CS-747

Foundations of Intelligent and Learning Agents

Adarsh Raj - 190050004

Assignment-2

October 2021

Task - 1

Implementation of MDP Sampling Algorithms:

Extra Functions Used: A function `calc_action_val()` is implemented to calculate the value of $Q^\pi(s, a)$ for all states and actions, given policy π. A function `calc_val()` is also implemented for solving the Bellman Equations to get $V^\pi(s)$ for all states, given π.

1. **Value Iteration:** The values of states are iteratively updated according to the value iteration rule, till machine precision. Here, machine precision is defined by a tolerance value (1^{-10}) difference between V^t and V^{t+1}.

   Assumption: For tie breaking, i.e, for choosing the max value for different actions for a state, the **first action** with maximum value was considered, using `np.argmax`. Also, the value iteration loop is terminated when L1 norm of V^t and V^{t+1} is close enough, i.e `np.linalg.norm(v - v_prev) < 1e-10`.

2. **Linear Programming:** Using `pulp` library, initialised a minimize optimisation problem for total sum of value of states and added constraints as per the slides to the problem. Since, for terminal states we have s_T, $V^*(s_T) = 0$, added additional constraints for them and used the default solver to get $V*$. Using V^*, policy π was calculated.

   Assumption: For tie breaking amongst maximum Q values, the **first action** was considered, using `np.argmax`. Also, the values of Q were initialised -1 to account for invalid actions, for Task 2.

3. **Howard's Policy Iteration:** Initially a random policy was taken, and while in loop, V^π was calculated using `calc_val()` function and then for each state and action Q was calculated and the **first action** with Q greater than $V^\pi(s)$ is considered and policy is updated. The loop continues till there is no improvable policy, i.e no actions such that above case holds.

   Assumption: For checking the diff in Q and $V(s)$, tolerance value was set to 1^{-6}, i,e if `q > v[s] + 1e-6`, then update the policy.

Note: The data structure used by me is a python dictionary, that is
`Transition[(s,a)] = [[s1, r1, p1], [s2, r2, p2], .....]`. Here, the key is a tuple of first state and action and the value is a list of 3 sized lists comprising of next state, reward and probability.

Since, the terminal states do not have a valid action, there are no entries in the transition dictionary. Therefore, a check for key presence is implemented every loop, which on false return, calls the continue operation, which is similar to keeping `Transition[(s_T,a)] = []`, where s_T are terminal states. Hence, in this way the terminal states are handled.

Task - 2

Anti Tic-Tac-Toe

The states in mdpfile are considered from 0 to `total_valid_states`, that makes total number of states 1 more than the valid states. This extra state is the terminal state (all the terminal states like lose, draw, win) are clubbed together into this state. Also, other state numbers corresponds to the lines in the state file, for example state 0 corresponds to line 1 of the state file given as input. Discount for the MDP created is 1.

1. **Encoder Script:** For this script, a function `check_term_state()` is implemented, which checks if the state provided in argument is valid, lose or draw for the player who called the function or win for the player. This is implemented using conditional expressions.

   All the valid states of player acting as agent are iterated and for different actions the future states are calculated and then the resultant status of states are calculated, if player wins/ loses/ draws for different actions, the probabilities are calculated by taking the sum and the final state of player agent is calculated. The corresponding transition is stored in similar data structure as explained in previous part.

   For invalid actions, a negative reward (-100) is added along with a transition to the terminal state for the player agent.

2. **Decoder Script:** In this script, simply return the original state expressions, with value of probability 1.0, for the optimal action from the optimal policy calculated using `planner.py` for player agent mdpfile created as above by `encoder.py`.

