CS 747 Assignment 2 Report

Shashank Roy - 180050097

Task 1:

After taking in input the **MDP** from the specified file various parameters like gamma, number of states and many more are extracted.

Transitions are stored in a multidimensional list named **mdp_trans**.
mdp_trans[state][action] gives a list of tuples.
Tuples are of format (new state, reward, probability).
Thus we have now encoded the transition representing the action taken from state and the new state and reward with corresponding probability.

For end states value function gives 0 with optimal action as 0 (although it is non takeable)

I also maintain for each state the set of takeable actions from that state.
We only iterate through those actions for which transitions are provided and compare **Q** only among them. If no transitions are provided and it is not an end state,again value function gives 0 with optimal action as 0.
This considerably lowers the time for the iterative algorithms.

If multiple actions give a tie for max **Q** we take the action with smallest index.

Eg. For a tie between action 3 and 5 we take 3.

The **default** algorithm is set to **Howard policy iteration algorithm**.

Value Iteration:

The algorithm follows what has been taught exactly in class. We initialize value function and action policy for each state as 0.

Then we run the following loop till convergence.

For each state, if it is an end state skip (as it's value is already 0) else we loop through the takeable actions, compute **Q(s,a)** and update policy and value function which gives max **Q(s,a)**.

For convergence we simply check if L2 norm of the difference between new and old value function tends to 0 (< 1e-10).

Howard policy iteration:

To keep things clean we initialize policy for each state with the smallest action index from the **set** of takeable states from that action. If the **set** is empty we initialize the action as 0.

As mentioned in **Moodle** discussions we run the loop 10 times.

In each loop we first compute the value function for given policy $V^{\pi}(s)$. This is calculated via iteration. The source for the algorithm has been mentioned in **references.txt**.

Then we loop through each state (skip if end state) and calculate **Q(s,a)** every takeable action.

We keep a record of the max **Q** with the corresponding action. We finally update the policy for that state. Thus we change the policy for every state to the action which gives the max **Q** for that state.

We break the loop prematurely if the policy does not change.

Linear Programming:

As explained in class we minimize the sum of all value functions for each state.

Therefore we model value function for each state as a **pulp** variable.

Then we add the variables using **pulp.lpSum** and insert it into the **pulp LPminimization** Problem variable **prob**.

Then we add the constraints. Finally we solve it using **PULP_CBC_CMD** solver available in the docker. This gives the value functions for each state.

For finding the policy, for each state we initialize **Q_s[a]** as vector of **-inf** and loop through the set of takeable actions to compute it using the value function obtained from above.

We set the action as the **argmax** of the **Q_s** vector.

In case of tie smallest index action is taken as explained above.

Task 2:

For encoding the policy as **MDP** file, let's say without loss of generality we take **policy of player 2** and **states of player 1**. Now with the player 1 as agent we need to create transitions for each valid action (if the move is playable).

Details about the MDP:

Number of actions = 9
Discount (gamma) = 0.9
MDP type = episodic

For the states we add **3** more additional states namely **win, loss** and **tie**.
We could have added just 2 states but just for analysis we add these.
In the original script reward for transition to **loss** and **tie** has been kept **0** as mentioned in problem statement. All the three states are **end states.**

Thus we encode the resulting states from 0 - N-1 (where N-3 is number of playable states for **player 1**) in the same order as mentioned in the states file.

Mappings for the end states:
win → N-3
loss→N-2
tie→N-1

To explain how transitions are created let's take an example:

General case:

Player 1 state s1= 010212000

Now we loop through set of takeable actions {0,2,6,7,8}

For action = **8**

Player 2 state s2= 010212001

Now player 2 can make the following actions {0,2,6,7} with probability {p1,p2,p3,p4} respectively as given in policy file.
So on action 2 we get s1' = 012212001 (new state for player 1)

So we add the following transition

transition **s1 8 s1' 0 p2**
Since on taking action 8 from s1 we go to s1' with probability p2

End game case:

Case where player 1 move leads to end of game:

After taking action from s1 we check if there is a policy for the new state for player2.

If not, this means the game is over. Player 1 has either lost (since he made the last move) or it's a draw.

Take s1 = 120120000
If player 1 takes action 6 s2 = 120120100 (player 1 has lost since 1st column is filled with 1). Hence s2 is a **loss** state and encoded as **N-2**.
Probability for s2 by action 6 is 1 (since on taking this action only s2 is possible).
Hence we add the transition
Transition s1 6 s2 0 1
The same goes for the tie case with s2 being **tie** state and encoded as **N-1**.

