A Nearly Optimal Computer Player in Multi-player Yahtzee

Jakub Pawlewicz
pan@mimuw.edu.pl
Institute of Informatics, Warsaw University

Abstract

Yahtzee is the most popular commercial dice game in the world. It can be played either by one or many players. In case of the single player version, optimal computer strategies both for maximizing the expected average score and for maximizing the probability of beating a particular score are already known. However, when it comes to the multi-player version, those approaches are far too resource intensive and thus are not able to develop an optimal strategy given current hardware.

This paper presents the first in-depth analysis of the multi-player version of Yahtzee. Proposed implementation of optimal strategy for the single player version significantly speeds up calculations. Resources necessary to memorize the optimal strategy for a two-player game are precisely estimated. It is shown that developing an optimal strategy for more players is not possible with the use of current technology. For this case, a heuristic strategy is suggested and by means of experiments created especially for this purpose it is proven that in practice this strategy is indistinguishable from the optimal one.

An experimental analysis of the actual advantage of the optimal strategy over suboptimal opponents like humans has also been conducted. Results show that Yahtzee is "highly" a game of chance and advantage of the optimal strategy is insignificant.

Keywords: Yahtzee, game solving, dynamic programming, game tree search, heuristic methods, perfect play.

1 Introduction

Yahtzee is the most popular commercial dice game in the world. More than 50 million Yahtzee games get sold annually. An estimated 100 million people play Yahtzee on a regular basis[1].

Players roll five 6-sided dice, score combinations and try to get the highest total score. The game can be played by an arbitrary number of people. In the single player version, the aim is to score the maximum number of points. If more than one player participates in the game, everyone tries to get more points than his opponents.

[1] Source [7]

Single player version is an attractive target to tackle with modern computer power because the number of states is small enough to fit in memory. In contemporary work the optimal strategy has been presented by several researchers, see [11, 12, 3]. They have solved the problem of maximizing the expected score. We repeated their calculations while optimizing the number of states and computational speed. Investment into optimizations pays off when it comes to constructing more complicated strategies, especially in case of the multi-player game.

Another problem of the single player game a researcher can aim at solving is maximization of the probability of beating a particular score. This has been solved by Cremers [1]. However, computing time required by his technique is not acceptable in practice if we expect a player to make decisions immediately. We introduce innovative approach with help of *distributions*. This structure replaces the usual single value associated with a game state. Exploiting this idea we are able to compute an optimal strategy once, and during a game make every move in fraction of a second.

This paper is the first published research of the multi-player version. All state space estimation performed for two-player version indicates that its solution is within range of only the top clusters of today. In case of more than two players it is practically impossible to solve the game. Techniques developed for the single player version do not transform to the multi-player one. In case of the two-player game, after optimization of the size of the state space, we demonstrate that the resources needed to build an optimal strategy are available today. Nevertheless even in this case the requested resources are still so significant that a different approach is called for.

For the multi-player case we have developed heuristic strategies by building high-quality evaluation functions based on distributions introduced for single player game. Surprisingly, this strategy turns out to be extremely effective. In case of the two-player game we are able to demonstrate – by means of specially prepared experiments – that the proposed heuristic strategy is "almost" optimal. Moreover, the technique used to construct the heuristic is universal and probably can be applied to other games.

Finally, since the game of Yahtzee includes a significant amount of stochasticity, one can ask whether applying strong strategies is observable against imperfect players. To answer this question we analyze nearly 25 million games played by humans with diverse range of playing strength.

2 Game characteristic

The single player version game consists of 13 turns. There is also a score table containing 13 categories. In each turn player rolls dice three times. After the first and the second roll player can decide which dice he wants to roll again. He selects those he wants to keep (so called keepers) and rolls the rest of them.

After the last roll player picks up a category and puts a score into it. The score value is derived from the number of pips on dice and the selected category. After putting a score into a category, it cannot be scored again, i.e. every category during the whole game must be scored exactly once. Additionally there are two bonuses. The exact scoring rules are listed in Appendix A.

In the multi-player version players play in rounds. In a single round every player makes one turn. In every round the players play in the same, fixed order.

3 Single player game

3.1 State space analysis

The game of Yahtzee can be seen as directed acyclic graph. Nodes represent states and edges describe moves between states. We distinguish two types of states as shown in Figure 1.

(a) Random event (b) Player choice

Figure 1: State types.

Blue node represents a random event (Figure 1(a)). This is a "cold" situation in which switching to next state depends on luck. Every succeeding state can be selected with some probability associated to the outgoing edge. This kind of state occurs when player has to roll dice.

Red node represents a situation in which a player has to choose an outgoing edge (Figure 1(b)). Such a situation is "hot" because here the player can influence the outcome of the game. This state takes place when player has to select keepers or has to select a category to score.

Let us call the state just after putting a score into some category and just before the first roll in the next turn a *score state*. This state occurs between two consecutive turns or at the beginning of the game or at the end. Every other state can occur only in the middle of a turn, therefore we call such a state *turn state*.

