NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CE 4046 Intelligent Agents

Assignment 2 Report

Three Prisoner's Dilemma

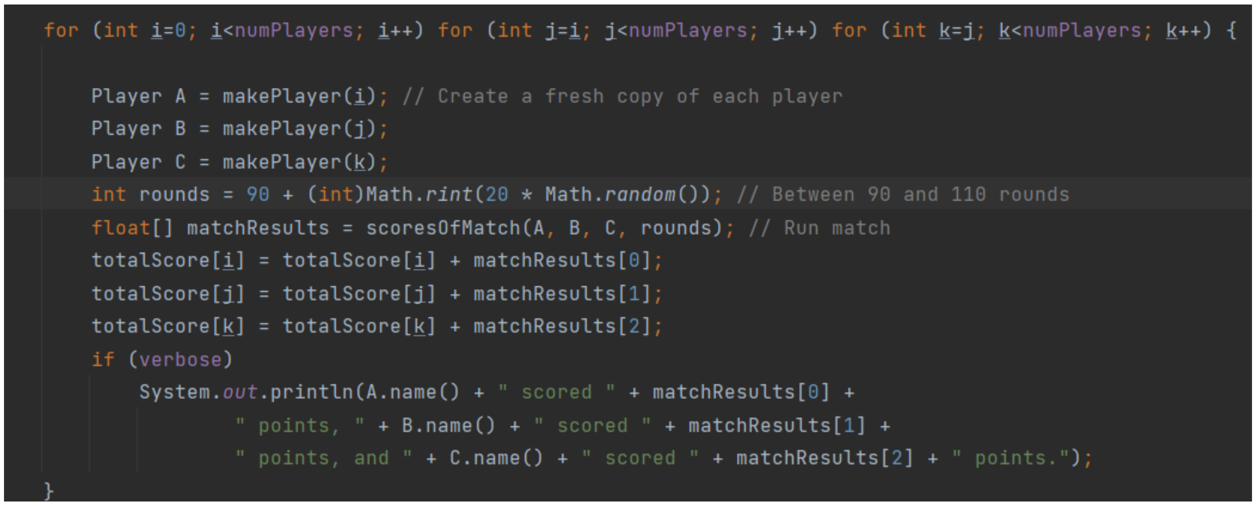
Mervyn Chiong Jia Rong U1921023K

Content Page

1 Project Description

Commonly known Prisoner's Dilemma's best choice of action would be for both players to defect. However, in the case of multiple games being played and the addition of a third player, this is no longer the case. Let us examine that data provided to us first.

1.1 Initialisation



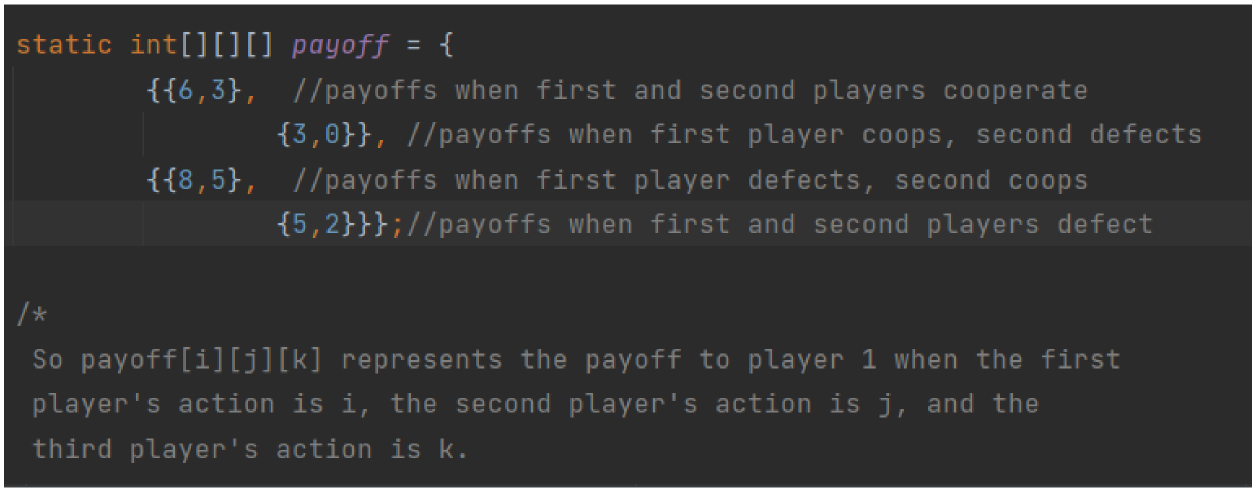
```
for (int i=0; i<numPlayers; i++) for (int j=i; j<numPlayers; j++) for (int k=j; k<numPlayers; k++) {

    Player A = makePlayer(i); // Create a fresh copy of each player
    Player B = makePlayer(j);
    Player C = makePlayer(k);
    int rounds = 90 + (int)Math.rint(20 * Math.random()); // Between 90 and 110 rounds
    float[] matchResults = scoresOfMatch(A, B, C, rounds); // Run match
    totalScore[i] = totalScore[i] + matchResults[0];
    totalScore[j] = totalScore[j] + matchResults[1];
    totalScore[k] = totalScore[k] + matchResults[2];
    if (verbose)
        System.out.println(A.name() + " scored " + matchResults[0] +
                " points, " + B.name() + " scored " + matchResults[1] +
                " points, and " + C.name() + " scored " + matchResults[2] + " points.");
}
```

Figure 1: Code Snippet for Round and Player Creation

The total number of rounds in a single match is actually derived from the above fig. Evidently, The total number of rounds in a match is somewhere between 90 to 110. This also will create 3 random players, meaning that there could be players with the same strategy fighting against each other.

