# Arbitrage Detection in Financial Markets: A Graph Theory Approach Using the Bellman-Ford Algorithm

Nathaniel Jonathan Rusli - 13523013[1,2]
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
[1]13523013@std.stei.itb.ac.id, [2]omgitsnathaniels@gmail.com

***Abstract*—This paper presents a model designed to detect arbitrage opportunities, focusing on triangular and cross-market arbitrage. Utilizing the Bellman-Ford algorithm and graph theory, the model effectively identifies negative cycles indicative of potential arbitrage in high-liquidity environments, incorporating both dummy and real-time data. Whereas it proves particularly effective for triangular arbitrage, the model requires further refinement to enhance its effectiveness in cross-market scenarios. In real trading scenarios, the model faces significant challenges such as the need for rapid execution, the impact of transaction fees, and the demands of a volatile financial market. The study discusses necessary model enhancements to improve real-world applicability and execution efficiency.**

***Keywords*—Arbitrage detection, Financial markets, Graph theory, Bellman-Ford algorithm**

## I. Introduction

Arbitrage, the practice of exploiting price discrepancies across different markets or exchanges to generate a profit, is a crucial strategy in financial markets. In highly liquid markets like foreign exchange, cryptocurrencies, and equities, slight price differences between exchanges presents opportunities for arbitrage traders to generate risk-free profits by buying low in one market and selling high in another. Although arbitrage trading might appear to be illegal, it is completely legal in most countries. Additionally, the execution arbitrage trading restores market efficiency, as the lower-priced assets are bid up and the higher-priced assets are sold off [1].

Arbitrage detection is the holy grail for traders and institutions with high-frequency trading, a type of algorithmic trading incorporating high volume of stocks at high speeds, as these opportunities often exist for only a short period and require rapid execution. Furthermore, this concept also holds significant importance in the field of financial academia. With the growing complexity of markets, modern approaches to arbitrage detection have been increasingly essential, as traditional methods, such as manual monitoring or rule-based systems, are no longer reliable due to their slowness and inefficiency [2].

Graph theory offers a robust framework for modelling the complex relationships between financial entities, such as exchanges and asset pairs. In this context, financial markets are represented as graphs, where vertices represent assets or exchanges and edges represent the rates between them. By utilizing graph-based algorithms, profitable arbitrage opportunities can be detected more efficiently. This paper will explore the Bellman-Ford algorithm, a classical shortest-path algorithm in graph theory, which is particularly suited for this task as it is able to handle negative edge weights, a characteristic of profitable arbitrage cycles in financial networks [3].

This paper aims to demonstrate how graph theory, combined with the Bellman-Ford algorithm, can be applied to detect arbitrage opportunities in financial markets. By modeling exchanges and asset price differences as a weighted directed graph, we can uncover negative cycles indicative of arbitrage opportunities. This paper also discusses the implementation of this approach, the results obtained from real-world financial data, and the challenges encountered, such as transaction costs and market liquidity. The findings support the use of graph-based models for automated arbitrage detection.

## II. Literature Review

### A. Graph

*1) Definition:* A graph $G$ is a mathematical structure used to model pairwise relations between objects. It is defined as $G = (V, E)$. $V$ is a non-empty set of vertices or nodes. $E$ is a set of edges or links that connect pairs of vertices. This set can be empty, indicating a graph without any edges [4].

*2) Terminologies:* "In the field of graph theory, various key terminologies, which are essential to this study, are used to describe its elements and properties [4]:

*a) Vertex:* An element of set $V$, representing entities such as cities, stations, or data points.

*b) Edge:* An element of set $E$, representing the connection or relationship between two vertices.

*c) Adjacent:* Two vertices are adjacent if there is an edge connecting them directly. For instance (see Fig. 1), vertex $A$ is adjacent to vertices $B$, $C$, and $D$.

*d) Degree:* The number of edges connected to a vertex. For instance (see Fig. 1), vertex *A* has a degree of 3 and vertex *E* has a degree of 2.

