# Forex Arbitrage with Bellman-Ford and Cycle Caching

High Frequency Trading Technologies
May 2025
Walker Bagley, Varun Taneja, Eamon Tracey

## 1. Background

### Foreign Exchange Overview

The foreign exchange (FX or forex) market is a decentralized network of banks and financial institutions trading pairs of currencies. This market trades 24 hours per day and five days per week across four sessions, New York City, London, Tokyo and Sydney. Rather than trading assets at one centralized exchange, each member of this network offers their own bid and ask spreads for various currency pairs.

Currency pairs are listed as `[base currency]/[quote currency]`. Bid and ask prices are listed in terms of the quote currency. Buying a currency pair converts the quote currency to the base currency; selling is the inverse transaction. For example, selling EUR/USD at a price of $0.95 means converting 0.95 Euros to 1 US Dollar. This means we can define the exchange rate from EUR to USD as the market bid price since that is the price at which we can sell. The opposite exchange rate equals either the bid price of USD/EUR or the multiplicative inverse of the ask price of EUR/USD.

The goal of our project is to find and analyze cyclic arbitrage opportunities in the forex market. An arbitrage opportunity is a cycle of currency pairs that we can trade back to the initial currency and make a per-unit profit. For example, suppose we can sell USD/EUR for 0.9, EUR/GBP for 0.72, and GBP/USD for 1.4. Then, we could make a small profit as illustrated in *Figure 1*.

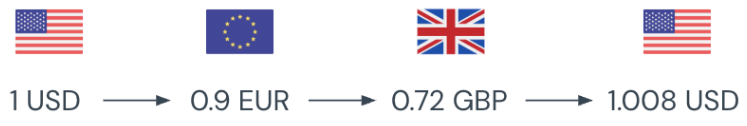

*Figure 1. Example Arbitrage Cycle between USD, Euros and Pounds*

### Bellman-Ford

We sought to analyze the arbitrage opportunities that may arise when provided with quotes for several currency pairs over time. To do this, we construct a weighted, directed graph with each currency as a node. The edges represent the exchange rates, or in our case, the bid prices for each currency pair. Then, if we have a cycle in the graph, we can compute the product of the cycle's edge weights to find the profit of that cycle on a per unit basis. If this profit exceeds one, then the cycle is considered a potential arbitrage and in an ideal world we would fill several exchanges to capture the profit. This is also why we only consider selling each currency pair, so that in a successful arbitrage, a trader would end up completely neutral except for their base currency. That being said, in an arbitrage cycle, the base currency can be any currency involved in the cycle as the profit is independent of the entry and exit. Our initial attempt to find these arbitrage opportunities used the Bellman-Ford algorithm for detecting negative cycles.

Bellman-Ford is a single-source-shortest-path algorithm that not only detects negative cycles, but also works with negative edge weights, unlike Dijkstra's, another shortest path algorithm. Negative cycles are important in the context of forex arbitrage as we can reduce our arbitrage condition to finding a negative cycle (*Figure 2*).

$$\begin{aligned} e_1 \cdot e_2 \cdots e_n &> 1 \\ \frac{1}{e_1} \cdot \frac{1}{e_2} \cdots \frac{1}{e_n} &< 1 \\ -\log(e_1) - \log(e_2) - \cdots - \log(e_n) &< 0 \end{aligned}$$

*Figure 2. Reduction of Arbitrage Condition to Finding Negative Cycle.*

Our implementation relies on a graph of V currency nodes and their respective exchange rates, however we add a source node with edges of weight zero connecting it to each currency. This allows us to be non-deterministic in selecting a starting node when running our implementation of the algorithm. We keep two arrays, distance and predecessor, which store the distance from our source to each node and the nearest predecessor to each node, respectively. An example of the Bellman-Ford algorithm can be seen in *Figure 3*.

The algorithm then executes the following steps:

1. Iterate over all edges between currencies V - 1 times. An edge corresponds to a currency pair ($c_1$, $c_2$).
    a. If distance[$c_1$] + weight($c_1$, $c_2$) < distance[$c_2$] then we update distance[$c_2$] and predecessor[$c_2$].
    b. Performing this V - 1 times ensures that all edges are fully relaxed.
2. Perform the prior step with one more iteration and if any edge relaxes, there must be a negative cycle, which we can reconstruct using the predecessor array.

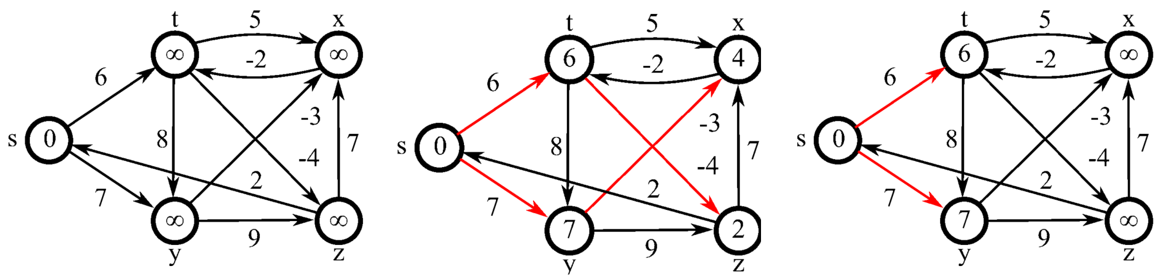

*Figure 3. Example Steps of the Bellman-Ford Algorithm*

While it is a useful tool in finding potential arbitrage opportunities, Bellman-Ford is not perfect. It does not find all possible negative cycles but rather one per execution. This is also an issue since finding all possible negative cycles is required in order to determine the most profitable opportunity in any given state of the graph. In fact, this problem can be reduced to finding a Hamiltonian cycle, which is NP-complete.

