Simulating the Iterated Prisoner's Dilemma in a Population

Daniel Kang and Yachen Sun

1. Background and Introduction

The Prisoner's Dilemma is a classic game theory scenario where one player chooses to either cooperate with or betray the other player, and whether he/she gets rewarded or punished depends on the choice of the other player. The results of these cooperation-defection games can be described by the payoff matrix as shown in 1a. Specifically, if T > R > P > S, the game becomes a Prisoner's Dilemma. For a single gameplay, rational players who wants to maximize their payoff should choose to defect; however, mutual cooperation is also preferred over mutual defection, resulting in the "Dilemma."[1]

	C	**D**
C	R	S
D	T	P

1a. The payoff matrix. *"C" represents cooperation, and "D" represents defection. The matrix should be read this way: if i represents the row number and j represents the column number, Entry (i,j) represents the payoff the player gets when it adopts Strategy i and its opponent adopts Strategy j.*

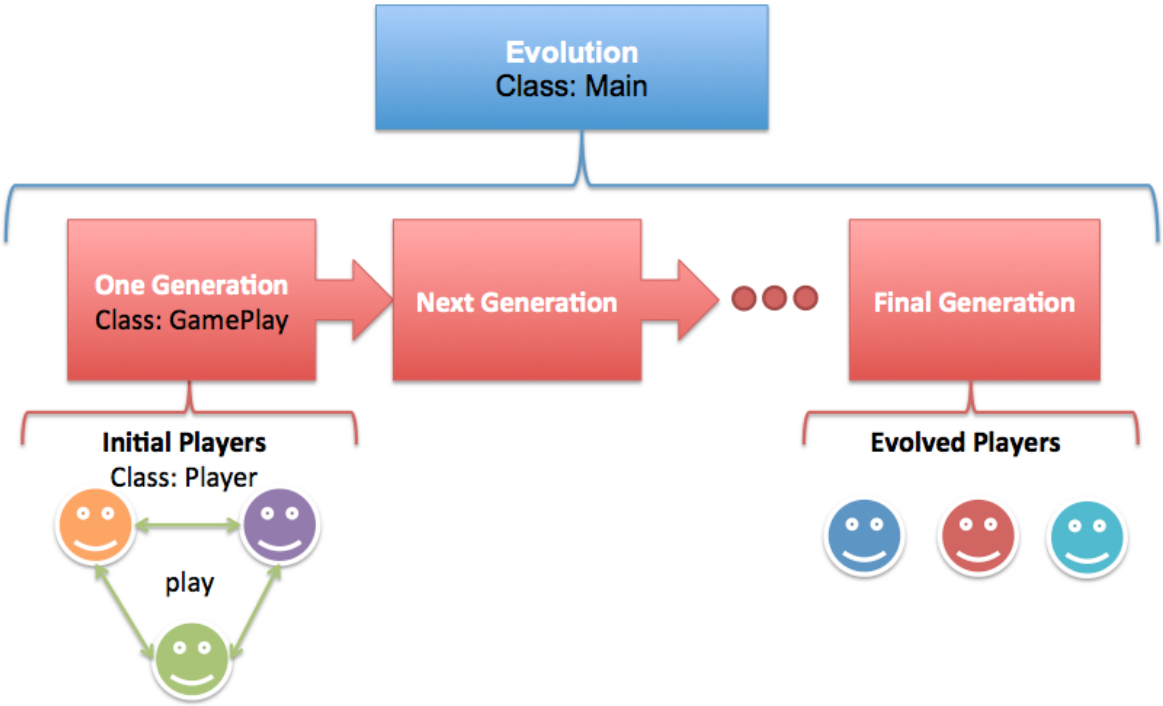
In the Iterated Prisoner's Dilemma (IPD), players play the Prisoner's Dilemma with each other for multiple rounds, which offers a better model of what happens in human societies. Many strategies (these strategies are computer algorithms that decide the next move based on existing iterations) have been developed since the first IPD tournament organized by Robert Axelrod[2]. Martin Nowak subsequently took some of these strategies to make a strategy space for a population of players to choose from. He simulated these players in an evolutionary context, where players occasionally "mutate" to switch strategies, and successful players get to have more offspring. These simulations demonstrated incredible phenomena that play crucial roles in theories of the origin of human cooperation.

In our project, we will simulate the IPD using the evolution of behavior in a population of "well-mixed" players playing against each other. We will allow players to adopt mutations within a mutation space to change their strategies, and the probability of these mutations are dependent on the cognitive abilities associated with each player. We will also use the principles of natural selection and evolution to determine the fitness of each player and let the players' behavior evolve over generations. The goal is to observe the trends of how well different strategies perform in the population at different stages of the game.