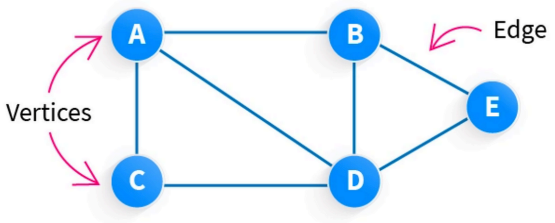


Fig. 1. Illustration of graph's vertex and edge [Source: *https://blog.stackademic.com/graph-basics-explained-from-vertex-to-edges-a0b240041fe7*, Accessed: Dec. 31, 2024.]

*e) Path:* A sequence of vertices where each adjacent pair is connected by an edge.

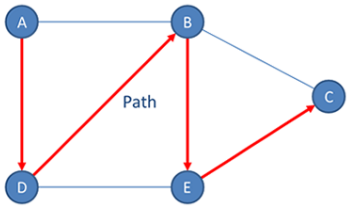


Fig. 2. Illustration of a graph with paths [Source: http://www.computersciencebytes.com/array-variables/graphs/, Accessed: Dec. 31, 2024.]

*f) Cycle:* A path that starts and ends at the same vertex.

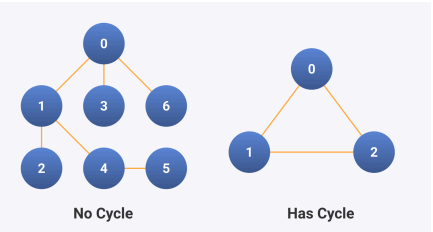


Fig. 3. Illustration of graphs with and without cycle [Source: https://workat.tech/problem-solving/practice/detect-cycle-in-undirected-graph, Accessed: Dec. 31, 2024.]

*g) Isolate Vertex:* A vertex with no connecting edges.

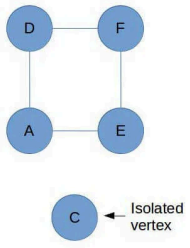


Fig. 4. Illustration of isolated vertex [Source: https://chercher.tech/python-data-structures/graph-theory-python, Accessed: Dec. 31, 2024.]

*3) Representations:* Graphs can be represented in several ways to facilitate different operations [5]:

*a) Adjacency Matrix:* A square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

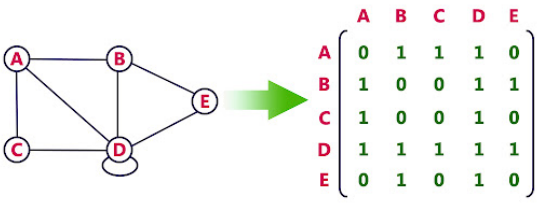


Fig. 5. Graph's adjacency matrix representation [Source: http://www.btechsmartclass.com/data_structures/graph-representations.html, Accessed: Dec. 31, 2024.]

*b) Incidence Matrix:* A matrix that shows the relationship between vertices and edges. The rows represent vertices and columns represent edges, with entries indicating which vertices are connected by which edges.

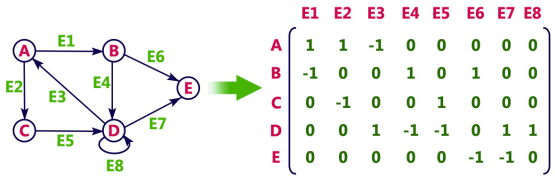


Fig. 6. Graph's incidence matrix representation [Source: http://www.btechsmartclass.com/data_structures/graph-representations.html, Accessed: Dec. 31, 2024.]

*c) Adjacency List:* This uses lists to represent the adjacent vertices for each vertex. It is more space-efficient in terms of sparse graphs than an adjacency matrix.

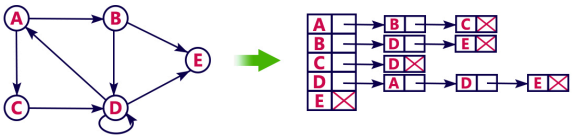

Fig. 7. Graph's adjacency list representation [Source: http://www.btechsmartclass.com/data_structures/graph-representations.html, Accessed: Dec. 31, 2024.]

