Simulating Yamslam in R:
Optimal Strategy of a Weighted Dice Game

by

Noah Wren Peterman

Submitted to the Department of Economics

School of Natural and Social Sciences

In partial fulfillment of the requirements

For the degree of Bachelor of Arts

Purchase College

State University of New York

December 16th, 2022

First Reader: Iuliia Chikish

Second Reader: Cedric Ceulemans

Abstract

Games of chance offer a level of mathematical strategy that can maximize the chances of success. Yamslam, a token-related spinoff of the famous Milton Bradley game Yahtzee, naturally creates strategic opportunities for players that aim to achieve optimal gameplay. This project explores dice roll simulations in R for modeling the best response strategies in several particular scenarios. Using the simulated expected values, I consider the possible actions and identify the optimal decisions. My results lay a groundwork for simulating the entire game of Yamslam and pinpoint the strategies that maximize the game points and therefore the chances to win.

Table of Contents

Introduction 4
Games Rules 4
Literature 6
Strategy with Bonuses 9
Theoretical Probabilities 11
Simulated Probabilities 13
    Combination Tracker Matrix 14
    Combination Code List 16
    Results Tabulation Matrix 20
Analysis & Results 24
    General Results Matrix 24
    Optimal Decision with 3-of-a-Kind 25
    Optimal Decision: [2,2,3,3,4] 29
    Adjusted Expected Value 30
Limitations 31
Conclusion 32
References 34
Appendix 35

Introduction

Games involving the use of dice are often assumed to be pure chance, or up to "the roll of the dice." However, when factoring in strategies that rely on probability, this notion of pure chance loses some of its merit. The simplistic nature of rolling dice hides just how complex the odds of the game are. In this examination of Yamslam, I explore the empirical probabilities of rolling dice through simulation. The data collected is tested through a series of combination codes that identify whether the group of values complies with conditions in the game. The two metrics used to produce explanatory results are probability and expected value. With this base set of data, conclusions such as optimal decision and strategy are formed and evaluated.

This paper aims to explain the entire process of determining optimal strategies with simulations. I detail how the code was written using R, and which data metrics were used to produce answers. Aside from manually checking to make sure the code functioned properly, equation-based probabilities were compared to the simulation probabilities.

Games Rules

Yamslam is a game built upon rolling dice to achieve combinations. While it is possible to play by yourself, it is recommended to be played with two to four people. The game materials consist of five 6-sided dice, with odd numbers being black and even numbers red. The scoring system uses a set of chips that pertain to one of the seven combinations, with only four chips per combination (there are eight total combinations but only seven different chips, which will be explained in a later section). The objective of the game is to have the highest score at the end of the game.

Per turn, each player is allowed a maximum of three rolls, although they are not required to use them all. To decide who goes first, each player rolls a single die (highest roll starts, followed by next player in clockwise direction). Play starts and continues until no more of the seven chip combinations remain.

The combinations available to pick from in chip form are 2 Pair (5 pts), 3-of-a-Kind (10 pts), Small Straight (20 pts), Flush (25 pts), Full House (30 pts), 4-of-a-Kind (40 pts), and Large Straight (50pts), per *Figure 1*. They all rely on some form of duplicate or sequential faces showing. There is an eighth combination called a "Yamslam" that is achieved from having all five dice show the same number. There is no chip assigned to this combination, the player who rolls this gets to pick any chip they'd like and take another turn. If all chips of a certain combination are gone, players can no longer score that combination. Additionally, if a player cannot score any combination by the end of their third roll, they do not score that turn.

Figure 1. Yamslam Chip Details

Yamslam Chip	Chip Value	Explaination of Chip
2-Pair	5	Two different pairs of dice. E.g. [**2,2,5,5**,1]
3-of-a-Kind	10	Three of the same number. E.g. [**1,1,1**,3,4]
Small Straight	20	Four numbers sequentiallly in a row. E.g. [**1,2,3,4**,3]
Flush	25	All dice have the same color. E.g. [**1,3,3,3,5**]
		In Yamslam, odd numbers are black and even numbers are red
Full House	30	Three of one number and two of another number. E.g. [**1,1,1,4,4**
4-of-a-Kind	40	Four of the same number. E.g. [**3,3,3,3**,6]
Large Straight	50	Five numbers sequentiallly in a row. E.g. [**1,2,3,4,5**]
Yamslam	50*	Five of the same number. E.g. [**6,6,6,6,6**]
		*When a player rolls a Yamslam, they can pick form any chip on the board. In most cases, the 50pt chip is most valued.

When all players from a single round fail to score a chip, the available chip of highest value is removed from play. The only way to collect a chip from this discarded pile is to roll a Yamslam.

Strategy strongly comes into play when bonuses are decided at the end of the game. There are four different bonuses that players can receive:

50 Point Bonus – Collect one of each chip combination (seven different chips)

20 Point Bonus – Collect six of the seven different chip combinations (cannot be combined with 50pt bonus if applicable)

30 Point Bonus – Collect a full stack of four chips (Ex: score all "3-of-a-Kind" chips)

20 Point Bonus – Take the last chip to end the game

Any applicable bonuses are added to each player score at the end of the game. The winner is decided by the highest score total[1].

Literature

To compare the probability results from an R dice rolling simulation, it is necessary to compute the probabilities of each combination theoretically. Litwiller and Duncan (1979) demonstrate how to compute probabilities of mutually exclusive, sequential events involving dice; they use the game of "Craps" to explore this concept. In Craps, players bet on the outcome of two rolled dice. The winner depends on who rolls the dice, and what sum shows up. "If on the first roll a sum of 7 or 11 is obtained, the shooter wins all the covered bets". The shooter (player who rolls the dice) wins with a sum of 7 or 11 only on the first roll (called a natural). All the other

[1] All Instructions adapted from Blue Orange Games

players win if the shooter rolls a 2, 3 or 12 (called craps). "If any other sum is rolled, it is called a point number." If a point number is rolled, the shooter keeps re-rolling the dice until either the point number is rolled again (the shooter wins) or a sum of seven is rolled (the other players win). The probabilities of Craps are conditional, and mutually exclusive with replacement.

To calculate probabilities, the authors created a table to compare the sums of high probability and low probability. This technique is a simplified version of what calculations the Yamslam combinations will require.

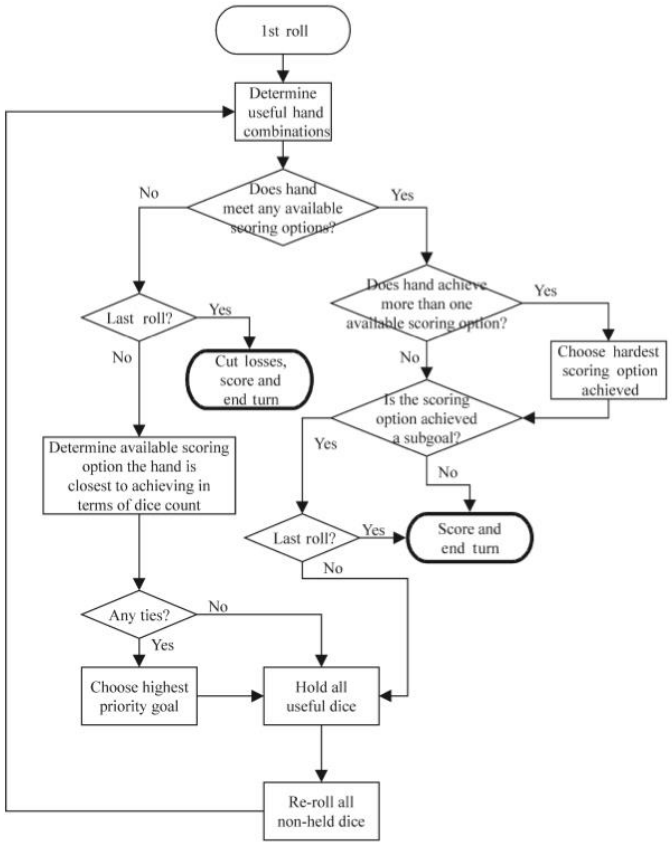
Figure 2. Game of Craps: Dice Outcomes & Probability

Sum	Ordered Pairs	Probability
2	(1, 1)	1/36
3	(1, 2), (2, 1)	2/36
4	(1, 3), (3, 1), (2, 2)	3/36
5	(1, 4), (4, 1), (2, 3), (3, 2)	4/36
6	(1, 5), (5, 1), (2, 4), (4, 2), (3, 3)	5/36
7	(1, 6), (6, 1), (2, 5), (5, 2), (3, 4), (4, 3)	6/36
8	(2, 6), (6, 2), (3, 5), (5, 3), (4, 4)	5/36
9	(3, 6), (6, 3), (4, 5), (5, 4)	4/36
10	(4, 6), (6, 4), (5, 5)	3/36
11	(5, 6), (6, 5)	2/36
12	(6, 6)	1/36

Source. Litwiller & Duncan, 1979

Once the probabilities of each combination are flushed out, the decision-making strategy becomes relevant. While dice games are largely dependent on chance, there is room for strategy to play a stronger role in maximizing your chances of winning. I believe that a decision tree will be necessary to be able to devise several different strategies that players could use. The decision tree pictured below is from a study performed on the game "Yahtzee," one consisting of similar combinations to Yamslam.

Figure 3. Decision Tree for Yahtzee - First Turn



Source. Maynard et al., 2001

Alike to Yamslam, a single Yahtzee turn allows for a maximum of three rolls with the goal of achieving some combination of numbers on a scoring sheet. Maynard et al. (2001) describe how the decision on each roll should be made, depending on what combinations are present. "The best opportunity is the one with the highest number of dice achieving the highest priority scoring option". The decisions assume an expert player is making them. Strategies must be fluid and players must be willing to alter their strategies in any given turn. In conjunction, it was determined that players must establish priorities to ensure the highest score possible. "First, the player should choose the most probable scoring option. For tiebreakers, the player should choose the highest

priority available option" . For expert players, there's a logical order of how decisions are made with respect to maximizing their expected value.