Task - 3

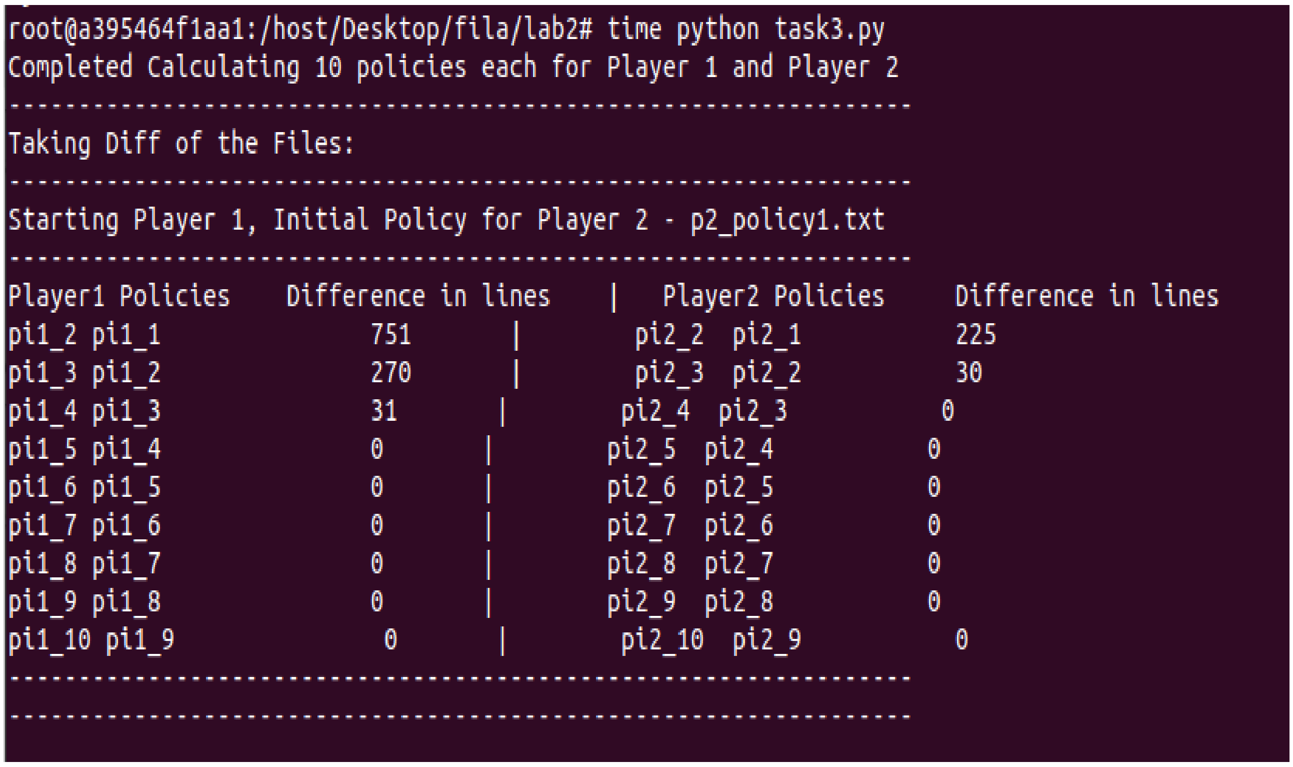
In `task3.py`, hardcoded the initial fixed player and the policy from one of those given(initial policy does not matter though in convergence). After running ten iterations of encoder, planner and decoder using policies from previous iterations, it was observed that optimal policies for both players converges.

Note: To change the initial player and initial policy of opponent, change the global variables in `task3.py`.

Ten policies corresponding to the number of iteration were calculated for each player and then the difference between these policies was calculated with the final policies obtained that are `pi1_10` and `pi2_10`. These policies get generated by the script `task3.py` and get stores in task3 folder created in the main directory.

For different initialisation and first player, the script was run four times, corresponding to both players having opponent with two fixed policies. Successively the difference between policy files was calculated and given as output to console, **the convergence** can be verified from the screenshots attached.

