The L.A.M.A card game

Author: Pablo Romero Marimon[*]
Eindhoven University of Technology, Department of Applied Mathematics.
(Dated: February 21, 2022)

Abstract: In this report we describe an implementation of the L.A.M.A card game that uses an object oriented approach. We describe and implement two given strategies. We analyze the outcomes of the game when the players use these strategies, showing that the first one is better. Then, we design a more sophisticated strategy which includes the action of folding (not considered in the previous two). Finally, we show that this third strategy performs better than the other two.

I. INTRODUCTION

Stochastic simulations are a key tool to get a deeper insight into complex processes where randomness plays a role. From the chaotic diffusion of microscopic particles to the study with stochastic numerical simulations. Moreover, many real-life processes are too complex to be posed in an analytical framework, making stochastic simulations not only a useful tool, but also the only the possible way to study some systems.

In this report we use stochastic simulations to study the card game L.A.M.A. Although at first sight it seems apparently simple, this game poses multiple questions that are very hard to address from a rigorous and analytical point of view. Among others, it is not trivial which is the role of randomness in this game. Therefore, stochastic simulations will be vital in understanding how our intuition describes the game's outcomes and to study to which extent randomness plays a role in determining the winner. However, since we will compute all the probabilities as the number of times a certain event takes place over the total amount of trials, these values are not going to be exact. Instead, we can only be sure (with a certain probability) that these values lay within a certain interval, the so-called confidence intervals. Of course, this approach only makes sense when we consider a very large set of samples.

Here, we implement the game and all actions that can be taken and different strategies that can be followed to attempt to win. In the results part, we enlarge on evaluating all strategies to see which one works better.

However, we want to stress that this report does not contain a single mathematical proof. Instead, we just use computational simulations to get an idea of how the game behaves in terms of probabilities. To carry out a Computer Assisted Proof (CAP), among many others, we should implement the code from an intervalar arithmetic approach, which is way beyond the scope of this project.

In the following pages, we first describe how we have implemented the game using an object-oriented approach. Then, we describe the different strategies that we have implemented and analyse them, in order to see which one is more likely to win a game. Finally, we design a better-performing strategy that includes the action of folding, avoided in the other two strategies. The results are shown in the report in terms of probabilities and we indicate the number of runs that we have considered to obtain them. For visual purposes, we use plots instead of tables to present the results. Nevertheless, the reader can find at the end of this report the detailed values with the corresponding confidence intervals for 4 different scenarios. Together with this report, we provide a directory with all the codes and documentation needed to carry out simulations of the game and to obtain the statistical results.

II. GAME IMPLEMENTATION

In this section we explain how we have implemented the code to play the L.A.M.A card game in Python. As anticipated in the introduction, we have used an object oriented approach. This means, that the first step is to decide which classes we are going to use and how we want to structure them. Of course, there is not a unique way to do this, so in the following lines we describe the way that seemed most logical and convenient for us.

First of all, we note that there are some objects that we need to describe, namely the deck, the discard pile and the cards of each player. Probably, the most intuitive at first sight is to define a class to describe each of these objects. However, we followed a different strategy, which only uses two classes.

The first one is the class `Table`, which describes the cards that during the game are laying on the table and all the possible interactions with these cards. The table includes both the deck and the discard pile. The second one is the class `Hands`, which describes the cards that the players have in their hands and all their possible actions. The motivation behind this choice is that a `Table` object is completely independent of a `Hands` object. Therefore, there are no crossed references between the two classes used, which makes the implementation more clear. On the other hand, notice that a `Hands` object needs a `Table` object as an input, as this class describes the actions that the players can take, which clearly involves the deck and the discard pile.

Finally, apart from the aforementioned classes and

*Electronic address: p.romero.marimon@student.tue.nl

their corresponding methods, we implemented two functions, `nextMove` and `playGame`, to be able to simulate a full game. The first computes the next movement of a given player according to a certain strategy and the second carries out different turn plays in a `while` loop to play a full game.

Let us now dive into the classes that we used. For a thorough description of all the methods, refer to the comments made on the source codes.

A. The Table class

As mentioned above, this class describes the deck and the discard pile. The main motivation of including both in the same class was the fact that they are interconnected: when the deck runs out of cards, we need to recall the pile to know which cards we need to shuffle. Also, a class exclusively for the pile seemed unnecessary.

The variables of a `Table` object are the number of cards, the deck and the pile. The first one is a fixed integer, equal to 56, while the others are arrays. Although the pile has initially only one card, it is convenient to define it as a single-element array to make sure that `len(table.pile)` is always well defined.

The constructor of a `Table` object should shuffle all the cards, put one card in the pile and all the others in the deck. However, since this has to be done in the beginning of each round, it is convenient to define a method to do this, which we call `restartTable`. Hence, the constructor simply calls this method.

