Should I attack in Risk? (The Board Game)

tl;dr

I played around a bit trying to calculate what can happen when a country attacks another one in risk. It turned out to be a fun problem to think about, so I did a bit of an overkill. Scroll down to see a few exemplary results!

1 Motivation and Rules

The Risk games in my circle of friends rarely came to an end. Furious negotiations, out-of-game brawls over broken contracts, but also endless dice rolls in epic 30 vs. 20 troops battles often kept the game going until deep into the night. Most of the time we would break off, document the score in order to play it out on another day and then never finish the game. The dice rolling in particular is a big problem, because we play with what we call "Die Michelsonsche Verzögerungstaktik". This is just like the official one, except that only one troop can be lost per dice roll:

The attacker rolls

- 1 dice with 1 attacking troop (if there are x troops on a land, he can attack with $x - 1$)

- 2 dice with 2 attacking troops

- 3 dice with 3 or more attacking troops

The defender rolls

- 1 dice with 1 defending troop

- 2 dice with 2 or more defending troops

After attacker and defender rolled their dice, the highest of each is compared. If the defender's highest dice is equal to or higher than the attacker's highest dice, the attacker looses a troop. Otherwise, the defender looses one.

When 30 attackers fight against 20 defenders, this really takes some time. So naturally the question arises: Can this be calculated? Of course!

2 Single Probabilities

The possible outcomes of a single dice roll can be calculated by simple counting of possible outcomes. In total there are 6 possible scenarios for (1v1, 1v2, 2v1, 2v2, 3v1 or 3v2 dice). With a couple of for-loops, this is quickly done, e.g. for the case 2 attackers vs 1 defender:

```
wins_for_attacker_2v1 = 0
for a1 in range(1, 7):
    for a2 in range(1, 7):
        for d1 in range(1, 7):
            if d1 < max(a1, a2):
                wins_for_attacker_2v1 += 1
p_2v1 = wins_for_attacker_2v1 / 6**3
```

The probabilities for the attacker to win a single dice roll are:

attackers \ defenders	2 or more	1
3 or more	47%	66%
2	39%	58%
1	25%	42%

Not too impressive for the attacker if you ask me!

3 Complete Attacks

Now that we have the single probabilities, the next step is to think about an entire attack: What is the probability to win with a attackers against d defenders?

3.1 Top Down

The first idea that comes to mind is recursion:

To lead from a vs d to either victory (? vs 0) or defeat (0 vs ?), all possible paths have to be calculated, weighed with their individual probability and summed up. But from a local perspective, this simplifies as there are really only two possibilities: You either win a single roll or you loose one:
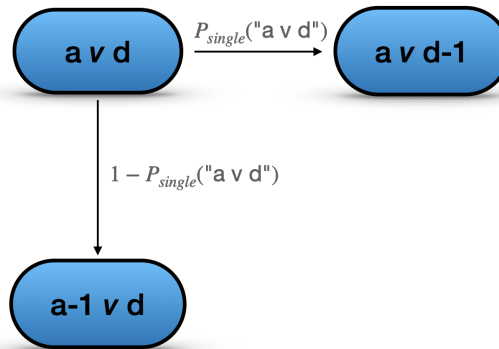
So the relation is obvious:

$$P_{total}(\text{``}a \text{ vs } d\text{''}) = P_{single}(\text{``}a \text{ vs } d\text{''}) \cdot P_{total}(\text{``}a \text{ vs } d-1\text{''})$$
$$+ (1 - P_{single}(\text{``}a \text{ vs } d\text{''})) \cdot P_{total}(\text{``}a - 1 \text{ vs } d\text{''})$$

The base cases are where no attacker or defender is left:

$$P_{total}(\text{``}a \text{ vs } 0\text{''}) = 1 \text{ and}$$
$$P_{total}(\text{``}0 \text{ vs } d\text{''}) = 0$$
$$\forall a, d \geq 1$$

In code, this is even easier to read:

```python
def p_recursive(self, a, d):
    """Returns the probability to win an attack with a attackers against d defenders.
    """
    if a == 0:
        return 0
    elif d == 0:
        return 1
    else:
        return self.basic_probs.p_swin(a, d) * self.p_recursive(a, d - 1) + \
            (1 - self.basic_probs.p_swin(a, d)) * self.p_recursive(a - 1, d)
```

For small attacks, this works just fine. For example, $P_{total}(\text{"10 vs 10"}) = 36.9\%$ (which is surprisingly low). But when the numbers increase, e.g. with 15 vs 15 my laptop already gives up. But why?

