CS 4100 artificial intelligence / CS 5100 foundations of AI

Khoury College, Northeastern University

Programming Assignment 2

Instructions

- **ALERT:** Expect high runtimes. Start early.
- If you discuss this problem set with one or more classmates, all parties must declare collaborators in their individual submissions within a code comment. Such discussions must be kept at a conceptual level, and no sharing of actual code is permitted.
- You may use any generative AI tool available to you, as long as it is appropriately cited in a code comment. I recommend using AI tools for debugging or conceptual understanding rather than to produce actual functions.

Deadlines

- Submissions should be uploaded to Gradescope by 6:00 PM on **10/26/24**.
- Gradescope will show a 'late' deadline of **10/29/24**. This is intended solely for any students who may wish to invoke the freebie, outlined in the course policies.
- Any submissions received after 6:00 PM on **10/26/24** will be considered late, and will automatically invoke the use of your freebie. If you have used your freebie on a previous assignment, your submission will not be accepted for credit.
- Regrade requests must be submitted on Gradescope within 1 week of receiving your grade, after which no further requests will be entertained.

Reach Out!

If at any point you feel stuck with the assignment, please reach out to the TAs or the instructor, and do so early on! This lets us guide you in the right direction in a timely fashion and will help you make the most of your assignment.

Escape the Castle

Overview

In this assignment, you will use your knowledge of MDPs and reinforcement learning to play a game, The Escape the Castle. The game is set on a 5x5 grid, where the player's goal is to navigate from the top-left corner (grid position (0,0)) to the bottom-right corner (grid position (4,4)). The environment is populated with four hidden guards (whose positions are randomly initialized on every run). Each guard possesses distinct attributes, such as their strength in combat or perception ability in detecting players. The player can only discover a guard's presence by entering the same cell as the guard. The game dynamics are governed by an underlying Markov Decision Process (MDP) with probabilistic movement and guard interactions.

