Final Year Project Report

Optimising Strategy and Creating a Game Simulator for Liar's Dice

Stephen O' Farrell | 15459202

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 25, 2019

Abstract

Liar’s Dice, sometimes referred to as Perudo or 067, is a dice game popular in many parts of the world. This thesis will explore the various strategies associated with the game of Liar's Dice and attempt to find an optimal setup which the average player can employ to improve their game. In this report, I investigate the importance of using certain strategies and try to find out the most effective level of emphasis to place on each individual strategy. I will outline how I built up an AI capable of playing the game piece-by-piece and tested various strategies against each other in order to get an idea for how important they were. My end goal was to create a bot that could use the common strategies employed by humans at an optimal level in the hope that it could provide a challenge to experienced players, in turn leading to an ‘encoding’ of the strategy that could be followed by humans. I set up several variants of AI and a server that could be used to simulate games against them. I found that strategies such as bluffing, foreshadowing and recalling were very successful and that an optimal game plan involved using all 3 at a certain level.

Contents

1 Introduction

1.1 Topic

Liar’s Dice, also referred to as Perudo in South America or 067 in certain parts of China, is a multiplayer, stochastic dice game with many parallels to Poker. The rules are simple enough:

1.2 Rules

1.2.1 Starting the Game

Each player gets a cup & 5 dice which they must roll, concealing their dice from the other players. Ones are wild, meaning they can be considered any value from 1 to 6. One of the players is then chosen at random to make a ‘bet’, which consists of a value and the number of that value the player believes are collectively in the game. Consider an example game where 3 players are playing; Player 1 may say “Three fives”, which indicates Player 1 is saying there are at least three fives in the game, shared between everyone’s dice.

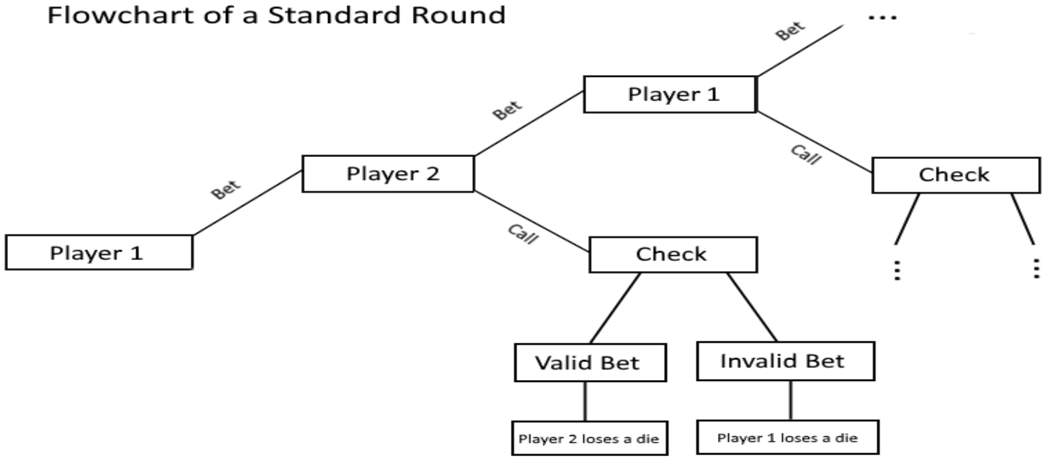
1.2.2 Each turn

The next player must then either ‘call’ or ‘raise’ this bet. Raising the bet is an easy concept, if we consider “Three fives” to be 35, then the next bet must be higher than 35. As such, “Four twos” (42), “Three sixes” (36) or “Seven fives” (75) are all valid bets but “Two ones” (21), “Three fours” (34) or repeating “Three fives” are examples of invalid bets. Once a valid bet is made, the onus is then passed on to the next player and they must make an increased bet. This continues until a bet is ‘called’.

1.2.3 Calling

Calling the bet is saying that the bet was a bluff, and everyone shows their dice. Using the earlier example, if Player 2 was to “call” after Player 1’s bet, then all three players show their dice and count the total amount of fives (and ones, as they are wild). If there are at least three fives between all fifteen dice, then the bet was fair, and Player 2 loses the call. If there are fewer than three fives, then Player 1 loses the call. Whoever lost the call removes one die from their cup and they start the next round. All players then reroll their dice and the game starts again with the losing player, who holds one die fewer.

Flowchart of a Standard Round



1.2.4 Ending the game

The game ends when there is only one remaining player with dice. This player is then declared the winner of the game.

