
Yahtzee Strategy Performance Evaluation

Brad Hill
ISYE 6644 – Fall 2022

I. Abstract

Yahtzee is a dice game that can be played alone as a solitaire-style game or against others. In this study, we explain the rules and scoring methodologies used in Yahtzee, create a toolset for simulating solitaire Yahtzee games and building reroll/scoring strategies, and analyze and compare a few simulated strategies. We use paired-t confidence intervals to evaluate the mean differences between our top two strategies (determined by average game score) to find that, despite not meeting the heights of Verhoeff's Optimal Strategy, our brute force strategy is the best included strategy for Yahtzee. That said, this is a difficult strategy for a human to carry out in real time, so a human-friendly best strategy is also selected and discussed.

II. Background

i. Purpose

The purpose of this project is to develop a series of functions that allows simulation of the dice game Yahtzee and evaluate several strategies using simulation. The functions must adhere to the rules of the game (II.iii) and must be flexible enough to use them in a number of ways depending on the strategy employed.

The strategies employed in this paper are not all “optimal” strategies. In fact, many of the strategies used throughout this work are inherently suboptimal nearly by definition. The purpose here is both the development of the simulation functionality and the simulation and comparison of multiple strategies. That said, one of the last strategies we employ is a sort of brute force “optimal” strategy, but due to time constraints this strategy has not been optimized for fast simulation runs, clocking in at about 14 seconds per game. For those interested in a more thorough attempt at the optimal strategy, Tom Verhoeff at the Eindhoven University of Technology has defined what he believes to be the “Optimal” Yahtzee strategy in a draft of a paper entitled How To Maximize Your Score in Solitaire Yahtzee^[1].

ii. Organization

This paper is intended to be laid out in an intuitive and easy to follow way, ideally answering questions as they arise for the reader. We have already discussed the motivation behind this paper, (II.i) but we follow this section with an introduction to the rules and objectives of Yahtzee (II.iii) and a detailed rundown of how scoring works in the game. (II.iv) This paper spends a considerable amount of time explaining the scoring mechanics of Yahtzee as this is a fundamental piece of the development (III) of the code used to simulate the game.

After discussing the code development for the game, we will discuss how to build a strategy, implement a number of different strategies (IV) and evaluate the results of their simulations, followed by a discussion of our conclusions (V). Additional figures and code can be found in the included zip file, and, while this report does not necessitate a large amount of literature review, citations can be found at the end of the document.

iii. Yahtzee Rules and Objectives

Yahtzee is a dice game, the objective of which is to score the most points given a set of 5 dice and a scoring template. The rules^[2] stipulate that a player may keep or reroll any number of dice up to two additional times for a total of three rolls per turn. At the end of their turn, players must score the dice using an appropriate scoring scheme. The scorecard used in Yahtzee is broken into two sections, the Upper Section, containing sums of specific dice values, and the Lower Section, containing specific dice patterns such as a straight or a full house, each with their own set point total.

Yahtzee may be played with multiple people, in which case the objective is to achieve more points than the rest of the opponents, or alone, with the objective being to achieve a personal high score. While most Yahtzee strategy evaluation (including this one) limit themselves to solitaire-style Yahtzee, it is worth noting that strategies change drastically given additional players. Playing against an opponent changes the objective from maximizing score to simply outscoring the other player, causing certain scoring decisions that may not maximize score but could maximize win percentage/likelihood. These are situations that are interesting to consider and work on but are outside the scope of this paper.

iv. Yahtzee Scoring

The Upper Section

As mentioned previously, the scorecard used in Yahtzee consists of two parts, the Upper Section and the Lower Section. The Upper Section contains six categories, one for each corresponding die value, Aces to Sixes. Scoring in these categories involves taking the sum of *only* the matching dice values. For instance, if a player rolls 6-6-5-6-3, scoring this roll in the Sixes category would award the player 18 points, the sum of the dice showing 6. The Upper Section also awards a bonus, known as the Upper Section Bonus, if the sum of the Upper Section is at least 63. This is the equivalent of rolling three of the appropriate values in each category. The Upper Section Bonus is an additional 35 points.