1.2 Reward System



```
static int[][][] payoff = {
        {{6,3},  //payoffs when first and second players cooperate
                {3,0}}, //payoffs when first player coops, second defects
        {{8,5},  //payoffs when first player defects, second coops
                {5,2}}};//payoffs when first and second players defect

/*
 So payoff[i][j][k] represents the payoff to player 1 when the first
 player's action is i, the second player's action is j, and the
 third player's action is k.
```

Figure 2: Reward System for Player Decisions

The above Fig showcases the overall reward system for the game. To make it simpler to understand,the below figures showcases what happens should 3 players all play at the same time.

Letter Representation	Agents	Rewards to Agent A
	A|| B ||C	
CCC	000	6
CCD	001	3
CDC	010	3
CDD	011	0
DCC	100	8
DCD	101	5
DDC	110	5
DDD	111	2

Figure 3: Reward Table for Agent A

Putting all the results and outcomes for all Agents, we will get the below table.

If Agent A Coops		Agent C	
		Coop	Defect
Agent B	Coop	(6,6,6)	(3,3,8)
	Defect	(3,8,3)	(0,5,5)

Figure 4: Reward Table for Agents if A Cooperates

If Agent A Defects		Agent C	
		Coop	Defect

Agent B	Coop	(8,5,3)	(5,0,5)
	Defect	(5,5,0)	(2,2,2)

Figure 5: Reward Table for Agents if A Defects

Looking from the above tables, ideally our agent should aim to have an average score of 5 at all times. Which require players to actually defect as a result. Hence, we will need to ensure players keep defecting but do not feel exploitation whilst maintaining a high score average.

2 Environment Testing

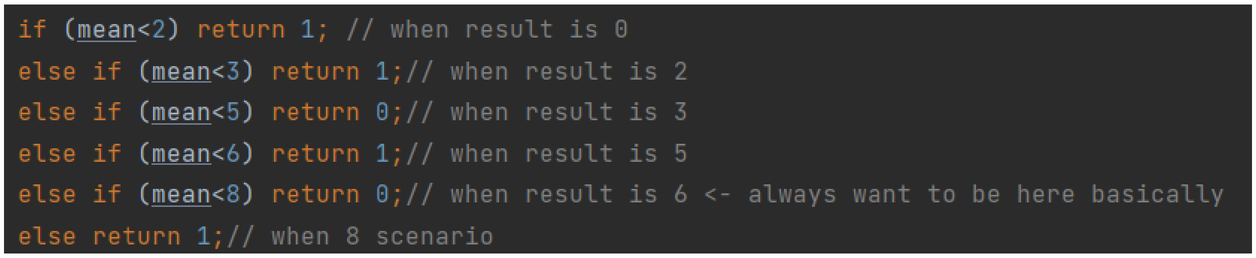
Firstly, before we create an agent, we need to know what our opponents will and can play to ensure our agent is adaptive enough and not predictable.Hence, we added several agents into the environment field.
NOTE: The code for these implementations can be found within the ThreePrisonerDilemma.java file included in the zip file

2.1 Agent: PROBER

Prober is based on the handshake belief which is to get a gist of what others are playing and then play appropriately. Hence, it will first play Defect followed by co-operate and cooperate.(for explanation purposes, i will say defect as =1 and co-operate as 0)Depending on the outcome from the 3 rounds. If the opponents have played 0 and 0 for rounds 2 and 3 this would imply that they are using a “always cooperate” or a “forgiving strategy” and our response is to capitalise on such actions,which we would play 1 in response. If this is not the case, it will default back to Tit for Tat stratagem.
One down side regarding this strategy is that there is no adaptability to its strategy and will stick to the method until the end of the match. Works more so for a smaller match and round size.

2.2 Agent: ADAPTIVE

Similar to the handshake theory, we will play a sequence of 0,0,0,0,0 followed by 1,1,1,1,1. From this outcome we calculate the average mean score after every turn. From the average mean score we take the appropriate action in this case, we adopted a very simple way of approaching the strategy.



```
if (mean<2) return 1; // when result is 0
else if (mean<3) return 1;// when result is 2
else if (mean<5) return 0;// when result is 3
else if (mean<6) return 1;// when result is 5
else if (mean<8) return 0;// when result is 6 <- always want to be here basically
else return 1;// when 8 scenario
```

Figure 6: Code working

Explanation:
When the player’s average is <2 this would mean that the result of the precious round is probably 011 or 111, based on the score chart, our responsive action should be to return 1, assuming that if they both do not change you will still get a average of 2 and should even 1 of them change your will get a value of 5. It is with this line of logic that we service the values for the remainder cases with the only exception being when mean is <8. This is dangerous as this is not where you would want to be ideally. Because it would imply that the opponents get 5,3 looking at fig 1.2.4. Based on the number of scenarios, it makes the most sense to return a

value of 1. 3 scenarios where returning a 0 will result in a decent outcome with 1 major disadvantageous outcome. However, returning a 1 will result in 4 relatively positive outcomes.