To improve the success of the decision strategies mentioned previously, players should look to decrease their problem space to maximize their scores as well. "… by decreasing the number of dice rolled on each turn, i.e., the problem space, a skilled player increases the chances of achieving a high score" (Maynard et al., 2001). This same strategy concept can be used for Yamslam; players can have the goal of achieving a higher value chip by only rolling the dice that do not contribute at all towards the numbers required.

Similarly to Maynard et al. (2001), a decision tree is pertinent for developing strategies in Yamslam. Their study will help me create decision trees for players at different levels, ranging from beginner to expert. As an example, beginner players may be naïve and have the sole goal of achieving the highest value chip every turn, while experts believe that may not be efficient in the long run.

Strategy with Bonuses

One of the many strategies associated to this game is based on bonus scoring. Players look to obtaining one or many of the previously mentioned bonuses to increase their score at the end. It would behoove a player to factor the final bonuses into their full game strategy to also increase their chances of getting one. Below are a few scenarios that explore strategy with respect to bonuses.

Scenario #1: Player 1 has four "3s" and one "5" at the end of turn three – They have 6 of the 7 different chips, the missing one being a "three-of-a-kind" worth 10pts. There is only one "three-of-a-kind" chip remaining, and Player 1 wants to get the 50pt bonus.

Player 1 could either take the “four-of-a-kind” chip worth 40pts and risk losing the bonus (if another player after them takes the last “three-of-a-kind”), or they take the “three-of-a-kind” 10pt chip, secure the 50pt bonus, but pass up on the 40pt “four-of-a-kind.”

Scenario #2: Player 2 has these five different die faces showing at the end of turn three: 1,2,3,4,5. Player 2 could take a larger straight worth 50pts, or a small straight worth 20pts. Player 2 already has three “small straight” chips but zero larger straight chips.

If Player 2 chooses the “large straight”, they risk not getting the fourth “small straight” and further risk the 30pt bonus. If Player 2 takes the “small straight,” they secure all four “small straights” and receive an extra 30 points. Simultaneously, that choice would prevent their opponent from getting the 50pt bonus, but also hinder themselves if they don’t have one yet.

Scenario #3: There are two chips remaining in play, a “2-pair” and a “large straight.” Player 1 has two “2s”, two “4s” and a 3 at the end of turn two.

Player 1 could roll both the duplicate “2” and “4” to attempt to get a large straight (2, 3, 4 plus a 1 & 5, or 5 & 6) or they could not roll at all and settle for a guaranteed chip, in 2-pair. The thought process here factors in the last chip bonus. Given that the remaining chip is now a large straight, the probability of rolling it is very small compared to a 2-pair. Therefore, the probability that Player 1’s opponent rolls nothing their next turn is high. If this is the case, it would give Player 1 the last chip by default; either Player 1 gets nothing on his final attempt for the large straight (last chip = 2-pair) or they roll a large straight (last chip = Large Straight). Player 1’s strategy is risk averse because they chose the safer option which guaranteed points over the non-guaranteed higher-valued option.

Due to limitations in the coding process (that are explained in a latter section), bonuses are not written into the R code; manual entry is necessary to account for their involvement. Bonuses have a strong influence on a player's decision towards the end of the game.

Theoretical Probabilities

The probabilities of each chip combination are calculated to set an estimate for what probabilities the simulation should produce. I present a tabulation of the comparison between probabilities in the Appendix.

Two Pair (A,A,B,B,X)

$_6C_2 \times {}_5C_2 \times {}_3C_2 \times 4 = 15 \times 10 \times 3 \times 4 = 1800$

1800 = 2-Pairs not including Full Houses (300/7776)

1800 + 300 = 2100 = 2-Pairs not including 4-of-a-Kind (150/7776)

(2250 / 7776) = 0.2893519 = **28.935% of Two Pair**

Three-of-a-Kind (A,A,A,X,X)

$_5C_3 \times 6 \times 5^2 = 1500$

Add 150 (4-of-a-Kind #s) + 6 (Yamslam #s) to the 1500 to satisfy all 3-of-a-Kind's

(1500 + 150 + 6) = 1656

(1656 / 7776) = 0.21296 = **21.296% of 3-of-a-Kind**

Small Straight (1,2,3,4,X) or (2,3,4,5,X) or (3,4,5,6,X)

2 x 5! = 240 Small Straights w/o repeating numbers

$12 \times (3! \times {}_5C_2) = 12 (6 \times 10) = 720$ Small Straights with repeating numbers

2 x (5!) = 240 = Large Straights (because all Large Straights = Small Straights)

(240 + 240 + 720) / (7776) = 0.15432 = **15.432% of Small Straight**

Flush (R,R,R,R,R) or (B,B,B,B,B)

$(1/2)^5 = 0.03125$

0.03125 x 2 = 0.0625 = **6.25% of Flush**

Full House (A,A,X,X,X)

6 x 5 = 30 types of Full House combinations

${}_5C_3$ = 10 ways to roll a single Full House combination

30 x ${}_5C_3$ = 300 total combinations of a Full House

(300 / 7776) = 0.03858 = **3.858% of Full House**

Four-of-a-Kind (A,A,A,A,X)

6 x 5 = 30 types of 4-of-a-Kind combinations

${}_5C_4$ = 5 ways to roll a single 4-of-a-Kind combination

6 x 5 x (${}_5C_4$) = 150 combinations of 4-of-a-Kind

(150 / 7776) = 0.01929 = **1.929% of 4-of-a-Kind**

Large Straight (1,2,3,4,5) or (2,3,4,5,6)

2 x (5!) = 240 combinations of 5 unique dice

(240 / 7776) = 0.03086 = **3.086% of Large Straight**

YamSlam (Five-of-a-Kind)

(1,1,1,1,1), (2,2,2,2,2), (3,3,3,3,3), (4,4,4,4,4), (5,5,5,5,5), (6,6,6,6,6)

6 / 7776 = 0.0007716 = **0.077% of Yamslam**

Simulated Probabilities

The inspiration behind creating this project was to figure out if there was a way to optimally play Yamslam. To add a personal anecdote, I'd often come across situations when playing that required a decision. Such decisions were often made with my gut, as opposed to statistics. To incorporate statistics into the decision-making process of each roll, a model that simulates gameplay was necessary. There are five main components to the Yamslam simulation code: a random number generator, a dice roll matrix, a combination tracker matrix, a combination code list, and a results tabulation matrix.

Random Number Generator & Dice Roll Matrix

The job of this function is to generate five random numbers, ranging between one and six, and list them in a row. Each number generated represents a single side of a six-sided die. To create a table that shows each roll, per *Figure 6*, a matrix had to be created to house the data. The function titled [roll] creates a matrix with [n] rows (*nrow = n*) and five columns (*ncol = 5*). The first argument in the matrix function would usually be the values in the table, but because they will be randomly generated, a separate function is used to identify the five random numbers in each row (so it is left blank, NA), as seen in *Figure 5*. To repeat this function a desired amount of times, the number of rows (*nrow = n*) was set equal to a variable, n. Therefore, the user must set the variable, n, equal to the number of times they want to generate five random numbers, and the code will run accordingly. E.g., n = 10,000 will randomly generate five numbers in a row, 10,000 times. To fill each roll with five random numbers, the script in *Figure 4* was created.

Figure 4. Random Number Generator Script

```
3  n <- 5  #Set number of roles equal to __
4  roll <- matrix(, nrow = n, ncol = 5)
5  # Run "Roll" to view empty matrix created
6
7  for (k in 1:n) {
8    roll[k,] <- sample(x = 1:6, size = 5, replace = TRUE)
9  }
```

This while loop is the basis of conditional statements throughout this code and is used to repeatedly sample numbers and distribute the results into the table created, specifically [n] number of times. The variable, k, dictates which row the five numbers will be placed in. This "roll[k,]" function will now repeatedly sample five random numbers and place them in row [k] of the matrix until the loop has reached its end. The loop will finish when it has completed its roll [n] and placed the data in the row [k]. Each number row is correlated with each number roll, e.g., the value of [k] increases by one whole number every generation until it reaches the final roll, n [for (k in 1:n)].

Figure 5. Blank Number Matrix

```
> n <- 5  #Set number of roles equal to __
> roll <- matrix(, nrow = n, ncol = 5)
> roll
     [,1] [,2] [,3] [,4] [,5]
[1,]   NA   NA   NA   NA   NA
[2,]   NA   NA   NA   NA   NA
[3,]   NA   NA   NA   NA   NA
[4,]   NA   NA   NA   NA   NA
[5,]   NA   NA   NA   NA   NA
>
```

Figure 6. Random Number Generator Matrix Results

```
> for (k in 1:n) {
+   roll[k,] <- sample(x = 1:6, size = 5, replace = TRUE)
+ }
> roll
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    4    2    4
[2,]    5    6    6    2    2
[3,]    5    6    2    4    1
[4,]    6    2    4    3    6
[5,]    2    4    5    4    2
>
```

Combination Tracker Matrix

The combination matrix is used to track what combinations you generate based on each individual roll. The structure is very similar to the dice roll matrix, however the dimensions and values in the table are different. To generate the structure, the matrix function is used again, with the parameters of a variable, n, number of rows (*nrow = n*), and eight columns (*ncol = 8*); this

function is titled [combination]. The first argument is set equal to zero, but will be replaced as the simulation progresses, per *Figure 7*.

Figure 7. Blank Combination Matrix

```
> combinations <- matrix(0, n, 8)
> combinations <- data.frame(combinations)
> combinations
  X1 X2 X3 X4 X5 X6 X7 X8
1  0  0  0  0  0  0  0  0
2  0  0  0  0  0  0  0  0
3  0  0  0  0  0  0  0  0
4  0  0  0  0  0  0  0  0
5  0  0  0  0  0  0  0  0
>
```

Combination labels are assigned to the column headings to show which column represents which chip, per *Figure 8*.