**Cycle Caching**

Another implementation we examined after conducting analysis on the arbitrage cycles found by the Bellman-Ford algorithm was caching cycles and tracking only a specific subset of all possible cycles for arbitrage opportunities. Rather than running a fairly intense $O(n^3)$ algorithm on the graph every time it updates, which in the trading world is completely unfeasible, we store arrays of cycles along with thresholds that would make each cycle profitable. The initial profit of a cycle is calculated by multiplying the first exchange rates for each pair in the cycle. Then, we can update a profit given a new exchange rate for a particular currency pair by multiplying the profit by the new rate and dividing by the old rate. Equivalently, a cycle will be profitable if the ratio of the new rate to old rate exceeds the inverse of the current profit. We can call this the threshold and use it to generate a table for an FPGA to use (more discussed later).

By caching cycles, we can linearly check each cycle we care about to quickly determine whether an arbitrage opportunity exists as well as determine amongst the cached cycles which is most profitable at the moment. This breaks down in detecting whether or not there exists an arbitrage opportunity amongst the entire graph of currencies as we reduce our search set to some preselected set of potential cycles. That, however, is the reason this is much more efficient with larger graphs. If we can determine some of the most frequent and profitable opportunities and cache them, we can greatly enhance the speed at which we are able to trade on those opportunities. For example, with ten currencies, there are over a million possible cycles we could encounter, but if we only consider cycles that trade through five or fewer currencies, this set is reduced to just over 7,600 cycles, roughly 150 times smaller and thus faster.

To count the number of cycles of length k given n currencies, we want the number of ways to select k currencies from n options. This is not enough, because the cycle USD/GBP/EUR is not the same as USD/EUR/GBP. In fact, there are k! ways to order the k currencies. Thus we arrive at another problem, because the cycle USD/EUR/GBP is actually the same as the cycle EUR/GBP/USD. We can think of any cycle that is rotated to start with a different currency, but all currencies follow in the same order to be the same as the original cycle, and there are k ways to rotate a k length cycle. With some simple math we see that the number of cycles of length k given n currencies is nCk * k! / k = nCk * (k - 1)!. *Figure 4* shows the number of unique cycles of various lengths based on the number of currencies.

| | Cycle Length | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Currencies** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **Total** |
| **3** | 3 | 2 | | | | | | | | 5 |
| **4** | 6 | 8 | 6 | | | | | | | 20 |
| **5** | 10 | 20 | 30 | 24 | | | | | | 84 |
| **6** | 15 | 40 | 90 | 144 | 120 | | | | | 409 |
| **7** | 21 | 70 | 210 | 504 | 840 | 720 | | | | 2,365 |
| **8** | 28 | 112 | 420 | 1,344 | 3,360 | 5,760 | 5,040 | | | 16,064 |
| **9** | 36 | 168 | 756 | 3,024 | 10,080 | 25,920 | 45,360 | 40,320 | | 125,664 |
| **10** | 45 | 240 | 1,260 | 6,048 | 25,200 | 86,400 | 226,800 | 403,200 | 362,880 | 1,112,073 |

*Figure 4. Number of Unique Cycles by Number of Currencies and Cycle Length*

## 2. Dataset

Polygon.io generously provided free access to their historical FX tick data, sourced from multiple financial institutions. This process consisted of individually downloading eighty days worth of tick data from January 1st to March 31st. In total, we ended up with roughly two billion individual quotes and 129 unique currencies with a one-second timestamp resolution. Unfortunately with our low resolution data, it is impossible to determine how long any one individual arbitrage opportunity lasts and thus we cannot say how quickly we would need to trade in order to take advantage of such an opportunity. Still, we benchmarked our arbitrage algorithms.

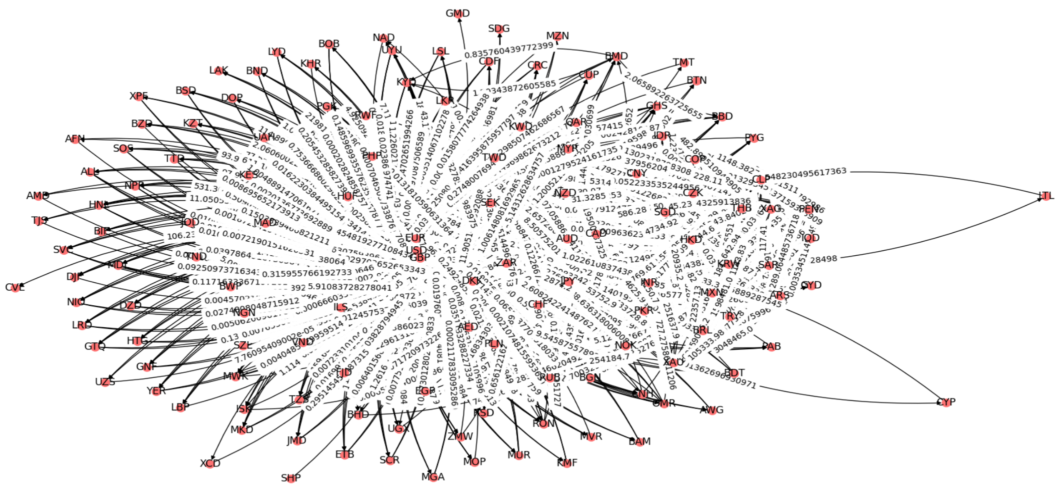

*Figure 5. Directed Graph with 129 Unique Currencies*

To keep our analysis tractable, we retained only the ten most liquid currencies: USD, EUR, JPY, GBP, CNH, AUD, CAD, CHF, HKD, and SGD. Upon isolating these ten currencies, we sorted the data by timestamp to provide a time series of the quotes. A sample view of our currency graph at a single timestamp can be seen in *Figure 6.* The data includes quotes from three different banks, however no information was provided by Polygon.io on how to distinguish between quotes from each bank, so we removed any duplicate quotes from each timestamp by keeping only the last quote. We saved the filtered data and used it as a stream for the arbitrage scripts.

