Islamic University of Technology

CSE 4618: Artificial Intelligence Lab

Lab 05

Submitted by-

Name: Maliha Zaman

ID: 200042114

Group: B

Date: 31/05/2024

Task1:

Problem Analysis

Implementing a Value Iteration agent for a Markov Decision Process (MDP) is the task at hand. It was instructed to create a way to calculate the Q-values for state-action pairings based on the value function, as well as the value iteration algorithm to compute the state values across a number of iterations.

Solution:

I iterated over the amount of iterations in the Value Iteration Implementation (runValueIteration) first. I determined the highest Q-value for each state based on all feasible actions, updated the values in a temporary counter, and then changed the values for the agent. I used the formula to get the Q-value. Ultimately, I went through every action that might be taken, calculated each one's Q-value, and then returned the action that had the highest Q-value.

Interesting findings:

The implementation uses the batch update method, which means that all states values uses values from previous iterations and updated simultaneously. More iterations can improve the value estimation but increases computation time as well. Less iterations decreases computation time but may be less accurate.

Challenges:

No challenges faced as this was solved by sir.

Task2:

Problem Analysis

In the BridgeGrid scenario, the agent starts near a low-reward terminal state with a high-reward terminal state on the other side of a narrow bridge. Crossing the bridge is risky due to the high negative rewards in the surrounding chasms. The agent's decision to cross the bridge depends on two parameters: **Discount Factor (γ):** Determines how much future rewards are valued compared to immediate rewards. **Noise:** Represents the probability of ending up in an unintended state when taking an action.

Solution:

To make the agent cross the bridge, **Discount Factor:** Increasing the discount factor above 0.9 would place even more value on future rewards, but it might not be enough to overcome

the high risk associated with the noise. **Noise:** Reducing the noise decreases the likelihood of the agent falling into the chasm when attempting to cross the bridge. Given the higher risk associated with noise, reducing the noise parameter is a more effective strategy to make the agent cross the bridge. So I kept the discount factor 0.9 and decreased the noise to 0.0

Interesting findings:

Reducing noise to 0.0 eliminates the risk of unintended transitions, making risky paths like the bridge more viable.

Challenges:

No challenges faced.

Task3:

Problem Analysis

The DiscountGrid layout presents a grid world with two terminal states offering positive rewards and several terminal states offering negative rewards, known as the "cliff." The agent must navigate this environment, balancing the desire to reach high-reward states quickly with the risk of falling into the cliff. The challenge is to tune the discount factor, noise, and living reward parameters to achieve specific behaviors in the agent's optimal policy.

Solution:

3(a) Prefer the close exit (+1), risking the cliff (-10):

The agent quickly aims for the close exit, even with cliff risks. A high discount factor (1) values future rewards, while moderate noise (0.5) adds risk. A living reward of -1 encourages quick action despite the danger.

3(b) Prefer the close exit (+1), avoiding the cliff (-1):

The agent cautiously heads to the close exit, avoiding the cliff. A lower discount factor (0.3) focuses on immediate safety, and low noise (0.2) minimizes unintended moves. A living reward of -0.9 promotes cautious but quick decisions.

3(c) Prefer the distant exit (+10), risking the cliff (-10):

The agent targets the distant exit for a high reward, accepting risks. A high discount factor (0.9) values future rewards, and low noise (0.1) allows risk-taking. A living reward of -0.5 balances boldness with caution.

3(d) Prefer the distant exit (+10), avoiding the cliff (-10):

The agent aims for the distant exit while avoiding the cliff. A moderate discount factor (0.5) balances immediate and future rewards. Low noise (0.2) ensures safe navigation, and a minimal living reward (-0.01) encourages longer paths.

3(e) Avoid both exits and the cliff (so an episode should never terminate):

The agent avoids exits and the cliff, continuing indefinitely. A discount factor of 0 ignores future rewards, and no noise (0) ensures control. A living reward of 1 motivates continuous movement without termination.

Interesting findings:

Noise Impact: Reducing noise significantly influences the agent's willingness to take risks. Higher noise levels deter the agent from risky paths due to increased unpredictability.

Discount Factor: A high discount factor drives the agent towards long-term rewards, even if it involves higher risk. Conversely, a low discount factor focuses the agent on immediate, safer rewards.

Living Reward: The living reward parameter balances the urgency of reaching terminal states versus maintaining safe behavior. A highly negative living reward pushes the agent to reach terminal states quickly, while a positive living reward can lead to prolonged navigation to accumulate more rewards.

Challenges:

Took me a while to find the right balance between discount, noise, and living reward parameters to achieve specific behaviors. I tried using many different parameter combinations, systematically increasing and decreasing values to observe their effects on the agent's behavior.

Task4:

Problem Analysis

The problem implements an asynchronous value iteration algorithm for a Markov Decision Process (MDP). Unlike standard value iteration, asynchronous value iteration updates one state at a time in a cyclic order. This approach makes the process more efficient by breaking it into smaller, sequential updates.

Solution:

To solve this problem, I created an `AsynchronousValueIterationAgent` class that inherits from `ValueIterationAgent`. The key method to implement is `runValueIteration`, which performs the cyclic value iteration. Here's the implementation:

1. **Initialization**: The constructor initializes the MDP, discount factor, number of iterations, and values dictionary. It then calls `runValueIteration`.
2. **Cyclic Value Iteration**: The `runValueIteration` method iterates through the states, updating the value of one state per iteration, and then cycles to the next state.

The algorithm updates states sequentially by utilizing a queue system, processing one state at a time and rotating through the queue in a cyclic manner.

Interesting findings:

Unlike in Q1, where all states are updated simultaneously in each iteration, the asynchronous approach updates only one state per iteration. This leads to faster convergence for certain states, especially in large state spaces. By updating one state at a time, the asynchronous method distributes the computational load more evenly across iterations, potentially reducing the peak computational resources needed compared to the batch update method in Q1.

Challenges:

No challenges faced.

Task5:

Problem Analysis

In this task, the goal is to implement a Prioritized Sweeping Value Iteration Agent, which enhances the value iteration algorithm by focusing updates on states that are likely to change the policy significantly. This approach improves the efficiency of the value iteration process by prioritizing states with the largest discrepancies between their current values and their expected optimal values.

Solution:

The solution involves implementing the `PrioritizedSweepingValueIterationAgent` class, which inherits from `AsynchronousValueIterationAgent`. The main method to implement is `runValueIteration`, which performs the prioritized sweeping value iteration. For each state, I identified its predecessors—states that can transition to it through some action. Then I created an empty priority queue. For each non-terminal state, calculated the absolute difference between its current value and the maximum Q-value across all possible actions and pushed the state into the priority queue with this difference as the priority. For a specified number of iterations, popped the state with the highest priority from the queue, update its value if it's not terminal, and then updated the priorities of its predecessors if their value differences exceed a threshold.

Interesting findings:

Increasing θ: Fewer updates, slower convergence.
Decreasing θ: More updates, faster convergence.

Challenges:

Understanding the motive of this algorithm was challenging.