CSC 2534

Decision Making Under Uncertainty

Final project: Yaht

Ady Ecker

1 Introduction

In this project I analyze a "real-world" traditional dice game known as "Yaht" or "Yahtzee". In essence, the game is a Markov decision process. This game is tractable since it is fully observable, has finite horizon, and relatively small number of states. The first part of the work was an exploration of the game's structure. This analysis led to a compact representation of the state transitions. Based on these ideas, the complete value function for obtaining the maximal expected score could be computed. Having the optimal value function, I could evaluate several simple heuristic strategies that approximate the value function. While some heuristics had a poor estimation of the value function, they actually performed near-to-optimal in simulations. After analyzing the expected scores, I looked their variances. The variance of the scores under the optimal policy was computed for every state. Based on these variances, I implemented a simplified multi-player strategy that slightly improves the probability to win against strategies that don't look at their opponent.

2 Game rules

There are several different versions of Yaht rules. Most computer programs differ in the amounts of bonus points they give in various cases. I implemented a simplified version without bonuses. The same approach could be used to handle bonuses. A traditional version of the game has more sophisticated rules (e.g. reserving rolls at one turn and transferring them to another turn) that are not modeled by any computer program I have seen, including mine.

The program assumes the following:

- There are players taking turns.
- The winner is the one who got the maximal score.
- In each turn, a player gets points for achieving one sub-goal out of 13 possible sub-goals.
- If a player had already achieved a specific sub-goal, he cannot get points for that sub-goal later in the game.
- To make his turn, the player rolls 5 dice. He can select a subset of the dice to roll again. Than he may select a subset of the dice to roll again. After the third roll he must assign the outcome of the 5 dice to a sub-goal.
- If an outcome of the dice doesn't fit the sub-goal to which it is assigned, the player gets 0 points for that sub-goal.

The scores for the sub-goals are summarized in the following table. The order of the dice is not important.

Sub-goal	Structure	Score
Ones	[1 *n* times][5-*n* other faces]	*n*
Twos	[2 *n* times][5-*n* other faces]	2*n*
Threes	[3 *n* times][5-*n* other faces]	3*n*
Fours	[4 *n* times][5-*n* other faces]	4*n*
Fives	[5 *n* times][5-*n* other faces]	5*n*
Sixes	[6 *n* times][5-*n* other faces]	6*n*
Three of a kind	*xxxyz*	3*x*+*y*+*z*
Four of a kind	*xxxxy*	4*x*+*y*
Full house	*xxxyy*	25
Small straight	1234*x* or 2345*x* or 3456*x*	35
Large straight	12345 or 23456	40
Yaht	*xxxxx*	50
Chance	*vwxyz*	*v*+*w*+*x*+*y*+*z*

3 Game analysis

This section presents an analysis of a game with a single player whose objective is to maximize the expected score. The aim of this section is to show that the game is indeed tractable. The structure of the game leads to a very compact representation of the states. The stages within each turn (the three rolls) are represented as a DAG. The stages between turns are managed using a value function table. Simple combinatorics shows that the size of the state-space is relatively small, so the value function can be computed by dynamic programming.

3.1 Decisions in the game

Before we get into the combinatorial calculations, it is worthwhile to understand what decisions the player has to do and what is the information needed to make these decisions. Each turn involves three decisions in three stages:

- At the beginning of a turn, rolling the 5 dice is compulsory. No decision is made.
- Stage 1: Decide which subsets of dice to roll again.
- Stage 2: Decide which subsets of dice to roll again.
- Stage 3: Assign the outcome of the dice to a sub-goal (the outcomes fit more than one sub-goal. For instance, every outcome fits the "chance" sub-goal).

In order to make his decision, the player has to consider three competing factors:

- The player would like to maximize the expected gain from each turn. The information needed to describe the state within each turn includes the current state of the five dice, the stage number (1,2 or 3), and the set of remaining sub-goals.

The performance of the player in previous turns is irrelevant and therefore the decision process is Markovian.

- In order to maximize the score of the whole game, the player should direct himself to selection of sub-goals such that the gain from the present turn plus the expected reward from achieving the remaining sub-goals in future turns is maximal. When a player can assign an outcome to two sub-goals, he may decide to go for a sub-goal that yields lower score in the present turn if he expects to get higher scores from the other sub-goal in future turns. The player needs a value function to tell him the expected reward for every set of remaining sub-goals in future turns.
- In a multi-players game, a player would strive to maximize his probability to win, rather than maximize his expected score. A player may play conservative moves when he believes he leads the game and risky moves when he feels behind. The information needed to make a decision is the current state of all players, including their current score.

