

---

# Final Year Project Report

## AI for Modified Perudo and Game Hosting Service

---



**Maynooth  
University**  
National University  
of Ireland Maynooth

Dheeraj Putta | 15329966

A thesis submitted in partial fulfilment of the requirements for the  
B.Sc. Computational Thinking

Advisor: Dr. Phil Maguire

*Department of Computer Science  
Maynooth University, Ireland*

March 19, 2019

## **ABSTRACT**

The objectives of this project were to create and implement strategies for artificial intelligence agents that could play Perudo with a modified rule set, develop code that is extensible to make it easy to add new strategies, and implement a method that allows the agents to play against each other. The developed agents are compared against each other but with different parameters as to compare and contrast on the effects of each of the different values. After picking the best agent from each round, the best configuration for each agent is compared against all the different agents to try and discover the best agent. Next, I compare the best agent picked against myself playing against that agent. Finally, we discuss possible improvements and other strategies that could be implemented.

# CONTENTS

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Rules . . . . .	1
1.3	Example of a round . . . . .	2
1.4	Approach . . . . .	3
1.5	Metrics . . . . .	4
<b>2</b>	<b>Technical Background</b>	<b>5</b>
2.1	Topic Material . . . . .	5
2.2	Technical Material . . . . .	6
<b>3</b>	<b>Problem Analysis</b>	<b>7</b>
<b>4</b>	<b>The Solution</b>	<b>8</b>
4.1	Analytical Work . . . . .	8
4.2	Server Design . . . . .	9
4.3	Methods . . . . .	11
4.4	Class Structure . . . . .	15
4.5	Implementation . . . . .	15
<b>5</b>	<b>Evaluation</b>	<b>18</b>
5.1	Selecting Parameters . . . . .	18
5.2	Results and Analysis . . . . .	19
<b>6</b>	<b>Conclusions</b>	<b>20</b>
6.1	Project Approach . . . . .	20
6.2	Results Discussion . . . . .	20
6.3	Future Work . . . . .	20
	<b>Bibliography</b>	<b>22</b>
	<b>Appendices</b>	<b>23</b>
<b>A</b>	<b>Appendix A</b>	<b>24</b>
<b>B</b>	<b>Appendix B</b>	<b>25</b>
<b>C</b>	<b>Appendix C</b>	<b>27</b>

# LISTS OF FLOATS

---

## LIST OF TABLES

1.1	Player's Dice . . . . .	3
5.1	prob v.s. bluff for DumbAIPlayer . . . . .	18
5.2	prob v.s. bluff for SLDumbAIPlayer . . . . .	18
5.3	prob v.s. bluff for LDumbAIPlayer . . . . .	18
5.4	Call Accuracy and Number of games won for each AI . . . . .	19
5.5	Call Accuracy, Number of Games won and the Average number of dice left when won . . . . .	19

## LIST OF FIGURES

1.1	Flowchart of a game of Perudo . . . . .	2
1.2	Round being played . . . . .	3
4.1	Order of operation during a game . . . . .	10
4.2	Skewnorm compared to dice probability . . . . .	11
4.3	Minimax Tree . . . . .	12
4.4	Final Tree after performing $\alpha$ - $\beta$ pruning . . . . .	14
4.5	UML Class Diagram of Project . . . . .	16
4.6	Package Structure of Project . . . . .	17

## LIST OF LISTINGS

4.1	Sending Player information . . . . .	9
A.1	Generating Bets . . . . .	24
B.1	Equation (4.2) . . . . .	25
B.2	Equation (4.4) . . . . .	25
B.3	Equation (4.5) . . . . .	25
B.4	Placing a bet . . . . .	26
C.1	Minimax Implementation . . . . .	27
C.2	Minimax with $\alpha$ - $\beta$ pruning . . . . .	28
C.3	Placing a Bet with MiniMax . . . . .	28

# INTRODUCTION

---

Perudo is a popular dice game that is played all over the world. Perudo is a version played in South America, where it is called many different names such as Dudo, which is Spanish for *I doubt*. It is more well known as Liar's Dice due to the game being called that in "Pirates of the Caribbean" movie franchise and the "Red Dead Redemption" video game series.

## 1.1 BACKGROUND

Perudo is an example of a game with imperfect information. That is, at any point when making a decision we do not know all the information on the board. In this respect, it is similar to Poker, in Perudo we have no idea what the dice that our opponents holds are and in Poker we don't know what cards the opponent is holding. This causes difficulties in detecting possible bluffing. Hence, the optimal strategy can only be estimated with some probabilistic method that heavily depends on the opponents moves.

Perudo, however is not as difficult as Poker as it does not have as large a range of possible card combinations as well as different types of hands. From this one could think of Perudo as a simplified poker.

## 1.2 RULES

The majority of these games have a common rule set:

1. Each player starts with 5 dice.
2.  $\square$ 's are wildcards, i.e. they count as being every number.
3. After the previous player bets that there are at least  $n$  dice with a certain value, the current player has to raise the bid by raising either the number of dice, the value of the dice or both. In general you can think of each bet as having a score, defined by:



$$s(n, x) = 10n + x \tag{1.1}$$









For example, If a player bets that there are at least 2  $\square$ 's then it would have a value of 23. The current players bet is valid if the score of their bet is greater than the score of the previous bet.

4. Only the current player may call or raise the bid.
5. If a player calls, then all dice are revealed and the previous bet is checked. If it was a valid bet the player who has called will loose a die and a new round will start. If the bet was not valid then the previous player loses a die. Whoever loses a die will start the next round.

6. A game is finished when there is only one player left with dice.

In Perudo there are 2 special rules that we have to consider.

**PALAFICO** When a player gets to one die, a special turn (called *Palafico*) occurs. Whatever die value that player begins with cannot be changed. For example, if the *Palafico* player bets that there are 2 s, then every subsequent bet must raise that bet by saying that there are  $n$  s. A player only gets one *Palafico* round per game. If they survive that round, play goes back to normal.

**ACES** Instead of the usual bidding, players also have the option of trying to predict how many s there are. In this round, the number of s predicted has to be **at least** half the previous amount. For example if the previous bet was 4  then a bet of 2  is valid. Fractions are always rounded up. After a call of *Aces*, the next player may either raise the quantity of s or they can revert back to the normal method by doubling the quantity of s called. For example, following a bet of 3 s, the next bid has to be at least 4 s or 6 of any other number.

However for this project, we did not include those 2 special rules and only played with the common rule set.

