

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
SC4022 - INTELLIGENT AGENTS

Assignment 2 - Repeated Prisoners Dilemma
Report

Cholakov Kristiyan Kamenov, U2123543B

Table of Contents

1. Introduction.....	2
1.1 The Prisoners Dilemma.....	2
1.2 Repeated Version of the Dilemma.....	2
1.3 Assignment's Case.....	2
2. Review of Existing Approaches.....	3
2.1 Existing Approaches.....	3
2.1.1 Tit-for-Tat.....	3
2.1.2 Tit-for-Two-Tats.....	4
2.1.3 Generous Tit-for-Tat.....	4
2.1.4 Win-Stay, Lose-Shift.....	4
2.1.5 Grim Trigger.....	4
2.2 Existing Approaches Evaluation.....	5
3. Ideation and Influences.....	5
3.1 Memory & History Analysis.....	5
3.2 Forgiveness.....	6
3.3 Be Nice & Always Cooperate the First Round.....	6
3.4 Endgame Strategy.....	6
3.5 Prediction & Intuition.....	6
3.6 Retaliation & Punishment.....	6
3.7 Maximize the Cooperation.....	7
4. Agent Design.....	7
4.1 Concept.....	7
4.2 Implementation.....	7
4.2.1 Agent's Structure.....	7
4.2.2 Cooperative Start Implementation.....	9
4.2.3 History Analysis Implementation.....	9
4.2.4 Endgame Strategy Implementation.....	10
4.2.5 Grim Trigger Implementation.....	11
4.2.6 Punishing Intentional Defecting.....	11
5. Agent Evaluation.....	11
5.1 Given Agents.....	12
5.2 GitHub Agents.....	13
6. Conclusion.....	15
References.....	16

1. Introduction

1.1 The Prisoners Dilemma

The Prisoner's Dilemma is one of the most popular and widely studied scenarios in game theory. It helps with understanding significant rules applied in agent design, the most notable being the fact that individuals may not work together even when it is best for them. Originated as a single-iteration situation of two separated criminals separated from each other and offered the same deal: if one confesses and the other doesn't, the one who confessed is free while the other one gets a long sentence. If both criminals stay quiet, they both get a short sentence. But if they both confess, they will get a medium sentence. The game shows that it is hard to trust the other parties if there is a situation which clearly benefits only one of the players. Later it was discovered that the scenario can be applied in many real life aspects, like in business or managing resources. Thus, this dilemma is still relevant and studied today.

1.2 Repeated Version of the Dilemma

In the context of the single-iteration game, people have reached the verdict that the dominant strategy for a player is to defect rather than cooperate. The reasoning behind this approach is based on the Nash Equilibrium - the player's best response is to defect because in this case the player will not regret its decision no matter the action of the other prisoner. The repeated version of the Prisoners Dilemma is based on the original idea but the game is extended to multiple rounds. This version introduces the possibility of developing trust or competitiveness among the players, thus making the game even more realistic and applicable in real life because it covers long-term interactions. The Repeated Prisoners Dilemma provides a much more complex environment where coming up with a strategy is difficult due to the dynamic and unknown strategy of the other players.

1.3 Assignment's Case

In this assignment, the tournaments consist of the matches between all possible triples of players and every match is taking a random number of rounds between 90 and 110. Table 1 represents the payoff matrix given by the assignment, the columns represent the action taken by the corresponding agent (0 - cooperate, 1 - defect).

The knowledge our agent will get from the environment is:

- **n** - the number of rounds elapsed so far
- **myHistory** - the history of our agent's actions
- **oppHistory1 / oppHistory2** - the history of our opponents' actions

Table 1. Given Payoff Matrix from the Assignment

Us	Opponent 1	Opponent 2	Payoff
0	0	0	6
0	0	1	3
0	1	0	3
0	1	1	0
1	0	0	8
1	0	1	5
1	1	0	5
1	1	1	2

2. Review of Existing Approaches

In order to develop a good strategy that performs well against other players' ones, we need to analyze the existing approaches for the 3-Player Repeated Prisoners Dilemma as our opponents' algorithms are likely based on them.

