Assignment 1 Report

Reinforcement Learning CISC474

Shreyansh Anand 20053370

Anne Liu 20069271

Part 1: Context of the GridWorld Problem

The GridWorld problem in class discusses a Markovian environment with a grid example. The task is to find the optimal state-value number which would indicate the correct policy to follow to get the most reward in the grid configuration.

Dynamic Programming Solution

The program is implemented in Python and solves the GridWorld problem for a grid environment of 5x5 and 7x7 where the placement of A,A',B,B' are different. The program takes in the parameters of grid size (dimension of the grid) and discount rate.

The program's main method is the *grid_iterations()* method which loops through iterations of applying the linear equation to the grid and uses policy evaluation where the program will stop iterating once the state-value for iterations converge. The resulting values are assumed to reflect the 'optimal policy'.

Solution

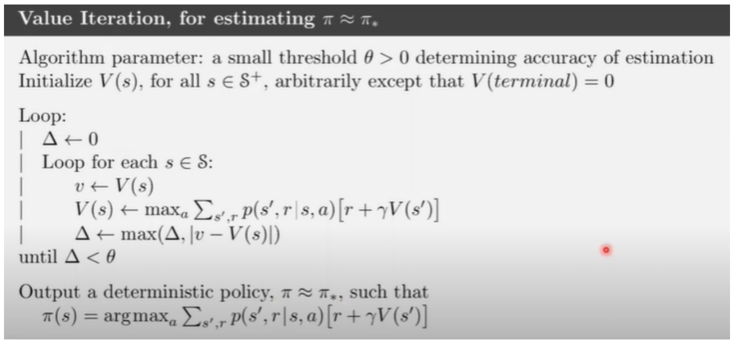
The program first initiates the values needed by instantiating global values within the class. Depending on the dimension (5x5 or 7x7) the position of A, A', B, B' are set as per the assignment instructions. Theta is set to 0.01 to facilitate convergence.

The process function applies the optimal equation for v* to the grid in iterations. The function applies the function manually depending on where the player is on the grid, taking into account the environment and rules of the scenario (for instance, corners and edges are special cases as well as A and B). The Bellman optimality equation applied is as follows:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma v_*(s')\right]. \end{aligned}$$

Dynamic Programming- Value Iteration

The process above is iteratively run and on iteration, policy evaluation is run where we compare the difference of values between iterations to a theta value that is set above. Once, delta is less than theta, the values have converged and the while loop terminates. This process will output the optimal policy. This process is based on policy iteration discussed in class.



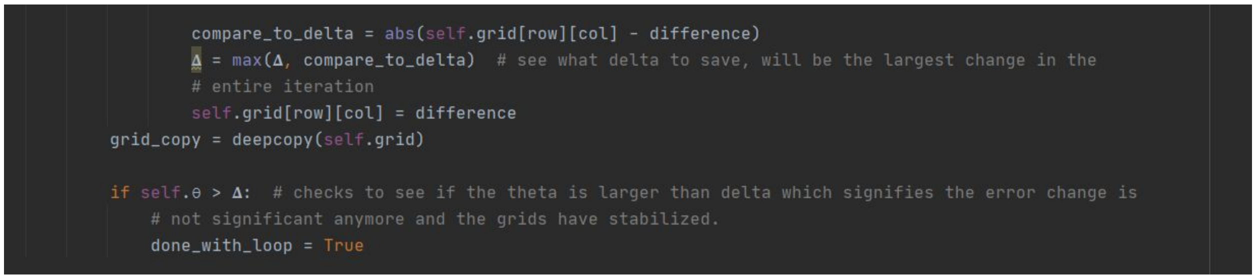
Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
```

The program is based on the Value Iteration algorithm. Some benefits of calculating everything in one loop instead of two allows the program to conduct a single sweep for policy and generate better performance.



```
                compare_to_delta = abs(self.grid[row][col] - difference)
                Δ = max(Δ, compare_to_delta)  # see what delta to save, will be the largest change in the
                # entire iteration
                self.grid[row][col] = difference
        grid_copy = deepcopy(self.grid)

        if self.θ > Δ:  # checks to see if the theta is larger than delta which signifies the error change is
            # not significant anymore and the grids have stabilized.
            done_with_loop = True
```

Results

5x5 Case: 0.85 discount rate

```
[2.9, 9.1, 4.1, 5.2, 1.2]
[1.2, 2.7, 1.9, 1.6, 0.3]
[-0.1, 0.6, 0.5, 0.3, -0.4]
[-0.9, -0.3, -0.3, -0.5, -1.0]
[-1.6, -1.1, -1.0, -1.2, -1.7]
```

5x5 Case: 0.75 discount rate