Simulation of a game

Player 1 Dice	Player 2 Dice	Round Events	Result
⚀ ⚀ ⚂ ⚂ ⚄	⚁ ⚁ ⚃ ⚄ ⚅	1: 3 3 | 2: 3 6 | 1: 4 3 | 2: Call	Player 2 loss
⚀ ⚄ ⚄ ⚅ ⚅	⚀ ⚀ ⚁ ⚅	2: 3 5 | 1: 4 5 | 2: 4 6 | 1: 5 6 | 2: 6 6 | 1: Call	Player 1 loss
⚁ ⚂ ⚃ ⚄	⚁ ⚂ ⚂ ⚅	1: 2 6 | 2: 3 6 | 1: Call	Player 2 loss
⚁ ⚂ ⚂ ⚄	⚀ ⚄ ⚄	2: 3 3 | 1: 3 5 | 2: 4 5 | 1: Call	Player 1 loss
⚀ ⚂ ⚂	⚀ ⚁ ⚅	1: 3 3 | 2: 3 6 | 1: 4 3 | 2: Call	Player 2 loss
⚁ ⚂ ⚂	⚀ ⚀	2: 2 4 | 1: 3 2 | 2: 3 4 | 1: Call	Player 2 loss
⚄ ⚅ ⚅	⚅	2: 2 5 | 1: 2 6 | 2: Call	Player 2 loss

Final Result: Player 1 wins with 3 dice to 0

1.3 Variants

There are many variants of this ruleset. Perudo, a version of the game played in South America, has two main differences. They can “bid aces”, which means they can halve the number of the bet needed by calling ones, rounding up. For example, if the previous bet was “Five sixes”, they can say “Three ones”. To follow from this, the next player must either increase the number of ones (“Four ones”) or increase the number of their bet to double the number plus one (“Seven ___”, as double three plus one is seven). They can also call “Spot on” to add on to someone’s call. This means that they believe the called bet is exactly right. So, if the bet “Five fours” was called and a subsequent “Spot on” call was made, then there is a reward for the person calling “Spot on” if there are exactly five fours in the game.

Liar’s Dice has also become a phenomenon in China, but as a drinking game featured in almost every bar and club across the country. This variation is colloquially referred to as 067 in the Western regions. These numbers represent the two strongest deals in the game, 0 meaning the straight [2,3,4,5,6] and the 6 and 7 referring to five ones. In this version of the game, no dice are removed and losing a call simply means you take a drink.

For the purpose of this project, I will be using the normal ruleset outlined in the previous sections.

1.4 Motivation

My motivation for this project was to try and research an optimal strategy in a game which is very popular in many social circles but not so much in academic ones, especially when compared to the likes of Poker which has had extensive research done on optimisation. I didn’t want to create a bot which could play the game to levels above human skill similarly to the work done in games such as Chess and Go, as Liar’s Dice is a much less popular game which has little-to-no online presence. There are no Liar’s Dice tournaments or online portals where large amounts of money can be made, so I decided to take the approach of simply finding a bot that could play similarly to a human, but try to find out how to shape a strategy that can be followed by both bot and human to improve their chances of winning.

1.5 Problem Statement

The question of “how can you make an optimal bot for Liar’s Dice” is an interesting one but doesn’t have much bearing in the real world. Creating an AI that can play the game perfectly and win every time simply isn’t feasible in a game that’s built on chance. As such, it’s an exercise in futility to try and achieve this. So, we aim to create an AI that can play the game as well or better than a human player. The main challenge is introducing aspects of human play that machines struggle to implement, such as intuition or noticing physical tics. If we can manage to find out the best way to play against a bot, it can give us some insight into the best way to play the game in general, assuming the bot plays at a similar level to humans.

1.6 Approach

To solve this, I had to consider the main strategies employed by humans and try to emulate that in a bot capable of playing the game. I considered bluffing, anticipating, recalling and making calls as the main attributes used to describe someone's playstyle. I then had to create a couple of control bots who could play the game at a very basic level. My logic was that by enhancing a basic bot with these strategies and pitting them against the original, I could figure out how important certain aspects were and thus figure out the importance of each individual strategy. I then had to create an arena capable of hosting the game and simulating large amounts of games between these bots to try and find out how often each variant would win in a heads-up matchup. This arena also allows humans to play so it can test how well these bots fare against actual players. To test the optimal levels of each strategy, I used a form of A/B testing. The specialists each have their own parameter and they randomly select a value *n* for this, within a certain range. This value is then sent to the arena which documents *n* along with whether they won the round or not. This is then run thousands of times and the values are sent to a csv file where they can be further analysed. I then compare all the metrics and interpret the data to find the optimal strategy.

1.7 Metrics

There are 3 main metrics that I track to gauge how well the bots performed; win rate, strength of win and call accuracy.

