Optimal Yatzy strategy

Marcus Larsson
marcular@kth.se

Andreas Sjöberg
ansjob@kth.se

August 10, 2012

Abstract

Yatzy (originally known as Yahtzee) is a classic game of chance, played for fun either as a solitaire to pass time or competitive versus an opponent. The purpose is to get pairs, triplets, et cetera (similar to poker), by choosing which dice to hold and which ones to roll. We have adapted a formerly known algorithm for the original version of the game. Our algorithm finds the optimal strategy for the Scandinavian version of the game (Yatzy). The expected score for a game played with the computed optimal strategy is 248.63.

The maximum expected utility of our results includes both using the strategy to create an AI in a Yatzy game (possibly on a mobile client) and ideas behind the algorithm may be used to create an AI for a similar game, like poker for example.

Keywords: Yatzy, dynamic programming, game solving, computer strategies, perfect play

Statement of collaboration

We have worked together on this project, sitting next to each other working on the same computer. We have both been involved in background research, implementation, and authoring this report. It is very difficult to tell who did what and at what time, but Marcus had more a focus on the algorithm and Andreas on implementation details.

Contents

1 Introduction

Among the popular games of skill and chance, played either alone or with others, Yatzy proves to be quite interesting to analyze with respect to finding a winning strategy. Strategies for simpler games, like Tic-Tac-Toe, are easy enough to figure out simply by playing and analyzing a couple of games. For the more complicated games, such as Chess, the optimal strategy is too difficult even for modern computers to calculate. Yatzy however does not have an obvious optimal strategy, but it is still not too complicated for a computer to find. In this paper we find and compute this optimal strategy for the Scandinavian version of Yahtzee, referred to as Yatzy.

1.1 Problem statement

The problem is to find a policy for the game of Yatzy that tells the player what action to take at any given point in the game. This policy should be optimal with regards to the average final score.

2 Background

2.1 The Yatzy game

Yatzy is a very popular dice game played worldwide. The game was invented by a wealthy Canadian couple aboard their yacht. It became very popular among their friends, where they simply called it "Yacht". When asked by the couple to produce a few sets of the game, Mr. Edwin S. Lowe (a successful toy and game maker) grew so fond of the game he wanted to buy the rights for it. The couple agreed, settling for the price of 1000 sets of the game to give to their friends. The game was given the name "Yahtzee" and eventually gained great popularity. Around the year of 2007 it was estimated that 100 million people play the game on a regular basis [1].

2.2 Yatzy rules

When reading up on the subject, we found that Yatzy rules differ greatly depending on where you acquired the rule set or game manual. The rules for Yatzy as they are known in the U.S., that have been previously researched, are completely different from those we are familiar with. In this thesis, we use the Scandinavian (Swedish) rules for Yatzy, which are fully described in Appendix A.

2.3 Optimal strategy

By optimal strategy, we mean a policy for the player's decisions, which will maximize the average final score. This strategy has already been found and computed and the expected score for a game of Yatzy is 254.59 [2–4]. However, as mentioned above, this strategy applies only to the specific set of rules used in the U.S. Fortunately, the rules differ mostly in how the score is calculated, and not so much in how the game is actually played. This means that the same principles and ideas used to compute the optimal strategy for the American Yatzy can be applied, with only slight modification, when computing the optimal strategy for Yatzy as we know it.

3 Method

3.1 Preliminaries

By viewing the game as a graph with nodes corresponding to states in the game, and edges symbolizing the transitions between these, we have a good starting point for solving the problem.

One of the most important things to realize when finding the optimal strategy is that no matter where you are in the game at any given point, how you ended up in that situation has no impact your future decisions for the optimal strategy. For example, having filled the Ones, Twos, and House categories, it does not matter in what order these were filled in for your next decision. If you rolled four 5s and a 1 with these categories filled in, you will still want to keep the 5s in an attempt to get the fifth 5 for a Yatzy, no matter in what order you filled the previous categories.