3.2 Combinatorics

Since the order of the dice is irrelevant, it is more convenient to represent the outcomes of the dice as a vector $(n_1,n_2,n_3,n_4,n_5,n_6)$, where n_i stands for the number of times face i showed up. When rolling k dice, we have $0 \le n_i$ and $n_1+n_2+n_3+n_4+n_5+n_6=k \le 5$.

Given that the current state of the dice is $(n_1,n_2,n_3,n_4,n_5,n_6)$, a roll-move is another vector $(m_1,m_2,m_3,m_4,m_5,m_6)$, such that $0 \le m_i \le n_i$. m_i counts the number of i-faces rolled. For instance, if the current configuration of the dice has two dice showing "1", the player can roll 0, 1 or 2 dice showing "1".

This representation makes it easier to enumerate all the possible roll-moves for each configuration of the dice. The number of possible moves is just $\Pi(1+n_i) \le 2^5=32$. This is because we have to choose a subset of the 5 dice to roll, and there are no more than 32 such different sub-sets (in case some faces in the current configuration are the same, there will be less possible moves).

To simulate the move, we roll $m_1+m_2+m_3+m_4+m_5+m_6$ dice and obtain a random result $(r_1,r_2,r_3,r_4,r_5,r_6)$. The new configuration of the dice is simply the vector whose components are $n_i-m_i+r_i$.

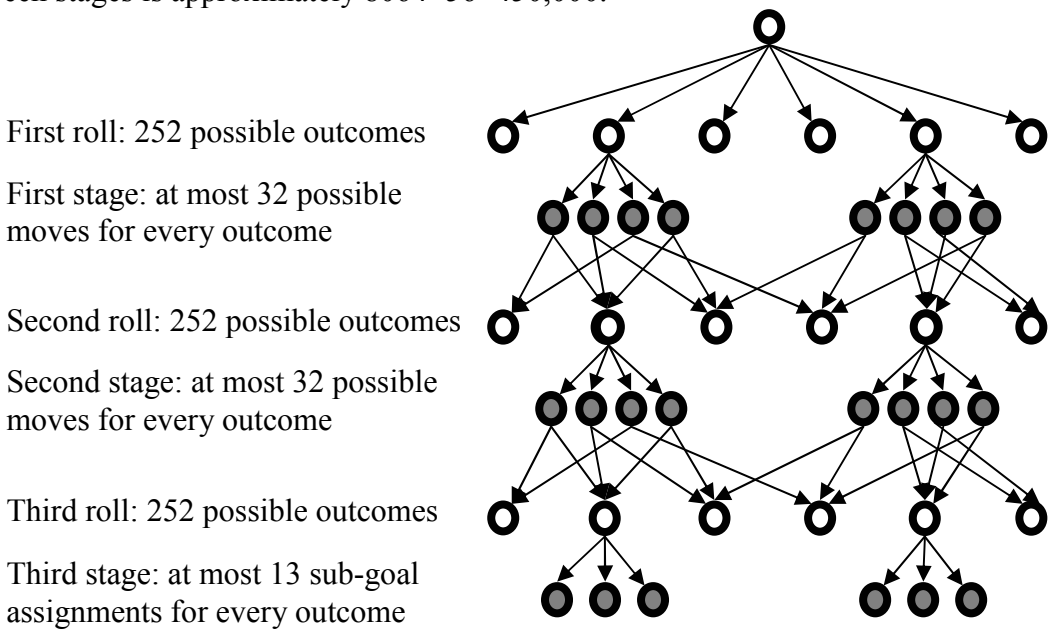
Obviously, the results of rolling the dice in this representation are not equiprobable. Rolling $k=m_1+m_2+m_3+m_4+m_5+m_6$ dice, the probability to obtain the outcome vector $(r_1,r_2,r_3,r_4,r_5,r_6)$ is $\frac{k!}{6^k r_1!r_2!r_3!r_4!r_5!r_6!}$.