- **Win Rate** is the percentage of times they reduce their opponents dice to zero and is the primary performance metric.
- **Strength of win** is the average amount of dice they have left when they win. A bot who wins every game with an average of four dice remaining is a more dominant player than one who has an average of one and a half.
- **Call accuracy** deals with how often their calls are correct. While it's not as important as the other two metrics, it's still worth tracking as it gives an insight into playstyle when put into context. A bot with low call accuracy but high win rate is playing a high-risk style very well, while another with high call accuracy and low win rate is playing a controlled style with little success.

1.8 Project

I managed to find a good estimate for optimal values of each metric. This means that I found a strategy which works optimally against bots and can be followed by humans.

I also set up a server which could be used to test various bots against each other. Any students looking to do a similar project would only have to add a few lines to their bot in order to use the server. This can be used as a measure of how good a bot is over any amount of games.

2 Technical background

2.1 Topic Material

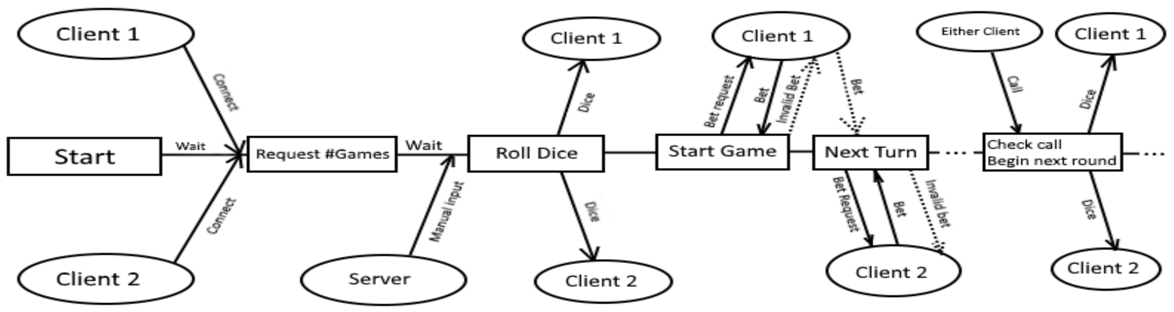
Liar’s Dice is not a very well-researched game and there wasn’t much precedent to go on when compared to other similar games. BOROS/KALLÓS authored a great paper which explored the different strategies that could be applied to an AI capable of playing the game[1]. Many parallels can be drawn between this thesis and their results. They also used bots that could implement the strategies used in this thesis, but they didn’t attempt to find a good value for them. Their research showed that the strategies were successful but failed to find a measure for how important each one was.

Todd W. Neller and Steven Hnath also published a paper about Liar’s Dice[2]. They tried to implement Counterfactual Regret Minimisation but ultimately found that applying CFR to a Liar’s Dice AI wasn’t a very valuable strategy.

2.2 Technical material:

2.2.1 Sockets

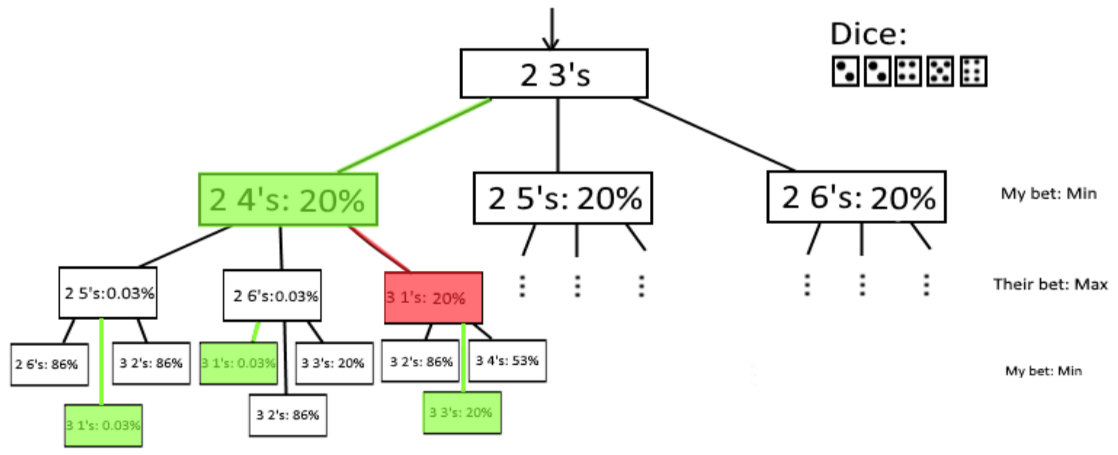
In terms of technical material used for my project, the main aspect would be the sockets that formed the basis of the server. To figure out how to do this, I used the bones of a Python echo server[3] and adapted it to understand the rules of the game. When the server starts up, it waits until two clients connect and then asks how many games you would like to play. The game then begins. The server rolls the dice and sends these out to each client before choosing a player to begin the game. It sends out a bet request to that client and, once the bet is received, checks whether the bet was valid. If valid, the next player is sent a bet request and so on. If not valid, it asks for a valid bet and keeps asking until it receives one. The server keeps this up until a ‘call’ is made, at which point the server checks if the previous bet was valid. It acts accordingly and starts the next round by rolling the dice again. This continues until either player runs out of dice and then the next game begins. This then continues until the number of games specified earlier have been played. All of this is based on sockets as the server uses these sockets to send and receive all the messages. As such, any clients that want to connect to the server must be coded to be able to receive and send these messages in a certain order.



