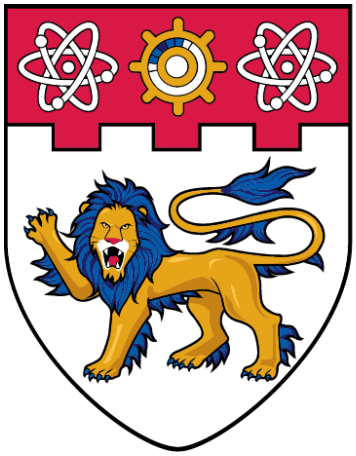


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Assignment 2 : Three Prisoners Dilemma

CZ4046 – INTELLIGENT AGENTS
(Semester 2, AY 2019/2020)

WILSON THURMAN TENG
U1820540H

1. Overview

1.1 Problem Description

```
int rounds = 90 + (int)Math.rint(20 * Math.random()); // Between 90 and 110 rounds
```

Fig.1 - Number of Rounds

Design a strategy for an agent to perform optimally in a 3-person Repeated Prisoner's Dilemma (PD) game. Each game will be between 90 to 110 rounds, randomly decided (as shown in *Fig.1*).

1.2 Score Chart

Fig.2 - Score Chart

C = cooperate, D = defect

(A, B, C)	**POINT SYSTEM** *(higher is better)*
(C, D, D)	0
(D, D, D)	2
(C, C, D) (C, D, C)	3
(D, C, D) (D, D, C)	5
(C, C, C)	6
(D, C, C)	8

As we can see from the score chart, the action combination that fetches the most points in 1 round is the one where our agent defects and both opponent agents cooperate. However, that would disincentivize other agents to cooperate in future rounds. As such, we should aim to design our agent to maximise cooperation while also aiming to prevent exploitation by other nasty agents.

1.3 Prisoner's Dilemma Matrix

IF A DEFECTS

		C	
		defect	no defect
B	defect	(2, 2, 2)	(5, 5, 0)
	no defect	(5, 0, 5)	(8, 3, 3)

IF A DOES NOT DEFECT

		C	
		defect	no defect
B	defect	(0, 5, 5)	(3, 8, 3)
	no defect	(3, 3, 8)	(6, 6, 6)

where (SCORE1, SCORE2, SCORE3) == (A, B, C)

Fig.3 - PD Matrix

We can observe from the PD matrix that the action combination that gives the highest average score for all agents is [C, C, C] at 6 per agent.

2. Agent Evaluation

2.1 Description:

After our analysis in previous sections, it is critical that our agent cooperates to maximise it's score in the long run. However, we should be wary of exploitation from other agents.

To evaluate our agent, the steps below are followed:

1. An environment with the strategies given in the template is first created.
2. The tournament of 100 rounds will then be run 10000 times to obtain the cumulative rankings of all the players.
3. Divide the cumulative rankings of all players by 10000 to obtain the average ranking for our agent.

Step 1 helps to simulate the variety of strategies in the actual tournament. The measure from Step 3 gives us a good gauge of how consistently our agent performs compared to other agents.

2.2 Added Agents

To make the environment more dynamic, 3 additional strategies have been added:

- **WinStayLoseShift** :
    - Starts with cooperative action.
    - If the previous action grants the agent either 6 or 8 points, return the same action. Else, return the opposite action.
- **Trigger** :
    - Starts with cooperative action.
    - If both opponents defected in the last round, defect for subsequent rounds regardless.
- **Alternate** :
    - Starts with cooperative action.
    - Return the opposite of the previous action for subsequent rounds.