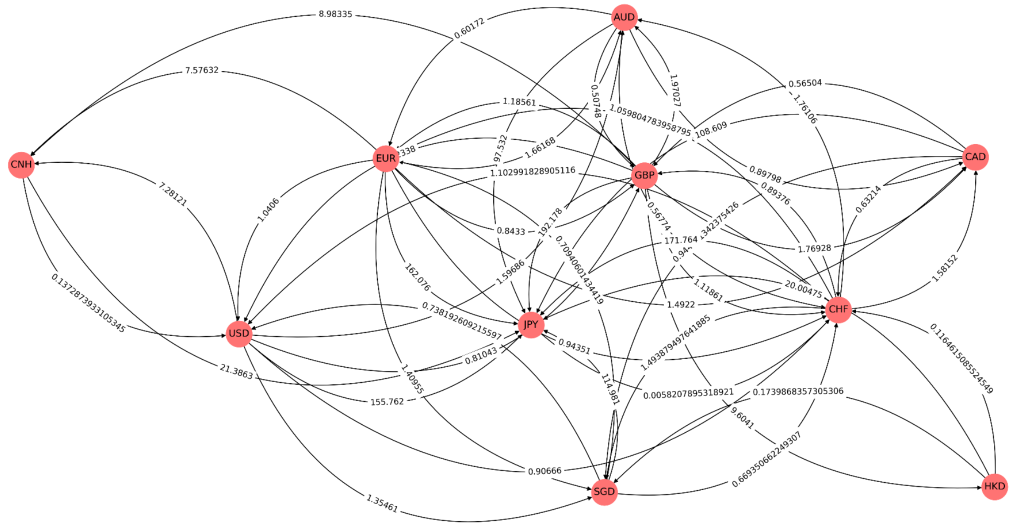

*Figure 6. Directed Graph with the Ten Retained Currencies*

## 3. Trade Implementation

### Bellman-Ford

Code

- ➔ forex_arbitrage.cc
- ➔ forex_arbitrage.h

The Bellman-Ford implementation involves a class named `ForexArbitrage` with 3 primary methods: `Update`, `IsArbitragePossible`, `FindArbitrageOpportunity`. The `ForexArbitrage` object is initialized with a vector of currencies represented as strings. These string currencies are mapped to numeric IDs. The class stores a `double **graph_` which stores the graph connecting currencies with weights equal to the negative logarithm of the corresponding exchange rates.

The `Update` method simply updates the graph using a new exchange rate for a given currency pair.

The `IsArbitragePossible` method contains the Bellman-Ford implementation. Since the Bellman-Ford algorithm must begin with a source node, a sentinel node with weight 0 to every other node (currency) is used as the source node. First, the `distance` array is initialized with values of infinity. Then, the edge relaxation process updates these distances. The edges are relaxed C times where C equals the number of currencies, aligning with the Bellman-Ford algorithm. If no edge relaxes in any of the C relaxation attempts, the function immediately returns that there is no arbitrage opportunity available. Then, the function attempts to relax the edges one more time. If an edge relaxes, there must be a negative cycle, so the function immediately returns true; otherwise, the function returns false.

The `FindArbitrageOpportunity` method contains the same Bellman-Ford implementation extended to reconstruct the negative cycle (the arbitrage cycle). If a negative cycle is detected, the function iterates backwards using the `predecessor` array to find a node inside the negative cycle. Then, the function iterates forward again to reconstruct the cycle in the correct order. When the function completes iterating through the entire cycle, the function returns the detected arbitrage cycle and profit. Note that this function will always return the first negative cycle that it finds. This means that if there exists more than one arbitrage cycle, it will ignore all the others, whether or not they are more profitable. Further, this means the found arbitrage cycles are inadvertently tied to the order in which the currencies are passed to the `ForexArbitrage` object.

**Cycle Caching**

Code

- ➔ casche_arbitrage.cc
- ➔ casche_arbitrage.h

The Cycle Caching implementation involves a class named `CascheArbitrage` with 4 primary methods: `Update`, `NewArbitrage`, `FindAllArbitrageCycles`, `GenerateTable`. The `CascheArbitrage` object is initialized with a vector of cycles. These cycles correspond to the cycles of currency pairs to track. Only these cycles will be analyzed for arbitrage opportunities. Further, all these cycles and all the contained currency pairs are mapped from the string forms to numeric IDs. Internally, the class stores the map `pair_to_cycles_` to track the cycles to which a given pair belongs. Also, the class stores the map `cycle_to_pairs_` to track which pairs (in order) belong to a given cycle. Further, the class stores `pair_to_rate_` mapping a currency pair to the exchange rate and `cycle_to_profit_` mapping a cycle to its profit if traded. A profit greater than 1 indicates an arbitrage opportunity. All four aforementioned maps are `std::vector`'s where the index corresponds to the numeric ID of the currency pair or exchange. Using integer indexing rather than string hashing+indexing improves efficiency.

The `Update` function updates the `pair_to_rate_` and `cycle_to_profit_` maps given a new exchange rate for some currency pair. Precomputing these maps means it can be very fast to find an arbitrage cycle for a hypothetical new market data message. The update function is much slower than `NewArbitrage` since it must update all cycles with the given currency pair.

The `NewArbitrage` function finds an arbitrage opportunity given a new exchange rate for a given currency pair. The function notably does not update the internal data structures; rather it returns whether an arbitrage opportunity exists if the new data were to be applied. The function calculates `new_old_ratio = new_rate / old_rate`. Then, for any given cycle, if `profit * new_old_ratio > 1`, then there must be an arbitrage opportunity. The function will also return the detected cycle with the arbitrage opportunity. The function has an O(n) time complexity where n is the number of cycles to which the given currency pair belongs.

The `FindAllArbitrages` function performs what it says: finds all arbitrage cycles currently present. The function loops through the `cycle_to_profit_` map and appends all profitable cycles (profit > 1) to the resultant vector.