Let us enumerate all states in a single turn. We start from some nonterminal score state. From here we walk through turn states. After the first roll we have $\binom{6+5-1}{5} = 252$ distinct arrangements of 5 dice with 6 sides. Then we can keep any number of dice from 0 to 5[2]. This results in $\binom{7+5-1}{5} = 462$ different possible combinations of keepers. Next we have the second roll, and the second time we select keepers. Finally, after the third roll we score and move to a score state with one more category filled in. Resuming, there are $3 \cdot 252 + 2 \cdot 462 = 1680$ turn states reachable from any nonterminal score state.

All turn states can be split into groups of 1680 elements. Each group is associated to some nonterminal score state. Size of resources needed during computation depends only on the number of score states. This is because, as we will see later, at run time we can quickly calculate value of a given turn state from score state values. Therefore we should focus mainly on reducing the number of score states.

[2]Keeping 5 dice means that we do not roll at all. This way we can simulate immediate scoring after the first or second roll.

The simplest way to describe score state is scoring history — a sequence of filled categories and values put into them. From player's point of view such complete information is not needed. In fact, we are not interested in how we got to a particular state, but rather only interested in possibilities for future scoring. If scoring possibilities of two states with a different history are identical then they can be merged. Formally, we say that two states are identical iff subgraphs induced by outgoing edges from these states are isomorphic.

Using the above criterion we can extract the essential information sufficient to describe a score state. It turns out that for every category we only need binary information whether it has already been scored. Because of bonus rules we additionally need a ternary value instead of binary for one category, and also one number from the interval $[0, 63]$ to remember the sum of scores of six upper categories. This gives the total of $3 \cdot 2^{12} \cdot 64 = 786\,432$ score states in the game. A more careful analysis leads to a further reduction to no more than 40% of that number: 310 656 score states. For details, see [8].

3.2 Maximizing average score

The usual strategy is to optimize average score in the long run, i.e. we want to find a strategy giving us the highest possible expected value of the total score. Such optimal strategy can be constructed by propagating values back from final states to the initial state. We associate a single value to each state: the expected number of points that can be scored starting from this state using the optimal strategy. It means we take into account only those points which can be scored in succeeding turns.

The algorithm calculating all values is very simple. To find the expected score for some given state we take expected scores of its successors, and depending on type of the state we take the weighted sum (if it is a random event) or the maximum (if it is player's choice) (Figure 2).

(a) $E = p_1E_1 + p_2E_2 + p_3E_3$ (b) $E = \max(E_1, E_2, E_3)$

Figure 2: Calculation of the expected score.

We should note that if edge leading to a successor state represents writing score into a category, then the value E_i used in the formula is not directly the expected score of the successor state, but it must be increased by the total score gained by changing the score table.

In addition to the reduction of score states space, we made major optimizations in calculations of turn state values in a single turn. This is important because there we spend an overwhelming part of the computing time. From the formulas we see that the time is proportional to the number of edges. Appendix B describes how to reduce this number by an order of magnitude by means of dynamic programming.

Solving the whole game this way takes about 30 seconds on todays PC whereas Tom Verhoeff's implementation [11], which lacks edge reduction optimizations needs nearly 10 minutes. Some of these ideas have appeared in articles by James Glenn [3, 4]. 10 minutes is acceptable time to tabulate values, because once we do it we can use the values immediately in any game in every state. We note that values for turn states have to be recalculated from score states, and the time needed for this is negligible. Why then we need any optimizations? For the single player game they give a nice acceleration, which is going to benefit us when tackling harder problems such as maximizing the probability of reaching a particular score. We take advantage of optimizations especially in case of the multi-player version where the amount of time gained gets at least squared.

3.3 Maximizing the probability of beating a given score

Suppose that we are presented with a list of highest scores and we would like to put our name there. That means we no longer aim at maximizing the expected average score, but rather we fix some number and ask what is the optimal strategy to achieve a score of that number of points or more. Such a strategy should maximize the probability of reaching the given score.

We can try an approach analogous to maximal average score calculations. We want to propagate values back from the final states. The problem we have to deal with is what kind of value we should store in every state. Suppose we want to generate a strategy which maximizes the probability of reaching a fixed score s. For the initial state I we want to find a single real variable – the probability, which we denote as $\Pr(S_I \geq s)$. Consider all score states which can be reached in a single turn. What value we need from these states to be able to compute $\Pr(S_I \geq s)$? Let A be one of these states. The score state A has one category more filled in than the state I. Suppose this new category's score is v. That means that in the remainder of the game we need to get at least $s - v$ points. Therefore we need to calculate the value $\Pr(S_A \geq s - v)$. Unfortunately, the score v is not unique. Score put into some selected category may vary depending on dice values. Additionally we may reach the same state A with different edges with a different total score gain. Thus storing a single real variable in the state A is not sufficient.