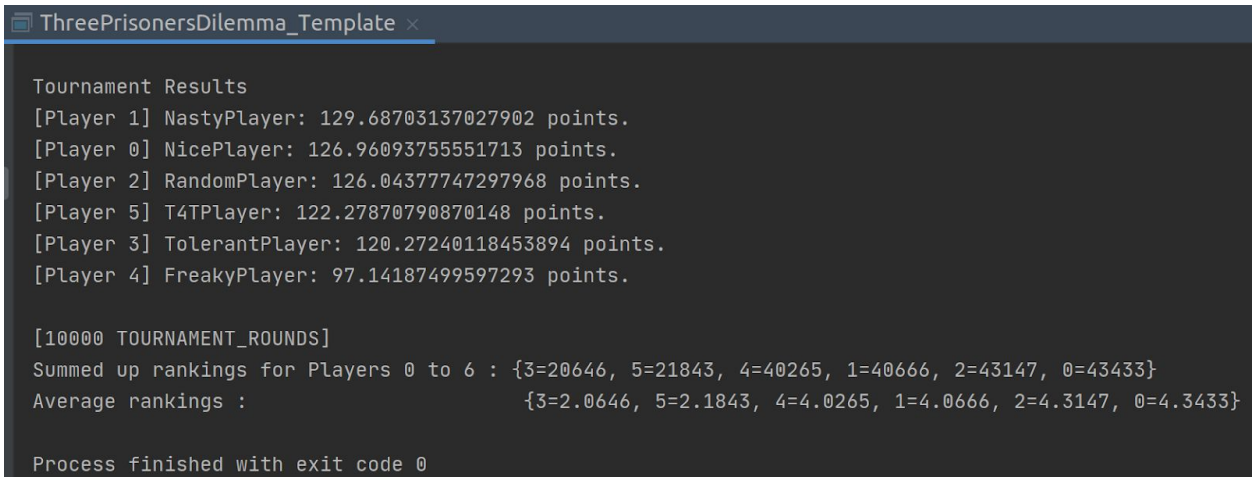
2.3 Code implementation

For the code implementation of the modified main method discussed in 2.1, and the 3 added agents, refer to the Code Appendix.

3. Initial Observations

The setup found in this section is available at this link.

3.1 Original template (Link)



The rankings are as follows:

1) [P3] TolerantPlayer
2) [P5] T4TPlayer
3) [P4] FreakyPlayer
4) [P1] NastyPlayer
5) [P2] RandomPlayer
6) [P0] NicePlayer

We can observe that strategies that are unreactive (e.g. nasty, nice or random) do not perform well. Instead, strategies that react to the opponent's actions generally perform better. This highlights the importance of reactiveness for our agent.

3.2 Extra Agents added (Link)

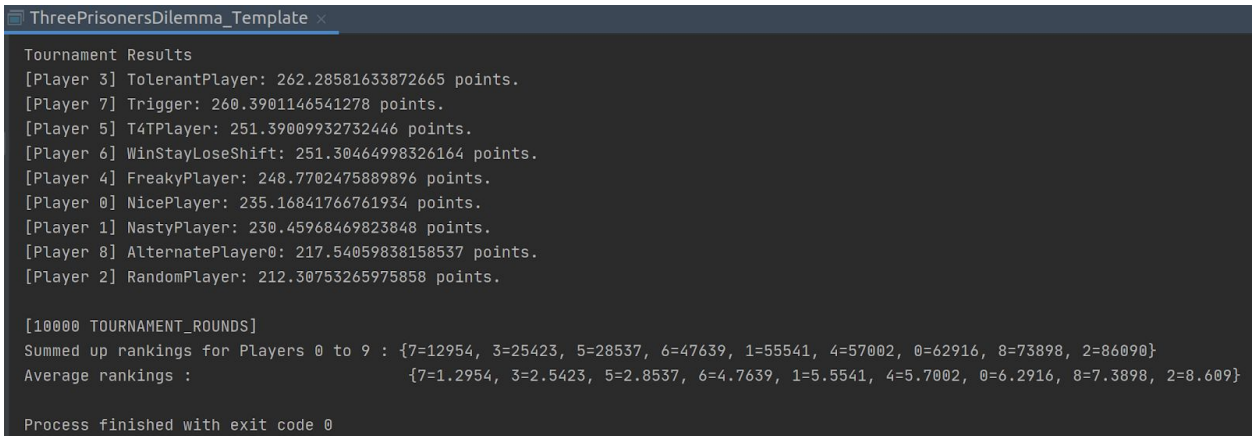


Fig. 5

The rankings are as follows (Added agents are in bold):

1) **[P7] Trigger**
2) [P3] TolerantPlayer
3) [P5] T4TPlayer
4) **[P6] WinStayLoseShift**
5) [P1] NastyPlayer
6) [P4] FreakyPlayer
7) [P0] NicePlayer
8) **[P8] AlternatePlayer0**
9) [P2] RandomPlayer

After the Trigger, WinStayLoseShift and Alternate strategies are added, we see that the tolerant and T4T agents are still performing well. More interestingly, Trigger is at the top of the rankings. This suggests that harsh punishment to opponents is crucial for our agent design.