It turns out, by design this 2d recursion scales pretty catastrophically. The first node gets called only once, but subsequent ones get called increasingly often:
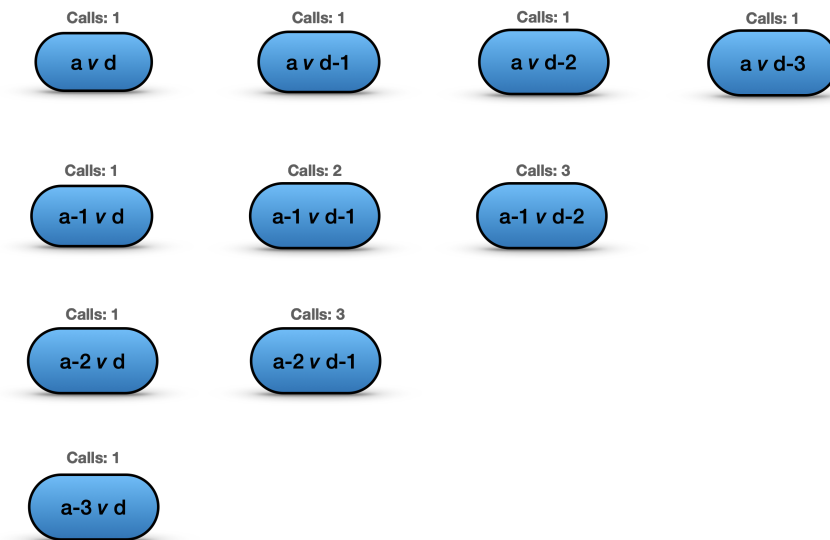
Notice that each number of calls is of course just the sum of the number of calls for the node above and the one on the left of the current one. The ones on the outside can only get called once each, as they only have one left or upper neighbor.

Pascal's triangle appeared out of nowhere!

So how many possible paths are there to get from the starting field to another one, assuming that they have a difference of a attackers and d defenders?

$$M(a, d) = \frac{(a + d)!}{a!d!}$$

This can be shown with induction, using the previous equation as Induction Hypothesis (IH): The start with $(a, d) = (0, 1)$ or $(1, 0)$ is trivial. Now going from (a, d) $(a + 1, d)$:

$$\begin{aligned} M(a + 1, d) &= M(a, d) + M(a + 1, d - 1) \\ &\overset{\text{IH}}{=} \frac{(a + d)!}{a!d!} + \frac{(a + 1 + d - 1)!}{(a + 1)!(d - 1)!} \\ &= \ldots \\ &= \frac{(a + 1 + d)!}{(a + 1)!d!} \end{aligned}$$

This is symmetric w.r.t exchanging a and d, since the critical step in line 2 applies equally (any field can be reached from the fields left and above of it). When fighting with 30 against 20, the field $(1, 1)$ alone is called $\frac{(29+19)!}{29! \cdot 19!} \approx 10^{13}$ times. No wonder my laptop gave up!

Although I didn't realize it at the time I initially did this calculation (it was pretty much my first program), the easiest solution would be to use a cache, e.g. the `functiontools.lru_cache` decorator. The value of each node is calculated only once and looked up from the cache when needed again. This means that all child calls that would be triggered on subsequent calls fall away too, and therefore it should solve the whole issue.