In [1] the above problem is solved by recursive computations of all needed values from the initial state. Since the set of values to remember is definitely too large to fit in a memory, the main problem in such approach was an efficient caching structure. That solution is also too inefficient to use in a practice.

Our solution is to store not a single probability, but probabilities for all possible scores simultaneously. We achieve that by introducing a *distribution* structure. For our needs by a distribution we denote a non-increasing function P mapping integers to a real interval $[0, 1]$. $P(s)$ for an integer s denotes the probability of reaching a score s. Instead of storing a single value in a state we store the whole distribution. Because for almost all possible scores s, the probability of reaching s is 1 or 0, we need little memory to store a typical distribution. Namely, for the distribution P we need:

- the smallest integer s_0 such that $P(s_0) < 1$,
- the largest integer s_1 such that $P(s_1) > 0$,

- the vector of values $[P(s_0), P(s_0+1), \ldots, P(s_1)]$.

There are several advantages of the presented approach. First is that we are able to propagate distributions from the final states back to the initial state similarly as we did it for the maximal average score strategy. The only difference is that we operate on distributions instead of real values. Moreover, we calculate all values simultaneously and the problem of which probabilities to remember simply does not exist. The operations needed in calculations of the maximum average score translate to corresponding operations on distributions (Appendix C).

Second advantage is that in fact we reduce the number of values we need to memorize especially for states near the end of the game. For instance in the final state the stored distribution is represented by $s_0 = 1$, $s_1 = 0$ and the empty vector of values, which means that the probability of reaching any positive score is 0 and any non-positive is 1.

Our main profit is possibility to store distributions of score states on a hard disk similarly as we did it with the optimal expected scores. If we need to know a score state we simply read it from the disk; in case of a turn state we read a few distributions of score states we may reach within a single turn and propagate values back to it. Calculation of distributions for all states in a single turn takes no more than 150 milliseconds on our computer, which makes the approach practical.

Size of the tables is 1.3 GB; calculation takes 6 hours on a 2.8 GHz processor. Figure 3 shows the distribution of the initial state. This curve matches the one generated in [1].

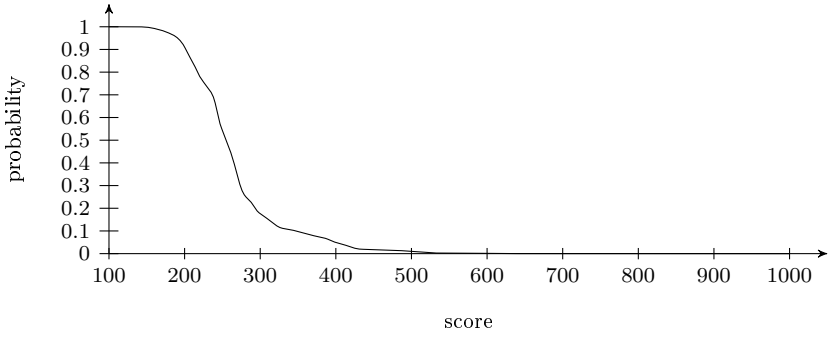

Figure 3: Distribution at the beginning of the game

4 Multi-player game

In this case strategy of maximizing average score is no longer optimal. This is because we should adjust our strategy depending on the scores of the opponents. If we are trailing by many points we should take higher risks; conversely, if we are leading with a clear advantage we should play it safe.

There are several choices what a "good" strategy should aim at. One is to always maximize the probability of winning the game. But if we are in situation when winning is almost impossible then we should maximize the probability of

taking the second place. Generally, the most natural strategy is to maximize our expected rank, and we will focus on that.

For the two-player game maximizing the expected rank is almost the same as maximizing the probability of winning. Only almost because there is possibility of a draw when both players finish a game with the same score. Because this is a zero-sum game the optimal strategy exists and we can try compute it. We make an analysis of the two-player game in Appendix D. Conclusion is that in this case top hardware available today is sufficient to tabulate the optimal strategy.

In case of more than two players, finding optimal strategy is more complex because of several reasons like opponents' cooperation. See for example [10] for discussion about problems that occur in multi-player games. Because of those additional difficulties and the size of the state space we have not tried to find the optimal strategy in this case.

4.1 Nearly optimal multi-player strategy

Resources needed to construct the optimal strategy are huge and – in case of more than two players – clearly beyond capabilities of today's hardware. Therefore for the multi-player game, heuristic methods are required.

To build an evaluation function of the states, distributions – developed for the strategy of maximizing the probability of reaching a given score – become helpful. For every player we are able to instantly generate a distribution of his state. Such distribution gives us some information about his scoring possibilities. By combining distributions from all players we can construct effective evaluation functions.

