CS747 assignment

Malhar Kulkarni

Roll No. 19D070032

1 Introduction

This document comprises of the explanation of each of the algorithms used in the first task, along with a small equation pertaining to each of them.
For the last task, I will give a brief description of my algorithm, along with my answer for the given question, given in a rather ad hoc manner (that is, I would explain the algorithm and an approach which roughly justifies the output obtained).

2 Task 1

2.1 Explanation of Code

For the first task, I have used *planner.py* to implement take as input the list of actions of the MDP, in order to determine the ideal policy and value function.
However, do note that I have not considered the episodic and continuous tasks as separate, mainly because for the episodic tasks, the terminating state would have probabilities and rewards of 0 for all actions; which is initialized to 0, by default for all the algorithms.
So, the given algorithms would not change, whether the MDP is episodic or continuous.

- **Value Iteration**
  For this algorithm, I have initialized two separate lists as Value Function(t-1) and Value Function(t), which would get updated by the while loop, according to the given formula:

$$V(s,t) = \max_a \sum_{s'} T(s,a,s').(R(s,a,s') + \gamma V(s', t-1))$$

  Here, there wouldn't be any tie-breaking situation required.

- **Linear Programming**
  NOTE: On implementing this algorithm, PuLP generates an extra output displaying the details of the variables fed into the solver.
  In this case, I have fed n variables into the solver as variables, which are just the ideal values of the value function for each of the given states.
  The constraints used are given as:

$$V*(s) \geq \sum_{s'} T(s,a,s').(R(s,a,s') + \gamma V*(s'))$$

  This applies for every action a, and hence, the number of constraints are the number of states times the number of actions.
  The function to maximise is just the sum of the value functions for every single state.
  Here, in order to prevent any tie-breaker, I have assumed that the ideal policy would be the larger

state of the ones given.

- **Howard's Policy Iteration**
  In this case, we would be using Numpy's Linear solver to solve the system of n-variables of the value functions, so as to obtain the value function and the Q-values for any given policy.
  Here, I have started with an arbitrary policy, where each state would follow the 0th action.
  Then, according to the values of Q and value function, I obtain the number of improvable actions and improvable states, for every state-action pair for which the Q value is greater than the value function.
  Here, the tie-breaking choice to determine which state to improve upon was done by simply taking the first state-action pair which is improvable.
  However, an improvement to this method could have been made by taking the difference between the Q and value function, and improve that particular state which has the largest such difference (in a greedy algo fashion).

3 Task 2

For the anti-TTT task, I have used Howard's Policy Iteration as a default algorithm to determine the best policy at each state.
Here, I will give a brief description for each of the three .py files I have implemented in order to obtain the ideal policy at each state.

- **Encoder**
  The encoder for the attt task comprises of various secondary functions such as $reward()$ and $ends()$ so as to find whether or not a given configuration on the attt board terminates, and if it does, whether or not it yields a reward for the current player.
  However, in the main function, I firstly convert this problem into an episodic MDP, which has only one termination state, called 'T', which is an extra state I have added so as to properly complete the policy for the MDP.
  Then, I iterate through the list of states given. Note that the policy given to us in the encoder is that of the opponent player.
  So, we must first iterate through 9 different possible moves on the attt board, which the current player could play (these would be the actions for the policy according to every given state).
  Let us call this the current state.
  Then, the next state would be obtained using the opponent's policy file. That is, we would obtain the probability of reaching various states (for the current player), according to the move made by the opponent given in the opponent's policy file.
  This policy file would give us the s' for every given s and a from the current player's side.
  Hence, we would have a complete MDP, such that the list of states and actions are given; the transitions would be obtained using the above mentioned procedure to find (s, a, s') (note that s' would be 'T' whenever the game terminates); and the rewards will be given through the configuration of the board (either 0 or 1); and the probabilities of reaching any given state are given directly in the opponent's policy file.
  Combining all these parts into one encoding method, we obtain the list of transitions for any given state and policy files.
  This MDP is then passed to the Planner.

- **Planner**
  The planner is rather straightforward, as it just uses HPI to solve the MDP and obtain the optimal policy and value function (note that as there are far more states than the other standard problems, the time required to complete this implementation is really high)
  Once the optimal policy and value function is obtained, we pass this file to the decoder

- **Decoder**
  The decoder is also straightforward. When trying to find an optimal policy, there wouldn't be a requirement for any randomness in the required move for any given state.
  So, the answer for this problem, which is just the ideal policy, would just comprise of probabilities of 0s and 1s for each state.
  Hence, the policy file would just be obtained by putting 1.0 at the optimal action for every given state. The other actions are given a probability of 0.0 in the policy file.
  This would give the policy followed by the current player.

4 Task 3

For this task, the code utilized is pretty rudimentary. There is a for loop that reiterates for a specified length (in this case, 10 times); where we simply swap between whoever is the current player at any instance, and then, pass that player and their corresponding states and policy to the encoder-planner-decoder loop.
This would then return a new policy for the current player, which would be fed to the same function for the next player, and so on.
However, the main question here is whether or not this pair of policies would converge, that is, after as the number of loops tend to infinity, what is the expected behaviour of the current player's policy, that is, would it converge, diverge or oscillate.
To get a rough judgement of the same, let us, without loss of generality, assume that the first player is player 1 and the policy used by player 2 to counter it is given by $\pi_2(0)$

Let us also define a term to obtain the expected reward of the given function, following a said policy. This would be $R(n, \pi_1, \pi_2)$; where n is the player whose reward is being calculated and π_1 and π_2 are the players' respective policies.

We can also define $S(n, t)$ which is just the value of R at any given instance of time.

Now, assuming that we have player 2's policy of $\pi_2(0)$, and we found player 1's policy $\pi_1(1)$ using the above algorithm.
Note that this policy is the optimal policy to maximise player 1's expected reward.

Now, player 2 goes through the same loop, and obtains a policy $\pi_2(2)$
Note that this is the optimal policy against $pi_1(1)$
So, no other policy player 2 follows can get a higher expected reward than $\pi_2(2)$
Hence,

$$R(2, \pi_1(1), \pi_2(2)) \geq R(2, \pi_1(1), \pi_2(0))$$
$$\therefore S(2, 2) \geq S(2, 0)$$

The equality here can only be established if:

$$\pi_2(2) = \pi_2(0)$$

Here, we don't care about the case where equality is established, because in the tie-breaking situation of my algorithm, there isn't any randomness involved. We simply take the latter policy in a list of policies which

have the maximum value function.
So, every next policy we obtain is completely deterministic and is solely dependant on the current policy of the opponent.

Hence, if $\pi_2(2) = \pi_2(0)$, the sequence stops here itself, as every next policy would just be this same one for player 2, and $\pi_1(1)$ for player 1.

However, a more general scenario that takes place is that the policies oscillate.

That is, because there is only a finite number of policies each player can take up.
If any of these said policies repeats, the whole sequence would be completely periodic, with the period of the sequence being the smallest duration between any two same policies.

Hence, we would expect the policies to look something like this:

$$...\pi_2(0), \pi_1(1), \pi_2(2), \pi_1(3)...\pi_1(n)\pi_2(0)\pi_1(1)...$$

Here, do note that the expected period of such a sequence requires extensive calculation to find.
However, this sequence cannot diverge to a new policy at each new instance of time.
Instead, the sequence of policies converges to a given subsequence, which may be of length 1.