Icepool: Efficient Computation of Dice Pool Probabilities

Albert Julius Liu

Independent [*]
California
ajul1987@gmail.com

Abstract

Mechanics involving the roll of multiple dice—a "dice pool"—commonly appear in tabletop board games and role-playing games. Existing general-purpose dice pool probability calculators resort to exhaustive enumeration of all possible sorted sequences of rolls, which can quickly become computationally intractable. We propose a dynamic programming algorithm that can efficiently compute probabilities for a wide variety of dice pool mechanics while limiting the need for bespoke optimization. We also present Icepool, a pure Python implementation of the algorithm combined with a library of common dice operations.

Introduction

Dice are a common feature in tabletop games, including both board games and role-playing games (RPGs). In terms of the formal rules of a game, dice can be seen as a mechanism for sampling from a discrete probability distribution. Designers may be interested in the probability distribution produced by a dice mechanic in order to adjust the balance towards absolute targets as well as relatively between choices available to the players.

However, the question of what probability distribution is produced by a dice mechanic is interesting *not* because the designer's first and only concern is sampling from a preplanned probability distribution. If it were so, the designer could resort to the use of a look-up table or a computer. Rather, the experience and appeal of rolling dice in a tabletop game is not just a matter of probability—it is a physical, tactile activity as well. Furthermore, a game designer may not start with a specific probability distribution in mind. They may have other considerations, including:

- Can the human players—the "hardware" on which the game is run—evaluate the results of a dice roll easily, quickly, and reliably? For example, a computer can easily sample from a normal distribution at high precision; on the tabletop, the human will likely settle for using the sum of a few dice thrown at once.
- What physical dice does the mechanic require? Using more common, off-the-shelf dice types, especially the ordinary six-sided die, may make a game more financially and logistically accessible. On the other hand, nonstandard dice can attract attention to a game and give it a unique flavor.
- Does the dice mechanic fit the designer's aesthetic and cultural vision for the game? For example, dice mechanics may be inspired by aspects of a fictional setting, an overt instance being the "d616" system of the Marvel RPG (Forbeck et al. 2022), 616 being the number of the primary continuity in that fictional multiverse.

As such, designers often start from these other considerations to create a prospective dice mechanic, and then ask what probability distribution is produced by that dice mechanic, iterating as desired. In turn, players may be interested in knowing the likely results of a given action in the game so that they may come up with scenarios and strategies. Utilities for computing dice probabilities such as Troll (Mogensen 2009) and AnyDice (Flick 2012a) have emerged in order to answer these questions.

Dice Pools While some dice mechanics determine the result from a roll of a single die, others have a player or players rolling a "pool" of multiple dice. For most such mechanics, all of the dice are thrown simultaneously and without order, with the dice being treated as indistinguishable other than the number they show. In other words, the roll of a pool is fully described by a multiset. The in-game consequences of the roll are then evaluated as a function of the multiset according to the rules of the game.

Equivalently, we can consider the sorted sequence of rolls, also known as order statistics. In some cases, the rules explicitly specify that the rolls should be sorted, such as *RISK* (Lamorisse 1957) and *RISK*-like (Mogensen et al. 2016) mechanics, or games in which the players keep some number of the highest individual results out of the multiset, such as the RPGs *Legend of the Five Rings* (*L5R*) up to 4th edition (Carman et al. 2010) and *Cortex Prime* (Banks et al. 2020).

In other cases, sorting the rolls is convenient for evaluation even if the rules do not explicitly say to do so. This includes poker-like game mechanics such as looking for matching sets and/or straights, which appears in board games such as *Yahtzee* (Lowe et al. 1996) and RPGs such as

[*]The author is currently employed by Google. However, this work was done independently. All opinions are the author's own.

CthulhuTech (Grau 2010) and *Legends of the Wulin* (Ramos 2012).

Dice pools may even appear beyond the official rules of the game. For example, one mechanic in *Dungeons & Dragons* (Gygax and Arneson 1974; Mearls, Crawford et al. 2014) and similar RPGs is the random generation of ability scores. The 5th edition (the most recent at time of writing) calls for rolling four six-sided dice and summing the highest three to generate each ability score, repeating the process for each of six scores. Notwithstanding this official guidance, player-invented ability score generation methods remain a perennial subject in the RPG community, e.g. (Sage et al. 2018), with proposals ranging from simple modifications such as rolling seven scores and picking the highest six, to more involved methods such as snake drafting. These too may be analyzed as dice pools, with objectives including targeting a particular average result or achieving some sense of fairness between players.