```
[2.2, 9.4, 3.3, 5.1, 0.7]
[0.7, 2.2, 1.3, 1.2, 0.1]
[-0.2, 0.4, 0.3, 0.2, -0.4]
[-0.7, -0.3, -0.2, -0.3, -0.8]
[-1.3, -0.8, -0.7, -0.8, -1.3]
```

7x7 Case: 0.85 discount rate

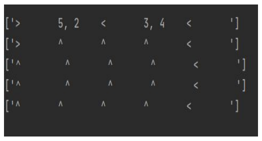
```
[-0.4, 0.3, 0.2, 0.3, 1.3, 4.9, 1.0]
[1.0, 2.5, 1.3, 0.7, 0.8, 1.3, 0.2]
[2.7, 9.1, 2.7, 0.9, 0.4, 0.3, -0.4]
[1.0, 2.5, 1.3, 0.5, 0.1, -0.2, -0.7]
[-0.1, 0.5, 0.3, 0.1, -0.1, -0.4, -1.0]
[-0.9, -0.4, -0.3, -0.3, -0.5, -0.7, -1.3]
[-1.6, -1.1, -1.0, -1.0, -1.1, -1.3, -1.8]
```

7x7 Case: 0.75 discount rate

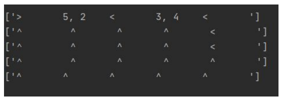
```
[-0.7, -0.0, -0.1, -0.0, 0.9, 4.9, 0.7]
[0.5, 2.0, 0.8, 0.3, 0.5, 1.0, -0.0]
[2.1, 9.4, 2.2, 0.6, 0.2, 0.1, -0.4]
[0.6, 2.1, 0.9, 0.3, 0.1, -0.1, -0.6]
[-0.3, 0.3, 0.2, 0.0, -0.1, -0.2, -0.7]
[-0.7, -0.3, -0.2, -0.2, -0.3, -0.4, -0.9]
[-1.3, -0.8, -0.7, -0.7, -0.7, -0.9, -1.3]
```

Policies

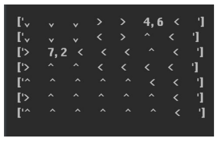
The optimal policies generated support the common sense hypothesis. Each action moves the player to the grid that moves it closer to either A or B (shown by the arrows pointing towards the coordinates of A and B) as more rewards will be generated once the player Travels to A' or B'.



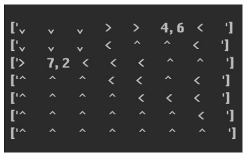
```
['>    5, 2   <    3, 4   <    ']
['>    ^      ^    ^      <    ']
['^    ^      ^    ^      <    ']
['^    ^      ^    ^      <    ']
['^    ^      ^    ^      <    ']
```



```
['>    5, 2   <    3, 4   <    ']
['^    ^      ^    ^      <    ']
['^    ^      ^    ^      <    ']
['^    ^      ^    ^      ^    ']
['^    ^      ^    ^      ^    ']
```



```
['v   v    v   >   >   4, 6  <   ']
['v   v    v   <   >   ^     <   ']
['>   7, 2 <   <   <   ^     <   ']
['>   ^    ^   <   <   <     <   ']
['^   ^    ^   ^   ^   <     <   ']
['>   ^    ^   ^   ^   ^     <   ']
['^   ^    ^   ^   ^   ^     <   ']
```