### 1.3 EXAMPLE OF A ROUND

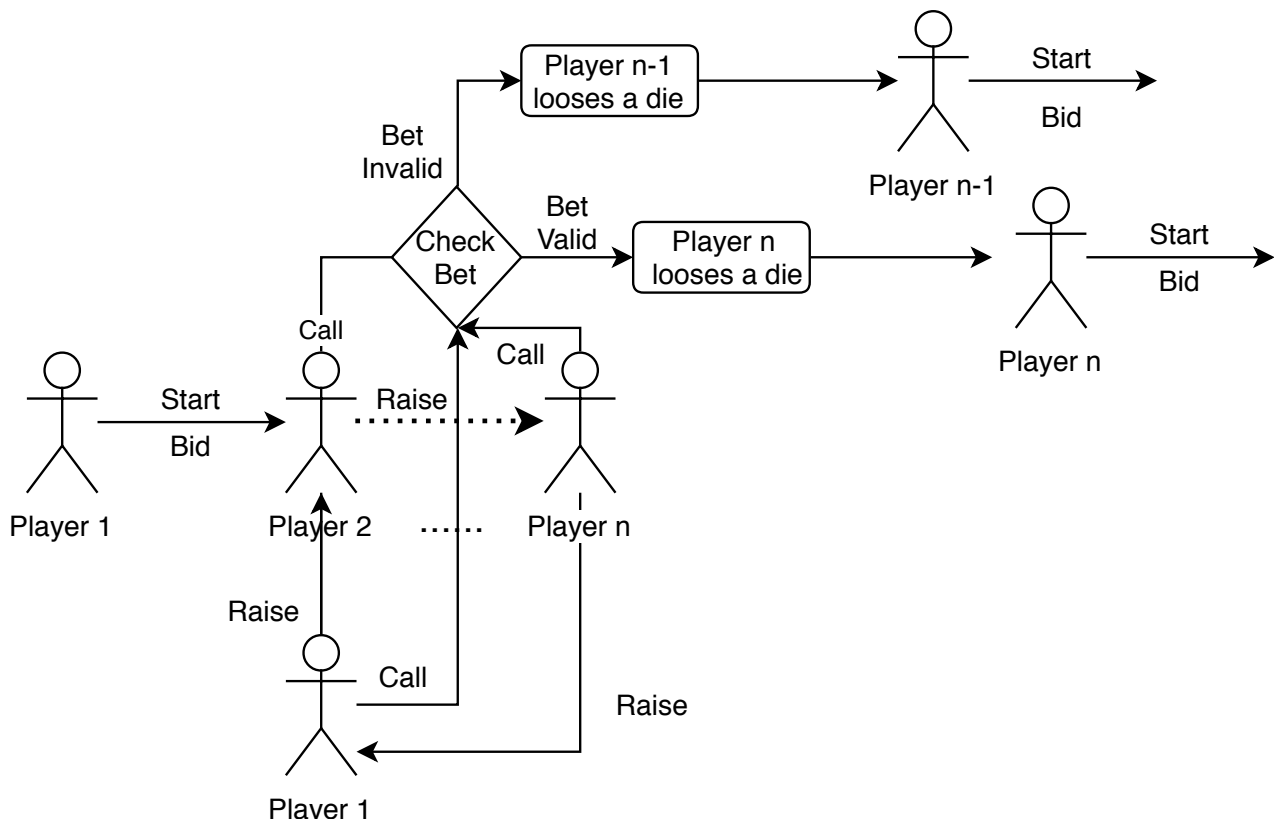





























Figure 1.1: Flowchart of a game of Perudo

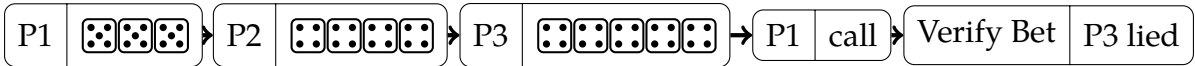
Using the above flowchart we will play a round of Perudo to get familiar with the rules.

Suppose that each player has the dice in **Table 1.1** and suppose that the sequence of bets seen in **Figure 1.2** are placed.

- Player 1 starts off the round with by betting 3 s.
- Player 2 then raises that bet by betting 4 s. This is a valid bet because if we look at **Equation (1.1)**,  $s(3, 3) = 33 < s(4, 4) = 44$ .
- Player 3 then raises this bet by betting 5 s. This is a valid bet because  $s(5, 4) = 54 > s(4, 4) = 44$ .
- Player 1 calls Player 3's bet. Since there were only 4 s, 1  in Player 1's hand (remember that  are wild cards and count as every number) and 3 s in Player 2's hand, Player 3 loses a die.
- The round has finished and a new round starts again with Player 3 placing the starting bid.
- If Player 3 raised by betting 4 s and Player 1 called that bet, Player 1 would have lost a die as there are at least 4 s currently in play and a new round starts again with Player 1 starting.

Player 1	Player 2	Player 3
     	     	     

**Table 1.1:** Player's Dice



**Figure 1.2:** Round being played

## 1.4 APPROACH

Several approaches were used to try and create AI agents. First, a purely probabilistic approach was taken where the agent only considers the probability of the previous players bet when it decides whether it should raise the bet or call. Other approaches that were tried were modifications to this strategy but they took into account information such as the player's current dice and previous bets made. The final approach developed was using the MiniMax algorithm. All agents had 2 parameters that could be tweaked, one for deciding at what the cut-off probability is for deciding if a bet is false and the other parameter that decided what probability the player would bluff with.

To make the code easily extensible, an Object Oriented approach was used throughout the project.

A terminal based application was written that used TCP sockets to communicate with each player to allow them to place bets and receive updates on the game state after each player's turn. This application also allowed a number of games to be played which allows for easy simulation.

## 1.5 METRICS

The evaluation of the AI agents was done through 2 main methods:

- **Number of wins:** This is the main metric that will be used. It allows us to measure how well the agent performs. By comparing the number of wins each of the approaches had it will allow us to evaluate how good that agent performs.
- **Call Accuracy:** This metric will allow us to find the approach that is the best at calling other players bluff.

Using these two metrics will help find the agent that is good at both winning and calling and not just good at one of these tasks.



# TECHNICAL BACKGROUND

---

## 2.1 TOPIC MATERIAL

All of the papers mentioned in this section seem to focus on head-to-head (only 2 players) games of Perudo so they are not a direct comparison to the methods implemented in this project.

One paper [1] uses methods similar to the ones in this project except that the rule set that they used includes the *Aces* rule which is not the rules used for this project. In this paper, 4 different strategies are implemented:

1. A basic strategy which bids based on the basis of analysing expected values. It also takes in an extra parameter, which sets an upper bound for raising a bid. If the expected value is higher than this bound then an *Aces* is called.
2. An extended strategy – based on the basic strategy – which deduces back, from the initial bids, the opponent’s dice. This method won more when it played against the basic strategy.
3. A foreseeing strategy which keeps track of all bets made and tries to use that to deduce the opponents dice.
4. A bluff strategy. This method was built from the idea that instead of winning one round it tries to win the whole game instead. This method not only considers the best theoretical build but other variations such as sending the minimum possible bet or by bluffing. The bluffing was not random and was based on the probabilities of the dice on the table, how ever no mention is made how they calculated the probabilities.