2.1 Existing Approaches

We will analyze some of the most significant basic strategies because most of the complex models are based on several of them with added dynamics on top.

2.1.1 Tit-for-Tat

Tit-for-Tat is one of the most famous strategies in the Repeated Prisoners Dilemma. The main idea behind this approach is to cooperate in the first round and mimic the opponent's previous behavior in the next rounds. As observed in [1], despite the strategy's simplicity it often performs well in tournaments with various approaches. The strategy is considered nice as it starts with a cooperative decision, however, it is also provokable as it will immediately counter-attack to opponents' choice to defect. Tit-for-Tat is also forgiving as it will choose to cooperate if the opponent starts to behave again. Although, Tit-for-Tat has one very significant limitation - the lack of forgiveness. The strategy can be quite vulnerable to noise such as mistakes or miscommunication between the players.

2.1.2 Tit-for-Two-Tats

Tit-for-Two-Tats is a more forgiving variation of Tit-for-Tat, firstly introduced in [2]. Based on this new strategy, the player defects only if the opponent has behaved against the player twice in a row. This variation of Tit-for-Tat aims to reduce the negative impact of mistakes or misunderstanding between the players, it tries to avoid the scenario of players starting to play against each other every single round.

2.1.3 Generous Tit-for-Tat

As mentioned in [3], Generous Tit-for-Tat improves the Tit-for-Tat and Tit-for-Two-Tats strategies by introducing flexible forgiveness based on probability. This design is more rigid to exploitative strategies as it relies on its adjustable configuration. While it offers forgiveness and resilience to errors, the Generous Tit-for-Tat also maintains a likelihood of revenge.

2.1.4 Win-Stay, Lose-Shift

Introduced in [4], the Win-Stay, Lose-Shift strategy as mentioned in the name consists of two parts:

- **Win-Stay:** If the outcome of a move is successful, the player repeats the same action in the next round.
- **Lose-Shift:** If the outcome of a move is unsuccessful, the player changes their action in the next round.

Successful round is defined as one in which the player receives a high payoff, while an unsuccessful round is one when the player receives a low payoff. Typically, the Win-Stay, Lose-Shift strategy is more effective in unstable environments with frequent attacks, errors and misunderstanding between the agents. This is due to the fact that the dynamics of the strategy are directly connected to the outcome being favorable or not, rather than the specific actions of the opponent.

2.1.5 Grim Trigger

As mentioned in [5], the Grim Trigger strategy is based on the initial Nash Equilibrium for the single-iteration Prisoners Dilemma. The strategy is designed to start by cooperating, but any opponent error or attack against the player will trigger the defecting part of the strategy and the agent will continue to defect forever (until the end of the game). This agent design aims to penalize any attempt of the other players to trick the agent. The main limitation of the strategy is defecting forever as this may trigger the

same response from the other players too. Thus, this repeating Nash Equilibrium of all players defecting will lead to lower payoffs.

2.2 Existing Approaches Evaluation

As all approaches presented in Section 2.1 are nice strategies, we will test them against a random agent whose decision is random every time, and a nasty player who always defects. The agent designs will be tested on the provided tournament environment.

Table 2. Basic Existing Approaches Evaluation

Agent	Score
WSLSPlayer	174.79268
GTFTPlayer	172.12141
GTPlayer	171.9437
TFTPlayer	171.69212
TF2TPlayer	170.70714
NastyPlayer	133.23854
RandomPlayer	126.49852

As seen in Table 2, the Win-Stay, Lose-Shift performs the best with Generous Tit-for-Tat and Grim Trigger taking the second and third places. Based on these results, we will experiment with these strategies for our agent's design.

3. Ideation and Influences

In this section, good practices, rules and strategies, apart from the basic ones in Section 2, will be covered. They are specific to the repeated version of the Prisoners Dilemma. Now, we will discuss some of these strategies that will help our agent in the Repeated 3-Player Prisoners Dilemma game.

3.1 Memory & History Analysis

As mentioned in the assignment instruction, the agents will be provided with their own actions history and the ones of their opponents. The information about the previous actions of our opponents can be used to calculate their cooperation rate and detect patterns in their behaviors.

