CS747 Programming Assignment 2

180050096 - Shashank Agrawal

Part 1: MDP Planning

Overview of code in planner.py

The planner.py contain certain variables and functions, let me elaborate these one by one :-

- We have s, a as number of states and number of actions respectively in the MDP.
- We have *ends* array which contains end states in case the *mdptype* is episodic, *gamma* is the discount associated with MDP.
- The three 2 dimension arrays t, p, and r corresponds to structure of MDP and each of dimension $s \times a$, $t[i][j]$ is just the list of states where a state i goes to after taking action j and $r[i][j]$ and $p[i][j]$ are the corresponding reward and probability associated respectively. I made sure that all of entries in all of $p[i][j]$s are non-zero. I made these arrays this way, so that they won't take much memory, as s^2a will be much larger than number of transitions for larger value of s and a.
- We also have *totpr* 2 dimension array which is you can say a part of pre-computation, it is also of dimension $s \times a$, and $totpr[i][j]$ contains summation over multiplication of corresponding values of $p[i][j]$ and $r[i][j]$, it will come in handy when we compute action value function or we find value function for certain policy.
- I also computed the maximum reward (*maxrw*) and minimum reward (*minrw*), which will just be used in initializing value function in case of value iteration algorithm.
- We have function *get_mdp*, which takes path to MDP file as an input and fill all the values for this MDP in the above mentioned variables.
- We have *nonts* which is the number of non-terminal states in MDP, it will be equal to s in case of continuing MDP.
- We have *state_map* which basically maps a state to it's corresponding index when we only consider non-terminal states.
- We have *get_vpi* function which takes policy as an input and outputs corresponding value function, it solves Bellman's Equations as $AV = B$ and compute V as $A^{-1}B$, here policy and B both are of shape (*nonts*, 1).
- We have *get_Q* function which takes value function as an input and outputs corresponding action value function.
- We have *get_full* function which takes a policy and a value function as an input which have shape (*nonts*,1) and outputs policy and value function which have shape (s, 1)

Task1 MDP Planning Algorithms

We have three functions which run three different algorithms and these are :-

- **Value Iteration** :- Here I have initialized value function with values uniformly picked in [*minrw*, *maxrw*], we are basically iterating in value functions and stop when we find optimal value function. I used two Vs, $V[0]$ and $V[1]$. $V[0]$ is first initialized then we apply Bellman optimality operator,B^*, on $V[0]$ and put it in $V[1]$, in next iteration, we apply B^* to $V[1]$ and put it in $V[0]$ and it goes on like this until we have achieved convergence upto machine precision.
- **Howard Policy Iteration** :- Here I initialized the policy with uniformly picked integers in [0, a). I took **np.argmax()** action value function to switch, which just returns the lowest index at which max appears and ties are resolved, and I switched all of improvable states only, by using numpy masking.
- **Linear Programming Formulation** :- Nothing much here it's just that I used pulp module of python to solve this LP Formulation and I have *nonts* variables and $nonts * a$ constraints.

I chose Value Iteration as the default algorithm, because here we have upper bound on number of iterations that is polynomial in s, k, B, and $1/(1 - gamma)$, where B is number of bits used to represent MDP, where as we have exponential upper bound in case of Howard Policy Iteration and exponential time bound in LP formulation.
There are certain **assumptions** that I took :-

- Whenever mdptype is continuing we will have end followed by -1 in mdpfile.
- Terms are given in the same order as it is in example mdp files,i.e., first line will contain number of states, followed by number of actions, followed by ends and then followed by transitions and so on.