This Paper We propose a dynamic programming algorithm that can efficiently compute probabilities for dice pool mechanics, as long as they can be efficiently expressed as a incremental calculation in the form of a state transition function. In some cases, this algorithm can compute the solution to previously intractable problems at interactive speed. The scope of bespoke optimization can be limited to minimizing the state space; in most cases, the simplest description of the dice mechanic in this form will suffice. Along with this, we present Icepool, a pure Python implementation of the algorithm combined with a library of common operations on dice.

Prior Work

Multiset Enumeration: General but Not Fast

Troll Troll is a domain-specific language and dice roller available both online and as source code. The accompanying paper (Mogensen 2009) gives a mechanic used in *L5R* up to 4th edition as their example of a lengthy computation, which we shall use as a case study here.

The dice pool mechanic of this game works as follows:

- Roll a pool of `N` 10-sided dice, whose faces are labeled from 1 to 10 inclusive ("d10s").
- For any dice that roll a 10, roll that die again and add it to the result of that die. Repeat as long as you keep rolling 10s. This is sometimes referred to as an "exploding" die.
- The result is the sum of the `M` highest such dice.

The corresponding script given is

```
sum (largest M N#(sum accumulate x := d10
    until x<10))
```

"where `M` and `N` depend on the situation. With `M = 3`, `N = 5` and the maximum number of iterations for accumulate set to 5, Troll takes nearly 500 seconds to calculate the result, and it gets much worse if any of the numbers increase." After recompiling Troll and running it on a more modern Intel i5-8400, this same computation took about 100 seconds. However, the final warning remains dire—increasing the number of dice rolled `N` from 5 to 6 resulted in the computation taking over 6200 seconds. For reference, in the actual game, `M` and `N` can each reach up to 10.

For purposes of comparison, we will continue using the same iteration limit. In principle, the actual maximum number of iterations is unbounded. However, Troll outputs an explicit enumeration of possible final outcomes and their probabilities, which is a convenient format for the user; AnyDice also uses this format, and we have chosen to do so for Icepool as well. The iteration limit is a response to the impossibility of actually outputting an infinite number of final outcomes.

AnyDice At time of writing, AnyDice (Flick 2012a) is perhaps the most popular online dice probability calculator. While AnyDice is not open-source, its API and performance characteristics are consistent with enumerating multisets. Like Troll, AnyDice uses its own domain-specific language. Here is the same calculation as above in AnyDice syntax:

```
set "explode depth" to 4
output [highest 3 of 5d[explode d10]]
```

Note that Troll counts the base d10 as an iteration while AnyDice does not count it as part of the depth, so the actual number of iterations is the same in these two examples.

AnyDice is faster on this, taking only about 2 seconds (running on unknown hardware). However, increasing the number of dice rolled from 5 to 6 causes the script to exceed AnyDice's 5-second timeout. Indeed, in an accompanying article (Flick 2012b), AnyDice itself also gives *L5R* as an example of a difficult-to-calculate mechanic.

The Fundamental Inefficiency In general, any given dice mechanic will correspond to some function over the roll of a dice pool. If such a function is allowed to depend on the entire ordered sequence of dice rolls, it would have to be evaluated once for each possible such sequence, the number of which is exponential in the number of dice in the pool. If such a function only depends on the sorted sequence of dice rolls, the number of possible sequences is polynomial in the number of dice if the number of possible outcomes per die is kept constant. However, this polynomial may be of quite high order—for example, (Mogensen 2009) notes that the number of possible sorted sequences of n d10s grows as $\Theta\left(n^9\right)$, and this is without the "explosion" mechanic used in the above example.

Convolution: Fast but Not General

Efficient convolution-based algorithms do exist for the specific above case of "roll n dice and sum the m highest". That the probability distribution of the sum of two dice can be expressed as a convolution is a classical result in statistics, and this can be repeated to sum multiple dice. (Huber 2016) gives a convolution-based algorithm for dropping the single lowest die, which was later extended by (Scheuer et al. 2020) to drop multiple dice. Unlike multiset enumeration, these algorithms are jointly polynomial with relatively low order in all parameters: the number of faces per die, the number of dice rolled, and the number of dice kept and summed. In fact, with the use of fast Fourier transforms, these are asymptotically faster than the algorithm we present here.

However, these convolution-based algorithms only handle this particular class of dice pool mechanic. Here are some dice pool mechanics that fall outside this class:

Keeping / Dropping Dice at Arbitrary Indexes In certain situations *Cortex Prime* (Banks et al. 2020) may call for dropping both some number of the highest and some number of the lowest dice. In general we may imagine counting dice at particular indexes after sorting them. While in principle it would be possible to extend the above algorithms to cover these cases, the expressions become increasingly complex.