```
['v   v    v   >   >   4, 6  <   ']
['v   v    v   <   ^   ^     <   ']
['>   7, 2 <   <   <   ^     ^   ']
['^   ^    ^   <   <   ^     <   ']
['^   ^    ^   ^   <   <     <   ']
['^   ^    ^   ^   ^   ^     <   ']
['^   ^    ^   ^   ^   ^     ^   ']
```

Policies generated by the dynamic programming algorithm (Cases are in the same order as the results)

Part 2 Two-Player Matrix Games: Prisoners' Dilemma- Context

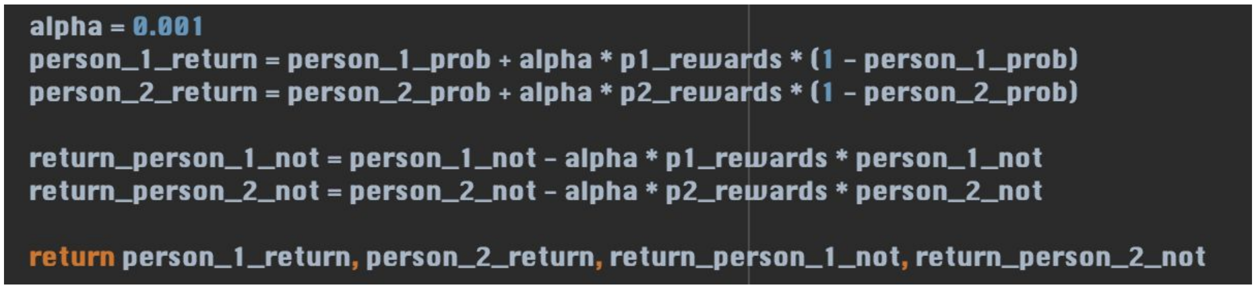
The Prisoner's Dilemma game involves two criminals that need to decide whether they will cooperate with the police and defect on the accomplice or whether they will cooperate with the accomplice and lie to the police

Solution

The solution applies a variation of the TD algorithm discussed in class, calculating the policies upon many iterations. Upon every visit to the state, a policy iteration is performed. States are selected according to the probability matrix and the following update rule is used:

$$p_c^j(k+1) = p_c^j(k) + \alpha r^j(k)\left[1 - p_c^j(k)\right], \qquad \textit{if action } c \textit{ is taken at time } t$$

$$p_o^j(k+1) = p_o^j(k) - \alpha r^j(k) p_o^j(k), \qquad \textit{for all other actions } o \neq c$$



```
alpha = 0.001
person_1_return = person_1_prob + alpha * p1_rewards * (1 - person_1_prob)
person_2_return = person_2_prob + alpha * p2_rewards * (1 - person_2_prob)

return_person_1_not = person_1_not - alpha * p1_rewards * person_1_not
return_person_2_not = person_2_not - alpha * p2_rewards * person_2_not

return person_1_return, person_2_return, return_person_1_not, return_person_2_not
```

The policy to update probabilities

Results and Optimality

The optimal policy is one that determines the best action to take at each state. As seen on the graphs 1.1 and 1.2 in the Appendix, the actions of player one and two converge to: person 1 probability: [0.0, 1.0] person 2 probability: [0.0, 1.0]. This shows the action of defection as the optimal policy because there is no better action to take given the other opponent will continue the same action of defection (As stated in the Nash equilibrium game theory).

final game value for person 1: 0.999999999999889

final game value for person 2: 0.999999999999889

Two-Player Matrix Games: Matching Pennies- Context

The matching pennies game consists of two players each holding a penny and independently showing a side per play. If both pennies show the same figure (two heads or two tails), player 1 wins the penny and has a reward of 1. If the pennies show different figures, player 2 wins the penny and player 1 has a reward of -1.

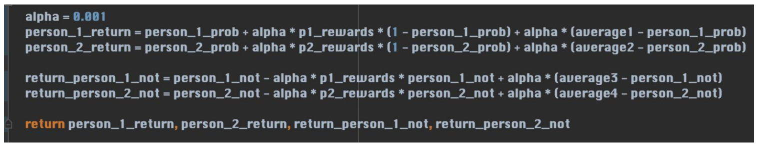
Solution

Two solutions were implemented. The first solution closely follows the implementation of the Prisoners' Dilemma problem above.

For the second solution, another term is added as an alternative solution to facilitate convergence up to a certain threshold (as the first algorithm is a 'mixed' algorithm). The algorithm also iteratively calculates optimal policies.

$$p_c^j(k+1) = p_c^j(k) + \alpha r^j(k)[1 - p_c^j(k)] + \alpha\{\mathbb{E}[p_c^j(k)] - p_c^j(k)\}, \qquad \textit{if action c is taken at time t}$$

$$p_o^j(k+1) = p_o^j(k) - \alpha r^j(k) p_o^j(k) + \alpha\{\mathbb{E}[p_o^j(k)] - p_o^j(k)\}, \qquad \textit{for all other actions } o \neq c$$