3.3 Learning Points

From our initial observations, our agent should possess the following properties to perform well:

- Be Reactive
- Punish defects harshly

4. Agent Design

4.1 Assumptions

The design of this agent was formed under the following assumptions:

- Opponents are relatively well-versed in game theory since it was taught in the CZ4046 course. This implies that:
    - Opponents will try to be as cooperative as possible while being cautious (perhaps excessively) against exploitations.
- Opponents are highly incentivised to generate complex strategies since this tournament is part of the course grade. This implications are two-fold:
    - Opponent strategies will be hard to decode.
    - There will not be any destructive or self-destructive agents.

Therefore, this agent will be cooperative whenever opponents have proven to be, and will not try to decode the strategies of opponent agents. Also, since we assume opponents are excessively cautious of cooperating, this agent will depend on the most important measure — its own score, to decide instead.

4.2 Rules

The following rules will be followed:

- [Rule 0] “Be trustworthy and unpredictable at the same time.”
- [Rule 1] “Protect myself.”
- [Rule 2] “Cooperate in a cooperative environment.”
- [Rule 3] “If I am losing, turn it into a lose-lose situation.”

Rule 0 will be the overarching rule, while rules 1 to 3 will be followed in order.

➢ [Rule 0 : “Be trustworthy and unpredictable at the same time.”]

We want our agent to be trusted and cooperative so that our opponent can always cooperate with it. However, we do not want to be unpredictable and cause our agent to be vulnerable to exploitation. Therefore, this agent will perform its intended action 99% of the time.

➢ [Rule 1 : “Protect myself.”]

If we can accurately determine our opponents probability to cooperate (e.g. after 100 rounds), we can predict with a certain confidence that the agent is prone to defecting. In the situation where we observe that the opponents are not as cooperative as we hope, we will defect.

➢ [Rule 2 : “Cooperate in a cooperative environment.”]

Before we are able to accurately decide whether our opponents are relatively nasty or not, we use a more lenient threshold to decide if opponents are cooperative. If opponents were cooperative in the round, cooperate with them.

➢ [Rule 3 : “If I am losing, turn it into a lose-lose situation.”]

Opponents do not seem overly cooperative or defective. Our agent is unsure of what to do. Therefore, focus on self-preservation in the hopes that the opponent will make a mistake. If our agent does not have the highest score, defect. Else, cooperate.

4.3 Agent Implementation

Snippets of the code implementation of the agent is shown below. For complete code, please refer to the code appendix or this GitHub Link.

```java
/**
 * Law [#0]: This utility method introduces noise to an agent's action, allowing it to be unpredictable.
 * @param intendedAction The agent's intended action.
 * @param percent_chance_for_intended_action The percentage chance the agent will perform it's intended action.
 * @return The agent's final action.
 */
private int actionWithNoise(int intendedAction, int percent_chance_for_intended_action) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>() {{
        put(intendedAction, percent_chance_for_intended_action);
        put(oppAction(intendedAction),  v: 1-percent_chance_for_intended_action);
    }};
    LinkedList<Integer> list = new LinkedList<>();
    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
        for (int i = 0; i < entry.getValue(); i++) {
            list.add(entry.getKey());
        }
    }
    Collections.shuffle(list);
    return list.pop();
}
```

Rule 0

```java
/** [PROTECT MYSELF]: -> Law [#1]
 When it is nearing the end of the tournament at 100 rounds, if both players are known to be relatively nasty
 (cooperate less than 75% of the time). Defect to protect myself.
 */
if ((n>100) && (opponent1Coop_prob<STRICT_THRESHOLD && opponent2Coop_prob<STRICT_THRESHOLD)) {
    // Law [#0] Added
    return actionWithNoise( intendedAction: 1,  percent_chance_for_intended_action: 99);
}

/** [REWARD COOPERATION]: -> Law [#2]
 At any point in time before we are able to accurately decide if opponents are nasty or not. We set a lenient
 threshold (0.705) to gauge if opponents are cooperative. Additionally, we check if both opponent's last action
 was to cooperate. If yes, we will cooperate too.
 */
if ((opp1LA+opp2LA ==0)&&(opponent1Coop_prob>LENIENT_THRESHOLD && opponent2Coop_prob>LENIENT_THRESHOLD)) {
    // Law [#0] Added
    return actionWithNoise( intendedAction: 0,  percent_chance_for_intended_action: 99);
}
else
/** [I WILL NOT LOSE] -> Law [#3]
 However, if opponent is not cooperative, we will check if we have the highest score.
 If we have the highest score, we are appeased and will cooperate. Else, we will defect.
 */
    return SoreLoser();
```