We have tried several different functions, but in the end the simplest approach turned out to give extremely good results. Namely, we can treat a distribution describing maximal probabilities of reaching successive scores as a real probability distribution of player's total score. That means, if we know that the maximum probability of reaching scores s and $s+1$ are $\Pr(S \geq s)$ and $\Pr(S \geq s+1)$ respectively, then we assume that the probability of player's total score in the end of the game will be exactly s equals to $\Pr(S \geq s) - \Pr(S \geq s+1)$. Now, having probability distributions of total scores for all players we can arrive at many different statistics such as the expected rank, which is what we actually need. We call this evaluation function *EvalEMP*[3].

There is a drawback with the above approach. The probability distribution of total score we get that way cannot represent a probability distribution of the total score of any true strategy. This is because the expected value of the total score is greater than the value got from the strategy with the highest possible expected score. For instance in the beginning of the game the highest possible expected score is 254.589, but the expected value of the distribution generated for this state is 264.449. Nonetheless, total score probability distributions we construct represent scoring possibilities of a player with the caveat that the expected value is inflated. Fortunately, this side effect occurs for every player so probability distributions of all players still can be used to estimate their expected ranks.

The described heuristic is surprisingly highly accurate. In fact, coupled with

[3]EMP is abbreviation of Expected Match Points. This is an equivalent measure to the expected rank, but we want to be consistent with the work [8].

a few moves deep recursive search, it results in a strategy which is "almost" optimal. We demonstrate it experimentally.

Let the strategy using evaluation function *EvalEMP* coupled with a recursive search of depth d be denoted as *EMP*(d). If $d = 0$, we apply the evaluation function immediately to the states and the best move is the one with the highest value. For $d = 1, 2, \ldots$ we do the recursive search suspending the evaluation d turns later.

4.2 Comparison to the optimal strategy

How to measure the strength of a heuristic strategy? For a random game like Yahtzee the most accurate way should be comparison to the optimal strategy.

Suppose we have a game and we would like to extract information about quality of moves in this game. Let us pick up some situation. Assume there are n possible moves. Let $s_1, \ldots, s_n$ denote situations each of those moves leads to. Suppose we are able to apply the optimal strategy for this situations. That means we can calculate optimal values $v(s_1), \ldots, v(s_n)$. Without loss of generality we can assume that $v(s_1) \geq v(s_2) \geq \ldots \geq v(s_n)$. So the move s_1 is the optimal one. If a non-optimal strategy makes another move s_i then the error of this move is $e = v(s_1) - v(s_i)$. Intuitively, this value means that if we make the move s_i we, on average, will win e games less than we would if we played optimally.

Now, to measure the strength of a strategy S we generate a vast number of random games. For every situation in every game we apply the strategy S to see what move it proposes and using the optimal strategy we evaluate the error value of this move. Finally, we take an average over all error values as the measure of the strategy S.

Of course to perform such measurement, we have to be able to apply the optimal strategy for every situation. We were able to do it only for some states, namely for all states where the total number of empty categories (of both players combined) does not exceed 10. We have calculated values for all possible states with 5 or less empty categories, and stored these tables to the disk. For states with 6 or more empty categories we used recursive search to a depth reaching stored tables.

Another point, which we should be careful about is generation of random games. In fact the games should not be truly random. They should be "typical games", i.e. games played by "typical players". Because a game depends not only on player's choices, but also highly on random events, for a typical player we could take any strategy which does not make any flagrantly bad moves. Therefore by the typical player we take the *EMP*(1) strategy. Moreover we add randomness to the selection of the moves in such a way that:

- the best move (in the view of the *EMP*(1) strategy) is played with the highest probability,
- moves with a similar value are played with a similar probability (so if there is more than one good move with almost equal values then any of them could be played with a reasonable probability),
- weak moves should almost never be played.

We achieve that by selecting moves randomly with the weight:

$$w(s_i) = e^{100\cdot\left(v(s_i)-\max_j v(s_j)\right)},$$

where $v(s_i)$ is the value of the move s_i.

For each number of empty categories (of both players combined) we generated games separately. This number D, the "depth" of the game, varies from 1 to 10. For each depth we have generated $2^{16} = 65\,536$ games, so the total number of generated games is 655 360. The results are presented in Figure 4.

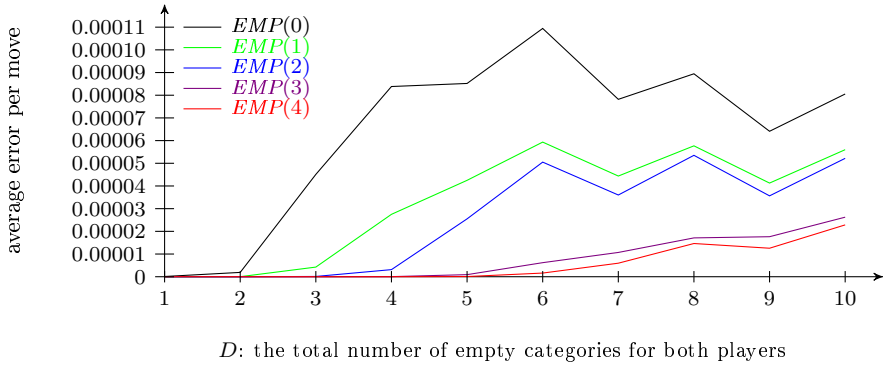

Figure 4: The strength of $EMP(\cdot)$ strategy.

