Swarada Bharadwaj

18D170031

CS 747: Assignment 2 Report

Task 1:

The MDP details input to planner.py are stored in a dictionary ‘mdp’ containing all of the information about the mdp. The key “states” is for the number of states, “action” is for the number of actions, “discount” is for the discount factor, “end” is for a list containing the ids of all the terminal states. The keys “r” and “t” are for the three dimensional matrices for storing transition probabilities and rewards respectively (mdp[“t”][s, a, s’] = T(s,a,s’) and mdp[“r”][s, a, s’] = R[s, a, s’]). These matrices are initialized with all zeros and then filled in with any positive values encountered in the mdp file.

1. Value Iteration:

   valuefs is the array used to store the value function values, and is initialised with all zeros. The Bellman Optimality operator is then repeatedly applied to this array, with values from the previous iteration stored in ‘prev’ for this purpose. To break all ties while taking the maximum value over all actions as part of the operation, np.amax() and np.argmax() is used. These functions select the value with the lowest index to break any ties. So, in case of more than one action resulting in the optimal value function for a state (multiple states have the same highest value function in that iteration), the action with the lowest index is chosen.

2. Howard Policy Iteration:

   A first arbitrary policy is initialised randomly (using np.random.randint), with an array of randomly selected numbers between 0 and the total number of actions. A random seed of 10 is set.

   The policy_eval function’s purpose is to provide a reasonably accurate approximation for the value functions for each state (returned in the vfs array) for a given input policy and mdp. For this it uses the Iterative Policy Evaluation algorithm.

   The value functions and subsequently action value functions for each action of all states are calculated with the help of policy_eval, as well as the number of improvable states (states with more than 0 improving actions). The improving actions are those for which the action value functions in the qvalues array are greater than the corresponding value functions in valuefs for that state. The improvactions array is a 2-dimensional array consisting of 1’s and 0’s for each state-action pair to denote if that action is an improving action for that state (1 if it is improving, 0 if it is not).

   A loop is implemented which terminates when the number of improvable states is 0. For each iteration, for all improvable states, a new improving action is selected and updated in the policy. In case of many possible improving actions, the action with the lowest index is selected.

   The value functions and action value functions for every action for each state are then updated and stored, improvactions is updates and the loop repeats. At all times, the action value function corresponding to the action the state is to follow as per the current policy should be equal to the value function of that state for that policy:

$$\text{For } \pi \in \Pi, s \in S\text{: } Q^{\pi}(s, \pi(s)) = V^{\pi}(s).$$

   However, the Policy Iteration Algorithm implemented in policy_eval calculates an approximation of the value functions, and due to the small amount of error in this approximation, Q(s, π(a)) and V^{π}(s) may not be exactly equal in the qvalue and valuefs arrays. So in the improvactions array, these values are set to 0 via a for loop after every update (np.greater() might assign the value 1 for that state and action when it is theoretically an equality).

3. Linear Programming:

   Here, the PuLP library is used to input the objective function and constraints and solve the linear program to find the optimal value functions for each state (stored in the valuefs_soln array). After the optimal value functions are calculated, the optimal policy is found by applying the Bellman Optimality Operator to the optimal value functions and using np.argmax() to find the actions for each state for which the RHS of the Operator equation is maximum. As np.argmax breaks all ties by choosing the value with the least index, in case of more than one action resulting in the optimal value function, the action with the lowest index is chosen.

Task 2:

The mdp representing the agent is stored in a dictionary 'mdp' with the same format as task 1.

The number of states of the mdp is equal to the number of actionable states (input from the state file) and the number of possible terminal states of the tic tac toe game. The terminal states are formulated by the term_states function in encoder.py which outputs 3 arrays – the number of terminal states with no empty spots left on the grid, terminal states with empty spots left but where the agent wins, and terminal states with empty spots left where the opponent wins. The transition probabilities for all terminal states through all actions remain zero, along with their corresponding rewards.

States are encoded as 9 digit numbers while formulating the mdp, but in the mdp, states are referred to as their state ids, which are just numbers from 0 to (n-1), where n is the total number of states.