While the above might be considered somewhat obvious, what is less intuitive is that also the scores filled in the various categories (other than Ones through Sixes) do not affect future decisions either [2]. The above example with the Yatzy is true no matter if you scored 0 or 28 points in the House category. What matters is which categories are still free to fill.

As mentioned, the scores filled in for categories Ones through Sixes actually do matter. This is because of the Bonus rule in the game. Once the sum of all the upper categories is 63 points or more, the player is awarded 50 bonus points. This means the scores filled in those categories actually do have impact on our future decisions. However, this condition can be simplified, as we are actually only interested in the upper categories total score, not each of the individual categories' scores. What we want to know is how many points away from acquiring the bonus we are, and once we have 63 points or more, the upper total doesn't matter anymore, because we know we received the bonus already [4]. The state of having scored 4 points in Twos, 0 points in Fours and 15 points in Fives is equivalent to the state of having filled 0 points in Twos, four points in Fours and 15 points in Fives. Both of these cases have the upper total of 19, and have marked the exact same upper categories.

3.2 State definition

We can now define a state in the game as follows:

$$S = (t, D, C, u)$$

where t is the throw (1-3), $D = \{a, b, c, d, e\}$ the multiset of dice one has rolled, C the set of categories that have been filled, and finally u the upper categories total score. Because the ordering of the dice does not matter in Yatzy, we keep D as a sorted multiset, for simplicity's sake ($D = \{3, 1, 4, 5, 4\}$ is equivalent to $D = \{1, 3, 4, 4, 5\}$). This game state contains all the necessary information to find the optimal strategy [2–4].

From any given state in the game, there is a set of possible next states. Assuming we know the expected score for each of the next states, we can take actions that will maximize this expected score, and also calculate the expected score of the state we are currently in. This requires us to constantly know all of the expected scores of possible next states. Traversing the graph in any manner would result in a too heavy workload, and the key to this problem is to traverse the graph from the end of the game, reusing already computed values. This is possible due to the previously mentioned fact that no matter how we ended up in a given state, the situation is equal from there on.

3.3 Action definition

The different actions one can take in the different states are either:

1. which dice to hold, or
2. in what category to score the current dice

The action for holding a set of dice can be represented by a binary mask of length 5, where a "1" on position i would represent holding the i:th die, and a "0" corresponds to rolling it. This mask fits in a byte. The action for choosing a category can be represented by a number between 1 and 15, a number corresponding to each category in the score card. This also fits in a byte. This means we can store the optimal strategy in a data structure with size in bytes as big as the number of nodes in the graph.

Yatzy rules allow the player to submit the score after any of the three rolls. To model this, we define one action to be "hold all dice". If this is repeated for the remaining rolls, the effect is the same as submitting the score immediately. This allows marking actions only to take place after the final roll, which simplifies the algorithm.

3.4 Analysis of graph size

Because of the observations on the irrelevant factors made above, we have substantially reduced the state space for the game. Producing the graph G with the nodes representing above states, we can find $|G|$ with simple combinatorics.

- The number of distinct dice outcomes in the game equals the number of ways to pick $n = 5$ dice out of $m = 6$ categories is given by the formula

$$\binom{n+m-1}{m-1} = \binom{5+6-1}{6-1} = \binom{10}{5} = 252$$

  different outcomes.

- The number of differently filled scorecards that need to be considered are $2^{15} = 32\,768$. The 15 comes from the number of categories, which can be either filled ("1") or unfilled ("0").
- The different upper categories totals we are interested in are 0 to 63, where 63 represents an upper total of at least 63 (meaning the bonus has been achieved). This gives 64 different upper categories totals.
- The roll number, indicating at what roll we are currently at. This ranges from 1 to 3.