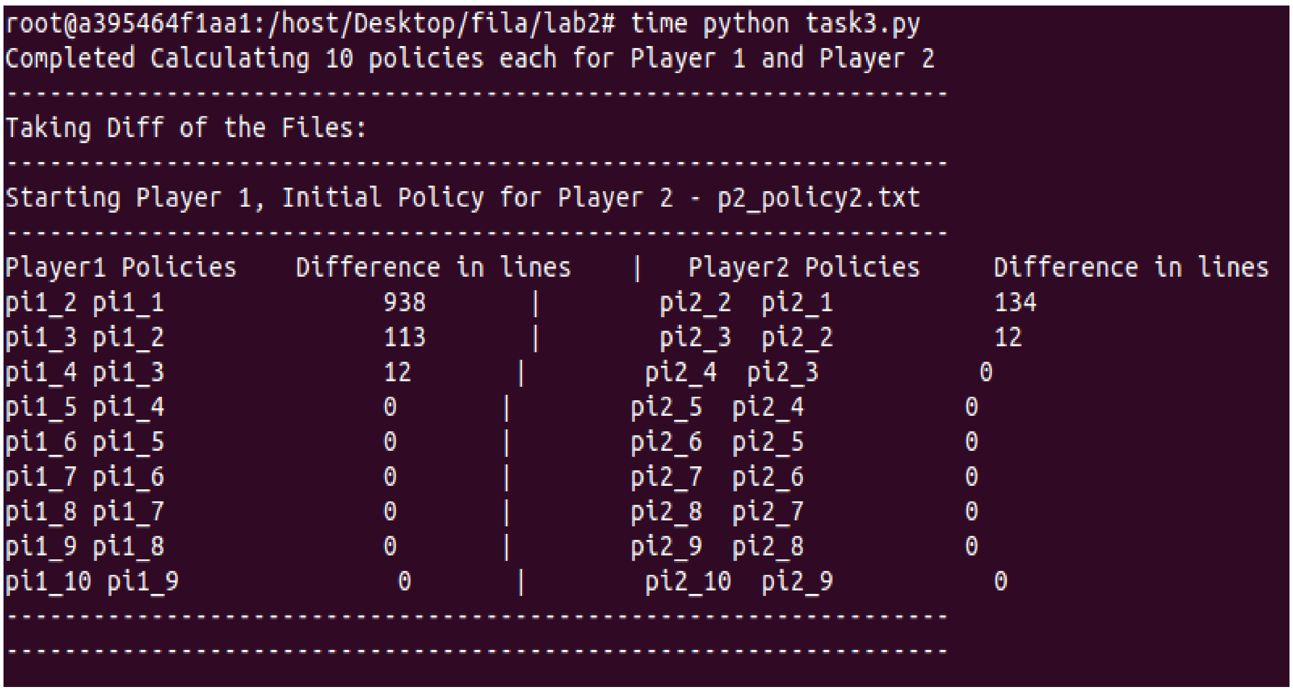
Starting Player 1
Policy of Player 2 - policy1



```
root@a395464f1aa1:/host/Desktop/fila/lab2# time python task3.py
Completed Calculating 10 policies each for Player 1 and Player 2
-----------------------------------------------------------------
Taking Diff of the Files:
-----------------------------------------------------------------
Starting Player 1, Initial Policy for Player 2 - p2_policy1.txt
-----------------------------------------------------------------
Player1 Policies    Difference in lines    |   Player2 Policies     Difference in lines
pi1_2 pi1_1               751       |       pi2_2  pi2_1           225
pi1_3 pi1_2               270       |       pi2_3  pi2_2           30
pi1_4 pi1_3               31       |       pi2_4  pi2_3          0
pi1_5 pi1_4               0       |       pi2_5  pi2_4          0
pi1_6 pi1_5               0       |       pi2_6  pi2_5          0
pi1_7 pi1_6               0       |       pi2_7  pi2_6          0
pi1_8 pi1_7               0       |       pi2_8  pi2_7          0
pi1_9 pi1_8               0       |       pi2_9  pi2_8          0
pi1_10 pi1_9               0       |       pi2_10  pi2_9          0
-----------------------------------------------------------------
-----------------------------------------------------------------
```