[1] Nowak, Martin A. *Evolutionary dynamics*. Harvard University Press, 2006.
[2] Axelrod, Robert, and William D Hamilton. "The evolution of cooperation." *Science* 211.4489 (1981): 1390-1396.

2. Overall Program Structure

The overall structure of the program is shown in Diagram 2a. The entire evolution process is executed by the Main Class, which runs through a fixed number of generations of players to get to a final group of evolved players. Specifically, each generation is represented by an instance of the GamePlay Class, which includes a group of players playing the IPD against each other. After several rounds of play, the fitness scores of all of the players in that generation will be computed and be used to determine which players get to contribute to the "traits" of the next generation. The players in the next generation inherit from the players with better performances in the previous generation, and also go through some mutations to get their own traits. After repeating this process for a certain number of generations, we arrive at the final population of evolved players. The hierarchical nature of the system means that Object-Oriented Programming offers great tools for implementing this simulation.

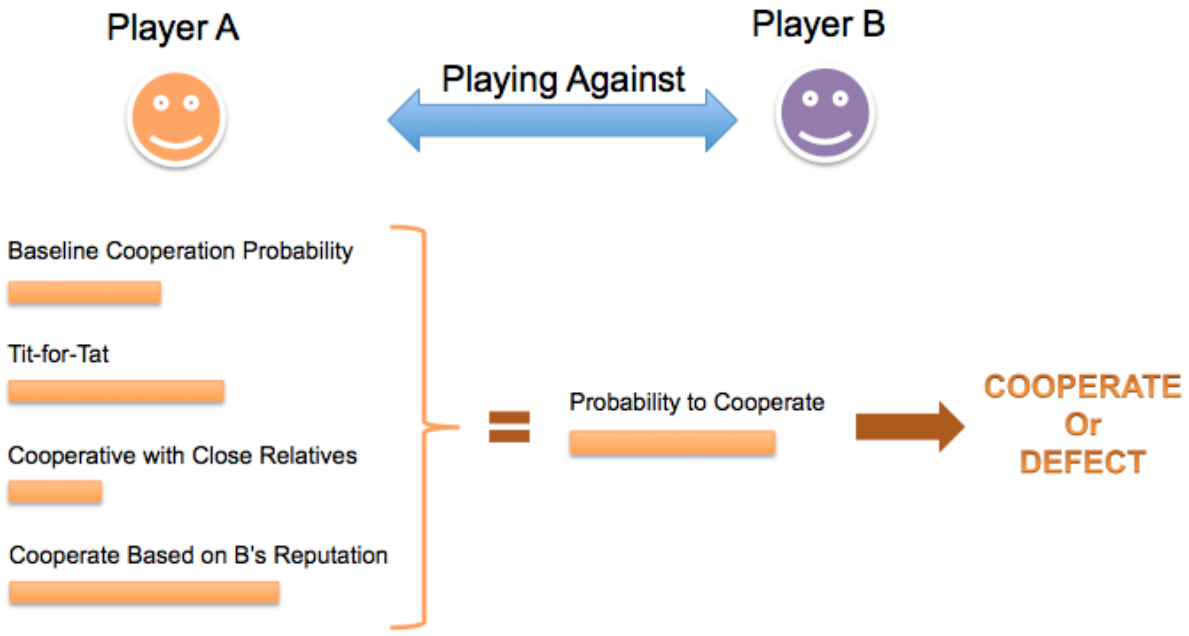


2a. The overall structure of the program.

Since we assume that our population is well-mixed, within each generation, players play the IPD against all the other players in the same generation. For any game between two players, each player's decision is based on that particular player's own unique strategy, as shown in Diagram 2b. All of the player's strategies are a weighted combination of four basic strategies: (1) cooperate based on a constant probability. For example, if the baseline probability is 0, then the strategy is to always defect. (2) Tit-for-tat. For the initial interaction with the opponent, always cooperate. For the subsequent rounds, always repeat the opponent's move in the pair's last game. (3) Cooperate with close relatives. The more "related" the player is to its opponent, the more likely it is to cooperate. (4) Cooperate based

on the opponent's reputation. The better the opponent's reputation is, the more likely the player is to cooperate.

The difference between individual players is that each player weight the four strategies differently. Some may blindly choose to cooperate 100% of the time and pay little attention to the other three strategies, while others might treat all four strategies equally and summarize the result of the four strategies to make their decision. Using a function detailed in Section 3.2, we use a particular player's weights of the four strategies to calculate an overall probability that the player may choose to cooperate in a particular game, and then the player will decide to cooperate or defect based on that overall probability.



2b. Individual player's decision making process during a game.