The number of states will then, using the above numbers, equal:

$$252 \cdot 32\,768 \cdot 64 \cdot 3 = 1\,585\,446\,912$$

By encoding each action as a byte, the size of the data structure containing the complete optimal strategy would be $\approx 1.5GB$.

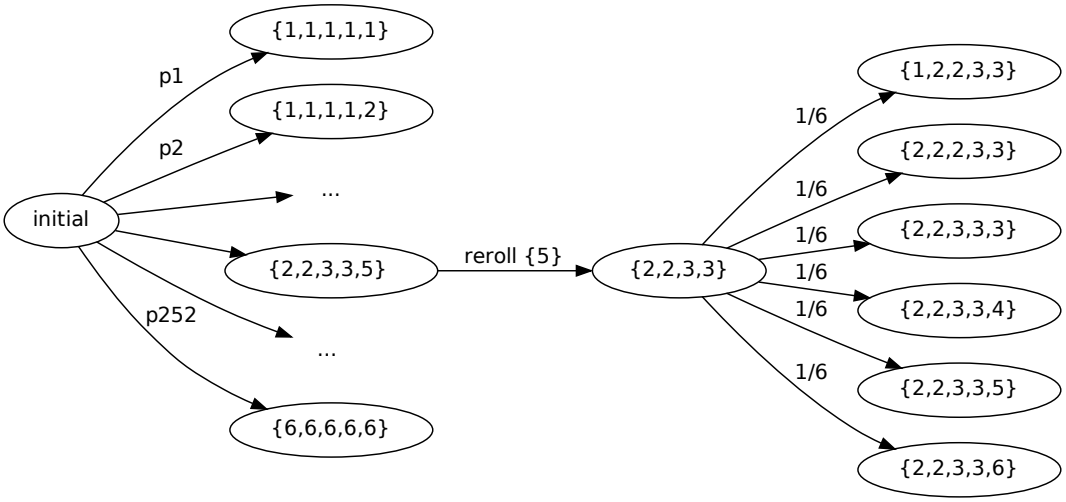

Figure 1: Sample states during the game

3.5 Dynamic programming algorithm

As mentioned, the computation of the expected score begins at the end of the game, traversing the graph towards the beginning of the game, in a breadth first manner. By traversing the graph this way, we always know the expected score of future states, and can decide upon the optimal action for the current state. This can be done recursively until we are at the beginning of the game.

In figure 1 an example initial roll state is illustrated. This figure depicts the player rolling, choosing which dice to keep, and then rolling the remaining dice. There are two kinds of transitions here, one that is probabilistic, and one where the player makes a decision. First, there are 252 states for each of the possible dice outcomes. Then, the player may choose a subset of the dice to keep, represented by the next set of states (only one possible subset is shown in the figure). Finally, the remaining dice or die are rolled. The final states of this figure would correspond to the next state in G, namely the one corresponding to the (possibly) different dice outcome for the next roll (with the same scorecard and upper categories total score).

3.5.1 Base cases

The absolute base case of the algorithm is when there is only one category left to fill in, and after the last throw. The score awarded for a chosen roll is given simply by applying the Yatzy rules for the remaining category and the dice. By iterating over all possible roll outcomes for all possible categories unfilled, it is possible to compute the expected scores for all the nodes that correspond to the final states of the game. In this step, these score are the actual scores for each state, as no probabilities are involved yet.

For example, using figure 1, letting the final states of the figure be the final roll in the game when we have the House category left unfilled. There are two dice outcomes that award

points at this point ($\{2,2,3,3,3\}$ and $\{2,2,2,3,3\}$) and the score for these are respectively stored in those states (13 and 12). The other states are worth 0 points.

3.5.2 Recursion rolling steps

When we want to calculate the optimal holding action for a given state, we iterate over every possible way to hold the dice, and calculate the weighted average of the outcomes. The actions that give the highest such average is saved, and this average is stored as the expected score for the current state.