2.3 Agent : PAVLOV

For this agent, I've implemented 2 different variations, namely PAVLOV1 and PAVLOV 2.
PAVLOV1:
Play Tit For Tat for the first 6 moves as such it will first lead with 0 then starts to identify what strategy the opponents are playing based on the subsequent responses.This is easily tracked by summing the outcome that you've played. (I.e) if the sum=3 this means you have played 3x1s and 3x0s.
From that,I've categorised 4 different strategies:

- Tit for Tat, when sum == 0, implying that it has been all 0s. Hence the response to such would be to carry on cooperating.
- Always Defect, When the sum is >=4, in a scenario where the majority are defecting the best choice of action would be to defect.
- Suspicious Tit for Tat, when the sum ==3, this would require tit for 2 tats, to recover.
- Finally, Random strategy, which for the purpose of the experiment will be any strategy aside from the above mentioned. The response will be to always defect as a result.

PAVLOV2:
Similar to PAVLOV1, with the only difference in that it will re-calculate the outcome based on a preset threshold value. Comparing the average rewards from past iterations to see if the outcome is optimal and if it is not, restart the function by playing Tit for Tat and replay the whole agent.

2.4 Agent: User Created

Aside from the above mentioned agents, we also included agents from other past works on github. Such as nasty2 and win stay lose shift.

Nasty2: from “alnightyGOSU” on github, which counts the number of times agents defect. Also checks if the agent “wants” to cooperate based on a threshold value set.

Winstayloseshift: Straightforward, if the payoff is >=5 then stay(win scenario). Else return the opposite of what was previously played

Mundhra_Shreyas: Usage of probability distribution to derive utility values to determine optimal action to take

2.5 Environment Results

Running these agents in the environment for a total of 10 matches will net the following results.

Agent	Match 1 Score	Match 2 Score	Match 3 Score	Match 4 Score	Match 5 Score
Nasty2	433.64984	430.74582	433.98618	439.90634	433.6979
Tolerant Player	430.98288	424.48035	444.4245	439.8311	437.60086
T4TPlayer	423.35968	430.59467	413.3729	425.97882	434.69254
WinStayLose Shift	421.0413	411.12955	415.8834	429.25876	421.99048
PAVLOV2	417.10516	420.88846	428.3451	421.56488	436.8025
PAVLOV1	413.34647	432.62317	424.9523	430.13293	432.06165
NicePlayer	392.28516	399.2767	413.06442	402.3773	392.50385
FreakyPlayer	358.69818	384.99603	356.96173	374.88718	378.46066
NastyPlayer	339.14554	344.39178	350.7623	328.61548	344.9456
PROBER	336.34592	335.09958	330.71667	348.0479	355.85028
ADAPTIVE	324.53995	349.22562	349.74515	345.39352	341.34015
RandomPlay er	320.53162	334.61002	331.6714	325.11777	339.92517

Figure 7: Agent Scoreboard 1

Agent	Match 6 Score	Match 7 Score	Match 8 Score	Match 9 Score	Match 10 Score	Average Score
Nasty2	419.74216	424.21652	430.97467	434.53488	433.4958	431.49501
Tolerant Player	430.40427	443.49246	440.90073	431.83694	430.0345	435.39886
T4TPlayer	431.05344	417.4378	434.43527	427.39645	415.44946	425.3771
WinStayLoseShift	431.659	412.08112	410.7341	409.9171	430.29907	419.39939
PAVLOV2	430.4749	416.84363	417.83975	428.95526	430.28333	424.9103
PAVLOV1	436.15375	435.69714	437.65918	423.51074	429.77304	429.59104
NicePlayer	401.50653	396.85025	390.90472	402.05972	389.1621	397.99908
FreakyPlayer	393.18854	360.28098	377.88794	373.80286	348.83682	370.80009
NastyPlayer	346.05408	348.08832	331.81433	339.8575	344.3024	341.79773
PROBER	321.65958	332.01514	333.211	334.9226	335.37827	336.32469
ADAPTIVE	341.7294	342.21313	333.20697	331.32367	337.86438	339.65819
RandomPlayer	323.93204	329.10306	324.40674	329.6428	322.71893	328.16596

Figure 7: Agent Scoreboard 2

From the average score we can then create a ranking as follows:

1) Tolerant Player
2) Nasty2
3) PAVLOV1
4) T4TPLAYER
5) PAVLOV2
6) WinStayLoseshift
7) Nice Player
8) Freaky Player
9) Nasty Player
10) Adaptive
11) Prober
12) Random Player

As such we will be modelling our agent based on the top 5 agents.

*Disclaimer, shreyas's agent was not added because the agent realistically managed to get an average of 1.6 in the rankings. Which is something worth studying altogether and I will go more in depth in the later chapters because it is something I have adopted in my agent design.

2.6 Conclusions Derived

From the above mentioned figures, we can see that it is imperative to have an adaptive strategy and not stick to a single methodology throughout the game to avoid exploitation from agents. Additionally, the agent must not be predictable so as to ensure other agents do not assume we are taking from a well known stratagem. Checking the match results of opponents is also required as tolerant players relied on such a strategy.

Additionally, Shreyas' agent relied on probability distribution to determine his response to what his opponents were playing. However, there was something that I realised that Shreyas could have included to further make the model better.