Rules 1 to 3

```java
/** Law [#3]:
 * Cooperates if agent currently has the highest score, else defect.
 * @return
 */
private int SoreLoser() {
    if (iAmWinner()) return 0;
    return 1;
}
/* Function to check if agent is loser or not. Agent is a winner if it has the highest score. */
private boolean iAmWinner() {
    if (myScore>=opp1Score && myScore>=opp2Score) {
        return true;
    }
    return false;
}
```

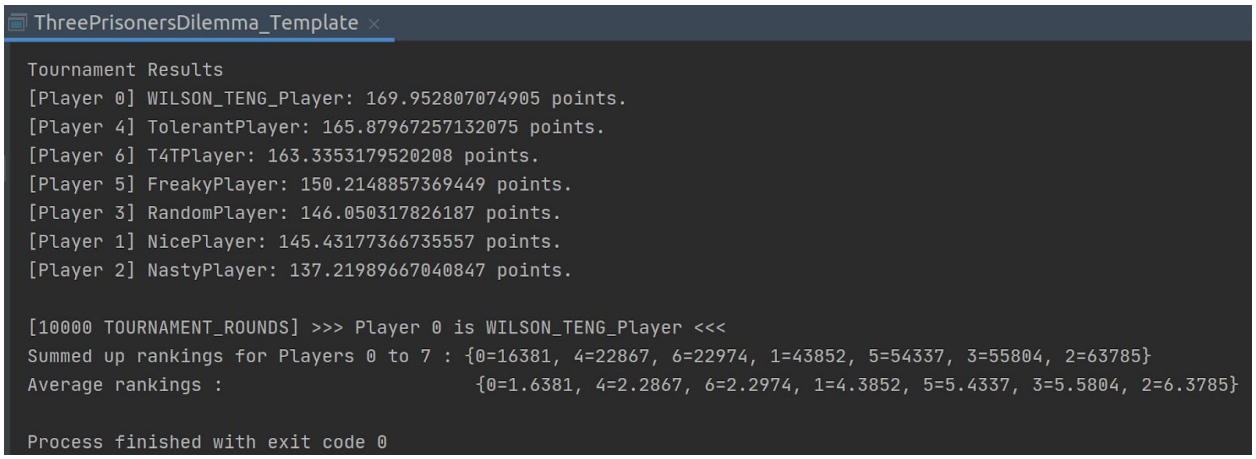
Rule 3 (Further Details)

As shown above, Rule 0 is applied to the other rules 1 and 2. Rule 3 is applied as the last resort, when we are unsure of whether the agent is defective or cooperative.

5. Agent Performance

The setup found in this section is available at this link.

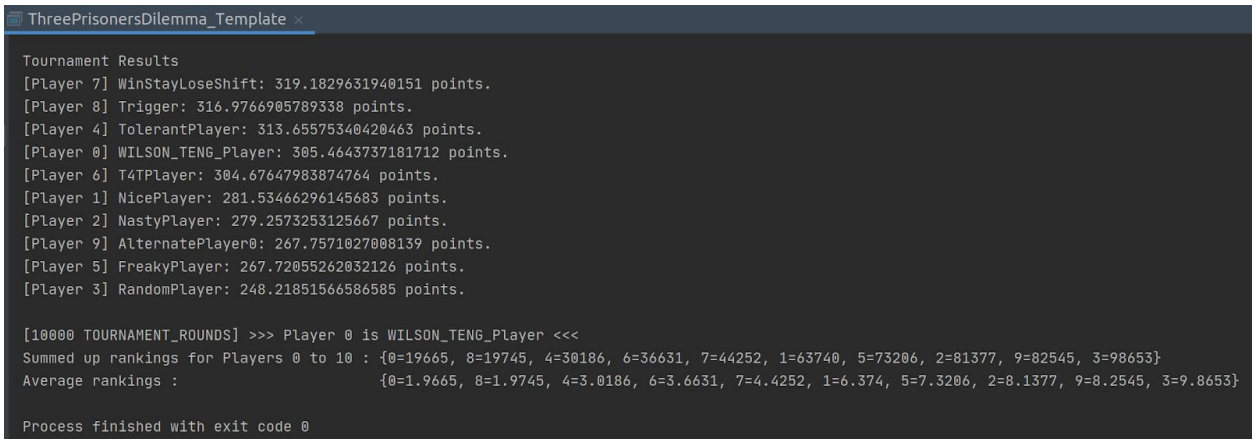
5.1 Original template (Link)



