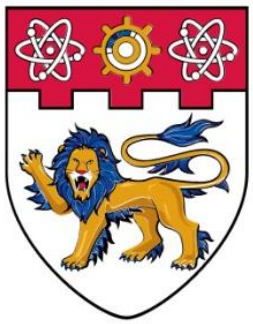

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4046 INTELLIGENT AGENTS

ASSIGNMENT 2 REPORT

Name: **Bhatia Ritik**

Matriculation Number: **U1822529C**

Tutorial Group: **CS4**

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

ACADEMIC YEAR 2021/22

Contents

Problem

1.1. Overview

The Prisoner's Dilemma is a **paradox** in decision analysis in which two individuals acting in their own self-interests does not result in the optimal outcome. However, it can also be described as a **thought experiment** that depicts why individuals may choose to defect (not cooperate) even if it appears beneficial for both parties involved in the process, to do so.

The reason for it being a dilemma is that individual rational action is to defect which guarantees a higher worst payoff than by cooperating. However, intuition suggest that best outcome is that both agents **should cooperate** and get an even higher payoff. This is what makes the problem interesting, and for this assignment, we are given a slightly modified version of the problem.

In this game, we are given the *Three Player Repeated Prisoner's Dilemma* problem. This means that we have three players instead of two, and multiple rounds will be played repeatedly, which makes the **lack of cooperation disappear** and instead favors agents with smart strategies. More precisely, in our problem, three agents will play a match of **100** rounds each and the ranking is decided by computing the average reward of the game.

The Payoff Matrix for the same is shown below:

Our agent	***Opposing Agent 1***	***Opposing Agent 2***	***Payoff***
Cooperate	Cooperate	Cooperate	**6**
Cooperate	Cooperate	Defect	**3**
Cooperate	Defect	Cooperate	**3**
Cooperate	Defect	Defect	**0**
Defect	Cooperate	Cooperate	**8**
Defect	Cooperate	Defect	**5**
Defect	Defect	Cooperate	**5**
Defect	Defect	Defect	**2**

The reward from each game can be determined from the set of actions chosen by each agent in each round, by using the above Payoff Matrix.

The goal is to devise a strategy that can result in the best ranking possible. Each strategy will be given a **history** of actions of the agent itself as well as the history of actions of the opponents.

1.2. Some observations

A strategy in a repeated game specifies what action the agent should take in each stage of the game, given all the actions taken by all players in the past. There are **infinitely many** possible strategies, but there are a few popular ones that perform well in a general scenario.

For example, **Tit-for-Tat** is a strategy that works very well in the **two-player Prisoner's Dilemma**. The strategy is simple – the agent starts off by cooperation to show goodwill and then repeats whatever the opponent did in the last round. This means cooperation of opponent is rewarded with cooperation while defection is appropriately punished with a consequent defection.
Another strategy is **soft majority** where the agent continues to cooperate as long as the number of times the opponent has cooperated is greater than or equal to the number of defections that it has done.

The above show one of the best results in empirical tests in a two-player Prisoner's Dilemma. However, the main concern for our given problem is whether these (and other well-known strategies) work equally well when three players are involved. Further, the Tit-for-Tat strategy aims to **tie** the game instead of aggressively trying to beat the opponent/s and gain maximum payoff. Similar shortcomings can be found in other well-known strategies and given the nature of evaluation of our strategy (competition with other strategies); it is favorable to devise a **custom strategy** for maximum payoff.

Nevertheless, using these well-known strategies is a very good starting point. In devising my custom strategy, I have built off of these strategies in an attempt to gain best the rankings.

2. Considerations

Some basic assumptions that can be taken for such an environment are:

- Agents are **self-interested**: they have a preference over how the environment is.
- Agents will not necessarily choose the **rational** choice (due to the above assumption) and hence our agent should be **exploitative** when given the opportunity.
- The scenario is **highly competitive**, however, a strictly competitive / zero-sum strategy will not be the best assumption, as per Axelrod's suggestions.
- It is impossible to estimate the strategy of the opponents since there are infinitely many strategies possible.
- Due to multiple rounds, if a certain agent defects, it is still possible to recover cooperation through goodwill.