We can see some regularities. The most important is that the accuracy of the strategy is better if the search depth is higher. For the discussion of other regularities we refer to [8].

We can see that the average error per move is very low. To get an intuition how low this number is, suppose we play a game using the strategy $EMP(1)$. We see that the highest average error produced by this strategy occurs at $D = 6$. It is very probable that the average error does not increase for larger D, even for $D > 10$. We argue that the more moves remain to the end of the game the less important the value of a move is. This is because the game is random, and influence of a wrong move is smaller if we are at the beginning of the game. We are thus safe to assume that the average error per move of the strategy $EMP(1)$ is less than 0.00006. We have at most $13 \cdot 3 = 39$ moves per game, thus the sum of errors per game does not exceed 0.00234. This number can be thought of in the following way: having played 1000 games, we lose 2.34 games more by using the $EMP(1)$ strategy instead of the optimal strategy. This is only a rough estimation; actual result is even better especially in case of strategies $EMP(d)$ with larger d.

4.3 Comparison to other strategies

In the previous section we showed experimentally that quality of the $EMP(\cdot)$ strategy is close to the optimal strategy in the sense of the average error per move or per game. One may ask what is the order of magnitude of this average error? Are there better strategies or maybe humans are able to achieve lower error rates? We have tried several different strategies and here we present the results. In the next section we show quality of human play.

We have searched among various heuristic evaluation functions, but *EvalEMP* was the best one. Having distributions for all players we can build another natural evaluation function as follows. Suppose our distribution is P and opponent distributions are P_i where i iterates through all opponents. Assume further we aim at maximizing our probability to reach score s and our chances equal to $P(s)$. Now we make an assumption that every opponent also aims to maximize the probability of beating the score s and we win with him if and only if he does not manage it. i-th opponent does not reach the score s with the probability $1 - P_i(s)$, thus the expected number of beaten opponents is

$$P(s) \sum_i \big(1 - P_i(s)\big). \tag{1}$$

The last step is to choose s. We simply maximize (1) over all possible values of s. Note that s may vary during the game, therefore any strategy based on this evaluation function can be called a *dynamic fixed score strategy*. We denote this heuristic as *EvalDynS*.

When we created strategies $EMP(d)$ we based them on the evaluation function *EvalEMP*. Similarly here we build strategies based on *EvalDynS*. We call them $DynS(d)$ where d denotes the depth of the recursion as in $EMP(d)$. Figure 5 presents quality of introduced strategies. Measurement of the quality was done in the same way like in the $EMP(\cdot)$ case (see Section 4.2). For comparison,

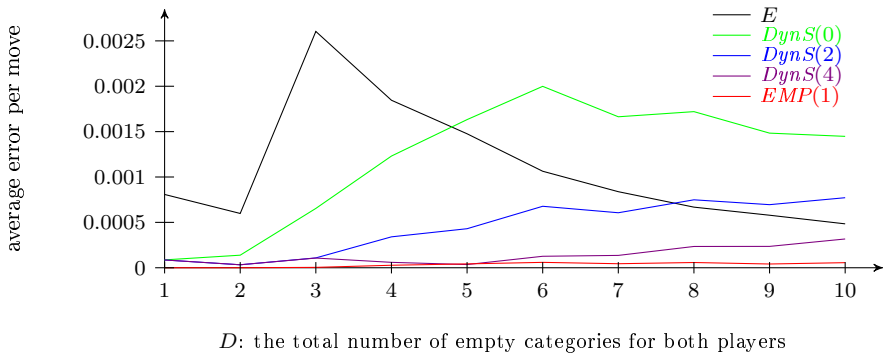

Figure 5: Strength comparison of strategies $DynS(\cdot)$ and the single player strategy maximizing average score (denoted as E) to $EMP(1)$.

we also add the single player strategy maximizing average score and strategy $EMP(1)$.

As we may expect quality of the $DynS(d)$ strategy increases with depth d. Nevertheless, the average error is sufficiently larger than for the $EMP(1)$ strategy. This tells us that either the strategy $DynS(\cdot)$ is weak, or the strategy $EMP(1)$ is unusually strong. The first hypothesis seems to be false, because for instance strategy $DynS(4)$ is definitely stronger than strategy E. In the next section it is shown that the best humans play no better than strategy E, which additionally diminishes chances that somewhat natural strategy $DynS(\cdot)$ is accidentally badly developed. Moreover, comparison was made against $EMP(1)$. Remember that strategies $EMP(d)$ for larger d are more and more effective. We conclude that the heuristic *EvalEMP* and strategies $EMP(\cdot)$ based on this heuristic are outstandingly accurate.

