Parinaz Shiri (202381576), Himani Thakkar (202292672)
Reinforcement Learning (DSCI 6650)
August 10, 2024

Project 3

Part 1

Problem Description & Visualization – We created a canvas using tkinter of size 5, for the grid world. Then we defined the environment in the following manner: The agent starts at the blue state (4,0). Certain cells are designated as red states with a high negative reward of -20 to simulate obstacles or walls. When the agent enters those states it returns to the start state which is (4,0) as a penalty to start from scratch. These states are (2, 0), (2, 1), (2, 3), and (2, 4). The black states are terminal states with no rewards, located at (0, 0) and (0, 4). All other cells are normal states with a reward of -1 for each move to simulate a cost for each step taken. The reward for stepping outside the grid would be -1 and the agent remains in the same state The goal of the problem is to pass through the opening between the "red" states and reach one of the terminal states.

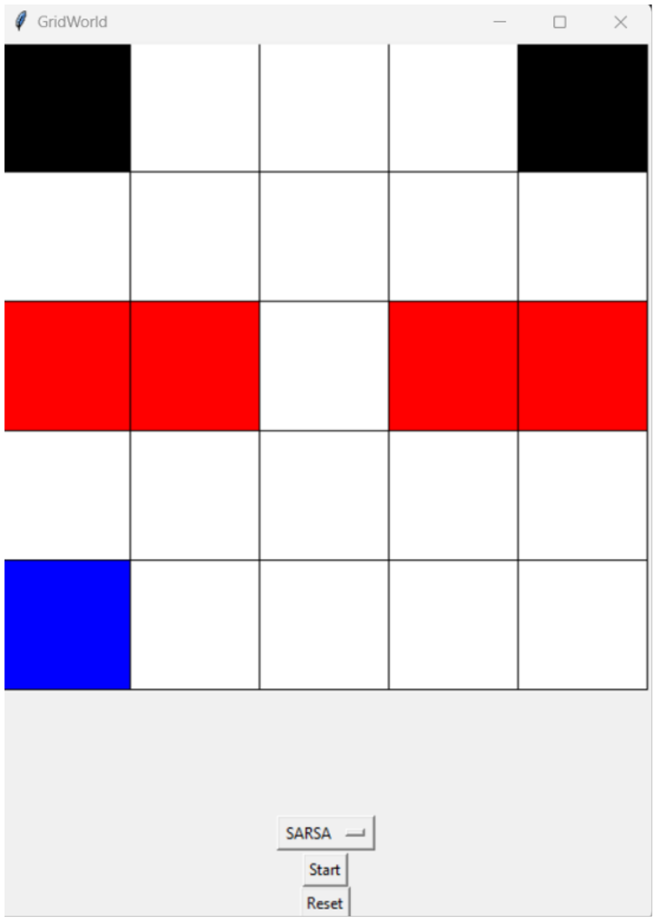


FIGURE 1. Grid Environment

The user can select SARSA or Q-learning from the drop-down button and press start to run the algorithm. To run another simulation, the user can use the reset buttons to reset all the values. The output would be an optimal policy using SARSA or Q-learning.

Introduction

Temporal-Difference (TD) Learning combines elements of Monte Carlo methods and Dynamic Programming. TD methods learn directly from experience without requiring a model of the environment (a feature borrowed from Monte Carlo methods). Unlike Monte Carlo methods, which wait until the end of an episode to update values, TD methods incorporate other learned estimates during the learning process through bootstrapping, a concept from Dynamic Programming. This allows TD methods to predict the value of a policy π more efficiently by updating estimates based on other estimates rather than waiting for complete episodes.

TD Learning includes various approaches, such as TD(0) and TD(λ), each with its own characteristics and use cases. For example, TD(0) provides immediate updates, while TD(λ) allows for a more nuanced integration of past experiences. TD Learning is particularly useful in environments where it is impractical to wait for episodes to complete or when a model of the environment is not available. By combining the strengths of Monte Carlo and Dynamic Programming, TD methods offer a powerful tool for learning and decision-making in reinforcement learning contexts.

SARSA (state-action-reward-state-action)

It is an on-policy reinforcement learning algorithm where, in the current state (S_t), an action (A_t) is taken and the agent gets a reward (R_{t+1}) and ends up in the next state (S_{t+1}) and takes action (A_{t+1}) in S_{t+1}. Therefore, the tuple (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}) stands for the acronym SARSA. SARSA is an on-policy TD control method, which starts with action-value pairs rather than a state-value function. We use epsilon-greedy to keep a balance between exploration and exploitation, which ensures that exploration is always happening. The choose_action function uses an epsilon of 0.01 for choosing a random action.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

From here, the selected action is taken, and the reward (R) and the next state (S1) are observed. $Q(S, A)$ gets updated and the next action $A1$, based on the updated Q-value.

SARSA updates the Q value of a given (s, a) combination, using the instantaneous rewards that the agent receives in any step and the Q value of the resulting state-action pair, i.e., (s , a).

Action-value estimates of a state are also updated for each current action-state pair present, which estimates the value of receiving a reward for taking a given action. This is repeated, until the black terminal state is reached, and the episode ends.

Implementation & Parameters

We ran the function for 10,000 episodes with a learning rate of 0.1, a discount factor of 0.96, and epsilon 0.01. It returns a list of total rewards obtained in each episode.