The specific data structures and member functions of the classes and processes will be explained in detail in the next section.

3. Documentation of Class Data Structures and Member Functions

3.1 Trait

Trait is a class owned by each player. It includes variables that explain the strategies of the players. It also has member functions that are used to define Tit-for-Tat players, cooperative players, defective players, as well as any other custom player. A trait comprises the following factors :

- base : describes the baseline/innate probability that one cooperates.
- basefac, memfac, distfac, repfac (1~100) : each describes the importance of base, memory (past gameplays for Tit-for-Tat strategy), distance (between two players), and reputation (of other player, which depends on past gameplays), respectively, in decision of a move. For example, if a player has high distfac and memfac, it will mostly base its decisions on the distance between the players and the reputation of the other player.
- x,y (1~1000): current positions of the player. They do not only represent the geographical distance, but should ideally represent "social distance." That is, how closely related they are. Additionally, children are "born" in the vicinity of one of the parents, so this can also be considered a measure of the "biological" relatedness between players.

Member functions in trait are used to define a player's characteristic. They are as following :

- Constructors : we have four types of constructors. The default constructor randomly assigns every value within the trait. The second constructor takes in every value except the location as parameters, while the third constructor takes in every value as parameters. The last one is a copy constructor that copies the values from another trait.
- Destructor : Since we do not dynamically allocate any memory, we do not have anything in the destructor.
- Accessor functions getBase(), getX(), getY(), getFac(int): these functions allow outside class to access the factors within trait class. Last one, getFac(int), returns different values depending on specified int. They return basefac(1), memfac(2), distfac(3), repfac(4).
- setFac(int, int) : the first int specifies the trait in the same way as getFac(int), and the second value is the new value assigned to that trait.
- makeTitForTat(), makeCooperate(), makeDefect() : makes the trait represent titfortat, cooperative, defective, respectively. makeTitForTat() sets memfac as 100 and all other values as 0, so that the player bases one's decision solely on the past gameplay. makeCooperate() sets base and baseFac as 100 and all other values as 0, so that the player always cooperate. On the contrary, makeDefect() sets base as 0 and baseFac as 100.
- mutate() : randomly change the traits/factors depending on the mutation rate, which is defined in player.h.

3.2 Player

Instances of the player class represent individual players as shown in Diagram 2a. Instances of this class are used in the decision making of the games and also the production of the next generation of players. This class also keeps track of the generation and reputation of individual players. The data members of this class are:

- mytrait: this is an instance of the "trait" class, and describes the player's strategy in playing games.
- id: this is a unique number assigned to the player for easy identification among one generation of players.
- gen: this number records the generation that the player belongs to.
- team: this number records the team that the player belongs to. The default value is 0, which means that the player does not belong to any team.
- reput (1~100): this value records the current reputation of the player. When players are created, all of them start with a reputation level of 50.

The member functions in this class are:

- Constructors: this class has three constructors. The default constructor creates a player with a random trait instance. The second constructor allows the creation of a customized player with a given trait instance, generation number and id number. The third constructor is a copy constructor.
- Destructor : since we do not dynamically allocate any memory, we do not have anything in the destructor.
- Accessors: there are accessors for all of the data members of this class.
- increaseReput() & decreaseReput(): these functions are used to change the reputation level of a particular player. When a player defects in a game, its reputation level decreases by a certain increment, and the reputation level increases by a certain increment when it chooses to cooperate.
- setTeam(int): changes the team that the player belongs to.
- measureDist(player): this function calculates the Euclidean "social" distance between the current player and another player, which will be used in the "play" member function.
- play(player, int): this function calculates the decision (cooperate or defect) that the current player makes in a game against another player. The function is shown below:

    - prob of cooperation = (basefac * base + memfac * tft + distfac * dist + repfac * rep)/(basefac + memfac+distfac+repfac).
    - The variables used in this function are: basefac, memfac, distfac, repfac, (all between 1 and 100) which shows the relative importance of each strategy, and the probabilities of cooperation associated with each strategy, base, tft, dist, rep (all between 0% and 100%).
    - For basefac, there is a baseline probability of cooperation ("base") that is innate to the player and not influenced by anything else.
    - Memfac is associated with the Tit-for-Tat strategy; specifically, "tft" = 100% if the player has never met the opponent; "tft" = 100% if the last time the two players met, the opponent chose to cooperate; "tft" = 0% if the last time the two players met, the opponent chose to defect.
    - Distfac is associated with the strategy of favoring closely related players in cooperation. Specifically, "dist" is a measure of "social distance" between the player and opponent. If the Euclidean distance between the two players are larger than 500, "dist" is 0%; if the Euclidean distance between the two players are smaller than 500, "dist" linearly increases to 100% as the distance decreases.
    - repfac is used to weight how important the strategy of basing the decision on the opponent's reputation is. "rep" is a probability value that is proportional to the opponent's reputation value.
- reproduce(player, int): this function generates a new player from two parent players. The traits and other properties of the new player are determined as follows:
    - location: the new player will be located in the vicinity (plus or minus 20 in x and y) of one of its parents.
    - base and the four factors in trait: most of these factors will be the same as the parent invoking this function. However, some of these factors may also be from the other parent, and the chance is determined by the crossover rate. Additionally, all of the factors has some probability of going through a mutation, which means that the value may increase or decrease by a small amount (<20).
    - generation: the child's generation will be 1 larger than the parents'.