Figure 1: Difference in policies files

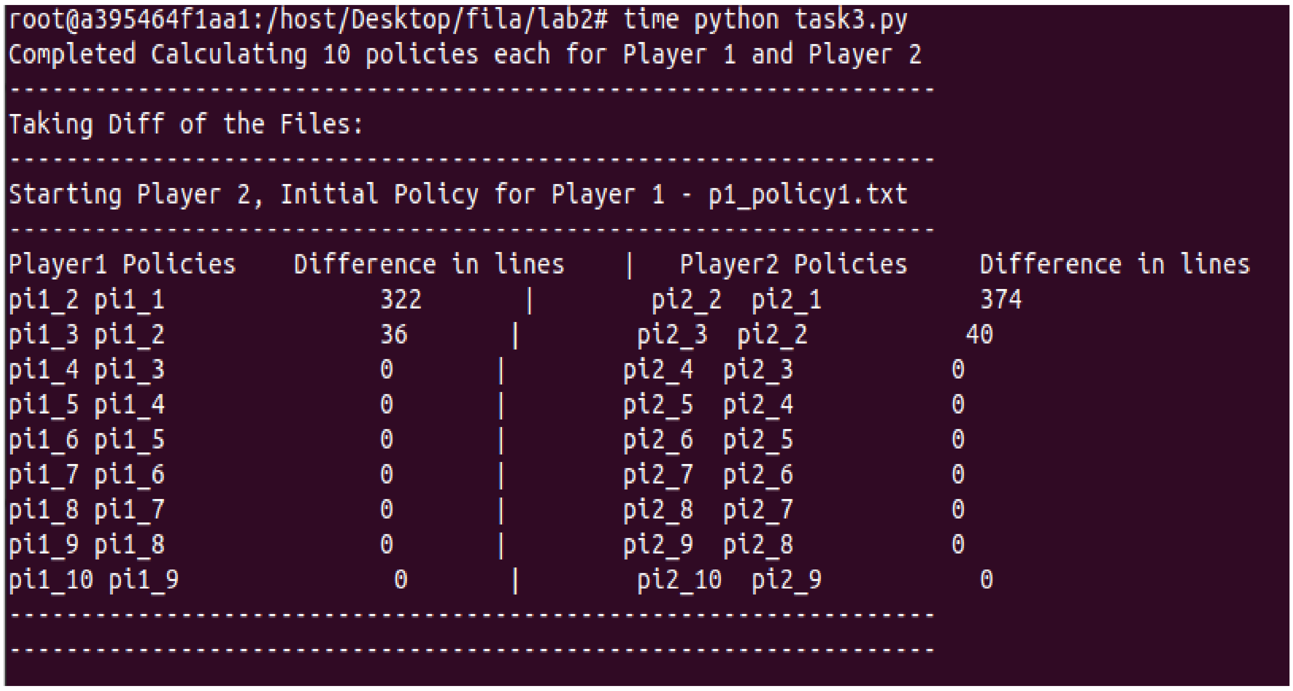
Starting Player 1
Policy of Player 2 - policy2



```
root@a395464f1aa1:/host/Desktop/fila/lab2# time python task3.py
Completed Calculating 10 policies each for Player 1 and Player 2
----------------------------------------------------------------
Taking Diff of the Files:
----------------------------------------------------------------
Starting Player 1, Initial Policy for Player 2 - p2_policy2.txt
----------------------------------------------------------------
Player1 Policies    Difference in lines    |   Player2 Policies     Difference in lines
pi1_2 pi1_1               938       |        pi2_2  pi2_1           134
pi1_3 pi1_2               113       |        pi2_3  pi2_2           12
pi1_4 pi1_3               12       |        pi2_4  pi2_3           0
pi1_5 pi1_4               0       |        pi2_5  pi2_4           0
pi1_6 pi1_5               0       |        pi2_6  pi2_5           0
pi1_7 pi1_6               0       |        pi2_7  pi2_6           0
pi1_8 pi1_7               0       |        pi2_8  pi2_7           0
pi1_9 pi1_8               0       |        pi2_9  pi2_8           0
pi1_10 pi1_9               0       |        pi2_10  pi2_9           0
----------------------------------------------------------------
----------------------------------------------------------------
```