The Lower Section

The Lower Section consists of 7 scoring schemes:

- Three of a Kind: Sum of all dice if at least 3 dice match one another
- Four of a Kind: Sum of all dice if at least 4 dice match one another
- Full House: 25 if 3 dice match each other and the remaining 2 match each other
- Small Straight: 30 if there is a sequential roll of 4 dice (i.e., [1-2-3-4]-6)
- Large Straight: 40 if there is a sequential roll of 5 dice (i.e., 2-3-4-5-6)
- Yahtzee: 50 if all 5 dice match one another
- Chance: Sum of all dice

The official rules also use what is known as the “forced joker rule”. Under this rule, rolling a Yahtzee when the player has already scored a Yahtzee awards an additional 100 point

Yahtzee Bonus, and forces the player to score that Yahtzee in the corresponding Upper Section category. For instance, a Yahtzee of 4s would be scored in the Fours category if a 50 has already been scored in the Yahtzee category. If the corresponding Upper Section category has already been used, or “blanked” (given a score of 0), the player may score the Yahtzee in one of the available Lower Section categories, even if the Yahtzee does not match the pattern. If the player has already blanked the Yahtzee category and rolls a Yahtzee, the joker rules still apply, but no Yahtzee bonus is awarded.

III. Development

i. Introduction

The development of this project was done in R, utilizing its many statistical packages as well as data manipulation packages such as `tidyr` and `dplyr`, commonly referred to as the `tidyverse`. Functions built for this project are sorted into four categories, Dice Functions, Scoring Functions, Gamesim Functions, and Strategy Analysis Functions. A fifth category, Brute Force Functions, is included and will be discussed more in the corresponding strategy section.

ii. Dice Functions

The most fundamental piece of Yahtzee is, of course, the dice roll. While there are numerous ways to implement an even 6-sided die roll, the simplest is to use R’s built-in `sample()` function, taking 5 samples (with replacement) of the integers 1 to 6. This is a very fast and efficient method. To allow for the flexibility needed for rerolling certain dice, an argument, `n`, is used to define the number of dice to roll. This is further used in the `reroll()` function, in which users can define which dice to keep and which to reroll using the `keep` argument.

With the dice functions, I also include a bevy of helper functions designed to aid in the building of a strategy. In the code, I have labeled these as “Keep Helpers”. Some examples here include `keep_pips()`, which keeps a certain dice value, `keep_top_n()`, which returns the top `n` dice by value, and `keep_mode()`, which keeps dice that have most matches in a given roll. These functions all return the index of the dice in a provided roll, so to keep these dice, you would pass the results of these functions to the `reroll()` function with your current roll. For certain keep helpers, we also designed “Reroll Helpers” that work in tandem without having to pass the keep helper explicitly.

The last category of dice function is “Check Functions.” These are a series of functions that check the provided roll for a certain number of consecutive values. The logic is subtly different in each one as certain functions have to account for instances in which the consecutive values include a duplicate die somewhere in the run (e.g., 3-4-4-5). Explaining each of these in this paper would be prohibitively time-consuming, but the code is available for review alongside this paper. These check functions are strung together into another keep helper, `keep_consecutive()`, for use when trying to reroll into a small or large straight.

iii. Scoring Functions

Our scoring functions start with the Upper Section functions, which are simply the sum of rolled values matching the category (e.g. 3-3-2-4-1 scores a 6 in the Threes category). Separate functions were utilized here for a more straightforward scorecard construction later,

but this could certainly be done as a single function with a category parameter. These functions are then combined into an `upper_section()` function that determines a roll's score for all Upper Section categories. An initialization value of 0 is given for the Upper Section Bonus, discussed later.