The next state and reward are obtained by taking a step from the current state using the current action. The next action is chosen for the next_state using the `choose_action` method. Q(S,A)[q_sa] defines the Q value for a state-action pair, and the Q(S',A') [q_s_a] is the Q value for the next state-action pair. After reaching the terminal state and exiting out of the loop, the total reward is appended to the SARSA rewards for all episodes.

Output The output displays an optimal policy, which starts from the start state and includes all states having actions that would lead through the state between the red states, reaching either of the two terminal states.

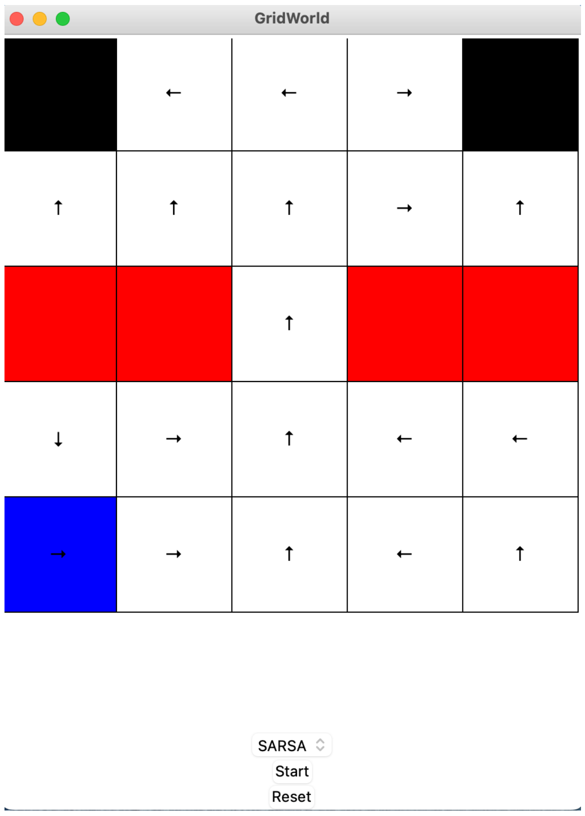


FIGURE 2. Sarsa Optimal Policy

The two states right next to and below the terminal states would point to the black state closest to that state, i.e., (1,4) would lead to (0,4) and (1,0) would lead to (0,0) terminal state. The state in the middle of terminal states (0,2) usually always changes directions with each run depending on actions chosen during the episodes, but in the end having a result that would lead to the terminal state.

We also accumulated the rewards calculated for 10,000 episodes and then created a plot to display the rewards accumulated throughout the episodes.

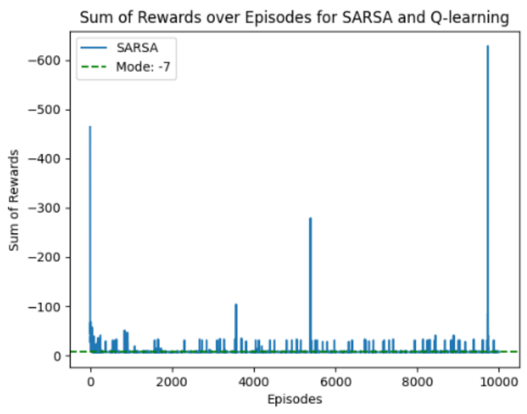


FIGURE 3. SARSA Accumulative Reward Plot

The cumulative rewards are higher at first episodes but by updating the policy according to the previous episodes the reward is converging more to the optimal path which has the reward -7 according to the environment setup. It is observable that after a while most episodes result in an accumulative reward of -7. There were instances where the reward accumulation was higher for an episode, but that is because of the negative rewards due to certain actions and states. Furthermore, the other reason is the fact that the algorithm explores 1% of the time which concludes in a large negative reward in some episodes and even in last episode generations for instance in episode range of 9000. However, towards the last episode and for most of the episodes, the accumulated reward for the episode would be -7.

Moreover, on running the algorithm on 10,000 and 100,000 and 1,000,000 episodes, we noticed the same trend. The total time taken for the algorithm to run on 1,000 episodes was between 0.2-0.3 seconds and on 1,000,000 episodes it was 25-35 seconds.

ADDITIONAL - When we removed the value for epsilon, in a way that it always chooses a greedy action and never explores, the algorithm accumulated the highest reward (-7) in most cases, and was able to find the optimal policy.

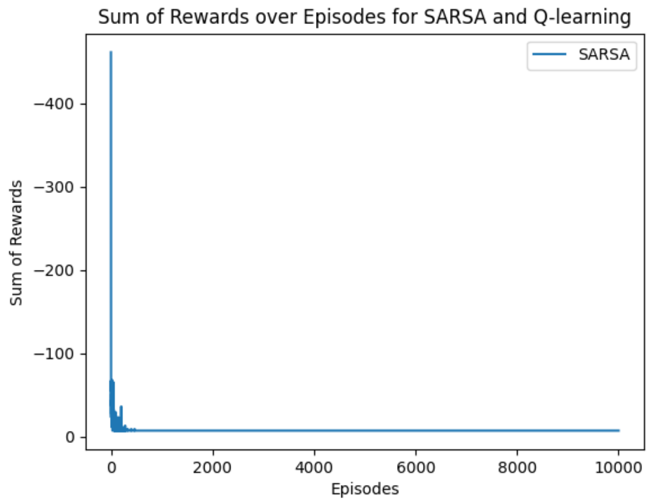


FIGURE 4. $\epsilon = 0$

However, this is good in this environment and scenario, but in more complex environments, the algorithm will never explore and just exploit, any may not be able to gain the highest rewards.