## *B. Shortest Path Algorithm*

*1) Definition:* The shortest path algorithm is a core concept in graph theory, used to determine the minimum distance or cost required to travel from one node (or vertex) to another within a graph. The shortest problem can be tackled using various algorithms, each suited for different types of graphs and conditions [6].

Shortest path algorithms can be applied towards various types of graphs. A general graph is a mathematical structure made up of vertices (nodes) and

edges (connections). An aspatial graph is a type of graph where the positions of vertices are not interpreted as physical locations in space. A spatial graph, on the other hand, incorporates vertices that have specific locations, represented by the endpoints of the edges. A planar graph is a graph that can be drawn in two dimensions without any edges crossing each other, and the edges can be curved [7].

*2) Types of Shortest Path Algorithm:* Generally, shortest path algorithms can be divided into two broad categories:

*a) Single-Source Shortest Path Algorithm:* These algorithms calculate the shortest paths from a single source vertex to all other vertices within the graph. They are particularly advantageous in scenarios where the starting point is fixed, and the shortest paths of all reachable destinations within the graph need to be explored. Examples of single-source shortest path algorithms are The Dijkstra Algorithm and The Bellman-Ford Algorithm.

*b) All-Pairs Shortest Path Algorithms:* These algorithms calculate the shortest paths between every pair of vertices in a graph, serving a comprehensive solution where all possible routes between vertices are of interest. An example of this algorithm is the Floyd-Warshall algorithm.

Each of these algorithms has its own strengths and is suited for different types of graph structures. In practical applications, the choice of algorithm depends on the specific requirements of the problem at hand [6].

## C. Negative-Weight Cycle

*1) Definition:* Negative-weight cycle in a graph is defined as a cycle where the sum of the edge weights is negative. Traversing this cycle would decrease the total weight, leading to infinitely decreasing path lengths.

*2) Characteristics:* Negative-weight cycles can be determined by several characteristics [8]:

*a) Sum of Weights:* The total weights of the edges in the cycle must be less than zero or negative. If a cycle consists of edges with weights $w_1$, $w_2$, …, $w_k$, then it is classified as a negative-weight cycle if:

$$w_1 + w_2 + \ \dots \ + w_k < 0 \qquad (1)$$

*b) Impact on Shortest Paths:* If a graph contains a negative weight cycle that is reachable from a source vertex, the shortest path to that vertex is not well-defined. This is because one can keep traversing the negative cycle to reduce the path indefinitely, effectively leading to a distance of -∞.

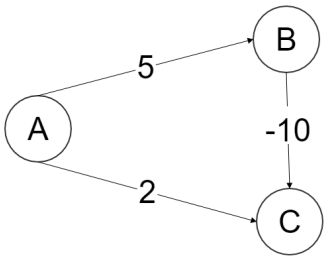


Fig. 8. Example of a negative-weight cycle [Source: https://library.fiveable.me/graph-theory/unit-8/bellman-ford-algorithm-negative-edge-weights/study-guide/5kXLDBHbuRIa0ogE, Accessed: Dec. 31, 2024.]

## D. Bellman-Ford Algorithm

*1) Definition:* The Bellman-Ford algorithm is a widely used method for finding the shortest paths from a single source vertex to all other vertices in a weighted directed graph. It is particularly notable for its ability to handle graphs that contain edges with negative weights, which many other shortest path algorithms, such as Dijkstra's, cannot accommodate. The algorithm utilizes dynamic programming techniques, specifically the "Principle of Relaxation", where it repeatedly updates the shortest path estimates until they converge on the optimal values [9].

*2) Algorithm:* The Bellman-Ford algorithm works by these 3 general steps [9]:

*a) Initialization:* Begin by assigning an initial distance to each vertex in the graph. Set the distance of the source vertex to zero, and assign infinity as the distance for all other vertices.

*b) Relaxation Process:* The algorithm then relaxes all edges in the graph repeatedly for $V - 1$ iterations (where $V$ is the total number of vertices). In each iteration, it checks if the current known distance to a vertex can be improved by going through an adjacent vertex. If a shorter path is found, the algorithm updates the distance.

*c) Final Check for Negative Cycle:* Once the $V - 1$ iterations are complete, the algorithm makes one final pass through all edges. If any distance can still be updated, this signals the presence of a negative weight cycle.

