Regular Paper

Ten Times Eighteen

Sebastian Böcker[1,a)]

Received: March 24, 2014, Accepted: January 7, 2015

Abstract: We consider the following simple game: We are given a table with ten slots indexed one to ten. In each of the ten rounds of the game, three dice are rolled and the numbers are added. We then put this number into any free slot. For each slot, we multiply the slot index with the number in this slot, and add up the products. The goal of the game is to maximize this score. In more detail, we play the game many times, and try to maximize the sum of scores or, equivalently, the expected score. We present a strategy to optimally play this game with respect to the expected score. We then modify our strategy so that we need only polynomial time and space. Finally, we show that knowing all ten rolls in advance, results in a relatively small increase in score. Although the game has a random component and requires a non-trivial strategy to be solved optimally, this strategy needs only polynomial time and space.

Keywords: dice rolling game, expected score maximization, dynamic programming, computational complexity

1. Introduction

When I was in twelfth grade at school, my computer science teacher introduced us to the following game: Assume that you are given a table with ten slots indexed one to ten. The game proceeds in ten rounds. In each round, three dice are rolled and the numbers are added. Then, you are allowed to put this number into any free slot. In the end, your table is completely filled with numbers between three and 18. For each slot, you multiply the slot index with the number in this slot, and then you add up the products. An example is given in **Fig. 1**. The goal of the Ten times 18 game is to maximize the sum of products. The smallest total score that you can reach is

$$(1 + 2 + \cdots + 9 + 10) \cdot 3 = 165,$$

the largest score is 990.

If you play Ten times 18, you will quickly come up with first ideas whether certain moves that are good or bad: For example, you should definitely put a "three" into slot number one, and you should put an eighteen into slot number ten. If these slots are not available, put them in the slot with the smallest or highest index available, respectively. But what about a roll of "five"? And what do you do if you roll a "seven" and all even slots have been taken? Is this basically the same problem as rolling a seven when all odd slots have been taken? (In fact, it is.)

The question this boils down to, is: How do we maximize the sum of products? That is, we are searching for a *strategy* that maximizes our chances of winning, that is, the points we can obtain. Clearly, playing only a single game is not sufficient to judge a strategy, so we repeat the game many times and for all these games, we again sum up the sum of products. Formally speaking, this boils down to: What is a strategy that maximizes the *expected* sum of products?

Back in 1987, my schoolmates and I came up with many different strategies for Ten times 18: These were based on statistical considerations, and even some simple machine learning strategies (play the game repeatedly and see what moves are favorable). Funnily, it is rather straightforward to find an optimal strategy if you are familiar with the concept of dynamic programming — and a tiny twist. In fact, you do not need a fancy computer to find this strategy. And with a little more statistics, we can even find a strategy that optimally plays practically any variant of Ten times 18: That is, the strategy requires only polynomial time and space. Finally, we show how to compute the advantage of an "all-knowing" strategy, which is allowed to look into the future before placing the rolls: Interestingly, this advantage is relatively small.

Playing Ten times 18 is different from many other solitaire games in that rolling dice is involved. Combinatorial games without chance (such as Rubik's cube) have been studied more frequently [1], in particular the complexity of playing an optimal strategy. The probably "closest relative" to Ten times 18 is Yahtzee, a popular dice game. Optimal solitaire strategies — again in the sense of maximizing the expected score — were independently developed by Tom Verhoeff [12] and James Glenn [6] around 1999, but never formally published. To this end, further authors developed optimal strategies for the solitaire game [11], [13]. Obviously, Yahtzee is much more involved than Ten times 18, and so is the analysis of the game. Other related stochastic problems include the stochastic knapsack problem [7], [8], [9], [10], the secretary problem [3], [4], and last-success-problems [2].

2. Preliminaries

Let $B := \{1, \ldots, 10\}$ be the slots, and let $A \subseteq B$ be the slots that have already been filled. The first important thing to notice,