But instead I came up with this idea, that works as well: One can change the conceptual view of going top down towards starting at the possible outcomes and working one's way up towards the initial starting point:

3.2 Bottom Up

The basic idea is to work from the bottom up. Given the probabilities in the fields under and on the right of any field, the probability to win starting from there can be calculated with exactly the same relation as before. The difference is just to start at the bottom. There the probabilities are already known, which sets up the boundary conditions:

- Wherever $a = 0$, no attackers are left and $P = 0$ (visualized in red, for unsuccessful attack)

- Wherever $d = 0$, no defenders are left and $P = 1$ (visualized in green)
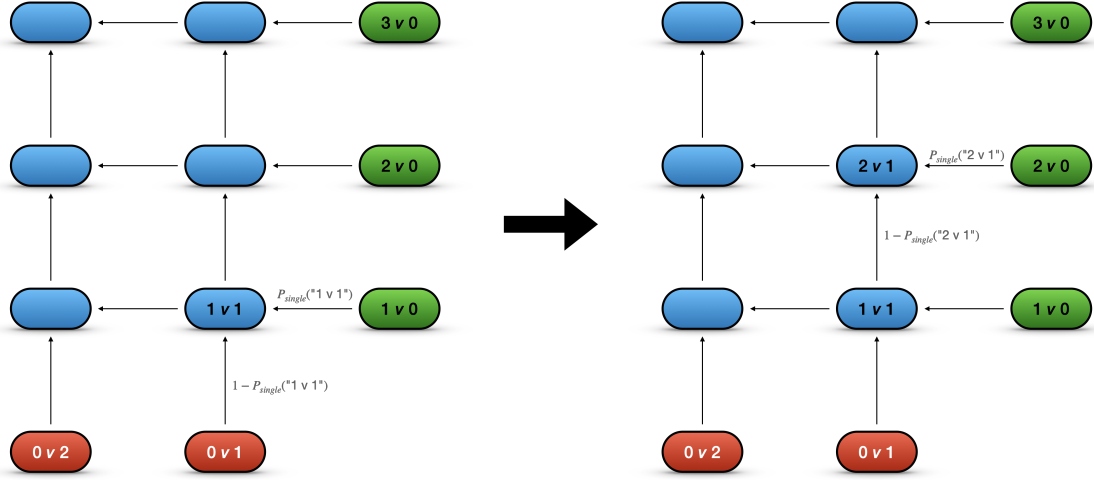
And so on: Now it can just be iterated from the bottom right going up the columns right to left or the rows bottom to top. All $a \cdot d$ fields must be filled up, but this time only once! Using this, values up to ~1000 can easily be reached: P_{total}("1000 vs 1000") $= 0.05\%$ (for such large numbers the 47% of a single 3v2 become very small)

The idea of boundary conditions is also convenient because they now can easily be changed. For example to answer questions like *"How probable is it to win with the attacker having exactly 3 remaining troops?"*, this would just mean changing all outcomes to red (or $P = 0$), except for the one where $a = 3$:
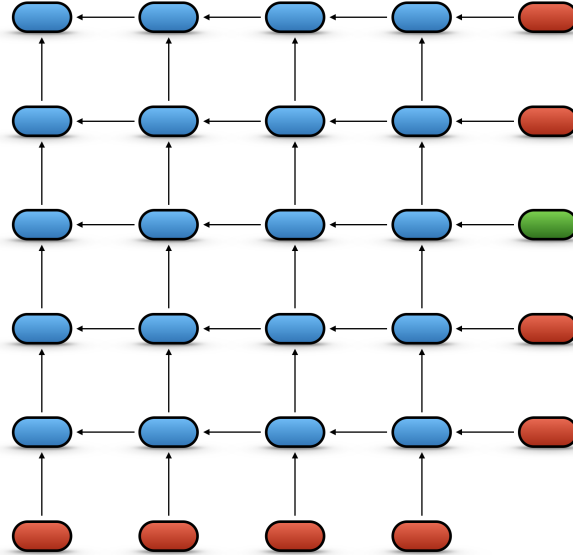

4   Results

Using this, here are some examples of attacking scenarios and the probabilities of each of their possible outcomes:

5 Angreifer gegen 5 Verteidiger.
Gesamtwahrscheinlichkeit für einen Sieg: 38.1%
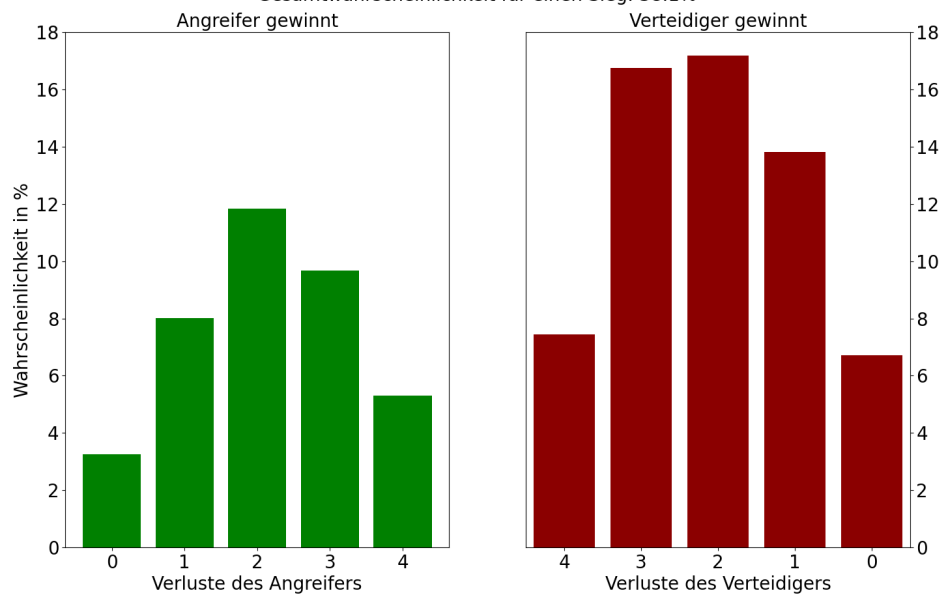
10 Angreifer gegen 8 Verteidiger.
Gesamtwahrscheinlichkeit für einen Sieg: 57.5%

30 Angreifer gegen 25 Verteidiger.
Gesamtwahrscheinlichkeit für einen Sieg: 58.9%
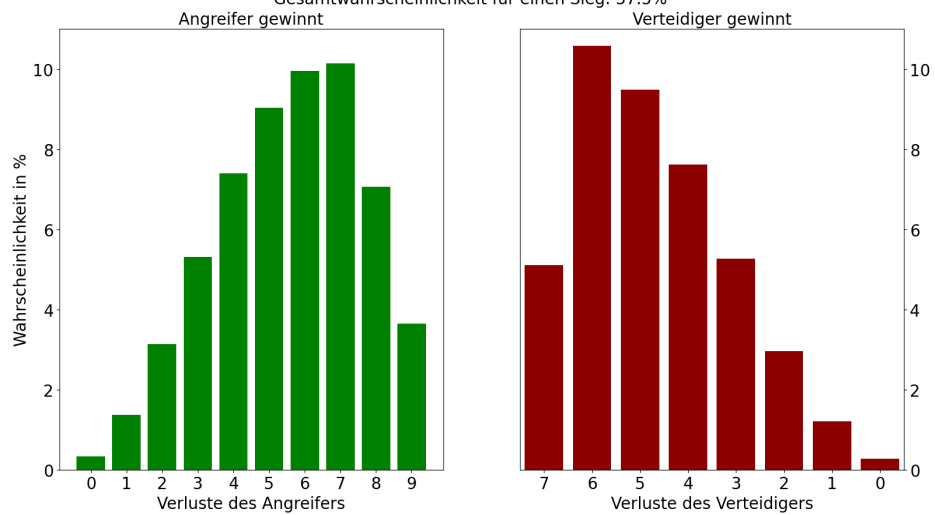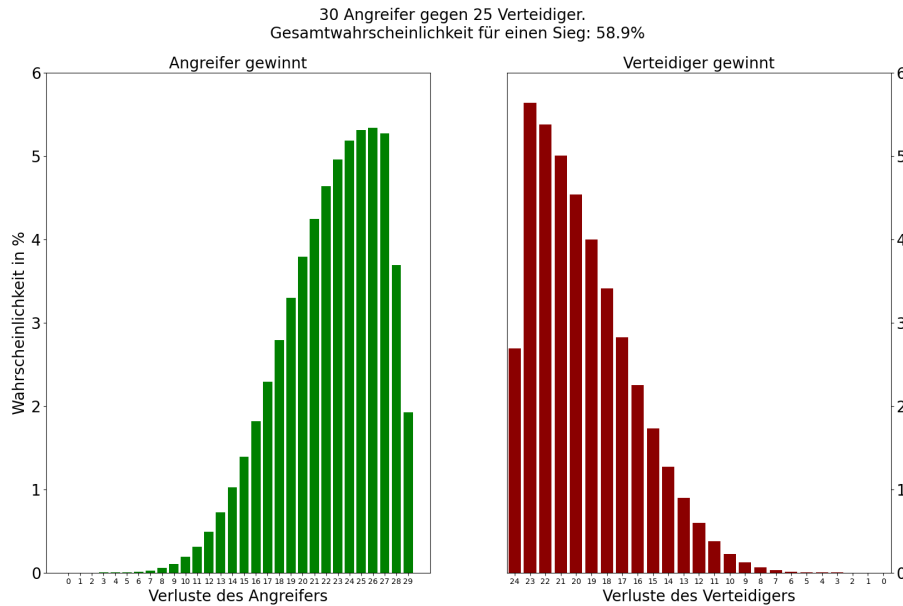
What can be seen is almost a binomial distribution, but with dips where the attacker or defender runs into cases where the number of dice that can be used gets reduced.