The rankings are as follows:

1) **[P0] WILSON_TENG_Player**
2) [P4] TolerantPlayer
3) [P6] T4TPlayer
4) [P1] NicePlayer
5) [P5] FreakyPlayer
6) [P3] RandomPlayer
7) [P2] NastyPlayer

5.2 Extra Agents added (Link)



The rankings are as follows:

1) **[P0] WILSON_TENG_Player**
2) **[P8] Trigger**
3) [P4] TolerantPlayer
4) [P6] T4TPlayer
5) **[P7] WinStayLoseShift**
6) [P1] NicePlayer
7) [P5] FreakyPlayer
8) [P2] NastyPlayer
9) **[P9] AlternatePlayer0**
10) [P3] RandomPlayer

The rankings observed here are relatively similar to section 5.1, with the exception of the random and nasty agents swapping rankings.

5.3 Performance Comparison against template strategies

5.3A [WILSON_TENG_Player] VS [Trigger]

The average ranks for "WILSON_TENG_Player" and "Trigger" of 1.9665 and 1.9745 respectively are relatively close. However, "WILSON_TENG_Player" wins by a margin perhaps due to the following rules working in synergy.

- ***[Rule 1 : "Protect myself."]***
    - Employed when Trigger becomes a NastyPlayer.
- ***[Rule 2 : "Cooperate in a cooperative environment."]***
    - Employed when Trigger is still a NicePlayer before being Triggered.

5.3B [WILSON_TENG_Player] VS [TolerantPlayer]

"WILSON_TENG_Player" is superior to "TolerantPlayer" because of the additional layer of reactivity that "WILSON_TENG_Player" has. While "TolerantPlayer" relies solely on the probability of cooperation of the opposing players to make its decisions, "WILSON_TENG_Player" also takes the opponent's previous actions into account in order to satisfy ***[Rule 2 : "Cooperate in a cooperative environment."]***.

5.3C [WILSON_TENG_Player] VS [T4TPlayer]

The reason "WILSON_TENG_Player" is able to outsmart "T4TPlayer" is perhaps due to ***[Rule 0 : "Be trustworthy and unpredictable at the same time."]***. While "T4TPlayer" is trustworthy and able to punish and incentivize other players well, it is too predictable. This causes opponents to exploit T4T easily.

5.3D [WILSON_TENG_Player] VS [WinStayLoseShift]

In theory, “WinStayLoseShift” seems to be a good strategy. However, I argue that “WinStayLoseShift” is too myopic as it only looks at whether it has won in the previous round and overlooks the entire history. “WILSON_TENG_Player”, on the other hand, sums up the score of its opponents and itself and compares them in ***[Rule 3 : “If I am losing, turn it into a lose-lose situation.”]*** to determine it’s next action.

5.3E [WILSON_TENG_Player] VS [NicePlayer]

“WILSON_TENG_Player” cooperates well with “NicePlayer” because of ***[Rule 2 : “Cooperate in a cooperative environment.”]***. However, the reason “WILSON_TENG_Player” is able to outperform “NicePlayer” is because “WILSON_TENG_Player” is less vulnerable to exploitation with ***[Rule 1 : “Protect myself.”]***.

5.3F [WILSON_TENG_Player] VS [FreakyPlayer]

FreakyPlayer is decided at the start to either be a Nice or Nasty player. Refer to sections 5.3E and 5.3G respectively.