1 Friedrich Schiller University, Jena, Germany
a) sebastian.boecker@uni-jena.de

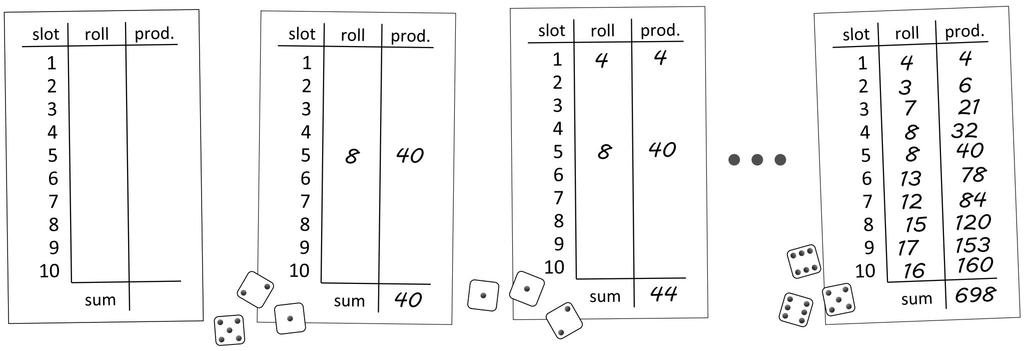

Fig. 1 Example of a Ten times 18 game. The first roll is '8', and the player chooses to place it in slot #5, resulting in product score 40. The second roll is '4', and the player chooses to place it in slot #1, resulting in product score 4 and sum of products 44. At the end of the game, the player has reached total score 698, an excellent score as we will see below, compare to Table 5. Note that due to incomplete information, the player has made several suboptimal choices.

Table 1 Probabilities for throwing three dice.

x	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$216 \cdot \mathbb{P}(X = x)$	1	3	6	10	15	21	25	27	27	25	21	15	10	6	3	1

is that for finding the best move at this point, it does not matter what numbers have actually been inserted into the slots that have been filled: You can simply think of it as a new game where an incomplete table has been given to you, and your task is to maximize the sum of products for the incomplete table. Doing so, we also maximize the sum of products for the complete table. The score of the complete table obviously depends on the previously filled slots; but we cannot change a previous decision.

We now formalize the problem a little bit: We model rolling the three dice as a *random variable* X with $X \in \{3, \dots, 18\}$. We denote the probability that some happens by $\mathbb{P}(\text{event})$. The probability that we roll a three is 1 in 216, which is formally written as $\mathbb{P}(X = 3) = \frac{1}{216}$. Similarly, we are given the probabilities $\mathbb{P}(X = 4) = \frac{3}{216}$ and so on, see **Table 1**. We assume that X is always an integer, and that there exist integer bounds $x_{\min} < x_{\max}$ such that $x_{\min} \le X \le x_{\max}$. Using this formal variable, allows us to re-use our thoughts below for other variants of the Ten times 18 game: For example, the dices might be loaded; we might want to throw two or four dices instead of three; or, we might even throw five twelve-sided dice. For all of these variants, the solution introduced below works, though you have to repeat the calculations.

For a given random variable Y we denote its expected value as $\mathbb{E}(Y)$. When the probabilities of all possible outcomes are known to us, we can compute the expected value by summing over the products of the probability times the outcome. For the three dice example with random variable X we can calculate

$$\mathbb{E}(X) = \frac{1}{216} \cdot 3 + \frac{3}{216} \cdot 4 + \cdots + \frac{1}{216} \cdot 18 = 10.5.$$

Clearly, there is a simpler way to calculate this: For two random variables X, Y we have $\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$. In other words, the expected value of the sum of three identical dices equals three times the expected value of a single dice. If we assume that X' is the random variable of a single dice, then $\mathbb{E}(X) = 3\,\mathbb{E}(X') = 3 \cdot 3.5 = 10.5$.

The simplest strategy that we can evaluate using the above considerations, is the "random strategy" where we assign each roll randomly to a slot. This strategy has the expected score

$$(1 + 2 + \cdots + 9 + 10) \cdot 10.5 = 577.5.$$

This score is what we have to compare our strategy against in the future.

3. Why is This Complicated?

Often, people who get to know Ten times 18 immediately start thinking about one or the other strategy to solve it. One particular, general approach easily comes to mind: Why not model the complete game as one large decision tree where nodes correspond to states of what has happened so far, and edges correspond to changing from one state to another? That is, we start with an initial state where all slots are empty. Then, we add 160 outgoing edges, one for each roll from 3 to 18 and one for each slot that we can fill with it. In the end, we will only have to store the optimal slot to be filled with each number; but as we are only in the process of determining this optimal slot, storing the complete tree appears to be inevitable [*1].