```
alpha = 0.001
person_1_return = person_1_prob + alpha * p1_rewards * (1 - person_1_prob) + alpha * (average1 - person_1_prob)
person_2_return = person_2_prob + alpha * p2_rewards * (1 - person_2_prob) + alpha * (average2 - person_2_prob)

return_person_1_not = person_1_not - alpha * p1_rewards * person_1_not + alpha * (average3 - person_1_not)
return_person_2_not = person_2_not - alpha * p2_rewards * person_2_not + alpha * (average4 - person_2_not)

return person_1_return, person_2_return, return_person_1_not, return_person_2_not
```

The policy to update the probabilities are similar as the one above, but with an anchor.

Results and Optimality

As seen on the graphs 2.1, 2.2 (algorithm not utilizing the anchor) compared to 2.3 and 2.4 (algorithm using anchor) in the Appendix, the anchor allows the problem that does not have a "pure strategy" to converge. The actions of player one and two converge to: person 1 probability: [0.7, .3] person 2 probability: [0.6, 0.4] with the anchor ; without anchor it is: person 1 probability: [0.5046, 0.4954] person 2 probability: [0.5005, 0.4995] . The policy (with anchor) is thus 50/50 and is optimal because both possible actions lead to better results in the game strategy and this is shown with the converging values in figures 2.3/2.4. Game values are shown below:

Pennies without anchor final value for person 1: 0.014705637822002647 final value for person 2: -0.014705637822002626	**Pennies without anchor** final value for person 1: -4.293967922018707e-07 final value for person 2: 4.2939679220188424e-07

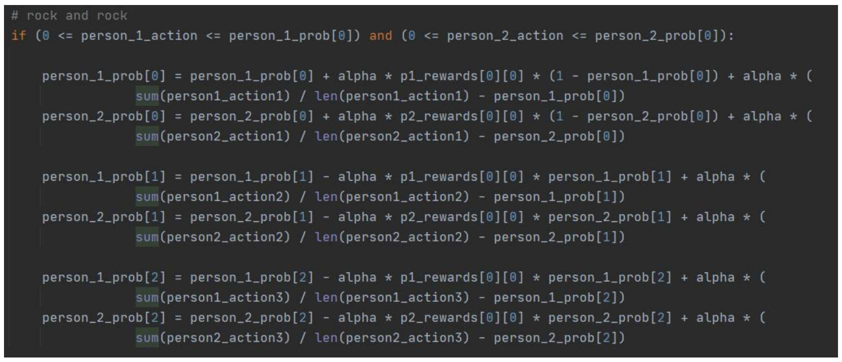
:

Two-Player Matrix Games: Rock Paper Scissors- Context

The game of rock paper scissors follows the generally known rules of paper covers rock, rock breaks scissors, and scissors cuts paper. If both players display the same thing, it is a tie.

Solution

The two solutions resemble the solutions of the Pennies game. The update algorithm is applied with and without the anchor. Actions were chosen based on probabilities and were reflected in a probability range. For instance, the probability at the first epoch of choosing a rock is between 0 and 0.6 (person_2_prob[0]).



```python
# rock and rock
if (0 <= person_1_action <= person_1_prob[0]) and (0 <= person_2_action <= person_2_prob[0]):

    person_1_prob[0] = person_1_prob[0] + alpha * p1_rewards[0][0] * (1 - person_1_prob[0]) + alpha * (
                sum(person1_action1) / len(person1_action1) - person_1_prob[0])
    person_2_prob[0] = person_2_prob[0] + alpha * p2_rewards[0][0] * (1 - person_2_prob[0]) + alpha * (
                sum(person2_action1) / len(person2_action1) - person_2_prob[0])

    person_1_prob[1] = person_1_prob[1] - alpha * p1_rewards[0][0] * person_1_prob[1] + alpha * (
                sum(person1_action2) / len(person1_action2) - person_1_prob[1])
    person_2_prob[1] = person_2_prob[1] - alpha * p2_rewards[0][0] * person_2_prob[1] + alpha * (
                sum(person2_action2) / len(person2_action2) - person_2_prob[1])