The above assumptions are extremely important to design the agent that can yield an optimal payoff throughout the tournament. Each design decision has its trade-offs; however, the win is by achieving the best balance.

To design the most optimal agent, I took **Axelrod's rules** into account which can be summarised as follows:

1. *Don't be envious* – Don't play as if the environment is zero sum.
2. *Be nice* – Start by cooperating and reciprocate cooperation
3. *Retaliate appropriately* – Always punish defection immediately, but use *measured* force (don't overdo the punishment as one might risk other's defections which might not be optimal for us)
4. *Don't hold grudges* – Always reciprocate cooperation immediately

From the above suggestions as provided by Axelrod, it is clear that the strategy of my agent would be such that is relatively **fair** – if other agents cooperate, it will attempt to cooperate as well (except certain **edge scenarios** such as the last round of the game where defection can help achieve an even higher payoff) and will punish opponents if they defect. However, retaliation would be **measured** to avoid entering into a **defection loop** and it will not hold grudges as it may again risk entering the defection loop since opponents would be doubtful of my agent's intention to cooperate.

The final consideration is to extend Axelrod's suggestions to our three-player repeated Prisoner's Dilemma such as retaliating harshly only if both opponents defect etc.

3. Strategy Implementation

To find the best strategy, I implemented a few well-known strategies to gauge their performance and to iteratively use them as **building blocks** to get the next best strategy.

A few strategies that I worked with, before finding the selected strategy are explained in the sections below.

3.1. Always Defect (Nasty Player)

Let's have a look at the Payoff Matrix, sorted by Payoff, below:

Our agent	***Opposing Agent 1***	***Opposing Agent 2***	***Payoff***
Defect	Cooperate	Cooperate	**8**
Cooperate	Cooperate	Cooperate	**6**
Defect	Cooperate	Defect	**5**
Defect	Defect	Cooperate	**5**
Cooperate	Cooperate	Defect	**3**
Cooperate	Defect	Cooperate	**3**
Defect	Defect	Defect	**2**
Cooperate	Defect	Defect	**0**

From the above table, it is very intuitive to see the **dominance of defection** to get a high payoff and hence, the first strategy that I tested was when the agent always defected (nasty player).

However, this strategy did not perform well (covered in detail in Section 4.) and instead ended up at the **bottom of the ranking** for the majority of the time. The main reason for this is that if the agent always defects, it is very easy for the opponents to figure that this agent is **extremely selfish**. Hence, they too defect in the subsequent rounds which leads to a much lower payoff of 2 (as can be seen from the table above).

Therefore, always defection is not a viable strategy to secure a good rank in this problem.

3.2. Tolerant Tit-for-Tat

Tit-for-Tat is a very popular strategy that gives great results in the two-player Prisoner's Dilemma (proven in several research papers). It provides a great **balance between cooperation and defection** by appropriately punishing the opponent if it defects.

To adapt to the modified three-player Prisoner's Dilemma, I implemented a **tolerant** version of this strategy, that is, my agent would defect only if the total number of defections of **both agents** is greater than the total number of time both agents cooperated. This is done to **avoid an aggressive strategy** that risks opponents defecting continuously in subsequent rounds.

The code implementation of this strategy is given in Figure 1 below:

```
class TolerantTitForTat extends Player {
    int selectAction(int n, int[] myHistory, int[] oppHistory1, int[] oppHistory2) {
            // at the start of the game, cooperate to show that you are nice
            if (n == 0) {
                return 0;
            }

            // since the current round is the last round, opponents cannot retaliate
            // hence, be selfish and defect for maximum payoff
            else if (n == 109) {
                return 1;
            }

            // if both competitors did the same move in the previous round
            // follow the same to punish / show friendliness
            if (oppHistory1[n - 1] == oppHistory2[n - 1]) {
                return oppHistory1[n - 1];
            }

            // initialize the total number of cooperations and defections of the opponent
            int numOppCoops = 0;
            int numOppDefects = 0;

            // iterate through the histories of both the opponents
            for (int i = 0; i < n; i++) {
                if (oppHistory1[i] == 0)
                    numOppCoops += 1;
                else
                    numOppDefects += 1;

                if (oppHistory2[i] == 0)
                    numOppCoops += 1;
                else
                    numOppDefects += 1;
            }

            // if the opponents as whole made more defections than cooperations, defect
            // else cooperate
            if (numOppDefects > numOppCoops)
                return 1;
            return 0;
    }
}
```