Q-learning

Temporal-Difference Learning combines the idea of Monte Carlo and Dynamic Programming. TD methods learn directly from experience without a model of the environment (Monte Carlo feature). TD method update incorporates other learned estimates without waiting for the final outcome of an episode (bootstrapping, which is the Dynamic Programming feature). TD methods use experience to predict the value of a policy π.

Q-learning is an off-policy TD control algorithm. Off-policy methods learn a different policy than the one that is used to generate actions. This means that episodes will be generated with random actions and the policy we form does not guide episode generation. The learned Q directly approximates q^*, the optimal action-value function, no matter which behavior policy is being followed. Q-learning learns the value of a policy which does not explore, since it takes the max action value.

In the implementation of the Q-Learning algorithm, we generate 10,000 episodes starting from the start state, which is the state (4,0). We initialize the total reward to 0 at each episode and then proceed with the following steps:

1) **Choosing an Action:** For each state, an action is chosen using the ϵ-greedy policy defined in the `choose_action` function. This ensures that there is a balance between exploration and exploitation.

The probability of selecting a random action (exploration) is controlled by ϵ (epsilon), which is set to 0.01. This means there is a 1% chance of choosing a random action and a 99% chance of choosing the best-known action.

2) **Taking a Step:** The chosen action is then executed, and the agent transitions to the next state. The environment provides a reward based on this transition, and the next state is determined.
3) **Updating Q-values:** The Q-value for the current state-action pair is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

Here, α is the learning rate, γ is the discount factor, R_{t+1} is the reward received, S_t is the current state, A_t is the current action, S_{t+1} is the next state, and $\max Q(S_{t+1}, a)$ is the maximum Q-value for the next state across all possible actions.

4) **Accumulating Rewards:** The reward received at each step is accumulated to compute the total reward for the episode.
5) **Episode Termination:** The state will be updated to the next state and before choosing the next action to take the algorithm checks if the new state is equals to the terminal states it is going to be terminate.
6) **Recording Results:** The total reward for each episode is recorded to analyze the performance and learning progress of the agent. The results will also be plotted for comparison.

Output

The output displays an optimal policy, which starts from the start state and includes all states having actions that would lead through the state between the red states, reaching either of the two terminal states.

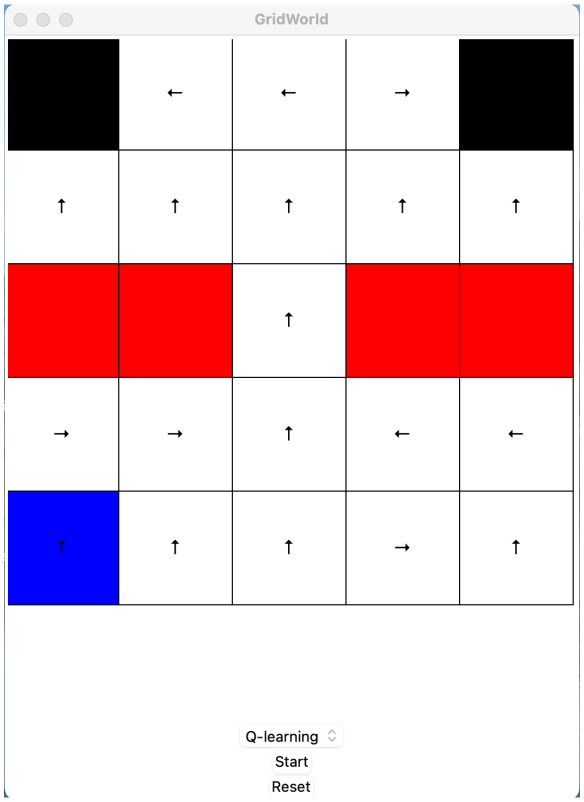


Figure 5. Q-Learning Optimal Policy

optimal policy provided by Q-learning algorithm, walks through the walls with more confidence and guides the agent to the terminal state. Through multiple runs of the Q-Learning algorithm, it has been observed that the direction of the optimal policy for reaching the terminal state alternates between moving left and moving right at state(0,2). This behavior occurs because from this state, transitioning to either the left terminal state or the right terminal state results in the same cumulative reward. As a result, the agent

may sometimes choose to go left and other times right, leading to an optimal policy that varies in direction across different runs.
We also accumulated the rewards calculated for 10,000 episodes and the created a plot to display the rewards accumulated throughout the episodes.

