# Maximizing Growth Rates of Option Portfolios

An asset management firm desired to reduce the run time of an optimization routine for a new investment strategy. The strategy focused on selling options where investors received an upfront premium and ran the risk of losses with high severity but with a very low probability. Due to the non-Gaussian distribution of portfolio outcomes, typical mean-variance optimization approaches were found to be insufficient. The fund manager elected to maximize the growth rate of the portfolio, or the geometric mean. Due to a non-linear correlation structure between the options in the portfolio, mathematical approaches became overly complex as the number of options in the portfolio increased.

The fund manager provided a simulation of gains and losses with 10,000 outcomes for ten potential investment options. The fund manager sought a routine to find the optimal allocation of capital to each of the ten options (in increments of 1%) with a run time of 60 seconds or less. I spent the last summer working on this problem. A brute force approach for ten potential investment options requires approximately $2 * 10^{25}$ calculations. As an aside, an explanation of the cost follows in the next paragraph.

Given a set of weights for each of the ten options ($w_1$, $w_2$, … $w_{10}$) and for a given simulated outcome ($o_1$, $o_2$, … $o_{10}$), to calculate the geometric mean we first need to find the weighted outcome ($w_1*o_1 + w_2*o_2 + \ldots + w_{10}*o_{10}$) which requires ten multiplications and nine additions. These 19 calculations need to be repeated for each of the 10,000 simulations for a total of 190,000 calculations to calculate all of the weighted simulation outcomes. The geometric mean is the 10,000th root of the product of the 10,000 weighted simulation outcomes. For the sake of simplicity, we will assume this is another 10,000 calculations for a total of roughly 200,000 ($2 * 10^5$) calculations to find the geometric mean for a given set of weights. A brute force approach requires us to consider all potential combinations of weights. Each option has roughly 100 potential weights (1% increments between 0% and 100%). For ten options, we would have to consider $100^{10}$ or $10^{20}$ potential combinations of weights. Each combination of weights requires $2 * 10^5$ calculations, so a brute force approach to finding the geometric mean for ten options requires $2 * 10^{25}$ calculations. More generally, for n options, the brute force approach requires $2 * 10^{(2n+5)}$ calculations. Reducing the number of weights we consider is our best opportunity to speed the optimization. For the sake of simplicity of this cost calculation, we ignore the comparisons required to find the maximum outcome as that number is also a function of the number of weights we consider.

To reduce the number of weights the optimization considered and minimize the run time, I developed a hill climber in Python. I also considered a heuristic approach, but due to the number of variables and the complexity of the non-linear correlations between the options, a useful heuristic was difficult to create.

I first built test code to solve the problem for two options. I used recursion to move the hill climber from point to point. At each point inside the recursion, nested for loops run through all the possible combinations of points we could move to next. The delta for the weight of each option is one percent. That is, in each recursion, each option is moved up one percent, down one percent, or left the same. All possible combinations of moves are evaluated, and the move with the best geometric mean is the move chosen. For example, let's say there are two options, and currently the weights for each are 40 and 45. The points that would be checked are (39,44), (39,45), (39,46), (40,44), (40,45), (40,46), (41,44), (41,45), and (41,46). After checking all the points, a recursive call is made to the point with the highest geometric mean. If the point with the highest geometric mean is the original point, then the recursion has reached a max and ends.

The benefits of the hill climber over the brute force approach for two options was roughly a factor of 10 in the worst case scenario. For two options, the brute force approach would require $2 * 10^{(2*2+5)}$ or $2 * 10^{9}$ calculations (excluding comparisons). The hill climber requires far less. Assuming the longest path for two options, we assume the answer is (100%,0%) and we start with the worst seed (0%,0%) then at most we need to take 100 steps of 1% each to arrive at the correct answer. (Note: the sponsor advised that the path would likely not require backtracking by the hill climber when starting at 0,0 as their research suggested that the geometric mean as a function of the selected weights generated a concave down curve.) With each step of the hill climber, we only need to calculate $3^{n}$ weight combinations (i.e a change of +1%, 0 or -1% for each weight) or in the case of two options, 9 weight combinations. The product of 9 weight combinations, a maximum path of 100 steps and 200,000 calculations to find the geometric mean for a each set of weights implies $1.8 * 10^{8}$ calculations.

After successfully writing code for two dimensions, I generalized the hill climber so that it works for any given dimension. The code is available at https://github.com/Ben-Tobben/Hill-Climber and further explained below. The hill climber significantly reduced the number of required calculations especially as n got large. The hill climber for n options using the worst case path would require $2 * 10^{7} * 3^{n}$ calculations (i.e. $3^{n}$ weight combinations, a maximum path of 100 steps and 200,000 calculations for each weight combination). This compares favorably to the brute force approach that requires $2 * 10^{(2n+5)}$

calculations as $3^n$ is smaller than $10^{(2n-2)}$ for all n > 1. In fact, for ten options, the ratio of reduction in calculations of the hill climber is more than the order of $10^{13}$ (i.e $3^{10} = 5.9 * 10^4$ divided by $10^{18}$) assuming the longest path. Despite these improvements, a little bit of testing revealed this was not enough to meet my sponsor's objective.