Our next step is to count how many vector outcomes are possible when rolling k dice, using a standard argument: Consider throwing 5 balls to an array of k+5 bins (each bin can hold one ball). Every outcome of this experiment could be linked to our case, by

setting r_1 to be the number of empty bins before the first ball, r_2 the number of empty bins between the first ball to the second ball, etc (the balls act like splitters, and there are going to be exactly k empty bins). The number of configurations is $\binom{k+5}{5}$. Rolling k=5 dice, there are only $\binom{10}{5}=252$ different outcome vectors (much less than 6^5=7776 unordered outcomes). For k=4, we have 126 possible outcomes, and for k=3 there are 56.

3.3 The DAG

The calculations made in the previous section suggest we can represent each turn in a very compact way as a DAG (see figure below). At the beginning of a turn, 5 dice are rolled. There are 252 possible vector outcomes. For each outcome, there are no more than 32 possible roll-moves in stage one. Altogether there are no more than 252×32=8064 possible moves at stage one. However, after the moves of stage one are performed and the dice are rolled, the configuration of the dice must go back to one of the 252 possible vectors. There are essentially many ways to reach the same configuration at the beginning of stage two. Once a configuration is reached at, it doesn't matter how we got there from stage one. At stage two we have again no more than 32 moves per outcome, which will take us once more to one of the 252 states. At stage 3, there are at most 13 possible assignments of the outcome to a sub-goal, contributing 252×13=3276 leaf nodes to the DAG. Note that the number of edges between two stages has reasonable limits. For each outcome, there is only one move that rolls all five dice (contributing 252 edges to the next stage), and only five moves that roll four dice (contributing 126 edges each). Most moves will contribute no more than 56 edges (rolling three dice), so the number of edges between stages is approximately 8064×56≈450,000.



3.4 The value function

Given the number of states in the DAG of a single turn, a natural question is how to represent the value function for the whole game. The idea is to identify critical points along the game where the state of the player contains less information. Observe that after a turn is completed, the state is determined only by the set of sub-goals that are remained to be achieved (the configuration of the dice is irrelevant). We are going to maintain the expected score only for these states. There are only 2^{13}=8192 possible sets of remaining goals for which the value function has to be specified. Mathematically, the value function is a function V: $2^{\{1,2,\ldots,13\}} \rightarrow \Re$ (that is, a function whose domain is sets).

For states within turns, we can use the DAG to compute the expected reward. Recall that the leaves of the DAG represent assignments for specific sub-goals. At run-time, we can determine for each leaf whether the assignment is legal (that is, the sub-goal was not already assigned in the game). Using the value function table we can predict the expected score from the sub-goals that will remain after the assignment. We can compute the expected score for each node in the DAG in the standard way, going from the leaves layer towards the root, computing the expectation at each chance node and choosing the best move at each decision node.

An important implementation detail is that the DAG is static and constructed only once. The transition probabilities, the connections between moves to outcomes and the gains from each assignment are computed when the program starts and don't have an effect on the running time. The DAG and the computation procedure are the same for all turns, all strategies and all players. The only difference is in the values that are loaded at the leaves of the DAG at run-time. The flexibility that remains in designing a strategy is in specifying the value function.

4 Results

In this section several value functions are compared. Some of these functions are close to the optimal function, and some are simple heuristics. However, all of them use the same optimal DAG to choose their moves within turns. It turns out that the use of this optimal component leads to near-optimal performance even when the estimation of the value function is far from the truth.

4.1 Examined strategies

4.1.1 Maximal expectation strategy

This strategy computes the optimal value function for achieving the maximal expected score in a single player game. It goes over the sub-sets of remaining sub-goals in order of

their size, i.e. starts with the last stage-to-go, than handles all possibilities for two-stages-to-go and so on. The computation over the 8192 states takes about 10 minutes.

4.1.2 Additive strategy

Let $S=\{g_1,\ldots,g_k\}$ be a set of remaining goals. Each goal i can contribute $V(\{g_i\})$ when it is maximized separately (for example, in the last turn). Computing $V(\{g_i\})$ is straightforward from the DAG. A simple approximation to the value function is $V(S)=\sum V(\{g_i\})$. This value is a lower bound on the optimal value function, since we can always decide beforehand which goals we would like to go for in each turn (in the real game we can change our mind after the rolls of the dice).

4.1.3 Regression strategy

In this strategy, the value function is modeled by affine function: $V(S)=a_0+\sum a_i I_i$, where $I_i=1$ if $g_i \in S$ and 0 otherwise. Instead of learning the model by reinforcement techniques, I fitted the affine model to the optimal value function. My aim was to test whether the structure of the optimal function could be modeled by affine function.