One can easily check that this approach suffers from the size of the tree that we have to compute and store: As noted above, there are $16 \cdot 10 = 160$ outgoing edges from the root node, resulting in the same number of nodes in the tree. Leaving every such node, there are $16 \cdot 9 = 144$ outgoing edges and a total of $160 \cdot 144 = 23{,}040$ nodes at the next level. In total, we reach

$$16^{10} \cdot 10! = 16 \cdot \cdots \cdot 16 \cdot 10 \cdot 9 \cdot \cdots \cdot 2 \cdot 1 = 3.99 \cdot 10^{18}$$

nodes at the last level of the tree. So, storing some value for each node of the tree is impossible on today's computers, and even beyond the capacity of any hard disk, as it requires several exabytes of memory. Hence, this road is blocked, in particular if you want to play Ten times 18 with more than ten slots, see below.

From a computational complexity viewpoint, the arguably

[*1] Similar trees are used for many games, in particular two-player games with complete knowledge and without chance, such as chess.

most interesting question is: can we decide with polynomial time and space upon the next optimal move, or is the problem NP-hard [5]? For a polynomial algorithm, we require that time and space are bounded by a polynomial in all aspects of the input: the number of highest roll and, in particular, the number of slots in the input. We will come back to this question in Section 6.

4. Dynamic Programming

Dynamic programming solves complex problems by breaking them down into simpler subproblems. To solve a problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. We make sure that each subproblem is solved only once, thus reducing the number of computations. Top-down dynamic programming simply means storing the results for all subproblems that we encounter. In bottom-up dynamic programming, we try to solve smaller subproblems first, and deduce the solution of larger subproblems by combining those of smaller subproblems. We will concentrate on bottom-up dynamic programming, so that our solution does not require any recursive calls.

Assume that slots $A \subseteq B := \{1, \ldots, 10\}$ have been filled before. We want to know what score we can reach for the rest of the game, if we play an "optimal strategy." This optimal score depends on the rolls that will happen in the future, so we cannot know the exact score that we will reach. But what we can study is the expected value of the score we can reach in the future; it is this score that we want to maximize. To this end, let $M[A]$ denote the maximum expected value of the score that we can reach using any strategy, where slots A have been previously filled. Then, $M[B]$ is the maximum expected score that we can reach for the complete game. Clearly, we are rather interested in the strategy that leads to this maximum expected score, and not so much in the score itself. But with dynamic programming, the structure of the solution can usually be inferred easily when the dynamic programming table has been filled. To this end, we will concentrate on computing the expected scores.

There exist $2^{|B|}$ subsets of the set B, including the empty set and the full set. This comes down to $2^{10} = 1{,}024$ subsets for $B = \{1, \ldots, 10\}$. For each subset $A \subseteq B$ we store the entry $M[A]$. In implementation, the subsets A will be represented as bit vectors, and every subset A can be easily transformed into a number between 0 and $2^{|B|} - 1$.

We have noted above that one trick of dynamic programming is to compute the solutions for each subproblem only once, and to store it so it can be accessed multiple times. Here, this means that we want to compute the entries of table M in the right order, and to use previously computed entries of M for deriving the next one. In particular, we want to make sure that any entry of the matrix M is accessed only after it has been computed. To this end, we first need an initialization to start from: If none of the slots has been filled so far, then the best expected score is obviously zero for doing nothing, so $M[\emptyset] = 0$. It is a well-known trick to initialize the dynamic programming table for an entry where, in fact, nothing has happened so far. If you do not like the empty set initialization, you can instead initialize

Table 2 The matrix M for all subsets $A \subseteq B = \{1, \ldots, 10\}$ of cardinality 9, rounded to five decimal places. This table is required to decide upon the first move of Ten times 18.

i	1	2	3	4	5
$M[B - \{i\}]$	618.32001	611.45000	603.39355	594.42809	584.66842
i	6	7	8	9	10
$M[B - \{i\}]$	574.16842	562.92809	550.89355	537.95000	523.82001

$$M[\{i\}] = i \cdot \mathbb{E}(X) \quad \text{for } i = 1, \ldots, 10$$

because moving any number to the last remaining slot i, the expected score for doing so is simply $i \cdot \mathbb{E}(X)$. This initialization is slightly more complicated but leads to exactly the same results.