Case where player 2 move leads to end of game:

If player 1 makes a move which does not result in the end of game we now check for player 2 moves which can end game.

Say s1 = 120120000

On action 8 s2 = 120120001 (Game has not ended)

If probability of player 2 taking action 7 in **p1**.
s1' = 120120021 (Player 1 has won, therefore s1' is a **win** state encoded as **N-1**.)

Hence we add the transition
transition **s1 8 s1' 1 p1**
Reward is 1 since player 1 wins with this state.

The tie game encoding proceeds with a similar manner. For player 1 state s1 on taking action ac we get s2. If probability of taking action for player 2 which results in a **tie** state is p1 we add the following transition

transition **s1 ac s1' 0 p1**

This transition will not be added for player 1 MDP since player 2 can't tie a game (5 1's and 4 2's thus player 1 will have to make last move for tie). However for the vice versa case, when we give player 1 policy optimizing player 2. The above transition will be added with s1 being state of player 2 .

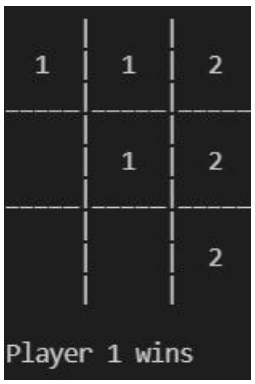
Thus we have successfully encoded the policy into MDP

For the **decoder** we simply ignore the last three lines of value policy file as they correspond to the end states.
For a given state if policy is 4 (say)
In the policy file we write the probability for action 4 as **1** and for the rest of the actions **0**. This gives us a deterministic player.

On removing seed from **attt.py** player 1 won most of the times (with very few ties and losses) for **p2_policy2.txt** which is not a deterministic policy and hence the state of end game was different almost every time.

For p2_policy1.txt player 1 wins but since both players are deterministic every game ends in only 1 state.

While testing on my system the end state only comes as:



The same goes when player 2 is optimized against player 1.
For **p1_policy2.txt** player 2 wins almost every time with a different end state almost each time the game is played, on account of player 1 not being deterministic according to given policy.

p1_policy1.txt is deterministic and hence games always end in one win state with player 2 winning. For my system

Task 3:

Assumption: The data folder is in the submission directory for hard coded states file path.

You have the option to select which player policy to initialize and start the optimization. Since no arguments are to be provided to the script, the variable **start_with** can be changed to provide initialization to different player.

If start_with = 1 we initialize player 2 policy and start optimizing player 1 against it and vice versa.

For the initialization of policy (for both players), each valid action probability is kept uniform

Example:
If start_with = 2. We initialize player 1 policy.
For state 220000011
Policy = [0 , 0 , 0.2 , 0.2 , 0.2 , 0.2 , 0.2 , 0 , 0]

For state 221000211
Policy = [0 , 0 , 0 , 0.3333 , 0.3333 , 0.3333 , 0 , 0 , 0]

Therefore using this policy player can take any of the valid actions uniformly.

Analysis has been done using other initializations as well, but this is provided in **task3.py**.

On running **task3.py** a new folder **task3_results** is formed in the same directory. It contains the initial policy and the subsequent generated value policy file (**p{1 or 2}_value_and_policy_file_iter_{i}.txt**) and the corresponding policy file (**p{1 or 2}_policy_iter_{i}.txt**) for each player in each iteration.

On the output console iteration number is printed along with the change in policies for each player.

Now talking about convergence.

For all kinds of initializations both player 1 and player 2 policy converge.

If we use our intuition, we can see that player 1 is always at a disadvantage since he always has to make greater than equal to number of moves than player 2. Thus player 2 can almost always manipulate player 1 into filling 4 boxes such that the last input by player 1 results in a tie or a win for player 2.

This is what exactly happens. Policies converge in 3 - 4 iterations itself. But as specified in problem statement , we run 10 iterations. After convergence player 2 becomes unbeatable.

Hence on optimizing player 1 results in almost no change for player 1 policy and since both **tie** and **loss** are equivalent, player 1 policy just loses or ties with player 2.
And since player 2 policy is already winning against player 1, on further optimizing player 2 policy, no changes are observed.

Even I tried beating the converged player 2 policy by interacting with attt.py. `python attt.py -p2 task3_results\p2_policy_iter_4.txt` (since policy is already converged).
But I lost almost every time with 1 or 2 ties here and there.

The player 1 policy against player 2 policy is not good at all since it treats loss and ties equivalently. It doesn't have the policy to win against player 2 and therefore loses or ties almost every time.