The `GenerateTable` function outputs a table containing currency pairs, thresholds, and cycles. In each row, if a new exchange rate for the currency pair exceeds the threshold, then the cycle should be traded immediately to make profit. This enables extremely fast trading via market data triggers. If an FPGA stores the table, then it can immediately trade when new market data offers an arbitrage opportunity. The table always contains all currency pairs with valid exchange rate data; however, the function is easily modifiable to favor the most profitable opportunities. An example table is shown below in *Figure 7*. The table indicates that if the CNH to USD exchange rate increases to 0.14, then there exists an arbitrage opportunity in the CNH → USD → EUR → CNH cycle.

| **Currency Pair** | **Threshold** | **Cycle** |
|---|---|---|
| USD/EUR | 0.9569011712470337 | EUR/USD/EUR |
| EUR/CNH | 7.637117789870582 | CNH/USD/EUR/CNH |
| JPY/GBP | 0.005182136939371882 | CNH/USD/JPY/GBP/CNH |
| CNH/JPY | 21.40516036175152 | JPY/USD/CNH/JPY |
| EUR/JPY | 162.10082671421625 | JPY/EUR/JPY |
| CNH/USD | 0.13750954091127995 | CNH/USD/EUR/CNH |
| EUR/USD | 1.0450958875476826 | EUR/USD/EUR |
| GBP/EUR | 1.1985249529600817 | GBP/EUR/JPY/GBP |

*Figure 7. Table of Arbitrage Pairs, Thresholds, and Cycles*

## 4. Benchmarking

Code

- ➔ benchmark_forex.cc
- ➔ benchmark_casche.cc

Though we did not have access to intra-second data to determine whether our algorithms would be capable of successfully entering and exiting the market in an arbitrage opportunity, we could benchmark how fast our algorithms perform arbitrage detection. We benchmarked how fast `ForexArbitrage` could detect and find an arbitrage opportunity split by whether or not an arbitrage opportunity existed. Further, we benchmarked how fast `CascheArbitrage` could find a singular arbitrage opportunity and find all present arbitrage opportunities. For each case, we varied the number of currencies that were tracked. More currencies generally implies a longer time to find arbitrage opportunities. Note that the code for benchmarks was compiled with the `-O3` clang compilation flag. The exact benchmarks are included in *Figure 8* and *Figure 9*.

| | time to detect no arb (ns) | time to detect arb (ns) | time to find no arb (ns) | time to find arb (ns) |
|---|---|---|---|---|
| **5 currencies** | 127 | 281 | 127 | 319 |
| **10 currencies** | 404 | 1,341 | 418 | 1,412 |
| **30 currencies** | 5,951 | 22,617 | 6,160 | 22,855 |

*Figure 8. Bellman-Ford Algorithm (ForexArbitrage) Benchmarks*

| | # of cycles | time to find no arb (ns) | time to find arb (ns) | time to find all arbs (ns) |
|---|---|---|---|---|
| **5 currencies** | 84 | 18 | 23 | 78 |
| **10 currencies** | 7593 | 279 | 115 | 4,869 |
| **15 currencies** | 81,277 | 1,580 | 610 | 51,104 |

*Figure 9. Caching Algorithm (CascheArbitrage) Benchmarks*

The above benchmarks show that `CascheArbitrage` is much faster than `ForexArbitrage`. This aligns with our expectations since `CascheArbitrage` trades space for time complexity gains. Tracking 5 currencies, our algorithm was able to detect an arbitrage opportunity in less than 25 nanoseconds (on average over all Polygon market data); we expect this benchmark to be competitive in real-world application, especially when paired with a market-data-triggered FPGA. Note that `CascheArbitrage` takes longer to find that there were no arbitrage opportunities than to find an arbitrage opportunity. This happens because the class must linearly search all relevant cycles before determining no arbitrage is available rather than exiting early when an arbitrage is found. We are quite happy with the efficiency and speed of the `CascheArbitrage` detection/search algorithm.

## 5. Analysis

Upon gathering data for all the arbitrage opportunities detected by both our Bellman-Ford and cycle caching implementations, we explored the opportunities we found, the profits they generated and how often they occurred. We discuss in more detail how we computed these metrics and what they mean for forex arbitrage.

**Bellman-Ford Arbitrage Density**

We wanted to figure out how many arbitrage opportunities we found in relation to the amount of data, so we computed the ratio of the number of opportunities per unique timestamp for each day, giving us an indicator of what proportion of each day we would be able to arbitrage (*Figure 10*). Interestingly, only six days in our dataset had densities lower than 20%, while about half had densities above 50% and even another six with densities above 80%. This indicates that there are in fact a reasonable amount of arbitrage opportunities to take advantage of given our dataset, though not all might be profitable after accounting for fees and order book volume.

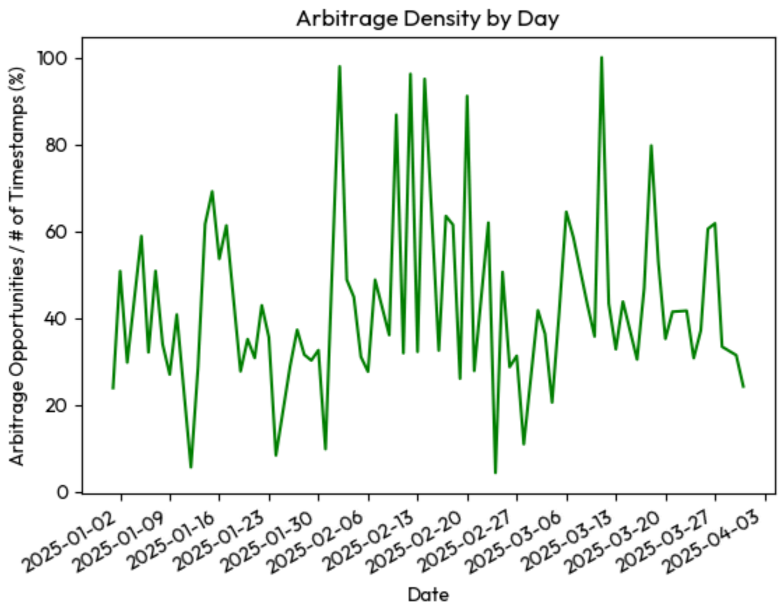

*Figure 10. Percentage of Timestamps with Arbitrage by Day*

**Bellman-Ford Arbitrage Volume**

We also visualized what days had the highest volume at higher profits to give us a sense of what days had more mispriced exchange rates. We bucketed arbitrage opportunities by five profit ranges and counted them for each individual involved currency. Then, we plotted this data over the course of the entire time series. In *Figure 11*, larger points higher in the graph indicate a higher volume of more profitable arbitrage opportunities while large points towards the bottom of the graph indicate there is a large volume of opportunities that would likely become unprofitable depending on the fee structure.