3.3 GamePlay

The key role of the gamePlay is simulating one round of IPD. Depending on the mode the user selects, it can run any specified number of rounds in a single generation, or predefined number of rounds per generation to simulate mutation and reproduction. The former is designed to offer closer look to individual gameplays and how one's traits affect them, while the latter focuses on mutation and generation change. The outline of a round is as follows (parentheses indicate associated functions) :

1. gamePlay object is created (constructor)
2. Players are added (addTFTPlayer, addCOOplayer, addDEFPlayer, addCUSPlayer, addPlayer)
3. Simulate a round of game (playRound)
    a. Make each pair of players play a game (playPair)
        i. Each player refers to the result matrix to get past play of the other player. This is especially important for players with high memFac, such as Tit-for-Tat players (getLastMove)
        ii. Each player, depending on one's own traits and circumstances, decides on a move ("move" - player.h)
    b. The result of the game is calculated and updated to the result matrix (playPair)
    c. Depending on user's choice, print the resulting matrix or the traits of the players (printMatrix, printTraits)
    d. If it is play-by-rounds, not much action is taken. If it is play by generation, we change generation. Offsprings are created, and mutations are made. (genChange(), "mutate"-trait.h , "reproduce"-player.h)
4. *Depending on user option, players may collude. They may join to make a team, join already existing teams, quit from a team, et cetera. (determineTeam)

We first look at two major data structures in this class : state[][][] and players[]

-state[][][] : The major data structure used in this class is a three dimensional array. Though such an array is not so inefficient in terms of memory space, it is indeed the simplest and fastest implementation there is. This array is the key data structure that is used to compute every single play as well as print out the results. Hence, we dive into the structure with more detailed description.

The three dimensional array has dimensions of (n+1)*(n+1)*3 where n is the number of players. The maximum number of n is same as the maximum number of players, which is defined in gamePlay.h. Let's denote each location in the array using standard notation of (i,j,k). Each value of i and j could be understood as the ID of a player. For example, row 3 and column 3 have data that deals with player 3. Note, that in each element, i is the player of importance. To understand this, consider (2,3,k) and (3,2,k). They both include information about game(s) played between player 2 and 3. However, (2,3,k) tells information that is more useful for player 2, such as whether player 3 cooperated or defect last time, or how much points he received in the round.

And k, the last dimension, has only 3 values. The first, (i,j,0), denotes the past action of player j against i. It is 1 if the past action was cooperation, and 0 if it was defect. The second, (i,j,1), tells how much points i received in this single round. The third, (i,j,2), tells the total amount of points i received in this generation.

Another noteworthy aspect is that 0th column and row represent the total sum for each player. (i,0,k) represent the sum of all the outcomes for player i, while (0,j,k) represent the sum of all the outcomes for players *who played against player j*.

For better understanding, refer to the following tables. (note that the diagonal is always 0, for one never plays oneself)

k=0	**column**	**0**	**1**	**2**	**3**
row		Number of cooperations the players received			
0	Number of coops the players gave out	#of rounds		A	
1		B	0		
2			C	0	
3					0

3.3a. Illustration of the 3D array for cases where k = 0. *This portion of the array records the history of cooperations in the gameplay process.*

In 3.3a, A represents the total number of cooperations player 2 gave out. That is, total number of cooperations other players received from player 2.

B represents the total number of cooperations player 1 received. It equals total number of cooperations other players gave to player 1.

C represents if player 2 received cooperation or defect from player 1 in the past game.

k=1	**column**	**0**	**1**	**2**	**3**
row		The total points other players gained from playing this particular player in a single round			
0	Total points player received in single round	#of rounds		A	
1		B	0		
2			C	0	
3					0

3.3b. Illustration of the 3D array for cases where k = 1. *This portion of the array records the interactions between pairs of players in the last round of gameplay.*

In 3.3b, A represents the total of points other players received from playing player 2 in this single round.

B represents the total of points player 1 received from playing other players.