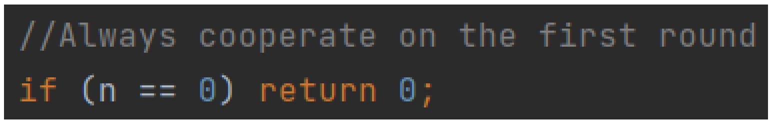
3 Agent Creation

To create a reliable agent we need to make some assumptions:

1) Other agents in the tournament will have the same strategy as me
    a) Simply put, it is highly possible that others will be using a variant of the same strategy as me. As such, in the event our agent is performing against itself it must take countermeasures to be one step ahead.
2) Other agents are extremely complex, reading or anticipating a specific strategy is not ideal.
    a) Other agents are smart, as such they will use a wide array of strategies to be on top. This also implies they are not self destructive in any way possible. In other words, conventional strategies such as Tit for tat will most likely not be used. Should not base prediction on strategies already out on the internet and instead base it on past history of actions.

3.1 Agent Explanation and Scenarios

Based on the assumptions mentioned above, we shall begin creating our agent. Because we are going with a more reactive approach the only time our actions are actually “clear” is the 1st move.

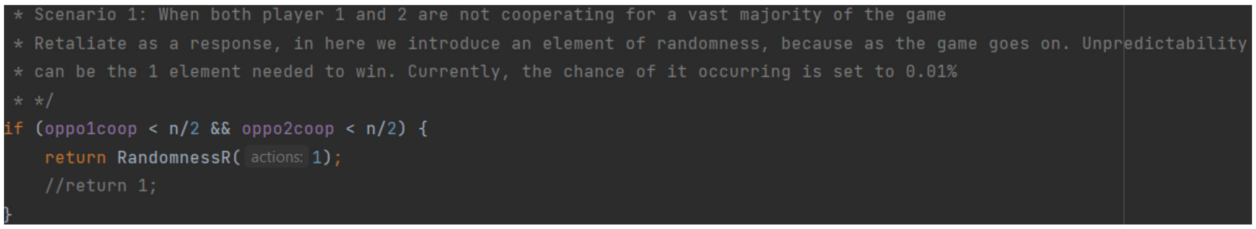


```
//Always cooperate on the first round
if (n == 0) return 0;
```

Figure 8: Starting Response

Always cooperate in the first round, no reason to defect. More often than not, most agents actually cooperate rather than defect based on analysis of multiple strategies.
Now that we have finished with our first round, this sets up the reactive strategies of our agent.

3.2 Scenario 1: 2 Defective Agents

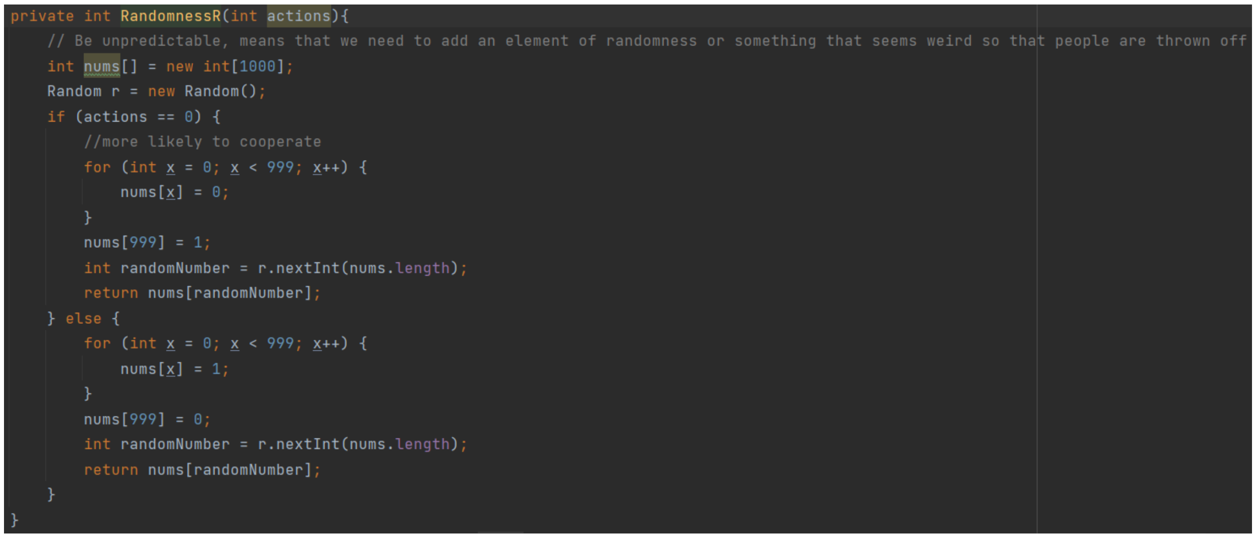


```
 * Scenario 1: When both player 1 and 2 are not cooperating for a vast majority of the game
 * Retaliate as a response, in here we introduce an element of randomness, because as the game goes on. Unpredictability
 * can be the 1 element needed to win. Currently, the chance of it occurring is set to 0.01%
 * */
if (oppo1coop < n/2 && oppo2coop < n/2) {
    return RandomnessR( actions: 1);
    //return 1;
}
```

Figure 9: Defective agents

In the event agents have been mostly defective for more than half the total game length. It is in our best effort to defect. However, this may not necessarily break the never ending cycle of defection. As such, we need to introduce an element of unpredictability.