Mixed Dice Pools Dice pools may consist of mixed types of dice. For example, in *Cortex Prime*, the pool may consist of a mixture of d4s, d6s, d8s, d10s, and d12s.

Non-Additive Mechanics Not all dice mechanics simply add the dice together. As noted earlier, *Legends of the Wulin* (Ramos 2012) and *CthulhuTech* (Grau 2010) call for finding matching sets and/or straights in the roll of a dice pool. *RISK* (Lamorisse 1957) and similar mechanics (Isaksen et al. 2016) involve rolling two opposing dice pools, sorting the results of each pool, forming pairs of one die from each pool, and determining a result based on which side had the higher die in each such pair.

Graphical Models

The problem of computing the probability distribution of a dice mechanic could be formulated as a weighted model counting problem where we seek to determine all marginal probabilities, or as an exact inference problem over a graphical model. These can be expressed using a probabilistic programming language such as `Dice` (Holtzen, Van den Broeck, and Millstein 2020) or Figaro (Pfeffer 2016) respectively. These languages are more expressive than Troll, AnyDice, or our own Icepool in the types of observations and inference queries they support.

However, these languages are less well-suited for the dice pool setting in a couple of ways. First, they tend to be considerably more verbose than Troll or AnyDice in this setting; taking the sum of 12d6 takes a few lines, and the *L5R* example given earlier has no obvious expression short of performing a full explicit sort. Second, the action of sorting the dice pool creates a dense network of dependencies between the variables representing the pre- and post-sort rolls. This poses a scaling problem for treewidth-sensitive algorithms such as variable elimination.

These approaches need not be mutually exclusive. Our algorithm could potentially be used to solve a subproblem as part of a larger probabilistic program, with other parts of the program being solved using variable elimination or other methods.

Towards a Fast, General Solution

Ilmari Karonen and Matt Bogosian proposed dynamic programming and iterating over the outcomes of the dice, e.g. 10, 9, 8, ..., 1 for d10s rather than the first die, second die, ...; then using binomial coefficients to weight the probability of a particular number of dice rolling each outcome. They applied these to produce efficient solutions to *Neon City Overdrive* (Karonen and Bogosian 2021; Russell 2020) and *L5R* (Karonen and Bogosian 2020). Beyond the tabletop dice pool setting, (Galgana et al. 2021) used similar ideas as part of an efficient algorithm for evaluating the joint cumulative distribution function of order statistics at a queried point. Our algorithm builds on these ideas to efficiently compute the full probability distribution of “single-pass” functions over discrete order statistics, focusing on dice pool mechanics often found in tabletop games.

Basic Algorithm

We begin with the basic algorithm, which is sufficient to demonstrate the broad form of the inputs and outputs as well as the core efficiency of the method.

Input: Dice Pool

The user provides two inputs: a **dice pool** to evaluate the dice mechanic on, and a **transition function** representing the dice mechanic.

In the most basic case, a dice pool is defined by:

- An individual **die**, defined as a totally ordered set of possible outcomes. Let ℓ be the cardinality of this set. We’ll start by considering the basic case where all outcomes are equally likely.
- The number of dice in the pool n.

Input: Transition Function

The transition function formulates the dice mechanic in question as an incremental calculation. When evaluating a particular roll of the dice pool, the transition function will be called once for each possible outcome of the dice in the pool. The arguments are the current state (or a null value at the beginning, such as `None` in Python), the outcome, and the number of dice in the pool that rolled that outcome. The transition function then returns the next state.

For example, when evaluating a particular roll of a pool of d10s, the sequence of calls would look like this:

```
state = None
state = next_state(state, 1, num_ones)
state = next_state(state, 2, num_twos)
state = next_state(state, 3, num_threes)
# ...
state = next_state(state, 10, num_tens)
```

effectively performing a single pass over the sorted sequence of individual dice, grouped by outcome. Typically the state will contain some sort of running total. For example, this transition function sums the dice in the pool:

```
def next_state(state, outcome, count):
    if state is None:
        return outcome * count
    else:
        return state + outcome * count
```

This is not particularly impressive by itself, as finding the distribution of the sum of dice rolls can be done through repeated convolution as noted previously. The advantage here is that different transition functions can be used to represent different dice mechanics. For example, this transition function produces the largest matching set in the pool and its outcome, e.g. a roll of 4, 4, 4, 5, 8, 8, 8, 9, 9 would result in `(3, 8)`:

```python
def next_state(state, outcome, count):
    if state is None:
        return count, outcome
    else:
        return max(state, (count, outcome))
```

(n.b. In Python, tuples are lexicographically ordered.)