4.4 Comparison to human players

Thanks to the game service `playok.com` (also known as `kurnik.pl`) [2] we were able to analyze a vast number of human games played in 2007. We took 23 352 929 two–person games played by a total of 209 207 registered users. For this games we calculated move errors by comparing the moves to the optimal strategy similarly as we did in the previous sections. Calculation of the optimal values for such huge number of games was possible only for positions with 8 or less empty categories for both players (18 or more filled).

For such positions the average error per move of the strategy $EMP(1)$ is about 0.000 021 and for the strategy of maximizing average score equals approximately 0.001 24. For humans this statistic looks much worse. An average human error per move oscillates around 0.003. The average error of top 10 humans (selected among players with more than 500 games) is below 0.001 35 and the best 3 is below 0.001 23 which is more than 50 times worse than strategy $EMP(1)$. It also means that only the best humans have a shot at playing better than the optimal strategy for maximizing average score.

By applying our "almost" optimal strategy, at a 100 games distance, we were able – on average – to win about 4 games more with best humans up to 9 games more with average players. These numbers tell us that advantages of strong strategies is observable only after a long run of games, which confirms random character of Yahtzee.

5 Conclusions and further applications

Our work presents deep analysis of Yahtzee. For the single player version we made an efficient implementation of the maximizing average score strategy with important optimizations and state reductions. In case of the strategy which tries to maximize the probability of reaching a fixed score, we introduced the term of distribution, and we were able to tabulate all values.

New results are presented in case of the multi-player version. In case of the two-player game we have calculated the size of resources needed to solve it. We have created software which is able to perform all calculations. We have also constructed heuristic strategy $EMP(\cdot)$ using distributions – previously introduced for the single player variant. This strategy performs "almost" as good as the optimal strategy. It was the most surprising discovery and to justify the claim we arranged series of experiments comparing this strategy to the optimal one, other strategies and humans.

Introduction of distributions and building heuristic functions based on them seems to be a new approach which can be applied to other games. Such game should be a full information game. Player states should be separated in such a way that scoring and moving possibilities of one player should not depend on a state of another player, although the final result can be created from scores of all players. *Can't stop* is a game which fulfills those criteria. Recently, this game has been analyzed by Glenn [5, 6]. It is much harder to analyze than Yahtzee because it contains cycles, and to propagate values we need more sophisticated techniques. Multi-player version of this game is definitively too complex to tabulate the optimal strategy. Distributions approach could lead to an "almost" optimal strategy.

References

[1] C.J.F. Cremers. How best to beat high scores in Yahtzee: A caching structure for evaluating large recurrent functions. Master's thesis, Fac. of Math. and CS, Technische Universiteit Eindhoven, The Netherlands, 2002.

[2] Marek Futrega. PlayOK.com – free online games (kurnik), 2007. `http://www.playok.com/`.

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

[4] James Glenn. Computer strategies for solitaire Yahtzee. In *2007 IEEE Symposium on Computational Intelligence and Games (CIG2007)*, pages 132–139, 2007.

[5] James Glenn, Haw-ren Fang, and Clyde P. Kruskal. A retrograde approximation algorithm for one-player Can't Stop. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *5th International Conference on Computers and Games (CG2006)*, volume 4630 of *Lecture Notes in Computer Science*, pages 148–159. 2007.

[6] James Glenn, Haw-ren Fang, and Clyde P. Kruskal. A retrograde approximation algorithm for two-player Can't Stop. In *Computers and Games Workshop*, 2007.

[7] The great idea finder – Yahtzee, 2006. `http://www.ideafinder.com/history/inventions/yahtzee.htm`.

[8] Jakub Pawlewicz. *Algorithmic techniques for solving games on the example of dice game Yahtzee.* PhD thesis, University of Warsaw, 2009. (In Polish).

[9] Jakub Pawlewicz. A nearly optimal computer player in multi-player Yahtzee (full version). 2010. `http://www.mimuw.edu.pl/~pan/papers/yahtzee.pdf`.

[10] Nathan Sturtevant. A comparison of algorithms for multi-player games. In Jaap van den Herik, Jos W.H.M. Uiterwijk, Martin Müller, and Eric Postma, editors, *3rd International Conference on Computers and Games (CG2002)*, volume 2883 of *Lecture Notes in Computer Science*, pages 108–122. 2003.

[11] Tom Verhoeff. Optimal solitaire Yahtzee advisor and Yahtzee proficiency test, 1999. `http://www.win.tue.nl/~wstomv/misc/yahtzee/`.

[12] Phil Woodward. Yahtzee: The solution. *Chance*, 16(1):17–20, 2003.