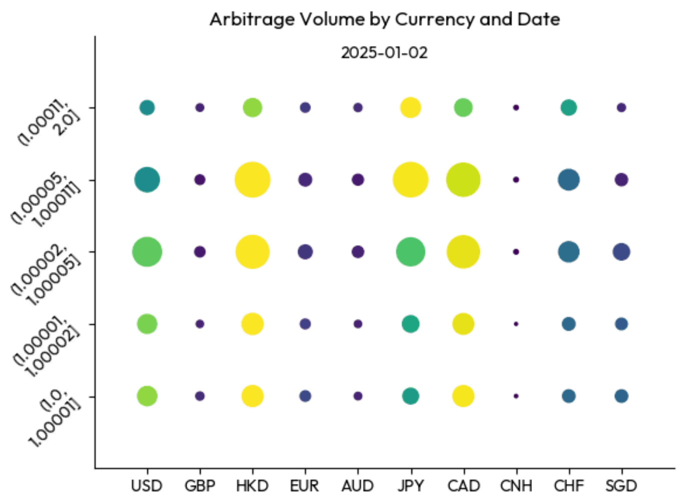

*Figure 11. Volume of Arbitrage Currencies Grouped by Profit by Day*

**Bellman-Ford Profit by Currency**

We were curious as to what currencies generally provided higher average profits when they are involved in an arbitrage opportunity, so we computed the average profits when grouped by currency. The Chinese Yuan showed the highest, with the British Pound a little ways behind, and the rest of the currencies somewhat even across the board, as seen in *Figure 12*.

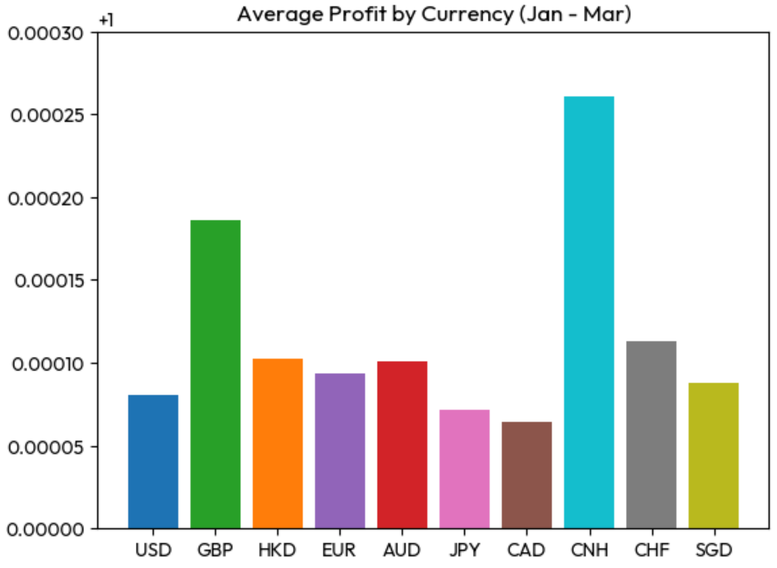

*Figure 12. Average Arbitrage Profit by Involved Currency*

## Arbitrage Opportunities and Profit by Commission

Forex trading, at least for retail investors, is broken into two accounts: commission-based raw spread accounts and commission-free spread accounts. The difference is that on raw spread accounts, the brokers are trying to offer the tightest spread possible, and in turn charging a commission for maintaining such tight spreads whereas commission free accounts make the brokers money on their spreads, which is why they are wider. Since we have been using the best bid and selling through cycles, it makes more sense to compare our strategies to the raw spread accounts, which charge fees based on the volume traded. These fees range from $2-10 per $100,000 traded and we can normalize this to a per unit rate.

We adjusted all of our arbitrage profits to take this into account for a number of different fees, remembering that in trading a cycle of length five, we would be losing five times the normalized fee rate since there are five different trades to execute. We also computed the average fee-adjusted profits by month to plot against the number of profitable opportunities remaining. In *Figure 13*, we can see an inverse correlation between average profits and number of opportunities due to the high volume of low profit opportunities that skew the average profit being removed by fee adjustment. The higher profit opportunities are less concentrated and thus become a much larger factor in the mean with a reduction in data points.