- Let a be an invalid state for state s (for example, if state s = 100000000 and a is 0). Then, T(s, a, s) = 1 and T(s, a, s') = 1 for every state s' which is not equal to s. This action would reroute the mdp back to the original state with a probability 1. However, to prevent these actions from showing up in the optimal policy after the mdp planning, the reward for this is highly negative. So, R(s, a, s) = -100 and R(s, a, s') = 0 for every state s' which is not equal to s.
- For all valid actions on a state:
  - If the agent is p1, then it is possible that the action a by the agent on state s results in a terminal state st (either the game ends in a draw or the agent loses). In this case: T(s, a, st) = 1, R(s, a, st) = 0 if it is a draw or a loss.
  - If the action a by the agent on state s results in a non-terminal state (which definitely occurs if the agent is p2), then it will result in an actionable state s2 for the opponent. Let π(s2) be the opponent's policy, which is a probability distribution over all actions. Let π(s2, a2) by the probability of the opponent taking action a when s2 is encountered as per policy π. After the opponent's action a2, the next state will either be an actionable state sa for the agent, or it will be a terminal state s2t ending with the opponent's loss (the agent's win). In the case of the agent being p2, it can also be a terminal state resulting in a draw.
    - If the opponent loses:
      - T(s, a, s2t) = π(s2, a2)
      - R(s, a, s2t) = 1
    - If another actionable state sa for the agent is reached:
      - T(s, a, sa) = π(s2, a2)
      - R(s, a, sa) = 0
    - If the agent is p2 and if it is a terminal state resulting in a draw:
      - T(s, a, s2t) = π(s2, a2)
      - R(s, a, s2t) = 0

- In the code implementation, if the state achieved after the opponent takes action a2 from s2 is a state with some empty spots left on the grid and if is not found in the array of actionable states of the agent, it is automatically assumed to be a win for the agent – since there are only three options for the nature of this state, and two have been eliminated.

As before, mdp["t"] refers to the T matrix and mdp["r"] refers to the R matrix. These are both initialised with zeros and then filled in appropriately whenever the above scenarios are encountered.

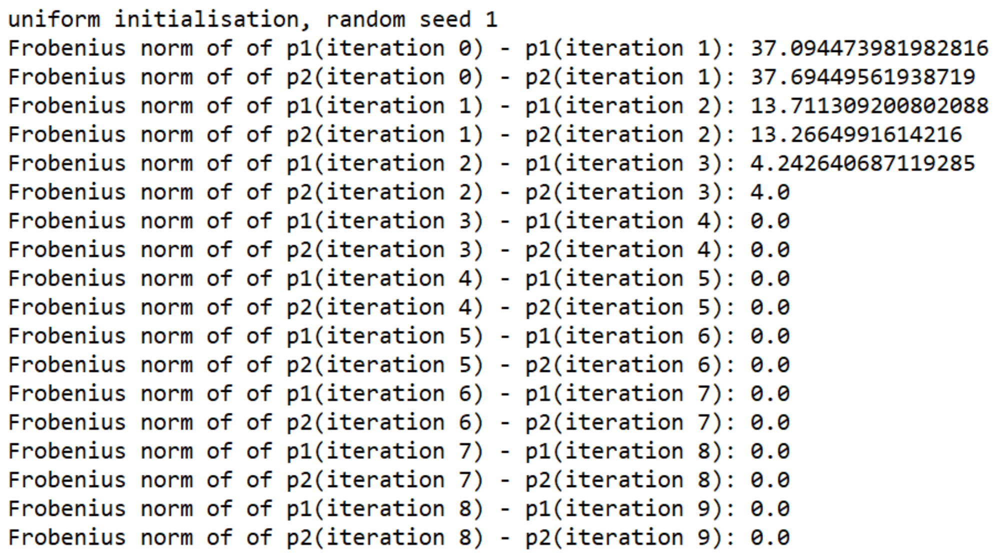
The t_r_grid function formulates the agent's mdp dictionary, which the write_to_output function writes it to an output file.

Task 3:

In task3.py, 3 ways of initialising p2's policy have been provided – a deterministic random policy ("deterministic"), a uniform random probability distribution across all the valid actions for a state ("uniform"), random non-deterministic policy ("random"). For all of these initialisations and for 5 different random seeds, the perform_iterations function was run. There are two versions of this function – one which initializes p1 in the 0th iteration and one which initializes p2 in the 0th iteration. This function performs the 10 iterations, generating 10 policies each for p1 and p2, and prints out a text file ("convergence.txt") in which the differences in p1's policies from one iteration to the next and the differences in p2's policies from one iteration to the next (measured through the Frobenius matrix norm), are recorded.