3.2 Forgiveness

The agent should be able to forgive previous attacks from its opponents. Some of these defects may be caused by errors or miscommunication between the agents' strategies. Forgiveness helps the agents to continue without being stuck in a loop where each agent defects. Defecting may be a Nash Equilibrium in the single-iteration of the game but for repeated games with around 100 rounds, it lowers the possible score significantly.

3.3 Be Nice & Always Cooperate the First Round

Despite that there are some strategies like Tit-for-Two-Tats which are tolerant to defecting in the first round, most dynamic and history-based approaches calculate the cooperation rate. Thus, acting by defecting in the first round will instantly trigger 0% cooperation rate in your opponents and will likely make them counter-attack you in the next rounds. Acting against your opponents, especially without a reason, will likely cause a chain reaction of defecting in the following rounds, and in many cases the only way to prevent the bad behavior from continuing will be cooperating while your opponents choose to defect, resulting in 0 points for the agent.

3.4 Endgame Strategy

Typically, the final round in the Repeated Prisoners Dilemma is known to be suitable for defecting action because there will not be any iteration after it, so the opponents cannot react against you.

3.5 Prediction & Intuition

The agents should be able to predict and prevent the intuitive actions against it. For instance, the endgame strategy is such a scenario. As defecting is the dominant strategy for the final round, most of the opponents will start defecting near the end of the match, hoping to stay unnoticed and get the maximum reward. The agent must intuitively predict that and try to defect first. Also, the agent will perform better if it is able to detect some predefined static strategies.

3.6 Retaliation & Punishment

The agent should not allow its opponents to continuously defect, the agent must punish such activity to prevent it from happening again. The agent design for retaliation may be inspired by Tit-for-Tat and tuned with different sensitivity to prevent continuous defecting from all sides.

3.7 Maximize the Cooperation

In the context of the single-iteration Prisoners Dilemma the Nash Equilibrium is achieved when both players defect. However, continuous defecting strategy is far from optimal in the Repeated Prisoners Dilemma. Even though the strategy does not allow the opponent to benefit from the agent, always defecting leads to extremely low results. Low results may be effective if we play only a single match with two other opponents. However, if an agent plays “nasty” during the tournament with many triples combinations, the agent will get a low average score from its matches while others may cooperate more in other matches, resulting in higher score.

4. Agent Design

4.1 Concept

Based on the research in the previous sections and many experiments with various strategy combinations, the provided solution is hybrid agent based on the following strategies:

- Always cooperate in the start
- History analysis
 - Monitoring the previous actions
 - Calculating the cooperation rate of the opponents
 - Monitoring the score of the agents in the match
- Endgame dynamic strategy
 - Sensing for static endgame strategy
 - Static endgame strategy for the extreme upper range of round
- Grim trigger
- Punish intentional defecting from opponents

Only the following strategies were considered for the agent’s design because all of them maintain the good behavior between the agents. No exploitation techniques were included in the structure of the agent as it may provoke negative actions from the opponents and decrease the average score of our agent in the whole tournament. The listed rules and practices above involve defecting only when needed, they even tolerate small errors from the other contestants and do not directly respond to them. Retaliation is applied only if intentional or repetitive attacks are detected against the agent.

