Parallel Swap Optimization

Patrick Gartland, Ph.D., David Jackson

Abstract

We have developed an algorithm to optimize a parallel swap of two tokens. The algorithm is ideal for the Ref-Finance paradigm, with multiple liquidity pools per token pair in the same contract. For a given swap-in amount of Token A, the algorithm maximizes the swap-out value of Token B, taking into account the liquidity and swap fees of the various A-B liquidity pools. The solution is elegant, mathematically verifiable (using calculus and algebra), simple to implement, avoids "brute force" operations, and only requires a single call to the NEAR API to retrieve all the liquidity pools for minimal complicity or intent broadcasting.

Problem Statement

As stated in the Near Ref Finance Gitcoin bounty,

"In ref finance, for a given Token-A, if you want to swap for Token-B, there are always multiple pools with different liquidities and fees. You may want to use the one with max wrap-out amount. But as $x \times y = K$ shows us, any swap action would cause the swap rate to go in your unpleasant direction, especially if your swap amount is so big or the pool liquidity is so small.

So the smartest way is to split swap into several pools in order to get the max total return. Based on that, this bounty aims to find the best combination. The best combination has only one criteria, that is the best total swap rate or the max swap-out amount, in the given swap-in amount."

Assumptions

Given Assumptions

1. There are several pools for (Token-A, Token-B) pair with different liquidities and fees.
2. A user wants to swap A for B several times with different amounts. The task is to find the best combination for each swap.

Additional Assumptions

1. Currently, the algorithm assumes the pools follow the constant product ($x \times y = k$) model.
2. It is generally assumed that the multiple pools demonstrate (approximately) the same price of B/A. However, this assumption is not strictly necessary for our algorithm to work.
    - If multiple pools are found at different prices, it is assumed that trading and arbitrage opportunities would be undertaken to reach equilibrium before using this algorithm.
    - If the price difference is large, the simple solution would be to trade at the best price, rather than splitting the swap. The parallel swap would make sense with pools of the same price or at least close to the same price. Again, having comparable prices is *not* a requirement for our algorithm to work.
3. For each pool, the swap-in amount of Token A cannot be negative.
    - The math assumes a fee based on Δa_i which would assume you gain fees by taking out Token A. Forcing Δa_i to be non-negative prevents having to account for the fees in a more complicated way.
    - For pools with the same price of B/A, a negative Δa_i is not expected. This would only be expected to happen due to price differences.
    - If the algorithm would prescribe a negative value for Δa_i, then pool i is removed from consideration. (This would be the case for a worst-price pool.) Note, the process of removing a pool follows directly from the implementation of our Lagrange Multiplier solution in the presence of inequality constraints. In this case, the inequality constraint is that the allocation of Token-A for each pool must be non-negative.

Mathematical Solution

Since the task is to maximize ΔB_{total} as a function of ΔA_{total}, subject to the constraints,

$$\Delta A_{total} = \sum_i \Delta a_i$$
$$\Delta a_i \geq 0$$

this optimization problem is ideally solved with the method of Lagrange multipliers.

A single pool is characterized by three terms:

- a: the amount of Token A in the pool

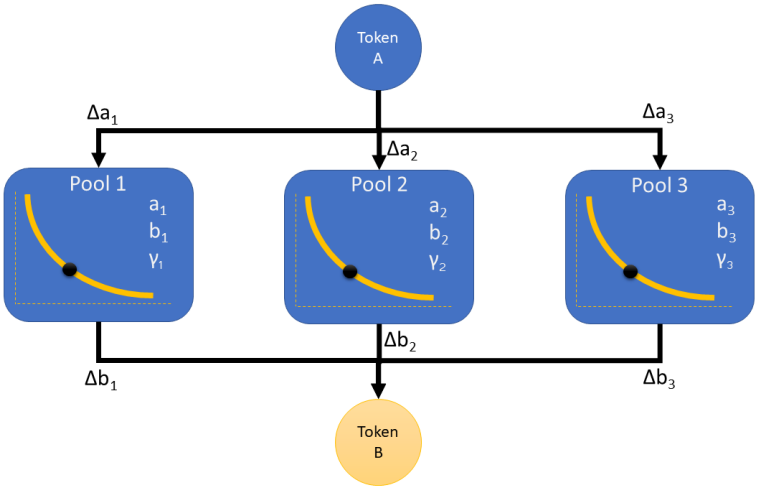

Figure 1: Illustration of the Parallel Swap Problem

- b: the amount of Token B in the pool
- ρ: the fee for trades in the pool (ie. 0.003 for a 0.3% fee)

Defining $\alpha \equiv \frac{\Delta a}{a}$ and $\gamma \equiv 1 - \rho$, we see that

$$\Delta b = \frac{\Delta a \cdot b \cdot \gamma}{a + \Delta a \cdot \gamma}$$

For multiple swaps in multiple pools, we would need to add the individual contributions to get the total.

$$\Delta B_{total} = \sum_i \Delta b_i$$

Lagrange Multipliers

We want to solve the equation

$$\lambda \nabla f = \nabla g$$

where

$$f(\Delta a_i) \equiv \Delta B_{total}$$
$$g(\Delta a_i) \equiv \sum_i \Delta a_i - \Delta A_{total} = 0$$

Note, we could also add an additional set of constraints and Lagrange multipliers per pool, along with slack variables, to account for the fact that the individual

components a_i must be greater than or equal to zero for all values of i, but this just leads to the equations that state that either the slack variable is zero, implying that the allocation to that pool should be zero, or the Lagrange Multiplier must be zero, implying that the constraint is inactive. What remains is the the equation above, which is separable, with no cross terms for different values of i. Therefore, we only need to solve

$$\lambda \cdot \frac{a_i \cdot b_i \cdot \gamma_i}{(a_i + \Delta a_i \cdot \gamma_i)^2} = 1$$

This reduces to

$$\mu \equiv \sqrt{\lambda} = \frac{\Delta A_{total} + \sum_i \left(\frac{a_i}{\gamma_i}\right)}{\sum_i \sqrt{\frac{a_i \cdot b_i}{\gamma_i}}}$$

leading to the solution for each pool:

$$\Delta a_i = \mu \sqrt{\frac{a_i \cdot b_i}{\gamma_i}} - \frac{a_i}{\gamma_i}$$

$$\Delta b_i = b_i - \frac{1}{\mu} \sqrt{\frac{a_i \cdot b_i}{\gamma_i}}$$

Also, we can recover the total ΔB_{total} output by summing over all pools, where below, b_{total} is the sum of reserves of Token-B across all pools.

$$\Delta B_{total} = b_{total} - \frac{\left(\sum_i \sqrt{\frac{a_i \cdot b_i}{\gamma_i}}\right)^2}{\Delta A_{total} + \sum_i \left(\frac{a_i}{\gamma_i}\right)}$$

Algorithm Overview

Here, we offer a flow chart summary of the algorithm implementation process. For a given set of pools and desired amount of input Token-A to trade in, you first calculate μ. Next, you can use μ to calculate the allocation of Token-A per pool, Δa_i. Next, check if any of the values of $\Delta a_i < 0$. If one exists, say, pool j, set the allocation to that pool, Δa_j, equal to zero, and remove that pool from the list of pools. Then, recalculate μ for the remaining pools. This will lead to a final output of the optimal values of allocations of Token-A, Δa_i, for each pool.