The Lower Section functions are all, much like the check functions above, subtly different from one another, but include the necessary logic to score each of the Lower Section categories. For each of the matching categories, the `table()` function is used to retrieve a count of matching dice. The categories involve a run make use of the previously mentioned check functions, namely `check_4()` for a Small Straight and `check_5()` for a Large Straight, and the Chance function is simply a sum of all dice. These functions are then combined into a `lower_section()` function that works the same as the `upper_section()` function discussed previously. Likewise, an initialization value of 0 is given for the Yahtzee Bonus.

Up to this point, we have only used rolls to determine scores. The Bonus functions, however, rely on the current state of the scorecard, so the `card` parameter is also used. For the `upper_bonus()` function, all that is needed is the scorecard as the 35 point bonus is awarded after scoring a minimum of 63 points in the Upper Section of the scorecard.

The `yahtzee_bonus()` function implements the logic for scoring the Yahtzee bonus if appropriate as well as where to score the Yahtzee using the aforementioned “forced joker rule”. This function first checks to determine whether the Yahtzee category has been scored or blanked. If the player has already scored a Yahtzee, the bonus is added to the Yahtzee Bonus category in the Lower Section, and the forced joker logic is carried out. If the player has blanked the Yahtzee category, no bonus is applied but the forced joker logic still triggers. The forced joker rules force the Yahtzee to be scored in the corresponding Upper Section category if available. If the appropriate Upper Section category is unavailable, the player must use the Yahtzee in a Lower Section box. If no Lower Section boxes are available, the player must then score a 0 in any available Upper Section box.

The bonus functions are unique in that they output a complete end-turn scorecard rather than all possible scores as other scoring functions do. The `yahtzee_bonus()` function is especially unique in that it not only updates the bonus category score but also makes a scoring decision based on available Lower Section categories, always preferring to score in the highest value category.

Finally, we have Scorecard functions. The main functions of note in this category are `scorecard()`, a function that calculates each categories score for a given roll, `score()`, a function that selects the best score given the players scorecard and roll and, optionally, weights for certain categories, and `update_scorecard()`, a function that collects the above functions (and a few others, including the aforementioned bonuses) and returns an updated scorecard at the end of a turn. The optional weight argument in the `score()` function is useful when trying different strategies.

iv. Gamesim and Strategy Analysis Functions

These two sets of functions, while in separate scripts, can be combined in discussion since they are relatively small and related. The Gamesim functions are functions designed to help facilitate a simulated game. The `start_game()` function initializes a blank scorecard, while `play_strat()` will play a full game given a user-defined strategy function. The

`sim_strat()` function uses the `furrr` package to implement parallel computing and simulate a number of games using the defined strategy function and return all games in a single data frame, grouped by game ID. A `card_full()` function and an `end_game()` function have also been defined to make the while loops used in strategy building more palatable and remove some of the redundancies. The Strategy Analysis functions are simply helper functions for summarizing and plotting simulated game data. The analysis functions also include a function that calculates paired-*t* confidence intervals given two sets of simulation data using the process in the Strategy Comparison section below.

IV. Strategies

i. Creating a Strategy

Helper functions discussed above have been designed to make implementing strategies relatively easy. There is still coding knowledge required, but the main pieces of a valid strategy are a) the `while` loop, b) the reroll logic, and c) scorecard updating. The while loop will always be the same, requiring a full run through of the game. This can be done using `while(!card_full(card)) {}`, with the contents of the brackets being the defined reroll logic and scorecard updating.

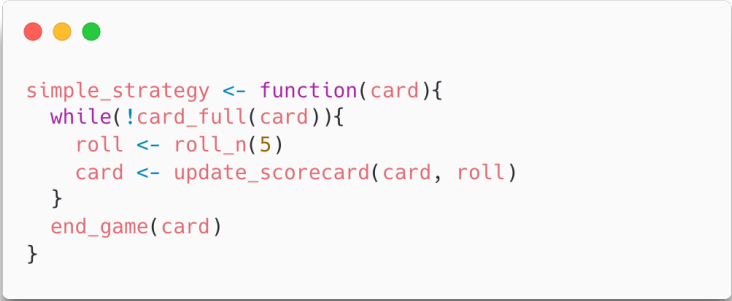
Reroll logic is the most intensive piece of creating the strategy. This is where the dice functions and keep/reroll helpers come into use. Reroll logic often requires a `for` loop, limiting rerolls to two (for a total of three rolls) and either stopping rerolls when landing on a specified category score, or simply crafting the logic in a way that will keep your desired roll on an additional reroll.