Figure 8. Matrix Combination Label Script

```
12  combinations <- matrix(0, n, 8)
13  combinations <- data.frame(combinations)
14  colnames(combinations) <- c("2P", "3K", "SS", "FL", "FH", "4K", "LS", "YS")
15  # Run "combinations" to view matrix with labels
```

A [for] loop is used again, but instead of generating random numbers for each row, it runs through a set conditions and checks them against each set of five random numbers to see if they are true. Each set is also sorted before conditions are checked for, per *Figure 9*.

Figure 9. Sort Roll Script

```
17  for (k in 1:n) {
18    x <- roll[k,]
19    sort(x)
```

If the check comes back true, a number one (1) will replace the number zero (0) in that position in the combination matrix table, per *Figure 10*. The code know knows which position to swap the numbers because of this script:

```
combinations[k,1] <- 1
```

E.g., this line refers to the [combinations] matrix, in the row [k] of the 1st column and sets that value equal to (1) instead of (0). This is known as a binary classification system, which uses either a (0) or (1) to categorize data. To reiterate, the variable, k, is correlated with the number of rolls, so for example, the 20th roll results will be placed in the 20th row.

Figure 10. Matrix with Labels

```
> combinations
  2P 3K SS FL FH 4K LS YS
1  0  0  0  0  0  0  0  0
2  0  0  0  0  0  0  0  0
3  0  0  0  0  0  0  0  0
4  0  0  0  0  0  0  0  0
5  0  0  0  0  0  0  0  0
>
```

Combination Code List

Each of the eight combinations have conditions that need to be satisfied for the combination matrix to effectively interpret each roll of five dice. While each combination requires different guidelines, certain functions were continually used between them all, such as:

unique(x): return the unique values in the argument, x.

Figure 11. Unique() Function

e.g.

```
x <- c(1,2,2,4,5) # generate a vector x
x # [1] 1 2 2 4 5
unique(x) # [1] 1 2 4 5
```

length(x): return the number of values in the argument, x.

e.g.

Figure 12. Length() Function

```
x <- c(1,2,3,5,5) # generate a vector x
x # [1] 1 2 3 5 5
length(x) # [1] 5
```

x[duplicated(x)]: return the values that have duplicates in the argument, x.

Figure 13. Duplicated Numbers Function

e.g.

```
x <- c(1,3,3,5,5) # generate a vector x
x # [1] 1 3 3 5 5
x[duplicated(x)] # [1] 3 5
```

sum(x): return the summation of all values in the argument, x.

Figure 14. Sum() Function

e.g.

```
x <- c(2,3,4,6,6) # generate a vector x
x # [1] 2 3 4 6 6
sum(x) # [1] 21
```

To explain further, both 3-of-a-Kind and Large Straight conditions are detailed for example. These two were chosen because their scripts differ in how they're constructed.

3-of-a-Kind

Figure 15. 3-of-a-Kind Script

```
27   #TESTING for 3-of-a-Kind
28   if(length(unique(x)) == 3 && length(unique(x[duplicated(x)])) == 1) {
29     combinations[k,2] <- 1
30   }
```

3-of-a-Kind tests for two conditions, and if both are true, the number (1) will replace the number (0) in the [combinations] matrix in the row [k] and column, 2. The conditions are as follows:

Figure 16. First Condition of 3-of-a-Kind

```
if(length(unique(x)) == 3 &&
```

_If the number of unique values in the argument, x, is equal to (3) AND

Figure 17. Second Condition of 3-of-a-Kind

```
length(unique(x[duplicated(x)])) == 1) {
```

_If the number of unique, duplicated values in the argument, x, is equal to (1), then that specific set of five randomly generated numbers contains 3-of-a-Kind.

_If one or both aren't true, 3-of-a-Kind is not present and the (0) in the combination matrix won't change.

Large Straight

Figure 18. Large Straight Script

```
76   #TESTING for Large Straight
77   x <- sort(x)
78   y <- rep(0,4)
79   for (i in 1:4) {
80     if (x[6-i] - x[5-i] == 1){y[i] <- 1}
81   }
82   if (sum(y) == 4) {
83     combinations[k,3] <- 1
84     combinations[k,7] <- 1
85   }
```

Large Straight tests the difference between each value in the five number generation, with the help of a [for] loop. The sum of the difference will signify whether or not a Large Straight is present, per *Figure 18.*

Figure 19. Code to Check Sequence of Numbers

```
78    x <- sort(x)
79    y <- rep(0,4)
80    for (i in 1:4) {
81      if (x[6-i] - x[5-i] == 1){y[i] <- 1}
82    }
```

_Sort the values of variable, x, in ascending order.

_Create a vector, y, consisting of (0) repeated four times.

_If the fifth value in [x] minus the fourth value in [x] is equal to (1), replace the first (0) in the vector, y, with a (1). If the fourth value in [x] minus the third value in [x] is equal to (1), replace the second (0) in the vector, y, with a (1). If the third value in [x] minus the second value in [x] is equal to (1), replace the third (0) in the vector, y, with a (1). If the second value in [x] minus the first value in [x] is equal to (1), replace the fourth (0) in the vector, y, with a (1).

Figure 20. Sum/Chart Input Script

```
83    if (sum(y) == 4) {
84      combinations[k,3] <- 1
85      combinations[k,7] <- 1
86    }
```

_If the summation of all values in vector, y, is equal to (4), then that specific set of five randomly generated numbers contains a Large Straight, as well as a Small Straight.

_If the summation is anything other than (4), a Large Straight is not present and the (0) in the combination matrix won't change.

Results Tabulation Matrix

The final component of the code returns a table of four different metrics that were measured in the duration of the code: The number of combinations rolled (#), the frequency that each combination was rolled in terms of percent (%), the chip value (CV) and expected value (EV), per *Figure 21*. A third matrix was used to create the table, but dissimilar to the previous ones, this matrix has a defined structure that isn't entirely reliant on the variable, n. There are 10 rows (*nrow = 10*), eight of them to list the combinations, one to use as a blank space (for separation between the combinations and totals), and the last row for totals, per *Figure 21*. The four columns are for the metrics described above (*ncol = 4*): "#", "%", "CV", "EV".

Figure 21. Matrix Creation and Labeling for Results

```
100  Chip_Prob <- matrix(, nrow = 10, ncol = 4)
101  Chip_Prob <- data.frame(Chip_Prob)
102  colnames(Chip_Prob) <- c("#", "%", "CV" , "EV")
103  rownames(Chip_Prob) <- c("2-Pair", "3-of-a-Kind", "Small-Straight", "Flush",
104  #Run "Chip_Prob" to show blank matrix with labels
```

Remaining labels from row #103 can be viewed in Appendix

A new function had to be introduced [data.frame] to properly store the data as equal-length vectors in the [Chip_Prob] matrix. The difference can be seen between *Figure 22* and *Figure 23*, regarding the matrix before and after it was converted:

Figure 22. Results Matrix without Labels

```
> Chip_Prob <- matrix(, nrow = 10, ncol = 4)
> Chip_Prob
      [,1] [,2] [,3] [,4]
 [1,]   NA   NA   NA   NA
 [2,]   NA   NA   NA   NA
 [3,]   NA   NA   NA   NA
 [4,]   NA   NA   NA   NA
 [5,]   NA   NA   NA   NA
 [6,]   NA   NA   NA   NA
 [7,]   NA   NA   NA   NA
 [8,]   NA   NA   NA   NA
 [9,]   NA   NA   NA   NA
[10,]   NA   NA   NA   NA
>
```

Figure 23. Results Matrix with Labels

```
> Chip_Prob <- matrix(, nrow = 10, ncol = 4)
> Chip_Prob <- data.frame(Chip_Prob)
> colnames(Chip_Prob) <- c("#", "%", "CV" , "EV")
> rownames(Chip_Prob) <- c("2-Pair", "3-of-a-Kind", "Small-Straight",
Straight", "YamSlam" , "|" , "Total_Rolls")
> Chip_Prob
                #  % CV EV
2-Pair         NA NA NA NA
3-of-a-Kind    NA NA NA NA
Small-Straight NA NA NA NA
Flush          NA NA NA NA
Full-House     NA NA NA NA
4-of-a-Kind    NA NA NA NA
Large-Straight NA NA NA NA
YamSlam        NA NA NA NA
|              NA NA NA NA
Total_Rolls    NA NA NA NA
>
```

The results in the table are produced using [sum] functions that pertain to specific locations in the matrix. E.g.,

Figure 24. Results Matrix Occurrence Script Example

```
106  Chip_Prob[1,1] <- (sum(combinations[,1]))
```

_the value located in the [Chip_Prob] matrix, in first row, first column, equals the summation of all values in the first column of the [combinations] matrix.

Figure 25. Results Matrix Probability Script Example

```
119  Chip_Prob[3,2] <- (sum(combinations[,3])/n)*100
```

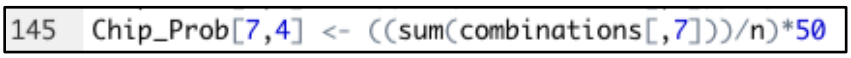
_the value located in the [Chip_Prob] matrix, in third row, second column equals the summation of all values in the third column of the [combinations] matrix, divided by variable, n, and then multiplied by 100.