Verification

Because the math reduces to such a simple form, some special cases can be proven analytically, and tested against the algorithm. Below, some test cases are

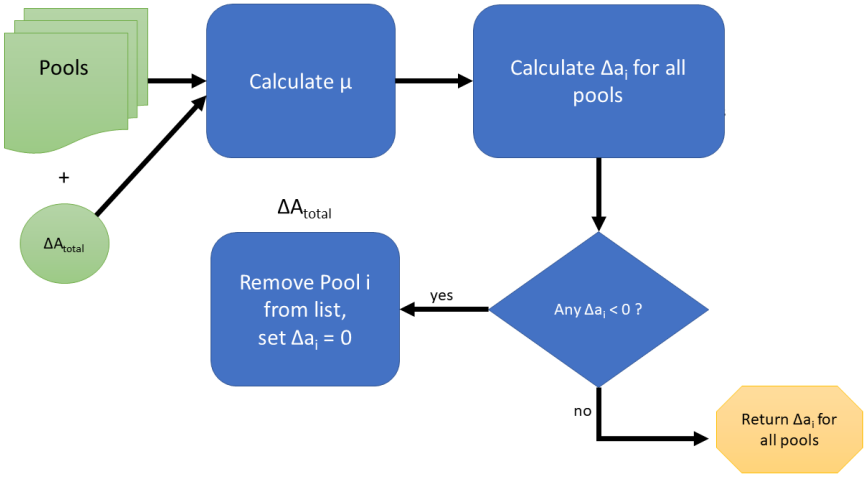

Figure 2: Algorithm Flow Chart

shown that have been verified with pencil and paper, as well as in the python and JS algorithm implementations.

Same Price, Liquidity, and Fees

Where two identical pools exist (same tokens, same price and liquidity, same fees), the best strategy is to split the trade in half. This solution also works for more than two pools. For n identical pools, each pool gets $\frac{1}{n}$ of the total ΔA.

Additionally, while our solution does not require a brute force search to find the optimal allocation of Token-A across all pools, we can nevertheless use an exhaustive brute force search in the form of Monte Carlo simulations to verify that our closed form solution agrees with that found by random exhaustive search. In the following figure, we ran 100,000 randomized Monte Carlo simulations per pool scenario and plotted the optimal allocation per pool. In the Monte Carlo simulations below, we assumed that the total input was 1.0 Token-A. For the following 3 figures, the price was identical in all constituent pools.

If Figure 3, we considered 4 identical pools, and found the expected result of 0.25 of the input of Token-A equally distributed among the 4 pools.

Different Liquidity

Where the price and fees are the same, slippage will be reduce by trading more of Token A in the pool with the larger liquidity. In fact, the trading between these pools is proportional to the square root of the liquidity. This also works with multiple pools.