Scorecard updating, the final piece of the strategy creation, could technically be handled in the same `end_turn()` function that tallies a final score, but with the addition of weighted categories, it is probably best to leave that decision to the player.

From this point forward, we will explore a few different strategies, some toy cases to illustrate the workflow and set baselines, and some more rigorous strategies to try to maximize our score. All strategies can be found in the `strategy_master.R` file, but each strategy also exists, with the simulation run and some analysis code, in its own individual file, referenced in its section.

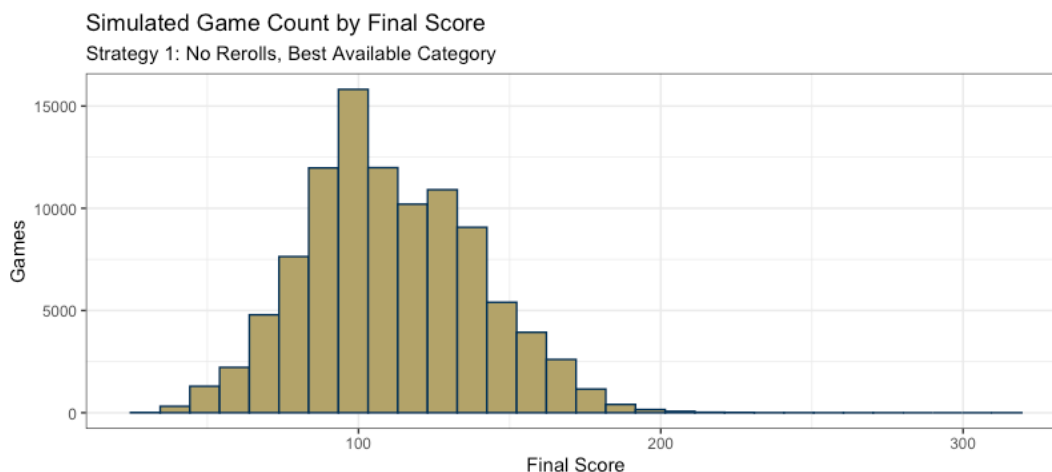
ii. The “It’s All I’ve Got” Strategy

To illustrate the workflow, I have implemented a simple strategy (fig. 1) that requires no rerolls and scores the roll in the highest available category.



```
simple_strategy <- function(card){
  while(!card_full(card)){
    roll <- roll_n(5)
    card <- update_scorecard(card, roll)
  }
  end_game(card)
}
```