5.3G [WILSON_TENG_Player] VS [NastyPlayer]

“WILSON_TENG_Player” is able to protect against exploitation from “NastyPlayer” because of ***[Rule 1 : “Protect myself.”]***.

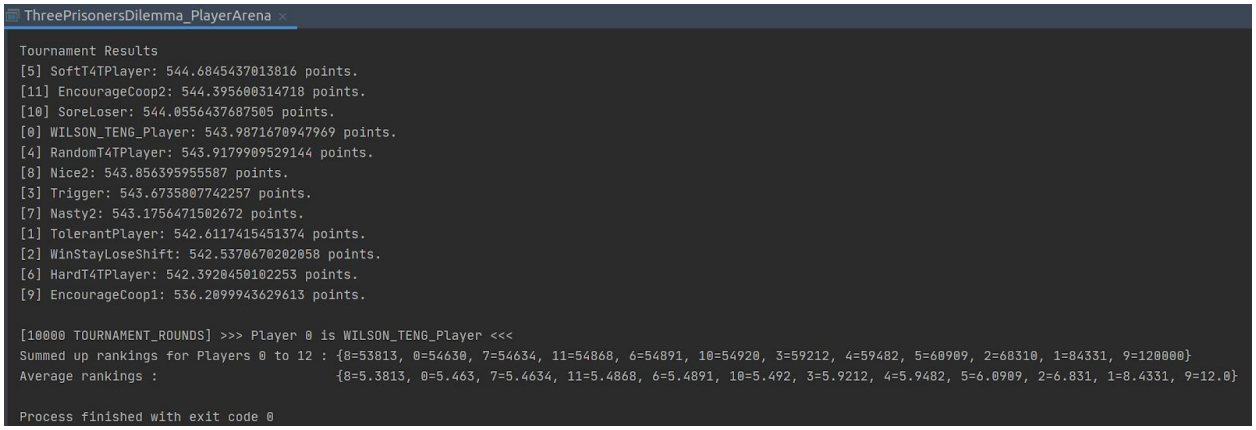
5.3H [WILSON_TENG_Player] VS [AlternatePlayer0]

Similar to T4TPlayer in section 5.3C, the “AlternatePlayer0” is too predictable and easily exploited by other agents.

5.3I [WILSON_TENG_Player] VS [RandomPlayer]

The reason “RandomPlayer” does not perform as well could be due to its unreactivity. It is unable to perform actions that are strategic and hence loses out. It is also at a disadvantage against other agents who have the ability to predict that it is using a random strategy, and this is easily countered by playing defect for the subsequent rounds.

5.4 [Extra Experiment] Performance Comparison with Github Agents (Link)



The rankings are as follows:

1) [P8] Nice2
2) **[P0] WILSON_TENG_Player**
3) [P7] Nasty2
4) [P11] EncourageCoop2
5) [P6] HardT4TPlayer
6) [P10] SoreLoser
7) [P3] Trigger
8) [P4] RandomT4T
9) [P5] SoftT4T
10) [P2] WinStayLoseShift
11) [P1] TolerantPlayer
12) [P9[EncourageCoop1

To simulate a more representative tournament environment, agents from Github as well as the extra agents were added and credited. Unreactive agents, for example, random, freaky, nice and nasty agents, were also removed. Although the rankings may change when the environment above is replicated due to randomness in some of the agents involved, the results show that our agent is consistently performing at around rank 5.5 on average, similar to the other agents.

6. Conclusion

To conclude, our agent outperforms other popular and common strategies due to the synergy between its 4 rules which allows it to be trustworthy enough for other agents to cooperate with while being unpredictable enough to win marginally. But if the strategies fail, our agent falls back to the primitive metric, its score in order to make its decision. This helps our agent to deal with unforeseen situations while also ensuring that its opponents do not get too far ahead because of exploiting our agent.