The formula for the number of calculations for the hill climber suggested the time the code would take to run would triple when another option is added, due to the number of directions (+1%, 0, -1%) considered at each point. This was confirmed by observed running times of the basic hill climber. Based on three options running in 20 seconds, four options running in one minute and five options running in three minutes, we predicted that running ten options would take close to twelve hours. There were three main areas that were explored to reduce the running time of the code:

A. Reducing the calculations performed in the geometric mean function - the 200,000 in our formula
B. Reducing the number of weight combinations that are considered at each point - the $3^n$ in our formula
C. reducing the number of steps the hill climber has to take before it terminates - the assumed 100 in our formula

Since the geometric mean function is the most expensive part of the code for n <=10 (i.e. 200,000 > $3^{10}$ = 59,049) and is called at the heart of the recursion, reducing its cost could speed up the code considerably. One of the ways that the function was improved the most was condensing the simulation data. The simulation had a number of identical outcomes. The sponsor confirmed that there should be a high number of identical outcomes as the probability of a loss for any of the options is low. We condensed the table by counting the number of times each unique row occurs. The modified geometric mean code sums the weighted outcomes as before, but then raises the result to the number of times that unique outcome occurs. The condensed table of simulation data only has 890 rows. Much of the time saved came from making this improvement. As a result instead of 200,000 calculations for the geometric mean, we only require 17,800 (i.e. (10 additions + 9 products) * 890 outcomes + 890 products). That is a reduction of more than 10x.

I also reduced the number of weight combinations (+1, 0, -1) calculated each time the recursive function is called. Checking a weight originally meant that the geometric mean function had to be called. We reduced the number of times geoMean is called for each call of the recursion using a couple of different checks.

- Each option has a maximum allocation (max_k) on a standalone basis. If the weight goes over the max k value for that option, then we know the returns will start going down. Directions where this happens are not considered.
- Since this is a hill climber, we decided that the array of directions to be considered should not sum to less than or equal to negative two. For example, if we have three options, which have weights of 20, 25, and 30, then 19, 24, 29 should not be considered. For a large number of options, every max geometric mean we've found has all of our cash allocated. There is no reason for the recursion to consider decreasing the amount of cash that is allocated by a significant amount when we are trying to allocate all of it. This modification was implemented in the main function where we built our table of offset values.
- I implemented a momentum flag based on the idea that if during the immediately prior iteration a weight for a given option increased then during the next iteration it would only increase or stay the same, but not decrease. Likewise if during the immediately prior iteration a weight for a given option decreased, then during the next iteration it would only decrease or stay the same, but not increase. This does not prevent us from going backward because if during the prior iteration a weight was unchanged, during the next iteration it could increase, stay the same or decrease.
- Weights which allocate more cash than we have, or that go below zero, are also ignored.

The benefit of most of these modifications are hard to estimate as they depend on the specific conditions of the hill climber. However, we can estimate the benefit of the second modification in the list above as it reduces the number of directions to be considered by 28% for ten options. If we assume +1, 0 and -1 are equally likely the momentum modification changes our $3^n$ to $2^{2n/3} * 3^{n/3}$. For 10 options, this is a benefit of a 93% reduction in directions considered ($2^{20/3} * 3^{10/3}$ = 3956 divided by $3^{10}$ = 59049). The benefit of the momentum modification is likely overstated due to the seeding techniques we outline in the next section especially if we have fully allocated all of our capital. Finally, these four tests are not all mutually exclusive and taking full credit for all of the benefits would lead to double counting.

The final area that I improved was the number of times the recursive function is called. By starting at a good spot (or having a good "seed"), the hill climber can reach its destination faster. One simple strategy is to start with all of the cash allocated. We know that given a large number (n>=4) of attractive options, all of the cash will be allocated. Starting with all the cash allocated means that not only are we closer, but all weights which increase the total amount of allocated cash will be ignored by our checks outlined in the section above. Since we want all of our cash allocated, a direction that sums to less than negative one is rarely going to be the best direction. We eliminated all these directions. Again, calculating the geometric mean as infrequently as possible speeds up the code, and removing the bottom portion of our directions is a substantial improvement. Starting on the edge and not checking weights that sum to less than negative one means that only 13 of 27 directions, of all possible directions, are considered in three dimensions. We spent a considerable amount of time working on heuristics to create great seeds. The sponsor would like to keep many of the seeding techniques proprietary, and so they will not be covered in this paper. However, using the heuristic's seed, with all the above improvements, the code runs in 56 seconds for 10 options. Without the seed, the code was running in about 3 minutes.

While the code we have developed achieves the sponsor's objectives, there are some changes that could improve the speed and accuracy of the hill climber. Currently the hill climber takes steps of exactly 1%. This means that when the code reaches a maximum, there could be better investments a step less than one away from that maximum. Another place where the step of 1% is a disadvantage is on the edge where all of our capital is invested. Since the premium is spent as well, there are times that the premium is an extra 4.8% cash we can spend. Since we only spend cash in 1% increments, we can't use that .8%. To improve the speed we think it is possible to modify the way the hill climber chooses its next position. By keeping track of the direction it is going, we could have the hill climber first check to see if continuing in the same direction still improves the geometric mean. If it does, then move to that point without checking any others. If it doesn't, then go back to the current routine of looking around and picking a new direction. Since the hill climber spends a lot of time "climbing a hill", I expect that the time saved by keeping the same direction and turning less frequently would be more efficient. The route to a max might be less direct, but it could be reached sooner since the geometric mean will be calculated many fewer times at each step. A final improvement could be bookkeeping. The hill climber occasionally looks at a weighting more than once, and keeping track of the weighted outcome for points we have previously been could reduce the running time.

I would like to thank my sponsor for this opportunity and plan to return over winter break to implement some of the additional enhancements outlined above.