4.1.4 Small regression strategy

This model is very similar to the regression model above, except of two differences. First, the affine model is fitted to the optimal value function only on the last four-stages-to-go. The question is whether an affine model on the last four turns can predict the value function for all 13 turns. The second change is that the number of stages-to-go was used as an additional variable in the regression. The value function gets bigger with more stages to go, since there is more possibility for synergy between the remaining goals.

4.1.5 Max DP strategy

This strategy begins by computing the optimal function for the last four stages. For the optimal value function we must have $V(S) \geq max(V(S')+V(S-S'))$, where the max ranges over all subsets S' of S. This is because we can decide to optimize first on the goals in S', and then go for the remaining goals in $S-S'$. We can approximate the value function by setting $V(S)=max(V(S')+V(S-S'))$, and use dynamic-programming to compute V for all sets S based on values that were already computed for smaller sets. However, the number of subsets S' for each set S is too large. The heuristic I implemented was to sample 25 subsets S' for each sets S, and take the maximum over these sub-sets.

4.1.6 Heuristic strategy

This strategy again uses the optimal value function for the last four stages. For other sets $S=\{g_1,\ldots,g_k\}$ of k goals, it uses the approximation $V(S)=\frac{1}{k-1}\sum_{i=1}^{k} V(S-\{g_i\})$. The rational behind this expression is that for large sets S, we expect the synergy within S and

S-$\{g_i\}$ to be similar. Since we sum over k sets containing k-1 elements, each element is "covered" k-1 times, so we divide by k-1.

4.1.7 Greedy strategy

Every game must have a strategy for beginners. The greedy strategy optimizes the reward from every turn without looking into the future. The value function is constant: $V(S)=0$.

4.2 Simulations

The strategies above were tested in simulated games. The results are summarized in the table below. Note that the scores vary a few points in different runs of the program. The columns of the table are:

- *Mean*: the mean score of playing the strategy 1000 games.
- *Std*: the standard deviation of the scores.
- *Estimation*: the value function predicted by the strategy before the beginning of the game.
- *Estimation Deviation*: absolute value of (*Mean*-*Estimation*).
- *Mean deviation from optimal*: the mean of the absolute differences between the value function of the strategy and the optimal value function. This mean is over all entries in the value function tables.

Strategy	Mean	Std	Estimation	Estimation deviation	Mean deviation from optimal
Max expectation	232.556	32.206	234.601	2.045	0
Additive	229.628	29.508	131.733	97.894	37.517
Regression	232.203	32.507	226.394	5.809	1.462
Small regression	232.855	31.87	203.257	29.598	6.325
Max DP	232.856	31.635	187.743	45.113	11.63
Heuristic	233.125	31.811	184.232	45.113	11.411
Greedy	215.315	32.297	0	215.315	103.384

The theoretical maximal expected score is 234.6. Note that except the greedy strategy, all strategies achieved approximately the same mean score. The differences in the means are very small compared to the standard deviation of the scores. This is explained by all strategies using the same DAG component. They make the same move in almost every state.

The heuristics differ in estimating the value function. The full regression gives a very accurate estimation, because it is based on the optimal value function. The regression over the last four stages gives reasonable estimation, suggesting the linear model of the value function is close to the truth but there is also a non-linear component. Other

strategies have worse estimation of the value function. Surprisingly, the additive strategy gets in practice 100 points more than its a-priori estimation.

I compared my results to the original "ML-Yaht", which was implemented in 1987 by Bob Lancaster (http://www.textmodegames.com/download/mlyaht.html). I ran "ML-Yaht" for 100 games. The average score was 224. After deducting all bonus points, the average score dropped to 192, which is even worse than my greedy strategy (intended for beginners…).

4.3 Multi-player strategy

As a multi-player game, Yaht is analogous to a race in separate lanes. Each player can see what the other players are doing, but cannot block them.

For a two-player game, the number of states for which the optimal value function has to be specified is still tractable. We have to choose subsets of remaining sub-goals for both players with the same number of remaining sub-goals. There are only $\sum_{i=0}^{13}\binom{13}{i}^2=\binom{2\cdot 13}{13}=10{,}400{,}600$ possibilities. The state also includes the score difference between the players, which is practically bounded by 300 states. In principle, it is possible to compute the optimal value function for the two-player game with a modern computer (although it will probably take more than a day).