To make sure that we only access entries of the table that have been previously computed, we iterate $k = 1, \ldots, |B|$, and in each step of the iteration we compute all entries $M[A]$ for all subsets $A \subseteq B$ with $|A| = k$. (If you have initialized the one-element subsets you can leave out $k = 1$ in the iteration.) To this end, assume that the table M has been filled for all $A \subseteq B$ where $|A| \leq k - 1$. We now show how to compute it for each entry $A \subseteq B$ with $|A| = k$. This means that we are allowed to distribute k rolls into the filled slots A. We concentrate on the next roll: The probability that some x with $x_{\min} \leq x \leq x_{\max}$ is rolled next, is $\mathbb{P}(X = x)$. Possible rolls are lower bounded by $x_{\min}$ and upper bounded by $x_{\max}$. If we decide to put roll x into slot i for $i \in A$ then we gain $i \cdot x$ in the sum of products. Playing the remaining slots, the best strategy will (by definition of M) reach expected score $M[A - \{i\}]$. Putting this together we get

$$M[A] = \sum_{x = x_{\min}, \ldots, x_{\max}} \mathbb{P}(X = x) \cdot \max_{i \in A} \Big\{ i \cdot x + M[A - \{i\}] \Big\}. \qquad (1)$$

How long does it take to fill the matrix M? There exist $2^{|B|}$ many entries in the matrix. For each entry we iterate over $x_{\max} - x_{\min} + 1$ many values for x, and $|A| \leq |B|$ different values for i, a total of $O((x_{\max} - x_{\min} + 1) \cdot |B|)$ entries [*2]. In total, filling the complete matrix requires $O(2^{|B|} \cdot (x_{\max} - x_{\min} + 1) \cdot |B|)$ time: That is, we need less than $c \cdot 2^{|B|} \cdot (x_{\max} - x_{\min} + 1) \cdot |B|$ summations, multiplications, and comparisons for some multiplicative constant c.

Now, the maximum expected score that any strategy can reach, can be computed as

$$M[\{1, \ldots, 10\}] = 642.2393504256.$$

This score can be computed using those entries $M[A]$ where A has cardinality 9, see **Table 2**. Due to space constraints, we cannot show all 1,024 entries of the table. Note that the expected score drops with higher i: This is as we would expect it, because for large i we have already used up more of the high-scoring slots.

5. Playing the Game

How does knowledge about the maximum expected score, $M[A]$, help us to come up with a useful move? This, in fact, is quite simple: Assume that slots $A \subseteq B$ have previously been filled, and that number x has been rolled in this move. From the above, it is straightforward to show that the maximum expected

[*2] The "big O" notation is used to describe the asymptotic behavior of some function, ignoring constant factors.

Table 3 The best strategy for the first move of Ten times 18. For each roll, slot index i^* has been chosen using Eq. (3).

roll	slot	$\mathbb{E}$(score)	roll	slot	$\mathbb{E}$(score)	roll	slot	$\mathbb{E}$(score)	roll	slot	$\mathbb{E}$(score)
3	#1	621.32001	7	#2	625.45000	11	#6	640.16842	15	#10	673.82001
4	#1	622.32001	8	#2	627.45000	12	#7	646.92809	16	#10	683.82001
5	#1	623.32001	9	#4	630.42809	13	#9	654.95000	17	#10	693.82001
6	#1	624.32001	10	#5	634.66842	14	#9	663.95000	18	#10	703.82001

score that we can reach after we have placed x is

$$\max_{i \in B-A} \Big\{ i \cdot x + M[A \cup \{i\}] \Big\}. \quad (2)$$

This follows because $M[A \cup \{i\}]$ is the maximum expected value that we can reach when slot $A \cup \{i\}$ have been filled previously. So, all we have to do is search for i^* such that

$$i^* \cdot x + M[A \cup \{i^*\}] = \max_{i \in B-A} \Big\{ i \cdot x + M[A \cup \{i\}] \Big\} \quad (3)$$

and then, place x in slot i^*. This can be achieved quickly: We need only $O(|B|)$ steps to find the maximum.

We have depicted the "maximum expected score" strategy for the first move of the game in **Table 3**, including the expected score that we can reach including this first move. There are at least two unexpected things to notice in this table: Firstly, even a roll of 6 should still be placed in the first slot, and similarly, even a roll of 15 should still be placed in the highest slot. This becomes understandable, though, if we consider that rolling a 3 to 6 has total probability of less than 10%; and the same holds for rolling a 15 to 18. Second, it never pays off to put the first roll into slots #3 or #8. It is doubtful that there is a simple explanation for this fact; it simply comes out of our calculations.

We can also ask for the "closest call" of the "maximum expected score" strategy: Which roll minimizes the difference between the expected scores resulting from the best and second best moves, and what are those moves? For the first move, this is a roll of 9: If we place it into slot #3 (instead of the optimal slot #4) we can still reach an expected score of 630.39355, the difference being only 0.03455. Similarly, we can place a roll of 12 into slot #8 instead of slot #7, with the same difference in score. For the complete game, the closest call is placing a roll of 10 when seven consecutive slots are available: Here, the runner-up placement of the roll decreases the expected score by 0.02989.