All the methods of the class `Table` and they purposes are explained here:

1. `cardsOnDeck()`: Checks whether or not there are cards left on the deck.

2. `shuffleDeck()`: Shuffles the cards on the discard pile (all but the one in the top) and put them back in the deck. This method is called when `cardsOnDeck()` returns `False`.

3. `restartTable()`: Sets the initial configuration of the cards in the table. This is applied in the beginning of each round.

4. `removeFromDeck()`: This method removes the first card from the deck. This is done when a player decides to draw a card.

5. `addToPile(card)`: This method puts a given card (a number from 0 to 6 where 0 is the LAMA card) in the top of the discard pile.

B. The Hands class

Here we describe the class we used to describe the actions involving the cards of the players. Firstly, we remind the reader that the constructor of a `Hands` object receives a `Table` object as an input. Also, the constructor receives the number of players as an input.

Now, before explaining the methods defined in this class, let us state its variables. First of all, we define the variables `NPLAYERS` and `CARDSXPLAYER`, which are useful constants.

Then, the constructor creates the variables `points` and `rounds`, which are arrays having the size equal to the number of players that count the number of points of each player and the number of rounds they have won respectively. Afterwards, the constructor calls the method `restartHands(table)`, which creates a variable to store the cards in the players' hands. This variable is called `cards`, and it is defined as a list of arrays (one array for each player). Then, the cards of the player i are the elements in the array `cards[i]`. This allows for the players to have a different number of cards each one, which could not be done if we had defined the cards as a 2-dimensional array. In languages like C or C++, an array is nothing more than a vector which has vectors as elements, so we can have a different number of elements in each row. However, this does not work in Python, and therefore we used a list of arrays.

Initially, `cards` is defined as a list of 9 elements (which is the maximum number of players, since there are not enough cards for 10 players). If the current number of players is lower, some elements of the list are not used at all.

To manage the turns we defined two variables: an array `turns`, which is defined at the beginning of each round, and an integer `index`. In `turns` we store the players that are currently playing in natural order (first Player 1, second Player 2 and so on). `index` is currently set equal to 0, because the first player always start. After a player has moved, `index` is either increased by 1 or set to 0 if the next player is the first one again. This way, the next player to play is given by `turns[index]`.

Notice that if the players never fold the array `turns` is not necessary. In that case, the integer `index` already gives us the player who plays next. However, if they are able to fold, the cycle of players' turns may change during a round. Hence, we need the variable `turns` to keep track of the players still laying. Then, if a player folds, we just need to remove its corresponding element in the `turns` array to make sure that it will not be his turn again in this round.

To deal with the possible moves that a player can make, we use integers. The action of discarding a card is represented by a number from 0 to 6, depending on the card to be discarded. Then, there are two other moves that all players can do at any time: fold (represented with a -2) and draw a card from the deck (represented with a -1). Thus, all the possible moves are represented by integers from -2 to 6.

Finally, we use two variables to describe the state of the game: it has either ended (`FINISHED`) or not (`CONTINUE`).

The methods of this class are stated in the following lines. In the following let table be an object of the class

`Table`.

1. `restartHands(table)`: Gives 6 cards from the deck to each player and restart the turns of the players. This is applied every time a new round starts.

2. `getPlayerTurn()`: Returns the player turn's.

3. `getPossibleMoves(table, player)`: This function uses the top card on the pile and cards on the `player`'s hand to obtain the possible moves of a certain player. It returns an array with 2, 3 or 4 elements. If the player cannot discard any cards, this array is $[-2, -1]$ corresponding to fold and draw a card from deck. If the player can discard a card, this is added to the array. Note that if a player can discard a certain card and he has multiple copies of this card, only one is added to the array. Hence, the output array will not contain more than 4 elements.

4. `makeMove(table, player, move)`: Given the player that goes next and a certain move, carries out the move. This method calls the methods `addToPile()` and `removeFromDeck()` of the `Table` class.

5. `evaluate(table)`: Checks whether the round has ended or not. Note that the round ends in two different scenarios: If all the players but one folded, or if a player runs out of cards. If the round has ended here the `endRound` method is called.

6. `endRound(table, winner)`: This method is called after a round has ended. Here the points are added and it is checked if another round needs to be played.

C. Validation

In order to validate the program we have carried out an extensive number of tests. First, we have tested every function and method separately, to make sure that we got the desired result. Then, we have also played several games by making every single moves to test that all the functions were properly connected. We investigated particular cases that could lead to problems, such as the case where more than one player wins the game, the case where all the players but one fold, etc. In addition, each of the authors conducted several tests separately to avoid mutual bias in the testing. Some examples of the tests that we did can be found in the documentation.