Figure 1. *Code for tolerant Tit-for-Tat*

3.3. History Rectification

Another strategy that I tested was *History Rectification*. In this strategy, unlike *Tit-for-Tat*, the agent looks through its **own history of actions** as well and compares it to that of the opponents'. During comparison, if the agent notices that is has made more defections in the past than both the opponents, then it **cools down** by cooperating in the current round, otherwise it defects. The cool-down period attempts to express goodwill towards opponents to get them to cooperate in the subsequent rounds while gaining greater payoff through defections in other scenarios.

The code implementation of this strategy is shown in Figure 2 below:

```
class HistoryRectification extends Player {
    int selectAction(int n, int[] myHistory, int[] oppHistory1, int[] oppHistory2) {
            // at the start of the game, cooperate to show that you are nice
            if (n == 0) {
                return 0;
            }

            // since the current round is the last round, opponents cannot retaliate
            // hence, be selfish and defect for maximum payoff
            else if (n == 109) {
                return 1;
            }

            // if both competitors did the same move in the previous round
            // follow the same to punish / show friendliness
            if (oppHistory1[n - 1] == oppHistory2[n - 1]) {
                return oppHistory1[n - 1];
            }

            // initialize variables to track number of defections
            int numOpp1Defects = 0;
            int numOpp2Defects = 0;
            int numMyDefects = 0;

            // iterate through all histories to get number of defections of each
            for (int i = 0; i < n; i++) {
                numOpp1Defects += oppHistory1[i];
                numOpp2Defects += oppHistory2[i];
                numMyDefects += myHistory[i];
            }

            // if my agent has been defecting more than both, cooperate to break cycle
            if (numMyDefects > numOpp1Defects && numMyDefects > numOpp2Defects)
                return 0;
            return 1;
    }
}
```

Figure 2. *Code for tolerant Tit-for-Tat*

The history rectification strategy performed better than *NastyPlayer*. However, to improve further, the next move was to get the best of both worlds by combining the *Tolerant Tit-for-Tat* strategy and *History Rectification* strategy.

3.4. Selected Strategy

The selected strategy is a **blend** of *Tolerance* and *History Rectification* with **minor optimizations to maximize reward** in edge cases. Following are the important components of the implemented strategy:

- In the start of the game, the agent begins with **cooperation** to show friendliness towards the opponents. This helps to minimize diminishing payoffs in the future rounds that can be caused if opponents become **hostile** and start defecting. This can be verified from the Payoff Matrix as well.
  This follows Axelrod's suggestion to *be nice and start by cooperating*.

```
// at the start of the game, cooperate to show that you are nice
if (n == 0) {
    return 0;
}
```

Figure 3. *Strategy for start of the game*

- If the round number is 109, that is, definitely the last round of the game (since there can only be a maximum of 110 rounds). We know that the opponents **cannot retaliate** and hence the agent will defect for maximum payoff.
  This is a slight tweak to Axelrod's suggestion since the number of rounds in a game are limited and hence such a selfish move can be taken.

```
// since the current round is the last round, opponents cannot retaliate
// hence, be selfish and defect for maximum payoff
else if (n == 109) {
    return 1;
}
```

Figure 4. *Strategy for end of the game*

- Given the histories of both the opponents, if both took the same move in the previous round, **reciprocate by following the same move** in the current round. Doing this has the advantages as mentioned in the table below:

Scenario	*Both opponents cooperated*	*Both opponents defected*
We cooperated	No negative impact as we continue to reward cooperation with cooperation and so should the other rational agents.	We punish the defection of both the opponents by defecting. At the same time, the opponents should be rationally rewarding our cooperation in the previous round.
We defected	There is a possibility that our opponents may now become hostile due to our defection. Hence, to avoid the chain of defections, cooperate to show friendliness.	Punish the defection of opponents in the previous round by defecting again. This also goes to show opponents that further defections cannot be exploited by them for their own gain.