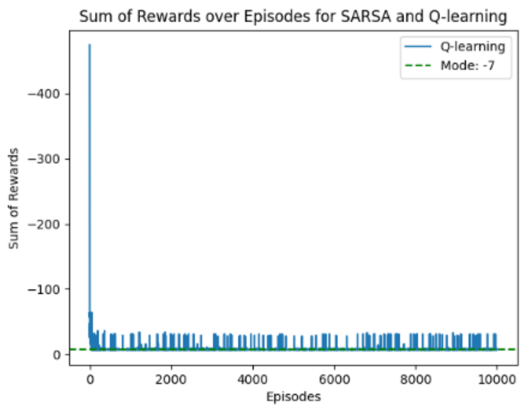


Figure 6. Q-Learning Accumulative Reward Plot

This plot indicates the sum of rewards over episodes for the Q-learning algorithm. Initially, there is a significant fluctuation in rewards, indicating the exploratory phase of the agent where it randomly tries different actions to learn about the environment. As the number of episodes increases, the sum of rewards gradually converges, reflecting the agent's learning and adaptation towards an optimal policy. By approximately 2,000 episodes, the rewards stabilize around a value close to -7, consistent with the setup of the environment where the optimal path yields a cumulative reward of -7. Occasional spikes or deviations in the reward are observed due to the epsilon-greedy policy used in Q-learning, which ensures a 1% chance of selecting a random action to maintain exploration. These exploratory actions sometimes lead to sub optimal rewards but are crucial for ensuring that the agent does not get stuck in local optima. Overall, the plot demonstrates the convergence of Q-learning towards an optimal policy, balancing exploration and exploitation, and consistently achieving near-optimal rewards as the training progresses.

Similarity and Difference among the Accumulative Reward in SARSA & Q-Learning.

Similarities

Both algorithms exhibit a high degree of variability in the rewards at the beginning of the training process. This is due to the exploratory nature of the ϵ-greedy policy, which introduces randomness in the action selection. Both algorithms show a trend toward stabilization of cumulative rewards as the number of episodes increases. This suggests that both algorithms are learning to navigate the environment more effectively over time.Both methods are sensitive to the learning rate α. A high learning rate can cause larger fluctuations in the sum of rewards for both methods, especially in SARSA, where on-policy updates mean that poor exploration choices can have a more immediate impact on the learned policy

Differences

The Q-learning plot demonstrates less fluctuation in cumulative rewards compared to the SARSA plot. Q-learning tends to exhibit more consistent convergence towards the optimal policy, as it always exploits the best-known action regardless of the exploration policy (off-policy). SARSA, being an on-policy algorithm, incorporates the exploration directly into its updates, leading to more variation in rewards as it adheres strictly to the -greedy strategy.

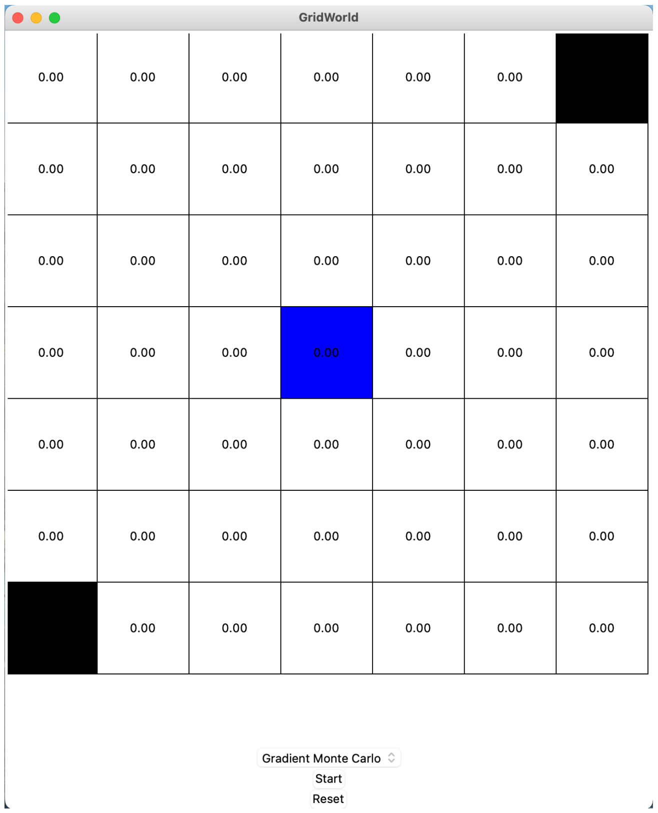
How does the sum of rewards over an episode behave for each of these two methods?

Q-Learning: The sum of rewards for Q-learning appears to stabilize relatively quickly and remains consistent throughout the episodes. This behavior indicates that Q-learning effectively finds and adheres to a potentially optimal policy early on. Its updates, based on the maximum possible future reward regardless of the agent's actual policy, allow it to consistently perform near the optimal level once it has sufficiently explored the state-action space.
SARSA: In contrast, the SARSA plot shows more significant fluctuations in the sum of rewards over episodes. This variation is characteristic of SARSA's on-policy nature, where the policy being evaluated and improved upon is the same policy used to make decisions, including the -greedy explorations. Therefore, SARSA's performance can vary more significantly from episode to episode depending on the balance between exploration and exploitation.

Part 2

Problem Description & Visualization – We created a canvas using tkinter of size 7, for the grid world. Then we defined the environment in the following manner: The agent starts at the blue state (3,3). The black states are terminal states with reward of 1 for upper right corner and reward of -1 for the lower left, located at (0, 6) and (6, 0). All other cells are normal states with a reward of 0. The reward for stepping outside the grid would be 0 and the agent remains in the same state.