The paper concludes that although the bluffing strategy is the best that they have implemented it still cannot be classified as a successful and “clever” player.

This paper [2] attempts to use a modified version of Counter Factual Minimisation (CFR) called Fixed-Strategy Iteration Counterfactual Regret Minimisation (FSICFR) to approximate an optimal strategy for Perudo. Essentially, CFR traverses extensive game subtrees, recursing forward with reach probabilities that each player will play to each node (i.e. information set) while maintaining history, and backpropagating values and utilities used to update parent node action regrets and thus future strategy. However, due to the large size of the information set in Perudo, the number of recursive visits to nodes grew exponentially with the depth of the tree.


In FSICFR, the recursive CFR algorithm is split into two iterative passes, one forward and one backward, through a Directed Acyclic Graph (DAG). In the forward pass, visit counts and reach probabilities of each player are accumulated, yet all strategies remain fixed. (By contrast, in CFR, the strategy at a node is updated with each CFR visit.) After all the visits are

counted and probabilities are accumulated, a backward pass computes utilities and updates regrets.

The new method introduced provides a quicker training time for a standard 5 die head-to-head match of Perudo compared to CFR.

## 2.2 TECHNICAL MATERIAL

### 2.2.1 SOCKET SERVER

This webpage [3] details how to write a simple programme, using the  socket package, that accepts connections from a user, through a socket, and allows the main server to reply to these messages. It also shows examples of how to send a receive messages from multiple sources and handle them accordingly.

# PROBLEM ANALYSIS

---

The main goal at the start of the project was to develop and implement artificial intelligence that could play the game at a reasonable level and also to implement an extensible base to allow for the addition of new AI's. However, mid-way through the project, the goal became the development of a game hosting platform that allows humans to play against the AI as well as the AI to play against other AI.

One of the problems associated with developing an AI for this game is because the project requires the AI to be flexible, i.e. it must work well for any number of players. This causes a lot of the algorithms that are already used for this topic to not perform well as they are all trained on data for a head-to-head game. Another problem with trying to train models for this game is the large search space. With 4 players playing, the number of different combinations of the number of dice each player has is 625 and with 5 players it is 3125. The search space grows exponentially with the number of players in the game.

As well as the development of the AI, a system, as described above, had to be developed that allowed for games to be simulated between AI agents as well as gather statistics, such as win rate for each player, throughout the simulation. The system also has to allow people to use it from within the same network.

# THE SOLUTION

---

## 4.1 ANALYTICAL WORK

To calculate the probability of a bet occurring, the following calculations are used.

Note that  $l \in \{2, 3, 4, 5, 6\}$  and that anywhere that mentions “occurrences of the value  $l$ ” should read as occurrences of the value  $l$  (or  $\square$ ).

- Since  $\square$ 's are wild, this means that the probability of landing on any die face except 1 is  $1/3$ .
- Let  $a_{n,k,l}$  be the event that among  $n$  dice, there are exactly  $k$  occurrences of the value  $l$ . Calculating the probability of  $a_{n,k,l}$  amounts to rolling  $k$  times the value  $l$  and  $n - k$  times one of the other faces. By taking the first  $k$  positions we get  $2^k \times 4^{n-k}$ . Now multiplying this result by the set of possible combinations of  $k$  positions among  $n$  dice, gives

$$\binom{n}{k} 2^k 4^{n-k}$$

We also have that  $\text{Card}(\{1, 2, 3, 4, 5, 6\}) = 6^n$ . From this we get that

$$P(a_{n,k,l}) = \binom{n}{k} \frac{2^k 4^{n-k}}{6^n} = \binom{n}{k} \frac{2^k 2^{n-k} 2^{n-k}}{6^n} = \binom{n}{k} \frac{2^n 2^{n-k}}{6^n} = \binom{n}{k} \frac{2^{n-k}}{3^n} \quad (4.1)$$

- Now let  $A_{n,k,l}$  be the event that among  $n$  dice, there are at least  $k$  occurrences of the value  $l$ . To calculate the probability of  $A_{n,k,l}$ , we have to sum the probabilities for all  $a_{n,i,l}, \forall i \in [k, n]$ .

$$P(A_{n,k,l}) = \sum_{i=k}^n \binom{n}{i} \frac{2^{n-i}}{3^n} \quad (4.2)$$

- We can improve upon Equation (4.1) by using the player's own dice to factor into the calculation. The event  $a_{n,k,l} \mid a_{m,j,l}$  can be interpreted as the event of having exactly  $k$  occurrences of the value  $l$  among  $n$  dice knowing that we already have  $j$  occurrences of the value  $l$  among the player's  $m$  dice. This is exactly the same as saying, we have exactly  $k - j$  occurrences of the value  $l$  among  $n - m$  dice. Therefore we have

$$a_{n,k,l} \mid a_{m,j,l} = a_{n-m,k-j,l}$$

and

$$P(a_{n,k,l} \mid a_{m,j,l}) = \binom{n-m}{k-j} \frac{2^{n-m-k+j}}{3^{n-m}} \quad (4.3)$$

- We can also improve upon Equation (4.2) by using the probability described in Equation (4.3). Let  $A_{n,k,l} \mid a_{m,j,l}$  be interpreted as having at least  $k - j$  occurrences of the value  $l$  among  $n - m$  dice. Then

$$P(A_{n,k,l} \mid a_{m,j,l}) = \sum_{i=k-j}^{n-m} \binom{n-m}{i} \frac{2^{n-m-i}}{3^{n-m}} \quad (4.4)$$

The implementation of Equation (4.2) and Equation (4.4) can be seen in listings B.1 and listings B.2.

## 4.2 SERVER DESIGN

The game hosting platform was built using the `socket` library that comes installed with Python using the terminal as the user interface.

### 4.2.1 THINGS TO NOTE

- The `pickle` package is used to encode and decode data. This package implements binary protocols for serializing and de-serializing a Python object structure.[4]. This allows data that was encoded with `pickle` to be decoded to it's original type instead of converting the data to a string, converting that string to bytes, sending it to the player, the player converting the bytes back to a string and then to whatever type it should be. This means that the player does not need to have any idea what type the data read in should be converted to. Another advantage of using this package is that allows the server to work in both Python2 and Python3 as the bytes object that Python3 provides does not work in Python2.
- Each player is a tuple (`conn`, `address`) where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection. So calling `player[0]`(4.1) allows the server to access the socket object to send information to the player.
- The `socket.recv()` method waits until it receives data to proceed to the next line of code.