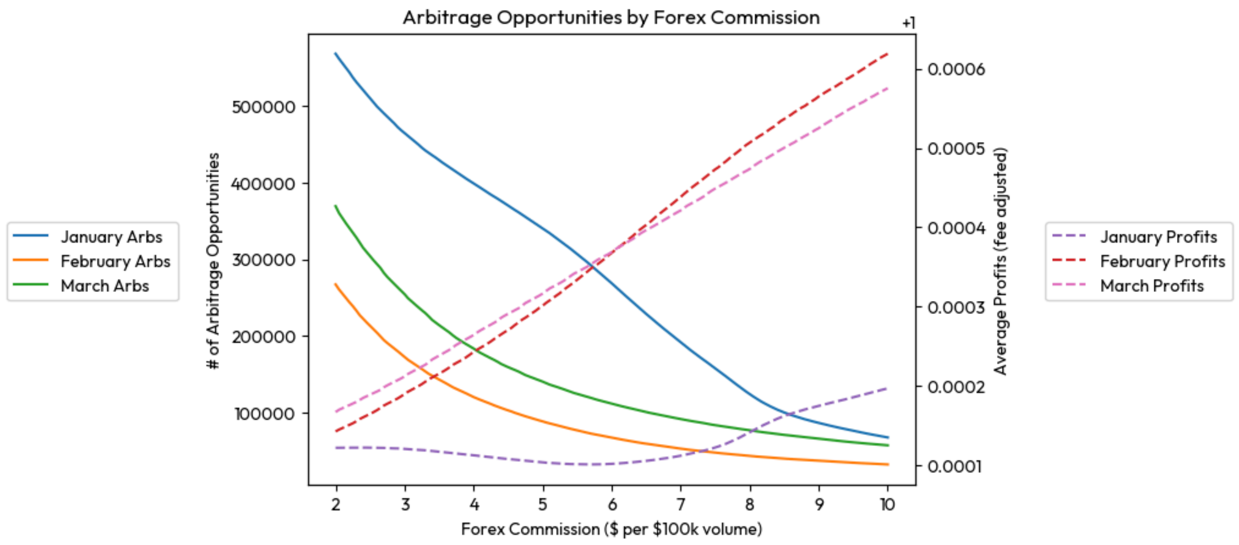

*Figure 13. Arbitrage Opportunities and Average Profit by Forex Commission*

We ran a similar analysis on the opportunities collected by the caching algorithm, which included all arbitrage opportunities trading through five or fewer currencies for our ten selected currencies. Across the three months of data, we saw a slightly concave up curve with regards to fees, as seen in *Figure 14*. That being said, with the highest fee of $10 per lot, there were still over ten million profitable arbitrage opportunities to take advantage of over this three month period.

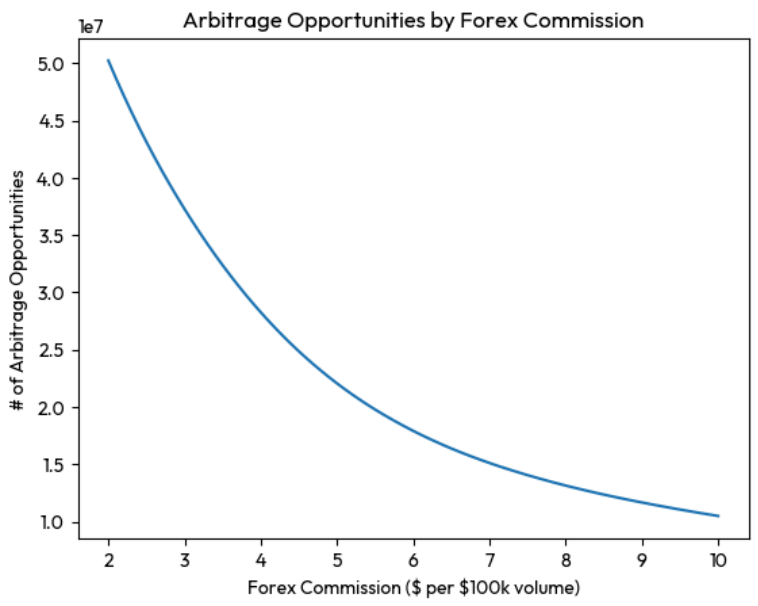

*Figure 14. All Arbitrage Opportunities by Forex Commission*

**Caching Algorithm Profits**

Upon finding over 107 million total arbitrage opportunities in our dataset with the selected ten currencies and cycles of length five or smaller, we plotted the average net profit on a logarithmic scale with respect to the date. In *Figure 15*, the orange line indicates the mean profit across opportunities from that particular day and there seem to be several spikes in mean profit at the start of each week, indicating higher forex market volatility after a weekend of settling. This graph excludes a small subset of arbitrage opportunities in late January that exceeded a profit of 1.02 which we considered to be outliers found by our algorithm. A couple of these opportunities are discussed below as they were our most profitable, however, they do not reflect the general trend of arbitrage opportunities we found in the dataset.

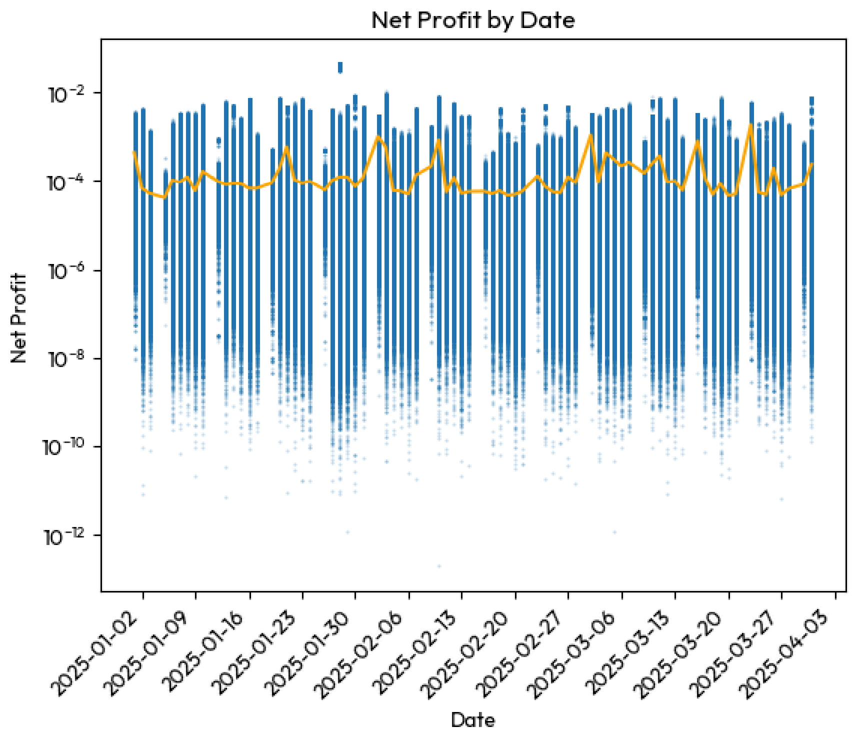

*Figure 15. Net Profit by Date Excluding Outliers*

**Most Profitable Opportunities**

We looked through all the arbitrage opportunities found by our Bellman-Ford implementation within the entire dataset to identify the most profitable opportunity. This opportunity came on January 28, 2025, a rather volatile day in the forex market. The cycle traded GBP to CHF to HKD back to GBP (as seen in *Figure 16*) with a profit of 1.043642. Four pence on the pound may not sound like much, but based on our findings in forex arbitrage, it is incredibly high, especially if the market happened to support some decent volume through this arbitrage cycle at this moment in time. In fact, when compared to the rest of the arbitrage opportunities detected by our Bellman-Ford implementation, it was roughly 445 times more profitable than the mean and about 990 times more profitable than the median arbitrage opportunities.