The goal of the problem is to approximate the value function for all states.



0.00	0.00	0.00	0.00	0.00	0.00	⬛
0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	🟦 0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00
⬛	0.00	0.00	0.00	0.00	0.00	0.00

Figure 7. Grid Environment

The user can select Gradient Monte Carlo, Semi-Gradient TD(0) and exact value function (using solving bellman equation) from the drop-down button and press start to run the algorithm. To run another simulation, the user can use the reset buttons to reset all the values.

Gradient Monte-Carlo method

Gradient Monte Carlo methods are a class of algorithms in reinforcement learning that aim to estimate the value function by directly optimizing a parameterized function using gradient ascent. Unlike traditional Monte Carlo methods, which estimate the value function using the average of sampled returns, Gradient Monte Carlo methods use function approximation techniques to represent the value function, allowing for more generalize and scalable solutions.
key formula for the Gradient Monte Carlo algorithm is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(G_t - \hat{v}(\mathbf{S_t}, \mathbf{w}))\nabla\hat{v}(\mathbf{S_t}, \mathbf{w})$$

Where:

- $\mathbf{w}$ are the weights of the linear function approximator.
- α is the learning rate.
- G_t is the return (cumulative reward) from time step t.
- $\hat{v}(\mathbf{S_t}, \mathbf{w})$ is the estimated value function for state S_t, given by the dot product $\mathbf{w} \cdot \mathbf{S_t}$.
- $\nabla\hat{v}(\mathbf{S_t}, \mathbf{w})$ is the gradient of the estimated value function with respect to the weights $\mathbf{w}$.

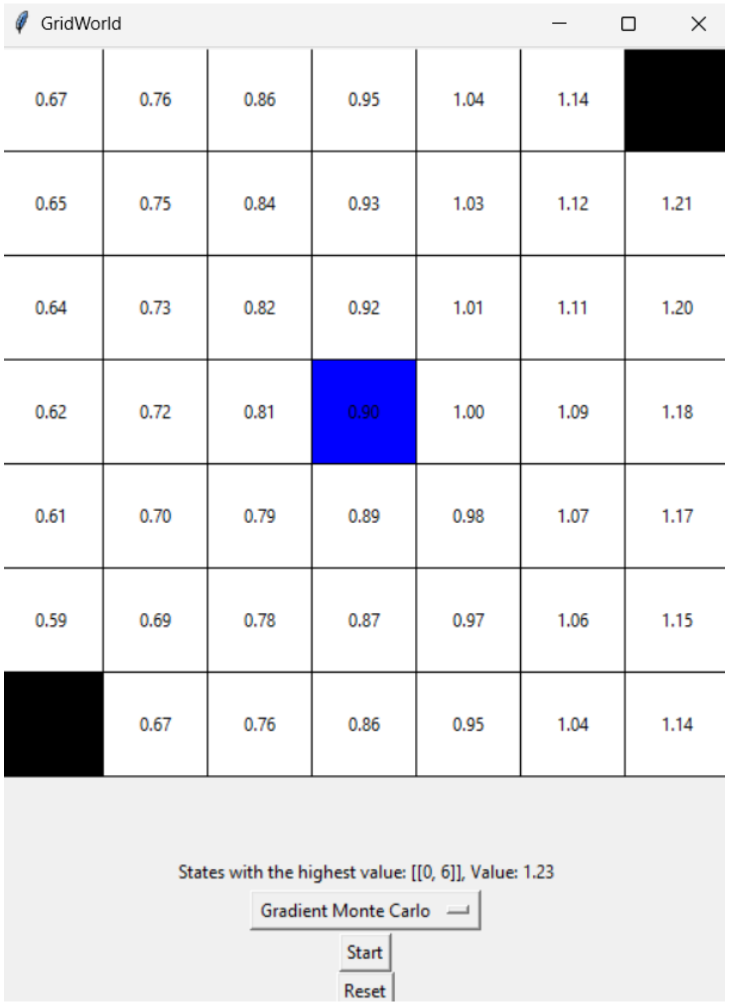
Explanation.

- **Weights $\mathbf{w}$**: These are the parameters of the linear function approximator that we are trying to optimize. The goal is to adjust these weights to minimize the difference between the predicted value and the actual return.
- **Learning Rate α**: This controls the step size of the gradient ascent updates. A smaller learning rate makes the updates more gradual, while a larger learning rate makes the updates more aggressive.
- **Return G_t**: This is the cumulative reward received from time step t to the end of the episode. It represents the total reward that the agent expects to receive, starting from state S_t.
- **Estimated Value Function $\hat{v}(\mathbf{S_t}, \mathbf{w})$: This is the predicted value of state S_t given the current weights $\mathbf{w}$**. It is computed as the dot product of the weight vector and the feature representation of the state.
- **Gradient $\nabla\hat{v}(\mathbf{S_t}, \mathbf{w})$**: This is the gradient of the estimated value function with respect to the weights. In the case of a linear function approximator, this gradient is simply the feature vector representing the state.