Figure 2: Difference in policies files

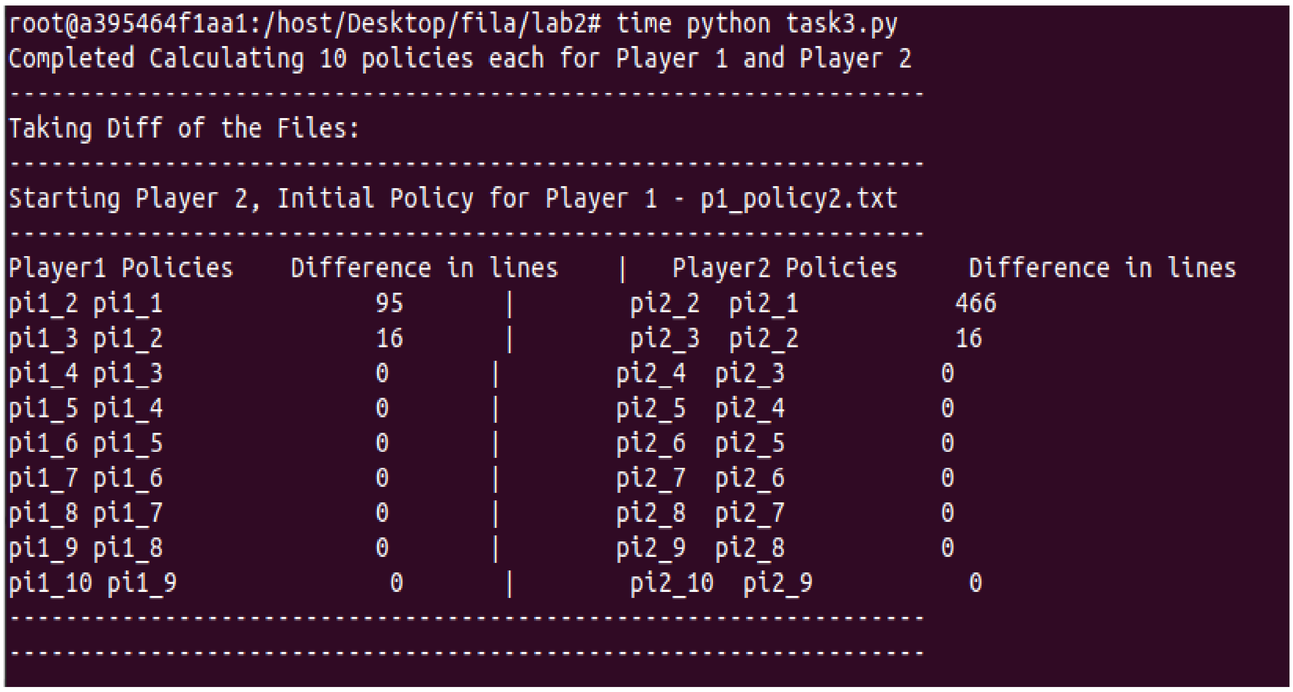
Starting Player 2
Policy of Player 1 - policy1



```
root@a395464f1aa1:/host/Desktop/fila/lab2# time python task3.py
Completed Calculating 10 policies each for Player 1 and Player 2
----------------------------------------------------------------
Taking Diff of the Files:
----------------------------------------------------------------
Starting Player 2, Initial Policy for Player 1 - p1_policy1.txt
----------------------------------------------------------------
Player1 Policies    Difference in lines    |   Player2 Policies     Difference in lines
pi1_2 pi1_1               322       |        pi2_2  pi2_1           374
pi1_3 pi1_2               36       |        pi2_3  pi2_2           40
pi1_4 pi1_3               0       |        pi2_4  pi2_3           0
pi1_5 pi1_4               0       |        pi2_5  pi2_4           0
pi1_6 pi1_5               0       |        pi2_6  pi2_5           0
pi1_7 pi1_6               0       |        pi2_7  pi2_6           0
pi1_8 pi1_7               0       |        pi2_8  pi2_7           0
pi1_9 pi1_8               0       |        pi2_9  pi2_8           0
pi1_10 pi1_9               0       |        pi2_10  pi2_9           0
----------------------------------------------------------------
----------------------------------------------------------------
```