Upper section		
Category	**Condition**	**Expression**
Aces	true	1 × the number of ones
Twos	true	2 × the number of twos
Threes	true	3 × the number of threes
Fours	true	4 × the number of fours
Fives	true	5 × the number of fives
Sixes	true	6 × the number of sixes
Lower section		
Category	**Condition**	**Expression**
Three of a Kind	≥ 3 same values	the sum of all values
Four of a Kind	≥ 4 same values	the sum of all values
Full House	3 same values + 2 other same values (∗)	25
Small Straight	4 subsequent values	30
Large Straight	5 subsequent values	40
Yahtzee	5 same values	50
Chance	true	the sum of all values

(∗) Values in triple and values in double must differ.

Table 1: Scoring rules.

A Yahtzee scoring rules

A.1 Categories

Categories are split into two sections: upper and lower. Number of points which should be written into a category is derived from the following rule:

$$\textbf{if } \textit{condition} \textbf{ then } \textit{expression} \textbf{ else } 0, \tag{2}$$

where *condition* describes accepted combination of values (the number of pips) on all 5 dice and *expression* provides a formula for evaluation of the actual score. Table 1 lists conditions and expressions for all 13 categories.

A.1.1 Joker

If one rolls a dice combination with all the same values (such a roll is called Yahtzee) then one can take advantage of the *Joker* rule. Having a Joker we can fill any category provided that

(1) the category is in the upper section and it corresponds to the value on dice, or

(2) any other category, but under a condition that the corresponding category in the upper section has already been filled.

Scoring as Joker changes the condition in rule (2) to true.

A.2 Bonuses

In the end of the game the total score is calculated by summing up scores from all 13 categories plus bonus points. There are two bonuses:

- upper section bonus,
- Joker bonus (called also Yahtzee bonus).

A.2.1 Upper section bonus

Upper section bonus of 35 points is awarded when the sum of scores from all six upper categories reaches threshold of 63 points. The threshold is chosen in a such way that it can be obtained by throwing six times exactly three dice of the same value. In such case the sum of scores in upper categories would equal to $3 \cdot (1+2+3+4+5+6) = 63$. Nevertheless, reaching the threshold also can be done in any other way.

A.2.2 Joker bonus

For each used Joker the bonus of 100 points is added but only if Yahtzee category is filled with 50 points. This bonus is not awarded if Yahtzee category is empty or scored by 0 points. In particular the bonus is not assigned for scoring 50 points to Yahtzee category.

Joker bonus is also known as Yahtzee bonus, because each Joker arises from rolling another Yahtzee (5 same dice).

B Optimizing calculations of turn state values

There are two types of turn nodes.

1. Roll nodes. These nodes represent states after player's roll. We characterize a roll state by a roll number in a turn and a sequence of five dice. Depending on the roll number, in this state player has to decide which dice to keep or what category to store.

2. Keeper nodes. These nodes represent states after player's choice of keepers. A keeper state is described by a roll number and a sequence of no more than five dice. This state is a random event; all its edges lead to roll states.

Every turn state is associated to a sequence of dice. This sequence can be represented by a multiset with values belonging to interval from 1 to 6. For a roll node cardinality of the multiset must equal 5, and for a keeper node it must be at most 5. Denote the set of all multisets containing values $\{1, 2, \ldots, 6\}$ with a cardinality at most 5 as $\mathcal{K}$. Let $\mathcal{R} \subseteq \mathcal{K}$ denote the set of all multisets with a cardinality exactly 5. $\mathcal{K}$ is a set of all possible dice keepings. Each $K \in \mathcal{K}$ represents some possible keepers. $\mathcal{R}$ is a set of all possible dice rolls. Each $R \in \mathcal{R}$ represents some possible roll.

B.1 Keeper nodes

Before the second or third roll in a turn, player selects some dice to roll and decides to keep the rest of dice. Let $K \in \mathcal{K}$ be a multiset of values of kept dice. Rolling selected dice can give a roll $R \in \mathcal{R}$ such that $K \subseteq R$. Denote as $\Pr(R|K)$ the probability of getting roll R starting from keeper state K. Suppose

we have calculated values for all possible rolls. The value of roll R is stored as $roll[R]$. We want to calculate the value of keeping K and store it as $keep[K]$. The straightforward formula is:

$$keep[K] = \sum_{R \supseteq K} \Pr(R|K) roll[R]. \tag{3}$$

The size of the sum on the right side grows exponentially in the cardinality of K. This sum can be substantially reduced when we want to compute values $keep[K]$ for all $K \in \mathcal{K}$ simultaneously. We achieve that by the following formula:

$$keep[K] = \begin{cases} roll[K] & \text{for } |K| = 5 \\ \frac{1}{6} \sum_{d=1}^{6} keep[K \cup \{d\}] & \text{for } |K| < 5 \end{cases} \tag{4}$$

where $|K|$ denotes the cardinality of K. Sum index d iterates over all possible die values. The trick is to reuse computation for $keep[K]$ with the higher cardinality $|K|$. That way we reduce the number of "outgoing edges" in (3) to constant 6 – the number of die sides in (4). For the proof of the formula (4), we refer to [8, fact 3.11].