For smaller troupe sizes, what strikes me is that the curves are pretty flat. This reflects what we all know: Attacks are often just wildcards and everything can happen.

5   Conclusion

It is surprising how much time one can dump into something like this. But at least now we have the possibility to simulate the outcome of an attack within seconds instead of fifteen minutes. So for the first time, we started actually finishing games! Now that extinguishing a player happens within a second instead of in a 15 minutes crushing face-off, the threshold of doing so significantly reduced. Funnily enough a new threat came up in the later stages of the game: *"If you attack me, I will not simulate"*, which means the whole table will have to wait for quite some time until they are done...

6   Outlook

Now that I've got a way to simulate 1000 vs 1000 one might say that's enough. This is of course not enough!

To reach the really high numbers, an analytical approach would be necessary. It might work like this:

7

Happy Field of Binomial Distribution

Geometric Border

Double Geometric Border
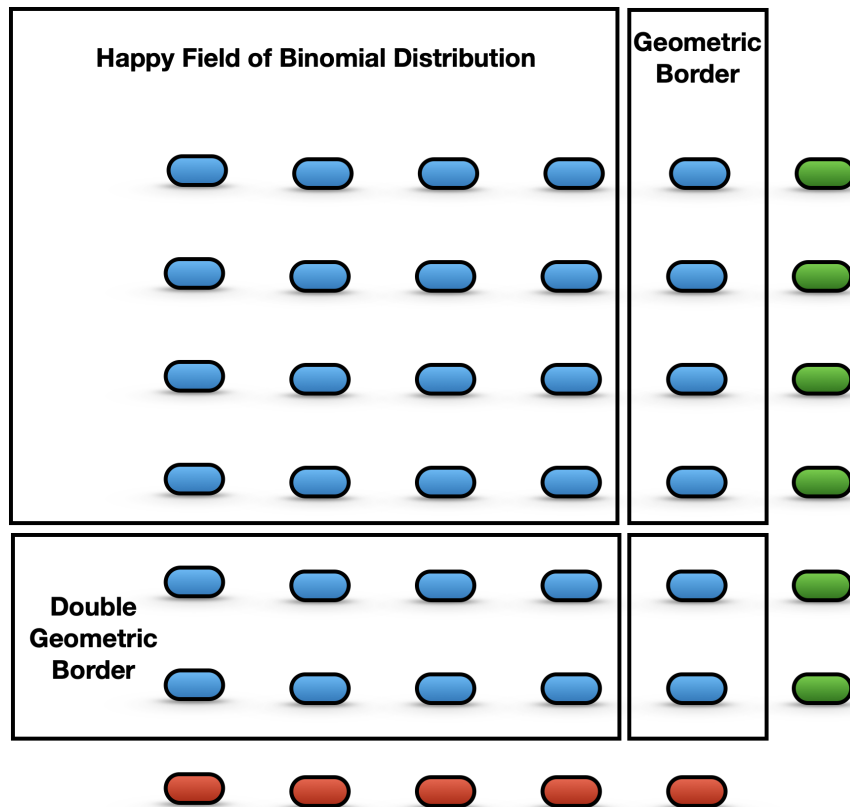
Within the Happy Field of Binomial Distribution, the probability of winning a single attack stays constant as the dice numbers don't change. So the probability to go from point A to point B can instantly be calculated using the binomial distribution. Then one would sum up all paths from the initial point to the border and weigh it the probability on the respective field on the border. For the right border ("Geometric Border"), this would come down to something like the geometric series, so it could be written down for each point. For the lower border, maybe some sort of "double" geometric series will come out... In total this seems pretty doable to me!