C represents the point player 2 received from playing player 1.

The "point" one receives from the game is calculated depending on the classic payoff matrix shown in Diagram 3.3c.

	Defect	Cooperate
Defect	0	2
Cooperate	-1	1

3.3c. The payoff matrix of games between players. *The matrix should be read this way: if i represents the row number and j represents the column number, Entry (i,j) represents the payoff the player gets when it adopts Strategy i and its opponent adopts Strategy j.*

Diagram 3.3c demonstrates the payoff a player gets using different strategies. For example, if player 2 defects and player 1 cooperates, player 2 receives 2 points and player 1 loses a point.

<table>
<tr><td>k=2</td><td>column</td><td>0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>row</td><td></td><td colspan="4">The total points other players gained from playing this particular player overall, from beginning until the most recent game</td></tr>
<tr><td>0</td><td rowspan="4">Total points player received overall</td><td>#of rounds</td><td></td><td>A</td><td></td></tr>
<tr><td>1</td><td>B</td><td>0</td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td>0</td><td>C</td></tr>
<tr><td>3</td><td></td><td></td><td></td><td>0</td></tr>
</table>

3.3d. Illustration of the 3D array for cases where k = 2. *This portion of the array records the total benefit/fitness level each player received from other players according to the gameplay.*

In 3.3d, A represents the total points other players received from playing player 2, from beginning to the current moment in the generation.

B represents the total points player 1 received, from beginning to the current moment.

C represents the total points player 2 received from player 3, from beginning to the current moment.

-players[] : Another noteworthy data structure is the array of players. It keeps track of the number of players we have and maintains a pointer to each player. Note that the matrix and this array is not directly linked in any form. Every time we update the array of players, we change the values of the matrix to match the change. In this manner, we manage and control all the players while keeping the result values in the matrix.

Now, we take a closer look at each of the functions :

- Constructors : We have three constructors, one default, one with an integer parameter, and one copy constructor. The default constructor sets up the matrix, creating the game. The one with parameter also calls on the default constructor, but checks the value that is being passed. The copy constructor copies all the information from the matrix and the array from the other object.
- Destructor : Note we have two dynamic structures to delete. We delete the three dimensional array, starting from the third dimension up to the first. We also delete the array of players.
- addTFTPlayer, addCOOPlayer, addDEFPlayer, addREGPlayer, addPlayer : add specified player to the game. This then calls the next function :
- chgMatrixForNewPlayer() : changes the matrix to match the number of updated players. Note that we are not really changing the size of the matrix. It is too much information to copy every round : hence we simply show the amount of matrix that are being used
- playRound() : plays a round of a game. It first resets the k=1 values of state[][][] as 0 to keep track of total of this round. Then for every pair of players, it calls playPair() to simulate each game.
- playPair() : plays prisoner's dilemma between two players. It first calls getLastPlay() so that each player can get the opponent's last action against himself, and make its decision depending on the given information and one's own traits. After each player makes a move, determines the outcome and updates the state matrix. Depending on the option, determineTeam() might be called.
- getLastPlay() : returns the last play of the game by looking up the corresponding value from state[][][]. This matrix is updated by playPair every time so that getLastPlay can always access the last play from the matrix.
- printMatrix(),printTraits() : prints out the state matrix and the traits of the players. Depending on the option, they can also print out to the log files.
- genChange() : Changes the generation by implementing Fitness proportionate selection. That is, parents with better fitness (or higher points in this case) have higher chance of being the parents. Based on such probability, pairs of avatars are chosen and offspring are born, with some resemblance to their parents. Note that number of offsprings born may be 0,1,2. Since there is an upper limit in number of players, the number stops increasing at the maximum level. On the other hand, there is a slight (but it is possible) possibility that the whole players may be extinct
- determineTeam() : depending on user option, players may be able to create teams. There are numerous ways to simulate this, yet we made it so that two players make a decision at the same time. Two players may create a team, one player may join another's team, both players may

quit the team, et cetera. These are determined by random probability as well as factors of reputation, outcome, distance, etc. One may be able to observe players cooperate or defect on each other from even more dynamic level.

We also have some constants that are widely used:

- gameMatrix[2][2], defectCoop, coopDefect, defectDefect, coopCoop : describes the game matrix which determines the outcome of the game
- DEFAULTPLAYERNUM : the default initial number of players
- PLAYERMAXNUM : Maximum number of players
- ROUNDSPERGEN : Number of rounds per generation
- ZEROCHILD, ONECHILD, TWOCHILD : the probability (out of 100) to get respective number of child

3.4 Main

Main class is the main program that the user runs. It incorporates all the functions defined in other classes and provide the interface which user can use to specify the options and run simulations. More detailed description of the program will be provided in the next section.