This flowchart shows a basic overview of how the server runs chronologically. Dotted lines indicate a possible route.

2.2.2 Minimax

I also had to figure out the minimax algorithm, though I implemented a custom version which would use the binomial probability formula to try and lead the game down favourable paths for the AI. The minimax algorithm is a very popular AI algorithm that looks ahead to different future scenarios[4], choosing the one which gives the program the best chance of success. My implementation looks three rounds ahead and chooses one of the three paths which would suit it best.



This shows a basic implementation of minimax where only the next 3 possible bets are considered. All three possibilities are equally promising with these dice so the AI would select one at random.

2.2.3 CSV

To collect and analyse the data, I had to learn how to cast it to a Comma Separated Value (CSV) file[5] in order to graph the results with R. This was used when I wanted to get an optimal level for each strategy. Each bot would send its randomly selected 'n' value and the server would log that, as well as whether or not the round was won. This data was then sent to a CSV file where graphs and analysis could be easily made from it.

2.2.4 Numpy

Finally, I used numpy's[6] mean function to get the averages of some of the data collected, again for easy and quick analysis.

3 The Problem

While the goal of the project is to try and create the best strategy for Liar’s Dice, there are clear limitations to how that can be done.

The game is not particularly popular outside of certain circles. In particular, there is a minimal online presence for the game. This means that it’s almost impossible to get real-world examples of the game which could be used for analysis or modelling. As such, we must simulate games between bots, meaning that any results wouldn’t be quite as useful as game data gained from games involving humans. Also, it doesn’t consider the possibility that the game could have multiple people. This shouldn’t be a significant issue as we’re simply testing strategies against each other, but it could still somewhat skew results. So the quest for a ‘perfect strategy’ may require more in depth research.

4 The Solution

My solution includes 8 different python files, all written in Python 3.7.0. One server, one player client and six separate bots.

4.1 Server, *diceserver.py*

The server acts as the arena, it uses sockets to connect to the bots and simulates the game a predetermined amount of times. It keeps track of the dice and tells each bot what dice they have and the bet that was just made. It understands the rules of the game, so it accounts for invalid bets and rejects them. Once n games have been simulated, it prints out the win ratio, the average strength of win and the call accuracy of each competitor. It also takes in all the values of ‘n’ that each specialist bot uses, along with whether that value won the round and sends it to a CSV file for further analysis. For example, if Historybot played and won a round with an ‘n’ value of 25, then the server would add 25,1 to the csv file (25,0 would be added if the round was a loss).

4.2 Player Client, *playerclient.py*

The player client is simple enough. It simply connects to the server and logs the server inputs to the terminal while sending the command line inputs back. It’s useful as it allows a human to connect to the server and play against the bots or another player.

4.3 Control bot #1, *randombot.py*

The first control bot is a bot that plays the game completely randomly. The only rule it understands is that it must either increase the bet or call. It doesn’t consider its own dice or calculate any probability. It’s not a very clever bot and is the absolute baseline measure for performance. It would be expected that any worthy player should hold a very positive win rate over Randombot.

4.4 Control bot #2, *basicbot.py*

The second control bot is an AI that understands the game to a basic level. It can look at its own dice and calculate some basic probabilities based on that information. It plays a very conservative style where it will always try to make bets that have a high probability of being true. It calculates the probability by using the cumulative binomial probability formula. For example, the probability of there being “X Ys” is as follows:

$$\sum_{i=X}^{o} \binom{o-c}{i} \left(\frac{1}{3}\right)^{i} \left(\frac{2}{3}\right)^{(o-c)-i}$$

Where o is the amount of dice the opponent has and c is the amount of Ys the bot holds. Should Y equal 1, then the probabilities go from 1/3 and 2/3 to 1/6 and 5/6 respectively, seeing as ones are wild. By pitting this bot against itself with different enhancements is a good way to test how important each strategy is