Finally, it is worth to mention that the game needs at least 2 players and can be played by 9 at the maximum. This is because there are not enough cards to be distributed among 10 players so that all of them start with 6 cards. However, one may notice that playing games with 9 players eventually can lead to an error. The reason why this happens is that when a 9-players game starts, there is only one card in the deck. Therefore, if the two first players, for instance, decide to draw a card from the deck, the second cannot complete this action. Then, the program initially led to an error, since the game cannot be continued. Now, when this happens we send the following thread to the terminal:

```
Game cannot continue: not enough cards on
the deck
```

and we stop the program. To avoid this situation, we will consider games with maximum 8 players from hereafter.

III. FIRST STRATEGY

The first strategy that we consider consists in never folding and discarding cards whenever possible. Also, when we can discard two different cards, we discard always the one that is equal to the one in the top of the discard pile.

This is implemented in the function `nextMove`, and it is done as follows. Given a player and a game situation, we compute the possible moves. If the possible moves is an array containing only two elements (meaning that no cards can be discarded) we draw a card from the deck. If not, the we look for a card equal to the one of the top of the pile in the player's hand. If we find one, we discard this card, and if we do not, we discard the only other card that can be discarded.

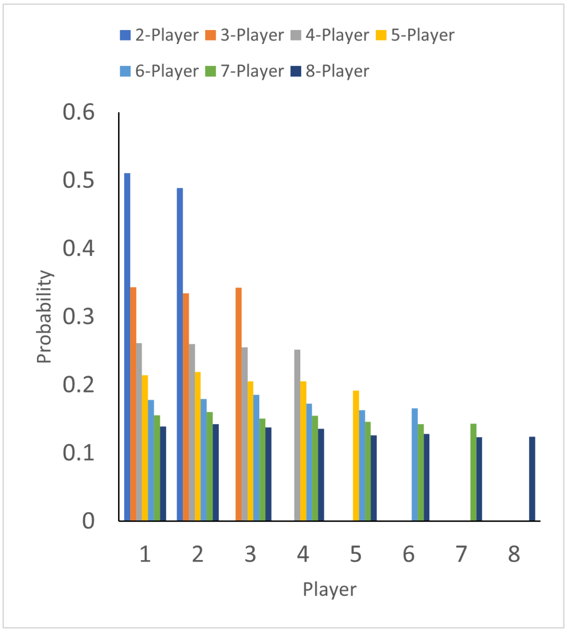

FIG. 1: Probability for each player to win the game depending on the number of players when all play according to the first strategy. The x axis corresponds to the index of each player.

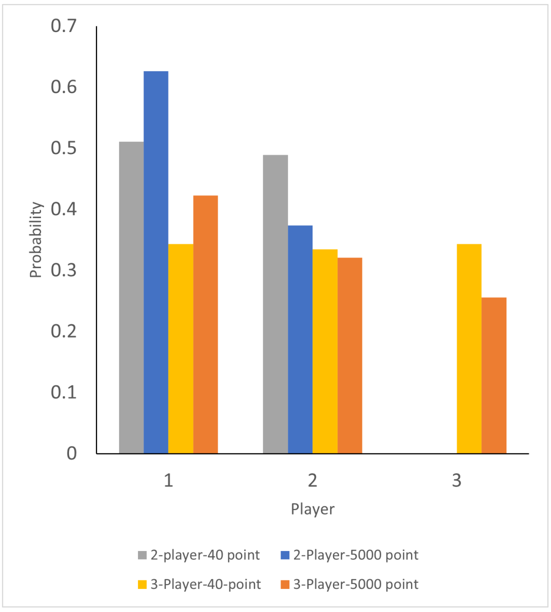

FIG. 2: Probability to win as a function of the number of players for two different cases: (i) Case where the game ends when the first player reaches 40 points. (ii) Case where the game ends when the first player reaches 5000 points. The x axis corresponds to the index of each player.

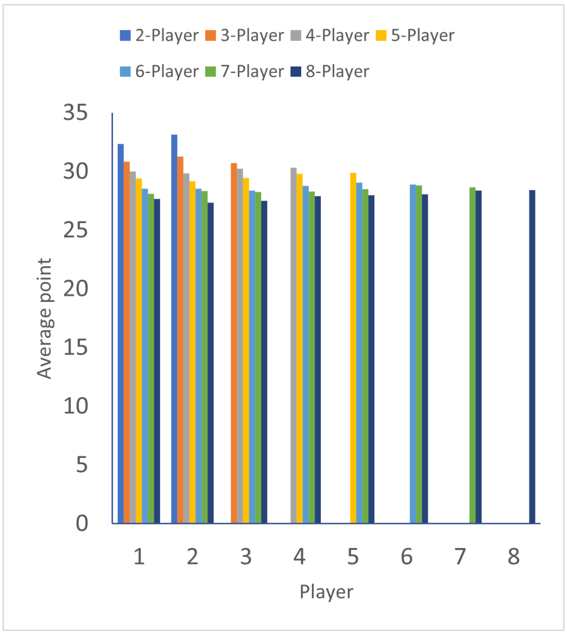

FIG. 3: Average number of points of each player depending on the number of players when all play according to the first strategy. The x axis corresponds to the index of each player.