    person_1_prob[2] = person_1_prob[2] - alpha * p1_rewards[0][0] * person_1_prob[2] + alpha * (
                sum(person1_action3) / len(person1_action3) - person_1_prob[2])
    person_2_prob[2] = person_2_prob[2] - alpha * p2_rewards[0][0] * person_2_prob[2] + alpha * (
                sum(person2_action3) / len(person2_action3) - person_2_prob[2])
```

Results and Optimality

As seen on the graphs 3.1, 3.2 (algorithm not utilizing the anchor) compared to 3.3 and 3.4 (algorithm using anchor) in the Appendix, the anchor allows the problem that does not have a "pure strategy" to converge. the actions of player one and two converge to: person 1 probability: [0.3323, 0.3097, 0.358] person 2 probability: [0.3314, 0.3025, 0.3661] with the anchor ; without anchor it is: person 1 probability: [0.1739, 0.4636, 0.3625] person 2 probability: [0.3767, 0.0256, 0.5977]. The policy (with anchor) that each action is equally likely and this is optimal because this strategy leads to the most beneficial results (Nash Equilibrium) and this is reflected by the converging values in figures 3.3/3.4. Game values are shown below. **Optimality for all problems is when the policy used is the best policy considering the other person does not change their own strategy. This is because we cannot rely on cooperativity of the other person. This idea holds for Prisoners, Pennis as well as Rock Paper Scissors.**