B.2 Roll nodes

Here we are after the first or the second roll in a turn. Suppose player rolled R. Player has to choose keepers K such that $K \subseteq R$. Having values of all keepings K as $keep[K]$ we simply maximize value $roll[R]$ of roll node R:

$$roll[R] = \max_{K \subseteq R} keep[K]. \tag{5}$$

The maximum on the right side is taken over all subsets of R, which can be as large as 2 powered to the number of dice, that means it is exponential in the number of dice. We can use dynamic programming to accelerate computations. The definition of array $roll$ can be extended to indices from set $\mathcal{R}$ to set $\mathcal{K}$ by the equation:

$$roll[K] = \max_{K' \subseteq K} keep[K'] \quad \text{for } K \in \mathcal{K}.$$

Array $roll$ can be computed for all $K \in \mathcal{K}$ in order of an increasing cardinality $|K|$ by the smaller formula:

$$roll[K] = \max\big(\{keep[K]\} \cup \{roll[K \setminus d] \,|\, d \in K\}\big). \tag{6}$$

The maximum on the right hand side of (6) is now taken over the set of size at most one plus the number of dice. This is an order of magnitude less than in (5).

C Distribution operations

C.1 Weighted sum

This operation is performed in random event nodes.

Let A be a set of all possible events for a such node. Let the probability of event $a \in A$ be $\Pr(a)$. Suppose that for each event $a \in A$ we calculated

score distribution P_a. What is the distribution for the considered node? We can calculate the probability of reaching a score s with the formula:

$$P(s) = \sum_{a \in A} \Pr(a) \cdot P_a(s). \tag{7}$$

Distribution P is called a *weighted distribution* of distributions P_a, if for all integer s the equality (7) holds, and we denote it by:

$$P = \sum_{a \in A} \Pr(a) \cdot P_a.$$

C.2 Maximum

In case of nodes where player has a choice, we take the maximum.

Let B be a set of all possible player's choices for a such node. Suppose for each choice $b \in B$ we calculated score distribution P_b. Now, how do we calculate the distribution for this node? To reach a score s with the maximal probability, we should choose such b that $P_b(s)$ is the largest, therefore:

$$P(s) = \max_{b \in B} P_b(s). \tag{8}$$

Distribution P is called a *maximum distribution* of distributions P_b, if for all integer s the equality (8) holds, and we denote it by:

$$P = \max_{b \in B} P_b.$$

C.3 Shift

After the last roll in a turn player chooses a category to score. In this case we have to prepare distributions for all choices and take the maximum over them.

Let us fix some empty category. Suppose that by filling it we can score p points and the distribution of the resulting state is P. For any s, probability of reaching score $s + p$ is now $P(s)$ because we already have p points. Therefore distribution representing this choice is a *shift distribution* denoted by $P+p$. For all integers s the following holds:

$$(P + p)(s + p) = P(s).$$

D Two-player state space analysis

Armed with a highly optimized calculation of the optimal strategy for the single player variant we can try and attack the two-player game. We have not found any detailed analysis of the complexity of the two-player version. Previous works contain only rough estimations like in [3] Glenn limits the number of states by 2^{48} which is too high for current hardware.

The optimal strategy in the two-player game is maximizing the expected rank which is, as mentioned earlier, almost the same as maximizing the probability of winning with the opponent. The optimal strategy can be found by propagating a single value back through the state space. This value describes the highest

possible expected rank of the player to move. To check whether this procedure is feasible we compute the exact number of states in this variant of the game.

Every state has to describe the current score state for each player. In any moment of the game one player can move and this player can go into turn states. We are only interested in score states, i.e. the states which appear between turns after scoring a category by one player and before the first roll by the second player. In fact to represent a score state we need the following information:

- a score state for each player,
- the total score difference between both players.

The highest possible total score is 1575. Applying the known number of states for the single player game we get the first estimate of the number of states $310\,656^2 \cdot 2 \cdot 1575 \approx 3 \cdot 10^{14}$. Roughly the same estimation was made by Glenn [3]. This number can be significantly reduced. There are some constraints on states which can be taken into account.

1. Player which started the game must have the same number of filled categories as the second player or one more if the second player is to move.
2. Suppose that at one point the difference between the scores of both players is so high, that even if the trailing player keeps scoring maximum available points and the other one a minimum, still the other player will end up winning. This means that each state implies both lower and upper bounds on the score difference.
3. We can also limit the total score difference right at the beginning of the game. In order to a state with a given list of categories filled in, the players can score only points available in these categories. We can find the minimum and the maximum number of points a player can score for a given score state and from this information we can deduce the bounds for the total score difference in a two-player game state.

Applying the above observations we arrive at a total number of states equal to 12 671 976 997 282, which is 24 times smaller than the first estimate. Assuming we use 8 bytes for a single value (double precision fixed point), we need 92 TB to memorize values for all states. We also estimate calculation time to be 120 GHz years using our optimized implementation. Such resources are within reach of today's clusters, especially because the computations can be easily parallelized.