3.2.1 RandomnessR

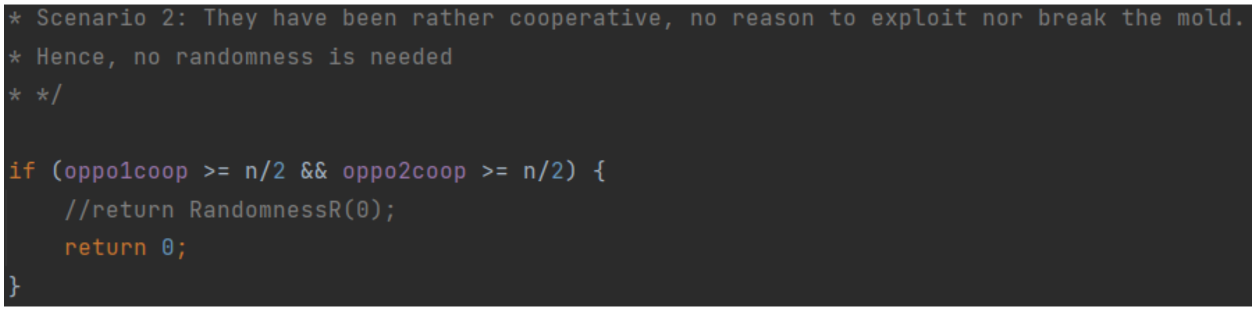


```java
private int RandomnessR(int actions){
    // Be unpredictable, means that we need to add an element of randomness or something that seems weird so that people are thrown off
    int nums[] = new int[1000];
    Random r = new Random();
    if (actions == 0) {
        //more likely to cooperate
        for (int x = 0; x < 999; x++) {
            nums[x] = 0;
        }
        nums[999] = 1;
        int randomNumber = r.nextInt(nums.length);
        return nums[randomNumber];
    } else {
        for (int x = 0; x < 999; x++) {
            nums[x] = 1;
        }
        nums[999] = 0;
        int randomNumber = r.nextInt(nums.length);
        return nums[randomNumber];
    }
}
```

Figure 10: Random Function

Based on the scenario, we adjust accordingly. The code is a simple number generator that will pick an outcome. (i.e) We have 2 highly defective agents, as such we create an array with 999 slots all with 0s, and leave the last slot to 1. We then use the random module to select an array position to get our outcome to return. This returns the outcome of the majority action to a 99.99% chance and having 0.01% chance that it will return the opposite. Logic as to why we do this, is because we do not want our agent to be predictable or telegraphed. Considering how the tournament is runned 1000 times out of 100 matches. The random aspect allows us to always remain as a wild card as agents won't anticipate it.

3.3 Scenario 2: 2 Cooperative Agents



```java
* Scenario 2: They have been rather cooperative, no reason to exploit nor break the mold.
* Hence, no randomness is needed
* */

if (oppo1coop >= n/2 && oppo2coop >= n/2) {
    //return RandomnessR(0);
    return 0;
}
```

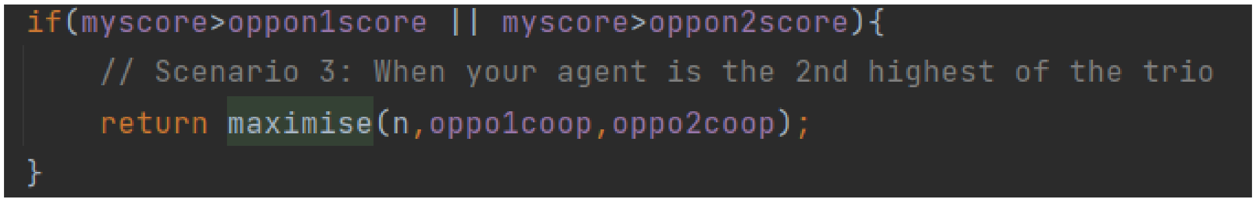
Figure 11: Cooperative Agents

In the event that most agents are cooperative for the vast majority of the game. It does not make any sense to defect and risk agents to be wary of a random aspect that could cause them to start defection. As such, removing the randomness aspect here has proven to fare slightly better to the implemented version.

3.4 Scenario 3: 1 Cooperative 1 Defective Agents

Since we have 2 agents with drastically different strategies. We can no longer base our actions on what the majority has played. As such we are looking to fully maximise our points to remain either 1st or 2nd based on what we think our opponents will do. This is the part that the above function RandomnessR is also imperative to have. In the event, we are facing a clone of our strategy, the opponent will not be able to predict our next move. As such, we are able to remain ahead of our opponents for that round and still maintain a healthy relationship to prevent over exploitation.