Figure 26. Results Matrix Chip Value Script Example

```
132  Chip_Prob[5,3] <- "30"
```

_the value located in the [Chip_Prob] matrix, in the fifth row, third column equals 30 (the fifth row in this matrix contains all metrics surrounding a Full House, who's chip value is 30)

Figure 27. Results Matrix EV Script Example

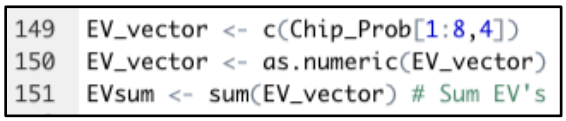


```
145  Chip_Prob[7,4] <- ((sum(combinations[,7]))/n)*50
```

_the value located in the [Chip_Prob] matrix, in the seventh row, fourth column equals the summation of all values in the seventh column of the [combinations] matrix, divided by the variable, n, and then multiplied by the corresponding chip value (the seventh row in [Chip_Prob] contains all metrics surrounding a Large Straight, who's chip value is 50)

The last component of the results matrix is determining the summation of all EVs in a single roll. This total differs from the "#s" total because the values it would add up are character types as opposed to numeric types. Therefore, RStudio would return an error message when attempting to take the sum of all EV's in the table. The solution was to define the EV column as a numeric type, and then take the summation of that, per *Figure 28*:

Figure 28. Sum of EV Script



```
149  EV_vector <- c(Chip_Prob[1:8,4])
150  EV_vector <- as.numeric(EV_vector)
151  EVsum <- sum(EV_vector) # Sum EV's
```

The final operations of the entire script that get run are [roll], [combinations], and [Chip_Prob]. These three functions generate matrices that show the dice rolls, the combination checklist, and the results table.

[roll] (results table located in the Appendix)

[combinations] (results table located in the Appendix)

[Chip_Prob] (results table located in *Figure 29*)

Figure 29. Results Matrix Output

```
> Chip_Prob #Produce a table with metrics pertaining to results of simulations
                    #      % CV      EV
2-Pair          14622 29.244  5  1.4622
3-of-a-Kind     10583 21.166 10  2.1166
Small-Straight   7748 15.496 20  3.0992
Flush            3149  6.298 25  1.5745
Full-House       2019  4.038 30  1.2114
4-of-a-Kind       988  1.976 40  0.7904
Large-Straight   1604  3.208 50   1.604
YamSlam            40   0.08 50    0.04
|                  --     -- --      --
Total_Rolls     50000           11.8983
>
```

This entire code is the structure for generating random rolls, holding all dice random. There are many circumstances when a player will keep some of their dice and roll the remaining. To simulate a situation when a certain number of dice are held constant as a certain number, the number generating portion of code changes. Instead of generating five random numbers and placing them in a single row, a variable number of constants are generated, and the resulting difference is then randomly generated.

Figure 30. Random Number Generator with Constants

```
 5  a <- 4
 6  b <- 5
 7  c <- 6
 8  for (k in 1:n) {
 9    d <- sample(x = 1:6, size = 1, replace = TRUE)
10    e <- sample(x = 1:6, size = 1, replace = TRUE)
11    Constant_Vect <- c(a,b,c,d,e)
12    roll[k,] <- Constant_Vect
13  }
```

Figure 30 depicts the code for a player keeping three dice with values of four ($a = 4$), five ($b = 5$) and six ($c = 6$), and rolling the other two, activating the functions [d] and [e], per *Figure 31*:

Figure 31. Random Number Generators

```
 9    d <- sample(x = 1:6, size = 1, replace = TRUE)
10    e <- sample(x = 1:6, size = 1, replace = TRUE)
```

The resulting vector is shown in *Figure 32*. As evident, the first three columns are kept constant, while the last two are random each roll. The eight combination functions will now run their conditions through the [Constant_Vect], instead of the previous [sample] function. All constant value simulation scripts are located in the Appendix.

Figure 32. Output of Random Number Generator with Constants

```
> roll
     [,1] [,2] [,3] [,4] [,5]
[1,]    4    5    6    1    6
[2,]    4    5    6    1    3
[3,]    4    5    6    6    4
[4,]    4    5    6    3    2
[5,]    4    5    6    6    1
>
```

Analysis & Results

General Results Matrix

Upon completion of writing the code, I conducted a simulation of 5,000,000 rolls. The purpose of this was to create a baseline of probabilities and expected values for each combination in the game. Through the analysis portion, combinations may be referred to by their nickname: 2-Pair (2P), 3-of-a-Kind (3K), Small Straight (SS), Flush (FL), Full House (FH), 4-of-a-Kind (4K), Large Straight (LS), Yamslam (YS). The results are shown in *Figure 33*.

Figure 33. Results of 5,000,000 Dice Roll Simulations

Combination	Probability	CV	EV
2 Pair	29.04%	5	1.452
3 of a Kind	21.31%	10	2.131
Small Straight	15.45%	20	3.09
Flush	6.26%	25	1.565
Full House	3.95%	30	1.185
4 of a Kind	2.00%	40	0.8
Large Straight	3.10%	50	1.55
YamSlam	0.08%	50	0.04

As expected, the chips with the lowest value have the highest probability of being rolled (2P, 3K, SS). However, that notion differs in expected value with the Flush having a slightly higher EV than 2-Pair, yet the difference between their probabilities is over 22%. The chips with the lowest probability and highest value are 4-of-a-Kind, Large Straight and Yamslam. When looking at expected value, the Large Straight is an outlier, given that it has a higher EV than both 4-of-a-Kind and Full House, yet higher chip value. This is believed to be a result of the simulation only rolling the dice once, as compared to the game where each turn allows three rolls.

Optimal Decision with 3-of-a-Kind

Through the use of probability and expected value, I was able to construct a decision matrix and decision tree to act as a guide for gameplay. This decision template is specific to the scenario in which the player has a 3-of-a-Kind plus two random dice, on any turn. *Figure 34* is a chart of all possible dice combinations when 3-of-a-Kind is present. The colors correspond to a subsequent matrix that details the characteristics and decisions of each situation, located in *Figure 35.*

Figure 34. All possible dice combinations with 3-of-a-Kind being present

Starting Dice: 3-of-a-Kind & (2) Randoms					
[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]
[1,1,1,1,1]	[2,2,2,2,2]	[3,3,3,3,3]	[4,4,4,4,4]	[5,5,5,5,5]	[6,6,6,6,6]
[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]
[1,1,1,1,3]	[2,2,2,2,4]	[3,3,3,3,1]	[4,4,4,4,2]	[5,5,5,5,1]	[6,6,6,6,2]
[1,1,1,1,5]	[2,2,2,2,6]	[3,3,3,3,5]	[4,4,4,4,6]	[5,5,5,5,3]	[6,6,6,6,4]
[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]
[1,1,1,1,2]	[2,2,2,2,1]	[3,3,3,3,2]	[4,4,4,4,1]	[5,5,5,5,2]	[6,6,6,6,1]
[1,1,1,1,4]	[2,2,2,2,3]	[3,3,3,3,4]	[4,4,4,4,3]	[5,5,5,5,4]	[6,6,6,6,3]
[1,1,1,1,6]	[2,2,2,2,5]	[3,3,3,3,6]	[4,4,4,4,5]	[5,5,5,5,6]	[6,6,6,6,5]
[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]	[3K] & [X,X]
[1,1,1,3,3]	[2,2,2,4,4]	[3,3,3,1,1]	[4,4,4,2,2]	[5,5,5,1,1]	[6,6,6,2,2]
[1,1,1,5,5]	[2,2,2,6,6]	[3,3,3,5,5]	[4,4,4,6,6]	[5,5,5,3,3]	[6,6,6,4,4]
[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]
[1,1,1,3,2]	[2,2,2,4,1]	[3,3,3,1,2]	[4,4,4,2,1]	[5,5,5,1,2]	[6,6,6,2,1]
[1,1,1,3,4]	[2,2,2,4,3]	[3,3,3,1,4]	[4,4,4,2,3]	[5,5,5,1,4]	[6,6,6,2,4]
[1,1,1,3,6]	[2,2,2,4,5]	[3,3,3,1,6]	[4,4,4,2,5]	[5,5,5,1,6]	[6,6,6,2,6]
[1,1,1,5,2]	[2,2,2,6,1]	[3,3,3,5,2]	[4,4,4,6,1]	[5,5,5,3,2]	[6,6,6,4,1]
[1,1,1,5,4]	[2,2,2,6,3]	[3,3,3,5,4]	[4,4,4,6,3]	[5,5,5,3,4]	[6,6,6,4,3]
[1,1,1,5,6]	[2,2,2,6,5]	[3,3,3,5,6]	[4,4,4,6,5]	[5,5,5,3,6]	[6,6,6,4,5]
[3K] & [Y,Y]	[3K] & [Y,Y]	[3K] & [Y,Y]	[3K] & [Y,Y]	[3K] & [Y,Y]	[3K] & [Y,Y]
[1,1,1,2,2]	[2,2,2,1,1]	[3,3,3,2,2]	[4,4,4,1,1]	[5,5,5,2,2]	[6,6,6,1,1]
[1,1,1,4,4]	[2,2,2,3,3]	[3,3,3,4,4]	[4,4,4,3,3]	[5,5,5,4,4]	[6,6,6,3,3]
[1,1,1,6,6]	[2,2,2,5,5]	[3,3,3,6,6]	[4,4,4,5,5]	[5,5,5,6,6]	[6,6,6,5,5]
[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]
[1,1,1,3,5]	[2,2,2,4,6]	[3,3,3,1,5]	[4,4,4,2,6]	[5,5,5,1,3]	[6,6,6,2,4]
[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]
----------	[2,2,2,1,3]	[3,3,3,2,4]	[4,4,4,3,5]	[5,5,5,4,6]	----------
[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]	[3K] & [X,Y]
[1,1,1,2,4]	[2,2,2,1,5]	[3,3,3,2,6]	[4,4,4,1,3]	[5,5,5,2,4]	6,6,6,1,3]
[1,1,1,2,6]	[2,2,2,3,5]	[3,3,3,4,6]	[4,4,4,1,5]	[5,5,5,2,6]	[6,6,6,1,5]
[1,1,1,4,6]	----------	----------	----------	----------	[6,6,6,3,5]