4.5 Specialist bot #1, *historybot.py*

Considering the previous bets that have been made is a very basic strategy in Liar’s Dice. That said, it’s a very important one as it allows the player to make better guesses on whether a bluff has been made. When you consider history, you get a better idea of what dice the opponent has and, as such, can make more educated guesses during the game. Historybot concerns itself with how much to consider the past when making a bet, it’s Basicbot equipped with the ability to take in the bets as they come and change its bet to take that into account. It takes the binomial probability from Basicbot and adds a small percentage to the probability if the bet has been heard before. For example, let’s say Y has been called twice. Then the probability of the opponent having “X Ys”:

$$\sqrt{n(t)} + \sum_{i=X}^{o} \binom{o-c}{i} \left(\frac{1}{3}\right)^{i} \left(\frac{2}{3}\right)^{(o-c)-i}$$

n is the parameter deciding how much emphasis to put on the calls being made and t is how many times a bet involving Y has been made so far. The square root is introduced to ensure the number added remains low during long rounds. Otherwise, Historybot will rarely ever make calls. This means that the bets the bot makes will tend to follow the trend of the game but will still allow it to play its own strategy. By finding the optimal value for n, we can determine how highly we should value this strategy.

4.6 Specialist bot 2, *bluffbot.py*

Bluffing is a massive part of the game. The fact that you don’t know what dice the opponent has means that you never know whether to believe them. Without bluffing, the game depends far too heavily on who rolls the better dice. Bluffing is the key to making the game strategic rather than pure randomness. A good player will take advantage of bluffing to try and deceive their opponent, being unpredictable is key in Liar’s Dice. Luckily, it’s a trivial matter to add bluffing to Basicbot. All we must do is make a random bet every so often. We can change Bluffbot so that it only bluffs n% of the time where, once again, this n is the parameter that we want to optimise.

4.7 Specialist bot 3, *futurebot.py*

Anticipating what move the opponent will make is a rather niche strategy but it can be used. Futurebot uses a simple minimax algorithm to look 3 bets ahead and tries to steer the game down a path that suits itself. It will look at the next n possible bets for 3 moves and select the move that has a high probability of working well in the future. Once again, we want to optimise n.

4.8 Final bot, *alphabot.py*

By equipping Basicbot with all three main strategies with optimal n values, alphabot is the final product and should be able to put up a very good challenge against human players. It can also be used to test the final parameter: How likely the bot is to make a call. This is a simple parameter; if the probability of the opponent’s bet is below n%, the bot calls. By finding the best value for n we can finally have an idea of how to play the game at a high level and a human player can change their strategy to match this.

5 Evaluation

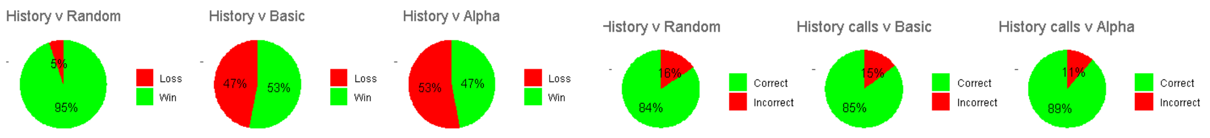
To evaluate my solution, I used the server to run 10,000 simulated games between bots in order to find out how each matchup went. I simulated games between the specialist bots and the control bots to find out how effective each strategy was. I then used the data gained from these simulations to find a good value for history, bluffing and foreshadowing and plugged these into the Alphabot. Once I found a good estimate for Alphabot's parameters, I ran more simulations between the control bots and Alphabot in order to find a good estimate for the calls parameter.

Finally, I incorporated human trials by playing 10 games against basic bot and then alphabot. However, this sample size is rather low so it's best to take these results in context.

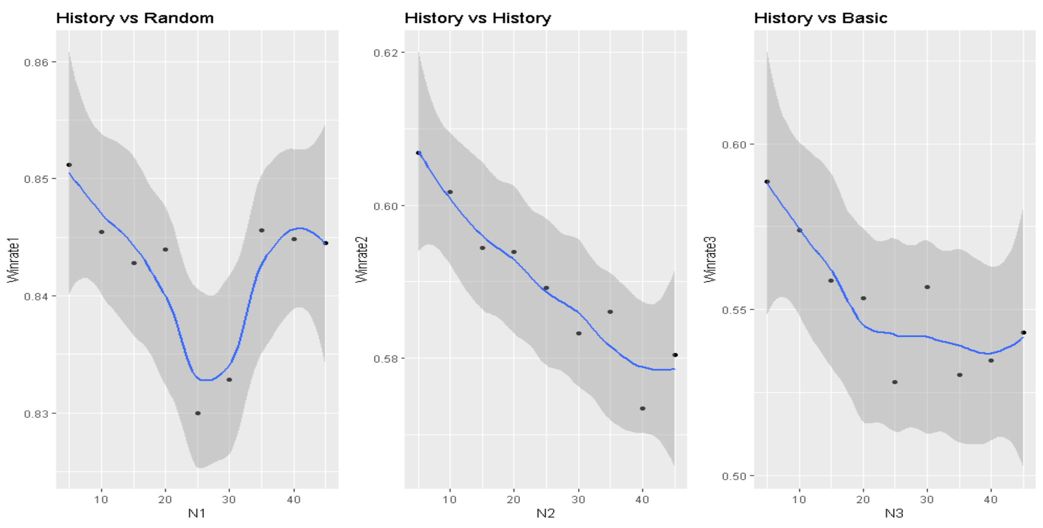
To test the software, I ran many simulations with the specific intention of breaking the server or clients. However, the fact that this is a closed system means that error handling isn't of the utmost importance and it was more important that the head-to-head strategies could be tested.