Pennies without anchor final value for person 1: -0.1302533486397415 final value for person 2: -0.13025334863974153	**Pennies without anchor** final value for person 1: 4.115078046504003e-05 final value for person 2: 4.11507804650383e-05

Appendix

*** The following graphs show graphs where the x axis as the episodes and y axis as the probability of action

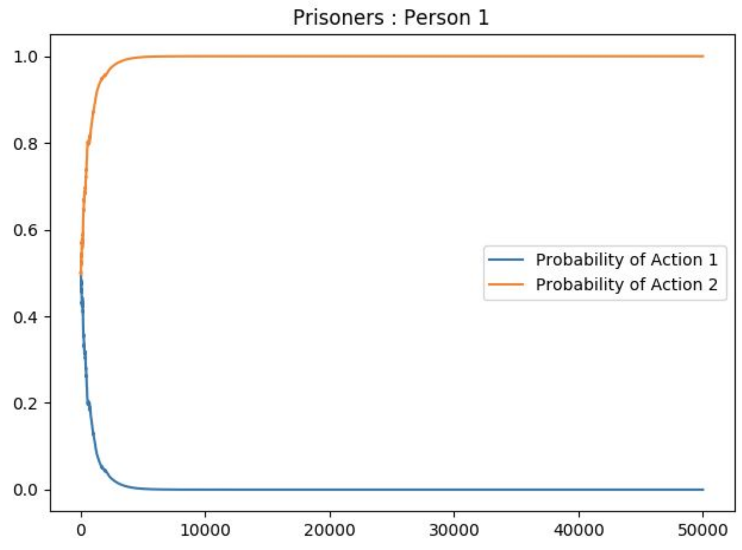


Figure 1.1 probability values for player 1 (prisoner's dilemma)

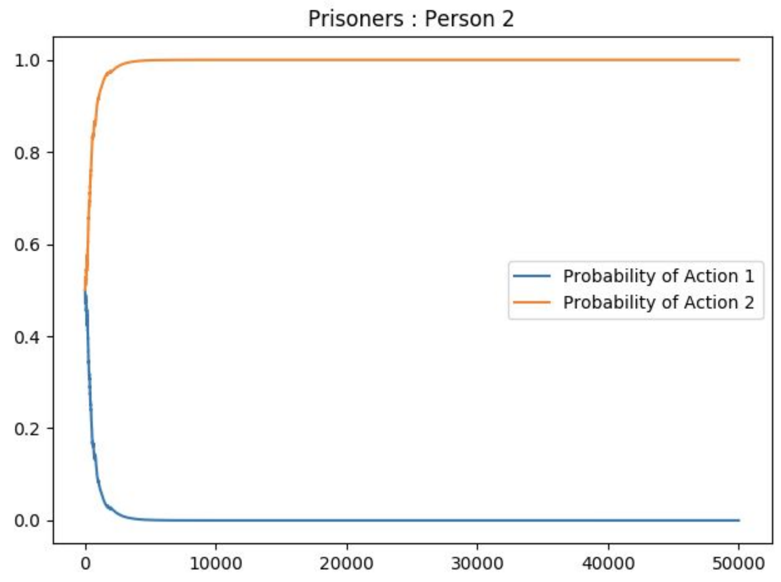


Figure 1.2 probability values for player two (prisoner's dilemma)

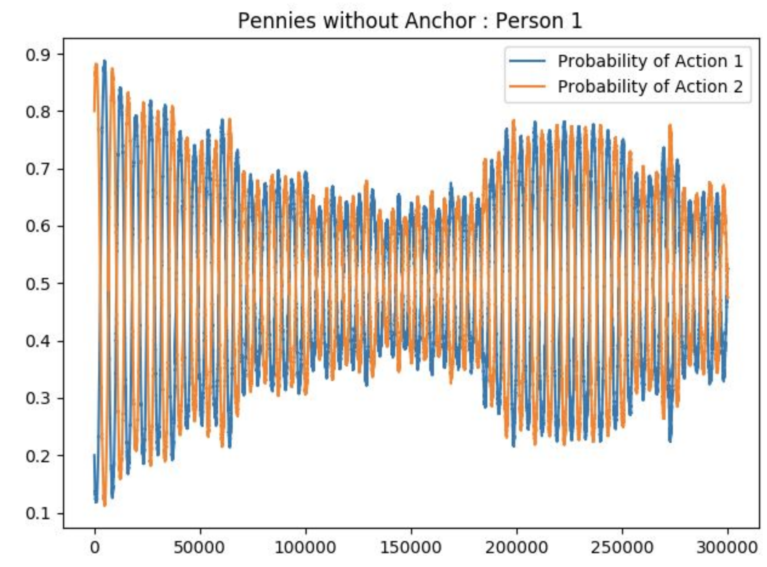


Figure 2.1 probability values for player 1 (no anchor)