3.4.1 Score is Middle of the Pack

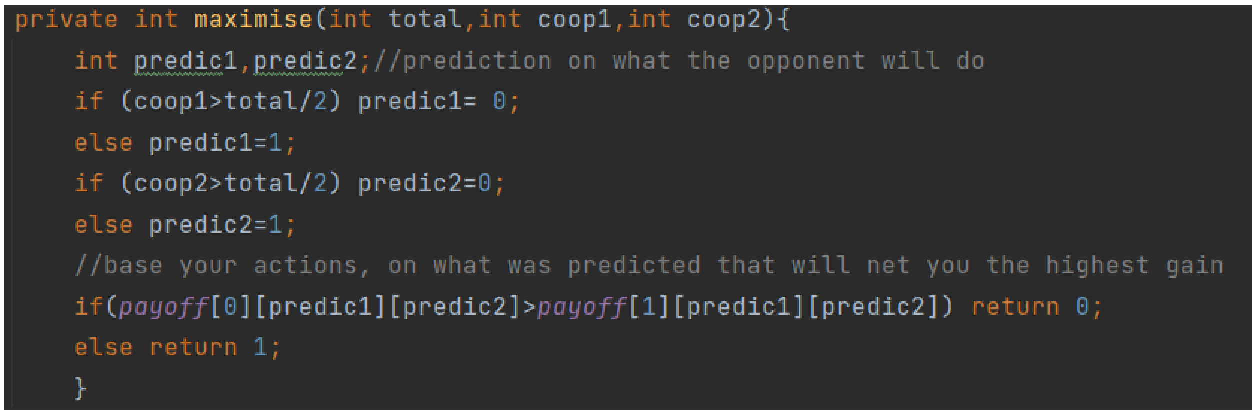


```
if(myscore>oppon1score || myscore>oppon2score){
    // Scenario 3: When your agent is the 2nd highest of the trio
    return maximise(n,oppo1coop,oppo2coop);
}
```

Figure 12: Score is More Than 1 Opponent

This is when your overall score is definitely higher than one of your opponents. With that in mind, this gives us the opportunity to maximise our next action. To do that, we would need to predict what our opponents will play and choose the action that nets us the higher payout.

3.4.2 Maximise



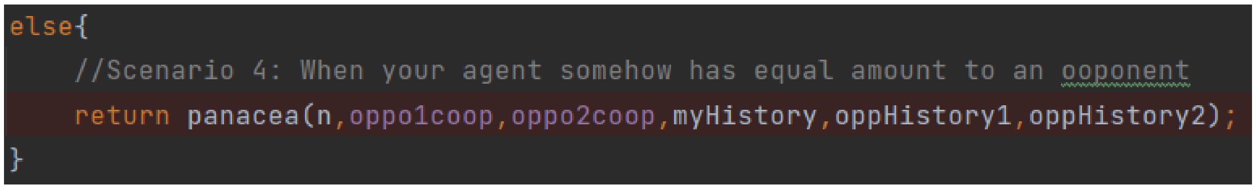
```
private int maximise(int total,int coop1,int coop2){
    int predic1,predic2;//prediction on what the opponent will do
    if (coop1>total/2) predic1= 0;
    else predic1=1;
    if (coop2>total/2) predic2=0;
    else predic2=1;
    //base your actions, on what was predicted that will net you the highest gain
    if(payoff[0][predic1][predic2]>payoff[1][predic1][predic2]) return 0;
    else return 1;
    }
```

Figure 13: Maximise Function

Our prediction is based on their overall actions for the course of the game so far. (i.e)In the event that for 50% of the game they have been playing defect. It is highly likely that defect will also be their next choice of action and vice versa for cooperation. As such, we simply check what is the highest payoff should we play cooperative or defect based on our 2 predictions.

3.4.3 Score is at Equilibrium

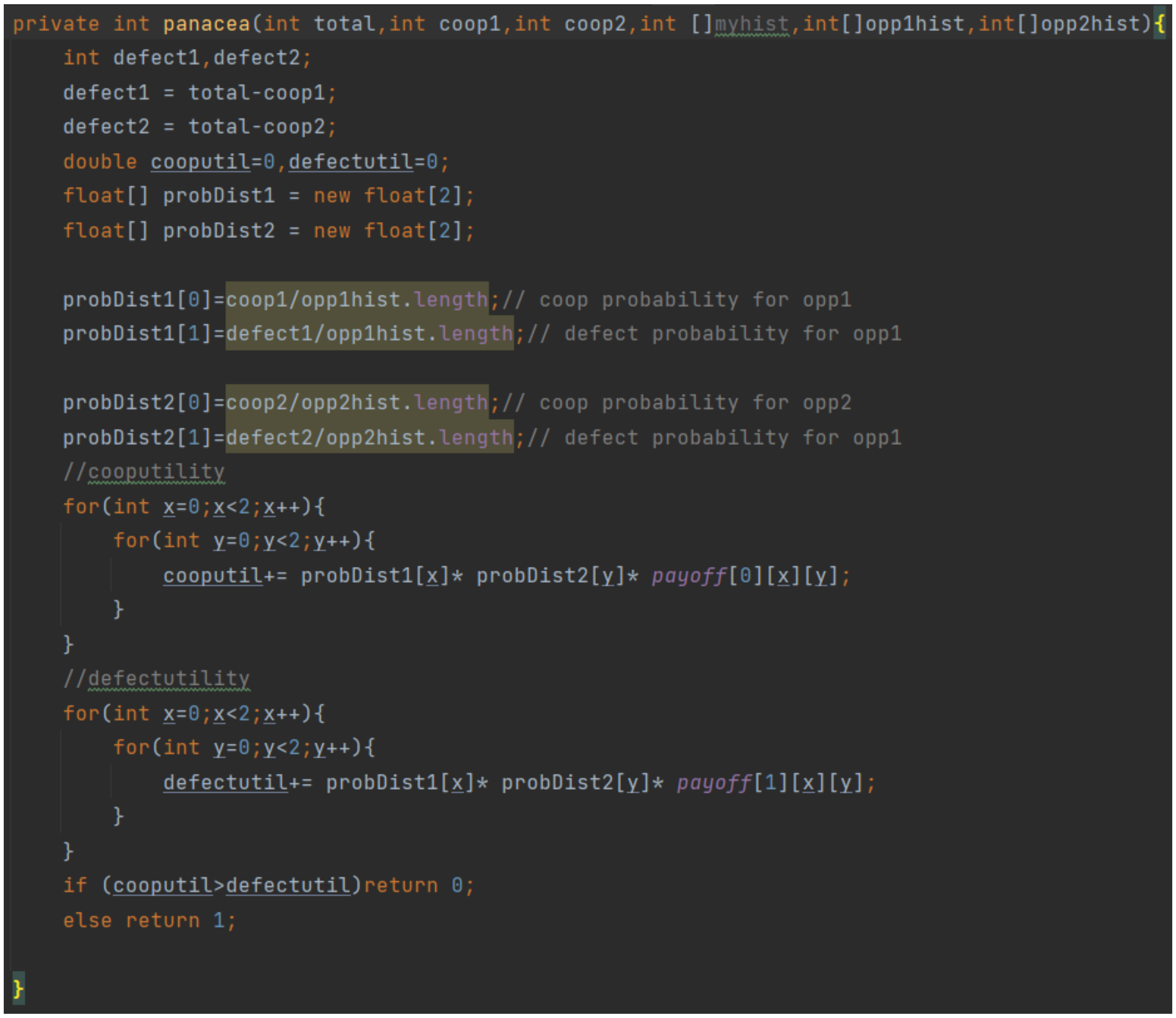
This happens when your score is equivalent to both agents. This raises the complexity in predicting our opponent's next move. As such, we make use of the probability distribution idea from shreyas and implement it.



```java
else{
    //Scenario 4: When your agent somehow has equal amount to an ooponent
    return panacea(n,oppo1coop,oppo2coop,myHistory,oppHistory1,oppHistory2);
}
```