4. Inputs and Outputs

The user has many options when he/she runs the program. Upon running the program, the user will see a screen that asks for a desired number of players. The number of players is predefined to be between 0 and 100. (This value as well as many other values predefined in the program can be changed in header files, yet the user is not expected to modify such variables.)

Once the user has set the number of players, there are 5 options of players the user can choose from.

- Tit-for-tat : This player has a memfac of 100, and all other factors as 1. It always performs tit-for-tat. That is, it repeats what the opponent had done last time. This strategy, though very simple, turns out to give incredible performance against most other strategies. We can observe such behavior in our program testing as well.

- Cooperative : This player has a basefac of 100, and all other factors as 1. Its base value is 100. This player always cooperates. If this player is simulated with a defective player, this player tends to do very poorly.

- Defective : This player has a basefac of 100, and all other factors as 1. Its base value is 0. This player always defects. If this player is simulated with a cooperative player, this player tends to do very well.

- Regular : This player is simply a regular player, whose traits are randomized. Some players naturally do better than others, and some might resemble other types of players after randomization.

- Custom : This player gets traits which the user defines. Once user decides to have custom players, user is asked to input five variables to define the player : base, basefac, memfac, distfac, and repfac, which were explained in the paper earlier. User can use custom players to test strategies that are not implemented here. An example of a tendency of a player that have performed well was basing its plays on others' reputation.

Having defined the players, the player have to make a choice between two options : 1. Generation 2. Rounds.

1. Generations

This option simulates the game in terms of generations. Individual rounds are not so important, but the change between the generations are : mutations and reproduction. Reproduction basically means new players are created after each "generation." The total population might increase or decrease; in fact, they might even "be extinct" at some point. The players, upon reproduction, also have chances of mutation. They change their behaviors so that they become more unpredictable. User may view the logs to observe what kind of changes happen, and how they affect the outcome.

There are two more options that one could specify running in generation. One is making more detailed log : the user will then be able to view what each round looked like in more detail. Of course, it is almost impossible/pointless to view every single game when the game is run in generations; there are 1000 rounds per generations, and each round consists of numerous games between pairs of players. The second option is allowing collusion between players. Based on certain reasoning, players may form a team, join a team, or leave a team. The probability is calculated depending on numerous factors including players' reputation, distance between each other, past gameplay, and random numbers. Collusion makes several teams to form and compete against each other. It makes some players to behave differently from their usual actions. Such collusion, interestingly enough, rather makes the result of players to average out rather than create greater discrepancies.

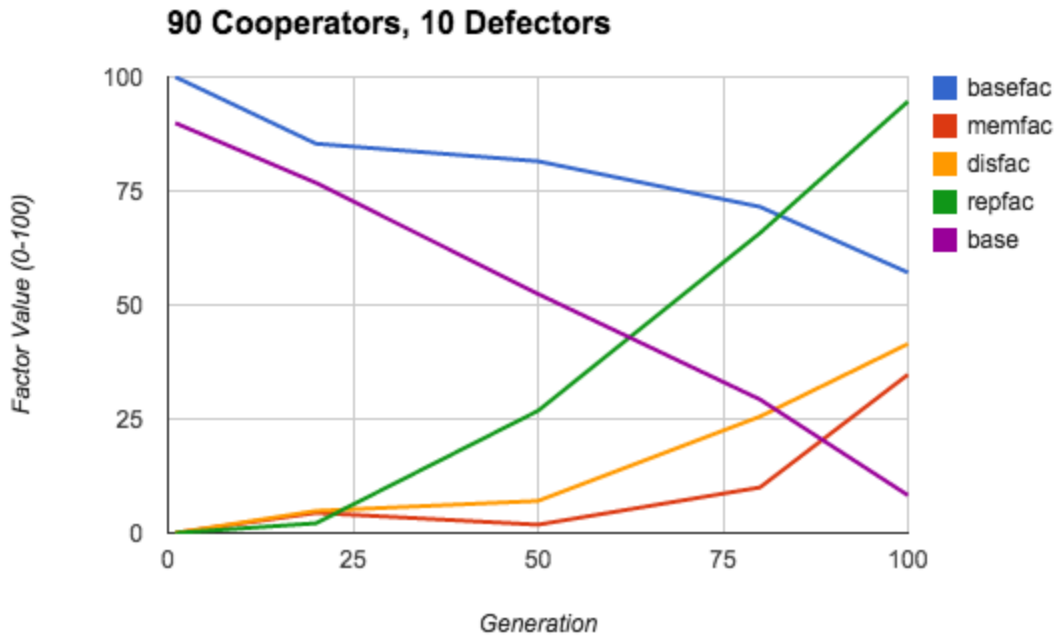
2. Rounds

When the game is run in rounds, the user can focus more on individual gameplays. It is easier to observe how trait affects one's gameplay. Most straightforward example is game between a defector and a cooperator. As they always have the same tendency, the result between them is always the same. Just as in generations, user has two more options. If user chooses to make logs more specific, all the games and the results are printed to the log. Then the user may focus on certain players or games, analyzing any specific outcome as needed. And again, user may allow the players to collude. Players divide into teams, mostly cooperate with teammates, but do unexpected behavior as well, such as suddenly breaking away from the team.