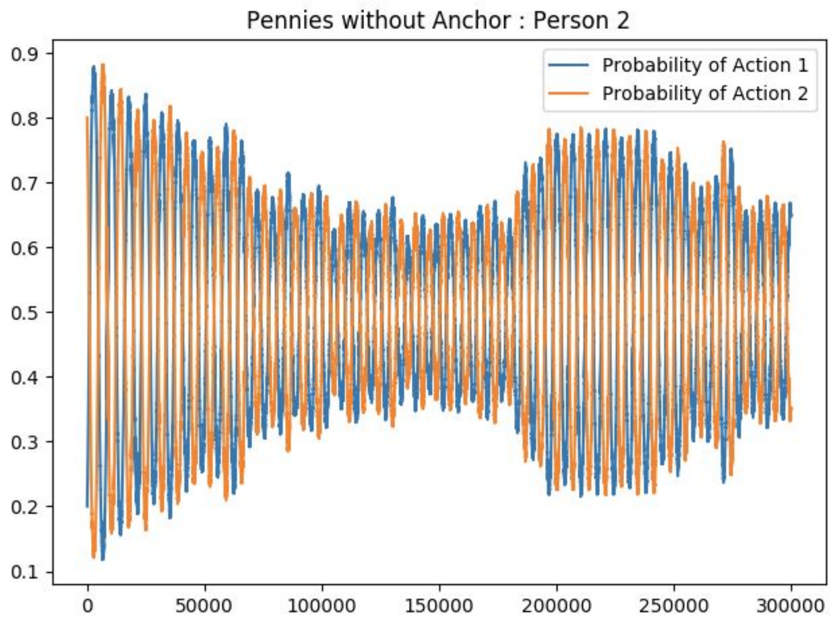


Figure 2.2 probability values for player 2 (no anchor)

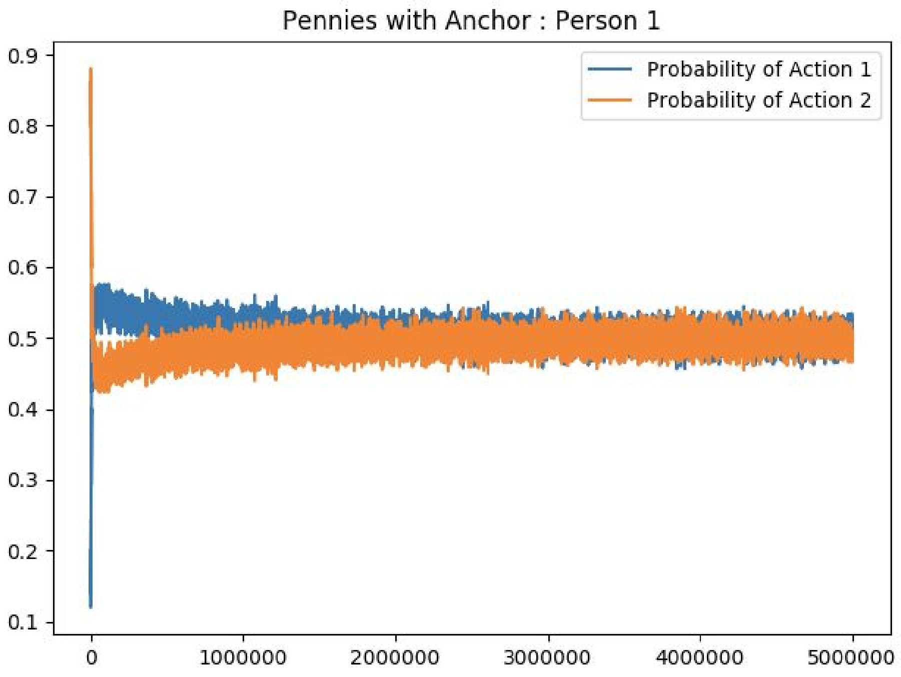


Figure 2.3 probability values for player 1 (anchored)

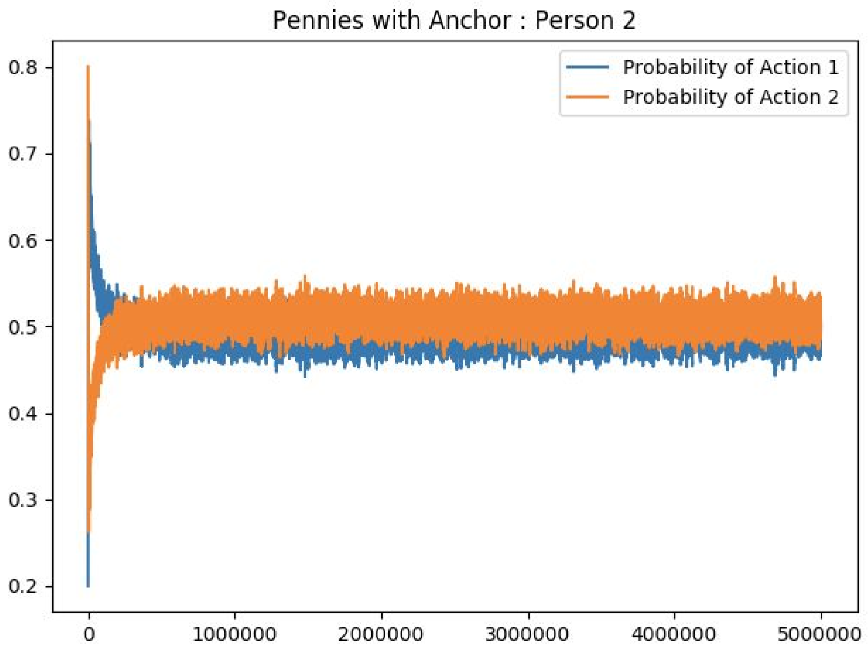


Figure 2.4 probability values for player 2 (anchored)

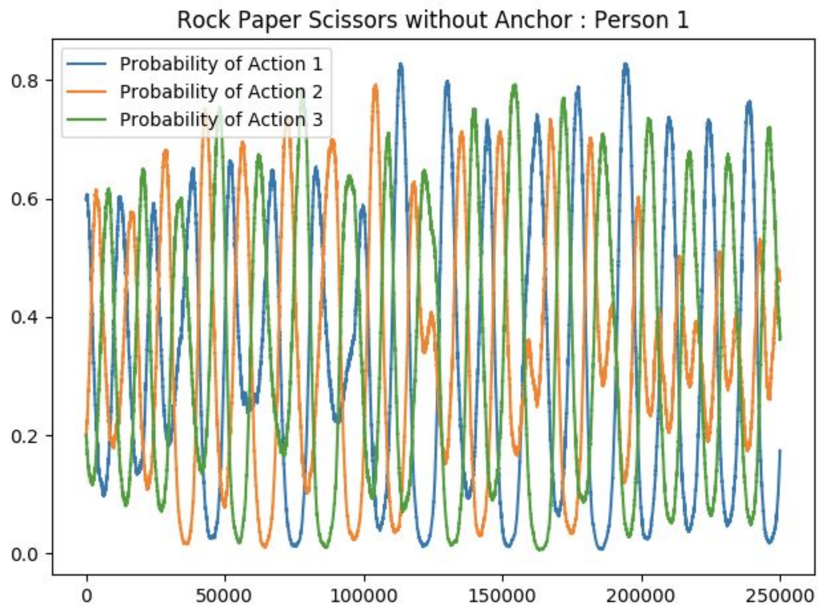


Figure 3.1 probability values for player 1 (no anchor)

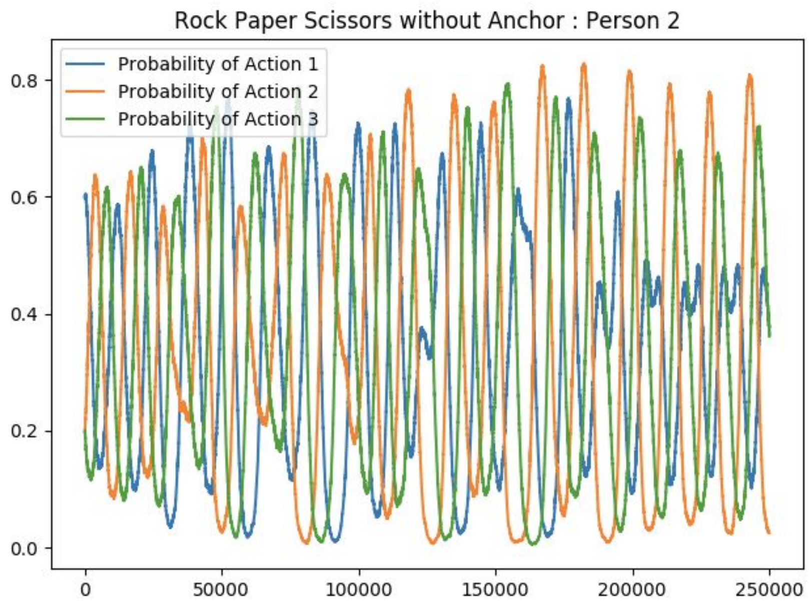


Figure 3.2 probability values for player 2 (no anchor)

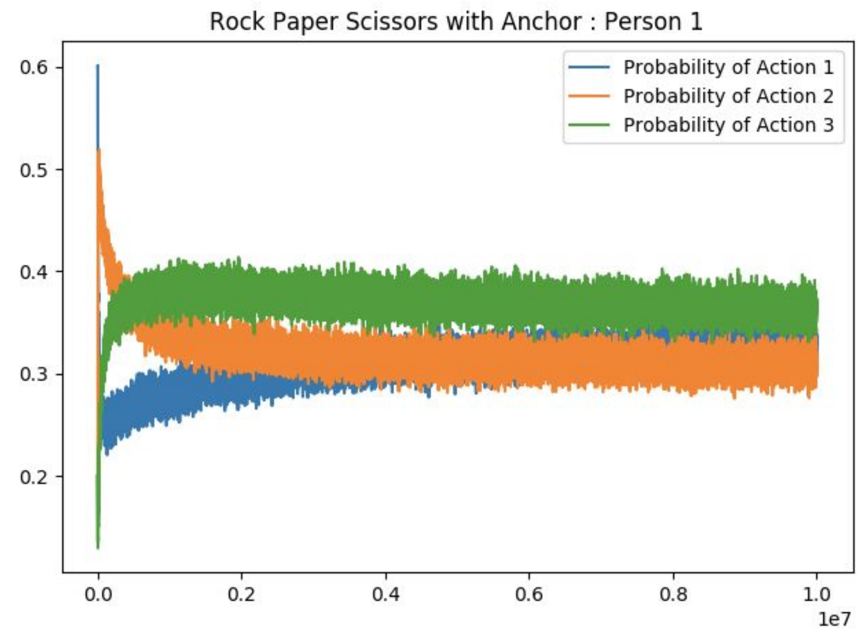


Figure 3.3 probability values for player 1 (anchored)

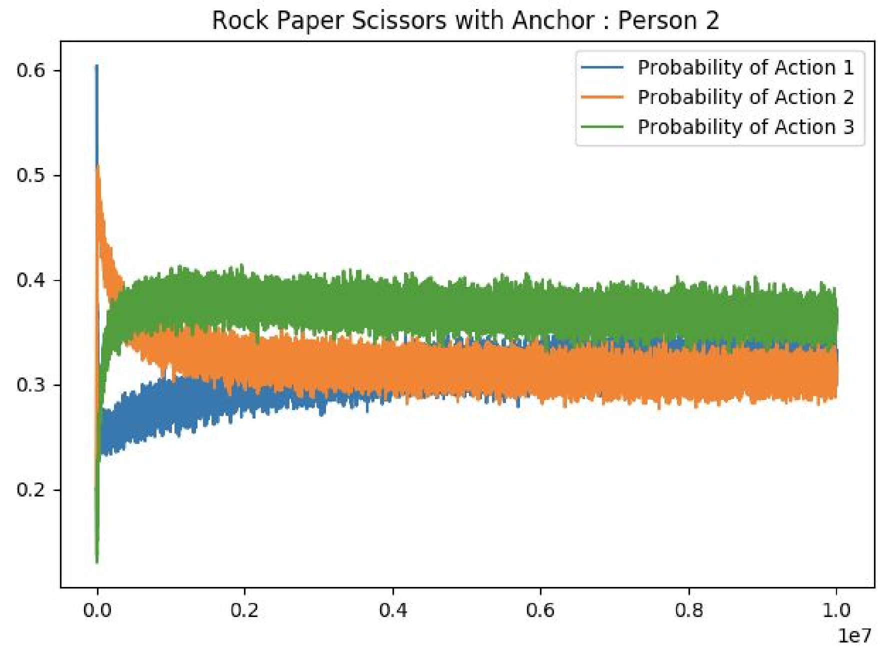


Figure 3.4 probability values for player 2 (anchored)