Or, the length of the longest straight, i.e. the largest subset consisting of consecutive numbers:

```python
def next_state(state, outcome, count):
    if state is None:
        best_run = 0
        run = 0
        prev_outcome = outcome - 1
    else:
        best_run, run, prev_outcome = state
    if count >= 1:
        if outcome == prev_outcome + 1:
            run += 1
        else:
            run = 1
        best_run = max(run, best_run)
    else:
        run = 0
    return best_run, run, outcome
```

The user may implement the optional `final_outcome` method to do any desired finalization and/or cleanup to the final states; for example, in the above case they might marginalize out the now-extraneous `run` and `prev_outcome` fields, leaving just `best_run`. Note that `final_outcome` occurs outside of the memoization, as all of these fields are necessary for computing future calls.

Output: Weight Distribution

The output of our algorithm is a distribution of weights equivalent to evaluating the transition function on all possible rolls of the pool and counting how many ended up in each final state. The output itself can be considered to be a die.

The Underlying Algorithm

However, we don't want to actually enumerate all possible rolls of the dice pool; as noted before, the number of such possibilities is a barrier to efficiency. Instead, the incremental formulation of the transition function allows us to use dynamic programming. To find the output for a dice pool of ℓ outcomes per die and n dice, the algorithm recursively uses memoized solutions for dice pools of $\ell - 1$ outcomes per die and $0 \ldots n$ dice.

Specifically, in each call we pop the greatest outcome from the die, and then for $k = 0 \ldots n$:

- Compute how many ways there are for k out of n dice to roll the current outcome. This is just the binomial coefficient $\binom{n}{k}$. These coefficients can be efficiently computed and memoized using Pascal's triangle.
- Recursively compute the solution for a pool with the current outcome removed from the die, and $n - k$ dice in the pool.
- For each state in the recursive distribution, apply the transition function, giving the current outcome and k as the other arguments. Then add the resulting state to the output distribution with weight equal to its recursive weight times the binomial coefficient.

If the last remaining outcome is popped, all n dice must roll that outcome, leading to the base case of a die with an empty set of outcomes and 0 dice in the pool. This is considered to produce a distribution consisting of just the state `None` (Python's null value) with weight 1.

Here is sample Python code for the basic algorithm:

```python
@cache
def solve(die, n):
    outcome, die = die.pop()
    if len(die) == 0:
        state = next_state(None, outcome, n)
        return {state : 1}
    result = defaultdict(int)
    for k in range(n + 1):
        tail = solve(die, n - k)
        for state, weight in tail.items():
            state = next_state(state,
                outcome, k)
            weight *= comb(n, k)
            result[state] += weight
    return result
```

An example call graph is shown in Figure 1:

- Vertexes are unique calls. Since the algorithm is memoized, the state distribution at each vertex will be computed exactly once.
- Edges are all calls. Each edge has weight equal to a binomial coefficient.
- Each path from a vertex to the sink (base case) corresponds one-to-one with a possible sorted sequence of dice rolls of the starting vertex's dice pool. The product of the weights of the edges on the path is the weight of rolling that sorted sequence. This is equivalent to the decomposition of a multinomial coefficient as the product of binomial coefficients as given in (Knuth 1997):

$$\binom{k_1 + k_2 + \ldots + k_\ell}{k_1, k_2, \ldots k_\ell} =$$
$$\binom{k_1 + k_2}{k_1}\binom{k_1 + k_2 + k_3}{k_1 + k_2} \cdots \binom{k_1 + \ldots + k_\ell}{k_1 + \ldots + k_{\ell-1}}$$

  where $k_1 + \ldots + k_\ell = n$. The edge weights on each path from top to bottom are exactly these binomial coefficients from right to left.

Thus, the incremental formulation allows us to consider only each edge (times some number of states) rather than each path.

Running Time

The total number of edges in the graph (= total calls) is $\Theta\left(\ell n^2\right)$. If the number of states in each distribution is