#### Listing 4.1: Sending Player information

```
1 player[0].sendall(pickle.dumps(self.total_dice))
2 player[0].recv(131072)
3 player[0].sendall(pickle.dumps(self.player_list[player].dice_list))
4 player[0].recv(131072)
```

### 4.2.2 SERVER COMMUNICATION

Using Figure 4.1, the order of operations of the server will be described.

1. The server sends each player the number of games they will be playing. Before sending the information to the next player, it waits until the previous player sends an "OK" response before continuing.
2. Each player is sent whether the player is out and if the game is over. Similar to above the server waits until it receives a response before proceeding.

3. If a player is not out then they are sent their dice.
4. A player is asked to place a bet.
5. The bet placed is then sent to all the other players.
6. Steps 2-5 are repeated until there is only one player left with dice.
7. After the game is finished, each player is sent a message stating if they have won or lost.

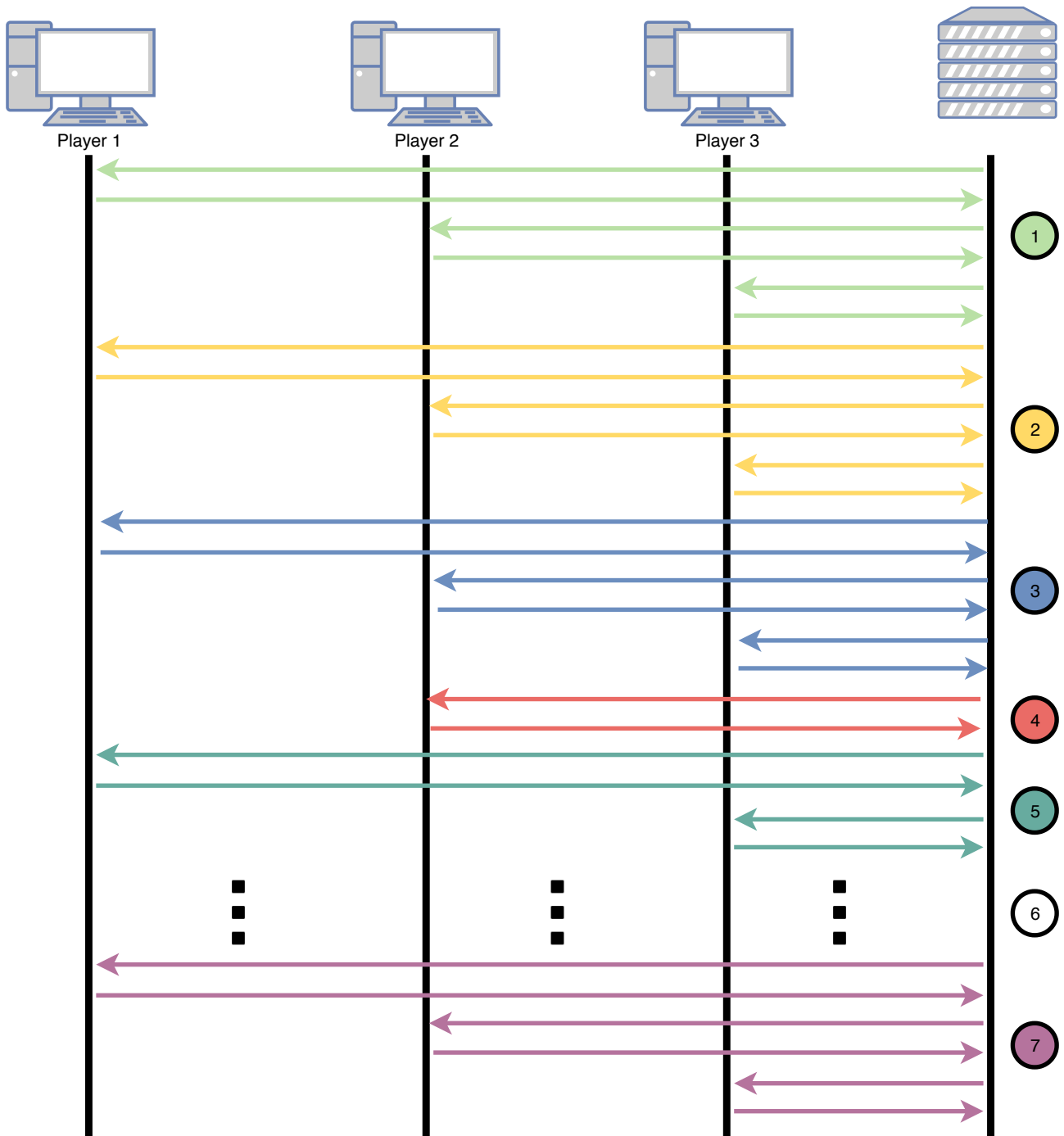


Figure 4.1: Order of operation during a game

## 4.3 METHODS

### 4.3.1 GENERATING BETS

The amount of dice to bet on were generated randomly using a skewed normal distribution using the `scipy` package and the die value to bet on was generated using previous bets. The implementation can be seen in [listings A.1](#).

The skewed PDF implementation in `scipy` is generated using

$$PDF_{\text{Skew}}(x, a, l, s) = \frac{2 * PDF(y) * CDF(a * y)}{s}$$
$$y = \frac{x - l}{s}$$

where  $a$  is a skewness parameter,  $l$  is a location parameter,  $s$  is a scaling parameter and  $PDF$  and  $CDF$  are the probability density function and cumulative density function, respectively, of the normal distribution. The graph seen in [Figure 4.2a](#) was generated with  $a = 5, l = .5, s = 4$ .



(a) PDF of a Skewed Normal Distribution



(b) Probability of a bet with 10 dice in play

**Figure 4.2:** Skewnorm compared to dice probability

A skewed distribution was used as, when placing bets, it is usually safer to place a low number of a die value. If a standard normal distribution was used instead of a skewed distribution, every amount would be just as likely when generating bets. However in the game this is not the case as bets that contain a high dice number are less likely to occur as seen in [Figure 4.2b](#), where the  $x$ -axis is the amount of dice that you are betting are on the table.

The die value itself was generated using bets made throughout the round. The code is more likely to generate a number that has already been bet on as it is more likely that there are more dice on the table, with a die value that has already been bet on.