4.2 Implementation

4.2.1 Agent’s Structure

Figure 1 shows the complete implementation of the provided solution.


```

class Cholakov_Kristiyan_Player extends Player { 1 usage new *
    // Previous round index
    int i; 12 usages

    // History arrays
    int[] myHist, opp1Hist, opp2Hist; 6 usages

    // Total scores
    int myScore = 0, opp1Score = 0, opp2Score = 0; 3 usages

    // Cooperation counts
    int opponent1Coop = 0; 2 usages
    int opponent2Coop = 0; 2 usages

    // Trigger flag for if defection strategy is detected
    boolean triggerDefect = false; 3 usages

    // Payoff matrix
    static int[][][] payoff = { 3 usages
        {{6, 3}, {3, 0}},
        {{8, 5}, {5, 2}}
    };

    int selectAction(int n, int[] myHistory, int[] oppHistory1, int[] oppHistory2) { 3 usages new *
        if (n == 0) return 0; // Always cooperate in the first round

        // Update class variables
        updateScoresAndHistories(n, myHistory, oppHistory1, oppHistory2);

        // Calculate cooperation probabilities
        double opp1CoopProb = (double) opponent1Coop / n;
        double opp2CoopProb = (double) opponent2Coop / n;

        // Endgame strategy
        if (n > 90) {
            if (n > 107) return 1; // Defect in the last rounds
            // Check for possible defection strategy
            if ((opp1Hist[n - 1] == 1 || opp2Hist[n - 1] == 1) && !(opp1Hist[n - 2] == 1 || opp2Hist[n - 2] == 1 || myHist[n - 2] == 1)) {
                triggerDefect = true; // Trigger unconditional defection
            }
        }

        // Check for possible defection strategy
        if (n > 10) {
            if (opp1CoopProb < 0.1 || opp2CoopProb < 0.1) {
                triggerDefect = true; // Trigger unconditional defection
            }
        }

        // If trigger is active, defect
        if (triggerDefect) {
            return 1;
        }

        // Check cooperation rate and defect if my score is lower than both opponents
        if ((opp1CoopProb < 0.5 || opp2CoopProb < 0.5) && (myScore < opp1Score || myScore < opp2Score)) {
            return 1; // Defect if any opponent is less cooperative
        }

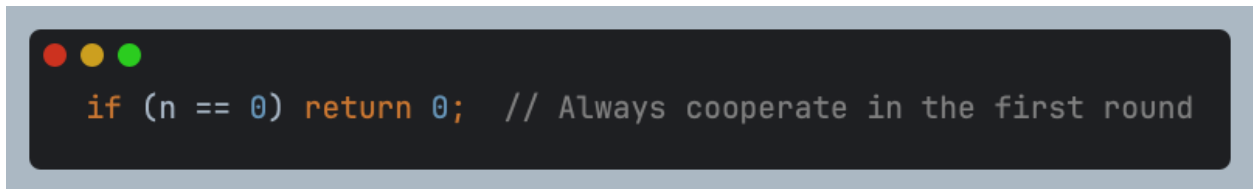
        // Respond based on the combined last action of opponents
        if (opp1Hist[n - 1] == 0 && opp2Hist[n - 1] == 0 && myHist[n - 1] == 0) {
            return 0;
        } else {
            return 1;
        }
    }
}

```

Figure 1. Agent's Structure

4.2.2 Cooperative Start Implementation

The implementation of the cooperative start is simply checking if the round's index is 0 (meaning this is the first round) and returning cooperate as decision.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains a single line of C++ code: `if (n == 0) return 0; // Always cooperate in the first round`.

```
if (n == 0) return 0; // Always cooperate in the first round
```

Figure 2. Cooperative Start

4.2.3 History Analysis Implementation


A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two C++ functions. The first function, `updateScores()`, updates the agent's score and the scores of two opponents based on a payoff matrix. The second function, `updateCooperationCounts()`, updates the cooperation counts for the two opponents based on their history.

```
// Update scores based on the payoff matrix
void updateScores() { 1 usage new *
    myScore += payoff[myHist[i]][opp1Hist[i]][opp2Hist[i]];
    opp1Score += payoff[opp1Hist[i]][myHist[i]][opp2Hist[i]];
    opp2Score += payoff[opp2Hist[i]][myHist[i]][opp1Hist[i]];
}

// Update cooperation counts
void updateCooperationCounts() { 1 usage new *
    opponent1Coop += opp1Hist[i] == 0 ? 1 : 0;
    opponent2Coop += opp2Hist[i] == 0 ? 1 : 0;
}
```


Figure 3. Score and Cooperation Count

As seen in Figure 3, the proposed agent monitors the score and the cooperation count for each opponent. Using the cooperation count, the cooperation rate is calculated each round as seen in Figure 4. These statistics are later used in the checks (rules) seen in Figure 5 and Figure 6. In Figure 5 the opponents' cooperation rates are checked after round 10 to examine if any of the opponents uses a mostly defective strategy. The check in Figure 6 is used to counter bad behavior from the opponents if the agent's score is less than the scores of the opponents.




```
// Calculate cooperation probabilities
double opp1CoopProb = (double) opponent1Coop / n;
double opp2CoopProb = (double) opponent2Coop / n;
```