Part 2: Anti-Tic-Tac-Toe Game

Overview of code in encoder.py

The encoder.py contain certain variables and functions, let me elaborate these one by one :-

- We are optimizing the play for $palyer\_id$ given the fixed policy followed by $player\_id2$, that's how $player\_id$ and $player\_id2$ are defined.
- We have $state\_map$ and rev_state_map, $state\_map$ basically contains all the states from states file (for $player\_id$), passed to the encoder, in the order that they are in the states file. And rev_state_map contains the reverse mapping, given the $state$, $rev\_state\_map[state]$ will give us the index at which this state is present.
- We have $probs$ dictionary which basically maps states in policy files to the probabilities of taking actions by $player\_id2$.
- We have $transitions$ which contains transitions of the formed MDP in the form of string.
- We have function get_state_map, which takes states file for as input and fills $state\_map$ and rev_state_map.
- We have function $terminal$, which takes state as an input and returns $True$ if some player has won in this state.
- we have function $get\_transitions$, which takes policy file for $player\_id2$ as an input and fills $player\_id$, $player\_id2$, $probs$, and $transitions$. To fill $transitions$, it calls $add\_possible\_transitions$.
- we have function $add\_possible\_transitions$, which given the state in $state\_map$ and an $action$ to take, it adds the possible transitions to the $transitions$.

Task2 MDP for Anti-Tic-Tac-Toe Game

I added one terminal state to the MDP, let us call this state end_state. Transitions are added as follows :-

- For all invalid $action$s for a $state$ are added as a transition to end_state with reward -100.0 and probability 1.0.
- After the valid $action$ applied to the $state$ and let's say we get a new_state1, there are two cases :-
  - If $terminal(new\_state1)$ is true then $player\_id$ loss or if all of the places are filled then it is a draw in either case we add a transition from $state$ to end_state with reward 0.0 and probability 1.0.
  - If not any of above is true, then $player\_id2$ can make a move and further process is mentioned in the next point.
- Valid $action$ by $player\_id$ is applied to $state$ and we got new_state1 now $player\_id2$ can make a move (other cases are handled in the previous point), Now we will iterate in all the $action$s and we know the probability with which $player_id2$ will do certain $action$ and this information is stored in $probs$. Now there are three cases after a certain $action$ is applied to a new_state1 by $player\_id2$ and let this time we get new_state2:-
  - If $terminal(new\_state2)$ is true then $player\_id$ wins, we add a transition from $state$ to end_state with reward 1.0 with corresponding probability in $probs$.
  - If new_state2 is full, i.e., $player\_id$ can no more make a move, then it's a draw, we add a transition from $state$ to end_state with reward 0.0 with corresponding probability in $probs$.
  - If not any of above is true, then $player\_id$ can make a move, we add a transition from $state$ to new_state2 with reward 0.0 with corresponding probability in $probs$.

Decoder just reads the optimal policy from value policy file and print the actions of policies in the required format, i.e., in one_hot_key representation, i.e., if according to policy if we have to take action a at state s, then line will be printed like this -¿ $'s\ (a\ zeroesseperatedbyspace)\ 1\ (8 - a\ zeroesseperatedbyspace)'$. And this lines will be printed after the $player\_id$ is printed on the first line.

Overview of code in task3.py

The task3.py assumes that we have states file for both players in *./data/attt/states/states_file_p1.txt* and *./data/attt/states/states_file_p2.txt* by default, if states file are present somewhere else you can pass them using flags *- -p1_states* and *- -p2_states*. It runs for 20 iterations by default, one iteration means generating optimal policy for certain player by fixing policy of other player, if you want to run it for certain number of iterations pass that number using flag *- -iterations*. I also kept a *- -keep* flag to which if we pass 1, it won't delete generated policies, it's default value is 1. Task3.py produces the resulting policies in folder named task3 with name as 'p{player_num}_policy{iteration_number}'.

Task3 Anti-Tic-Tac-Toe Optimal Strategy

We have the $p2$ as the first player by default (If you want to change you can id of first player using *- -first_player* flag), i.e., We will first generate valid policy for the $p2$ and then generate optimal policy for $p1$ for this policy of $p2$ and so on. So for generating this first policy, I just return the policy with actions corresponding to the first position at which 0 appears in states.
I tried initializing the first policy in many ways but it always converged within 20 iterations (last two policy files for both players were same, i.e., pi[18] = pi[20] and pi[17] = pi[19]). So, I tried proving convergence in a way that will also give the upper bound on number of iterations it will take to converge and I got one so I will prove that process will converge.