6. Polynomial Time and Space

The above "maximum expected score" strategy requires us to compute and store an array with $2^{|B|}$ entries. This is not a problem for $|B| = 10$, as the total size of the table is only 1,024. Even in the 1990s, practically every home computer came with a sufficient amount of memory to store such a table [*3]. But the important point is that memory requirement increases *exponentially* with the size of the set B. Whereas one could think of the analogous games with $|B| = 20, 30, 40$ as being twice (three times or four times, respectively) as hard as the original game, we need megabytes, gigabytes, or even terabytes to store the table M. This implies that for $|B| = 40$ tables are already much to large to be stored in the main memory of the average present-day computers. Given that the current rate of miniaturization integrated circuits is kept throughout the next years, it would still require more than a year so that we can increase the size of solvable instances by *one*. Even if every atom in the observable universe (approximately 10^{80}) would be used to store one entry of our table M, this would not allow us to play a game where $|B| > 265$. Therefore, it is an interesting question whether we can get away with less memory.

We can answer this question easily for one particular type of Ten times 18: That is, if we have only two possible outcomes for each throw (flipping a coin), such as 1 and 2. In this case, the problem becomes trivial: Just place any 1 into the first available slot (with smallest index), and place any 2 into the last available slot (with highest index). It is clear that this strategy reaches the optimum expected score, uses constant memory for storing two integers, and performs each move in constant time: Here and in the following, we assume that integer and real-valued numbers can be stored in constant space, and that elementary operations on these numbers can be executed in constant time.

But somewhat unexpectedly, we can still find a solution for our original game (and, in fact, any variant of Ten times 18 where slot multipliers are strictly increasing). For this, we need a little more statistics to show that we can actually solve the problem with polynomial memory and time. Assume that there are k slots left, and that

$$\lambda_1 \le \lambda_2 \le \cdots \le \lambda_k$$

are the score multipliers. For any deterministic or random strategy, let $Y_1, \ldots, Y_k$ be random variables such that Y_i is the roll the strategy places on slot i. Now, $Y := \lambda_1 Y_1 + \cdots + \lambda_k Y_k$ is the random variable for the score of this strategy, and we have

$$\mathbb{E}(Y) = \mathbb{E}(\lambda_1 Y_1 + \cdots + \lambda_k Y_k) = \lambda_1 \mathbb{E}(Y_1) + \cdots + \lambda_k \mathbb{E}(Y_k). \quad (4)$$

Note that the random variables Y_i are strongly correlated, as placing a roll of 18 into the highest slot will influence the expected values for all other slots; but Eq. (4) also holds for correlated random variables. Assume that there exist $i < j$ such that $\mathbb{E}(Y_i) > \mathbb{E}(Y_j)$. Then, the strategy cannot be optimal: simply exchange all moves of the strategy to slots i and j, which results in a strategy with expected score

$$\mathbb{E}(Y) + (\mathbb{E}(Y_i) - \mathbb{E}(Y_j)) \cdot (\lambda_j - \lambda_i) \ge \mathbb{E}(Y)$$

as $\lambda_j \ge \lambda_i$ and, by our assumption, $\mathbb{E}(Y_i) \ge \mathbb{E}(Y_j)$. This implies that for an optimal strategy, we have

$$\mathbb{E}(Y_1) \le \mathbb{E}(Y_2) \le \cdots \le \mathbb{E}(Y_k). \quad (5)$$

Assume that $k+1$ slots are empty, and that our roll is some $x \in \{x_{\min}, \ldots, x_{\max}\}$ — where will the best strategy to maximize the

[*3] The only notable exception that I am aware of was the Sinclair ZX81 where the basic model shipped with only 1 kilobyte of Random Access Memory.

Table 4 The expected values $E_j[i]$ necessary to decide upon any optimal move in TEN TIMES 18. The first row ($j = 10$) is not needed to play the game but only to compute the expected score of the strategy.

$E_j[i]$	$i = 1$	2	3	4	5	6	7	8	9	10
$j = 10$	6.720	7.868	8.730	9.466	10.160	10.840	11.534	12.270	13.132	14.280
9	6.870	8.056	8.965	9.760	10.500	11.240	12.035	12.944	14.130	
8	7.038	8.287	9.241	10.089	10.911	11.759	12.713	13.962		
7	7.239	8.553	9.570	10.500	11.430	12.447	13.761			
6	7.479	8.861	9.970	11.030	12.139	13.521				
5	7.765	9.254	10.500	11.746	13.235					
4	8.120	9.771	11.229	12.880						
3	8.599	10.500	12.401							
2	9.292	11.708								
1	10.500									

expected score place this roll? From Eq. (5) it is straightforward to understand that this roll must be placed on the i-th free slot such that $\mathbb{E}(Y_{i-1}) \leq x \leq \mathbb{E}(Y_i)$: We can easily show that placing x into any other free slot, will result in a suboptimal expected score. This means that the λ_i are not taken into consideration for deciding upon the best move.