Figure 4. Cooperation Rate Calculation



```
// Check for possible defection strategy
if (n > 10) {
    if (opp1CoopProb < 0.1 || opp2CoopProb < 0.1) {
        triggerDefect = true; // Trigger unconditional defection
    }
}
```

Figure 5. Extremely Low Cooperation Rate Check

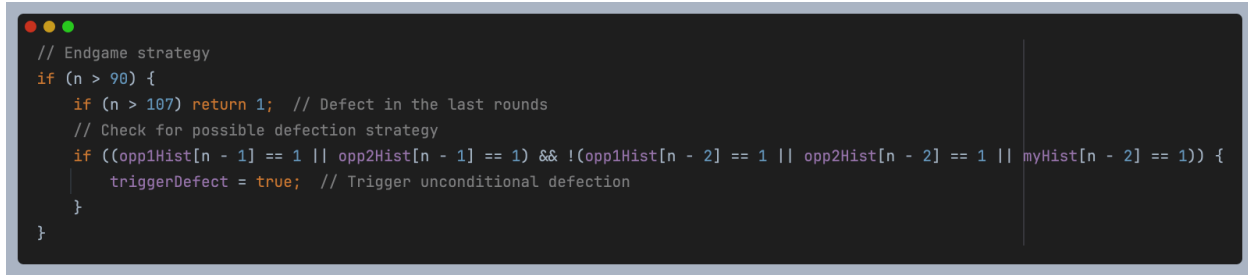


```
// Check cooperation rate and defect if my score is lower than both opponents
if ((opp1CoopProb < 0.5 || opp2CoopProb < 0.5) && (myScore < opp1Score || myScore < opp2Score)) {
    return 1; // Defect if any opponent is less cooperative
}
```

Figure 6. Combined Score and Cooperation Check

4.2.4 Endgame Strategy Implementation

The endgame strategy is implemented as seen in Figure 7. The proposed endgame rule consists of two parts: dynamic detection condition and purely static one. The purely static one simply forces the agent to defect if the round index is very close to the possible upper limit. Otherwise, the agent is trying to sense for a previous unintentional defect from any of the opponents. The examination of whether the defecting in the previous round was intentional is done by checking if any agent defected 2 rounds ago (the round before the previous round).




```
// Endgame strategy
if (n > 90) {
    if (n > 107) return 1; // Defect in the last rounds
    // Check for possible defection strategy
    if ((opp1Hist[n - 1] == 1 || opp2Hist[n - 1] == 1) && !(opp1Hist[n - 2] == 1 || opp2Hist[n - 2] == 1 || myHist[n - 2] == 1)) {
        triggerDefect = true; // Trigger unconditional defection
    }
}
```

Figure 7. Combined Score and Cooperation Check

4.2.5 Grim Trigger Implementation

Grim trigger implementations can be seen in both Figure 4 and Figure 7. The implemented trigger in our agent is activated if continuous bad behavior is detected in the start of the match, or if unprovoked defecting occurs during the last rounds of the match. If any of these conditions are active, they set the trigger variable which will always make the “if” statement in Figure 8 true, resulting in defecting forever.



```
// If trigger is active, defect
if (triggerDefect) {
    return 1;
}
```

Figure 8. Trigger Check

4.2.6 Punishing Intentional Defecting

Lastly, at the end of our agent implementation, we have a rule based on the Tit-for-Tat that tells the agent to defect if any agent in the previous round responded with a defect. We are including our agent too as most competitors will respond to a defect from our side with defects from their sides.

5. Agent Evaluation

In this part, we will evaluate our agent’s performance. We will compare our agent’s performance by including it in a tournament with the provided predefined agents in the environment given by the assignment. The provided code was modified slightly to support analysis for several tournaments in a row. The evaluation will be conducted by calculating the average position and score of the agents in 1, 100 and 1000 tournaments.