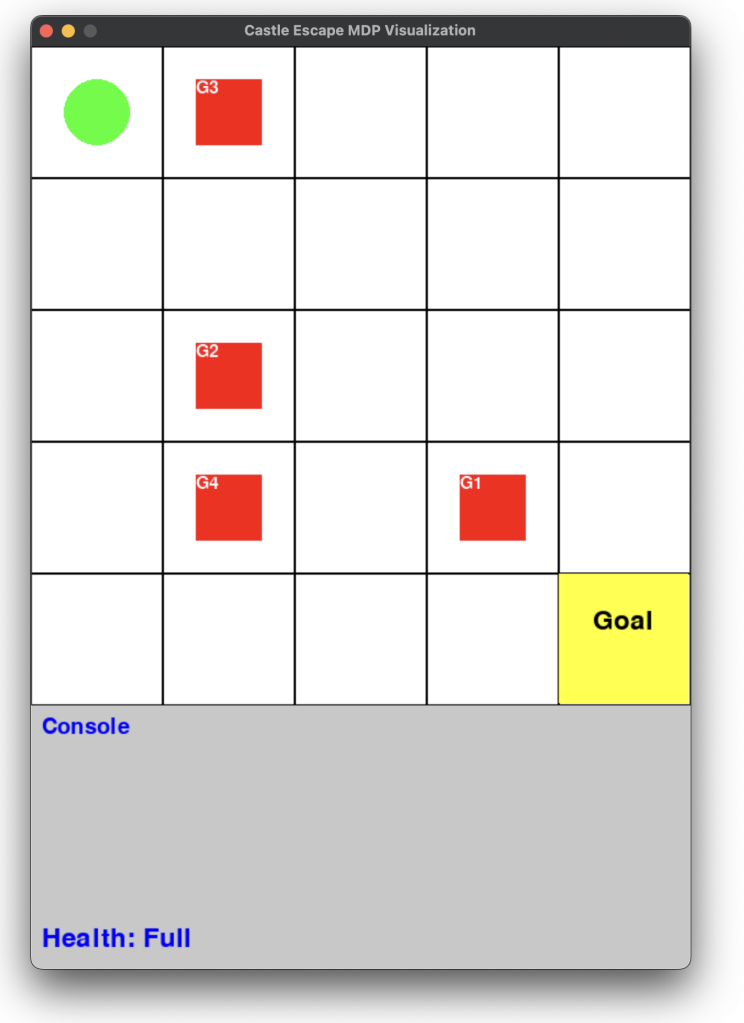

Figure 1: The Game Environment

Grid Layout

- The grid consists of 25 cells, numbered from (0,0) to (4,4).
- The player always starts in the top-left corner (0,0).
- The goal is located in the bottom-right corner (4,4).
- Four guards are randomly positioned across the grid at the start of each game, excluding the goal and starting positions. The player does not know the guards' locations until they encounter one in a cell.

Player and Guard Interactions

- When the player enters a cell with a guard, they are faced with two options: to hide or to fight. Movement actions are invalid in this phase.
- If the player fails to hide, they are forced into combat with the guard, i.e. the environment automatically executes a fight sequence and returns the outcome and associated reward/penalty.
- Each of the guards has distinct, hidden strengths (for combat) and perception ability (for detecting hiding attempts). The player is unaware of these characteristics before engaging with each guard.
- The player's interaction with a guard results in one of the following outcomes:
  - Victory in combat: The player wins the fight and is moved randomly to a neighboring cell and receives a reward.
  - Defeat in combat: The player loses, suffers a reduction in health, and is moved to a neighboring cell and receives a penalty.
  - Successful hiding: The player successfully avoids combat and is moved to a neighboring cell.
  - Failed hiding: The player fails to hide and is forced to fight.

  Regardless of the outcome, guard positions remain unchanged throughout the game.

Player's State

The player's current state is represented as a combination of three key factors:

- Position on the grid: The player can be in any of the 25 cells.
- Health status: The player has three possible health states—Full, Injured, and Critical. Losing combat results in a reduction of health. If the player's health reaches the Critical state and they lose another combat, the game ends in defeat.
- Guard positions: The four guards are randomly positioned at the start of each game. The number of possible guard placements is combinatorial, resulting in ${}^{23}C_4$ potential configurations, excluding the player's starting and goal cells.

Available Actions

The player can take one of six possible actions during the game:

- Movement:
  - UP
  - DOWN
  - LEFT
  - RIGHT
- Interaction:
  - HIDE
  - FIGHT

Movement: The player can move to any adjacent cell on the grid (up, down, left, or right), with a 90% success rate. The remaining 10% represents a "slip," where the player is moved to a random adjacent cell instead. Movement actions are not allowed when the player is in the same cell as a guard, forcing the player to either fight or hide. Attempting to fight or hide in an empty cell results in no effect.

Interaction:
HIDE: The player attempts to evade the guard. The success of this action is determined by the guard's keenness level.
FIGHT: The player engages in combat with the guard. The outcome of the fight is determined by the guard's strength.

Rewards and Penalties:
The player receives rewards and penalties based on their actions and outcomes:

- Reaching the Goal: +10,000 reward points for successfully reaching the goal at grid (4,4).
- Winning a Fight: +100 reward points for defeating a guard in combat.
- Losing a Fight: -1,000 penalty points for losing a fight and suffering health damage.
- Defeat: -5,000 penalty points if the player reaches the Critical health state and loses another fight, resulting in defeat.

Code Organization

The `mdp.py` file defines the underlying MDP, and provides interfacing methods through which your player can interact with the game environment. No changes should be made to this file. The `vis.py` file defines the PyGame environment that visualizes gameplay - doing so may help visually catch issues with your implemented algorithms. Once you get it right, it is immensely satisfying to see a properly trained RL agent at work.

States are defined by unique combinations of the following four items:

- The player's x coordinate (int $\in [0, 4]$)
- The player's y coordinate (int $\in [0, 4]$)
- The player's health (int $\in [0, 2]$)
- The guard in the player's current cell (one of `{'G1', 'G2', 'G3', 'G4', None}`)

To map each possible combination of the four items above to a unique integer, a hash function is provided in each code file. The output of the hash function is an integer in the range $[0, 374]$, and can be used to index and represent the corresponding state.