To this end, let us consider the "maximum expected score" strategy for j empty slots with strictly increasing slot weights: We define $E_j[i] = \mathbb{E}(Y_i)$ as the expected value of the i-th slot. It is easy to understand how $E_{j+1}[\cdot]$ can be computed from $E_j[\cdot]$: For $E_j[\cdot]$ and $x \in \{x_{\min}, \dots, x_{\max}\}$ we define $I_j(x)$ as the index such that

$$E_j[i-1] \leq x \leq E_j[i] \quad \text{for } i = I_j(x).$$

We may assume that $E_j[0] = -\infty$ and $E_j[j+1] = +\infty$. In case of a draw we can choose any such index. We infer the recurrence:

$$E_j[i] = \sum_{x=x_{\min},\dots,x_{\max}} \mathbb{P}(X=x) \cdot \begin{cases} E_{j-1}[i] & \text{for } i = 1,\dots,I_j(x)-1 \\ x & \text{for } i = I_j(x) \\ E_{j-1}[i-1] & \text{for } i = I_j(x)+1,\dots,j \end{cases} \tag{6}$$

Computing the table requires $O(|B|^2 \cdot (x_{\max} - x_{\min} + 1))$ time. In the end, the expected score of the "maximum expected score" strategy can be calculated as $\sum_{i=1}^{k} i \cdot E_k[i]$ which again results in the same score of 642.2393504256 as above.

We have depicted the complete table $E_j[i]$ for $j = 1, \dots, 10$ and $i = 1, \dots, j$ in **Table 4**. This table allows us to play the complete game using the "maximum expected score" strategy: Assume that there are $j + 1$ free slots and we have to place a roll of x. Find $i \in \{1, \dots, j+1\}$ such that $E_j[i-1] \leq x \leq E_j[i]$. (Recall that we assume $E_j[0] = -\infty$ and $E_j[j+1] = +\infty$.) Place x into the i-th *free* slot, sorted from smallest to largest multiplier.

As an example, assume that half of the slots have been filled, so $j + 1 = 5$. Row E_4 from Table 4 tells us that rolls 3 to 8 will be placed into the first free slot with smallest multiplier; roll 9 is placed into the second free slot; rolls 10 and 11 are placed into the third free slot; roll 12 is placed into the fourth free slot; and, finally, rolls 13 to 18 are placed into the last free slot with the highest multiplier.

7. Knowing the Future

The maximum expected score that we can reach, is significantly higher than the score of the random strategy, but not to an extent that one might initially think. In particular, the maximum expected score of 642.2 is much smaller than the highest score of 990. But the highest score can only be reached if we have ten rolls of 18, and the chances that this is going to happen are

$$1/221073919720733357899776 = 4.52 \cdot 10^{-24}.$$

For all other TEN TIMES 18 instances, the highest score is naturally unreachable. But with the same probability, we have ten rolls of 3, and any strategy will result in the minimum score of 165.

A better way of evaluating the performance of our strategy, is to compare it against an "all-knowing" strategy which is allowed to look into the future: To this end, assume that our strategy knows the outcome of all ten rolls before having to place the first roll. This "all-knowing" strategy will simply sort all rolls and then place them accordingly. This is related to the order statistics of the ten rolls (the sample) for all ranks.

Again, we cannot judge the performance of this strategy by evaluating a single game. Instead, we play many games and sum up the scores; this again boils down to the expected score of the strategy. This can be computed using "classical" dynamic programming; we do not have to take into account the set of slots that have been filled so far. Let $L := |B|$. We define $Q[y, l]$ as the partial score obtained by the "all-knowing" strategy for placing $L - l$ rolls $x \geq y$, whereas for the remaining l rolls we know $x < y$ but these have not been scored so far (that is, l free slots). Then, $Q[x_{\max}, L]$ is the expected score of the "all-knowing" strategy. We infer the recurrence

$$Q[y,l] = \sum_{k=0,\dots,l} \binom{l}{k} p_y^k (1-p_y)^{l-k} \cdot \Big(y \cdot S(l-k+1, l) + Q[y-1, l-k]\Big) \tag{7}$$

where

$$p_y := \mathbb{P}(X = y | X \leq y) = \frac{\mathbb{P}(X = y)}{\sum_{x \leq y} \mathbb{P}(X = x)}$$

and

$$S(i,j) := \sum_{k=i,\dots,j} k = \tfrac{1}{2}((j+1)j - i(i-1)).$$

We initialize $Q[y, 0] = 0$ for all $y = x_{\min}, \dots, x_{\max}$, and $Q[x_{\min} - 1, l] = 0$ for all $l = 0, \dots, L$.