5.1 Given Agents

In this subsection, we will compare our agent with the provided ones from the assignment:

- NicePlayer
- NastyPlayer
- RandomPlayer
- TolerantPlayer
- FreakyPlayer
- T4TPlayer

Average Score for Different Number of Tournaments

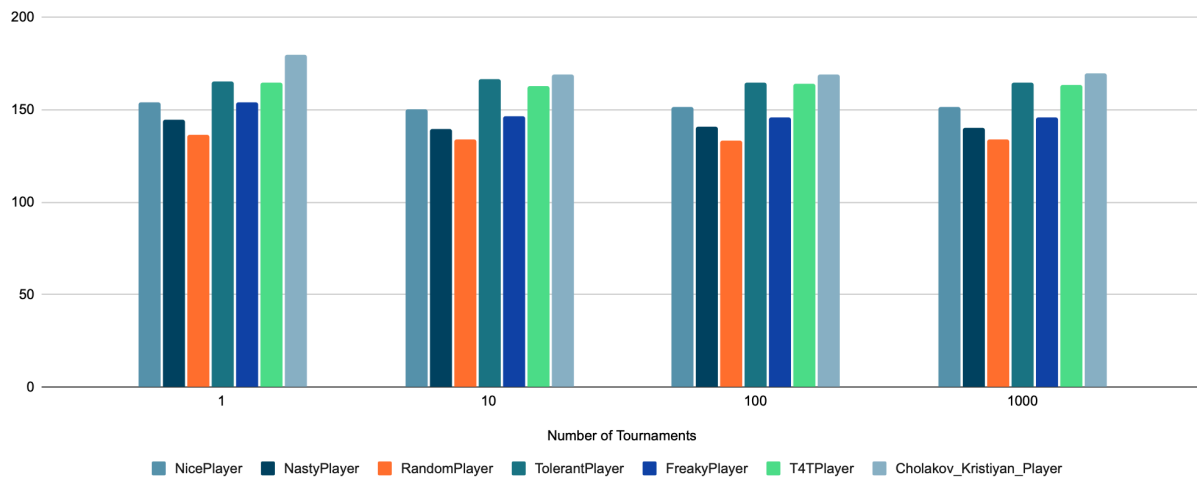


Figure 9. Average Score for Different Number of Tournaments

Average Position for Different Number of Tournaments

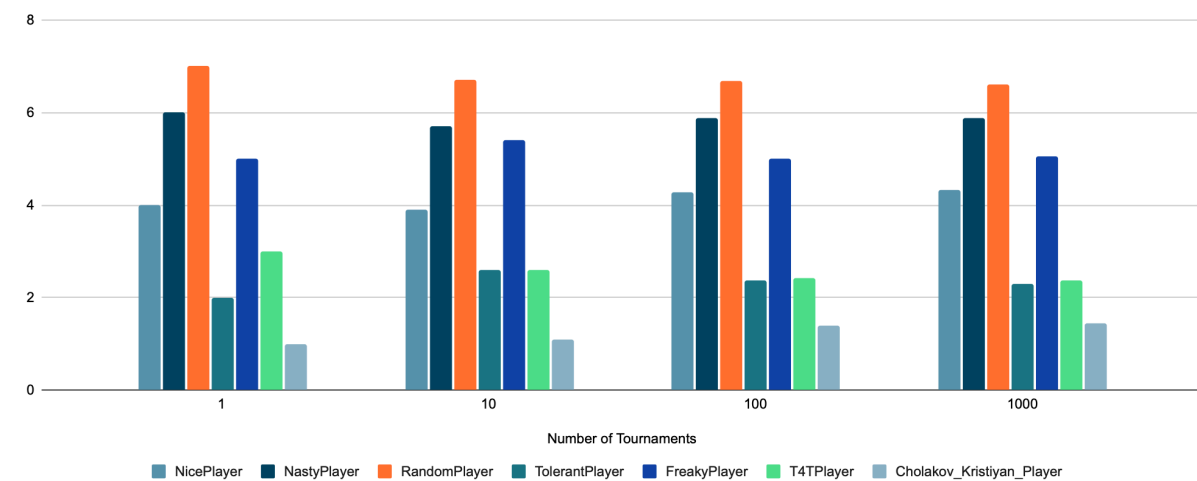


Figure 10. Average Position for Different Number of Tournaments

Figure 9 shows the average score of the agents for various series of tournaments. As it can be seen in the figure, our agent performs the best out of all agents. The agent achieves the highest score no matter the number of tournaments the calculations were done for.