Again using the example above, we want to end up in the most score awarding states, so we maximize our chances of getting there by choosing the kept dice with the maximum probability (weighted with score) of reaching those states. By keeping $\{2,2,3,3\}$ we only need either a 2 or a 3 to be in a score awarding state. This is naturally better than re-rolling all of the dice, as the chances of rolling a full house with 5 dice are smaller than rolling either a 2 or a 3.

Also, by calculating the expected score for any set of dice before the first roll, allowing none of the dice to be kept (effectively rendering those dice useless), we find the expected score for the state before the first real roll. This expected score is saved for future computation, as this is the expected score awarded for the given state of the scorecard, without regards to the dice.

3.5.3 Recursion marking steps

Once we have completed computations for the first throw, we need to step back to the previous states of the game, where one less category was filled in. By iterating all of the possible ways this can be done, and once again iterate over all possible dice outcomes for each of those ways, we can find the optimal category to fill in for that dice outcome in that state.

This is done by trying to fill each of the remaining unfilled categories, calculating the score awarded for filling the category with the dice. This score is then added to the expected score for the state corresponding to *before* the first roll for the other, remaining categories. Since we computed and saved this value in the last recursion rolling step (this is the score for the state of the scorecard without regards to dice), we can now easily compute this sum for each unfilled category, and compare the different expected scores they yield. The optimal marking action is then chosen to be the action that awards the maximum of these sums.

For example, let small straight and large straight be the remaining unfilled categories. If in the last roll of the current round we achieved neither of the score awarding dice combinations, we need to score 0 points in any of the remaining categories. Because we know the expected scores for the states corresponding to the scorecards where small straight is the only unfilled category and the scorecard where large straight is unfilled, we can choose to put the 0 in the category awarding the least points. In this case, it is optimal to put the 0 in the small straight category, because the probability of acquiring a small straight is equal to that of acquiring a large straight, but the large straight awards more points. In other words, we know that the scorecard where the large straight is unfilled is worth more than that where the small straight is left unfilled.

3.6 Optimization using Keepers

When calculating the expected score during the rolling steps, the straight forward algorithm is to take the score of the possible outcomes weighted by the probability of reaching said outcome. This is done as follows: For each roll outcome D and its score value $roll[D]$, and for each way K of keeping dice, compute the expected value $keep[K]$ (when keeping K), using the formula below.

$$keep[K] = \sum_{D \supseteq K} (Pr(D|K) \cdot roll[D]) \tag{1}$$

Given a certain scorecard and roll number we already have the expected score for the next possible states, $roll[D]$.

The method of choosing the dice, given by formula (1), is very inefficient since many of the outcomes are the same, causing redundant computations. For example, if the player has the dice $\{1, 1, 2, 3, 4\}$ and decides to roll one of the 1s, they can end up with the final dice $\{1, 2, 3, 4, 5\}$, which would then have to be evaluated. If they instead kept the other 1, the same outcome is possible, and must be evaluated again using this method. (In this particular case all of the outcomes are the same for saving either of the 1s, making it even more redundant.) By observing that this calculation has already been made and reusing those results, there is a way to substantially reduce the computational work for the rolling steps in the recursion.

With the above notation, we can compute the value of keeping K using dynamic programming, explained by the following formula:

$$keep[K] = \begin{cases} roll[K] & \text{if } |K| = 5 \\ \frac{1}{6} \sum_{d=1}^{6} keep[K \cup \{d\}] & \text{otherwise} \end{cases} \tag{2}$$

By iterating over $|K|$ from $|K| = 5$ to $|K| = 0$ we can reuse the results of the previous computations. For further explanation and proof of this formula, see [4].

Since we already know the values of $roll[K]$ when $|K| = 5$ (these are the expected scores for the different roll outcomes), we can start filling $keep[K]$ for those K. When this is done, we calculate the rest of the *keep* structure by applying formula (2). This is achieved by iterating over all possible keepers with $|K|$ decreasing. After *keep* has been filled, it is possible to iterate over every roll outcome, finding the corresponding holding actions K and their associated scores $keep[K]$. The optimal action is then, just like before, chosen to be the one with the greatest $keep[K]$ (maximizing expected scores).