We reach an expected score of 652.93403 for the "all-knowing" strategy. Somewhat surprisingly, this expected score is not much

Table 5 Different important scores for the Ten times 18 game. *Median scores were experimentally determined from one million runs, see Section 8.

Minimum possible score	165.0
Median score of the "random" strategy*	577
Expected score of the "random" strategy	577.5
Expected score of the "maximum expected score" strategy	642.23935
Median score of the "maximum expected score" strategy*	646
Expected score of the "all-knowing" strategy	652.93403
Median score of the "all-knowing" strategy*	654
Maximum possible score	990.0

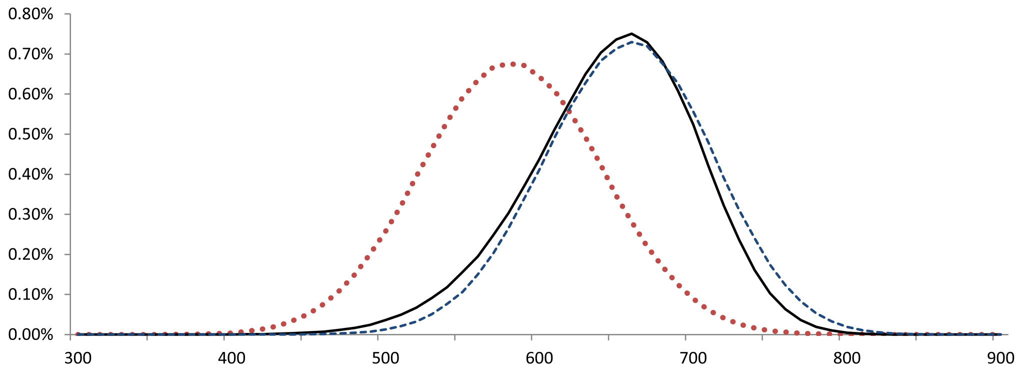

Fig. 2 Empirical score distribution for the random strategy (dotted line), the "maximum expected score" strategy (solid line), and the "all-knowing" strategy (dashed line). Calculated from one million runs, binned using bin width 10.

Table 6 Probabilities for throwing two loaded, twelve-sided dice.

x	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$169 \cdot \mathbb{P}(X = x)$	1	2	3	4	5	6	7	8	9	10	11	14	13	12	11	10	9	8	7	6	5	4	4

higher than the 642.23935 that our strategy can reach without knowing the future: Knowing the future only gives us an expected upper hand of about ten points. All scores are summarized in **Table 5**.

8. Implementations and Simulations

All algorithms presented in this paper were implemented in Groovy 1.8.6 and run on a laptop computer. All computations were carried out with high precision (40+ digits). In addition, we implemented both variants of the "maximum expected score" strategy and the "all-knowing" strategy and performed simulations. After one million runs, the average score of the "maximum expected score" strategy was 642.272639 (for both variants), and the average score of the "all-knowing" strategy was 652.947393. This agrees well with the theoretical values computed above. Running times of our computations were negligible. We have also computed median scores from these evaluations, see again Table 5.

The empirical distributions of scores are depicted in **Fig. 2**. We have smoothed the curves by binning ten values in each bin, $\{10n, \ldots, 10n + 9\}$ for $n = 15, \ldots, 99$.

9. Variants of Ten times 18

We have noted above that our computations are not limited to the Ten times 18 variant where three six-sided dice are thrown. To exemplify this claim, let us consider one more variant, namely throwing two "slightly loaded" twelve-sided dice: For each die, the probability for a roll of "12" is $\frac{2}{13}$, and the probability of all other rolls is $\frac{1}{13}$. The probabilities for throwing two loaded, twelve-sided dice are depicted in **Table 6**. The expected value of a single roll is 13.84615.

Table 7 The expected values $E_j[i]$ necessary to decide upon any optimal move in the game with two loaded twelve-sided dice. The first row ($j = 5$) is not needed to play the game but only to compute the expected score of the strategy.