### 4.3.2 IMPROVING THE PROBABILITY FORMULA

The formulas seen in [Section 4.1](#) do not take into account the bets that have been made in the current round. Suppose that  $D$  is a dictionary containing the numbers 2 to 6 as the keys

and the values being the highest amount of that die value that has been bet. Then

$$f(k, l) = P(A_{n,k,l}|a_{m,j,l}) + D[l] \quad (4.5)$$

where  $j, k, l, m$  and  $n$  are as described in [Equation \(4.4\)](#). Note that this is no longer a probability but rather a score metric as we can obtain values larger than 1. The implementation can be seen in [listings B.3](#)

### 4.3.3 MINIMAX

Minimax is a depth-first depth-limited search which examines all states . There are 2 phases in the algorithm

1. Descend the search tree up to the depth limit and apply the heuristic function.
2. Propagate those values up the tree.

The basic idea of Minimax is that during your move you want to maximize the heuristic value, as this is the best move for you, while the opponent tries to minimize the heuristic value as this will result in you performing a worse move.

#### EXAMPLE OF MINIMAX

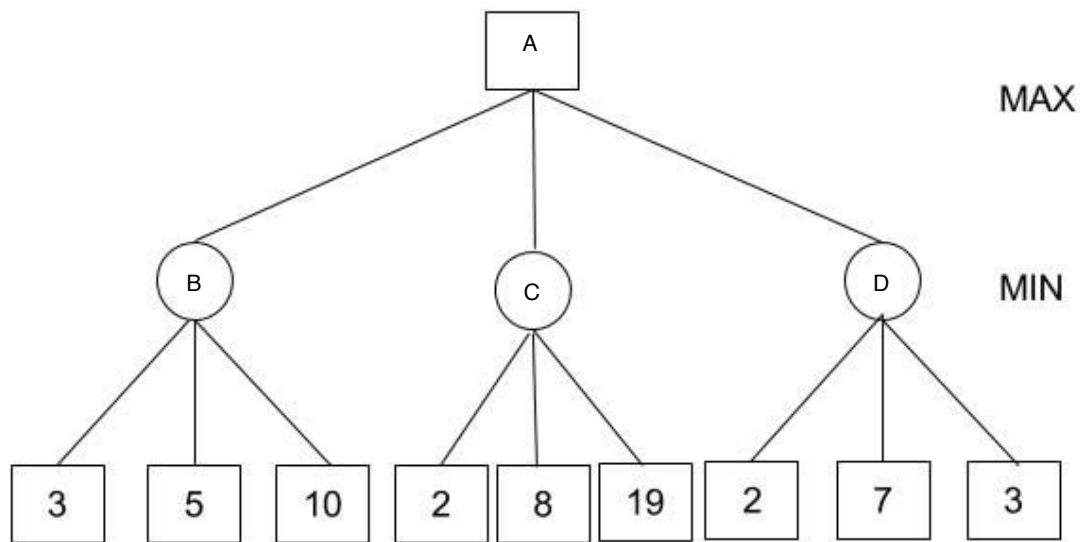


Figure 4.3: Minimax Tree

We traverse the tree depth-first, starting from the left branch.

1. Since B is a minimizer, it will pick the child node that has the lowest heuristic value. In this case B now has a value of 3.
2. C is a minimize so it will pick the node that has a heuristic value of 2. Now C has a heuristic value for 2.



3. D is a minimize, so it picks the child with a heuristic value of 2. D now has a value of 2.
4. Since A is a maximizer, it picks the child node with the highest heuristic value. In this case that would be B.
5. Therefore we will perform the action that node B represents.

The implementation for this can be seen in [listings C.1](#)

#### 4.3.4 $\alpha$ - $\beta$ PRUNING

The problem with regular Minimax is that it is an exhaustive method that explores unnecessary search spaces causing the algorithm to be slow and inefficient. For this reason  $\alpha$ - $\beta$  pruning will be used.

In  $\alpha$ - $\beta$  pruning there are two threshold values:

- $\alpha$ , which represents the lower bound of a maximizing level.
- $\beta$ , which represents the upper bound of the minimizing level.

We terminate a search when we are

- below a max node with  $\alpha \geq \beta$  of any of its min ancestors.
- below a min node with  $\alpha \geq \beta$  of any of its max ancestors.

Let  $b$  be the number of children for each node and  $m$  is the maximum depth of the tree then:

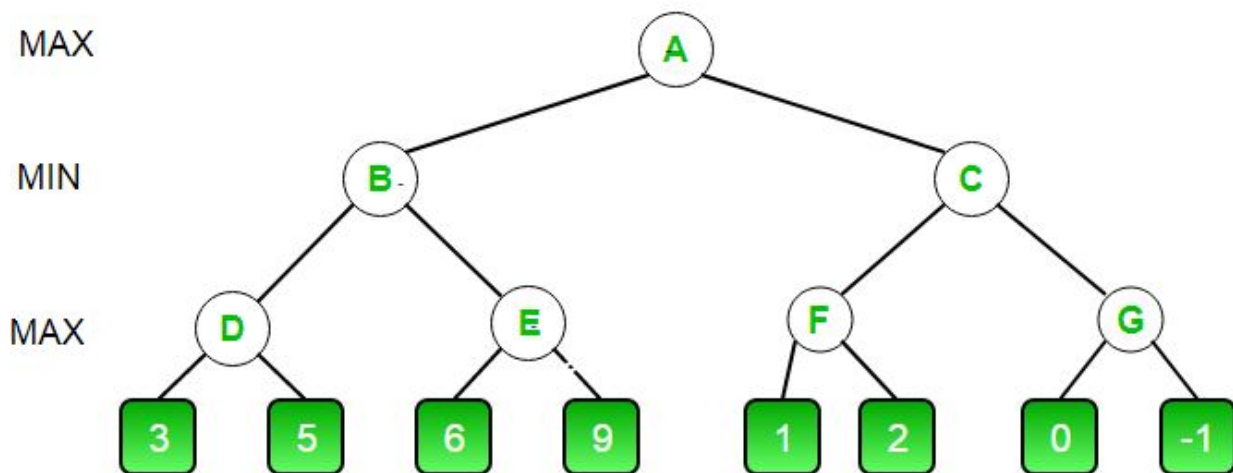
- For regular MiniMax, the time complexity is  $O(b^m)$ .
- For MiniMax with  $\alpha$ - $\beta$  pruning, the time complexity is  $O(b^{m/2})$

With the termination rules above, we can easily modify the implementation seen in [listings C.1](#) so that it implements  $\alpha$ - $\beta$  pruning, as seen in [listings C.2](#)

#### EXAMPLE

We will work through a partial example of how  $\alpha$ - $\beta$  pruning works by looking at the tree below.