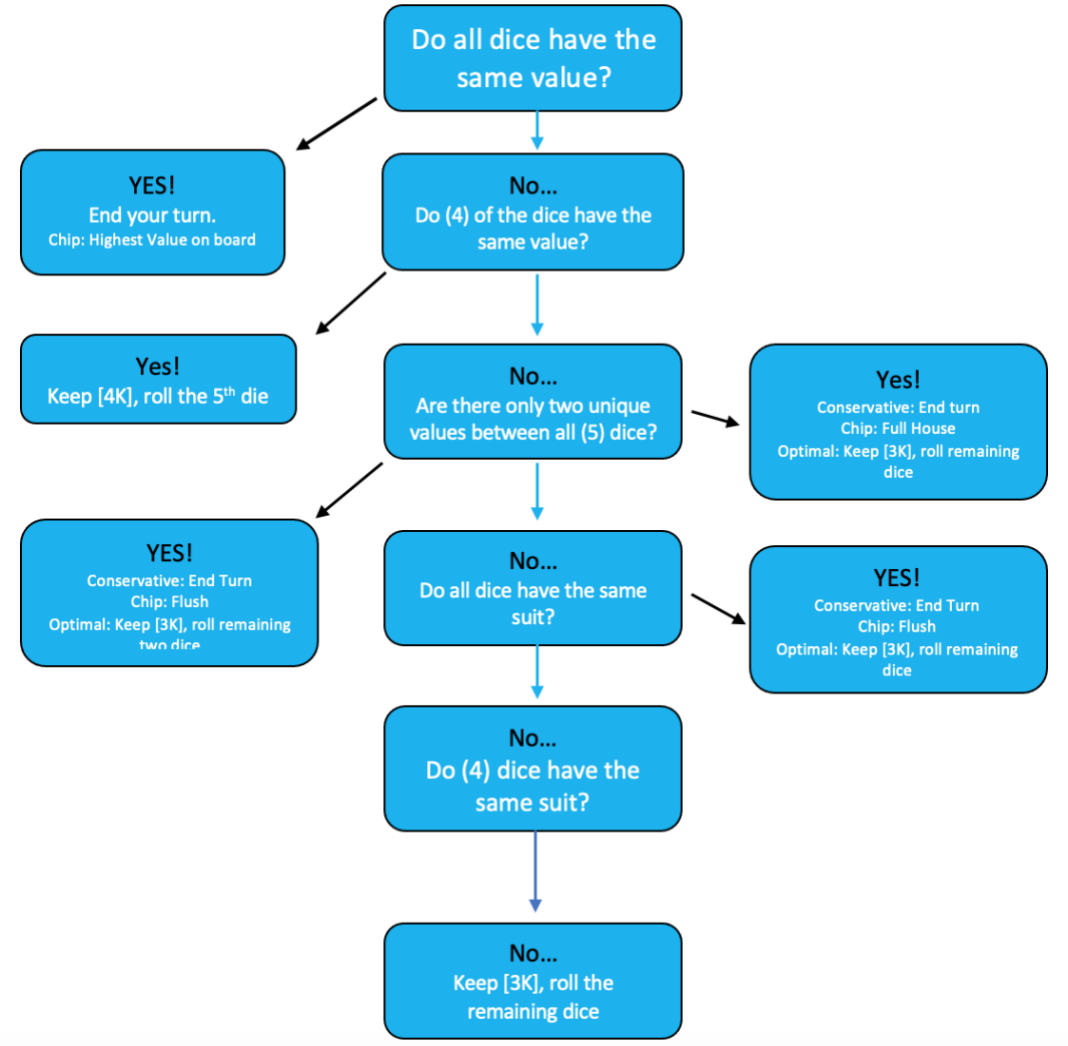
Figure 35. Characteristics and decisions relating to a roll with 3-of-a-Kind present

Current Dice Values	3DS	4FL#s	FL	FH	4K	YS	Example	Optimal Decision	All Decisions	EV	Comments
[3K] & [X,X]	----	YES	YES	YES	YES	YES	[2,2,2,2,2]	Keep All Dice; End Turn	Keep [YS]	50	Optimal roll.
[3K] & [X,Y]	----	YES	YES	----	YES	----	[2,2,2,2,6]	Keep [4K], roll remaining dice	Keep [YS]	48	Strategy factors into decision.
									Keep [FL]c	25	
[3K] & [X,Y]	----	YES	----	----	YES	----	[2,2,2,2,3]	Keep [4K], roll remaining dice	Keep [YS]	48	
[3K] & [X,X]	----	YES	YES	YES	----	----	[2,2,2,4,4]	Keep [3K], roll remaining dice	Keep [3K]	35	The EV's are similar in in value, so the decision may change if the player is more conservative, or is not trying for the highest value chip.
									Keep [FH]c	30	
									Keep [FL]c	25	
[3K] & [X,Y]	----	YES	----	----	----	----	[2,2,2,5,6]	Keep [3K], roll remaining dice	Keep [3K]	35	The EV for keeping either 3K or FL#'s is extremely close. Given the individial player strategy, the decision may change from turn to turn.
									Keep FL#s	34	
[3K] & [Y,Y]	----	------	----	YES	----	----	[2,2,2,5,5]	Keep [3K], roll remaining dice	Keep [3K]	35	Strategy factors into decision.
									Keep [FH]c	30	
[3K] & [X,Y]	----	YES	YES	----	----	----	[2,2,2,4,6]	Keep [3K], roll remaining dice	Keep [3K]	35	Strategy factors into decision.
									Keep [FL]c	25	
[3K] & [X,Y]	YES	------	----	----	----	----	[2,2,2,1,3]	Keep [3K], roll remaining dice	Keep [3K]	35	
									Keep 3DS	≤18	
[3K] & [X,Y]	----	------	----	----	----	----	[2,2,2,1,5]	Keep [3K], roll remaining dice	Keep [3K]	35	

Label Key
3DS = (3) Dice Straight (not a real chip, just a way to categorize certain results)
AOC = All Other Combinations
4FL#s = (4) or more numbers with the same suit
4K = (4) or more numbers with the same value
c = Conservative Strategy

The third iteration of this sequence of dice is a decision tree, located in *Figure 36*. It displays the simulation results in their most simplistic form for use. For the average player, this would be the most user-friendly version of a guide for optimal gameplay.

Figure 36. Decision tree relating to a roll with 3-of-a-Kind present



The chart was constructed using the results found in *Figure 35*. The "yes/no" options pertain directly to the expected values surrounding each set of dice. For any EV's that were close in number, a strategy choice was provided to give the player more information on their choice.

Optimal Decision: [2,2,3,3,4]

Scenarios often arise when a player must choose between what chip they attempt to achieve given the dice roll. Common decision situations that players come across are now made simple with the ability to determine the probability and expected value. Given that Player A rolls [2,2,3,3,4], their primary decisions are as follows:

Keep [2,2], Roll [3,3,4] – Theoretical Goal: YS (#1), 4K (#2), FH (#3), FL (#4), SS (#4), 3K (#5), 2P (#6)

Keep [2,3,4], Roll [2,3] – Theoretical Goal: LS (#1), SS (#2), 3K (#), 2P (#4)

Keep [2,2,4], Roll [3,3] – Theoretical Goal: FL (#1), 4K (#2), FH (#3), SS (#4), 3K (#5), 2P (#6)

Keep [2,2,3,3], Roll [4] – Theoretical Goal: FH (#1), 3K (#2), 2P (#3)

Figure 37. Results matrix from the decision to keep [2,2] and roll [3,3,4]

	#	%	CV	EV
2-Pair	44458	44.458	5	2.2229
3-of-a-Kind	44432	44.432	10	4.4432
Small-Straight	5547	5.547	20	1.1094
Flush	12788	12.788	25	3.197
Full-House	9768	9.768	30	2.9304
4-of-a-Kind	7417	7.417	40	2.9668
Large-Straight	0	0	50	0
YamSlam	452	0.452	50	0.226
|	--	--	--	--
Total_Rolls	1e+05			17.0957

Figure 38. Results matrix from the decision to keep [2,3,4] and roll [2,3]

	#	%	CV	EV
2-Pair	16782	16.782	5	0.8391
3-of-a-Kind	8381	8.381	10	0.8381
Small-Straight	55247	55.247	20	11.0494
Flush	0	0	25	0
Full-House	0	0	30	0
4-of-a-Kind	0	0	40	0
Large-Straight	10975	10.975	50	5.4875
YamSlam	0	0	50	0
|	--	--	--	--
Total_Rolls	1e+05			18.2141

Figure 39. Results matrix from the decision to keep [2,2,4] and roll [3,3]

	#	%	CV	EV
2-Pair	44034	44.034	5	2.2017
3-of-a-Kind	33259	33.259	10	3.3259
Small-Straight	11135	11.135	20	2.227
Flush	24897	24.897	25	6.22425
Full-House	8215	8.215	30	2.4645
4-of-a-Kind	2653	2.653	40	1.0612
Large-Straight	0	0	50	0
YamSlam	0	0	50	0
|	--	--	--	--
Total_Rolls	1e+05			17.50455

Figure 40. Results matrix from the decision to keep [2,2,3,3] and roll [4]

	#	%	CV	EV
2-Pair	1e+05	100	5	5
3-of-a-Kind	33257	33.257	10	3.3257
Small-Straight	0	0	20	0
Flush	0	0	25	0
Full-House	33257	33.257	30	9.9771
4-of-a-Kind	0	0	40	0
Large-Straight	0	0	50	0
YamSlam	0	0	50	0
|	--	--	--	--
Total_Rolls	1e+05			18.3028

EV ≠ 18.3028: EV[3K] will only occur if FH occurs. Therefore, EV[2,2,3,3] = 18.3028 – 3.3257 = **14.9771. *(Figure 38)*

Based upon the data produced from simulating each scenario, **the optimal decision is to keep [2,3,4] and roll [2,3], per empirical data.** Considering that most of the expected values are very close to one another, a conservative verse risk strategy may influence the decision. However, the optimal strategy is also the risk averse choice (highest probability = 55.247). If Player A plays with a high-risk, high-reward strategy, they would be inclined to keep [2,2,3,3] and roll [4], given the highest individual expected value results from that decision (highest EV = 9.9771, excluding 11.0494 because of a low-reward result). Additional dice combination strategies are evaluated and located in the Appendix.