The way the algorithm works is that for every episode, we generate an episode (state, action, reward) until time T, using policy π, and for each step of the episode, we update the weights based on the update rule. In our implementation of the algorithm, after defining the random walk environment and the rewards, we ran the simulation for 10,000 episodes with a learning rate of 0.01. We used a feature vector having the state coordinates i and j, and the Manhattan distances with respect to the terminal states and initialized the weights to 0.

By using a feature vector, we were able to generalize the value of the function across different states. Instead of learning a separate value for each states, the algorithm could learn weights that apply to the features of the states. Using Manhattan distances, adds relevant information about the state. The ones closer to the goal might have higher values, Manhattan distance is especially useful in path finding tasks. Another thing we updated in this environment set up is the display of the output, for every 100th episode, the values displayed on the grid would change, so as to give us a better visualization on how the algorithm works and updates values.

Output: On executing the algorithm, the values of states are displayed on the grid, with the highest value and state being displayed as well. We also added a heatmap to visualize the results better.



0.67	0.76	0.86	0.95	1.04	1.14	
0.65	0.75	0.84	0.93	1.03	1.12	1.21
0.64	0.73	0.82	0.92	1.01	1.11	1.20
0.62	0.72	0.81	0.90	1.00	1.09	1.18
0.61	0.70	0.79	0.89	0.98	1.07	1.17
0.59	0.69	0.78	0.87	0.97	1.06	1.15
	0.67	0.76	0.86	0.95	1.04	1.14

States with the highest value: [[0, 6]], Value: 1.23

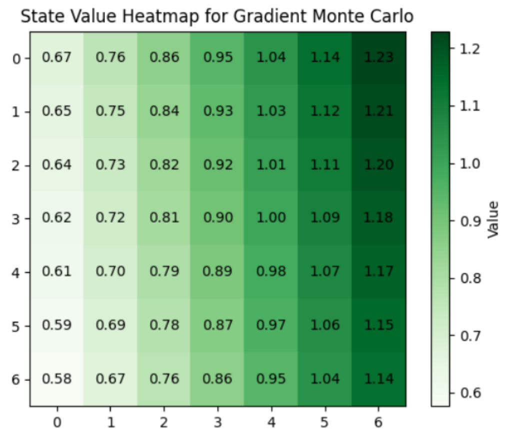


FIGURE 8. Gradient Monte Carlo

The states on the upper right have the highest values, compared to the weights on the lower left. This is due to the fact the reward at (0,6) is +1 and at (6,0) is -1. The algorithm is sensitive to learning rates, the value of weights initialized. The values of the state change with respect to all these parameters, however it is always the upper right states having the highest values and lower left having comparatively, a less value.

Semi gradient TD(0)

It is an extension of the temporal-difference learning TD(0) algorithm, and it combines the aspect of both temporal difference learning and function approximation. The key idea here, is to update the weight vector w, based on the TD error.
The term "semi-gradient" is used because the gradient is taken only with respect to the current state's value function approximation and not the target value which involves the value of the next state. The weight update rule is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}) \right] \nabla \hat{v}(S, \mathbf{w})$$

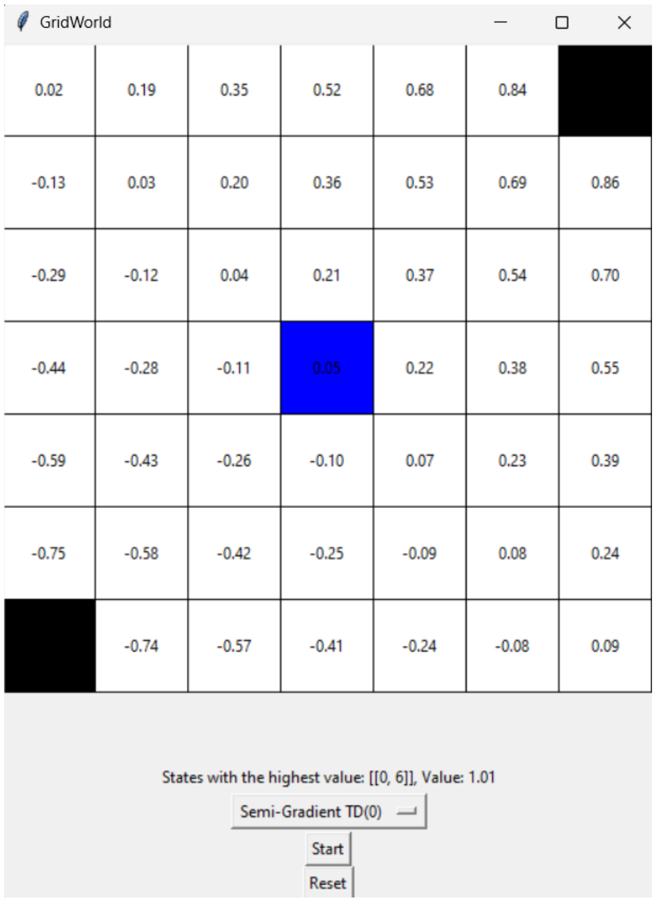
Where:

- $\mathbf{w}$ are the weights of the linear function approximator.
- α is the learning rate.
- R is the return from time step t.
- γ: Discount factor

- $\hat{v}(\mathbf{S}, \mathbf{w})$ is the estimated value function for state S, given weights $\mathbf{w}$
- $\hat{v}(\mathbf{S'}, \mathbf{w})$ is the estimated value function for state S', given weights $\mathbf{w}$
- $\nabla\hat{v}(\mathbf{S}, \mathbf{w})$ is the gradient of the estimated value function with respect to the weights $\mathbf{w}$.

