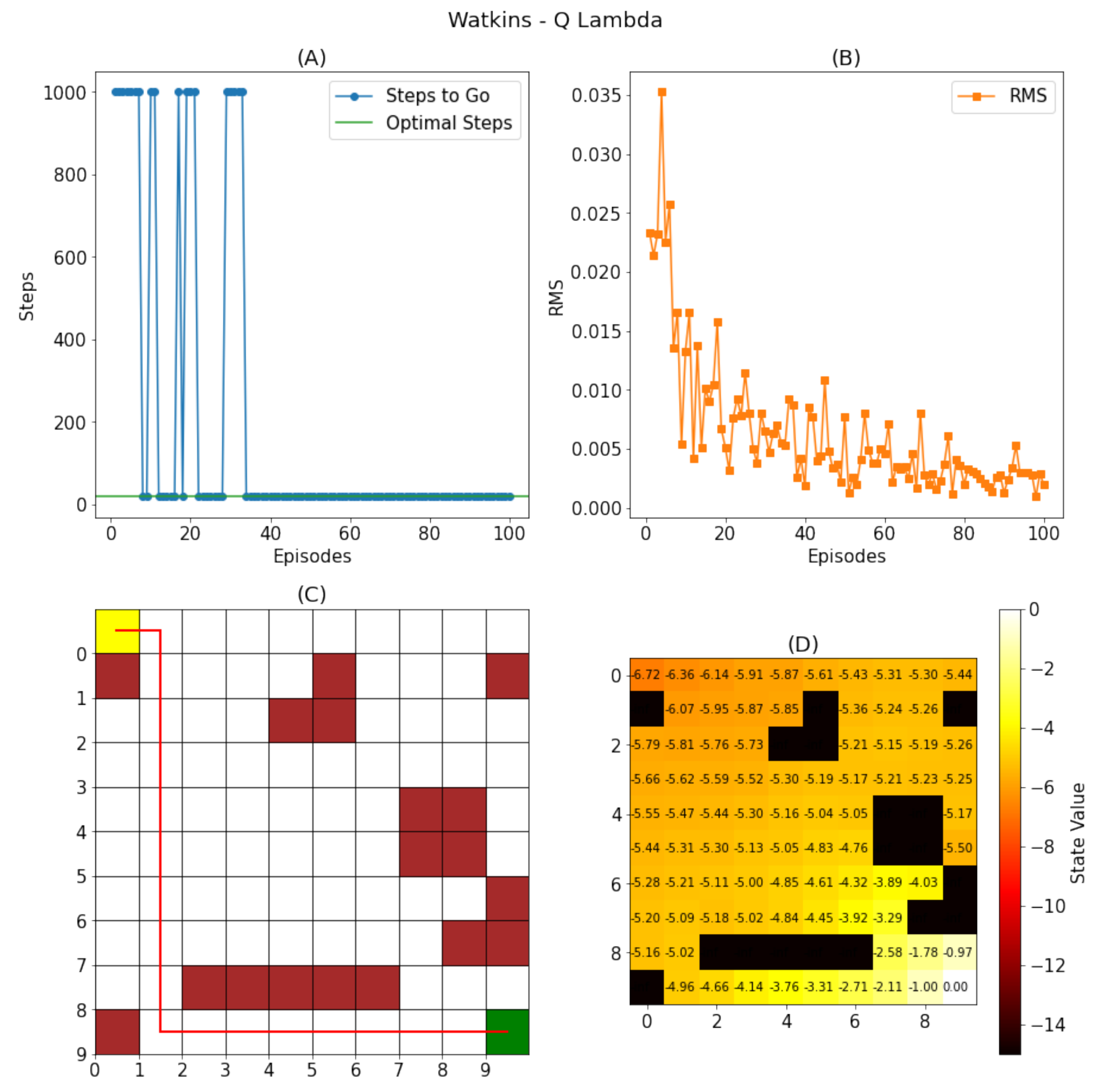
[1149]:
```
watkins_q_lambda_V, watkins_q_lambda_Q, grid, watkins_q_lambda_pi =␣
 ↪initialize_grid(M, N, NUM_BLK, random_state=10)
rng = np.random.default_rng(42)
V_pre = np.copy(watkins_q_lambda_V)
watkins_q_lambda_steps = []
watkins_q_lambda_rms = []
m, n = watkins_q_lambda_V.shape
```

```
for _ in range(100):
    # Initialize eligibility tracing
    E = np.array([[[0.0] * len(ACTIONS) for _ in range(n)] for _ in range(m)])
    i, j = 0, 0
    A = get_epsilon_greedy_action(watkins_q_lambda_Q, i, j, grid)
    while i != m - 1 or j != n - 1:  # One episode
        ni, nj = i + ACTIONS[A][0], j + ACTIONS[A][1]
        A_prime = get_epsilon_greedy_action(watkins_q_lambda_Q, ni, nj, grid)
        A_star = np.argmax(watkins_q_lambda_Q[ni, nj])
        if watkins_q_lambda_Q[ni, nj, A_prime] == watkins_q_lambda_Q[ni, nj,␣
↪A_star]:
            A_star = A_prime
        delta = -1 + GAMMA * watkins_q_lambda_Q[ni, nj, A_star] -␣
↪watkins_q_lambda_Q[i, j, A]
        E[i, j, A] += 1  # accumulating traces
        for p in range(m):
            for q in range(n):
                for k in range(len(ACTIONS)):
                    watkins_q_lambda_Q[p, q, k] += ALPHA * delta * E[p, q, k]
                    if k == A_star:
                        E[p, q, k] *= LAMBDA * GAMMA  # attenuation
                    else:  # action not greedy, thus its error shall not be␣
↪highly restricted
                        E[p, q, k] = 0
        i, j = ni, nj
        A = A_prime

    # update V
    watkins_q_lambda_V = np.max(watkins_q_lambda_Q, axis=2)

