Programming Assignment 2: CS 747

Richeek Das : 190260036

11th October 2021

Task: 1

For this task you'll find a script `planner.py`, containing a class `MDPPlanner`. Here we have implemented a bunch of functions to parse files, value iteration, howard's policy iteration, linear programming formulation and to print the output in the desirable format.

Design Considerations and Observations:

Value Iteration:

We have previously stored the details about the MDP transitions in an array named `MDP_TRANSITIONS`. It is a `numStates` $\times$ `numActions` 2D matrix with each cell containing a list of tuples: (state, reward, probability) it can transition to by taking that particular action at that particular state. This enables us to efficiently store the entire MDP without worring too much about running out of space.

We adopt the standard Value Iteration policy as discussed in class. We start with an arbirary policy of `zeros`. In the inner loop of Value Iteration, we just loop over the transitions with non-zero probability, hence speeding up the process.

Even though value iteration converges to an accurate result, we found it to take considerably slower than HPI or LP.

Howard's Policy Iteration:

We follow the same `MDP_TRANSITIONS` structure as discussed earlier. To find V^π at each iteration, we solve the Bellman Equations, using a `NumPy` linear equation solver. We build the Transition (T) and Reward (R) matrices. We aim to solve an equation of the form:

$$Ax = b$$

where $A = \mathbb{I} - \gamma \mathrm{T}$ and $b = \text{column-sum}(\mathrm{T} \odot \mathrm{R})$. x is the V^π we desire.

Once we have V^π we derive Q^π and update π as the actions which give the maximum value of Q^π. When two successive iterations have the same policy we break.

Linear Programming Formulation:

We use PuLP for this problem. We encode the following constraints:

$$\text{Minimize } \left(\sum_{s \in S} V(s) \right)$$

$$V(s) \geq \sum_{s' \in S} T(s,a,s') \left\{ R(s,a,s') + \gamma V(s') \right\}, \forall s \in S, a \in A$$

Once we retrieve V^* from this, we can calculate Q^* and derive the one of the optimal policies π^*.

Task: 2

MDP Formulation: Here we will discuss how we formulate our MDP, which mostly concerns itself with the code in `encoder.py`.

The core part of our code focusses on building the `MDP_TRANSITIONS` data structure. It is an `numStates` $\times$ `numActions` 2D array with each cell containing a list of tuples with details about the transition when the player at a certain state takes a certain action.

We introduce a new state, at the end of existing states in the state file. We treat this new state as the end state. All illegal and terminal moves, get redirected to this state. We loop through all pairs of states and actions. If a certain action is illegal, we point it to the end state with a reward of -1.0 and probability 1.0. If we reach a terminal state with some action, its either a loss or a draw, hence we redirect it to the end state with reward 0 and probability 1. If none of these hold, we pass it to the other player and check with its policy. The policy of other player can only lead to 3 cases: win, draw or a normal transition to another state. In case of a draw or transition to another state we set a reward of 0. For a win we set the reward to 1. For each case we use the probability as set by the policy of the other player.

We finally print the MDP details which are passed to the `planner.py`. Once its solved by the `planner.py` its passed to the decoder which mostly changes how we view the output!

Task: 3

The code for this part can be found in `task3.py`. The location of the state and policy files are hardcoded in this script, so it is expected this script and `data/` should be siblings.

We use `"data/attt/policies/p2_policy2.txt"` as the starting policy. We tabulate the differences (as in state wise differences, somewhat like Hamming Distance) between consecutive policies generated by running the players against each other multiple times. We observe a decline in the Hamming Distance between consecutive policies of both the players:

Player	Run. No.								
	2	3	4	5	6	7	8	9	10
1	1263	201	17	0	0	0	0	0	0
2	592	58	3	0	0	0	0	0	0

Table 1: Empirical demonstration of the declining differences between consecutive policies

You can observe the same output by running the script as `python task3.py`

Proof: We denote the policy of player k after i iterations as p_i^k.

We derive p_i^2 by keeping p_i^1 as reference. Similarly p_{i+1}^1 is derived by fixing p_i^2. Anti-tic-tac-toe must have a winning strategy just like we have a winning strategy for standard tic-tac-toe. This means, there exists an optimal strategy which cannot be beaten. If one of the players reaches that strategy its meaningless for the other player to continue/update its policy since loss/draw mean the same in our setting. Following this idea we propose every iteration must lead to a better policy. That is:

$$V^{p_i^1} \prec V^{p_{i+1}^1} \quad \text{,unless converged.}$$

To increase the value function we need to make an improvable action every iteration for at least one state. Since we have a finite number of actions and states, we will need a finite number of iterations to reach an optimal policy (policy with no value improvements). With no further increase in value, under our **tie-breaking assumptions among different policies with the same value function**, we can assume we will converge to a specific policy for both the players.