This algorithm is computationally efficient, as the weights are updated incrementally, and it uses information from a single transition. In our implementation of the algorithm, we used a discount factor of 0.95, a learning rate of 0.001. The weight vector was initialized with 0 and then updated based on the rule mentioned above.
The function iterates through multiple episodes to refine the value function. Each episode represents a full trajectory starting from the start state.
During each state transition, the agent first selects an action based on the current state using a predefined policy. It then takes this action, leading to a transition to the next state and receiving an associated reward. To estimate the value of both the current and next state, the agent computes feature vectors that represent the states. These feature vectors are then used to approximate the value functions via a dot product with the weight vector.
The agent updates this weight vector by calculating the TD error, which is the difference between the predicted value of the current state and the actual value derived from the received reward and the discounted value of the next state. This update adjusts the weights to reduce future errors in value prediction, thus incrementally improving the value function approximation throughout the episode.

Output



0.02	0.19	0.35	0.52	0.68	0.84	
-0.13	0.03	0.20	0.36	0.53	0.69	0.86
-0.29	-0.12	0.04	0.21	0.37	0.54	0.70
-0.44	-0.28	-0.11	0.05	0.22	0.38	0.55
-0.59	-0.43	-0.26	-0.10	0.07	0.23	0.39
-0.75	-0.58	-0.42	-0.25	-0.09	0.08	0.24
	-0.74	-0.57	-0.41	-0.24	-0.08	0.09

States with the highest value: [[0, 6]], Value: 1.01

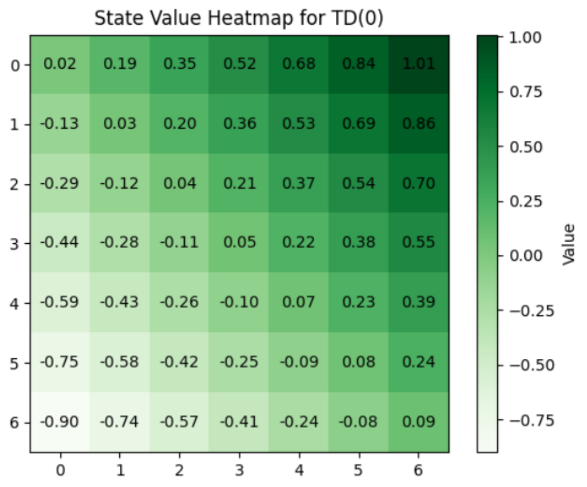


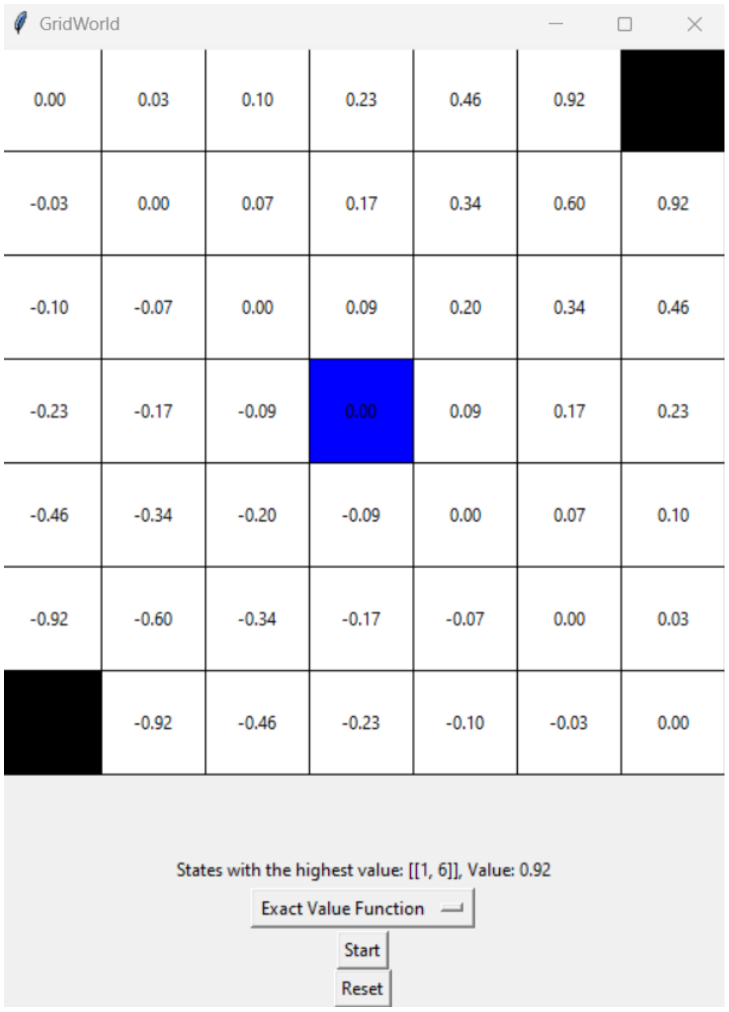
FIGURE 9. Semi-Gradient TD(0)

Looking at the output of the algorithm, it is evident that the states closer to the terminal state having a reward +1 have a higher value compared to the lower left terminal state. As the states move towards the terminal state with higher reward, the value increases. The output it similar to output generated by the gradient monte carlo.