Proof for the Convergence

I consider that the fixed policy of the other player is such that he uses deterministic policy (optimal policy for the fixed policy of first player) or you can say there is a fixed action (probability = 1) for every state that he uses. And will also consider the MDP that I formed in task 2.
We have finite states, every state will finally reach terminal state after some actions. Now let's define new variable for each state, its *height* such that :-

- The *height* of a terminal state is 0.
- The *height* of a non-terminal state is 1 + maximum of the *height* of the states it can jump to using valid actions.

Certain things to note :-

- There won't be any discrepancy in calculating *height*, as there isn't any cycle in our MDP because every time we take action we go to either a terminal state or to a state with less non-zero entries.
- The maximum *height* of any state can be 9, otherwise there has to be a state with all non-zero entries and still non-zero *height*.
- Now whenever a certain valid action is applied to a non-terminal state, it will go to some state of lesser *height*, because of the definition of *height*.
- It will be sure that there will be no transition from state with lesser *height* to state with greater *height*, so, value function of lesser height states won't depend on value function of greater height states.
- Value function can be 0 or 1 only because we will be using particular policy and reward 1 is only on the edges going to terminal state and value function for terminal state is 0.
- All the edges that corresponds to valid action going to a state with Value function 1 will have reward 0, as non-zero reward is only when the edge goes too a terminal state, which have 0 Value function.

Consider a particular state, and all the states it can jump to after performing the valid actions, now consider value of (value function of the reached state + corresponding reward),

- If all of these values are 0, then value function of the concerned state will be 0 too, because Value function is nothing but the maximum of the above values times the probability of taking that action.
- Otherwise If at least one of them is 1 then the smallest index action will be picked such that the corresponding value (value function of the reached state + reward) will be 1 and so is the value function of the concerned state.

Note that we can take decisions on the action of this state without changing value function of the states it can jump to, because optimal policy of lesser height states won't depend on the actions of greater height states.

Let we have a deterministic optimal policy of certain player for deterministic policy of other player, after two more iterations, If transitions between the states for other player with *height* $< x$ doesn't change, then optimal policy of the states with *height* $<= x$ for current player also won't change, as optimal policy of a state

only depends on the states with lesser *height*, and transitions within states with $height < x$ for a player only depends on the policy (fixed deterministic policy used by other player) of states with $height < x$.
Let's say we start with initial policy of certain player, let's call this a first player and the other one second player, then after two iteration we will get a deterministic optimal policy of first player for deterministic policy of second player.
Now in the order of points below :-

- After two more iterations, optimal policy and also the transitions for states with $height <= 1$ won't change for the first player as MDP that represents the game for certain player only depends on the fixed policy followed by other player and this optimal policy for states with $height < 1$, which is nothing but a terminal state, won't change trivially.
- After one more iteration, optimal policy and also the transitions for states with $height <= 2$ for the second player won't change as the optimal policy for states with $height < 2$ didn't change for the first player.
- This will go on like this and after seven more iterations, optimal policy and also the transitions for states with $height <= 9$ for the first player won't change, which is nothing but the whole optimal policy doesn't change for the first player as height of any state can be at max 9.

Hence proved, that the process will converge within 2+2+1+7, i.e., 12 iterations at max. It will take no more than 12 iterations to converge.

Observations or Properties of Converged Policy

Now regarding these converged policies of both players. I found out that player 1 always losses or draw, no matter how he play. So to find out that he losses or draws, I gave loss a reward of -1, I found out that if player 1 plays action 5 in the first turn then he can force a draw, otherwise he losses every time, I got this information by printing the action value function for initial state (000000000) corresponding to converged policy. Also when I rewarded a loss with -1, the optimal policy for player 1 to force a draw is, play action 5 and then whatever player 2 plays, player 1 will play the mirror position with respect to center, i.e., if player 2 plays action i, then player 1 plays action $10 - i$. If player 1 plays let's say any other than action 5 at first turn, player 2 wins.