# CS 154 -- Project Report

Name of the Project: - Prisoner's Dilemma

Submitted by:-

1. Ravi Bhoraskar (08005002)
2. Naman Agarwal (08005062)

## Report:-

- ## The Problem that we intend to Solve:-

  We have created a full simulation package for the Prisoner's dilemma problem. It is more of a simulation than a game. First let us give an introduction to what the prisoner's dilemma is.

  Here is a game matrix for 2 players.

| | **B** cooperates | **B** defects |
|---|---|---|
| **A** cooperates | **A** gets 3<br>**B** gets 3 | **A** gets 0<br>**B** gets 5 |
| **A** defects | **A** gets 5<br>**B** gets 0 | **A** gets 1<br>**B** gets 1 |

  For example in a two player game, the two players are called A and B, and the choices are called "cooperate" and "defect." Players A and B can play a single game by separately (and secretly) choosing either to cooperate or to defect. Once each player has made a choice, he announces it to the other player; and the two then look up their respective scores in the game matrix. Each entry in the matrix is a pair of numbers indicating a score for each player, depending on their choices. Thus, in the example above, if Player A chooses to cooperate while Player B defects, then A gets 2 points and B gets 3 points. If both players defect, they each get 1 point.

  Our project extends this to iterated rounds of n-player prisoner's dilemma game. In an iterated game, there are certain strategies possible to determine whether to cooperate or defect in each round. Some possible strategies are:

  - **Nasty:** Always Defect
  - **Patsy :** Always Cooperate
  - **Egalitarian :** Count the cooperates and defects in the previous rounds and do what is done more times
  - **Nabadu :** Cooperate if winning, defect otherwise
  - .....many other strategies are also possible

Our Project shall perform the following:

Input an a n-player game matrix from the user, and determine whether it is a Prisoner's Dilemma (defined by us)or not.

1. Generate a game matrix for an n-player prisoner's dilemma
2. Play a round for an n-player game and determine the score for each strategy. (One or more of the strategies can be a user choosing 'c' or 'd')
3. Play a tournament, where every 2 strategies play one another, and display the average score for each strategy.
4. Based on the history of previous rounds, we try and predict what strategy each of the n players is using. To do this we run through each previous round and see whether he behaved kike a certain strategy or not . If he did then we increment his character counter by 1 and then average the character count out. This gives the probability of a player playing with that particular strategy. And now we run the games again and using the probability data we calculate the expectation value of the scores.
   Hence we suggest to the user (if he is playing an n-player game), whether he should 'cooperate' or 'defect' in the next round to maximise his score.

- ## Functionality of the program

  We have created an initial list containing all the strategies which are predefined. At the start the function called (simulatesupergame) is called which asks the user to input whether he wants to play an n-player game or wants to run the tournament. Accordingly the function (simulatenplayers) or (simtournament) is called.

- Simulatenplayers

This function takes the number of players, humans , strategies to play against (through a different function inputstrat) and the number of rounds as input from the user and then calls a function (gamestart ... ...) using the inputs as arguements.

- SimTournament

  This function takes in the strategies to be pitted against as input from the user (using the same inputstrat function).Then calls the function (tournament .... ....)

- (Gamestart .. .. .. ..)

  This is the most important function as it runs through the game.Two 2-d vectors are initialised one for maintaining the history of the game so far and one for the value of characters assigned to the unknown players.

  The function **(Simulateround)** generates the result for a single round by taking

input from the human users and by generating the result from the strategies. Then it further calls **(UpdateHistory)** which updates the history vector and adds the result of the present round to it. Then a call to **(UpdateCharMatrix)** is made which updates the charvec (2d vectors of the characters) and further **(updateScore)** is returned which returns the updated list of scores (i.e. the scores at the end of this round)

While taking input from the user we do allow the user to take help. This is the function call that suggests the best possible alternative to the user.

If the user takes the help then a function (**processchar)** is called with the arguments the list of the character values. The function **(processchar)** then does the calculation for the expected value of the scores of both the alternatives and then returns the better alternative.

The game runs at the end of the specified number of rounds it returns the final score list.

- (Tournament)

  The tournament runs the a 2 players prisoners dilemma game taking all possible combinations of the given list of strategies and then prints out the scores of individual matches in a tabular form.

- # Limitations

  The strategy deciphering is limited to only a check through a few strategies it can be expanded to more strategies.
  Also the help function does take considerable time if it is run against a lot of players.

- # Special Features

  Effective use of map, zip , assq and other functions related to lists.
  Imperative programming is widely applied in the updatehistory and updatecharmatrix.