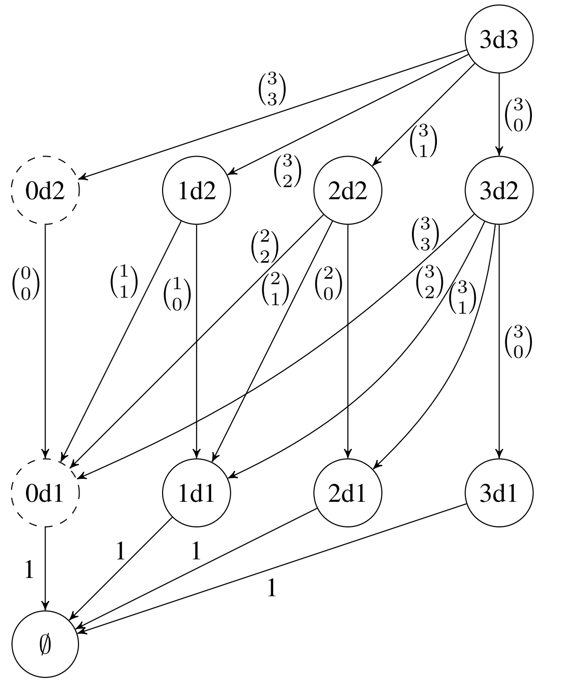

Figure 1: Call graph for 3d3. The vertexes (unique calls) are labeled with the dice pool using “d” syntax; for example, “3d2” means three dice with two sides each. The edges (all calls) are weighted with binomial coefficients. The dashed calls may be elided and redirected to the base case $\emptyset$.

$O\left(s\right)$ then the transition function will be evaluated $O\left(s\ell n^2\right)$ times. The overall running time can be bounded by multiplying this by the mean time accounted to each evaluation of the transition function. Contributors to this are evaluating the transition function itself; dictionary and memoization management (e.g. hashing); and multiplying and accumulating weights. Our Icepool implementation uses exact integer weights, paying an extra cost compared to floating-point in exchange for obviating any concerns about precision. If the total weight (see next section) per die is W, the denominator of n dice is W^n, and it takes $O\left(n \log W\right)$ bits to represent each weight. If the Karatsuba algorithm (Karatsuba and Ofman 1962) is used for integer multiplication, as is done in the default implementation of Python (Craig, Peters et al. 2002), each multiplication takes $O\left(\left(n \log W\right)^{1.585}\right)$ time.

For best efficiency, the state should store the minimum amount of information needed to compute the result, as the time needed to compute the full weight distribution could be up to linear in the number of possible states. In the extreme case, we could store the entire sequence of rolls inside the state, in which case we could see the entire sequence at the end, but this would effectively enumerate all possible multisets and thus provide no efficiency advantage. Effectively, our algorithm allows the transition function to selectively forget information in exchange for efficiency.

Extensions

With the basic algorithm in hand, we can start extending it to cover additional types of game mechanics. Except as noted otherwise, all of these extensions are mutually compatible. Furthermore, apart from the provision for multiple pools, the transition function interface remains unchanged, allowing these extended pool definitions to be flexibly composed with different transition functions.

Non-Uniform Weights

What if not all outcomes are equally weighted? In this case, for an outcome of weight w, the binomial coefficients $\binom{n}{k}$ must be replaced with $\binom{n}{k} w^k$. Fortunately, it is easy to compute a weighted Pascal’s triangle: before adding the previous row to a copy of itself shifted to the right by 1, multiply the shifted copy by w.

Keeping / Dropping Dice

Returning to our common example of dropping some number of the lowest dice from the pool, we can solve this by augmenting the pool definition with a **count-list** of length n, with the element in the ith position specifying whether the die in the ith sorted position should be counted. For example, “roll 5 dice and count the 3 highest” would correspond to the count-list [0, 0, 1, 1, 1]. Whenever we decide that k dice rolled the current outcome, we pop k elements off the end of the count-list, count how many are true, and send that as the “count” to the transition function.

This leads to some further extensions with no extra work:

- Dice can be kept at arbitrary sorted positions rather than just taking them from one end or the other; for example, [0, 1, 0, 1, 0] would drop the lowest, median, and highest dice out of five. This can be useful for e.g. evaluating mechanics where players take turns drafting rolls from a common pool.
- There’s nothing special about booleans: we could count an individual position multiple or even negative times. For example, [-1, 0, 0, 0, 1] combined with the summing transition function would produce the difference between the highest and the lowest die. In principle we could even attach arbitrary data to each sorted position, though so far we have not found practical use for anything beyond integers.

Optimization: Pruning Zeros This optimization is adapted from (Karonen and Bogosian 2020). If, at some point in the call graph, all of the remaining elements of the count-list are zero, then none of the remaining dice will be visible to the transition function—it will see a count of zero for all remaining outcomes regardless of the rolls of the remaining dice. We can therefore pre-emptively remove all dice from the pool in exchange for the product of the weights of the remaining dice in the pool. This prunes columns from one side of the call graph, which results in a significant speedup if most of the lowest or highest dice are dropped.