The code for the same is shown in Figure 5 below:

```
// if both competitors did the same move in the previous round
// follow the same to punish / show friendliness
if (oppHistory1[n - 1] == oppHistory2[n - 1]) {
    return oppHistory1[n - 1];
}
```

Figure 5. *Strategy when both opponents do the same action*

- **30%** of the times, the agent acts as a *Tolerant Tit-for-Tat* player. This means that it does not react to the cooperation and defection of each opponent individually. Instead, it considers the number of cooperation and number of defections as a whole. Further, in the tolerant phase, it does not take its own history into account and instead makes decisions solely based on the opponent's histories.

  If the total number of times the opponents cooperated is lesser than the total number of times they have defected so far, then my agent reacts with a defection as a punishment to the opponents as a whole.

If the total number of times the opponents cooperated is greater than the total number of times they have defected so far, then my agent reciprocates the general friendliness of the opponents by cooperating again. This also indicates to the opponents our preference for maximum payoff via friendliness.

The code implementation of this section is shown in Figure 6 below.

```
// for 30% of the times, be a tolerant player
// decision is made by seeing number of cooperations and defections of the
// opponents as a whole
if (randomNum < 0.3) {

    // initialize the total number of cooperations and defections of the opponent
    int numOppCoops = 0;
    int numOppDefects = 0;

    // iterate through the histories of both the opponents
    for (int i = 0; i < n; i++) {
        if (oppHistory1[i] == 0)
            numOppCoops += 1;
        else
            numOppDefects += 1;

        if (oppHistory2[i] == 0)
            numOppCoops += 1;
        else
            numOppDefects += 1;
    }

    // if the opponents as whole have made more defections than cooperations, defect
    // else cooperate
    if (numOppDefects > numOppCoops)
        return 1;
    return 0;
}
```

Figure 6. *Tolerant strategy for 30% of the time*

- However, for majority of the time (**70%**), my agent takes its history into account as well and takes rectifying steps to decide the action.

  If the number of defections of my agent so far surpass that of both the opponents, then it acts as a signal that my agent might be a bit aggressive and hence could lead to more defections from opponents in future rounds due to **mistrust and grudges**. Hence, it cooperates in the current round. In all other cases, it responds with a defection to punish the opponent who has been defecting more than us.

  The code implementation of this section is depicted in Figure 7 below.

```
// iterate through your history for the rest 70% of the times
else {

    // initialize variables to track number of defections
    int numOpp1Defects = 0;
    int numOpp2Defects = 0;
    int numMyDefects = 0;

    // iterate through all histories to get number of defections of each
    for (int i = 0; i < n; i++) {
        numOpp1Defects += oppHistory1[i];
        numOpp2Defects += oppHistory2[i];
        numMyDefects += myHistory[i];
    }

    // if my agent has been defecting more than both, cooperate to break cycle
    if (numMyDefects > numOpp1Defects && numMyDefects > numOpp2Defects)
        return 0;
    return 1;
}
```

Figure 7. *History rectification strategy for 70% of the time*

From the above, it is clear that my agent follows **Program Equilibria**, that is, my agent will be **willing to cooperate only if the opponents cooperate**. At the same time, it punishes defections using only measured force (does not overdo punishments). This is in line with Axelrod's suggestions.

4. Results

4.1. Process to judge performance

Since multiple strategies were used, it was essential to **average out** the performance of the different strategies on a common ground. For this, the provided code template for *ThreePrisonersDilemma.java* was changed a bit.