I decided to implement a simple variance-based policy. The first stage was to compute the variance for every set of remaining sub-goals assuming the strategy maximizes the expected score. These variances are stored in a variance table, similar to the value function table. For every chance state in the DAG, the variance is computed using the formula: Var(score)=E[Var(score | y)]+Var(E[score | y]). Here y is the random variable corresponding to the new state after rolling the dice (The variances and expectations are computed according to the probabilities of getting to the new state y). The computed a-priori standard deviation of the whole game is 31.953, which is comparable to the values achieved in the simulations. The magnitude of the variance in this game is non-negligible, suggesting that luck is more important in this game relatively to games like backgammon.

I made the following simplifying assumption: every (other) player will play according to the optimal strategy, and the distribution of scores is Gaussian with the known means and variances. Under this assumption, we can calculate the probability to win from every state where players have finished their turns. Suppose the score for player i is distributed $N(\mu_i,\sigma_i^2)$. Then $\Pr(\text{player i wins})=\int \Pr(\text{player i score}=t)\prod_{j\neq i}\Pr(\text{player j score}\le t)dt$. The first term in this integral is the normal density of player i, while the second product is the normal CDF of the other players. In the implementation, this integral is approximated by a sum where t is ranging two standard deviations from μ_i.

At run-time, the leaves of the DAG are loaded with the estimated probability to win from every leaf-state of the DAG instead of loading the expectations of the score. The dynamic programming procedure remains the same. The only difference is that here the player is trying to maximize his expected probability to win after the turn.

I tested this strategy against the other strategies in 1000 duals, and measured the percentage the other strategies won. Note that changes of 5% between runs are possible. The results are summarized in the following table:

Strategy	Wins percentage
Max expectation	0.478
Additive	0.45
Regression	0.448
Small regression	0.472
Max DP	0.483
Heuristic	0.474
Greedy	0.305

It seems that the variance strategy improves only slightly over the strategy that maximizes expectation and doesn't look what the opponent is doing. There could be two explanations for that. Either the estimation of the winning probability is erroneous, or the nature of the game is such that when one player leads the game, the other player cannot increase his probability to win by making "suicidal" moves. I guess the best strategy is almost always independent of the other players.

5 Conclusions

The emphasis of this project was on the analysis and implementation of a real-world game, rather on the theory of decision making. Since the game turned out to be tractable, I didn’t need sophisticated approximation techniques for the optimal strategy. It turned out that most strategies can perform near-to-optimal even with poor estimation of the value function. That means that the procedure for a single turn is at the hart of the game. As long as this procedure handles the probabilities correctly, the estimation of future reward is less important. However, there is still a large gap between the heuristic approximations of the value function to the optimal value function. Since the optimal value function is known exactly, this game could serve as a test case for more clever methods that approximate the value function.

6 Program description

This section is the user guide for experimenting with the program.

- *Installation*: Put all the files in the same directory, and run YahtProject.exe. The other files are the value function tables for each strategy.
- *The dice*: You can select dice for rolling by mouse left-click. Second click un-selects. With a mouse right-click you can change the face of a die. This feature was added to experiment with the program's behavior.
- *Roll button*: After selecting dice, the roll button will roll them. When no dice are selected, no dice will be rolled but the stage number will be incremented.
- *Left checkboxes*: After the third roll, you assign a sub-goal by clicking on one of the checkboxes on the left. Sub-goals that were assigned in previous turns are disabled. If you want to change your mind, click on another checkbox (you cannot un-check).
- *Strategy comboBox*: Select the strategy for the computer. The "complete" strategy will play the variance-based policy in case there is more than one player (otherwise it is identical to the max-expectation strategy).
- *Advice button*: This button performs a decision-move by the computer. After giving the advice, the strategy's estimation of the expected score of the game is printed in parenthesis. In case of the "complete" strategy and multi-player game, the estimated probability to win is printed. Note that this probability assumes the other players are at the beginning of their turns (it doesn't look at their dice, only on the set of remaining goals). In order to simulate a game, press "Advice"-"Roll" alternately.
- *Strategy statistics button*: This button will run 1000 duals against the "complete" strategy and prints the statistics of the scores. It takes around 30 minutes.