A. Results

Here we analyse the outcomes of the game if all the players follow the first strategy. Therefore, the main goal of this section is to discuss whether some players are more likely to win the game depending on when they start playing (player 1 goes first, player 2 goes second and so on). In order to address this question, we considered different scenarios, changing the number of players from $n = 2$ to $n = 8$, and for each one we carried out 10000 games. The outcomes of these games can be translated into probabilities. We show in Fig. 1 the probability that each player has to win as a function of the number of players. Also, see Tables I and II for a more detailed analysis for the cases $n = 4$ and $n = 5$ respectively.

In Fig. 1 we first see that for each player the probability to win decreases with n, as one may expect. We can also see in Tables I and II that the probabilities of winning a game for all players does not sum to one. This is simply because a single game can be won by more than one player. We also see that all the players have a similar probability to win the game. However, it seems that the players who start playing first are more likely to win, which suggests that the outcomes of the game depend on the order of the turns.

Before diving deeper into this question, we would like to remark that this is a celebrated and non trivial question in game theory in general. Just as an example, it is thought that in chess the whites could have an advantage over the blacks because they start playing, but the mathematical complexity of the game makes it very hard to address this in terms of probabilities.

However, if we analyse this simpler game, it makes sense that the first players have an advantage with respect to the others. This is because, provided that all players follow the same strategy, in average all need the same amount of turns to get rid of their cards. Hence, being the first to play makes it easier for you to discard all your cards first.

Nevertheless, one can argue that this behaviour is not very clear in Fig. 1 and Tables I and II. This is because this only supposes a very little advantage every time that a new round starts (since then the player 1 is first again) and in average only 3 rounds are needed to end a game. Thus, in order to see this behaviour better reflected in our results, we could increase the average number of runs needed to win a game. This can be easily done by setting that the game ends when the first player reaches 5000 points, instead of 40.

The results of this test are shown in Fig. 2. There, we show the results for the cases $n = 2$ and $n = 3$ when the game ends when the first player reaches 40 points, and when it ends when the first reaches 5000 points. We can clearly see that in the second case the winning probability highly depends on which player goes first, as we expected. We want to remark that the data obtained for the second case has been obtained from the statistical results of a total of 1000 games, instead than 10000. This is because these new games are much longer, meaning that the exponential time increased dramatically if we considered the same amount of games as before.

Fig. 3 shows the average number of points that each player gets after a game. Accordingly to the fact that

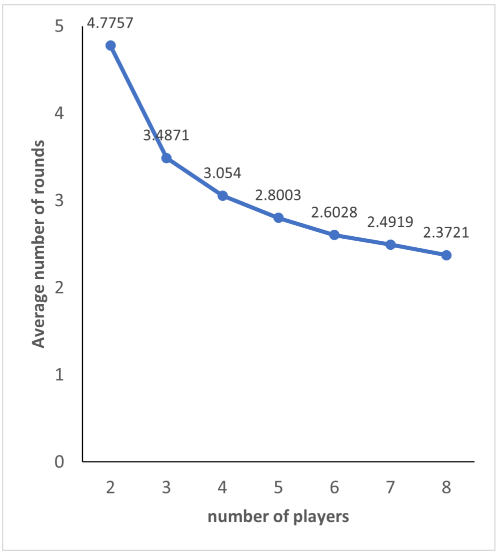

FIG. 4: Average number of rounds of a game as a function of the number of players involved when all play according to the first strategy.

the first players are more likely to win, we see there that to first players end the game with slightly less points in average. Of course, this behaviour would be more visible is we increase the number of points needed to finish the game (i.e. increase the number of rounds).

Finally, Fig. 4 shows the average number of rounds that needs to be played for a game to finish, as a function of the number of players. We see a decreasing behaviour, which can be explained by the fact that when more players are involved in a game, it is more likely that some of them ends the round with a bad result (high number of points).

IV. SECOND STRATEGY

The second strategy is very similar to the first one, the only difference being that in this when we are able to discard two different cards, we discard the one that is different from the one in the top of the discard pile.

Again, this is implemented in the function `nextMove`, and the implementation is quite similar to the one of the first strategy. Given a player and a game situation, we compute the possible moves. If the possible moves is an array containing only two elements (meaning that no cards can be discarded) we draw a card from the deck. If not, we look for the card that goes next to the one in the top of the discard pile in the player's hand. If we find it, we discard this card, and if we do not, we discard the card equal to the one in the top of the discard pile.