Figure 14: Equilibrium Score

3.4.4 Panacea



```java
private int panacea(int total,int coop1,int coop2,int []myhist,int[]opp1hist,int[]opp2hist){
    int defect1,defect2;
    defect1 = total-coop1;
    defect2 = total-coop2;
    double cooputil=0,defectutil=0;
    float[] probDist1 = new float[2];
    float[] probDist2 = new float[2];

    probDist1[0]=coop1/opp1hist.length;// coop probability for opp1
    probDist1[1]=defect1/opp1hist.length;// defect probability for opp1

    probDist2[0]=coop2/opp2hist.length;// coop probability for opp2
    probDist2[1]=defect2/opp2hist.length;// defect probability for opp1
    //cooputility
    for(int x=0;x<2;x++){
        for(int y=0;y<2;y++){
            cooputil+= probDist1[x]* probDist2[y]* payoff[0][x][y];
        }
    }
    //defectutility
    for(int x=0;x<2;x++){
        for(int y=0;y<2;y++){
            defectutil+= probDist1[x]* probDist2[y]* payoff[1][x][y];
        }
    }
    if (cooputil>defectutil)return 0;
    else return 1;

}
```

Figure 15: Panacea Function

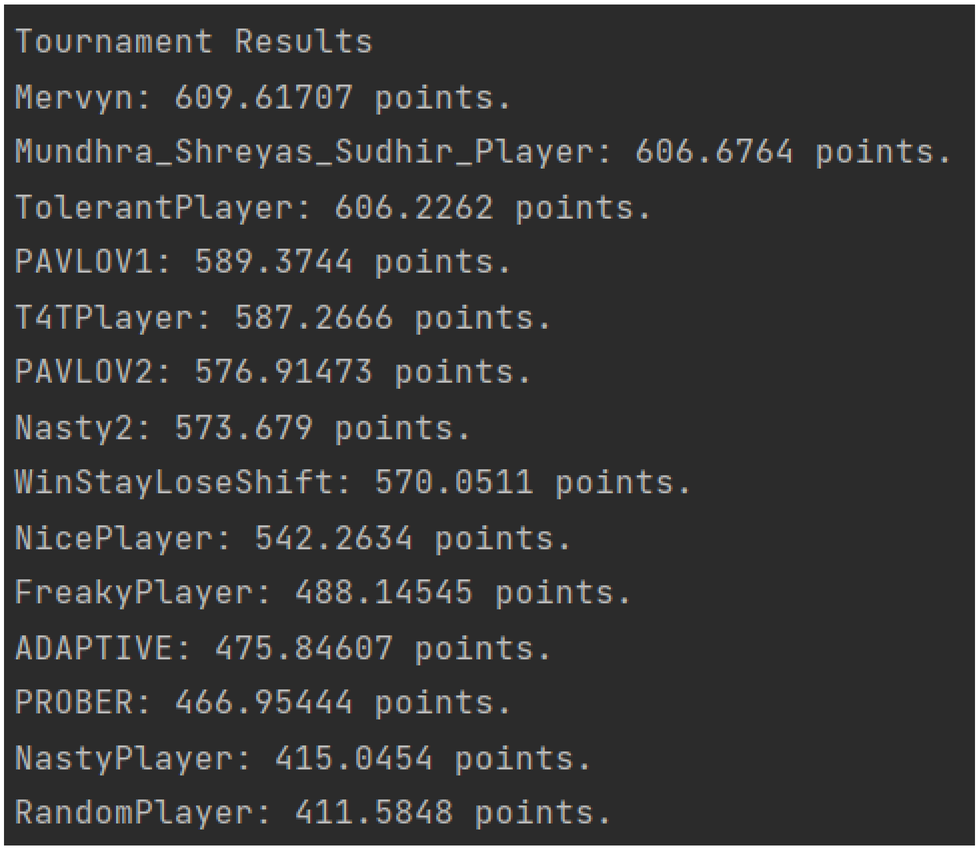
It calculates the co-op defect probability for both players 2 and 3. Then determines our probability distribution with respect to the payoff taking our action into account. Our action here refers to selecting either defect or cooperation.