7. Code Appendix:

Code found in this section can also be found in this GitHub Repository.

```
/* Modified main to run tournament rounds */
public static void main (String[] args) {
    int TOURNAMENT_ROUNDS = 10000; int NUM_PLAYERS = 10;
    boolean PRINT_TOP_3 = false;
    boolean VERBOSE = false; // set verbose = false if you get too much text output
    int val;

    ThreePrisonersDilemma_Template instance = new ThreePrisonersDilemma_Template();
    LinkedHashMap<Integer, Integer> hashMap = new LinkedHashMap<>(); // To store player's cumulative rankings.
    for (int player = 0; player < NUM_PLAYERS; player++)
        hashMap.put(player, v: 0); // initialize player's cumulative ranking to 0.
    for (int i = 0; i < TOURNAMENT_ROUNDS; i++) {
        int[] top_players = instance.runTournament(NUM_PLAYERS, VERBOSE);
        if (PRINT_TOP_3) for (int tp = 0; tp < 3; tp++) {
            System.out.println(top_players[tp]);
        }
        for (int p = 0; p < top_players.length; p++) {
            int tp = top_players[p];
            val = hashMap.get(tp);
            hashMap.put(tp, v: val + p + 1);
        }
    }
    // Sort players by cumulative ranking
    hashMap = (LinkedHashMap<Integer, Integer>) sortByValue(hashMap);
    float float_tournament_rounds = (float) TOURNAMENT_ROUNDS;
    float float_val;
    LinkedHashMap<Integer, Float> newHashMap = new LinkedHashMap<>();
    // Get average ranking
    for (int p=0; p<NUM_PLAYERS; p++) {
        val = hashMap.get(p);
        float_val = (float) val;
        newHashMap.put(p, v: float_val/float_tournament_rounds);
    }
    hashMap = (LinkedHashMap<Integer, Integer>) sortByValue(hashMap);
    newHashMap = (LinkedHashMap<Integer, Float>) sortByValue(newHashMap);

    System.out.print("[" + TOURNAMENT_ROUNDS + " TOURNAMENT_ROUNDS]");
    System.out.println(" >>> Player 0 is WILSON_TENG_Player <<<");
    System.out.println("Summed up rankings for Players 0 to " + NUM_PLAYERS + " : " + hashMap);
    System.out.println("Average rankings : \t\t\t\t\t\t " + newHashMap);
}
```

Section 2.1 - Modified main

```
/* Return same action if prev_round score was >=6. Else, return opp action. */
class WinStayLoseShift extends Player {
    int selectAction(int n, int[] myHistory, int[] oppHistory1, int[] oppHistory2) {
        if (n==0) return 0;
        int r = n - 1;
        int myLA = myHistory[r]; int oppLA1 = oppHistory1[r]; int oppLA2 = oppHistory2[r];

        if (payoff[myLA][oppLA1][oppLA2]>=6) return myLA;
        return oppAction(myLA);
    }

    private int oppAction(int action) {
        if (action==1) return 0;
        return 1;
    }
}
/* If both opponents defected in the last round, defect for subsequent rounds regardless. */
class Trigger extends Player {
    boolean triggered = false;
    int selectAction(int n, int[] myHistory, int[] oppHistory1, int[] oppHistory2) {
        if (n==0) return 0;
        if (oppHistory1[n-1] + oppHistory2[n-1] == 2) triggered = true;
        if (triggered) return 1;
        return 0;
    }
}
/* Start with 0. Return the opposite of the previous action for subsequent rounds. */
class AlternatePlayer0 extends Player {
    int selectAction(int n, int[] myHistory, int[] oppHistory1, int[] oppHistory2) {
        if (n==0) return 0;
        return oppAction(myHistory[n-1]);
    }
    private int oppAction(int action) {
        if (action == 1) return 0;
        return 1;
    }
}
```