1. The initial call starts from A. Initially,  $\alpha = -\infty$  and  $\beta = \infty$ . These values are passed down to subsequent nodes in the tree. We then explore the left branch of the tree.
2. At D, it has  $\alpha = -\infty$  and  $\beta = \infty$  as these are inherited from A.
3. D looks at its left child which returns a value of 3. Now  $\alpha = \max(-\infty, 3) = 3$ .
4. Since  $\alpha = 3 \not\geq \beta$  we continue searching other children of D.
5. D looks at its right child which returns a value of 5. Now  $\alpha = \max(3, 5) = 5$ .



6. D then returns a value of 5 to B. Now B has  $\beta = \min(\infty, 5) = 5$ .
7. Now the right child of B is explored. At E,  $\alpha = -\infty$  and  $\beta = 5$ .  $\beta \neq \infty$  as B passes its value down to E.
8. E looks at its left child, which returns 6. Now  $\alpha = \max(-\infty, 6) = 6$ .
9. Since  $\alpha = 6 \geq \beta = 5$ , we do not need to check the other child of E. This is because B is a minimizing node and as D guarantees that it will return at most 5, while after exploring the left child of E, we know it will return at least 6. Therefore exploring along the right child of E is useless as we will never pick E.

After performing the algorithm on the whole tree, we end up with the resulting search space seen in Figure 4.4

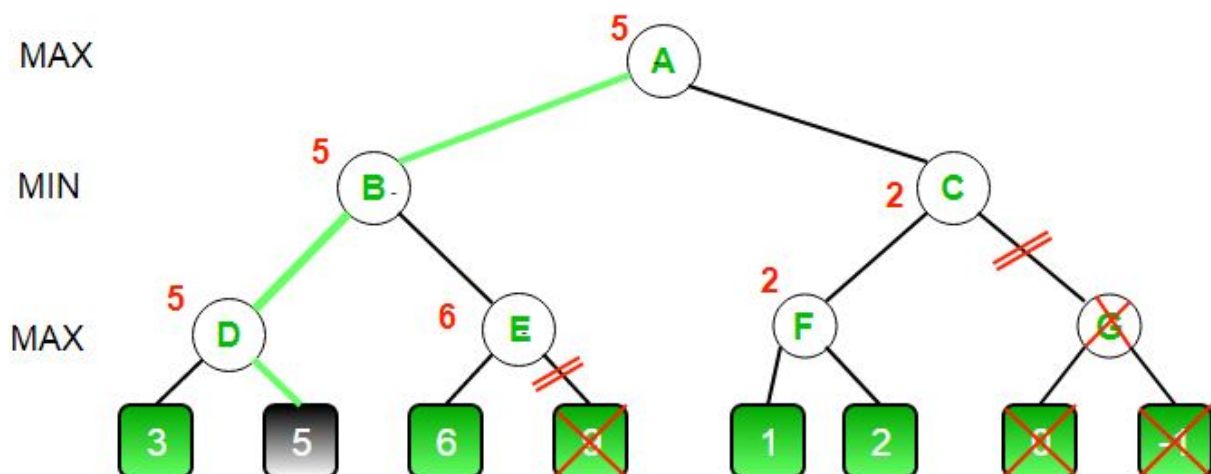


Figure 4.4: Final Tree after performing  $\alpha$ - $\beta$  pruning

In this example we saved time compared to the regular Minimax algorithm by not exploring 3 nodes. In larger graphs the number of nodes that are not explored in the  $\alpha$ - $\beta$  pruning version of Minimax is a lot larger than this. This means we save time searching and this allows us to search more nodes in the same amount of time.

## 4.4 CLASS STRUCTURE

The class structure of the project this project can see seen in [Figure 4.5](#) and the dependency graph can be seen in [Figure 4.6](#).

## 4.5 IMPLEMENTATION

### 4.5.1 STRATEGY FOR AIPLAYER'S

The general strategy for all the AI's except MiniMax is as follows

1. If you are placing a starting bet then bet that there are at least 2 to 4 dice of some number.
2. If you are not placing a starting bet, then generate bets using the method seen in [Section 4.3](#).
3. If the probability (or score), calculated using [Equation \(4.2\)](#) for DumbAIPlayer and [Equation \(4.4\)](#) for SLDumbAIPlayer and [Equation \(4.5\)](#) for LDumbAIPlayer, for any bet is less than the prob value passed into the function then the AI calls the previous players bet and goes back to step 1.
4. Otherwise, the AI then places a random bet from the generated bets with a probability given by the bluff variable or it places the bet from the generate bets that has the highest probability.

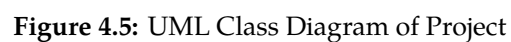
The actual implementation for placing bets can be seen in [listings B.4](#).

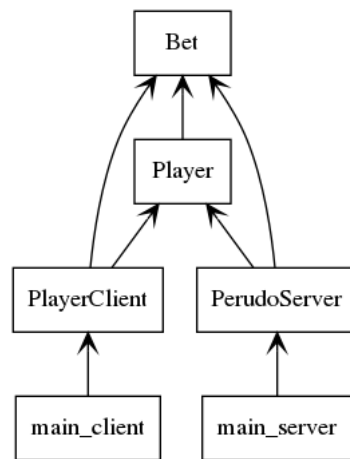
### 4.5.2 STRATEGY FOR MINIMAX

The strategy that the both LMiniMax and SLMiniMax players is as follows

1. If you are placing a starting bet then bet that there are at least 2 to 4 dice of some number.
2. If you are not placing a starting bet, then run the minimax( ) method ([listings C.2](#))
3. If the score, calculated using [Equation \(4.5\)](#) for LMiniMax and [Equation \(4.4\)](#) for SLMiniMax, for the bet returned by the minimax( ) method is less than the prob value passed into the function then the AI calls the previous players bet and goes back to step 1.
4. Otherwise, either the AI places a random bet like above or it places the bet output by the minimax( ) method.

The implementation can be seen in [listings C.3](#).





**Figure 4.6:** Package Structure of Project

# EVALUATION

---

The success of the AI's was measured by using the accuracy of the calls it made and the number of games it won as well. During the testing the order of the players was shuffled to ensure that the same players do not follow after each other and to prevent the same player going last in a round as whoever went last did better as they had the most information.

## 5.1 SELECTING PARAMETERS

All the AI's had to be tested against themselves but with different parameters to try and obtain the ideal values for `bluff` and `prob` for each AI. This test was done using 100 games.

	.1	.25	.4
.1	92.72%, 7	83.75%, 5	83.33%, 0
.25	74.2%, 23	73.29%, 18	75.41%, 4
.4	69.27%, 18	69.11%, 13	69.06%, 12

Table 5.1: `prob` v.s. `bluff` for `DumbAIPlayer`

As we can see `DumbAIPlayer` works the best with `prob` = .25 and `bluff` = .25.