With this, we can consider the equivalent of the opening example with 6 dice:

```
d10.explode(max_depth=4).keep_highest(6, 3)
```

Problem	Troll	AnyDice	Icepool
12d6	12	< 100	3 +60
L5R roll 5 keep 3	100 000	$< 2\ 100$	16 +60
L5R roll 6 keep 3	6 200 000	-	17 +60
L5R roll 10 keep 5	-	-	37 +60
RISK 3d6 vs. 3d6	370	$< 1\ 000$	4 +60
RISK 4d6 vs. 4d6	10 800	-	7 +60
RISK 5d6 vs. 5d6	390 000	-	12 +60
RISK 10d6 vs. 10d6	-	-	95 +60

Table 1: Comparison of execution times (in ms). Troll and Icepool were run on an Intel i5-8400. Troll does not report internal computation time; the figures for Troll include an estimated 10 ms of startup time. It takes about 60 ms to start Python and `import icepool`, indicated as "+60". AnyDice is hosted on a remote server with unknown hardware and an execution time limit of 5 000 ms, so we have only an upper bound (for computations that don't time out) or a lower bound (for computations that do). *L5R* refers to exploding d10s with at most 4 explosions, as introduced in the Prior Work section. The *RISK* mechanic computes the difference in the number of pairs won by each side, as given in Section 3.1 of (Isaksen et al. 2016), with the Troll and AnyDice programs being taken from (Mogensen et al. 2016).

(Note that, unlike Troll but like AnyDice, the `max_depth` parameter in Icepool does not count the initial roll.) This takes 17 milliseconds, which is over 300 000$\times$ as fast as Troll. Even if we increase the pool to roll 10 keep 5, the execution time is only 37 milliseconds. See Table 1 for a summary of timings.

Multiple and Mixed Pools

Some dice pool systems, such as *RISK* (Lamorisse 1957) and *Neon City Overdrive* (Russell 2020), involve rolling multiple independent pools of dice. In other cases, the mechanic may call for multiple types of dice with different weights for outcomes. These can be handled by simply generating independent counts for each pool and taking the joint distribution. The transition function then receives a count for each pool or a total of counts as appropriate. While relatively expensive (the running time grows exponentially with the number of pools), the number of pools is usually a small constant, so this suffices for most cases.

For example, a *RISK*-like transition function might look like this:

```
def next_state(state, outcome, a, b):
    net_score, advantage = state or (0, 0)
    if advantage > 0:
        net_score += min(b, advantage)
    elif advantage < 0:
        net_score -= min(a, -advantage)
    advantage += a - b
    return net_score, advantage
```

with the outcomes being seen in descending order. `advantage` is the number of unpaired dice that rolled a previous (higher) number. If positive, it favors side A; if negative, it favors side B. At each step we pair these off with any newly-rolled dice of the disadvantaged side.

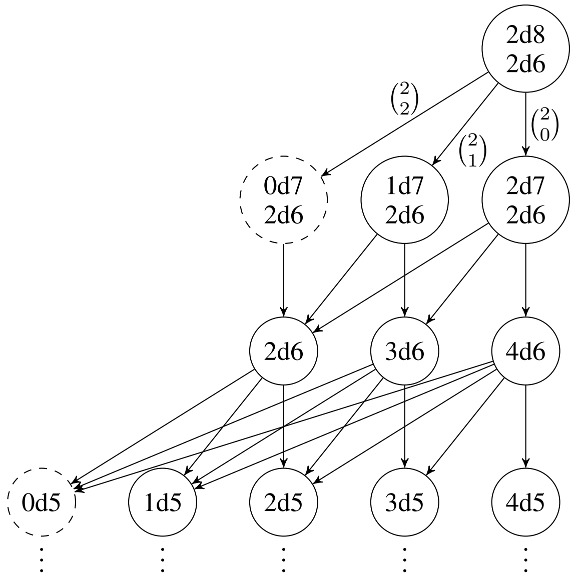

Figure 2: Partial call graph for mixed right-truncated dice, starting with a pool of two eight-sided dice and two six-sided dice ("2d8 and 2d6"). The call graph takes on a staircase rather than a rectangular shape, but otherwise works the same as the non-mixed case. The call at "0d7 and 2d6" may be elided, instead redirecting to "2d6", though guaranteeing consecutive outcome order may be convenient for some transition functions, such as finding straights.

