Icepool: Efficient Computation of Dice Pool Probabilities

Albert Julius Liu [*]

June 17, 2022

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 multisets, which can quickly become computationally infeasible. 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.

1 Introduction

Dice are a common feature in tabletop games, including both board games and role-playing games. While some dice mechanics in these games determine a result from a roll of a single die, others have a player or players rolling a "dice 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 raw roll of a pool is described by a multiset, or equivalently, a sorted sequence. The result of the roll is then computed as a function of the multiset.

Existing general-purpose approaches are based on exhaustive enumeration of all such multisets. While this makes them very flexible in terms of what dice mechanics they can represent, exhaustive enumeration can become computationally infeasible as the number of dice in the pool and/or the number of possible outcomes per die increases.

At the other end of the spectrum are specialized algorithms for particular dice mechanics, most notably convolution-based algorithms for summing the highest m dice for a pool. While efficient, these algorithms only solve this specific dice pool problem.

In 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 infeasible problems at interactive speed. The scope of bespoke optimization can be limited to minimizing the

[*]The author is currently employed by Google LLC. However, this work was done independently.

state space; in most cases, the simplest description of the dice mechanic as an incremental calculation will suffice. Along with this, we present Icepool, a pure Python implementation of the algorithm combined with a library of common operations on dice.

2 Prior work

2.1 Multiset enumeration: general but not fast

Troll

Troll is a dice roller available both online and as source code. The accompanying paper [17] gives a mechanic used in *Legend of the Five Rings* [3] 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 [6] 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. 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 [7], AnyDice itself also gives *Legend of the Five Rings* as an example of a difficult-to-calculate mechanic.

The fundamental issue

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 an sorted sequence of dice rolls, the number of possible sequences is polynomial in the number of dice if the number of faces per die is kept constant. However, this polynomial is usually of quite high order—for example, [17] 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 previous examples.

2.2 Convolution: fast but not general

In fact, 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. [10] gives a convolution-based algorithm for dropping the single lowest die, which was later expanded by [21] to drop an arbitrary number of dice. Unlike the previously-mentioned multiset-based algorithms, 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 (FFTs) they can be 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* [1] 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 scoring

Not all dice mechanics simply add the dice together. *Legends of the Wulin* [19], and *CthulhuTech* [8] call for finding matching sets and/or straights in the roll of a dice pool. *RISK* [16] and similar mechanics [11] 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.

2.3 Weighted model counting

The problem of computing the probability distribution of a dice mechanic could be formulated as a weighted model counting (WMC) problem where we seek to determine all marginal probabilities. Problems of this type were analyzed in the `Dice` probabilistic programming language [9], using factorization to gain a performance advantage over exhaustive enumeration. `Dice` supports integer-valued variables via a one-hot representation. However, this system is less well-suited for dice mechanics found in tabletop games in both syntax and performance; for example, `Dice` takes over 18 s to compute the distribution of the sum of 12d6, a task that Troll can complete in 12 ms with much simpler syntax.

2.4 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* [14, 20] and *Legend of the Five Rings* [13, 3]. Our algorithm builds on these basic ideas to produce a fast, general solution to dice pool probabilities.

3 Core algorithm

3.1 Input: dice pool

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

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

- A **base die**, which is the set of possible outcomes of each individual die in the pool along with a corresponding integer weight w for each outcome. Let q be the cardinality of the outcome set. We'll start by considering the basic case where all outcomes have weight 1.

- The number of dice in the pool n.

3.2 Input: state transition function

The state 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 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)
```

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 on its own, as addition of dice can be done through repeated convolution as noted previously. The advantage here is that the transition function can be changed 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)`.

```
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:

```
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
```

3.3 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.

3.4 The underlying algorithm

However, we don't want to actually enumerate all possible rolls of the dice pool; as shown 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 q outcomes per die and n dice, the algorithm recursively uses memoized solutions for dice pools of $q-1$ outcomes per die and $0 \ldots n$ dice.

Specifically, in each call the algorithm selects one outcome (e.g. the greatest) from the base 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 cached using Pascal's triangle.
- Recursively compute the solution for a pool with the current outcome removed from the base 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 there is only a single outcome left, all n dice must roll that outcome, leading to the base case of a base die with an empty set of outcomes and 0 dice in the pool. This is the base case and is considered to produce a distribution consisting of just the state `None` (Python's null value) with weight 1.

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.