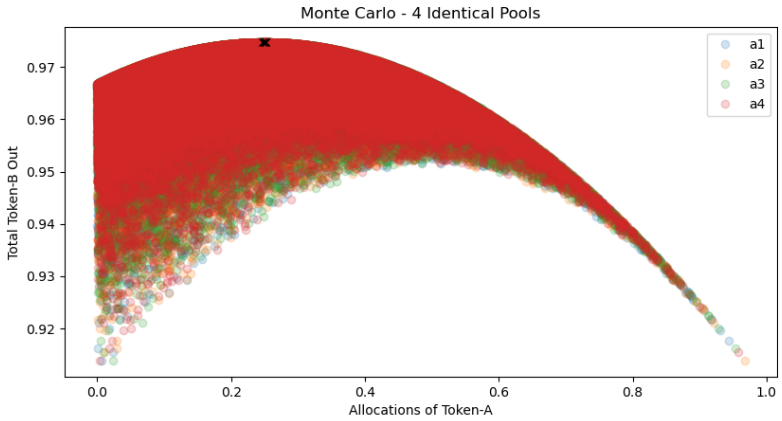

Figure 3: Monte Carlo Simulation Verification of Analytic Algorithm Solution

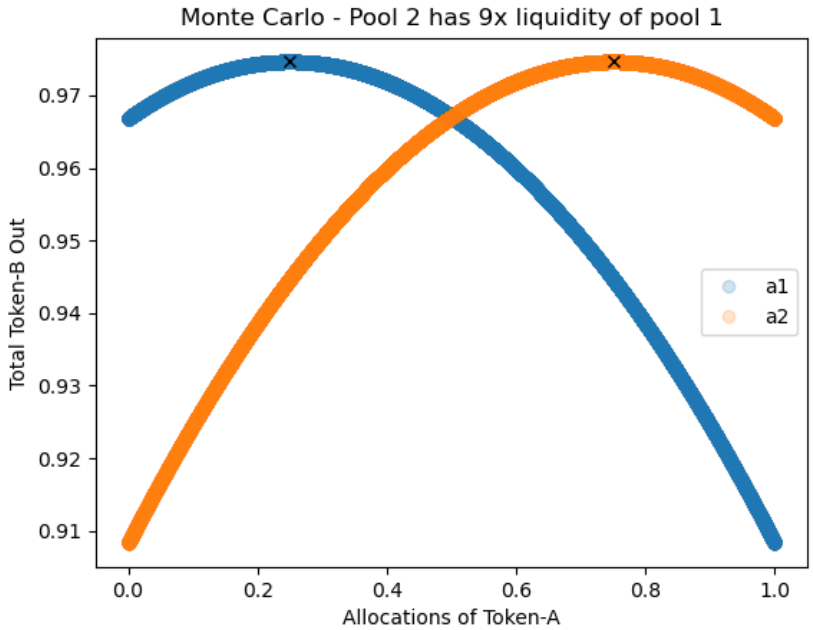

Figure 4: Monte Carlo Simulation Verification of Analytic Algorithm Solution

In Figure 4, we considered two different pools where the second pool had 9 times the liquidity of the first pool, and so it received an optimal 0.75 of the allocation of Token-A towards pool 2, and 0.25 allocation of Token-A to pool 1.

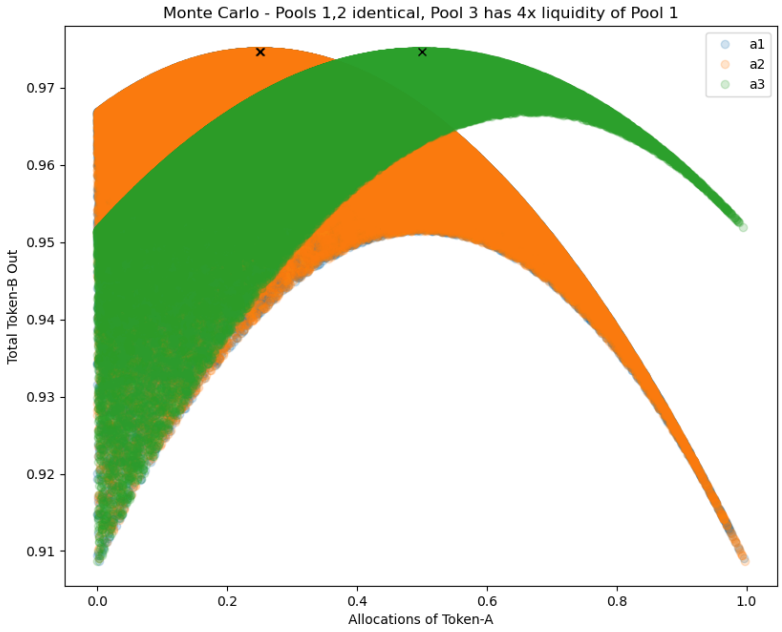

Figure 5: Monte Carlo Simulation Verification of Analytic Algorithm Solution

In figure 5, we considered 3 pools. The first two pools are identical, and the third pool has 4x the liquidity of the first pool. Therefore, we expect pool 3 to receive 0.5 of the allocation of Token-A, and pools 1 and 2 to receive 0.25 allocation of Token-A, each. Indeed, the results from our closed form solution optimization algorithm and the Monte Carlo results agree with our intuition of the expected result.