7 Addendum

About one and a half months after I posted this article as a TNG internal blogpost, Stephan Steinfurter, a friend and colleague of mine posted this fabulous answer, which is given here in original form:

Thanks again for the nice problem. I was a little nerd-sniped ;).

Indeed, one can obtain a solution linear in $(a + d)$, exactly as you already outlined, Jonas. For the borders, there exist simple analytical solutions and the "happy field of binomial distribution" was in fact not happy at all for me. It was rather painful to get all the indices right. But in the end one indeed just needs the values on the borders and add them up with appropriate pre-factors (so, it is indeed just counting). But getting the binomial coefficients right is a bit nasty.

I have a Jupyter notebook with all the nitty-gritty details. I may upload it to BitBucket, if anybody is interested. Here are the most important functions:

```
1  def Q_geometric_d_is_1(a, d):
2      assert d == 1
3      if a == 1:
```

```python
          return p(1, 1)
      if a == 2:
          return p(2, 1) + (1 - p(2, 1)) * p(1, 1)
      return 1 - (1 - Q_dp(3, 1)) * (1 - p(3, 1))**(a-3)

def Q_geometric_a_is_1(a, d):
    assert a == 1
    if d == 1:
        return p(1, 1)
    return Q_dp(1, 2) * p(1, 2)**(d-2)

def Q_geometric_a_is_2(a, d):
    assert a == 2
    a_ = p(2, 2)
    b_ = (1 - p(2, 2)) * Q_dp(1, 2)
    c_ = p(1, 2)
    return Q_dp(2, 2) * a_**(d-2) + b_ * c_ * (c_**(d-2) - a_**(d-2))/(c_ - a_)
```

To obtain the last one from the previous, one needs to observe that Q(1, d) is a simple exponential and then use sth. like this formula.

The rest is counting:

```python
from math import comb
def Q_linear(a, d):
    return p(3, 3)**(d-1) * sum(comb(a + d - 4 - aa, d - 2) * (1 - p(3, 3))**(a
    - 2 - aa) * Q_geometric_d_is_1(aa+2, 1) for aa in range(a-2))
```

or as a formula:

$$Q_{ad} = p_{33}^{d-1} * \sum_{a'=1}^{a-2} \binom{a + d - 4 - a'}{d - 2} (1 - p_{33})^{a-2-a'} * Q_{a'+2,1}$$

$$+ (1 - p_{33})^{a-2} \sum_{d'=1}^{d-1} \binom{a + d - 4 - d'}{d - 1 - d'} * p_{33}^{d-1-d'} * Q_{2,d'+1}$$

The binomial coefficients can be visualized in Pascal's triangle as the entries going up diagonally left/right from a particular point in the triangle. That particular point has N = (a - 3) + (d - 2) and k = d - 2 when written as "N choose k".

Is it enough to compute Q(1000, 1000)?

Well, in principle yes, sure. But in that particular version (^), I get an "OverflowError: int too large to convert to float" for "comb". I guess, I will just have to switch to scipy's comb with "exact=False" or sth. like that. But who cares about the actual values...