6 Conclusions:

6.1 History

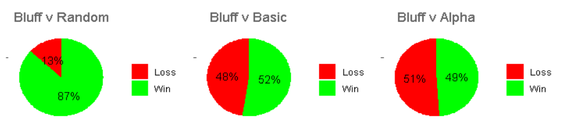


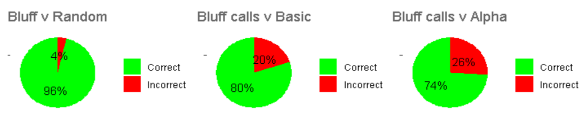
Using history is clearly an improvement, as it holds a decent win rate of 53.15% against Basicbot over 10,000 full games. The call rate is very high, indicating that using history leads to smarter decisions and
more correct calls. Historybot had an average of 2.46 dice when winning, meaning it didn't particularly dominate its games.



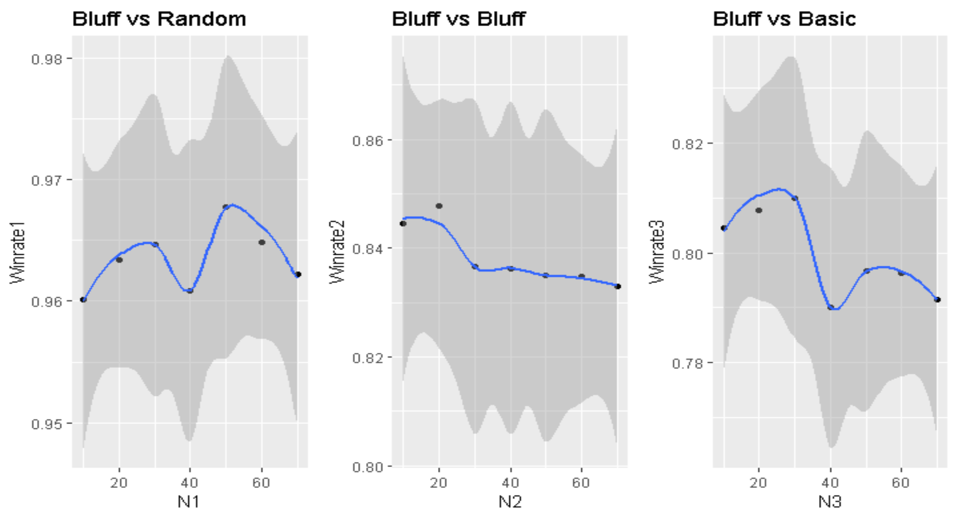
The win rate for Historybot was at its highest when n was less than 10, meaning that it had the most success when it put a low emphasis on history. It holds a decent call accuracy, but its games weren't particularly dominant. From this we can gather that the most success can seemingly be found by considering the history of the game, but not putting too much emphasis on it. It seems like setting n to 10 would be a good estimation.

6.2 Bluffing



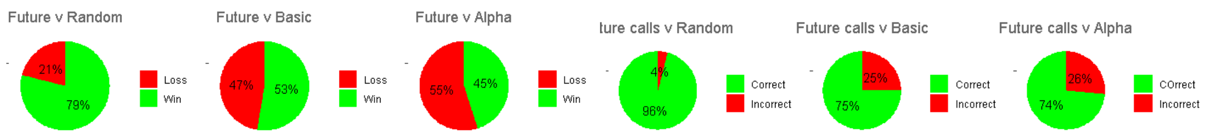


Bluffbot also holds a positive win rate over both
control bots, meaning it can be considered an important strategy. That said, bluffing is something that would likely be more powerful in a game involving humans. Physical nuances are an important part of it, and it lends itself more to the psychological side of the game. The way the bots are set up, they're rather vulnerable to being bluffed so these results should be taken with a grain of salt. Call success rates are very high, as would be expected. Bluffbot holds the title of most dominant contestant, with an average of 3.16 dice when it wins.

