Fast Arbitrage Detection

Claudio Raimondi

June 11, 2025

Abstract

Triangular arbitrage offers opportunities to exploit price inefficiencies between three currency pairs. While several algorithms exist for arbitrage detection, their computational efficiency and accuracy can vary. In this report, I present a method that combines established graph-based algorithms with **fixed-point** representation techniques to detect triangular arbitrage efficiently and accurately. The approach leverages the **Bellman-Ford algorithm** for cycle detection, enhanced by numerical optimizations to minimize floating-point errors. Experimental results demonstrate that this method improves detection speed and reliability, making it suitable for real-time financial applications.

Contents

1 Introduction

The process to exploit triangular arbitrage opportunities involves 3 generic steps, each crucial for ensuring the success of the strategy:

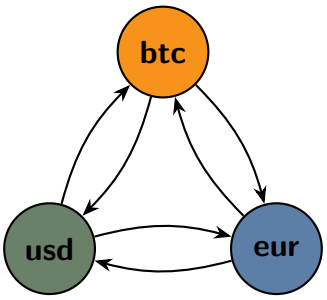
1. **Data Collection:** Gather real-time exchange rates for the currency pairs involved.
2. **Data Manipulation:** Use the collected data to detect profitable trades.

3. **Execution:** Execute trades to lock in profits.

For all three steps, the **speed** and **accuracy** of the algorithm are paramount. All the efforts become vain if somebody else collets, manipulates, and executes the arbitrage opportunity before you do. On this report, I will focus on the **second step**—*data manipulation*—, which is the most short-lived. The other two steps rely on external factors, such as networks and exchange APIs.

2 Representation

To visualize triangular arbitrage, we can represent it as a **directed graph** (more precisely a clique), where each node represents a currency and each edge represents an exchange rate.



3 Arbitrage Formula

3.1 Base Formula

The formula to detect a profitable triangular arbitrage path, given three exchange rates R, is the following inequality:

$$R_1 \cdot R_2 \cdot R_3 > 1 \tag{1}$$

3.2 Floating-Point Representation

In practice, each price is a floating-point number, and often times, if the *data collection* layer is serious about efficiency, they are provided as **mantissa** (m) and **exponent** (e) separately. So our inequality becomes:

$$(m_1 \cdot 10^{e_1}) \cdot (m_2 \cdot 10^{e_2}) \cdot (m_3 \cdot 10^{e_3}) > 1 \tag{2}$$

$$m_1 \cdot m_2 \cdot m_3 \cdot 10^{e_1+e_2+e_3} > 1 \tag{3}$$

$$m_1 \cdot m_2 \cdot m_3 > \frac{1}{10^{e_1+e_2+e_3}} \tag{4}$$

$$m_1 \cdot m_2 \cdot m_3 > 10^{-(e_1+e_2+e_3)} \tag{5}$$

3.3 Logarithmic Form

In most computer systems, where registers are limited to 64 bits, the product of three mantissas can easily overflow. To address this, we can use logarithmic properties to transform the **product** into an **addition** of much smaller numbers and the **power** into a **product**:

$$\log_b(m_1 \cdot m_2 \cdot m_3) > \log_b(10^{-(e_1+e_2+e_3)}) \tag{6}$$

$$\log_b(m_1) + \log_b(m_2) + \log_b(m_3) > -(e_1 + e_2 + e_3) \cdot \log_b(10) \tag{7}$$