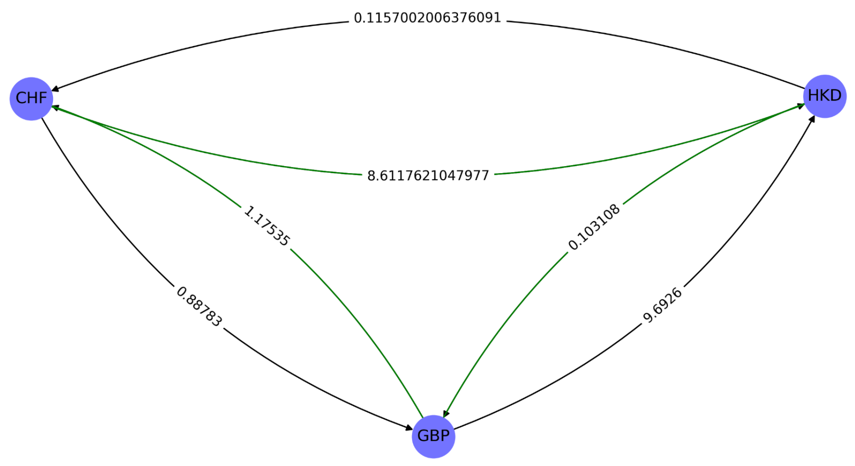

*Figure 16. Most Profitable Bellman-Ford Arbitrage Cycle*

After analyzing over 107 million arbitrage opportunities that occurred over this 3 month period using our caching algorithm, the most profitable opportunity we found was a superset of the most profitable opportunity found by our Bellman-Ford implementation. It traded USD for GBP for CHF for HKD and back to USD for a profit multiplier of just over 1.044237, about six hundredths of a unit better than the other cycle with only the addition of trading through the Hong Kong Dollar as well. This is intriguing because while we are only considering the non-adjusted profit in this analysis, trading through an additional currency for an extra 0.0006 profit makes no sense when fees and the increased risk of getting stuck with a non-neutral position are accounted for.

**Bellman-Ford Trading Simulation**

Out of curiosity, we wanted to see how much these arbitrage cycles would have made us in one day of our quote data. We selected an arbitrary day, March 14 to conduct this test with a few key assumptions. First, we assume in this scenario that the market could support trading $100,000 worth of each currency through the best bid quotes, which is almost certainly not the case in the real world. Second, we assume that our firm has the capital to make $100,000 trades an incredibly high number of times each day, which in reality might not necessarily be the case for a forex strategy, especially depending on how long trades take to clear. Third, we assume there are no transaction fees involved in our forex market, when in reality these would cut the number of profitable opportunities roughly five fold.

That being said, we calculated two metrics. First, a constant investment of $100,000 into each of the 29,080 arbitrage opportunities found that day which yielded $127,069 net profit by the end of the day. Second, if we compounded the arbitrages we executed over the course of the day at each timestamp, starting with a $100,000 on the first arbitrage, we would have expected to net

$256,309 in profit. These metrics actually mean very little in terms of real world performance, though they are fun to speculate about.

**Liberation (Tariffs) Day**

While working on this project (on April 2), President Trump announced “Liberation Day” tariffs; large tariffs were imposed on almost all countries with which the United States trades. We suspected this may induce increased volatility and mispricing in the forex market. In the January through March 2025 timespan, we observed an average of 19.6 total arbitrage opportunities per second (multi-counting the same arbitrage over multiple seconds). On April 2, this metric jumped to 52.3, confirming our suspicion. On this day, the most commonly arbitrageable currencies were USD, HKD, and CHF. The most profitable cycle was EUR → SGD → CHF → HKD → JPY → EUR with a profit of ~1.005.

## 6. Challenges, Ideas and Continued Learning

After presenting the project back in April, we received a lot of great feedback from both Professor Belcher and our peers. In addition to this feedback, we had a number of ideas for this project, many of which never made it into the final product, but some are worth discussing individually. We also experienced several challenges along the way that are included below.

**Dataset Size**

With a dataset comprising over two billion individual data points, one can imagine how difficult that is to handle on a personal laptop. The compressed archive from Polygon.io had a total size of 20 GB, which expanded to approximately 83 GB uncompressed. This was a significant burden on the local disk space of our computers posing a couple serious risks. First, a personal computer might not have 83 GB of disk space available for the data throughout the duration of the project even if that made sense. Second, repeating full decompression and recompression of data not only wastes time, but also risks mid-process crashes if there isn’t enough disk space. To combat these limits, we built a processing pipeline in Python that handles each file in one pass:

1. Skip processing if the filtered version already exists.
2. Read archive into dataframe.
3. Apply filters on the fly.
4. Sort by timestamp.
5. Keep only the 10 chosen currencies.
6. Remove duplicate intra-second ticks (retaining the final tick per timestamp).
7. Save filtered data to new files (between 10 and 200MB).
8. Skip generating arbitrage data if the arbitrage file already exists.

9. Run arbitrage binary on filtered file stream.
10. Save arbitrage information for post processing.

**Low Resolution Data**

Unfortunately our data did not specify timestamps less than one second apart, meaning that we could never know whether our arbitrage system would have been fast enough to execute on the exchange rates available. Therefore, we took a different approach to measuring the efficiency of our system with benchmarking. As mentioned in class, successfully trading in the high frequency world requires less than half a microsecond, so our goal was for our algorithm to be able to generate arbitrage opportunities in much less than that given we do not know the latency of the trading system we might connect our algorithm to. We also experimented with caching cycles and a more detailed analysis of the differences in both algorithm and efficiency was discussed above.