There are three files created as output of the data. Though the user views the results on the console as he runs the program, the information is temporary and limited. Hence, we print a csv file that include outcome matrixes as described before, as well as a document which include gameplays of the past. The user may perform any desired statistics on the excel, as well as make observations from the log file. We provide sample analysis below.

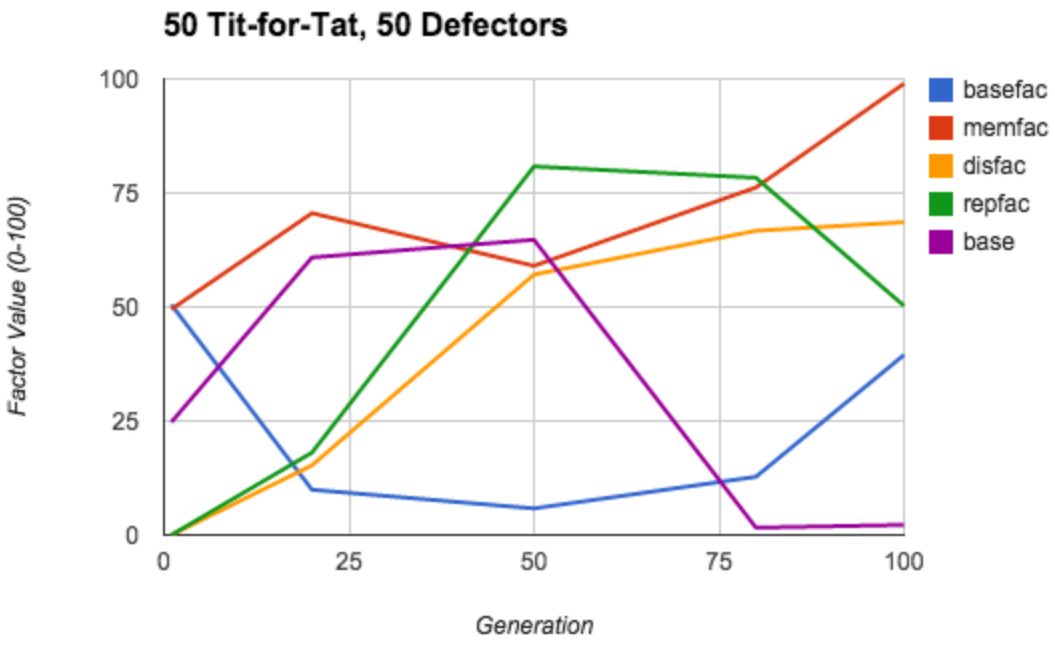
5. Result Analysis

We ran multiple trials with the same initial conditions except for the player composition of the initial population. The various constants we used in these trials can be found in our source code. Below we will display a few statistics produced in these trials.



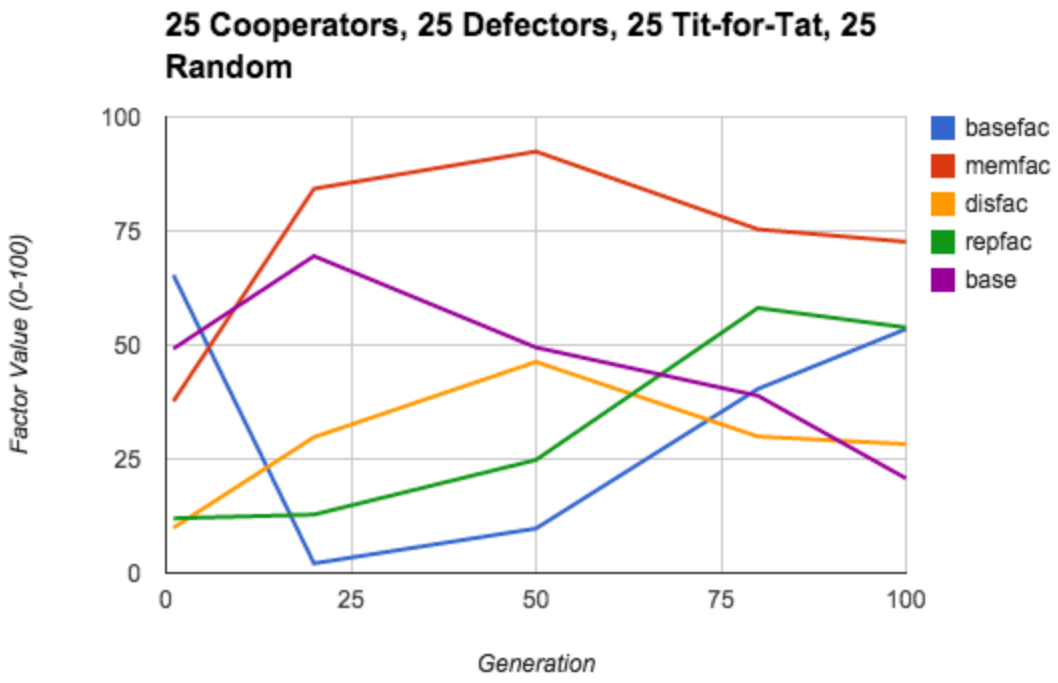
5a. The average values of various factors when the initial population contained 90 cooperators and 10 defectors.