In Figure 10, we are presenting the average ranking position of the agents when sorted by their scores in descending order, meaning the first position is the best (the agent with highest score). Again, we can see that our agent achieves an average position very close to the position 1. The difference with the other approaches is much more significant in this graph.

5.2 GitHub Agents

Despite that the proposed agent achieves good performance levels when compared to the given agents, in reality, the agent will be included with many other competitors' strategies which will be more complex than the ones from Section 5.1. In order to compare our solution with other existing ones for the same task, a search in google was performed, and the following agent architectures from previous years of the assignment were found:

- Ngo_Jason_Player:

https://github.com/NgoJunHaoJason/CZ4046/blob/master/assignment_2/Ngo_Jason_Player.java

- WILSON_TENG_Player:

https://github.com/wilsonseng97/Intelligent-Agents-2-ThreePrisonersDilemma/blob/master/src/com/cz4046/WILSON_TENG_Player.java

- Huang_KyleJunyuan_Player:

https://github.com/HJunyuan/cz4046-intelligent-agents/blob/master/assignment-2/Huang_KyleJunyuan_Player.java

- Naing_Htet_Player:

https://github.com/Javelin1991/CZ4046_Intelligent_Agents/blob/master/CZ4046_Assignment_2/ThreePrisonersDilemma.java

As we can see in Figure 11, the GitHub agents are much more competitive than the provided ones. Although our agent achieves the best results in Figure 11, this observation is not clearly seen. Thus, the average position is plotted in Figure 12, there it is more visible that our agent outperforms the other candidates on average. Now, we have a better understanding of how the agent may perform in the competition.