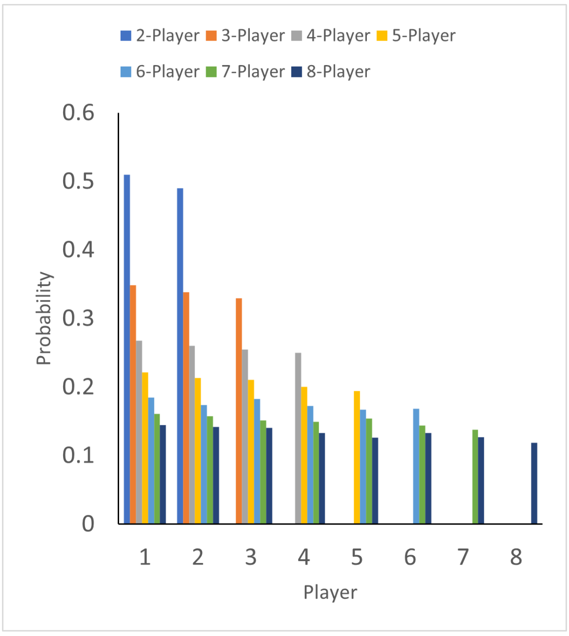

FIG. 5: Probability for each player to win the game depending on the number of players when all play according to the second strategy. The x axis corresponds to the index of each player.

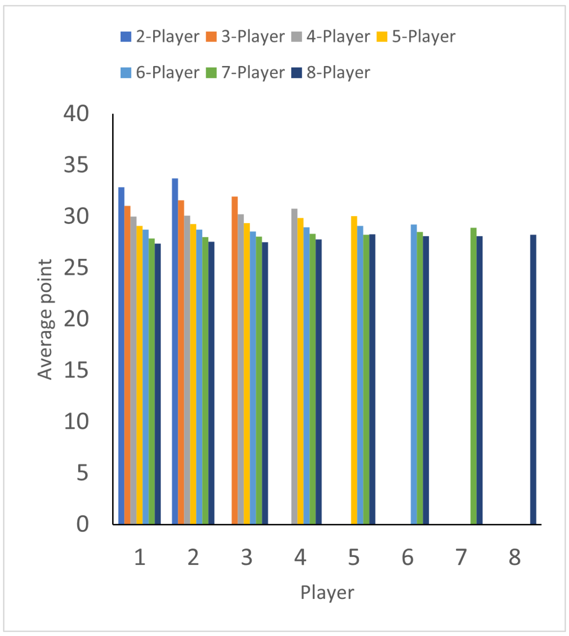

FIG. 6: Average number of points of each player depending on the number of players when all play according to the second strategy. The x axis corresponds to the index of each player.

A. Results

Here we proceed as in Sec. III A to analyse the results of the outcomes of a game where all the players follow the second strategy. Again, we considered different scenarios changing the number of players (from $n = 2$ to $n = 8$)

and for each one we carried out 10000 games. The results, in terms of probabilities, are shown in detail in Tables III and IV for the cases $n = 4$ and $n = 5$ respectively.

Fig. 5 shows the probability of each player to win when they all follow the second strategy. We notice that the results are quite similar to those corresponding to the first strategy: the probability to win decreases with the number of players and the first players seem to have an advantage with respect to the others, for the same reason we explained in Sec. III A.

Similarly, this results are also reflected in Fig. 6, where the average number of points is in general higher for the latest players, meaning that they in general lose more games.

Finally, Fig. 7 shows the number of rounds that need to be played in average for a game to finish. Similarly as in Fig. 3, that whenever more players are involved in the game, the amount of rounds needed decreases. This happens due to the same reason that we explained in Sec. III A. Nevertheless, we see that the number of rounds when $n = 2$ is considerably different in the two cases. Indeed, when the two players follow the first strategy, they need more than one round in average to finish the game. This means that the player that losses obtains more points in average when they are following the second strategy.

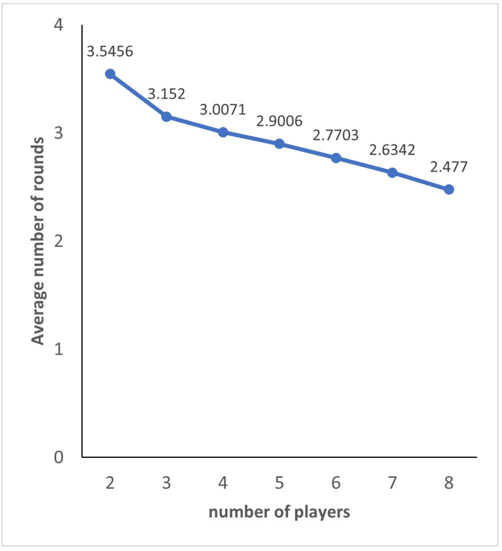

FIG. 7: Average number of rounds of a game as a function of the number of players involved when all play according to the second strategy.

V. COMPARISON BETWEEN STRATEGIES 1 AND 2

First of all, we can argue without the use of simulations that when $n = 2$ the first strategy is better than the second. The reason is that if only two players are playing, when you are able to discard two different cards, you always will be able to discard the other one if you now discard the one that it is equal that the one in the top of the pile. Instead, if you first discard following the second strategy, it is impossible that in the next turn you can discard the one that it is equal to the one in the pile.