Adjusted Expected Value

As seen in the previous example, the total expected value of rolls may need to be adjusted. There are multiple chip combinations that can be achieved from a single roll. However, only one chip can be taken per roll because that is a rule of the game. One of the limitations to this project is that adjusted EV isn't accounted for in the results table matrix. Therefore, the collective expected value that the simulation generates must be manually adjusted for certain scenarios. E.g., Player A decides to keep [2,2,2,6] and roll the fifth die. See *Figure 41* for a results table.

Figure 41. Results matrix from the decision to keep [2,2,2,6] and roll remaining die

	#	%	CV	EV
2-Pair	33343	33.343	5	1.66715
3-of-a-Kind	1e+05	100	10	10
Small-Straight	0	0	20	0
Flush	49925	49.925	25	12.48125
Full-House	16586	16.586	30	4.9758
4-of-a-Kind	16757	16.757	40	6.7028
Large-Straight	0	0	50	0
YamSlam	0	0	50	0
|	--	--	--	--
Total	1e+05			35.827

> |

If a Full House is achieved in the final roll, by definition, a 2-Pair is as well. Under no rational strategy would Player A prefer the 2-Pair over the Full House (assuming that all chips are available to pick from). In addition, Player A must only pick one chip, so the expected value of 2-Pair can be subtracted from the collective EV of that specific simulation.

35.827 – 1.66715 = **34.15985**

Limitations

As previously explained, the total expected value of specific rolls must be adjusted upon completion of the simulation. This is due to the conflicting structures between the true gameplay and code script. The code recognizes each combination individually and technically accounts for more than one chip to be taken per turn. The gameplay only allows for one chip to be taken per turn, thereby decreasing the total expected value. While this limitation is correctable, it decreases the autonomy of the entire simulation. With more practice in R, I'm confident in the ability to incorporate a solution into the code script.

Another limitation to the project is the ability to simulate every possible component of the game in code. What this refers to are the minority rules/aspects that contribute to the game, such as bonus scoring, turn taking and total chips per person. Bonus scoring is an important aspect that relates to EV, because as discussed previously, the incentive of a bonus may change a player's strategy in a given situation. Bonus scoring can be incorporated into total expected value manually, but not automatically in the simulation. The reason for that brings me to my next point, which is turn taking. The code is created to run single combination simulations, therefore, to simulate turns in a row, the code must be manually re-run. This does not affect the data results, however it limits one's ability to simulate an entire, three-roll turn automatically. That is why

bonuses are not incorporated into the code, otherwise it would skew the results. The third component of coding limitations is not having a chip counter. In the game, players keep track of their own score based on what chips they have taken. A table would be simple to create and track combinations, however due to the same reason as the previous two coding limitations, player turns aren't tracked so the need for a counter in this specific project is not yet necessary.

Conclusion

The inspiration behind simulating Yamslam grew out of curiosity. Given the possibility of gaining an advantage over an opponent using statistics was intriguing and became a basis for investigation. To conduct thorough research, I created a code that simulated a simplistic version of the game, tabulated the results, and produced a matrix that displayed the probability and expected value of the dice in question. After continuous testing, I determined that the code was in fact accurate, which allowed me to start documenting the optimal strategies in the game.

The scenario most analyzed contained three dice of the same value, with the other two being completely random [X,X,X,Y,Z]. I considered every possible outcome of the two remaining dice and created a matrix that displayed the results, subsequent strategies, and a simplistic display of the decisions (decision tree). The results indicate that when 3-of-a-Kind is present in any roll, the optimal decision is to keep those dice that are the same and roll the remaining two dice. In the case that one or both of the two remaining dice have the same value as the set in 3-of-a-Kind, then the strategy would be to keep the four or five of a kind present. To summarize, when at least 3-of-a-Kind is present in a roll, the optimal strategy is to keep all matching dice and roll the remainders. Other dice scenarios were simulated, with results attached in the appendix.

The two components that dictate a dice game are chance and strategy. Given that players are unable to control chance, they look to strategy to maximize their chances of winning against their opponents. Strategy is exactly what was tested and developed in this project. It is the single most important tool that a player possess'. With the ability to know the probability and expected value that results from each individual roll, players that use this method should theoretically maximize their chances of scoring the highest number of points possible, with respect to chance.

Building on my research, there are two primary components that can improve the functionality of the simulation. First and foremost, the total expected value script should be altered so that there is less manual adjustment involved. Overlapping combinations affect the generated total EV, so in an effort to automate the process, those scenarios would be adjusted. The second improvement would include the ability to simulate three rolls consecutively, and then remove a chip from the pool. This mimics a single turn and cuts down on manual involvement. It would help streamline the addition of bonuses and refine strategies. In keeping with the theme, improvements revolve around decreasing manual work, while increasing automation.

Overall, I consider the project a success. I have learned to simulate any combination of dice an finite number of times, and generate useful metrics that influence strategy and decision-making.

References

Litwiller, B. H., & Duncan, D. R. (1979). The probability of winning dice games. *The Mathematics Teacher*, 72(6), 458-461. http://www.jstor.org/stable/27961716

Maynard, Moss, P., Whitehead, M., Narayanan, S., Garay, M., Brannon, N., Kantemneni, R. G., & Kustra, T. (2001). Modeling expert problem solving in a game of chance: a Yahtzee© case study. Expert Systems, 18(2), 88-98. https://doi.org/10.1111/1468-0394.00160

Yamslam Instructions: Blue Orange Games http://www.blueorangegames.com/administrator/components/com_bogames/files/rules/yamslam-rules.pdf

Appendix

Appendix Figure A: Equation vs. Simulation Based Probabilities

	Probability	
Combination	**Equation**	**Simulation**
2 Pair	28.94%	29.04%
3 of a Kind	21.30%	21.31%
Small Straight	15.43%	15.45%
Flush	6.25%	6.26%
Full House	3.86%	3.95%
4 of a Kind	1.93%	2.00%
Large Straight	3.09%	3.10%
YamSlam	0.08%	0.08%
*5,000,000 Simulations		
*Probabilities rounded to two decimal places		

Appendix Figure B: Constant Value Matrix Scripts

```
#Yamslam_1Constant_Dice_Sim

n <- 100000
roll <- matrix(, nrow = n, ncol = 5)
# Run "Roll" to see the above function output

a <- 1
for (k in 1:n) {
  b <- sample(x = 1:6, size = 1, replace = TRUE)
  c <- sample(x = 1:6, size = 1, replace = TRUE)
  d <- sample(x = 1:6, size = 1, replace = TRUE)
  e <- sample(x = 1:6, size = 1, replace = TRUE)
  Constant_Vect <- c(a,b,c,d,e)
  roll[k,] <- Constant_Vect
}
# Run "Roll" to see the above function output
```

(1) Constant, (4) Randoms Sim Code

```
#Yamslam_2Constant_Dice_Sim

n <- 100000
roll <- matrix(, nrow = n, ncol = 5)
# Run "Roll" to see the above function output

a <- 2
b <- 2
for (k in 1:n) {
  c <- sample(x = 1:6, size = 1, replace = TRUE)
  d <- sample(x = 1:6, size = 1, replace = TRUE)
  e <- sample(x = 1:6, size = 1, replace = TRUE)
  Constant_Vect <- c(a,b,c,d,e)
  roll[k,] <- Constant_Vect
}
# Run "Roll" to see the above function output
```

(2) Constant, (3) Randoms Sim Code

```
#Yamslam_3Constant_Dice_Sim

n <- 100000
roll <- matrix(, nrow = n, ncol = 5)
# Run "Roll" to see the above function output

a <- 4
b <- 5
c <- 6
for (k in 1:n) {
  d <- sample(x = 1:6, size = 1, replace = TRUE)
  e <- sample(x = 1:6, size = 1, replace = TRUE)
  Constant_Vect <- c(a,b,c,d,e)
  roll[k,] <- Constant_Vect
}
# Run "Roll" to see the above function output
```

(3) Constant, (2) Randoms Sim Code

```
#Yamslam_4Constant_Dice_Sim

n <- 100000
roll <- matrix(, nrow = n, ncol = 5)
# Run "Roll" to see the above function output

a <- 5
b <- 2
c <- 3
d <- 4
for (k in 1:n) {
  e <- sample(x = 1:6, size = 1, replace = TRUE)
  Constant_Vect <- c(a,b,c,d,e)
  roll[k,] <- Constant_Vect
}
# Run "Roll" to see the above function output
```