	.1	.25	.4
.1	86.9%, 10	77.65%, 11	84.36%, 2
.25	76.52%, 24	76.64%, 18	70.85%, 4
.4	65.1%, 12	68.81%, 16	64.76%, 3

Table 5.2: `prob` v.s. `bluff` for `SLDumbAIPlayer`

As we can see `SLDumbAIPlayer` works the best with `prob` = .25 and `bluff` = .1. These are also the values that are used for `SLMiniMax`.

	.1	.25	.4
.1	25.02%, 12	18.45%, 5	15.08%, 2
.25	64.6%, 16	62.43%, 14	56.78%, 10
.4	66.46%, 22	66.32%, 9	65.34%, 10

Table 5.3: `prob` v.s. `bluff` for `LDumbAIPlayer`

As we can see `LDumbAIPlayer` works the best with `prob` = .4 and `bluff` = .1. These are also the values that are used for `LMiniMax`.

## 5.2 RESULTS AND ANALYSIS

### 5.2.1 RESULTS

500 games were simulated to obtain the results seen in the table below.

Type of AI	Call Accuracy	Games Won
DumbAIPlayer	64.71%	65
SLDumbAIPlayer	75.95%	200
LDumbAIPlayer	68.91%	225
SLMiniMax	40.4%	0
LMiniMax	43.16%	0
RandomAI	9%	0

**Table 5.4:** Call Accuracy and Number of games won for each AI

After this, a human played against LDumbAIPlayer and the results are collected below.

Player	Call Accuracy	Games Won	Average Number of Dice
Human	35.75%	9	2.6
LDumbAIPlayer	35%	11	3.2

**Table 5.5:** Call Accuracy, Number of Games won and the Average number of dice left when won

### 5.2.2 ANALYSIS OF RESULTS

From [Table 5.4](#), we can gather the following:

1. All of the AI's are better than the baseline RandomAI bot.
2. LDumbAIPlayer is the best AI developed as it has won the most games, however it is not as good as SLDumbPlayer at correctly calling. Perhaps this could be fixed by further tweaking the value of prob for LDumbAIPlayer.
3. The fact that both SLMiniMax and LMiniMax win no games is very surprising. Perhaps the extra information received by using a Minimax tree is detrimental to the way the game is treated in this project.

From [Table 5.5](#), we can see that LDumbAIPlayer is about as good as the human playing against it.

# CONCLUSIONS

---

## 6.1 PROJECT APPROACH

The approach for this project was to try and develop a code base that be easily built on and improved. To implement any other AI, one would have to have a class that inherits from the `Player` class and implements the `place_bet()` method and uses the `check_bet()` method to check if their bet is valid. After that you would have to add it to the `players` dictionary in the `PlayerClient.py` file.

## 6.2 RESULTS DISCUSSION

The results achieved by the algorithms developed in this project are good however due to the the way the parameters were selected it may not be as good as it could be. Playing each AI against a version of itself but with different parameters may not be a good idea as they all follow the same strategy. Maybe a better idea would be to try and play a number of games with every AI but with different parameters however this would involve playing 100 games with 27 players, which would take a long time. Also running the AI for more than 100 games each would provide more reliable results for the parameters.

Another problem with my approach is that I use the values of .1, .25 and .4 for the parameters but there may be better values which are being ignored.

## 6.3 FUTURE WORK

- To further improve upon this project one could develop a web interface. This would allow for an easier experience of playing the game against the AI as one would not need to have the Python files locally. This would also help with testing the quality of each of the AI's as it would allow for more wide spread play.
- The AI's seem to call unnecessarily more often when they get to a low amount of dice. With this problem addressed, I believe that the AI would perform even better.
- Something else that could be done to improve the project would be to allow for a player to keep track of all the other players bets and the amount of dice they have. Currently, only the dice the player has themselves is stored however with this improvement I believe this would allow more strategies based on probability to be implemented.
- Another AI could be implemented that instead of using just what `LDumbAIPlayer` used, it would also use the number of times that a bet with a certain die value has been place rather than just the largest bet on that die value.



- Given more time, it would be possible to use using strategies such as Reinforcement Learning or Neural Networks to create AI that models the game in a way that is not purely probabilistic. However the problem of having the model created be able to play with any numbers players still exists.

# BIBLIOGRAPHY

---

- [1] Norbert Boros and Gábor Kallós. “Bluffing computer? Computer strategies to the Perudo game”. In: *Acta Univ. Sapientiae, Informatica* 6 (2014), pp. 56–70. DOI: [10.2478/ausi-2014-0018](https://doi.org/10.2478/ausi-2014-0018). URL: <http://www.acta.sapientia.ro/acta-info/C6-1/info61-4.pdf>.
- [2] Neller, Todd W. and Hnath, Steven. “Approximating Optimal Dudo Play with Fixed-Strategy Iteration Counterfactual Regret Minimization”. In: *Advances in Computer Games*. Ed. by van den Herik H. Jaap and Aske Plaat. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 170–183. ISBN: 978-3-642-31866-5.
- [3] Nathan Jennings. *Socket Programming in Python (Guide) Real Python*. URL: <https://realpython.com/python-sockets/> (visited on 03/18/2019).
- [4] *pickle Python object serialization Python 3.7.3rc1 documentation*. URL: <https://docs.python.org/3/library/pickle.html> (visited on 03/19/2019).
- [5] J Sum and J Chan. “On a liar dice game - bluff”. In: *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693)*. Vol. 4. Nov. 2003, 2179–2184 Vol.4. DOI: [10.1109/ICMLC.2003.1259867](https://doi.org/10.1109/ICMLC.2003.1259867).
- [6] Jacek Madziuk. “Modeling the Opponent and Handling the Uncertainty”. In: *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 169–180. ISBN: 978-3-642-11678-0. DOI: [10.1007/978-3-642-11678-0\\_11](https://doi.org/10.1007/978-3-642-11678-0_11). URL: [https://doi.org/10.1007/978-3-642-11678-0%7B%5C\\_%7D11](https://doi.org/10.1007/978-3-642-11678-0%7B%5C_%7D11).
- [7] Jacek Madziuk. “Computational Intelligence in Mind Games”. In: *Challenges for Computational Intelligence*. Ed. by Jacek Duch Wodzisaw and Madziuk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 407–442. ISBN: 978-3-540-71984-7. DOI: [10.1007/978-3-540-71984-7\\_15](https://doi.org/10.1007/978-3-540-71984-7_15). URL: [https://doi.org/10.1007/978-3-540-71984-7%7B%5C\\_%7D15](https://doi.org/10.1007/978-3-540-71984-7%7B%5C_%7D15).
- [8] Daniel Livingstone. “Perudish: a game framework and modified rule-set for Perudo”. In: *COMPUTING AND INFORMATION SYSTEMS JOURNAL* 9.3 (2005), pp. 61–69.