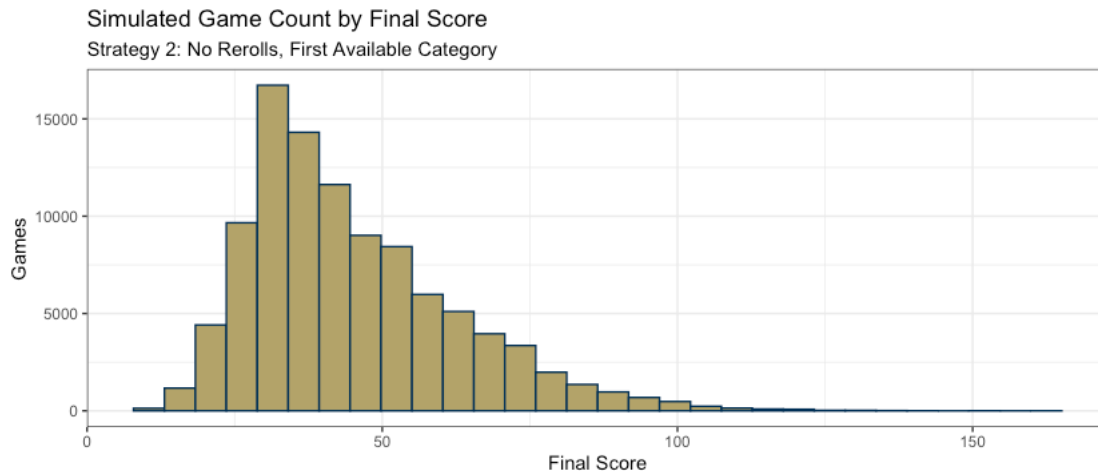
To play a game, we would use either `simple_strategy(start_game())` or `play_strat(1, simple_strategy)`, depending on whether we want the score attached to our results dataframe or as a separate list object. Naturally, the results of this strategy are not particularly impressive. Using seed 6644, running this strategy resulted in a final score of 80. Of course, that is not an accurate representation of how this strategy performs overall. To arrive at a more accurate representation, we can simulate a large number of games using this strategy by running `sim_strat(simple_strategy, 100000)`. This will run 100,000 instances of this strategy (in parallel – the default) and return a table of all games.



From this data, we can see that the average final score using this strategy is around 100, and in fact because we saved all output for all games, we know that the average actually sits at 111 points, with a standard deviation of 28 points. The long right tail of our data is due to a few games achieving a Yahtzee Bonus. With our results, we can also determine probabilities of scores for certain categories or the game as a whole. For instance, on our run of 100,000, we scored at least 100 a total of 62,184 times. This means the probability of achieving a score of at least 100 is 62.2% using this strategy. The same can be done for specific categories, e.g. we can determine that this strategy only provides roughly a 3% chance of scoring 18 in the Sixes category, a threshold that many try to meet when going for the Upper Bonus.

iii. The “I Hate Yahtzee” Strategy

This strategy is actually simpler than the above strategy but requires a little more coding to set up due to my implementation of the scorecard updating. This strategy is for the player that really doesn’t want to play Yahtzee: take the first roll and score it in the first available category down the line. No rerolls, no optimizing for best available score, just straight down the card.

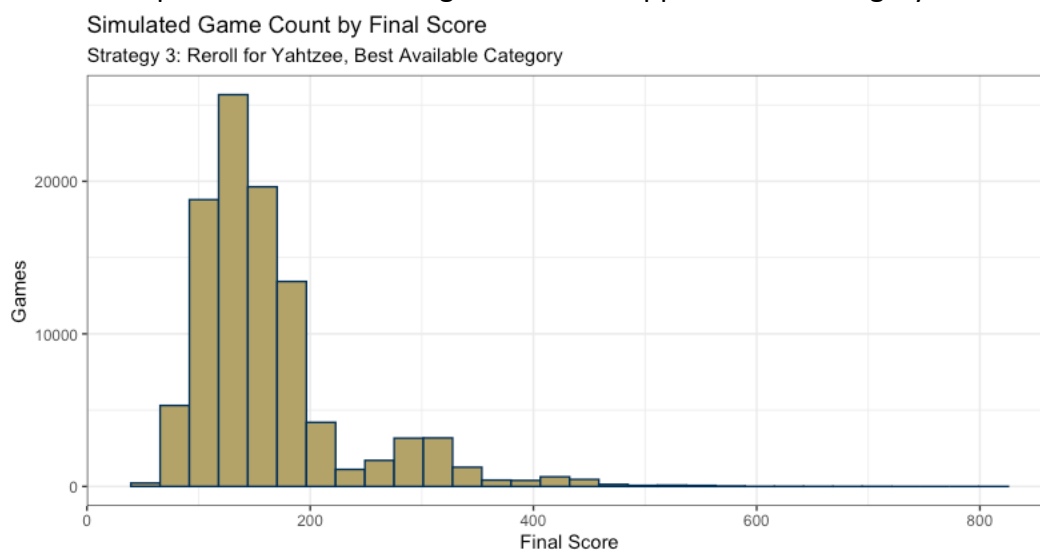


This strategy does extremely poorly, with a 67% probability of scoring below 50, and a 99.7% probability of scoring below 111, the average of our previous strategy. This strategy *is* interesting, though, in that the probability of scoring anything in any given category is the same as the probability of rolling the appropriate pattern on the first roll. In this strategy, we saw Large Straight blanked 96,948 times out of 100,000. That equates to a 3.1% chance of rolling a Large Straight on the first try. This is the same result as analytically approaching the problem.