3.7 Parallelizing

There are plenty of opportunities to parallelize the computations when generating the optimal strategy. Many of the computations required are independent of each other, and can therefore be executed simultaneously. For example, when generating the base cases, none of the calculations related to a certain final scorecard are related to the other (final scorecards). This means the base case generation can at least be broken down into $\binom{15}{1} \cdot 64 = 960$ independent units of work. The same is true for the recursive steps, except for the fact that one step has to be completed before the next. We can get between 960 parallel units in the base cases, and $\binom{15}{7} \cdot 64 = 411\,840$ parallel units of work during the recursion.

If these parallelizations are not enough, it is possible to do the keepers optimization mentioned in section 3.6 simultaneously. The number of parallel units of work here varies between

252 (the number of keepers K with $|K| = 5$) and 1 (the single K where $|K| = 0$). The keepers parallelization is of course done within each of the work units mentioned above, meaning that at some points during the execution over 10^6 work units could be running at the same time.

4 Results

4.1 Implementation

We chose to implement the algorithm in Java. The implementation was initially very slow and required great amounts of memory, but after optimizations (especially the Keepers optimization mentioned above) and better memory management it was fast enough to run in reasonable time, with memory requirements that a modern desktop computer can handle. The complete computations take about 50 minutes on an Intel Core i5 2500K, writing the optimal strategy to the 1.5GB table file to disk once computed.

We realized that the simplest way to do the work in parallel would not be feasible. This trivial idea was to put each work unit during one recursion step in a `List`, and feed the entire `List` to a `ExecutorService`. Since this `List` needs to hold hundreds of thousands of work units, it exhausts the computer's memory. A way to avoid this would be to utilize a consumer/producer pattern where the producer blocks if the queue is full.

4.2 Correctness

As the optimal strategy is found through recursion of the game states, we know that no matter what state we are in and how poor or good we played until reaching the state, we *will* play using the optimal strategy for the rest of the game. Seeing how we use this recursion from the end of the game all the way back to the beginning of the game, we will naturally play optimally throughout the whole game.

The strategy assumes the dice are fair. In the scenario where the game has progressed poorly due to unfortunate rolls, the strategy does not take this fact into account and try to compensate for this. Since it assumes that the dice are fair, statistically it will rarely roll poorly for an entire game.

4.2.1 Average score

The calculated (expected) average score for games played with this optimal strategy is 248.63. This number seems reasonable, seeing as it is lower than the American counterpart, and the Scandinavian rules are in fact less score-awarding in comparison.

We tested the algorithm by letting it play 20 000 games of solitaire Yatzy. The frequency of the different scores obtained is plotted in Figure 2. On average, the algorithm obtained a score of 248.92, which is convincingly close to the expected value.

As can also be seen in Figure 2, the distribution leans towards scores above 200. This is expected, since we are playing the optimal strategy which is similar to minimizing the probability of a low score. Also, the most common score observed was 244, which is again close to the expected score. The various peaks visible in the graph could be explained by the discrete scoring nature of some categories, such as Yatzy or straights. These categories do either award points, or nothing at all.

