EECS590 – Mini Project 1

Mini Project 1 Report

Author: **Davis Payne**

Course: **EECS590**

Date: January 16, 2026

Answers

Question 1

Throughout this assignment, I maintained a single working log to track my reasoning. This log documented design decisions, unresolved questions, experiments, and next steps. This approach ensured that confusion was addressed systematically and that design choices could be clearly justified in the final implementation.

Question 2

Below is a simple Python class structure representing a finite Markov Reward Process.

Python Code

```python
from abc import ABC, abstractmethod
from typing import Iterable, Dict

State = int


class AbstractMarkovRewardProcess(ABC):
    """
    Abstract blueprint for a finite Markov Reward Process (MRP).
    """

    def __init__(self, gamma: float = 0.9):
        self.gamma = gamma

    @property
    @abstractmethod
    def states(self) -> Iterable[State]:
        """Return the set (or list) of states."""
        pass

    @abstractmethod
    def reward(self, state: State) -> float:
        """Return the reward for a given state."""
        pass

    @abstractmethod
```

```
    def transition_probabilities(
        self, state: State
    ) -> Dict[State, float]:
        """
        Return transition probabilities from a state.
        Example: {next_state: probability}
        """
        pass

    @abstractmethod
    def value_iteration(
        self, tol: float = 1e-6, max_iter: int = 1000
    ):
        """
        Compute state values using Bellman value iteration.
        """
        pass
```

Question 3

The value of a state represents the expected long-term reward starting from that state. Value iteration is implemented below using the Bellman update equation.

```
class MarkovRewardProcess:
    def __init__(self, states, rewards, transitions, gamma=0.9):
        self.states = states
        self.R = rewards          # dict: R[s]
        self.P = transitions      # dict: P[s][s']
        self.gamma = gamma

    def value_iteration(self, tol=1e-6, max_iter=1000):
        V = {s: 0.0 for s in self.states}

        for iteration in range(max_iter):
            delta = 0.0
            V_new = V.copy()

            for s in self.states:
                V_new[s] = self.R[s] + self.gamma * sum(
                    self.P[s][s_next] * V[s_next] for s_next in self.P[s]
                )
                delta = max(delta, abs(V_new[s] - V[s]))

            V = V_new
            if delta < tol:
                print(f"Converged in {iteration} iterations.")
                break

        return V
```

Question 4

The grid world below is constructed from a 2D mask where *-inf* represents walls.

```
class GridWorldMRP(MarkovRewardProcess):
    def __init__(self, mask, gamma=0.9):
        self.rows = len(mask)
        self.cols = len(mask[0])
        self.state_map = {}
        self.states = []
        idx = 0

        for i in range(self.rows):
            for j in range(self.cols):
                if mask[i][j] != -math.inf:
                    self.state_map[(i, j)] = idx
                    self.states.append(idx)
                    idx += 1

        rewards = {self.state_map[(i, j)]: mask[i][j]
                   for (i, j) in self.state_map}

        transitions = {s: {} for s in self.states}

        for (i, j), s in self.state_map.items():
            neighbors = []
            for di, dj in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                ni, nj = i + di, j + dj
                if (ni, nj) in self.state_map:
                    neighbors.append(self.state_map[(ni, nj)])

            if not neighbors:
                transitions[s][s] = 1.0
            else:
                p = 1.0 / len(neighbors)
                for ns in neighbors:
                    transitions[s][ns] = p

        super().__init__(self.states, rewards, transitions, gamma)


def main():
    mask = [
        [0, -math.inf, -1, -1, -1],
        [-1, -math.inf, -1, -math.inf, -1],
        [-1, -math.inf, -1, -math.inf, -1],
        [-1, -1, -1, -math.inf, -1],
        [-3, -3, -3, -3, -3]
    ]

    grid = GridWorldMRP(mask, gamma=0.9)
    V = grid.value_iteration()
```

```
49
50      print("\nState Values:")
51      for (i, j), s in grid.state_map.items():
52          print(f"Cell {(i, j)} -> Value: {V[s]:.3f}")
53
54
55  if __name__ == "__main__":
56      main()
```

Question 5

The discount factor γ determines how much future rewards influence current values. Lower values emphasize immediate rewards, while higher values allow distant rewards to affect more states. Discounting ensures convergence of value iteration.

Yes, γ is necessary in practice. It reflects the idea that rewards received sooner are more valuable than rewards received far in the future, and it also helps ensure that the value function converges instead of becoming unstable in environments with ongoing transitions or loops. While it is possible to use different discount factors at different time steps, most models use a single constant γ because it keeps the system consistent, stable, and easy to interpret. We multiply by γ (and not γ^2) because discounting is applied one step at a time. Each step into the future is discounted once, so rewards two steps away naturally receive a γ^2 weight through repeated updates. To better understand how the discount factor affects the model, I experimented with several values of γ, specifically 0.1, 0.5, and 0.99. When γ was set to 0.1, the values were driven almost entirely by what happened immediately in each state, and rewards that were farther away had very little impact. Increasing γ to 0.5 allowed future rewards to matter more, but their influence was still mostly limited to nearby areas of the grid. When γ was set very high, at 0.99, rewards located far from a given state began to noticeably affect its value, causing the impact of both positive and negative rewards to spread across much larger portions of the grid. Overall, these experiments made it clear that γ controls how far rewards "reach" through the environment. As γ increases, the model places more weight on long-term outcomes, and the value function changes in a smooth and intuitive way as the reward structure is adjusted.

Question 6

Actions extend the model by making transitions dependent on decisions. When the system has no actions, movement is completely random, meaning that even if some states offer better rewards than others, the agent has no way to intentionally move toward them. The rewards exist, but the process itself does not try to improve outcomes and simply follows fixed transition probabilities. By introducing actions such as moving up, down, left, or right, the agent is given the ability to influence how transitions occur. Although the outcome of an action may still involve some randomness, the choice of action changes the likelihood of where the agent goes next. This shift turns the model from a passive process, where the world controls movement, into an active decision-making system, where the agent can ask which action will lead to the best long-term reward.

Question 7

For large or complex environments, storing full transition structures can be memory intensive. One possible approach is to represent environments using reusable tiles or components that are connected according to a higher-level structure, such as a tree. Transitions can then be generated dynamically rather than stored explicitly. This compositional approach allows for scalable and procedurally generated environments while avoiding large memory footprints. Such designs are particularly useful in reinforcement learning scenarios involving irreversible decisions or hierarchical structures.

Question 8

Markov process models are useful because they provide a clear and manageable way to think about random movement and long-term outcomes. Their simplicity makes them especially helpful for planning and decision-making problems where the future depends mainly on the current situation. However, this same simplicity can also be a limitation, since the Markov assumption ignores things like past experiences, memory, or information that isn't directly observable. In more complex or realistic settings, this can force the model to use very large state spaces just to capture what is really going on. Even with these limitations, Markov-based models remain a strong starting point, and extensions such as decision processes, partially observable models, and hierarchical approaches help address many of these issues while keeping the core ideas intact.

Conclusion

This project demonstrated how grid worlds can be modeled as Markov reward processes, how long-term state values can be computed through discounting and iterative methods, and how introducing actions transforms passive systems into decision-making models. While Markov processes have inherent limitations, they remain a powerful and flexible tool for understanding and designing systems that involve uncertainty and long-term planning.