$$P(\text{Large Straight}) = 5! * \frac{1^5}{6} * 2 = 0.0308$$

iv. The “Yahtzee Superfan” Strategy

Keeping the best parts of our first strategy (scoring on best available category), we are now going to consider rerolls. This strategy will incorporate rerolls that keep the most prevalent dice value. My guess here is that we will see fewer games with Straights scored, but potentially more Yahtzees and Four of a Kind scores, and certainly more Three of a Kind scores. I would also hypothesize that we will see more Upper Section Bonuses, as the threshold for achieving that bonus is the equivalent of 3 matching dice in each Upper Section category.

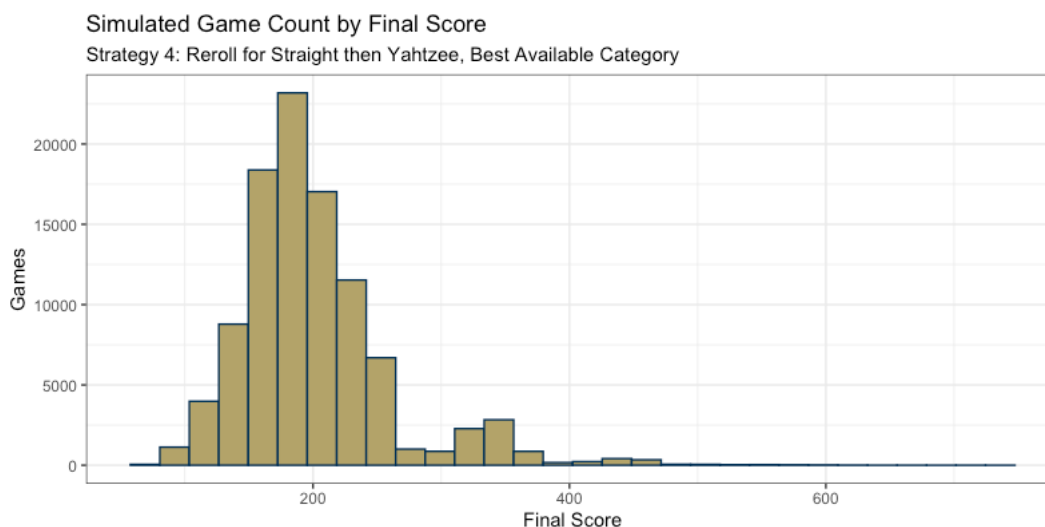


The distribution here, similar to the distribution of future strategies, is a little strange due to an abundance of Yahtzee Bonuses, which are still rare despite being very visible in this

simulation. This strategy still only averages about 2/3 of a Yahtzee per game, or 3 Yahtzees every 5 games. It has a mean of 162 points, which brings it above our current winning strategy, and the probability of scoring below our current best 111 is just 18%.

v. The “Straight Some Chaser” Strategy

This strategy makes use of a few additional `if` statements and dice functions to check for consecutive runs of dice, keep those dice if it is a run of at least 3, and roll for Yahtzee (like our last strategy) otherwise. This one also makes use of the weighting capability of the scoring functions to downweight the Chance category, since it tends to be best used as a throwaway category.