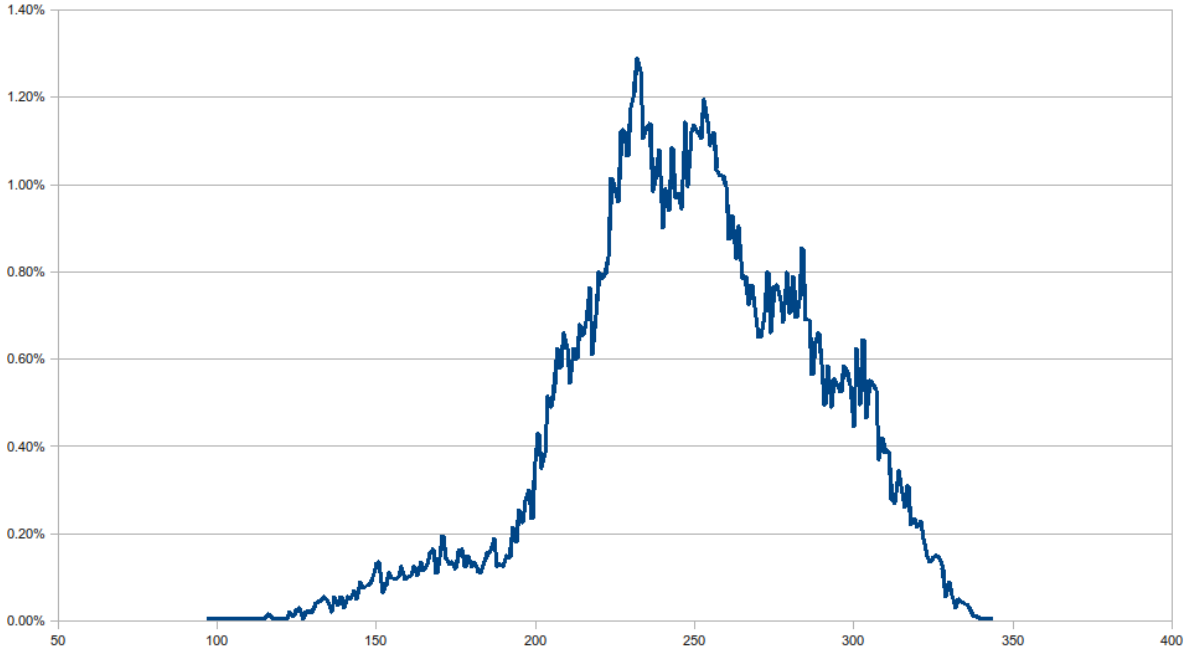

Figure 2: Breakdown of scores by the frequency they are obtained with the optimal Yatzy strategy. This graph was generated by simulating 20 000 games.

4.2.2 Sampling tests

To verify that the implementation really does the correct calculations we ran a couple of sampling tests. Some trivial samples have been taken on the strategy for obvious states in the game, for example after rolling a Yatzy when the Yatzy category is still free, and the suggested strategy for those cases have been correct.

We also ran a complete game simulation with carefully chosen dice rolls. The final roll was always the maximum allowed score for some category. This made it easy for us to verify the correctness of policies. Additionally, the dice chosen for the first and second roll had obvious keeping actions so the rolling strategy would also be easy to predict. This test was passed, which strengthens our confidence in the fact that all of the calculations are correct.

5 Conclusion

We have adapted an existing algorithm for optimal Yatzy to the Scandinavian rules. Usages of this strategy include the obvious AI for a Yatzy game. This strategy could also be used to evaluate heuristic strategies for the Yatzy problem.

While the computation time and resulting lookup table size are not that impressive, further work could include doing the same calculations in less time, and the strategy could be stored less explicitly to reduce its size. We could for example store only the expected scores for the different initial states, leaving the rest of the computational work to be done at runtime (during lookup). This results in a size reduction by several orders of magnitude. For other suggestions on how this could be done, including further computational optimizations, see [2, 4].

The methods used for this problem could also possibly be applied to other similar games

or systems. One general idea that is particularly useful, is to try to reduce the state space of a problem, by trying to find observations similar to those in the Yatzy problem, before examining all states exhaustively. When the state space has been reduced, dynamic programming proves to be a useful tool to solving the remaining problem efficiently. A concrete example of application would be the type of poker played on slot machines, which has some similarity to Yatzy.