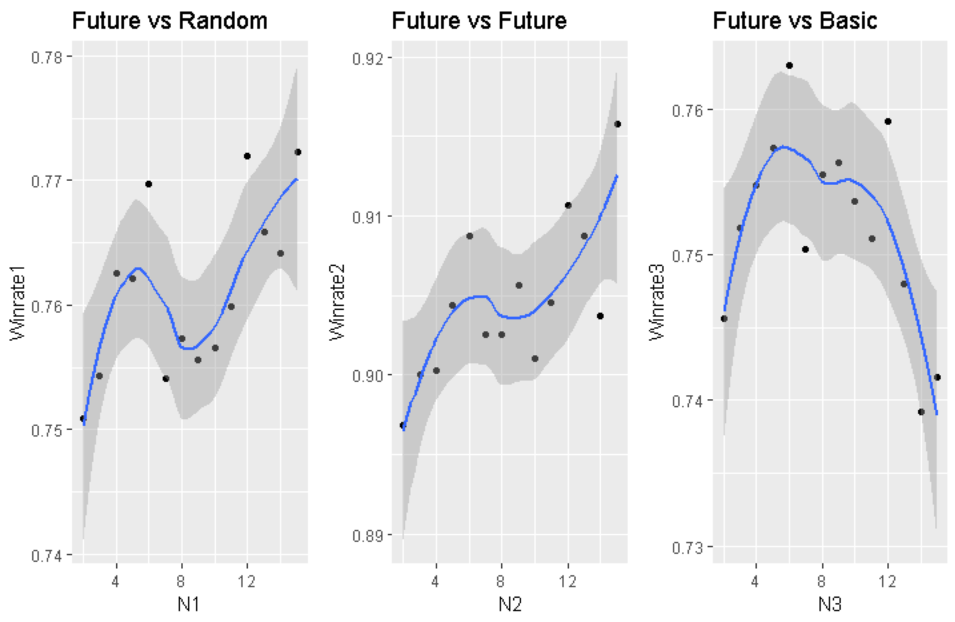


As we can see here, the success rate varies depending on which bot they go up against. However, a promising trend is the fact that win rate seems to decrease as the bluff rate increases to high levels, implying that the bots tend to notice heavy amounts of bluffing. Based on these graphs, it seems optimal to bluff around 20-30% of the time.

6.3 Future



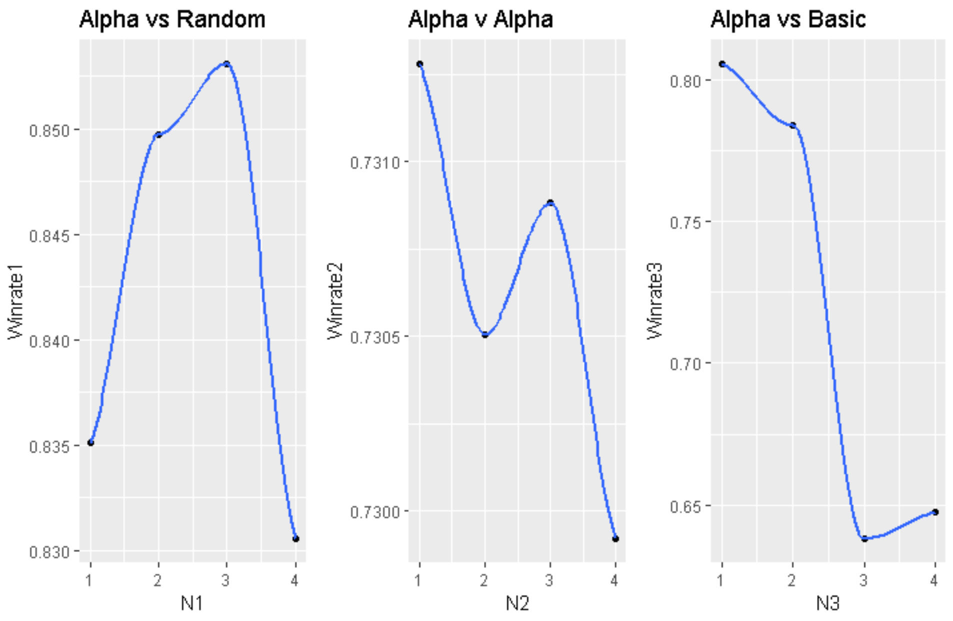
Futurebot follows similar patterns to the other
bots used so far. Though its win rate against Randombot is surprisingly low, which could be attributed to the fact that Futurebot cannot plan the game as easily if Randombot refuses to follow a logical playstyle. Call success rate is also standard. Futurebot is the least dominant player, averaging only 2.43 dice when it wins.



From this we can gather that there are many different values for n that would be usable, though 6 or 12 seem to be the most effective. As such, I'll elect to go with 12 for the parameter as it gives the larger range of values to consider. It's interesting to see that the win rate against Basicbot drops heavily as n increases past 12, but the opposite effect can be seen in the other matchups. It's possible that Basicbot's conservative playstyle works well against high n values.