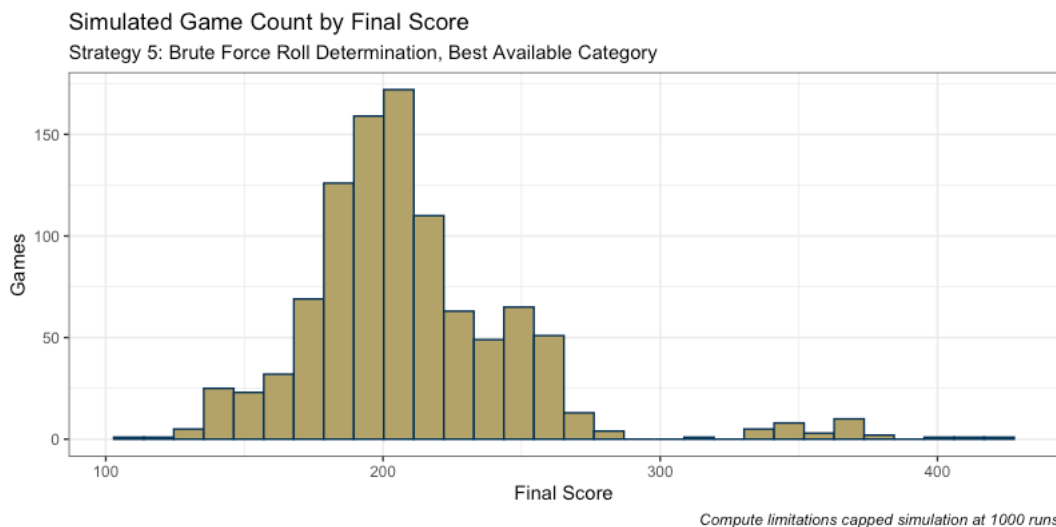
This strategy looks similar to the “Yahtzee Superfan” strategy above, but shifted right a bit. We can determine that our average score is 201, and we only have a 2.1% chance of scoring below 111, the score of our first strategy, and a 22.2% chance of scoring below 162, our current frontrunner. Using this strategy, we see a Small Straight scored in 97.6% of games, and a Large Straight scored in 85.2% of games.

vi. The Brute Force Strategy

This strategy is the most compute-intensive strategy. It uses functions from the `brute_force_functions.R` file to create a table containing every possible dice roll, a total of 7776 possibilities, and the scores associated with them. It also uses a separate function to generate each of the combinations of dice that could be kept for any given roll. The strategy then maps across each of those “keeper combinations” across every possible dice roll to find the highest expected point total, or average of a category’s score, across every possible next roll. This strategy could use some refinement, since it does not consider the following roll when determining “expected score” on the first reroll. The strategy is already computationally expensive, so while it would be a worthwhile endeavor to pursue, it is ultimately out of the scope of this project.

Worth noting, due to compute limitations, this strategy was run 1,000 times instead of 100,000 times for initial analysis. This makes our preliminary comparison suspect because of the varying sample sizes, but in the next comparison section we match run length for our two

strategies, bringing our other strategy down to 1,000 games. This should still be sufficient for comparison and analysis but is a major limitation of this analysis. Given more time, I could optimize this strategy to more efficiently run in parallel and increase the simulation length to something closer to the other strategies.



Again, we see a similar distribution to our previous strategies, but a higher average score than any of our previous attempts. This strategy has an average score of 210 and a standard deviation of roughly 40 points. Compared to the other strategies, the percentages of non-blanked categories are much higher across the board, but these comparisons, as mentioned above, are less than ideal due to sample size discrepancies.

vii. Strategy Comparison

One thing to consider is human playability. Even if we were to determine that the Brute Force strategy is the winner, it is not a reliably playable strategy for the average player. The only way to play that strategy is with computer assistance, removing most of the enjoyment from one's Yahtzee experience. This consideration will not change our statistical decision of the best model, but it would impact the strategy recommendation if one were to be made.