(4) Constant, (1) Randoms Sim Code

Appendix Figure C: RStudio Code Script

```r
## Yamslam_All_Random_Dice_Sim

n <- 100000
roll <- matrix(, nrow = n, ncol = 5)
# Run "Roll" to view empty matrix created

for (k in 1:n) {
  roll[k,] <- sample(x = 1:6, size = 5, replace = TRUE)
}

combinations <- matrix(0, n, 8)
combinations <- data.frame(combinations)
colnames(combinations) <- c("2P", "3K", "SS", "FL", "FH", "4K", "LS", "YS")
# Run "combinations" to view matrix with labels

for (k in 1:n) {
  x <- roll[k,]
  sort(x)

#TESTING for 2-Pair
  if (length(unique(x[duplicated(x)])) == 2){
    combinations[k,1] <- 1
  }


#TESTING for 3-of-a-Kind
if(length(unique(x)) == 3 && length(unique(x[duplicated(x)])) == 1) {
  combinations[k,2] <- 1
}


#TESTING for Small Straight
  x <- sort(x)
  z <- unique(x)
  v <- rep(0,3)
  f <- c(1,3,4,5,6)
  g <- c(1,2,3,4,6)

  for (i in 1:3) {
    if (length(z) == 4 && z[5-i] - z[4-i] == 1){v[i] <- 1}
  }
  if (sum(v) == 3){
    combinations[k,3] <- 1
  }

  if (all(x == f) == TRUE || all(x == g) == TRUE){
    combinations[k,3] <- 1
  }


#TESTING for Flush
if(length(unique(x%%2)) == 1) {
  combinations[k,4] <- 1
}

#TESTING for Full House
if(length(unique(x)) == 2 &&
   length(unique(x[duplicated(x)])) == 2) {
  combinations[k,1] <- 1
  combinations[k,2] <- 1
  combinations[k,5] <- 1
}
```

```
}

#TESTING for 4-of-a-Kind
if(length(unique(x)) == 2 &&
   length(unique(x[duplicated(x)])) == 1) {
  combinations[k,1] <- 1
  combinations[k,2] <- 1
  combinations[k,6] <- 1
}


#TESTING for Large Straight
x <- sort(x)
y <- rep(0,4)
for (i in 1:4) {
  if (x[6-i] - x[5-i] == 1){y[i] <- 1}
}
if (sum(y) == 4) {
  combinations[k,3] <- 1
  combinations[k,7] <- 1
}

#TESTING for Yamslam
if(length(unique(x)) == 1) {
  combinations[k,1] <- 1
  combinations[k,2] <- 1
  combinations[k,5] <- 1
  combinations[k,6] <- 1
  combinations[k,8] <- 1
}
}


Chip_Prob <- matrix(, nrow = 10, ncol = 4)
Chip_Prob <- data.frame(Chip_Prob)
colnames(Chip_Prob) <- c("#", "%", "CV" , "EV")
rownames(Chip_Prob) <- c("2-Pair", "3-of-a-Kind", "Small-Straight", "Flush",
                         "Full-House", "4-of-a-Kind", "Large-Straight",
                          "YamSlam" , "|" , "Total")
#Run "Chip_Prob" to show blank matrix with labels

Chip_Prob[1,1] <- (sum(combinations[,1])) # Number of combos logged
Chip_Prob[2,1] <- (sum(combinations[,2]))
Chip_Prob[3,1] <- (sum(combinations[,3]))
Chip_Prob[4,1] <- (sum(combinations[,4]))
Chip_Prob[5,1] <- (sum(combinations[,5]))
Chip_Prob[6,1] <- (sum(combinations[,6]))
Chip_Prob[7,1] <- (sum(combinations[,7]))
Chip_Prob[8,1] <- (sum(combinations[,8]))
Chip_Prob[9,1] <- "__"
Chip_Prob[10,1] <- n

Chip_Prob[1,2] <- (sum(combinations[,1])/n)*100 # Probability of combos logged
Chip_Prob[2,2] <- (sum(combinations[,2])/n)*100
Chip_Prob[3,2] <- (sum(combinations[,3])/n)*100
Chip_Prob[4,2] <- (sum(combinations[,4])/n)*100
Chip_Prob[5,2] <- (sum(combinations[,5])/n)*100
Chip_Prob[6,2] <- (sum(combinations[,6])/n)*100
Chip_Prob[7,2] <- (sum(combinations[,7])/n)*100
Chip_Prob[8,2] <- (sum(combinations[,8])/n)*100
Chip_Prob[9,2] <- "__"
Chip_Prob[10,2] <- ""
```

```
125
126 Chip_Prob[1,3] <- "5" # Chip Values in chart
127 Chip_Prob[2,3] <- "10"
128 Chip_Prob[3,3] <- "20"
129 Chip_Prob[4,3] <- "25"
130 Chip_Prob[5,3] <- "30"
131 Chip_Prob[6,3] <- "40"
132 Chip_Prob[7,3] <- "50"
133 Chip_Prob[8,3] <- "50"
134 Chip_Prob[9,3] <- "__"
135 Chip_Prob[10,3] <- " "
136
137 Chip_Prob[1,4] <- ((sum(combinations[,1]))/n)*5 # EV of combos logged
138 Chip_Prob[2,4] <- ((sum(combinations[,2]))/n)*10
139 Chip_Prob[3,4] <- ((sum(combinations[,3]))/n)*20
140 Chip_Prob[4,4] <- ((sum(combinations[,4]))/n)*25
141 Chip_Prob[5,4] <- ((sum(combinations[,5]))/n)*30
142 Chip_Prob[6,4] <- ((sum(combinations[,6]))/n)*40
143 Chip_Prob[7,4] <- ((sum(combinations[,7]))/n)*50
144 Chip_Prob[8,4] <- ((sum(combinations[,8]))/n)*50
145 Chip_Prob[9,4] <- "__"
146
147 EV_vector <- c(Chip_Prob[1:8,4])
148 EV_vector <- as.numeric(EV_vector)
149 EVsum <- sum(EV_vector) # Sum EV's
150 Chip_Prob[10,4] <- EVsum
151
152 roll #Produce a matrix of dice rolls
153 combinations #Produce a matrix of combos logged
154 Chip_Prob #Produce a table with metrics pertaining to results of simulations
155 write.csv(Chip_Prob, file = "book2.csv")
```

Appendix Figure D: [roll] Output

	V1	V2	V3	V4	V5
1	2	3	4	6	3
2	6	6	6	6	5
3	3	2	2	1	5
4	1	1	5	1	5
5	3	5	1	3	4
6	6	4	6	2	4
7	2	4	3	4	1
8	4	4	1	3	5
9	3	3	6	1	5
10	3	2	5	2	2
11	3	2	4	4	4
12	6	2	5	3	1
13	6	1	3	1	3
14	6	5	6	3	3
15	3	2	5	2	4
16	5	3	3	4	1
17	2	5	1	5	1
18	1	4	2	4	1
19	6	3	5	2	5
20	1	4	6	5	5
21	6	4	3	2	2
22	4	6	5	6	3
23	6	2	3	5	5
24	4	1	3	2	1
25	3	4	2	4	2
26	2	6	6	2	1
27	4	4	3	6	5
28	2	6	4	6	3
29	1	4	4	3	1
30	2	6	6	3	1
31	6	3	2	4	1
32	5	6	6	5	5
33	6	2	5	1	5
34	5	4	3	2	6
35	2	3	3	6	6
36	4	2	5	5	2
37	6	4	6	4	1
38	3	1	2	1	1
39	6	6	6	6	6
40	4	5	4	3	4
41	1	6	1	1	2
42	5	5	4	4	4
43	6	1	2	5	1
44	2	1	4	4	5
45	3	1	4	2	1
46	1	4	4	6	6
47	1	6	3	2	3
48	5	5	3	1	2
49	4	5	4	4	2
50	6	2	5	3	4
51	4	4	3	4	4
52	1	2	6	4	1
53	4	4	1	2	5
54	6	1	1	3	6
55	6	4	1	2	1
56	1	2	4	1	4
57	4	3	5	6	2
58	3	3	2	1	5
59	6	1	4	3	1
60	4	4	4	4	4
61	4	1	5	5	4
62	2	6	6	4	3
63	2	4	2	6	3
64	4	5	5	6	1
65	6	4	2	1	6
66	4	6	4	6	5
67	1	6	1	2	3
68	5	2	6	5	6
69	2	5	2	4	3
70	5	2	2	5	2
71	1	1	3	5	5
72	2	3	6	4	2
73	2	5	4	5	5
74	3	3	3	2	1
75	4	3	5	4	6

	V1	V2	V3	V4	V5
76	6	2	2	1	1
77	1	4	2	2	2
78	3	1	6	1	3
79	6	5	6	5	4
80	5	4	1	1	1
81	4	5	5	1	2
82	3	4	1	6	4
83	4	2	4	4	6
84	2	2	1	3	6
85	1	5	2	4	2
86	4	6	6	1	5
87	4	6	6	2	1
88	1	6	4	6	5
89	3	5	5	4	6
90	1	4	4	1	2
91	1	1	6	5	1
92	2	1	2	2	5
93	4	4	6	2	6
94	6	5	2	6	6
95	6	4	1	2	4
96	4	3	1	5	3
97	6	5	2	2	6
98	6	6	5	5	6
99	5	1	5	3	2
100	6	2	6	6	1
101	4	3	1	5	4
102	2	3	2	3	2
103	3	2	2	1	4
104	1	4	6	5	6
105	2	3	5	4	5
106	2	5	3	3	2
107	3	6	5	1	5
108	1	4	5	1	3
109	6	2	6	6	4
110	1	3	1	1	4
111	4	4	6	6	5
112	5	2	1	5	1
113	5	2	2	4	4
114	6	4	5	2	2
115	6	6	1	4	1
116	1	5	4	2	3
117	1	5	2	1	4
118	5	1	6	3	4
119	4	6	6	2	5
120	5	3	1	2	1
121	2	3	1	1	1
122	1	3	2	5	4
123	1	6	1	4	3
124	4	3	3	4	6
125	3	1	1	4	3
126	3	2	1	4	4
127	4	3	1	6	1
128	6	3	5	3	1
129	2	6	5	2	2
130	5	6	5	3	3
131	1	2	3	1	5
132	3	3	3	5	2
133	2	1	3	6	5
134	5	3	2	2	6
135	6	2	1	6	2
136	5	5	6	1	6
137	4	3	4	3	6
138	3	3	3	5	4
139	3	1	6	6	2
140	2	3	2	5	6
141	2	5	5	4	3
142	1	1	6	2	2
143	6	4	3	1	6
144	2	2	5	2	5
145	2	6	1	6	2
146	5	1	3	4	3
147	5	1	5	1	4
148	4	2	1	2	6
149	1	1	1	1	3
150	5	4	6	2	4