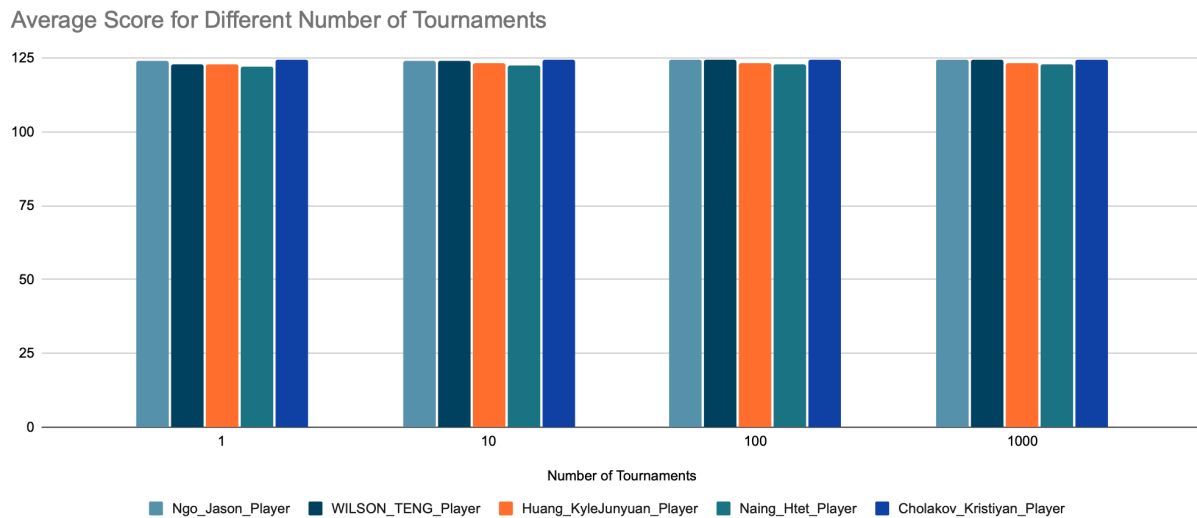


Figure 11. Average Score for Different Number of Tournaments - GitHub

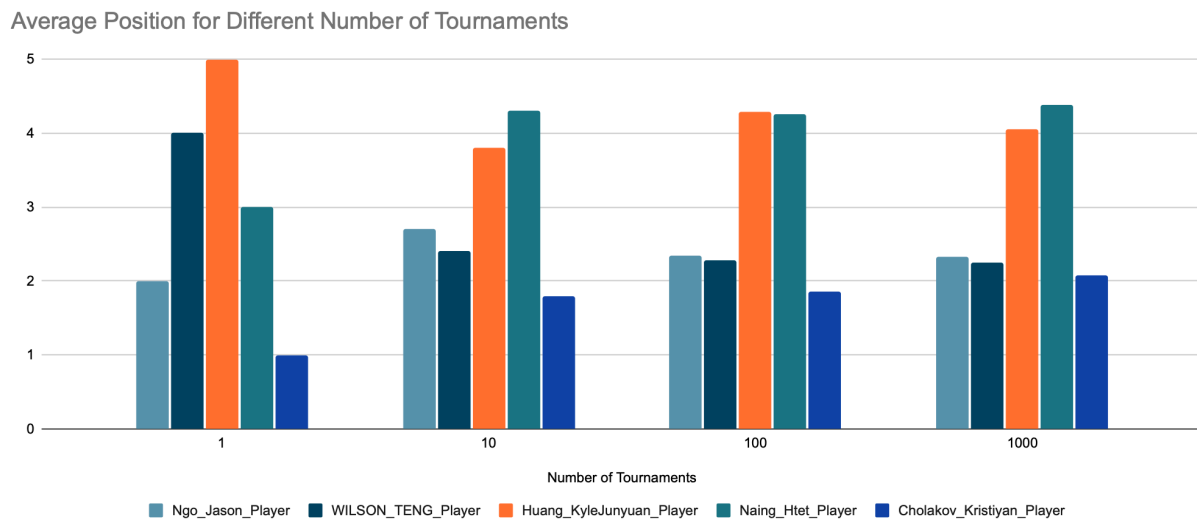


Figure 12. Average Position for Different Number of Tournaments - GitHub

Figure 11 and Figure 12 show how significant the type of the competition is when comparing agents' performance. Moreover, during the experiments we observed that it is not enough for the agent to outperform some complex opponents. For example, if we add additional competitors which use strategies that directly try to minimize our agent's performance, the performance of our agent will drop, and because only our agent is targeted in this imaginary situation, it will drop in the leaderboards.

6. Conclusion

In conclusion, all observations from the previous section show that our agent is competitive and achieves good results among other existing approaches. We believe this is due to its complex structure and the combination of different strategies. Furthermore, the proposed agent behaves nicely, fairly (does not try to exploit or cheat its opponents), and is tolerant to little errors or miscommunications between the other agents. On the other hand, this does not compromise its ability to punish undeserved betrayals. To summarize, the proposed agent has good manners (ready to cooperate) but expects the same nice behavior from the others, else it is prepared to counter-attack.

Designing an agent for the Repeated 3-Player Prisoners Dilemma is a complex task that does not have a single direct solution. In fact, the solution is not even limited by the rules or the given data by the game (payoff matrix), the biggest unknown in the equation here is the competition, an aspect we are not given any data about. This uncertainty disrupts the strategy the most.

References

- [1] R. Axelrod, *The Evolution of Cooperation*. 1984.
- [2] R. Axelrod, "More Effective Choice in the Prisoner's Dilemma," *The Journal of Conflict Resolution*, vol. 24, no. 3, pp. 379–403, 1980.
- [3] N. Player, "The Morality and Practicality of Tit for Tat", Accessed: Apr. 22, 2024. [Online]. Available: <https://pressbooks.lib.vt.edu/ppper/chapter/the-morality-and-practicality-of-tit-for-tat/>
- [4] M. A. Nowak and K. Sigmund, "The Alternating Prisoner's Dilemma," *Journal of Theoretical Biology*, vol. 168, no. 2, pp. 219–226, May 1994, doi: [10.1006/jtbi.1994.1101](https://doi.org/10.1006/jtbi.1994.1101).
- [5] J. W. Friedman, "A Non-cooperative Equilibrium for Supergames," *The Review of Economic Studies*, vol. 38, no. 1, pp. 1–12, 1971, doi: [10.2307/2296617](https://doi.org/10.2307/2296617).