3.4.5 Example of Distribution formula

(i.e) When action is cooperate, cooperative utility =0;
[1] Cooperativeutility += P(coop probability of player 2) * P(coop probability of player 3)* payoff[0][0][0]
[2] Cooperativeutility += P(coop probability of player 2) * P(defect probability of player 3)* payoff[0][0][1]
[3] Cooperativeutility += P(defect probability of player 2) * P(coop probability of player 3)* payoff[0][1][0]
[4] Cooperativeutility += P(defect probability of player 2) * P(defect probability of player 3)* payoff[0][1][1]

The final value is the cooperative utility. Which is then compared to the defect counterpart and based on whoever is higher we take the resulting action.

4 Agent Performance



```
Tournament Results
Mervyn: 609.61707 points.
Mundhra_Shreyas_Sudhir_Player: 606.6764 points.
TolerantPlayer: 606.2262 points.
PAVLOV1: 589.3744 points.
T4TPlayer: 587.2666 points.
PAVLOV2: 576.91473 points.
Nasty2: 573.679 points.
WinStayLoseShift: 570.0511 points.
NicePlayer: 542.2634 points.
FreakyPlayer: 488.14545 points.
ADAPTIVE: 475.84607 points.
PROBER: 466.95444 points.
NastyPlayer: 415.0454 points.
RandomPlayer: 411.5848 points.
```

Figure 16: 1 Tournament Iteration

The above figure is to show 1 iteration of the tournament of 100 matches. After running about 10-20 tournaments, we can safely say that our agent manages to attain an average of 1.5 in terms of ranking with these players in the tournament. Most of the time it will triumph over all the other agents with some exceptions like shreyas and Tolerant Player. Lets see the head to head match up to see why exactly is able to outperform other agents.

4.1 Comparison: Mervyn vs Shreyas

The only reason why our agent can perform better than shreyas in certain scenarios is because of the random element we added in. The times where shreyas does indeed outperform our agent is possibly due to the 0.01% chance for the opposite action to come up, actually did not show up. This problem should realistically not be an issue, as the assignment will run through multiple iterations. Resulting in a much higher chance to activate the 0.01% action.

4.2 Comparison: Mervyn vs Tolerant Player

Our agent and TolerantPlayer are rather similar actually as both do depend on the past history of the player's actions. TolerantPlayer defects when more than half of a player’s actions are to defect which is what we implemented in our agent. The issue with this, is that most of the time TolerantPlayer does not maximise its payoff despite getting second. This results in TolerantPlayer hovering around the top percentage but is not able to consistently triumph over our agent as our agent does maximise the result even if it is second place.

4.3 Comparison: Mervyn vs PAVLOV1 and 2

PAVLOLV strategy is based on already existing techniques on the internet. Which is something we have assumed that is not widely used as we will be put against one another. Hence, pavlov fails because it is unable to foretell what strategy we are employing. Thereby, assuming we are of the random category.

4.4 Comparison: Mervyn vs T4TPlayer

Titfortat is not realistically reliable as it simply lags behind the opponent and depending on a random choice you will choose which player’s action to replay. No chance to maximise nor anticipate what other players will be playing.

4.5 Comparison: Mervyn vs Nasty2

Nasty2 is essentially an agent that holds grudges and will retaliate when given the chance. However, it will also cooperate when both the other players wish to. Issue here is that it does not take into account when exactly to play the grudge. In the sense that you are not probably maximising your payouts and are simply defecting because of others’ earlier actions.

4.6 Comparison: Mervyn vs WinstayloseShift

WinstayloseShift bases its current action on the last previous move which is not great in the long term. This implies that you will always be one step behind your opponents. Worst case being, your opponents have already figured out your strategy. Our agent bases its actions on the overall number of cooperates and defects played instead of the last move. Allowing it to in a sense predict what the next move will be.

4.7 Comparison: Mervyn vs NicePlayer and NastyPlayer

Niceplayer will always cooperate and NastyPlayer always defects. Issue is that both agents leave room for exploitation as they become predictable which our agent will do to maximise its score in the event they are in the same game.

4.8 Comparison: Mervyn vs FreakyPlayer

FreakyPlayer is somewhat of a wildcard but should not be a large issue as it will still be exploited the same way as NicePlayer or it will respond accordingly if another nastyplayer is met

4.9 Comparison: Mervyn vs ADAPTIVE and PROBER

Similar to how pavlov failed, adaptive is unable to adapt when it does not encounter a strategy it is programmed to handle.

4.10 Comparison: Mervyn vs Random

Random has no real logic behind it. It can have the potential to be the best but also the worst. More often than not, it will be the worst. Unpredictability is somehow predictable. Our agent will heavily exploit it.

5 Conclusion

Our agent is able to consistently outperform common strategies and user created agents. Even if our agent fails in a round there are countermeasures in place to remain relevant.

Overall, the assignment was honestly rather fun to explore the different strategies on the internet and also seeing what our predecessors used.