$E_j[i]$	$i = 1$	2	3	4	5
$j = 5$	9.038	11.713	13.868	16.012	18.599
4	9.680	12.613	15.113	17.978	
3	10.532	13.861	17.146		
2	11.753	15.939			
1	13.846				

Again, we can calculate the table of expected scores for all positions, see **Table 7**. Assuming slot multipliers 1 to 5, we reach a score of 231.11229 for the "maximum expected score" strategy, and 236.97840 for the "all-knowing" strategy. In comparison, the random strategy reaches an expected score of 207.69231.

10. Conclusion

We have presented the game Ten times 18, plus a strategy to maximize the expected score. In addition, we have shown how to compute the expected score of an omniscient strategy.

Playing the strategy maximizing the expected score, does not maximize your chances to win a two-player game: That is, two players are given the same rolls and compete against each other to maximize the score reached in a single game. The player that wins the most games wins the match. Again, we assume that a sufficiently large number of games is played. Here, the "maximum expected score" strategy introduced in this paper will be hard to beat. But if you know your opponent is playing this strategy, then you can still get an upper hand against the score-optimal strategy: It is enough to be a few points ahead in most games, whereas the score of any lost game is unimportant. This takes us into the realms of game theory; in particular, there is no longer one optimal strategy but instead, there may be cases where strategy A beats strategy B, B beats C, but C beats A. Things will become even more complicated in multi-player games where several strategies compete simultaneously. But at least, it should be possible to come up with a strategy that beats the score-optimal strategy in a two-player game: we only have to consider the chances of some score being higher than that of the score-optimal strategy in every move. The state space will increase considerably, because now we have to consider the filled slots and the score obtained so far.

One possible way to approach beating the "maximum expected score" strategy is as follows: If we want to achieve (at least) a certain score x, what is the optimal strategy to maximize this probability? We can find an optimal strategy using dynamic programming in exponential time. But can we achieve this in polynomial time, possibly using some relationship with the "maximum expected score" strategy presented here? Another open question is: If each slot is scored by a (nonlinear) function and the total score is the sum of scores for the corresponding slots, what is the computational complexity of the problem?

Acknowledgments I want to thank my school teacher, Klaus Bovermann, for introducing this game to us, and getting us into developing strategies for it.

References

[1] Berlekamp, E.R., Conway, J.H. and Guy, R.K.: *Winning Ways for Your Mathematical Plays*, Vol.1–4, Taylor & Francis, second edition (2001–2004).

[2] Bruss, F.T.: Sum the odds to one and stop, *The Annals of Probability*, Vol.28, No.3, pp.1384–1391 (2000).

[3] Ferguson, T.S.: Who solved the secretary problem?, *Statistical science*, Vol.3, No.3, pp.282–289 (1989).

[4] Freeman, P.: The secretary problem and its extensions: A review, *International Statistical Review/Revue Internationale de Statistique*, Vol.51, No.2, pp.189–206 (1983).

[5] Garey, M.R. and Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co. (1979).

[6] Glenn, J.: An optimal strategy for Yahtzee, Technical Report CS-TR-0002, Loyola College, Department of Computer Science, Maryland (2006).

[7] Kleywegt, A.J. and Papastavrou, J.D.: The dynamic and stochastic knapsack problem, *Operations Research*, Vol.46, No.1, pp.17–35 (1998).

[8] Kleywegt, A.J. and Papastavrou, J.D.: The dynamic and stochastic knapsack problem with random sized items, *Operations Research*, Vol.49, No.1, pp.26–41 (2001).

[9] Ross, K.W.: The stochastic knapsack, *Multiservice Loss Models for Broadband Telecommunication Networks*, pp.17–70, Springer (1995).

[10] Ross, S.M.: *Introduction to stochastic dynamic programming*, Academic press New York (1983).

[11] Vancura, O.: *Advantage Yahtzee: The Official Handbook*, Huntington Press (2001).

[12] Verhoeff, T.: Solitaire Yahtzee: Optimal player and proficiency test, Open Source Software (2010), available from ⟨http://www.win.tue.nl/~wstomv/misc/yahtzee/⟩.

[13] Woodward, P.: Yahtzee: The solution, *Chance*, Vol.16, No.1, pp.18–22 (2003).

Sebastian Böcker is a full professor at the Friedrich Schiller University Jena, and leader of the Chair for Bioinformatics at the Institute for Computer Science. Previous to his academic career, he worked for three years in industry. He has a long-standing expertise in bioinformatics, algorithmics, and combinatorial optimization. He is the author of more than 100 refereed papers.