Rules

You **must** use the `env.step()` function to interact with the environment. Variables listed under the constructor (`def __init__(self):` in `mdp_gym.py`) may be accessed, but not modified directly from within your code. No other functions except `env.step()`, `env.reset()`, and `env.action_space.sample()` defined in `mdp_gym.py` or `vis_gym.py` may be called from within your code. Violations of these rules will lead to an automatic zero without the possibility of a regrade.

Your Tasks

Model Based Monte Carlo

We will first get some practice with Model-Based Monte Carlo (MBMC) methods. Implementing an end-to-end MBMC solution for this problem would be wildly inefficient, so instead we will restrict our focus to estimating a set of outcome probabilities using data gathered from the environment.

In the `MBMC.py` file, complete the `estimate_victory_probability()` function, which executes **random** actions in the environment until the specified number of episodes is completed, and aggregates the data. Next, the function - given the data from the episodes executed - computes the probabilities of defeating all four guards when a fight action is used. The function returns a NumPy array containing four probabilities, one associated with defeating each guard under the FIGHT action.

Model Free Monte Carlo

Next, we will turn our attention to Model-Free Monte Carlo (MFMC) methods - specifically, Q-learning - to build an agent that learns to play Escape the Castle. In the `MFMC.py` file, complete the `Q_learning()` function, which executes a specified number of episodes, but instead of taking random actions like we did with MBMC, here your agent should follow an ϵ-greedy policy (refresher below).

Simulate n episodes, using the `env.step()` function to interact with the environment. Calling `env.step()` at any cell gives you a new observation, the reward, a Boolean value indicating whether you are at a terminal state, and some miscellaneous info about the environment.
Starting with an empty Q-table, represented as a dictionary (see code file for complete specifications), for each observed state-action-reward-next_state sequence (s, a, r, s'), the Q-value estimate, $Q_{opt}(s, a)$ for the state-action pair (s, a) should be updated as follows:

$$\eta = \frac{1}{1 + \text{number of updates to } \hat{Q}_{opt}(s, a)}$$

$$\text{Estimate, } \hat{Q}^{t}_{opt}(s, a) = (1 - \eta)\hat{Q}^{(t-1)}_{opt}(s, a) + \eta[R(s, a, s') + \gamma \hat{V}^{(t-1)}_{opt}(s')]$$

$$\text{where } \hat{V}_{opt}(s) = \max_{a' \in A} \hat{Q}^{(t-1)}_{opt}(s', a')$$

At each time step, t, the action the agent plays from state s is chosen as follows:

$$\pi_{act}(s) = \begin{cases} \text{random}(a \in A); \text{with probability } P = \epsilon \\ \arg\max_{a \in A} \hat{Q}_{opt}(s, a); \text{with probability } P = (1 - \epsilon) \end{cases}$$

For a state-action pair that does not already have a corresponding entry in the Q-table dict, you should add an entry with the initial Q-value estimate of 0. The episode terminates when the agent reaches a terminal state, i.e., the agent either reaches the goal or is defeated in combat. The value of η is unique for every (s, a) pair, and should be updated as 1/(1 + number of updates to $\hat{Q}_{opt}(s, a)$). The number of updates to $\hat{Q}_{opt}(s, a)$ should be stored in a matrix of shape (num. states $\times$ num. actions), initialized to zeros, and updated such that executing `num_updates[`s, a`]` gives you the number of times $\hat{Q}_{opt}(s, a)$ has been updated. You can then calculate η using the formula given above. The value of epsilon should be decayed by multiplying its current value by a chosen decay rate at the end of each **episode**.

Submission

The `MFMC.py` automatically saves the updated Q-table at the end of training to a file named `Q_table.pickle`. Please upload this file, along with all code files (`MFMC.py, MBMC.py, vis_gym.py, mdp_gym.py`) directly to Gradescope (no need to zip files, please make sure you do not upload a directory). Your code will be evaluated for correctness by comparing your outputs to the expected range of values of the probabilities of defeating each guard for MBMC, and your trained agent's average reward across several episodes to the expected range of values for MFMC. For outputs outside the expected range, partial points will be awarded based on how far your outputs are from the expected range.