Figure 6 shows some example bonding curves for two different pools with the same price, but different liquidity.

Figure 7 shows the optimal solution for the corresponding bonding curves from Figure 6.

Different Fees or Different Price

Generally, if two pools have a different price or different fees for the same token, the best strategy is to only use the pool with the best price. The exception is if

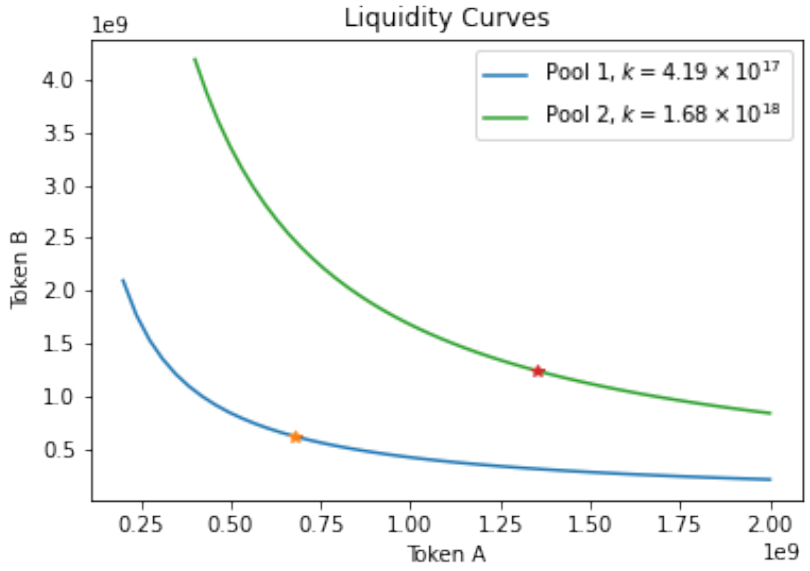

Figure 6: Bonding Curves for Identical price, different liquidity

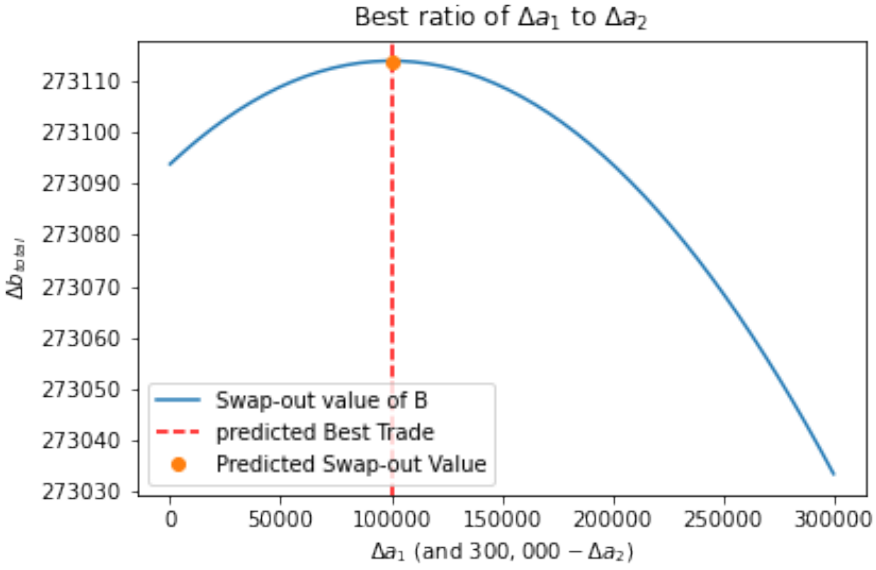

Figure 7: Optimal Solution

the fees and/or price difference is very small, and the trade is large enough that the slippage approaches the price/fee difference. If that happens, the trades will involve both pools.

This is the most general case; if the price or fees are not comparable for one pool, its delta will go to zero. The remaining pools with prices and fees that are close together will be used to calculate the Δa split.