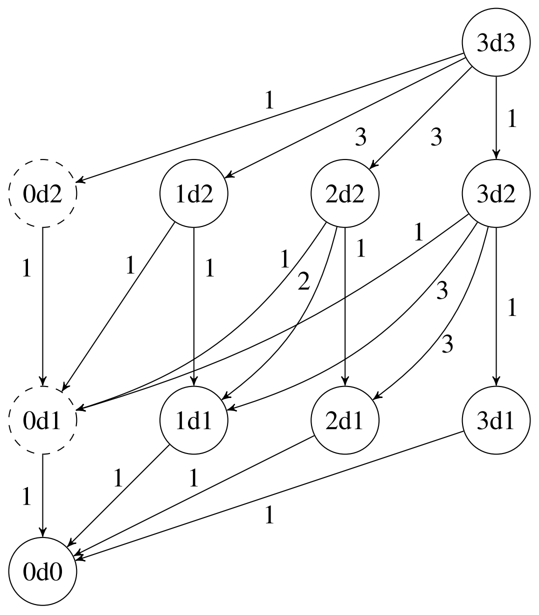

Figure 1: Call graph for 3d3. The vertexes (unique calls) are labeled with the dice pool using “d” syntax; for example, “3d2” means three dice with two sides each. The edges (all calls) are weighted with binomial coefficients. The dashed calls may be elided and redirected to “0d0”.

- 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 [15]:

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

  where $k_1 + \ldots + k_q = n$. 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 rather than each path.

3.5 Running time

The total number of edges in the graph (= total calls) is $\Theta\left(n^2 q\right)$. If the number of states in each distribution is $O(s)$ then the transition function will be evaluated $O\left(sn^2 q\right)$

times. The overall running time can be bounded by multiplying this by the mean time taken for each evaluation of the transition function. Contributors to this are the time needed to evaluate the transition function itself and the time needed to multiply and accumulate weights. Our Icepool implementation uses exact weights. If the total weight (see next section) per die is W, the denominator of n dice is W^n, and it takes $O(n \log W)$ bits to represent each weight. If the Karatsuba algorithm [12] is used for integer multiplication, as is done in the default implementation of Python [4], each multiplication takes $O\left((n \log W)^{1.585}\right)$ 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 provide no advantage over enumerating all possible multisets. Effectively, our algorithm allows the transition function to selectively forget information in exchange for efficiency.

4 Extensions

Except where noted otherwise, all of these extensions are mutually compatible.

4.1 Non-uniform weights

What if not all outcomes are equally weighted? In this case, for an outcome of integer 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.

4.2 Keeping/dropping dice

Returning to our common example of dropping some number of the lowest dice from the pool, the solution is simple: Augment the pool definition with a **count-list** of length n, with the element in the ith position specifying whether the ith highest die in the pool 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, pop k elements off the end of the count-list and count how many are true, and send that as the "count" to the transition function.

In fact, 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 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

Problem	Troll (ms)	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 about 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 Section 2.1. The *RISK* problem is as given in Section 4.3, with the Troll and AnyDice programs being taken from [18].

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 we have not found any practical use for anything beyond integers.