DIFFERENCES. The difference in performance between the Gradient Monte Carlo (GMC) method and the Semi-Gradient TD(0) method primarily stems from how they handle the trade-off between bias and variance in value function approximation. GMC updates the value function by averaging over full episodes, leading to unbiased but high-variance estimates since it waits until the end of each episode to learn. This can result in slower learning, in environments with long episodes (tested with 100,000 episodes). In contrast, Semi-Gradient TD(0) updates the value function after each step within an episode, incorporating immediate feedback and allowing for faster, incremental learning. However, this introduces some bias because it relies on estimates of future values, but the reduced variance generally leads to more stable and faster convergence. As a result, Semi-Gradient TD(0) often outperforms GMC in terms of learning speed and efficiency, especially in environments where timely updates are crucial.

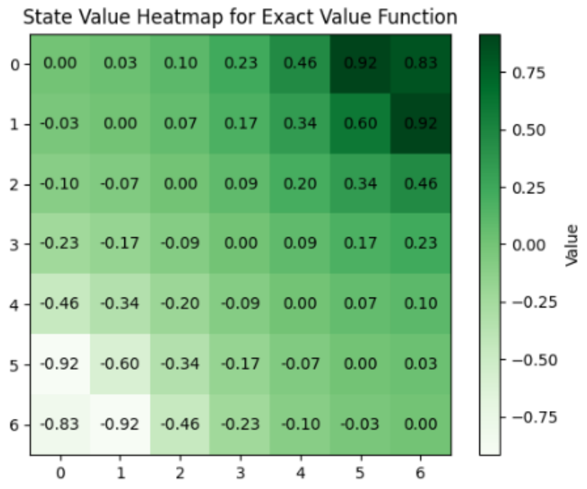
Comparing with exact value function. For the same grid world environments and conditions, using Bellman Equation Explicitly displays a similar output compared to the other two methods, where the states on upper right having higher values compared to states on the lower end of the grid. On executing the algorithm, the exact value function of the states is displayed in the grid format. Additionally, a heatmap provides a clearer visual representation of the value function distribution. However, the main difference is in the convergence speed. The exact value function converges more quickly than any other method.

Output:



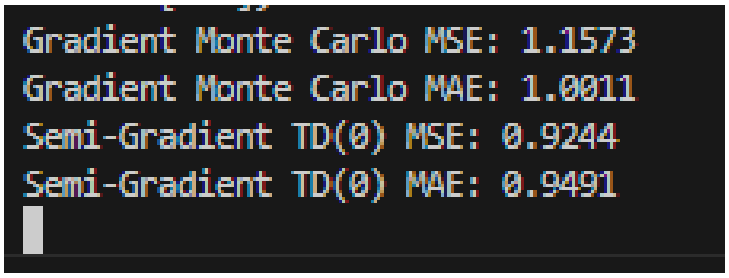
0.00	0.03	0.10	0.23	0.46	0.92	■
-0.03	0.00	0.07	0.17	0.34	0.60	0.92
-0.10	-0.07	0.00	0.09	0.20	0.34	0.46
-0.23	-0.17	-0.09	0.00	0.09	0.17	0.23
-0.46	-0.34	-0.20	-0.09	0.00	0.07	0.10
-0.92	-0.60	-0.34	-0.17	-0.07	0.00	0.03
■	-0.92	-0.46	-0.23	-0.10	-0.03	0.00

States with the highest value: [[1, 6]], Value: 0.92



	0	1	2	3	4	5	6
0	0.00	0.03	0.10	0.23	0.46	0.92	0.83
1	-0.03	0.00	0.07	0.17	0.34	0.60	0.92
2	-0.10	-0.07	0.00	0.09	0.20	0.34	0.46
3	-0.23	-0.17	-0.09	0.00	0.09	0.17	0.23
4	-0.46	-0.34	-0.20	-0.09	0.00	0.07	0.10
5	-0.92	-0.60	-0.34	-0.17	-0.07	0.00	0.03
6	-0.83	-0.92	-0.46	-0.23	-0.10	-0.03	0.00

FIGURE 10. Exact Value Function



```
Gradient Monte Carlo MSE: 1.1573
Gradient Monte Carlo MAE: 1.0011
Semi-Gradient TD(0) MSE: 0.9244
Semi-Gradient TD(0) MAE: 0.9491
```

Figure 11. Comparision using MSE and MAE

Conclusion. Figure 10 depicts the values by a true value function for this environment. Looking at the heatmap for approximate value function generated by the Gradient Monte Carlo Method and the semi gradient TD(0) methods are close to the exact value function. We compared the Mean Square Error and the Mean Absolute Error of the two algorithms with the exact value function and noticed that the values are similar to exact value function with the error being between 0 and 1. This depicts that that the algorithms are making a good approximation of the values. Moreover, the error due to the different choice of parameters such as learning rate and initialization of values including the feature vector.

REFERENCES

Maximizing Rewards with Policy Gradient Methods and Monte Carlo Reinforcement Learning
Reinforcement Learning: An Introduction
Off Policy Methods with approximation