**Finding Multiple Cycles**

This was a severe limitation of the Bellman-Ford implementation and one of the reasons we constructed the caching algorithm. By considering every possible cycle at every timestamp, we can discover every single arbitrage opportunity in the graph and then find the most profitable and most occurring cycles. This is something that we thought was of higher importance to implement and thus made it a part of the caching algorithm.

**Splitting the Currency Graph into Subgraphs**

The idea of taking the entire currency graph and splitting it into subgraphs brings a lot of questions. How should we determine what defines each subgraph? Should it be disjoint sets of currencies or overlapping? Should we try to detect individual arbitrage opportunities within each subgraph or say find half a cycle in one subgraph and connect it to another half from another subgraph. A complete graph with n vertices in our case has n * (n - 1) edges since each currency has an exchange rate to every other currency in the graph. Since this grows quadratically as the number of currencies increases, we can greatly reduce the number of edges we are tracking: for example one complete graph with ten currencies has ninety edges while splitting that into two disjoint subgraphs combine for forty edges.

Furthermore, if we recall from *Figure 4* above, there are over a million possible cycles for a graph with 10 currencies while there are only 84 in a graph with five currencies. Combine these facts with most of the arbitrage opportunities we found having fewer than five edges and it would make a lot of sense to implement something like this with the caching strategy. With a little bit of data analysis and backtesting, a reasonable way to split the currency graph disjointly into subgraphs producing a close to maximal number of arbitrage opportunities would be feasible and work well in a multiprogramming context. We could think of each subgraph creating a table and utilizing its own FPGA to run strategies across a wide array of currencies.

### Order Level Data

Unfortunately we did not have the time to get our hands on order level data, however, this would be imperative to backtesting and implementing an actual trading strategy. If we consider that we sell through a tick on a pair of currencies, our arbitrage opportunity could evaporate since they are incredibly slim margins. Thus, if our model is able to account for order data, then we can compute exactly how much of each currency we are able to trade through at the quotes we have. This would give a much more tangible idea of the kind of profit there is to be made on each of these opportunities and lends itself to much more accurate strategy development.

### Other Currencies

Including cryptocurrencies in this project was mentioned by numerous folks on several occasions, however this seemed beyond the scope of our limited timeframe. We did consider what that might look like and ultimately decided that it would not be worth pursuing as it would require a multitude of additional data that we did not have. Further, we are already operating on many assumptions about the forex market for this project and introducing further assumptions about verifying and staking blockchain transactions would only serve to reduce the validity of our findings with regards to real world consequences.

What would be more likely and much more feasible would be to consider more volatile currencies that we already had data for. This was discussed many times but ultimately never explored due to time constraints. This could be interesting when combined with the subgraph approach mentioned above.

## 7. Conclusion

Ultimately, we gained a much deeper understanding of the foreign exchange market throughout this project. Primarily, we discovered that there are in fact many arbitrage opportunities in these markets, albeit for incredibly slim profit margins. Part of that might be a result of our imperfect data which lacked individual order book information as well as intra-second quote information. This was a significant issue in determining how to measure the efficiency of our algorithms. Without this data, it is impossible to accurately measure how long we would have to take advantage of these opportunities or how much volume the market could handle.

We are incredibly pleased with our arbitrage detection strategy, especially after exploring the caching approach and finding success. In the end, it makes sense that it is much more efficient to cache data on many specific cycles than to actually run Bellman-Ford upon quote updates. As discussed, the caching approach is also very friendly to FPGA usage, which is crucial to modern high frequency trading strategies.

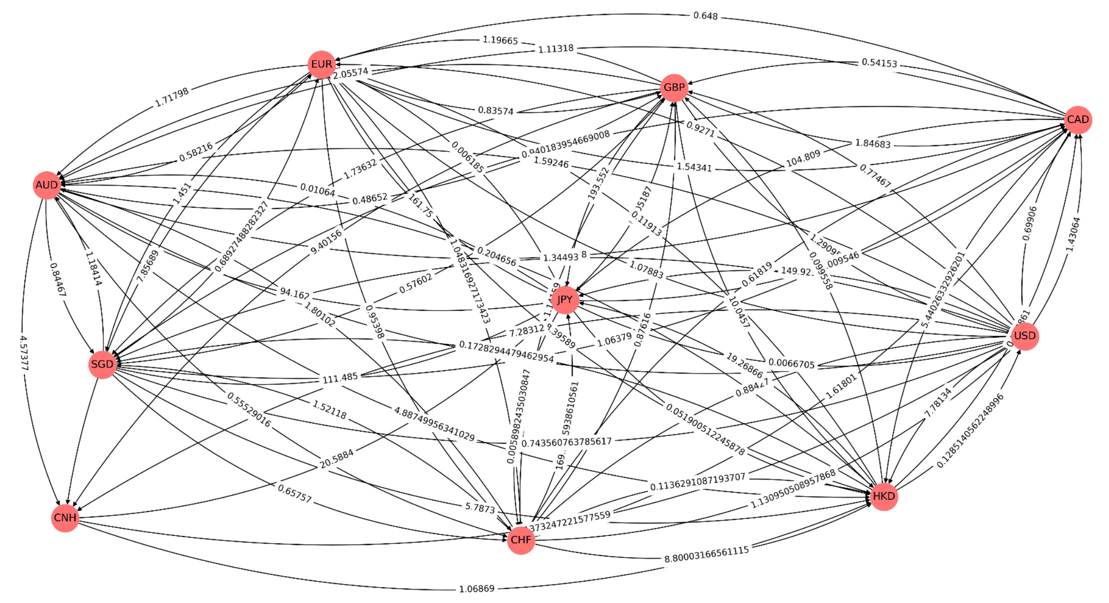

*Figure 17. Liberation Day Foreign Exchange Graph*