The above dice mechanic is the same as that of Section 3.1 of (Isaksen et al. 2016). Compare the number of evaluations done by various algorithms on a contest of 5d10 versus 5d10:

- $10^{10} = 10\ 000\ 000\ 000$ unsorted sequences
- $\binom{14}{9}^2 = 4\ 008\ 004$ pairs of multisets
- Our algorithm: $< 9\ 000$ state transitions (the exact number depends on whether pools of 0 dice are elided)

Mixed Right-Truncated Dice

If the dice in a pool differ only by right-truncation, then it is possible to compute the result without paying the penalty of using multiple pools. This covers the most common case of a mixture of standard dice: d4s, d6s, d8s, d10s, and d12s. We take advantage of the fact that, conditional on not rolling an 12 or 11, a d12 is no different than a d10; conditional on not rolling a 10 or 9, a d10 is no different than a d8, and so forth. If the call graph proceeds from the greatest to the least outcome, at each call all of the dice we pop an outcome from are identical, and the binomial weighting is still valid. An example call graph for a pool of two eight-sided dice and two six-sided dice is shown in Figure 2. In code, this could be expressed as

```
Pool([d6, d6, d8, d8])[-2:].sum()
```

where the `[]` operator on a pool is used to set the count-list, and the `sum()` method runs the summing transition function over the pool.

Descending Outcome Order

If all the dice in the pool are the same, presenting the outcomes to the transition function in descending order is trivial by symmetry. While many dice pool mechanics could be computed using either ascending or descending order, sometimes one direction is more convenient when writing the transition function, e.g. the *RISK* case above where the dice are paired in descending order; and/or more efficient, e.g. due to the zero-pruning optimization. A more troublesome case is when descending outcome order is combined with mixed right-truncation. In this case we can make the algorithm iterative instead of recursive, and evaluate vertexes in the call graph from top to bottom rather than bottom to top. However, this is less amenable to memoization across multiple queries since call trees tend to be more similar closer to the empty pool (bottom left in the figures).

Cards

A similar strategy can be used for dealing hand(s) of cards from a deck, i.e. sampling without replacement, albeit this formulation is perhaps relatively less congruent to typical card game mechanics. In this case, each weighted binomial coefficient $\binom{n}{k}w^k$ is replaced with $\binom{K}{k}$, where K is the number of cards in the deck showing the outcome under consideration. Truncation as described here is not applicable to card deals, but card deals can employ a count-list, they can be evaluated jointly with each other and/or dice pools, and they have the same considerations regarding iteration order.

Limitation: Post-Roll Decisions

Our algorithm does not handle many cases where the player makes decisions after the pool is rolled. For example:

- *Legend of the Five Rings* 5th Edition (Brooke et al. 2018) uses different dice than its predecessors, with each die having several possible symbols. Particular symbols may be more or less desirable depending on the situation and on the other rolls in the pool, making the choice of which dice to keep non-trivial.
- *Cortex Prime* (Banks et al. 2020) may have the player choose an "effect die" from the pool after rolling. That die cannot be counted as part of the total, but choosing a larger effect die can increase the benefits of a won contest. Thus the player may be forced to choose between a better chance of winning the contest and getting a bigger win if they do win.

While such post-roll decisions could be approached using e.g. Markov decision process techniques, the tabletop setting poses several special challenges:

- The optimal choice may not be an explicit part of the game rules; it may depend on the specific context in which the roll is made and/or on individual player preferences.
- Declaring an explicit policy ("if I roll X then I will choose Y"), or even a utility function, may be difficult for the user.
- Even with a policy in mind, it may not be efficiently expressible in a single pass over the sorted sequence of rolls.

The Icepool Python Package

We have implemented this algorithm as part of the Icepool Python package. In addition, Icepool implements common operators and methods of dice: arithmetic, comparisons, rerolling, exploding, substitution of outcomes, and so forth; as well as provisions for multidimensional and non-integer outcomes. Icepool is available on PyPi, with source code provided at https://github.com/HighDiceRoller/icepool.

Along with the source code, we also present a selection of interactive webpages and over twenty example JupyterLite (Tuloup et al. 2022) notebooks. These demonstrate solutions to a variety of dice pool mechanics found in published games, dice questions posed by users across the Web, and all of the *RISK*-like mechanics proposed in (Isaksen et al. 2016). In particular, the notebook for the last recomputes the entire collection of tables from that paper in under 2 seconds, including replacing Monte Carlo simulations with exact solutions.

In turn, these are powered by Pyodide (Droettboom et al. 2019), which allows Python scripts to be run in a web browser with ample performance for this purpose.

Conclusion

We have presented a general and fast algorithm for computing the probabilities of dice pool mechanics, along with Icepool, a pure Python implementation of the algorithm combined with a library of common dice operations. The efficiency and interoperability of this package enables a wider variety of game mechanics to be analyzed and applications to be developed. We hope this will be a useful tool for players, designers, and analysts of tabletop games alike.