# Appendices

# APPENDIX A

---

Listing A.1: Generating Bets

```
1 def gen_bets(self, total_dice, bet_history, last_bet=None, num_bets=50):
2     if last_bet:
3         bets = set()
4         i = 0
5         probs = skewnorm.pdf(range(last_bet.num_of_dice, total_dice + 1), 5, .5, 4)
6         probs = probs / sum(probs)
7         dice_probs = [1/5 for _ in range(5)]
8         for die in bet_history.keys():
9             dice_probs[i - 2] += bet_history[die] / 100
10        dice_probs = [x/sum(dice_probs) for x in dice_probs]
11        while len(bets) != num_bets and i < 150:
12            die_value = int(choice(range(2, 7), 1, p=dice_probs)[0])
13            die_amount = int(choice(range(last_bet.num_of_dice, total_dice + 1), 1,
14                                   ↪ p=probs)[0])
15            bet = Bet(die_value, die_amount)
16            if bet.verify_bet(total_dice, last_bet, verbose=False):
17                bets.add(bet)
18
19            i += 1
20
21        return list(bets)
22
23    return [Bet(x, y) for x in range(2, 7) for y in range(1, 4)]
```

# APPENDIX B

---

Listing B.1: Equation (4.2)

```
1 def calc_prob(self, total_dice, dice_value, num_of_dice):
2     return sum([self.ncr(total_dice, i) * (2**(total_dice - i) / 3**total_dice)
3                 for i in range(num_of_dice, total_dice + 1)])
```

Listing B.2: Equation (4.4)

```
1 def calc_prob(self, total_dice, dice_value, num_of_dice):
2     end_index = total_dice - len(self.dice_list)
3     our_dice = self.dice_list.count(dice_value) + self.dice_list.count(1)
4     start_index = num_of_dice - our_dice
5     return sum([self.ncr(end_index, i) * (2**(end_index - i)) / (3**end_index)
6                 for i in range(start_index, end_index + 1)])
```

Listing B.3: Equation (4.5)

```
1 def calc_prob(self, total_dice, dice_value, num_of_dice, bet_history):
2     end_index = total_dice - len(self.dice_list)
3     our_dice = self.dice_list.count(dice_value) + self.dice_list.count(1)
4     start_index = num_of_dice - our_dice
5     prob = sum([self.ncr(end_index, i) * 2**(end_index - i) / 3**end_index
6                 for i in range(start_index, end_index + 1)])
7     prob += bet_history[dice_value] / 100
8     return prob
```

#### Listing B.4: Placing a bet

```
1 def get_best(self, total_dice, bet_history, prob, bluff, last_bet):
2     if not last_bet and len(self.dice_list) == 1:
3         return Bet(self.dice_list[0], 1)
4     elif len(self.dice_list) == 1:
5         return "call"
6     else:
7         bets = super().gen_bets(total_dice, last_bet)
8         probs = [self.calc_prob(total_dice, bet.dice_value, bet.num_of_dice, bet_history,
9                               ↪ last_bet) for bet in bets]
10        index, value = max(enumerate(probs), key=op.itemgetter(1))
11        # print(probs)
12        if value <= prob:
13            return "call"
14        else:
15            if randint(1, int(1/bluff)) == 1:
16                return bets[randint(0, len(bets) - 1)]
17
18        return bets[index]
```

# APPENDIX C

---

Listing C.1: Minimax Implementation

```
1 def minimax(self, total_dice, last_bet, max_turn=True, max_depth=4):
2     if max_depth == 0:
3         return (last_bet, super().calc_prob(total_dice, last_bet.dice_value,
4             ↪ last_bet.num_of_dice) * (1 if not max_turn else -1))
5
6     bets = super().gen_bets(total_dice, last_bet, 10)
7
8     best_value = float('-inf') if max_turn else float('inf')
9     bet_to_make = ""
10
11     for bet in bets:
12         bet_to_place, value = self.minimax(total_dice, bet, not max_turn, max_depth - 1)
13
14         if value > best_value and max_turn:
15             best_value = value
16             bet_to_make = bet
17
18         if value < best_value and not max_turn:
19             best_value = value
20             bet_to_make = bet
21
22     return (bet_to_make, best_value)
```

### Listing C.2: Minimax with $\alpha$ - $\beta$ pruning

```

1 def minimax(self, alpha, beta, total_dice, last_bet, max_turn=True, max_depth=4):
2     if max_depth == 0:
3         return (last_bet, super().calc_prob(total_dice, last_bet.dice_value,
4             ↪ last_bet.num_of_dice) * (1 if not max_turn else -1))
5
6     bets = super().gen_bets(total_dice, last_bet, 10)
7
8     best_value = float('-inf') if max_turn else float('inf')
9     bet_to_make = ""
10
11    for bet in bets:
12        bet_to_place, value = self.minimax(alpha, beta, total_dice, bet, not max_turn,
13            ↪ max_depth - 1)
14
15        if value > best_value and max_turn:
16            best_value = value
17            bet_to_make = bet
18            alpha = max(alpha, best_value)
19            if beta <= alpha:
20                break
21
22        if value < best_value and not max_turn:
23            best_value = value
24            bet_to_make = bet
25            beta = min(beta, best_value)
26            if beta <= alpha:
27                break
28    return (bet_to_make, best_value)

```

### Listing C.3: Placing a Bet with MiniMax

```

1 def get_best(self, total_dice, bet_history, prob, bluff, last_bet):
2     if not last_bet and len(self.dice_list) == 1:
3         return Bet(self.dice_list[0], 1)
4     elif len(self.dice_list) == 1:
5         return "call"
6     else:
7         bet, _ = self.minimax(float('-inf'), float('inf'), total_dice, bet_history,
8             ↪ last_bet)
9         bet_prob = super().calc_prob(total_dice, bet.dice_value, bet.num_of_dice,
10             ↪ bet_history)
11         # print("Final bet was: {} with probability {}".format(repr(bet), bet_prob))
12         if bet_prob < prob:
13             return 'call'
14         elif randint(1, int(1/bluff)) == 1:
15             bets = super().gen_bets(total_dice, bet_history, last_bet)
16             probs = [self.calc_prob(total_dice, bet.dice_value, bet.num_of_dice,
17                 ↪ bet_history) for bet in bets]
18             if not probs:
19                 return "call"
20
21             sorted_bets = [(x, y) for x, y in sorted(list(zip(probs, bets)), key=lambda
22                 ↪ pair: -pair[0])]
23
24             return sorted_bets[randint(0, int(len(sorted_bets) / 2))][1]
25
26    return bet

```