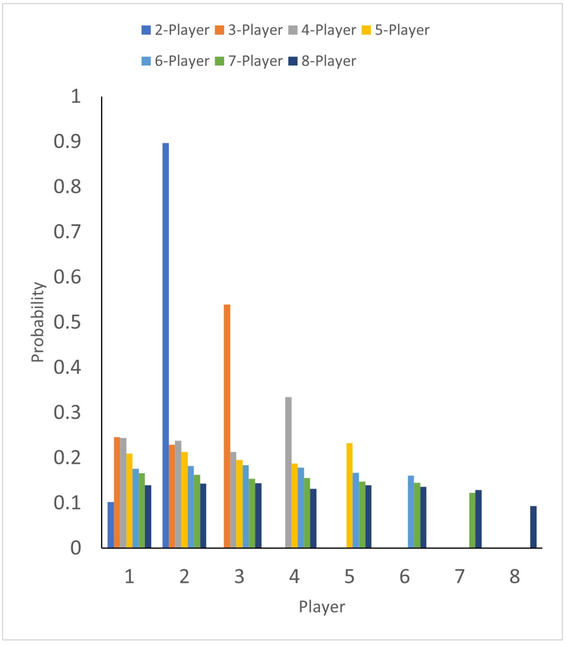

FIG. 8: Probabilities to win as a function of the number of players. In each case, the first $n - 1$ players play according to the second strategy and the last one follows the first one. The x axis corresponds to the index of each player.

Based on the same idea, we can argue that for a small number of players the first strategy is better than the second, since if you throw the same card as the one in the pile when you can chose, the chances that you can play the other card in your next turn are high. However, when the number of players is big, the two strategies tend to lead more similar outcomes. This is because when n increases it becomes harder to predict which card will there be in the pile the next turn.

To test these hypothesis, we considered several games where $n-1$ players followed the second strategy and the other followed the first one. Moreover, we considered that the one that follows the first strategy starts the last one, to not get results biased by the fact that the first ones has some advantage. The results are shown in Fig. 8.

In this figure, we see that when $n \in [2, 5]$ the last player (which follows the first strategy) is more likely to win. Then, when increasing the number of players further, the last player loses a slightly greater amount of times. As we argued, this is because the choice of the strategy now loses importance and starting the last one puts the player following the first strategy in a disadvantageous position.

VI. THIRD STRATEGY: INCLUDING FOLDING

In this section we present a better performing strategy that contemplates the option of folding, which until now has not play any role. The main goal of this section is not to define the best possible strategy to win the game, but a strategy that is sufficiently good to be qualified as better than the other two that we introduced.

From now on, we will restrict to games with 4 players, when 3 players follow the first strategy (which performs better than the second one, as seen in Sec. V) and one player follows the new strategy. In order to get rid off the bias induced by the order of the turns, we consider random orders in each game.

First of all, it seems very reasonable that whenever a player can discard a card, the best option is to do it. The next question that rises is what happens when more than one card can be discarded. What we found, is that this depends on the number of players. If the number of players is low, then it is better to discard according to the first strategy, because the chances of being able to discard the other card in the next round are high. However, when the number of players is high (8-9) it starts to become better to discard the card that corresponds to more points (note that this is not exactly the second strategy). This is because when the number of players increase, the card in the discard pile between two turns of the same players approaches a uniform random distribution among all possible cards. In short, when the number of players increase, it is more likely that if you first discard the card equal to the one on the pile, you cannot discard the other one in the next turn.

However, since we are considering games with just 4 players, it is advisable to follow the first strategy when more than one card can be discarded.

Now, let us look into the case where no cards can be discarded. Then, the player must decide whether to fold or draw a card and keep playing. To do so, a lot of different inputs could be taken into account. For simplicity, we considered just three factors, which we enumerate:

1. Amount of different cards the player has: the higher the amount of different numbers that the player has in his hands, the more likely it is that he will be able to play in the next turn. Therefore, the player should be tempted to play when he has many different cards and tempted to fold when this number is low (like 1 or 2).

2. Number of points the player has at the moment: the player takes into account the number of points that he would get if he folds. Besides, we will apply a conservative strategy (if the player has only a small amount of points he will be tempted to fold, and if he has a lot of points he will be tempted to keep on playing to try to revert the situation).

3. Expected time for the round to finish: the player takes into account the time that it may take for the round to finish. This can be computed by looking at how many players are still playing and how many cards do they have. Furthermore, since the other 3 players will be playing according to the first strategy, they will never fold, so we do not need to keep track of the amount of players still playing this round. Then, if the round is expected to finish soon, the player should be tempted to fold, since drawing a card from the deck momentarily increases the amount of points.