References

[1] The great idea finder. `http://www.ideafinder.com/history/inventions/yahtzee.htm`. 2012-03-30.

[2] James Glenn. An optimal strategy for yahtzee. Technical report, Department of Computer Science, Loyola College in Maryland, 2006.

[3] Ph. D. Olaf Vancura. *Advantage Yahtzee*. Huntington Press, Las Vegas, 2001.

[4] Jakub Pawlewicz. Nearly optimal computer play in multi-player yahtzee. In H. van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games*, volume 6515 of *Lecture Notes in Computer Science*, pages 250–262. Springer Berlin / Heidelberg, 2011.

[5] Alga Spel. Yatzy rules. `http://www.algaspel.se/Rules.aspx`. 2012-04-12.

A Yatzy rules

The following rules are based on Alga's Yatzy on the road ruleset [5], which correspond exactly to the rules we are familiar with.

The game consists of as many rounds as there are categories on the score card. In the beginning of every round you roll all of the five dice. After the initial roll you can choose to fill a category on the score card, or roll your dice again. If you choose to roll again you are also allowed to hold 0 to 4 dice (holding 5 is equivalent to not rolling). Holding means you save a die and don't include that dice in the next roll. Rolling 6, 6, 6, 6, 4 for example, you might want to attempt to get a Yatzy, saving the four sixes and rolling just the 4. In each round you may roll up to three times, including the initial roll, saving as many dice as you want between the rolls. After the third roll you must fill a (previously unfilled) category in the score card.

In our rule-set we have a score card with 15 categories: ones to sixes, pair, twopair, three of a kind, four of a kind, small straight, large straight, house, chance and Yatzy.

Categories ones to sixes are scored in the same manner. In the case of twos for example, you simply add up all the twos you rolled, and this becomes the score (1, 2, 2, 2, 5 becomes $2 \cdot 3 = 6$ points in twos). There's also a special rule about ones to sixes regarding the total of these scores. If the total score of these (usually called the upper categories) is equal to or greater than 63 you are awarded an additional 50 points, called the upper bonus.

For the pair category, you are awarded points equal to the sum of two identical dice. For twopair, you are awarded points equal to the sum of two sets of pairs like above. These pairs however, can not be the same (1, 2, 2, 4, 4 is a twopair, but not 1, 2, 2, 2, 2).

Three of a kind simply awards points for the sum of three identical dice, and four of a kind is just the same but with four identical dice instead.

Small and large straights award points equal to the sum of the dice if you have acquired either a small (1, 2, 3, 4, 5) or a large (2, 3, 4, 5, 6) straight with your dice. In other words these categories can award $1+2+3+4+5 = 15$ and $2+3+4+5+6 = 20$ points respectively.

The house category awards the sum of the dice if you acquire two groups of identical dice, three in the first group and two in the second. Additionally, the two dice groups cannot be of the same number (dice 6, 6, 6, 6, 6 is not a valid house). The best score in this category would be $28 = 3 \cdot 6 + 2 \cdot 5$ (dice 5, 5, 6, 6, 6).

The chance category is simply the sum of any combination of dice. The lowest score possible in this category would be five ones ($1 + 1 + 1 + 1 + 1 = 5$), and the highest would be five sixes ($6 + 6 + 6 + 6 + 6 = 30$).

Finally, the Yatzy category awards 50 points if you acquire any combination of five identical dice (1, 1, 1, 1, 1 or 2, 2, 2, 2, 2 etc.).

Whenever you end up with dice that doesn't fit a specific category, you may choose to "zero out" that category. For example, if you already filled all the upper categories (or don't want to fill in low scores in there) and end up with the dice 1, 2, 3, 4, 6, you can choose to either fill it in the chance category (for 16 points), or you might zero out a category that you are less likely to fill, like the Yatzy category. Doing this gives you 0 points in that category, and if you later on roll a Yatzy (or whichever category you zeroed), you can't fill it in that category.