Section 2.2 - Added agents

```java
class WILSON_TENG_Player extends Player {
    final String NAME = "[  ] WILSON_THURMAN_TENG";
    final String MATRIC_NO = "[  ] U1820540H";

    int r;
    int[] myHist, opp1Hist, opp2Hist;
    int myScore=0, opp1Score=0, opp2Score=0;
    int opponent1Coop = 0; int opponent2Coop = 0;

    final double LENIENT_THRESHOLD = 0.705; // Used for Law [#1]
    final double STRICT_THRESHOLD = 0.750; // Used for Law [#2]

    int selectAction(int n, int[] myHistory, int[] oppHistory1, int[] oppHistory2) {
        /**
         LAWS:
         [#0] Unless I am losing, be trustworthy and unpredictable at the same time.
         [#1] Protect myself.
         [#2] Cooperate in a cooperative environment.
         [#3] If I am losing, turn it into a lose-lose situation.
         */

        // Assume environment is cooperative. Always cooperate in first round!
        if (n==0) return 0;

        // Updating class variables for use in methods.
        this.r = n - 1; // previous round index
        this.myHist = myHistory;
        this.opp1Hist = oppHistory1;
        this.opp2Hist = oppHistory2;

        // Updating Last Actions (LA) for all players.
        int myLA = myHistory[r];
        int opp1LA = oppHistory1[r];
        int opp2LA = oppHistory2[r];

        // Updating Scores for all players
        this.myScore += payoff[myLA][opp1LA][opp2LA];
        this.opp1Score += payoff[opp1LA][opp2LA][myLA];
        this.opp2Score += payoff[opp2LA][opp1LA][myLA];

        // Update opponent's cooperate record.
        if (n>0) {
            opponent1Coop += oppAction(opp1Hist[r]);
            opponent2Coop += oppAction(opp2Hist[r]);
        }
        // Calculate opponent's cooperate probability.
        double opponent1Coop_prob = opponent1Coop / opp1Hist.length;
        double opponent2Coop_prob = opponent2Coop / opp2Hist.length;
```

Section 4.3 - Agent Implementation (Initializing variables)

```java
/** [PROTECT MYSELF]: -> Law [#1]
 When it is nearing the end of the tournament at 100 rounds, if both players are known to be relatively nasty
 (cooperate less than 75% of the time). Defect to protect myself.
 */
if ((n>100) && (opponent1Coop_prob<STRICT_THRESHOLD && opponent2Coop_prob<STRICT_THRESHOLD)) {
    // Law [#0] Added
    return actionWithNoise( intendedA
}

/** [REWARD COOPERATION]: -> Law [#2]
 At any point in time before we are able to accurately decide if opponents are nasty or not. We set a lenient
 threshold (0.705) to gauge if opponents are cooperative. Additionally, we check if both opponent's last action
 was to cooperate. If yes, we will cooperate too.
 */
if ((opp1LA+opp2LA ==0)&&(opponent1Coop_prob>LENIENT_THRESHOLD && opponent2Coop_prob>LENIENT_THRESHOLD)) {
    // Law [#0] Added
    return actionWithNoise( intendedAction: 0,  percent_chance_for_intended_action: 99);
}
else
/** [I WILL NOT LOSE] -> Law [#3]
 However, if opponent is not cooperative, we will check if we have the highest score.
 If we have the highest score, we are appeased and will cooperate. Else, we will defect.
 */
    return SoreLoser();
```

com.cz4046.ThreePrisonersDilemma_Template.WILSON_TENG_Player
final double STRICT_THRESHOLD = 0.750

Section 4.3 - Agent Implementation (Rules 1 to 3)

```java
/**
 * Law [#0]: This utility method introduces noise to an agent's action, allowing it to be unpredictable.
 * @param intendedAction The agent's intended action.
 * @param percent_chance_for_intended_action The percentage chance the agent will perform it's intended action.
 * @return The agent's final action.
 */
private int actionWithNoise(int intendedAction, int percent_chance_for_intended_action) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>() {{
        put(intendedAction, percent_chance_for_intended_action);
        put(oppAction(intendedAction), 1-percent_chance_for_intended_action);
    }};
    LinkedList<Integer> list = new LinkedList<>();
    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
        for (int i = 0; i < entry.getValue(); i++) {
            list.add(entry.getKey());
        }
    }
    Collections.shuffle(list);
    return list.pop();
}

/** Law [#3]:
 * Cooperates if agent currently has the highest score, else defect.
 * @return
 */
private int SoreLoser() {
    if (iAmWinner()) return 0;
    return 1;
}

/* Function to check if agent is loser or not. Agent is a winner if it has the highest score. */
private boolean iAmWinner() {
    if (myScore>=opp1Score && myScore>=opp2Score) {
        return true;
    }
    return false;
}
/* Utility method to obtain opposite action. */
private int oppAction(int action) {
    if (action == 1) return 0;
    return 1;
}
```

Section 4.3 - Agent Implementation (Methods Used)