For all initialisation (type and player) - random seed combinations, it was found that the policies for p1 and p2 converge (the squared elementwise differences between the policies of the last iteration and the current one for each player become 0) within 4 iterations.

A snapshot from convergence.txt:



```
uniform initialisation, random seed 1
Frobenius norm of of p1(iteration 0) - p1(iteration 1): 37.094473981982816
Frobenius norm of of p2(iteration 0) - p2(iteration 1): 37.69449561938719
Frobenius norm of of p1(iteration 1) - p1(iteration 2): 13.711309200802088
Frobenius norm of of p2(iteration 1) - p2(iteration 2): 13.2664991614216
Frobenius norm of of p1(iteration 2) - p1(iteration 3): 4.242640687119285
Frobenius norm of of p2(iteration 2) - p2(iteration 3): 4.0
Frobenius norm of of p1(iteration 3) - p1(iteration 4): 0.0
Frobenius norm of of p2(iteration 3) - p2(iteration 4): 0.0
Frobenius norm of of p1(iteration 4) - p1(iteration 5): 0.0
Frobenius norm of of p2(iteration 4) - p2(iteration 5): 0.0
Frobenius norm of of p1(iteration 5) - p1(iteration 6): 0.0
Frobenius norm of of p2(iteration 5) - p2(iteration 6): 0.0
Frobenius norm of of p1(iteration 6) - p1(iteration 7): 0.0
Frobenius norm of of p2(iteration 6) - p2(iteration 7): 0.0
Frobenius norm of of p1(iteration 7) - p1(iteration 8): 0.0
Frobenius norm of of p2(iteration 7) - p2(iteration 8): 0.0
Frobenius norm of of p1(iteration 8) - p1(iteration 9): 0.0
Frobenius norm of of p2(iteration 8) - p2(iteration 9): 0.0
```

Furthermore, the final policies for p1 and p2 after convergence were found to simply be: **Fill in the first empty spot in the 9x9 grid.**

A snapshot of the converged policies of p1 and p2:

p2’s converged policy:

```
uniform initialisation, random seed 5, iteration 7
100000000 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
121000000 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0
121210000 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121212010 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121211200 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
121210210 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121211020 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121211002 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
121210012 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121201000 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121221100 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
121221010 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121201210 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121201120 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
121201102 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
121201012 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121200100 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121220110 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121220101 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121202110 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
121202101 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
121200121 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121200112 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121200010 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121220011 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121202011 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121200211 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121200001 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121120000 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121122010 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

p1’s converged policy:

```
uniform initialisation, random seed 5, iteration 8
000000000 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
120000000 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
121200000 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121212000 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121212210 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
121212012 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121210200 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121211220 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
121211202 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
121210212 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121210020 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121211022 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121210002 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121221000 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121221102 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
121221210 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
121221012 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
121201200 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121201212 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121201020 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121201122 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121201002 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121220100 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121220112 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
121202100 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
121202112 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
```

With the above policy, player 2 will always win (player 1 completes the diagonal).

The game will go like this:

1. P1 – 0
2. P2 – 1
3. P1 – 2
4. P2 – 3
5. P1 – 4
6. P2 – 5
7. P1 – 6
8. P2 – 7
9. P1 – 8

The sequence of policies for each player is expected to converge.

In a game such as this, both players are maximising rewards against an opponent who is playing just as strategically as them, or an opponent also seeking to maximise rewards. Strategizing a win is hard in this case, as there are no vulnerabilities or 'bad moves' of the opponent to exploit. Therefore the policies will converge so as to bring about the most evenly matched game.

Here attaining a draw and losing have the same reward, so the only incentive is to win – a draw is not any better than a loss. The game is sequential, so decision making happens on a step-by-step basis. Therefore, the agent will make the move which ensures that taking into account the current state, it does not complete a row, diagonal or column on the board during this turn (so that the game doesn't end), and it does its best to force the other player to do so. Taking this into account, it is easy to see why players 1 and 2 converge to their respective policies as shown above. For both players, this is the best policy for the agent to attempt to win at every sequential step, as it minimizes the possibility of losing – neither 1 nor 2 complete a row, column or diagonal until the second to last turn. In the last turn, player one is forced to lose because it plays one extra move than player 2 (something the mdp planner does not account for), and the last open spot on the board is the one by which it completes a diagonal. Before this final step, this is an evenly matched game.