Appendix Figure E: [combinations] Output

	2P	3K	SS	FL	FH	4K	LS	YS
1	0	0	0	0	0	0	0	0
2	1	1	0	0	0	1	0	0
3	0	0	0	0	0	0	0	0
4	1	1	0	1	1	0	0	0
5	0	0	0	0	0	0	0	0
6	1	0	0	1	0	0	0	0
7	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0
10	0	1	0	0	0	0	0	0
11	0	1	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0
13	1	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0
15	0	0	1	0	0	0	0	0
16	0	0	0	0	0	0	0	0
17	1	0	0	0	0	0	0	0
18	1	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0
22	0	0	1	0	0	0	0	0
23	0	0	0	0	0	0	0	0
24	0	0	1	0	0	0	0	0
25	1	0	0	0	0	0	0	0
26	1	0	0	0	0	0	0	0
27	0	0	1	0	0	0	0	0
28	0	0	0	0	0	0	0	0
29	1	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	0	0	1	0	0	0	0	0
32	1	1	0	0	1	0	0	0
33	0	0	0	0	0	0	0	0
34	0	0	1	0	0	0	1	0
35	1	0	0	0	0	0	0	0
36	1	0	0	0	0	0	0	0
37	1	0	0	0	0	0	0	0
38	0	1	0	0	0	0	0	0
39	1	1	0	1	1	1	0	1
40	0	1	0	0	0	0	0	0
41	0	1	0	0	0	0	0	0
42	1	1	0	0	1	0	0	0
43	0	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0	0
45	0	0	1	0	0	0	0	0
46	1	0	0	0	0	0	0	0
47	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
49	0	1	0	0	0	0	0	0
50	0	0	1	0	0	0	1	0
51	1	1	0	0	0	1	0	0
52	0	0	0	0	0	0	0	0
53	0	0	0	0	0	0	0	0
54	1	0	0	0	0	0	0	0
55	0	0	0	0	0	0	0	0
56	1	0	0	0	0	0	0	0
57	0	0	1	0	0	0	1	0
58	0	0	0	0	0	0	0	0
59	0	0	0	0	0	0	0	0
60	1	1	0	1	1	1	0	1
61	1	0	0	0	0	0	0	0
62	0	0	0	0	0	0	0	0
63	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0
65	0	0	0	0	0	0	0	0
66	1	0	0	0	0	0	0	0
67	0	0	0	0	0	0	0	0
68	1	0	0	0	0	0	0	0
69	0	0	1	0	0	0	0	0
70	1	1	0	0	1	0	0	0
71	1	0	0	1	0	0	0	0
72	0	0	0	0	0	0	0	0
73	0	1	0	0	0	0	0	0
74	0	1	0	0	0	0	0	0
75	0	0	1	0	0	0	0	0

	2P	3K	SS	FL	FH	4K	LS	YS
76	1	0	0	0	0	0	0	0
77	0	1	0	0	0	0	0	0
78	1	0	0	0	0	0	0	0
79	1	0	0	0	0	0	0	0
80	0	1	0	0	0	0	0	0
81	0	0	0	0	0	0	0	0
82	0	0	0	0	0	0	0	0
83	0	1	0	1	0	0	0	0
84	0	0	0	0	0	0	0	0
85	0	0	0	0	0	0	0	0
86	0	0	0	0	0	0	0	0
87	0	0	0	0	0	0	0	0
88	0	0	0	0	0	0	0	0
89	0	0	1	0	0	0	0	0
90	1	0	0	0	0	0	0	0
91	0	1	0	0	0	0	0	0
92	0	1	0	0	0	0	0	0
93	1	0	0	1	0	0	0	0
94	0	1	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0
97	1	0	0	0	0	0	0	0
98	1	1	0	0	1	0	0	0
99	0	0	0	0	0	0	0	0
100	0	1	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
102	1	1	0	0	1	0	0	0
103	0	0	1	0	0	0	0	0
104	0	0	0	0	0	0	0	0
105	0	0	1	0	0	0	0	0
106	1	0	0	0	0	0	0	0
107	0	0	0	0	0	0	0	0
108	0	0	0	0	0	0	0	0
109	0	1	0	1	0	0	0	0
110	0	1	0	0	0	0	0	0
111	1	0	0	0	0	0	0	0
112	1	0	0	0	0	0	0	0
113	1	0	0	0	0	0	0	0
114	0	0	0	0	0	0	0	0
115	1	0	0	0	0	0	0	0
116	0	0	1	0	0	0	1	0
117	0	0	0	0	0	0	0	0
118	0	0	1	0	0	0	0	0
119	0	0	0	0	0	0	0	0
120	0	0	0	0	0	0	0	0
121	0	1	0	0	0	0	0	0
122	0	0	1	0	0	0	1	0
123	0	0	0	0	0	0	0	0
124	1	0	0	0	0	0	0	0
125	1	0	0	0	0	0	0	0
126	0	0	1	0	0	0	0	0
127	0	0	0	0	0	0	0	0
128	0	0	0	0	0	0	0	0
129	0	1	0	0	0	0	0	0
130	1	0	0	0	0	0	0	0
131	0	0	0	0	0	0	0	0
132	0	1	0	0	0	0	0	0
133	0	0	0	0	0	0	0	0
134	0	0	0	0	0	0	0	0
135	1	0	0	0	0	0	0	0
136	1	0	0	0	0	0	0	0
137	1	0	0	0	0	0	0	0
138	0	1	0	0	0	0	0	0
139	0	0	0	0	0	0	0	0
140	0	0	0	0	0	0	0	0
141	0	0	1	0	0	0	0	0
142	1	0	0	0	0	0	0	0
143	0	0	0	0	0	0	0	0
144	1	1	0	0	1	0	0	0
145	1	0	0	0	0	0	0	0
146	0	0	0	0	0	0	0	0
147	1	0	0	0	0	0	0	0
148	0	0	0	0	0	0	0	0
149	1	1	0	1	0	1	0	0
150	0	0	0	0	0	0	0	0

Appendix Figure F: Results of [1,2,2,4,5] Strategy Simulation

Keep [1,2,4,5], Roll [2]

	#	%	CV	EV
2-Pair	0	0	5	0
3-of-a-Kind	0	0	10	0
Small-Straight	16644	16.644	20	3.3288
Flush	0	0	25	0
Full-House	0	0	30	0
4-of-a-Kind	0	0	40	0
Large-Straight	16644	16.644	50	8.322
YamSlam	0	0	50	0
|	--	--	--	--
Total	1e+05			11.6508

> |

EV ≠ 11.6508: EV[SS] will only occur if LS occurs. Therefore, EV[1,2,4,5] = 11.6508 – 3.3288 = **8.322.

Keep [2,2], Roll [1,4,5]

	#	%	CV	EV
2-Pair	44508	44.508	5	2.2254
3-of-a-Kind	44485	44.485	10	4.4485
Small-Straight	5520	5.52	20	1.104
Flush	12411	12.411	25	3.10275
Full-House	9804	9.804	30	2.9412
4-of-a-Kind	7345	7.345	40	2.938
Large-Straight	0	0	50	0
YamSlam	451	0.451	50	0.2255
|	--	--	--	--
Total	1e+05			16.98535

>

Keep [2,2,4], Roll [1,5]

	#	%	CV	EV
2-Pair	44477	44.477	5	2.22385
3-of-a-Kind	33394	33.394	10	3.3394
Small-Straight	11227	11.227	20	2.2454
Flush	25246	25.246	25	6.3115
Full-House	8350	8.35	30	2.505
4-of-a-Kind	2788	2.788	40	1.1152
Large-Straight	0	0	50	0
YamSlam	0	0	50	0
|	--	--	--	--
Total	1e+05			17.74035

> |

Keep [2,4,5], Roll [1,2]

	#	%	CV	EV
2-Pair	16677	16.677	5	0.83385
3-of-a-Kind	8322	8.322	10	0.8322
Small-Straight	30337	30.337	20	6.0674
Flush	0	0	25	0
Full-House	0	0	30	0
4-of-a-Kind	0	0	40	0
Large-Straight	10879	10.879	50	5.4395
YamSlam	0	0	50	0
|	--	--	--	--
Total	1e+05			13.17295

> |

Optimal Choice: Keep [2,2,4], roll [1,5] yields **EV = 17.74035**

Risk Averse: Keep [2,2], roll [1,4,5] yields highest probability chips **= 44.508 (2P) & 44.485 (3K)**

Risk Tolerant: Keep [1,2,4,5], roll [2] yields highest **EV, = 8.322**

Appendix Figure G: Results of [1,2,4,5,6] Strategy Simulation

Keep [1,2,4,5] or [2,4,5,6], Roll [2] or[1]

	#	%	CV	EV
2-Pair	0	0	5	0
3-of-a-Kind	0	0	10	0
Small-Straight	16686	16.686	20	3.3372
Flush	0	0	25	0
Full-House	0	0	30	0
4-of-a-Kind	0	0	40	0
Large-Straight	16686	16.686	50	8.343
YamSlam	0	0	50	0
|	--	--	--	--
Total	1e+05			11.6802

Keep [1,2,4] or [4,5,6], Roll [5,6] or [1,2]

	#	%	CV	EV
2-Pair	16850	16.85	5	0.8425
3-of-a-Kind	8206	8.206	10	0.8206
Small-Straight	30605	30.605	20	6.121
Flush	0	0	25	0
Full-House	0	0	30	0
4-of-a-Kind	0	0	40	0
Large-Straight	5584	5.584	50	2.792
YamSlam	0	0	50	0
|	--	--	--	--
Total	1e+05			10.5761

EV ≠ 11.6802: EV[SS] will only occur if LS occurs. Therefore, EV[1,2,4,5] or EV[2,4,5,6] = 11.6802 – 3.3372 = **8.343.

Keep [X], Roll remaining (4)

	#	%	CV	EV
2-Pair	28909	28.909	5	1.44545
3-of-a-Kind	21543	21.543	10	2.1543
Small-Straight	10171	10.171	20	2.0342
Flush	6254	6.254	25	1.5635
Full-House	3772	3.772	30	1.1316
4-of-a-Kind	2108	2.108	40	0.8432
Large-Straight	1848	1.848	50	0.924
YamSlam	81	0.081	50	0.0405
|	--	--	--	--
Total	1e+05	1e+05		10.13675

Optimal Choice: Either keep [X] or keep [1,2,4]/[4,5,6]; Total EV's are within 0.5 of each other

Risk Averse: Keep [1,2,4]/[4,5,6], & roll [5,6]/[1,2] yields highest probability correlated with highest EV, = **30.605% to 6.121**

Risk Tolerant: Keep [1,2,4,5]/[2,4,5,6] & roll [6]/[1] yields highest **EV, = 8.343**