    # Compute error metrics
    watkins_q_lambda_steps.append(steps_to_go(watkins_q_lambda_V, grid))
    watkins_q_lambda_rms.append(np.linalg.norm(watkins_q_lambda_V - V_pre) / (M␣
↪* N))
    # update V_pre
    V_pre = np.copy(watkins_q_lambda_V)

update_policy(watkins_q_lambda_V, watkins_q_lambda_pi, grid)
watkins_q_lambda_steps = np.array(watkins_q_lambda_steps)
watkins_q_lambda_rms = np.array(watkins_q_lambda_rms)

# Pickle everything
with open('watkins_q_lambda_steps.pickle', 'wb') as f_obj:
    dill.dump(watkins_q_lambda_steps, f_obj)
with open('watkins_q_lambda_rms.pickle', 'wb') as f_obj:
    dill.dump(watkins_q_lambda_rms, f_obj)
```

```
with open('watkins_q_lambda_V.pickle', 'wb') as f_obj:
    dill.dump(watkins_q_lambda_V, f_obj)
with open('watkins_q_lambda_pi.pickle', 'wb') as f_obj:
    dill.dump(watkins_q_lambda_pi, f_obj)
```

[1150]:
```
with open('watkins_q_lambda_steps.pickle', 'rb') as f_obj:
    watkins_q_lambda_steps = dill.load(f_obj)
with open('watkins_q_lambda_rms.pickle', 'rb') as f_obj:
    watkins_q_lambda_rms = dill.load(f_obj)
with open('watkins_q_lambda_V.pickle', 'rb') as f_obj:
    watkins_q_lambda_V = dill.load(f_obj)
with open('watkins_q_lambda_pi.pickle', 'rb') as f_obj:
    watkins_q_lambda_pi = dill.load(f_obj)

plot(
    watkins_q_lambda_steps,
    watkins_q_lambda_rms,
    watkins_q_lambda_V,
    watkins_q_lambda_pi,
    OPTIMAL_STEPS,
    'Watkins - Q Lambda',
)
print('First episode reach optimal steps', np.where(watkins_q_lambda_steps ==␣
 ↪OPTIMAL_STEPS)[0][0])
try:
    print('First episode converge', np.where(np.isclose(watkins_q_lambda_rms, 0.