3.4 Inverse Direction

As of now, we have only considered the **forward direction** of the triangular arbitrage, i.e. the forward cycle path. But since the rates are bidirectional, we can also exploit price inefficiencies in the **reverse direction**. Assuming we don't have the rates for the reverse direction, we can compute them as the inverse of the forward rates, and the formula becomes:

$$\frac{1}{R_1} \cdot \frac{1}{R_2} \cdot \frac{1}{R_3} > 1 \tag{1}$$

$$\frac{1}{m_1 \cdot 10^{e_1}} \cdot \frac{1}{m_2 \cdot 10^{e_2}} \cdot \frac{1}{m_3 \cdot 10^{e_3}} > 1 \tag{2}$$

$$\frac{1}{m_1 \cdot m_2 \cdot m_3 \cdot 10^{e_1+e_2+e_3}} > 1 \tag{3}$$

$$m_1 \cdot m_2 \cdot m_3 \cdot 10^{e_1+e_2+e_3} < 1 \tag{4}$$

$$m_1 \cdot m_2 \cdot m_3 < \frac{1}{10^{e_1+e_2+e_3}} \tag{5}$$

$$m_1 \cdot m_2 \cdot m_3 < 10^{-(e_1+e_2+e_3)} \tag{6}$$

$$\log_b(m_1 \cdot m_2 \cdot m_3) < \log_b(10^{-(e_1+e_2+e_3)}) \tag{7}$$

$$\log_b(m_1) + \log_b(m_2) + \log_b(m_3) < -(e_1 + e_2 + e_3) \cdot \log_b(10) \tag{8}$$

4 Algorithm

4.1 The Spread Problem

Once the formula is derived, the algorithm is just a matter of applying the two inequalities. From a first analysis, it seems that the only difference between the forward and reverse inequalities is their direction. However, we have to account for the **spread**, which is the difference between the bid and ask prices. This means that we have to consider the **ask** price for the forward direction and the **bid** price for the reverse direction.

4.2 The choice of the base

Up to this point, we have considered the base b of the logarithm to be arbitrary. But in reality things are more complicated:

- A larger base b will result in a smaller logarithm, which can lead to an increase of precision[1].
- $\log_2$ is often faster to compute on most hardware, thanks to bitwise operations[2].
- $\log_{10}$ allows for a simplification of the formula: $\log_{10}(10) = 1$.

//TODO argument the choice of the base2

4.3 Precomputation

Assuming exponents change very rarely, we can use an optimistic approach and **precompute** the entire right side of the inequality, storing it in a constant variable, raising an exception or even updating the precomputed constant only when the exponents change.

4.4 The Representation of real numbers

The logarithms produce real numbers. To address this in an effient manner, we must avoid floating-point numbers once again. This leaves us with two options:

1. `log2(integer x)` $\rightarrow$ `FixedPoint`
2. `log2(integer x)` $\rightarrow$ `[Mantissa, Exponent]`

While both are great alternatives, the latter is preferable because we can choose the base of the returned floating-point representation. Furthermore, if we choose the same base as the logarithm, there is a lot of room for optimization.

4.5 The Actual Computation

Combining all the previous steps, we can derive the final steps to compute the left side of the inequality:

$$\log_2(m_1) + \log_2(m_2) + \log_2(m_3) \tag{1}$$

$$m'_1 \cdot 2^{e'_1} + m'_2 \cdot 2^{e'_2} + m'_3 \cdot 2^{e'_3} \tag{2}$$

computing the maximum exponent $e'_{\max}$ for normalization:

$$(m'_1 \cdot 2^{e'_1 - e'_{\max}}) \cdot 2^{e'_{\max}} + (m'_2 \cdot 2^{e'_2 - e'_{\max}}) \cdot 2^{e'_{\max}} + (m'_3 \cdot 2^{e'_3 - e'_{\max}}) \cdot 2^{e'_{\max}} \tag{3}$$

$$2^{e'_{\max}} \cdot (m'_1 \cdot 2^{e'_1 - e'_{\max}} + m'_2 \cdot 2^{e'_2 - e'_{\max}} + m'_3 \cdot 2^{e'_3 - e'_{\max}}) \tag{4}$$

rewriting the powers of two as bitshifts:

$$(m'_1 \ll (e'_1 - e'_{\max}) + m'_2 \ll (e'_2 - e'_{\max}) + m'_3 \ll (e'_3 - e'_{\max})) \ll e'_{\max} \tag{5}$$

5 Bellman-Ford Algorithm

6 Relative Error

7 Optimization

References

[1]

[2]