So far we have introduce the factors on which we will base the decision of folding or not, and how we can quantify them. Now, we can define a very simple linear function that takes all of them into account. To this end, let us define x as the number of different cards the player has in his hands, y as the number of points the player has in his hands and z as the number of cards of the player that has the least amount of cards. Then, we can define f as

$$f(x, y, z) = g(x) + (y - C) + Dz, \tag{1}$$

where

$$g(x) = \begin{cases} -A & \text{if } x = 1 \\ Bx & \text{if } x > 1 \end{cases} \tag{2}$$

being A, B, C and D parameters that need to be adjusted. Note that the terms in f are positive when they should tempt the player to keep on playing and they are negative when they should tempt the player to fold, according to the criteria we described above. Thus, we can decide that the player keeps on playing when $f(x, y, z) > 0$ and he folds otherwise.

Naturally, we still need to adjust the parameters. To do this we proceed as follows. We first consider a trial set $\mathcal{P}_{A,B,C,D} = (10, 5, 4.4, 1)$ with parameters that we find somehow reasonable. We now define the configurations space of these parameters as a subset of $\mathbb{R}^4$ containing $\mathcal{P}_{A,B,C,D}$, namely

$$\mathcal{S} = [5, 20] \times [1, 15] \times [2, 7] \times [0.2, 2] \subset \mathbb{R}^4. \tag{3}$$

Finally, in order to sample the configurations space of the parameters to find ones that leads to a better performing strategy, we apply a Monte Carlo based approach for the sampling. Then, we consider a `for` loop of 1000 steps. In each step we randomly select a point in $\mathcal{S}$ using a uniform distribution and perform 1000 games where one of the players follows the third strategy with these given parameters. After all the games we analyse the results, and if these are better that the ones we had so far we keep this configuration of parameters.

We want to remark that this is a very simple Monte Carlo implementation. Normally, the accepting criteria contemplates the acceptance of configurations that are worst than the one that we have so far to sample better the space of configurations and not get stacked in a local minimum. Also, normally not all the parameters are changed in every step. However, this implementation is only to illustrate a potential use of the Monte Carlo algorithms, which are widely used in physics. Here, we can understand that we use it to sample a big number of configurations without having to try them one by one.

A. Results

In this section we present the results for the performance of the third strategy. We anticipate that this strategy is consistently better than the ones that we introduced in previous sections. Nevertheless, we found an important flaw in the Monte Carlo approach that we followed, which means that the final parameters we present here for the third strategy are not properly optimized.

The main issue we have observed is that since the outcomes of the games are stochastic, given a set of parameters, we do not get exact values for the winning probabilities. Normally, we overcome this problem by giving as well a 95% confidence interval. However, if we carry out a 1000 steps Monte Carlo algorithm, it is likely that in some cases, the winning probability for the player playing according to the third strategy is higher than the real value. Therefore, when we get a set of parameters that results in a very high probability for this player to win, we cannot say that these parameters will always lead to this winning probability.

In our case, after running the Monte Carlo, we got that the set of parameters that led to a higher average winning probability was $\mathcal{P}_{A,B,C,D} = (17.576, 12.24196, 3.920, 1.222)$, with a resulting winning probability of 0.334 and a confidence interval of $(0.304, 0.363)$. However, if we carry out another simulation with these parameters and increasing the number of games to 10000 in order to narrow the confidence intervals, we find that the probability decreases to 0.302 with confidence interval of $(0.295, 0.318)$.

This shows that the result we obtained in the beginning is probably not the best set of parameters. Instead, this set probably led to a value of the winning probability that was above the upper bound of the confidence interval. This is because when we simulated the performance of these parameters but considering 10 times more games, the winning probability was slightly worse.

Nonetheless, the final result of the winning probability is considerably higher than 0.25, which is the probability that each of the players has to win if all play completely at random. Therefore, we can conclude that this strategy is better than the first strategy we introduced in this report.

We now can proceed similarly to see if whether the strategy that includes folding is better than the second one. Indeed, we obtained that the player following this strategy in this case has a winning probability of 0.352 with a confidence interval of $(0.344, 0.369)$.

Finally, we remark that although the strategies have the transitivity property in terms of performance, that could not be the case (i.e. A better than B and B better than C does not necessarily implies A better than C).

VII. DISCUSSION AND CONCLUSIONS

In this report we have described an implementation of the L.A.M.A card game. We have done this by considering an object oriented approach in Python. Then, we discussed the multiple advantages of considering stochastic simulations to study the outcomes of this game, which involves randomness.

Firstly, we introduced two different strategies that did not consider all the possible actions that can be taken in the game. Namely, these strategies could be perfectly implemented without implementing the action of folding in the game implementation.

By considering a very large amount of runs, we could draw probabilistic results from our computational simulations. Indeed, we used this approach to study the possible outcomes we can expect when using each structure.