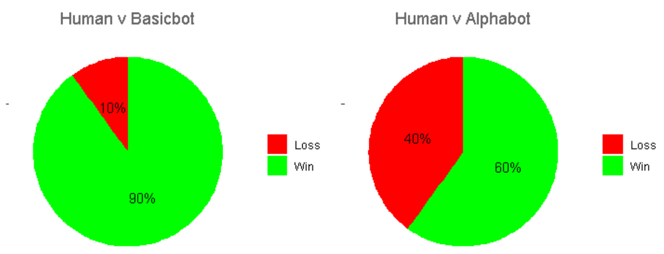
6.4 Calls

Alphabot is the second most dominant player behind Bluffbot, winning with an average of 2.83 dice.



Interestingly, there isn't much that can be drawn from this as the trends were very, very different. That said, you would expect this call success rate to decrease as N increases, so the jumps around 3-4 seem very promising and would lead me to believe that we should choose a value around 3.5.

6.5 Human Trials



As we can see from this win rate chart. Alphabot outperforms Basicbot when playing a 10-game set against a human player. This makes sense as the unpredictable nature of Alphabot makes it a much better candidate to match up well against human players. From this we can conclude that this final iteration of Alphabot likely represents a good strategy to follow in order to succeed at Liar's Dice.

7 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.url:https://www.researchgate.net/publication/267439621_Bluffing_computer_Computer_strategies_to_the_Perudo_game
2. Neller, Todd W. and Hnath, Steven. “Approximating Optimal Dudo Play with FixedStrategy 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/#echo-server (visited on 02/27/2019).
4. Baeldung. Introduction to Minimax Algorithm url:https://www.baeldung.com/java-minimax-algorithm (Visited on 02/12/2019)
5. Jon Fincher. Reading and Writing CSV Files in Python url:https://realpython.com/python-csv/ (Visited on 03/14/2019)
6. Numpy. url:http://www.numpy.org/ (Visited 10/30/2018)

8 Appendices

Appendix 1: Server

1.1: Handling Connections

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind(('127.0.0.1', 65432))
    s.listen()
    players = []
    for i in range(2):
        connection, address = s.accept()
        players.append((connection,address))
```

1.2: Rolling dice

```
def deal(totaldice):
    playerdice = []
    alldice = []
    for i in totaldice:
        for j in i:
            playerdice.append(ra.randint(1,6))
        alldice.append(playerdice)
        playerdice = []
    return alldice
```

1.3: Checking if a call was correct or not

```
def check(alldice, bet):
    amount = bet[0]
    number = bet[1]
    count = 0
    for j in alldice:
        for i in j:
            if(i == number or (i == 1 and number != 1)):
                count+=1
    return count>=amount
```

1.4: Casting to CSV

```
with open('ab.csv', 'w') as csvfile:
    fieldnames = ['Percentage', 'Win']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    for i in abcsv:
        writer.writerow({'Percentage':i[0],'Win':i[1]})
```

Appendix 2: Bots

2.1: Choosing a bet (Basicbot)

```
def combet(enemyamount,enemynumber,enemydice,dice):
    count = 0
    possible = []
    for i in range(6):
#parameter 3
        enemynumber+=1
        if(enemynumber>6):
            enemyamount+=1
            enemynumber=enemynumber%6
        if(enemynumber == 1):
            temp2 = enemyamount-dice.count(enemynumber)
        else:
            temp2 = enemyamount-dice.count(enemynumber)-
dice.count(1)
        temp = chance(enemydice,enemynumber,temp2)
        possible.append([temp,enemyamount,enemynumber]) #maybe come
back
    possible.sort(reverse=True)
    choice = ra.randint(0,2)
    betamount=possible[choice][1]
    betnumber=possible[choice][2]
    return (betamount,betnumber)
```

2.2 Choosing a bet (Future/Alphabot)

```
def possibilities(branch, enemydice, dice,n):
    x = branch[1]
    x1=x
    y = branch[2]
    total = []
    for i in range(n):
        y += 1
        if(y > 6):
            x += 1
            y = (y%6)
        if(y!=1):
            x1 -= dice.count(1)
        else:
            x1 -= (dice.count(1)+dice.count(y))
        total.append([chance(enemydice,y,x1),x,y])
    return total

def combet(branch ,enemydice , dice,n):
    x = possibilities(branch,enemydice,dice,n)
    xs = [possibilities(i, enemydice,dice,n) for i in x]
    for i in xs:
        count=0
        for j in i:
            t = possibilities(j,enemydice,dice,n)
            t.sort(reverse=True)
            j[0]=t[0][0]
            i[count][0] = t[0][0]
            count+=1
        i.sort()
    count=0
    for i in xs:
        x[count][0]= i[0][0]
        count+=1
    x.sort(reverse=True)
    return (x[0][1],x[0][2])
```