Optimization: pruning zeros

This optimization is adapted from [13]. 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)
```

(Note that, unlike Troll but like AnyDice, the `max_depth` parameter here does not count the initial die.) 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 run time is only 37 milliseconds. See Table 1 for a summary of timings.

4.3 Multiple and mixed pools

Some dice pool systems, such as *RISK*-like [16] mechanics and *Neon City Overdrive* [20], 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. 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. The user may use 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 remove the now-extraneous `advantage` field, leaving just `net_score`, the difference in the number of pairs won by each side.

The above dice mechanic is the same as that of Section 3.1 of [11]. 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: 8 815 state transitions

4.4 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. The most common case is 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, all of the dice we consider at each call are identical, and the binomial weighting is still valid. To represent such a pool, we can choose the largest die as the base die, and then augment the pool definition with a truncation threshold for each die in the pool. 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.

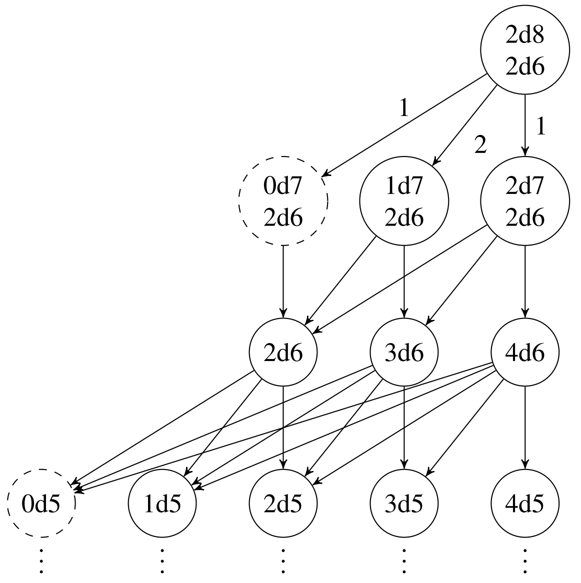

Figure 2: Partial call graph for mixed right-truncated dice, starting with a pool of two eight-sided dice and two six-sided dice ("2d8 and 2d6"). The call graph takes on a staircase rather than a rectangular shape, but otherwise works the same as the non-mixed case. The call at "0d7 and 2d6" may be elided,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

4.5 Outcome iteration order

Reversing the 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, often one direction is more convenient, 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 slightly more troublesome case is the mixed-right truncation case. 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 persistent caching since call trees tend to be more similar closer to the base case (bottom left in the figures).

Why not arbitrary order?

We could imagine iterating over the outcomes in other than a strictly monotonically ascending or descending order, which could allow the order to be driven by the transition function and/or allow for efficient evaluation of more types of mixed pools. However, the benefits seem to be outweighed by the costs:

- A monotonic order makes it easier to use the API and to reason about the efficiency of a particular transition function.
- The count-list extension only works when outcomes are taken from the ends.
- We've seen few practical cases where a different iteration order would make the difference in whether a problem is solvable.

4.6 Cards

A similar strategy can be used for drawing a hand of cards from a deck, i.e. sampling without replacement. In this case, each binomial coefficient $\binom{n}{k}$ is simply replaced with $\binom{K}{k}$, where K is the number of cards in the deck showing the outcome under consideration. Weighting and truncation are not applicable to card draws, but card draws 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.

5 The Icepool Python package

We have implemented this algorithm as part of the Icepool Python package, available on PyPi and at https://github.com/HighDiceRoller/icepool. In addition to this, Icepool implements common operators and methods of dice: arithmetic, comparisons, rerolling, exploding, subroutines, and so forth; as well as provisions for multidimensional and non-integer outcomes. Along with the package itself, we also present a selection of interactive webpages and example JupyterLite [22] notebooks. In turn, these are powered by Pyodide [5], which provides reasonable performance even when running through a web browser. These applications and notebooks demonstrate solutions to several published dice pool mechanics, dice pool questions posed by users across the Web, and all of the *RISK*-like mechanics proposed in [11]. In particular, the notebook for the last recomputes the entire collection of tables from that paper in under 2 seconds.

6 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 [2] uses different dice than its predecessors with several possible symbols on each die. 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* [1] 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 RPG 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 the incremental formulation used by our algorithm.

7 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. 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, developers, and analysts of tabletop games alike.

References

[1] Cam Banks et al. *Cortex Tabletop Roleplaying Game*. Fandom Tabletop, 2020.

[2] Max Brooke et al. *Legend of the Five Rings Roleplaying Game 5th Edition*. Fantasy Flight Games, 2018.

[3] Shawn Carman et al. *Legend of the Five Rings Roleplaying Game 4th Edition*. Alderac Entertainment Group, 2010.

[4] Christopher A. Craig, Tim Peters, et al. Cautious introduction of a patch that started from sf 560379: Karatsuba multiplication. https://github.com/python/cpython/commit/5af4e6c739c50c4452182b0d9ce57b606a31199f, 2002.

[5] Michael Droettboom. Pyodide: Bringing the scientific python stack to the browser. https://hacks.mozilla.org/2019/04/pyodide-bringing-the-scientific-python-stack-to-the-browser/, 2019.

[6] J Flick. Anydice dice probability calculator. https://anydice.com, 2012.

[7] J Flick. Legend of the five rings: Keeping dice. https://anydice.com/articles/legend-of-the-five-rings/, 2012.

[8] Matthew Grau. *CthulhuTech*. WildFire, 2010.

[9] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang. (OOPSLA)*, 2020.

[10] William Huber. Formula for dropping dice (non-brute force). https://stats.stackexchange.com/a/242857/351712, 2016.

[11] Aaron Isaksen, Christoffer Holmgård, Julian Togelius, and Andy Nealen. Characterising score distributions in dice games. *Game and Puzzle Design*, 2(1):14, 2016.

[12] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[13] Ilmari Karonen and Matt Bogosian. Roll and keep in anydice. https://rpg.stackexchange.com/a/166663/72732, 2020.

[14] Ilmari Karonen and Matt Bogosian. How to calculate the probabilities for eliminative dice pools (dice cancelling mechanic) in neon city overdrive? https://rpg.stackexchange.com/a/194712/72732, 2021.

[15] Donald Ervin Knuth. *The art of computer programming*, volume 1. Addison Wesley Longman, 1997.

[16] Albert Lamorisse. *RISK*. Parker Brothers, 1957.

[17] Torben Ægidius Mogensen. Troll, a language for specifying dice-rolls. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1910–1915, 2009.

[18] Torben Ægidius Mogensen et al. Risk style resolution probabilites. https://forum.rpg.net/index.php?threads/risk-style-resolution-probabilites.773382/, 2016.

[19] David Ramirez Ramos. *Legends of the Wulin*. EOS SAMA, LLC, 2012.

[20] Nathan Russell. *Neon City Overdrive*. Peril Planet, 2020.

[21] Markus Scheuer. 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, 2020.

[22] Jeremy Tuloup et al. Jupyterlite. https://github.com/jupyterlite/jupyterlite, 2022.