To compare these strategies, we can use the confidence interval analog to a paired- t test^[3]. The reason that this is possible is because, the way our simulation has been implemented, each strategy is provided with the same initial rolls, meaning each game in a strategy simulation is comparable to the same game in a different strategy simulation. If the first roll of turn 3 in strategy 1 is 1-2-3-5-5, the first roll of turn 3 in strategy 2 is also 1-2-3-5-5. The differences arise explicitly from the reroll logic implemented, the heart of our strategies, so comparison in this manner is appropriate.

We must first calculate the differences in final score across each game in our two strategies. That is:

$$D_j \equiv Z_{1,j} - Z_{2,j} \text{ for } j = 1, 2, \dots, b$$

The sample mean and variance of these difference can then be calculated as such:

$$\bar{D}_b \equiv \frac{1}{b} \sum_{j=1}^b D_j \text{ and } S_D^2 \equiv \frac{1}{b-1} \sum_{j=1}^b (D_j - \bar{D}_b)^2$$

With these values at hand, we can now calculate the $100(1-\alpha)\%$ Paired- t Confidence Interval with the following equation:

$$\mu_1 - \mu_2 \in \bar{D}_b \pm t_{\alpha/2, b-1} \sqrt{\frac{S_D^2}{b}}$$

Using these equations, we can compare our strategies against each other to find which is, statistically, the best. A selection of comparisons is included in the table below. In each of the comparisons, we can see that, because the 95% CI is entirely to the right of 0 and our goal is a high score, Strategy 1 is better than Strategy 2. The confidence interval provides us a range of improvement to expect by using Strategy 1 over Strategy 2.

Strategy 1	Strategy 2	95% CI for Strategy 1 vs. Strategy 2
Brute Force	Run Yahtzee	[8.09, 14.61]
Brute Force	Keep Mode	[45.76, 53.15]
Run Yahtzee	Keep Mode	[38.07, 38.63]
Run Yahtzee	Simple Strategy	[89.41, 90.21]
Simple Strategy	Fill Down	[65.64, 66.02]

V. Conclusions

i. Limitations and Future Work

Compute limitations led to our comparison work relying on significantly smaller (100x) simulation runs than we were capable of producing for most of our strategies. With additional time, I could optimize the Brute Force strategy for speed, reducing the amount of time it takes to run each game, thus reducing the amount of time needed to run the simulation as a whole. Future work could also be focused on improving the Brute Force strategy by including second roll probabilities in the expectation calculations. While this is an interesting pursuit, if the overall goal is for a person to maximize their Yahtzee score, none of the brute force methods are going to be viable gameplay options, as they are too intensive to calculate without assistance.

ii. Summary

In summary, we have discussed the design aspects of the Yahtzee simulation, how to create strategies using the framework put forth in this project, and evaluated an increasingly complex series of strategies using simulation. We then compared our strategies to find that the Brute Force strategy reliably offers the highest average score of the four strategies we evaluated. However, recommending a strategy is more than just determining the highest score. As discussed, the objective of a game of Yahtzee can differ whether the player is playing alone or against another player, and the ability for the player quickly parse through reroll logic can play a factor in what is a personal best strategy.

I was, frankly, surprised by how often certain events occurred in the simulations. For instance, one of the games scored >800 points in our “Yahtzee Superfan” strategy, which is a shockingly high score that required 6 Yahtzees to be rolled in a single game. Checking into this simulation, however, showed that the average behavior of the simulation was not incorrect, and this is a perfect example of the reason we do simulation evaluation. Using averages are an okay stand-in, but anomalous behavior happens all the time and simulation is one of the best tools we can use to account for this.

VI. Citations

[1] Tom Verhoeff (1999) How to maximize your score in solitaire Yahtzee (unfinished)
<http://www-set.win.tue.nl/~wstomv/misc/yahtzee/yahtzee-report-unfinished.pdf>

[2] Yahtzee Instructions (1996) <https://www.hasbro.com/common/instruct/yahtzee.pdf>

[3] Dave Goldsman (2020) Comparing Systems [Lecture Slides]
<https://www2.isye.gatech.edu/~sman/courses/6644/Module10-ComparingSystems-201128.pdf>