Figure 3: Difference in policies files

Starting Player 2
Policy of Player 1 - policy2



```
root@a395464f1aa1:/host/Desktop/fila/lab2# time python task3.py
Completed Calculating 10 policies each for Player 1 and Player 2
------------------------------------------------------------------
Taking Diff of the Files:
------------------------------------------------------------------
Starting Player 2, Initial Policy for Player 1 - p1_policy2.txt
------------------------------------------------------------------
Player1 Policies    Difference in lines    |   Player2 Policies     Difference in lines
pi1_2 pi1_1               95       |       pi2_2  pi2_1           466
pi1_3 pi1_2               16       |       pi2_3  pi2_2           16
pi1_4 pi1_3               0       |       pi2_4  pi2_3           0
pi1_5 pi1_4               0       |       pi2_5  pi2_4           0
pi1_6 pi1_5               0       |       pi2_6  pi2_5           0
pi1_7 pi1_6               0       |       pi2_7  pi2_6           0
pi1_8 pi1_7               0       |       pi2_8  pi2_7           0
pi1_9 pi1_8               0       |       pi2_9  pi2_8           0
pi1_10 pi1_9               0       |       pi2_10  pi2_9           0
------------------------------------------------------------------
------------------------------------------------------------------
```

Figure 4: Difference in policies files

Convergence of Policies

From the empirical results, i.e using `task3.py`, it is evident that the policy for both player converges to optimal policies. Also, one can test that the optimal policies are independent of initial policies for both players, and at last the optimal policy of player 2 is also such that it never loses, on setting that policy in `attt.py`.

The game Anti Tic Tac Toe can be viewed as a state transition model (or directed acyclic graph) having multiple branches from a state to different states depending on the action taken. Now, in our algorithm for policy calculation, we are using `np.argmax` for tie breaking between same maximum values which can lead to the optimal policies, but usually it is not the case. So first such actions will be considered using argmax. Since, the tie-breaking is **deterministic** and not changed after consecutive iterations, encoding mdp for first player taking counter policy of another player will change the action for the states which were not playing optimal actions against the opponent whose actions corresponding to the same action state pair just became optimal. Hence, in this way states which will have just one difference in board positions filled will get optimized. The same idea can be reasoned as below for multiple iterations.

Now, the transition graph has multiple branches for different possible states and actions, and the terminal states clubbed into one state, the levels can be considered having those states such that each state will have same number of empty positions to be filled by players, as the level of graph. The level can also be taken as the distance from empty states. Now, starting from empty states, after each iteration of encoder, planner and decoder for all states and for both players, states for latter player will get optimized having filled fields number same as the distance from empty states. Hence, as we increase the number of iterations, the states with distance from empty states d (here, distance equals number of filled positions), will get optimized after d operations for one of the player and after $d+1$ iteration for another player. Now, since there are total 9 possible distances from empty states, the policies are definitely having optimal actions after iterations ≥ 10.

The above reasoning is only possible due to the deterministic nature of tie breaking for our policy evaluation algorithms. For the policies given in `data` folder, the policies converge after 3-4 iterations (≤ 10), which is desirable. After running the script using random policies the number of iterations was observed to go upto 6-7.