Artificial Intelligence Laboratory Assignment

Tóth Bálint
(PME4BQ)

1 Introduction

In this project, a Policy Iteration algoritm was implemented in a custom-created Gridworld labyrinth environment using the Python programming language. The environment is compatible with OpenAI Gym's Env class and can be installed on a local machine as a module using pip. The environment is fully observable, and the agent created uses the matrix-vector form of the Bellman Expectation Equation for the Iterative Policy Evaluation steps and a simple greedy policy improvement strategy. The agent has to navigate to a target found in the lower right corner of the labyrinth.

2 Environment

The basis of the environment is a rectangular grid, where each cell can be either a path or a wall. Agents can move from paths to neighbouring path cells in four directions: left, right, up and down. If an agent wants to take an action that leads it to a wall, its state would remain unchanged.

The environment is implemented in the maze_world.py script file and is called MazeWorld that extends the OpenAI Gym's (now maintained by Farama Foundation under the name Gymnasium) Env class. It uses a helper class called Maze located in mazer.py to generate a maze using a simple backtracking algoritm. The maze's parameters can be configured with the following constructor arguments:

- height: must be an odd number, it sets the height of the world in grid-space.
- width: must be an odd number, it sets the width of the world in grid-space.
- seed: number value, it sets the seed of numpy's random number generator.
- loops: boolean value, it instructs the generator to introduce loops in the maze.
- num_loop: number value, it gives the number of walls to be removed, hence introducing alternate paths.

The environment itself has the following constructor parameters:

- node_space_width: width of the maze in terms of possible junction nodes.
- node_space_height: height of the maze in terms of possible junction nodes.
- seed: same as for Maze, default value is 20231120.
- loops: same as for Maze, default value is False.
- num_loop: same as for Maze, default value is 2.

Node space is necessary, because the walls in the maze also take up a cell, and to make the initialisation simpler, this way the user can set the dimensions of the world if the walls were zero-width. The final height and width of the maze can be thought of twice the node space equivalent plus one.

Agents can take four actions: “up”, “down”, “left” and “right”. After each episode, the step function returns:

- observation: dictionary containing fedback to the Agent.
- reward: reward associated with the state the action lead the agent to.
- terminated: boolean flag telling the agent if it reached the goal.
- truncation condition: always false, there for compatibility.
- info: Manhattan distance from the target in a dictionary accessible with the “distance” key.

Observations contain the following items in a Python dictionary:

- “agent”: location of the agent in a two element numpy array, where index 0 is the y and 1 is the x coordinate of the agent.
- “target”: location of the target in a two element numpy array, where index 0 is the y and 1 is the x coordinate of the target.
- “environment”: the fully observable grid world as a numpy array, where 1 represents path and 0 represents wall.

The agent always starts at the top left corner and it has to reach the target located in the lower right corner. Reward is given based on the state the agent ends up after a step. Reward is -1 for every path except the final, of which the reward is 0.

3 Agent

The agent was developed specifically for the aforementioned environment. It has the same moveset as defined in the environment, and the class stores the states and transition probabilities in matrices. For each move, there is an associated matrix that takes into account that the agent will remain in the same state if it wants to step into a wall. The rewards and state values are stored in column vectors. The policy is represented with a dictionary, that associates a vector same size as the number of states, with the moves. The vector's value gives the probability of the agent taking the action in that state denoted by the index in the vector.

Iterative Policy Evaluation was implemented with it's matrix form, using the state transition matrix, the previous state value vector, the reward vector and a user-defined discount factor. The algorithm applies the Bellman Expectation Equation for all four state transition matrices associated with the action, also multiplying the result with the policy vector. Note that it is not a legal vector multiplication in mathematical sese, but a pairwise operation. To make it mathematically correct, the reward should be implemented as a diagonal matrix, but in terms of the implementation this method is more memory and comuptational time efficient. After the one step lookahead, the four subresults are added together, giving the final state values. Policy improvement works in a simple greedy fashion: if for a given state there exists a maximum value for the neighbouring states, the policy to the action taking the agent to that state will be changed to one and zero for all other actions. Policy iteration will stop if there exists an action with a policy value of one for all states.

The agent's functionallity can be accessed with its solve() method. Solve gets the following arguments:

- discount: discount factor for the Bellman Expectation Equation.
- eval_max: number of iterations for the Iterative Policy Evaluation.
- iter_max: hard limit for the Policy Iteration.

4 Usage

The environment can be installed with the use of pip locally from the gym_maze directory.

```
 pme4bq $> pip install -e gym_maze
```

The agent directory contains the agent implementation in the simple_agent.py script file as well as a test script. The test script contains the following:

```
from simple_agent import SimpleAgent
import gymnasium as gym

def main():
    env = gym.make("gym_maze/MazeWorld-v0",
                   node_space_width=10,
                   node_space_height=10,
                   loops=True, num_loop=5)
    agent = SimpleAgent(env)
    agent.solve(1.0, 10, 100)

if __name__ == '__main__':
    main()
```

The script creates an environment with node dimension size 10 by 10 and five walls removed to create loops. The agent is then created with the environment and the solve function is called with discount factor 1.0, 10 iterations per evaluation and 100 as maximum iteration count for the policy iteration.