5a demonstrates how the population changes if we start with 90 cooperators and 10 defectors. The value of "base," which is the innate probability that a player cooperates, decreases continuously towards 0. Additionaly, "basefac" remains at a relatively high level (it shows a somewhat decreasing trend as well, which is probably a result of the players adopting more variable strategies). These two facts together means that cooperators are dying out and defectors are becoming dominant in the population, which is consistent with the predictions of classic IPDs[1]. Another notable fact is that we see a significant increase in "repfac." We suspect that this increase is the result of the reputation strategy "hitchhiking" the dominance of defectors. Since defection seems to be the dominant strategy in this population, most players have very low reputation levels, which means that players adopting the reputation strategy will behave just like a defector and this strategy will gain similar popularity. Lastly, we see that at the end of 100 generations, "distfac" and "memfac" starts to rise, as players are probably finding new strategies that can help them combat the dominance of defectors.



5b. The average values of various factors when the initial population contained 50 tit-for-tat players and 50 defectors.

5b demonstrates how the population changes if we start with 90 cooperators and 10 defectors. "memfac" shows an increasing trend, meaning that the "Tit-for-Tat" strategy is doing very well in this population, which is consistent with the previous notion that "Tit-for-Tat" is a great strategy against defectors. Another interesting fact is that "distfac" and "repfac" is also gaining a lot of popularity in this population, which means that as defectors die out, people are more and more willing to cooperate with each other based on other attributes such as social relationship and the other's reputation. Nevertheless, after Generation 80, we see that defectors are starting to grow in numbers again. This is likely because defectors tend to do very well when there are a lot of cooperative players.



5c. The average values of various factors when the initial population contained 25 cooperators, 25 defectors, 25 tit-for-tat players, and 25 players with random strategies.

5c demonstrates how the population changes if we start with a population of players with mixed strategies. First, "memfac" shows that the Tit-for-Tat strategy generally works very well even among a myriad of other strategies. Next, we observe that cooperators and defectors both die out very quickly in the initial rounds, but after a while, defectors start to prosper again, which is similar to what is happening in 5b. Reputation and social distance also gains some popularity as the game progresses, but remain subordinate strategies compared to Tit-for-Tat. This suggests that in some ways, these strategies might offer improvements to Tit-for-Tat.

Conclusions and Future Steps

First, we would definitely like to try out more diverse initial conditions with larger generation numbers and explore the complex patterns that emerge in these different situations. Our attempt to model the interactions in the world using only few variables doubtlessly limits our portrayal of it. Moreover, slight change in such variables causes significant effects, so that it was important to find the right numbers that made sense. The simulation will be more trustworthy if we could choose our numbers depending on some real-world data.

We also hope to improve our current group behavior design in the model. In the current design, a player is admitted into a group as long as it "gets along well" with one of the group members. Nevertheless, in more realistic situations, an individual usually needs to be approved by a large portion of the existing group members before it is accepted by the group. Thus, we need to come up with a better algorithm to model the group dynamics.

Moreover, we only tested the simulation with only up to 100 players, which is a relatively small number. To get a better depiction of the world, we will need to implement more efficient data structures and algorithms in terms of space and time. An idea would be creating characteristic vectors rather than classes or structs of traits.

It was clear that OOP was adequate for simulating a system with autonomous agents and hierarchical structure. We gathered interesting information from the simulation and the Object-Oriented design of this system allows easy integration of more complex features into this system to study more dynamic IPD conditions.