Following are important components of the code to judge performance of different strategies:

- A total of **22 different strategies** were made to play with each other. This included simple strategies like *NastyPlayer*, *NicePlayer* etc.
- The tournament was run **500 times** to average out the performance of strategies across tournaments and to generate a more high-level overview of the performance of each strategy in terms of the **probability of it securing a particular rank**.
  After each tournament, the rank of the strategy of interest was recorded in an array *ranks*. Hence, *ranks[i]* represented the number of times rank *i* was obtained.

```
int numTournaments = 500;
for (int i = 0; i < numTournaments; i++) {
    System.out.println("--------------------------------");
    System.out.println("Tournament: " + Integer.toString(i));
    System.out.println("--------------------------------");
    System.out.println();

    ThreePrisonersDilemma instance = new ThreePrisonersDilemma();
    int currentRank = instance.runTournament();
    ranks[currentRank]++;
}
```

Figure 8. *Code snippet to run 500 tournaments*

- At the end of the 500 tournaments, the results were displayed on the command line itself. The results stated the probability of the selected strategy (in my case, the strategy selected was *Bhatia_Ritik_Player*) to secure different ranks.
  The probability of obtaining a specific rank *i* was calculated by dividing the number of times rank *i* was obtained during simulation, divided by total number of tournaments.

```
System.out.println("--------------------------------");
System.out.println("Results");
System.out.println("--------------------------------");
for (int i = 0; i < 22; i++) {
    System.out.println("Rank " + Integer.toString(i + 1) + " with probability: "
            + Float.toString((ranks[i] / (float) numTournaments) * 100) + "%");
}
```

Figure 9. *Code snippet to display results after 500 tournaments*

4.2. Performance of NastyPlayer strategy

The results of the *NastyPlayer* strategy are shown in Figure 10 below. As can be seen, the results of this strategy are abysmal – it ranked **last** in **all** the tournaments (500 tournaments were run), proving that an always defect strategy will not lead to a good ranking during the evaluation process in this assignment.

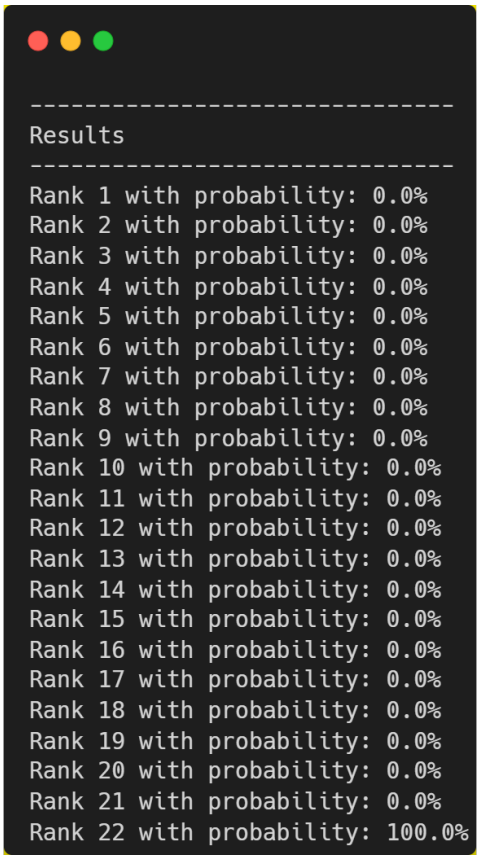

Figure 10. *Results for NastyPlayer strategy*

4.3. Performance of Tolerant Tit-for-Tat strategy

The results of the *Tolerant Tit-for-Tat* strategy are shown in Figure 11 below.
As is evident from the figure, the results of this strategy are much better than the *NastyPlayer*, proving that using the qualities of this strategy in the final strategy would be a good option.

The overall results of this strategy are:

- Probability of being in the Top 5: **44.5%**
- Probability of being in the Top 10: **81%**

The results obtained above are pretty impressive, thereby proving that incorporating *Tolerant Tit-for-Tat* in the final strategy was a step in the right direction.

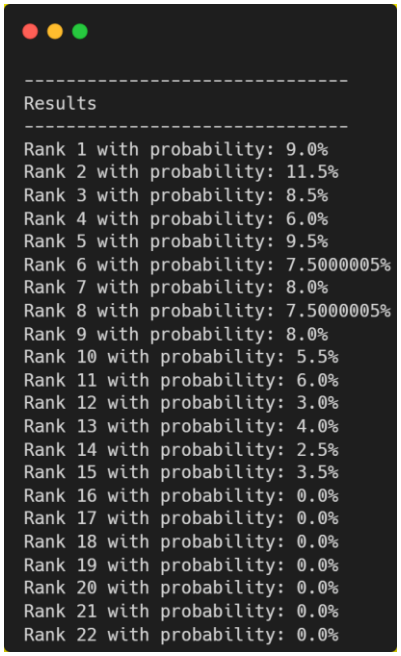

Figure 11. *Results for Tolerant Tit-for-Tat strategy*

4.4. Performance of History Rectification strategy

The overall results of this strategy are:

- Probability of being in the Top 5: **49.5%**
- Probability of being in the Top 10: **81.5%**

Again, the results of this strategy are very impressive (much better than *NastyPlayer* and slightly better than *Tolerant Tit-for-Tat*), proving correct, the decision of incorporating it too.

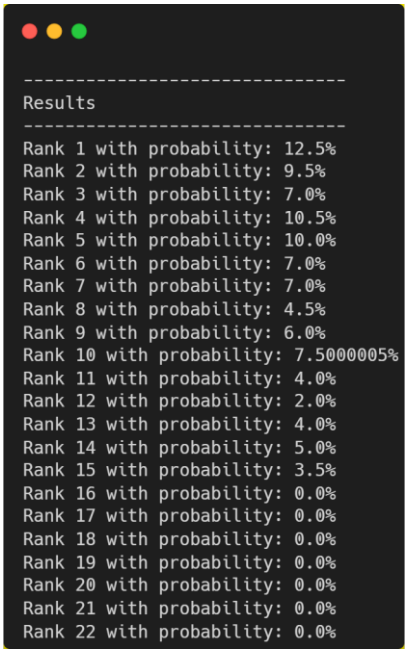

Figure 12. *Results for History Rectification strategy*

4.5. Performance of selected strategy

The performance of the selected strategy was measured by running the custom *ThreePrisonersDilemma.java* file with the modifications mentioned in section 4.1. above.

The result of one of the tournaments is shown in Figure 13 below. It shows the performance of *Bhatia_Ritik_Player* strategy against all other strategies (including the example strategies originally provided in *ThreePrisonersDilemma.java*)