 ↪0))[0][0])
except IndexError:
    print(f'RMS not converged yet after {len(watkins_q_lambda_steps)} episodes')
```

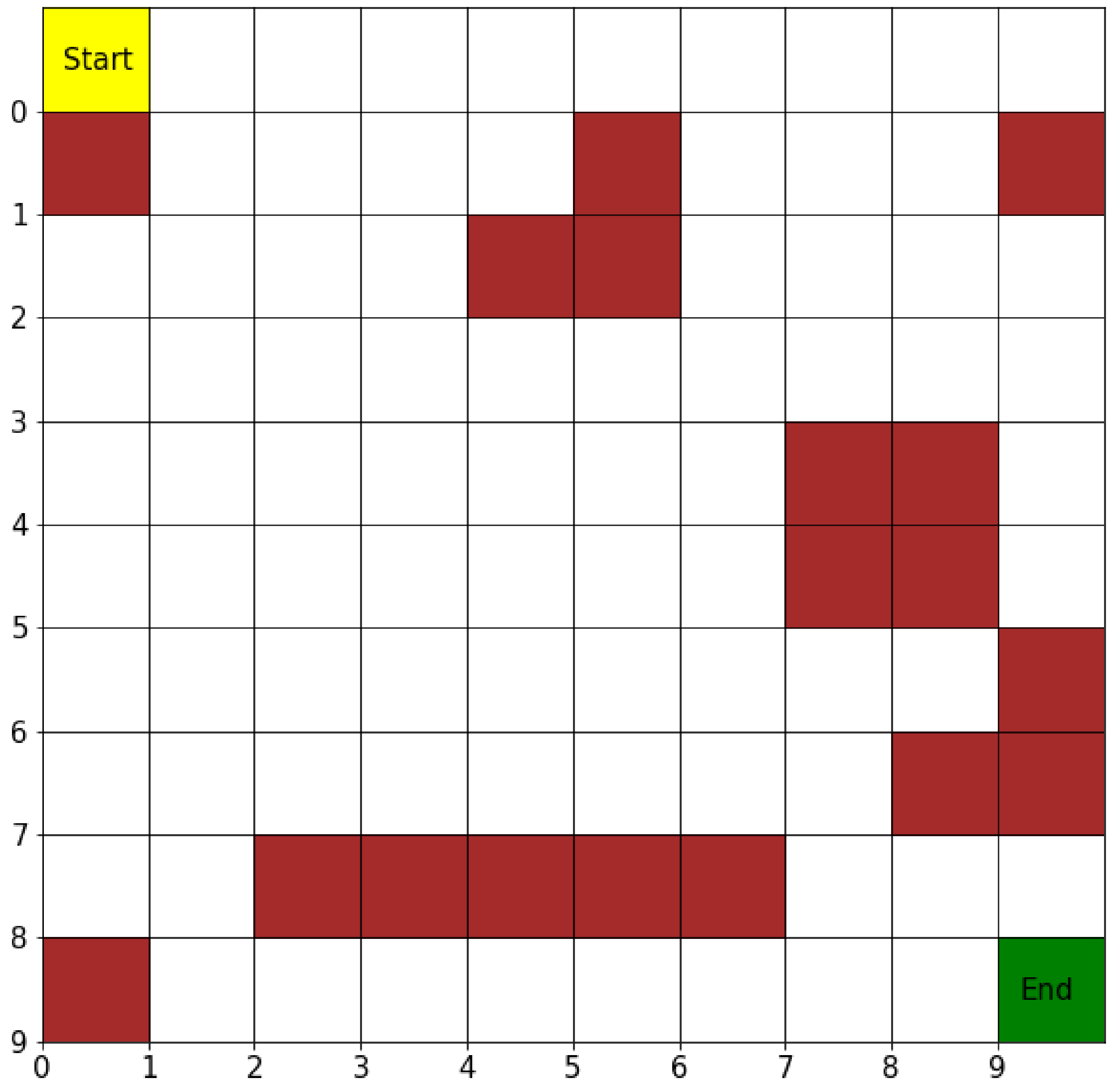
```
First episode reach optimal steps 7
RMS not converged yet after 100 episodes
```



8 Draw Grid World without Path

[1119]:
```python
_, _, grid, _ = initialize_grid(M, N, NUM_BLK, random_state=10)
fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.grid(True)
ax.set_xticks(np.arange(N))
ax.set_xlim(left=0, right=N)
ax.set_yticks(np.arange(M))
ax.set_ylim(top=M - 1, bottom=-1)
ax.invert_yaxis()  # invert y axis such that 0 is at the top

# label start, end and blocks
```

```
ax.add_patch(Rectangle((0, -1), 1, 1, fill=True, color='yellow'))
ax.annotate(xy=(0.2, -0.4), text='Start')
ax.add_patch(Rectangle((n - 1, m - 2), 1, 1, fill=True, color='green'))
ax.annotate(xy=(n - 1 + 0.2, m - 1.4), text='End')
for i, j in zip(*np.where(grid == 1)):
    ax.add_patch(Rectangle((j, i - 1), 1, 1, fill=True, color='brown'))

plt.savefig('grid_world.pdf')
```



```
[ ]:
```