Acknowledgments

We would like to thank Ilmari Karonen and Matt Bogosian for discussions during the development of this project, and the anonymous reviewers for their helpful feedback on the presentation and context of this paper.

References

Banks, C.; et al. 2020. *Cortex Tabletop Roleplaying Game*. Fandom Tabletop.

Brooke, M.; et al. 2018. *Legend of the Five Rings Roleplaying Game*. Fantasy Flight Games, 5th edition.

Carman, S.; et al. 2010. *Legend of the Five Rings Roleplaying Game*. Alderac Entertainment Group, 4th edition.

Craig, C. A.; Peters, T.; et al. 2002. Cautious introduction of a patch that started from SF 560379: Karatsuba multiplication. https://github.com/python/cpython/commit/5af4e6c739c50c4452182b0d9ce57b606a31199f. Accessed: 2022-08-08.

Droettboom, M.; et al. 2019. Pyodide. https://pyodide.org/en/stable/. Accessed: 2022-08-08.

Flick, J. 2012a. AnyDice Dice Probability Calculator. https://anydice.com. Accessed: 2022-08-08.

Flick, J. 2012b. Legend of the Five Rings: Keeping Dice. https://anydice.com/articles/legend-of-the-five-rings/. Accessed: 2022-08-08.

Forbeck, M.; et al. 2022. *Marvel Multiverse Role-Playing Game Playtest Rulebook*. Marvel.

Galgana, R.; Shi, C.; Greenwald, A.; and Oyakawa, T. 2021. A Dynamic Program for Computing the Joint Cumulative Distribution Function of Order Statistics. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*, 160–170. SIAM.

Grau, M. 2010. *CthulhuTech*. WildFire. ISBN 9780984583607.

Gygax, G.; and Arneson, D. 1974. *Dungeons & Dragons*. Tactical Studies Rules, 1st edition.

Holtzen, S.; Van den Broeck, G.; and Millstein, T. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang. (OOPSLA)*.

Huber, W. 2016. Formula for dropping dice (non-brute force). https://stats.stackexchange.com/a/242857/351712. Accessed: 2022-08-08.

Isaksen, A.; Holmgård, C.; Togelius, J.; and Nealen, A. 2016. Characterising score distributions in dice games. *Game and Puzzle Design*, 2(1): 14.

Karatsuba, A. A.; and Ofman, Y. P. 1962. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, 293–294. Russian Academy of Sciences.

Karonen, I.; and Bogosian, M. 2020. Roll and Keep in Anydice. https://rpg.stackexchange.com/a/166663/72732. Accessed: 2022-08-08.

Karonen, I.; and Bogosian, M. 2021. How to calculate the probabilities for eliminative dice pools (dice cancelling mechanic) in Neon City Overdrive? https://rpg.stackexchange.com/a/194712/72732. Accessed: 2022-08-08.

Knuth, D. E. 1997. *The art of computer programming*, volume 1. Addison Wesley Longman.

Lamorisse, A. 1957. *RISK*. Parker Brothers.

Lowe, E. S.; et al. 1996. *Yahtzee*. Milton Bradley Company.

Mearls, M.; Crawford, J.; et al. 2014. *Player's Handbook*. Wizards of the Coast LLC, 5th edition.

Mogensen, T. Æ. 2009. Troll, a language for specifying dice-rolls. In *Proceedings of the 2009 ACM symposium on Applied Computing*, 1910–1915.

Mogensen, T. Æ.; et al. 2016. RISK style resolution probabilites. https://forum.rpg.net/index.php?threads/risk-style-resolution-probabilites.773382/. Accessed: 2022-08-08.

Pfeffer, A. 2016. *Practical probabilistic programming*. Simon and Schuster.

Ramos, D. R. 2012. *Legends of the Wulin*. EOS SAMA, LLC.

Russell, N. 2020. *Neon City Overdrive*. Peril Planet.

Sage, O.; et al. 2018. How can I avoid problems that arise from rolling ability scores? https://rpg.stackexchange.com/questions/133350/how-can-i-avoid-problems-that-arise-from-rolling-ability-scores. Accessed: 2022-08-08.

Scheuer, M.; et al. 2020. Generating Function for sum of N dice [or other multinomial distribution] where lowest N values are "dropped" or removed. https://math.stackexchange.com/a/3792882/1035142. Accessed: 2022-08-08.

Tuloup, J.; et al. 2022. JupyterLite. https://github.com/jupyterlite/jupyterlite. Accessed: 2022-08-08.