Actually, we focused part of the discussion in a very celebrated question in the framework of game theory: until which extend does the order of the turns determinates who is more likely to win? From what this concerns, we saw that this is indeed important, and that it becomes more noticeable when the number of rounds of an average game increases. To check this, we reformulated the rules of the game so that it involves more games.

The next step was to design our own strategy. In this case, we included the action of folding as a possibility. Therefore, the main question that we addressed was when it was convenient to fold. To this end, we introduced several variables that needed to be considered and defined a very simple decision function, which had some parameters that needed to be adjusted. Then, we consider a Monte Carlo algorithm to sample the space of configurations of the parameters. We found that the third strategy, the one that includes the possibility of folding, performed quite better than the the other two.

Player Id	P(i wins game)	P(i loses game)	P(i wins round)	P(i gains more than 30 points)	Average points
1	0.262 (0.253, 0.27)	0.738 (0.73, 0.747)	0.279 (0.273, 0.284)	0.49 (0.48, 0.499)	30.015 (29.698, 30.332)
2	0.26 (0.252, 0.269)	0.74 (0.731, 0.748)	0.26 (0.255, 0.266)	0.482 (0.472, 0.492)	29.851 (29.539, 30.163)
3	0.255 (0.247, 0.264)	0.745 (0.736, 0.753)	0.239 (0.232, 0.242)	0.498 (0.488, 0.508)	30.246 (29.933, 30.56)
4	0.252 (0.243, 0.26)	0.748 (0.74, 0.757)	0.223 (0.219, 0.229)	0.505 (0.495, 0.515)	30.32 (30.006, 30.633)

TABLE I: Results for the first strategy when $n = 4$.

Player Id	P(i wins game)	P(i loses game)	P(i wins round)	P(i gains more than 30 points)	Average points
1	0.214 (0.206, 0.222)	0.786 (0.778, 0.794)	0.232 (0.226, 0.237)	0.474 (0.465, 0.484)	29.418 (29.117, 29.72)
2	0.219 (0.211, 0.227)	0.781 (0.773, 0.789)	0.216 (0.209, 0.219)	0.461 (0.451, 0.47)	29.178 (28.879, 29.477)
3	0.206 (0.198, 0.213)	0.794 (0.787, 0.802)	0.202 (0.198, 0.208)	0.468 (0.459, 0.478)	29.43 (29.132, 29.728)
4	0.206 (0.198, 0.214)	0.794 (0.786, 0.802)	0.181 (0.177, 0.186)	0.488 (0.478, 0.497)	29.802 (29.504, 30.099)
5	0.192 (0.184, 0.2)	0.808 (0.8, 0.816)	0.17 (0.165, 0.174)	0.482 (0.473, 0.492)	29.9 (29.599, 30.2)

TABLE II: Results for the first strategy when $n = 5$.

Player Id	P(i wins game)	P(i loses game)	P(i wins round)	P(i gains more than 30 points)	Average points
1	0.268 (0.259, 0.276)	0.732 (0.724, 0.741)	0.275 (0.268, 0.278)	0.497 (0.487, 0.507)	29.982 (29.668, 30.296)
2	0.26 (0.251, 0.268)	0.74 (0.732, 0.749)	0.262 (0.256, 0.267)	0.493 (0.483, 0.503)	30.096 (29.782, 30.411)
3	0.254 (0.246, 0.263)	0.746 (0.737, 0.754)	0.24 (0.236, 0.246)	0.495 (0.486, 0.505)	30.208 (29.894, 30.522)
4	0.25 (0.241, 0.258)	0.75 (0.742, 0.759)	0.222 (0.22, 0.23)	0.514 (0.504, 0.523)	30.771 (30.457, 31.086)

TABLE III: Results for the second strategy when $n = 4$.

Player Id	P(i wins game)	P(i loses game)	P(i wins round)	P(i gains more than 30 points)	Average points
1	0.221 (0.213, 0.229)	0.779 (0.771, 0.787)	0.231 (0.225, 0.235)	0.46 (0.45, 0.47)	29.067 (28.776, 29.359)
2	0.213 (0.205, 0.221)	0.787 (0.779, 0.795)	0.219 (0.214, 0.224)	0.464 (0.454, 0.474)	29.279 (28.984, 29.573)
3	0.211 (0.203, 0.219)	0.789 (0.781, 0.797)	0.199 (0.194, 0.204)	0.468 (0.458, 0.478)	29.368(29.075, 29.66)
4	0.201 (0.193, 0.209)	0.799 (0.791, 0.807)	0.18 (0.176, 0.185)	0.481 (0.471, 0.491)	29.843 (29.551, 30.135)
5	0.194 (0.187, 0.202)	0.806 (0.798, 0.813)	0.171 (0.167, 0.176)	0.481 (0.472, 0.491)	30.037 (29.745, 30.329)

TABLE IV: Results for the second strategy when $n = 5$.