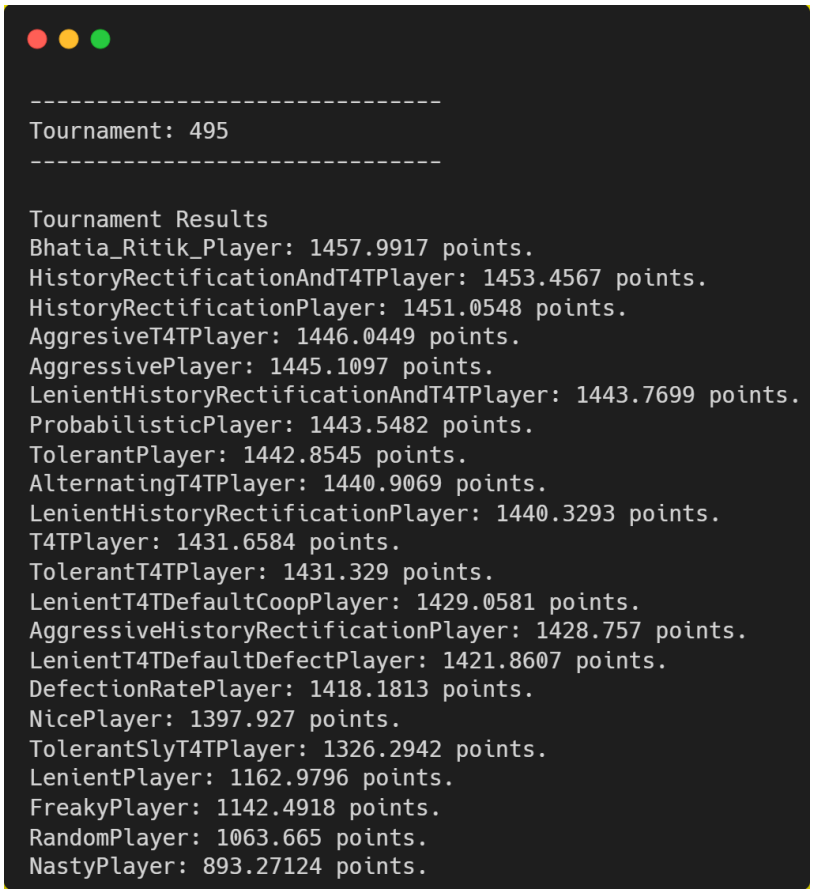

Figure 13. *Tournament Results display format*

As we can see from results of the last tournament in Figure 13, my agent *Bhatia_Ritik_Player* performs very well and ends up in the first position. However, this is only a single tournament and not representative of the overall performance of the strategy (since the number of rounds can vary between 90 and 110).

The actual performance of the model can be judged from Figure 14 below. As can be inferred from the same, the average probability of securing a rank in the **Top 5** for *Bhatia_Ritik_Player* is **50.6%** and the average probability of securing a rank in the **Top 10** is **82.4%**.

Although the set of competitor strategies is far smaller than the actual size that will be used for evaluation, it still includes one of the best performing strategies. Further, since there is no strategy that is guaranteed to always produce an optimal result, the performance of *Bhatia_Ritik_Player* among 22 competitor strategies **is decent** and hence is a **good choice** of strategy for submission.

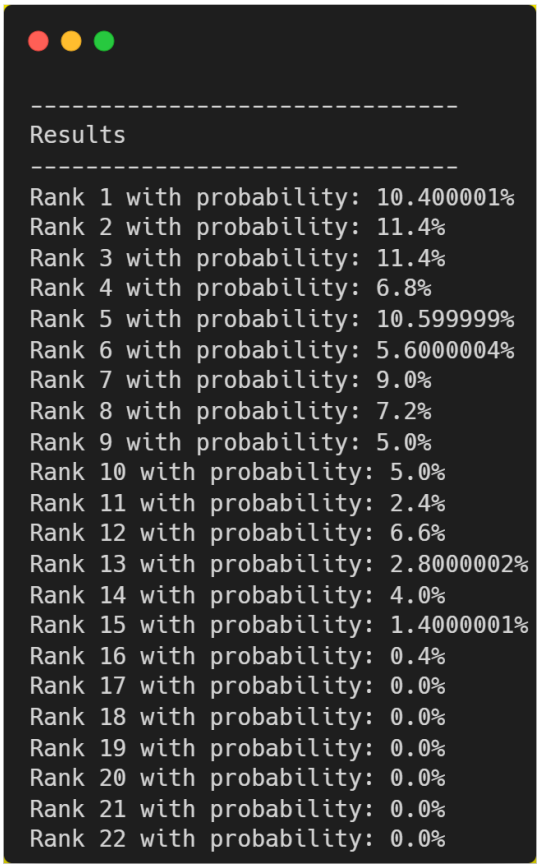

Figure 14. *Probability distribution of the rank of Bhatia_Ritik_Player strategy*

Another point that can be noted from the results in Figure 14 is that the probability of my strategy securing a lower rank **rapidly diminishes to 0 for the lower ranks**, which gives further confidence that the strategy can perform well during the actual evaluation.

5. Conclusion

It is impossible to develop a strategy that can always guarantee a win. This is because of the **sheer number of variables** and the **possibly infinite number of strategies** that can be made for this. However, several research studies have performed empirical tests on the most common strategies and their performance evaluation has helped find common characteristics of relatively successful strategies.

For this assignment, the strategy that I have designed **aims to build off of commonly successful algorithms with certain tweaks** to perform better, given the conditions of the problem. Although securing rank 1 is again not always assured, it is possible to secure a decent rank by factoring in the traits of generic successful algorithms such as *Tit-for-Tat* and *soft majority*, as well as the suggestions given by Axelrod for this tournament.

The agent that I have designed for this assignment is a blend of the *Tolerant Tit-for-Tat* and *History Rectification* strategies. For the majority (70%) of the time, my strategy actively looks at the history of all the players in the tournament (including itself) and chooses the action that **tones down aggression or ups the ante when being too lenient**. However, for the remaining 30% of the times, my agent follows a *Tit-for-Tat* approach irrespective of the actions that it has taken so far.