*BELLMAN-FORD (G, W, s)*

1: INITIALIZE-SINGLE-SOURCE *(G, s)*
2: for $i = 1$ to $|V| - 1$
3: do for each edge $(u, v) \in E$
4: do RELAX $(u, v, W)$
5: for each edge $(u, v) \in E$
6: do if $d(v) > d(u) + W(u, v)$
7: return FALSE
8: return TRUE

Fig. 9. Bellman-Ford Algorithm [Source: *Introduction to Algorithms*, 2nd ed. Cambridge, Accessed: Dec. 31, 2024.]

*3) Complexity:* The Bellman-Ford algorithm has a time complexity of $O(V \cdot E)$, where $V$ represents the number of vertices and $E$ is the number of edges in the graph. While this makes it less efficient than Dijkstra's algorithm for graphs that do not have negative weights, it is more versatile because it can handle graphs with negative edge weights.

## *E. Arbitrage Detection Using Graph-Based Models*

*1) Types of Arbitrage and Their Graph Representations:* Various types of arbitrage can be exploited using the graph theory and the Bellman-Ford algorithm [2], [3]:

*a) Currency Arbitrage:* Currency arbitrage involves taking advantage of price differences in exchange rates across various markets.

Each currency is represented as a vertex and the edges represent the exchange rates between the currencies. The weights of the edges are the logarithmic values of the exchange rates, with negative edges indicating arbitrage opportunities.

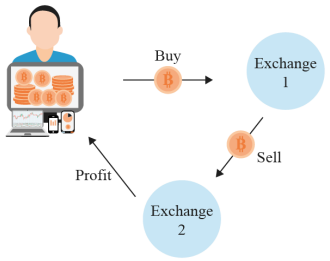


Fig. 10. Illustration of simple currency arbitrage [Source: https://bookmap.com/blog/what-is-arbitrage-trading, Accessed: Dec. 31, 2024.]

*b) Triangular Arbitrage:* Triangular arbitrage occurs when discrepancies between the exchange rates of three currencies can be exploited. For example, if a trader can convert currency A to B, then B to C, and finally convert C back to A at a better rate, it forms a profitable triangular loop.

The nodes represent each currency in the triangle and the edges represent exchange rates. The weights of the edge are the logarithms of the exchange rates, with negative weights indicating a profitable cycle.

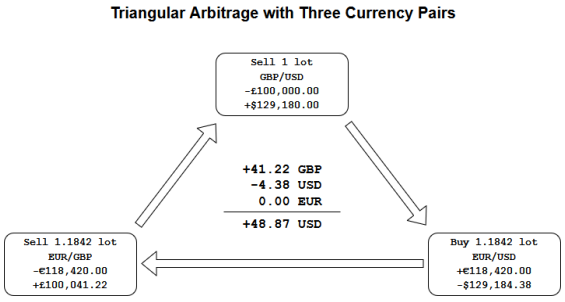


Fig. 11. Illustration of triangular arbitrage [Source: https://www.earnforex.com/guides/arbitrage-for-retail-forex-traders/, Accessed: Dec. 31, 2024.]

*c) Cross-Market Arbitrage:* Cross-market arbitrage happens when the same asset is priced differently across two or more markets. Traders exploit these differences by buying low in one market and selling high in another.

Each market or asset is represented as a node and the edges represent price differences between assets or markets. The edge weights represent the logarithmic price differences between markets.

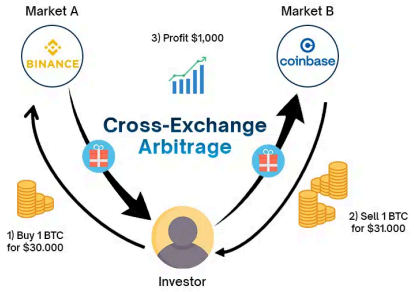


Fig. 12. Illustration of cross-market (cryptocurrency) arbitrage [Source: https://www.antiersolutions.com/crypto-arbitrage-trading-bot-development-key-strategies-for-2024/, Accessed: Dec. 31, 2024.]

*2) Logarithmic Transformation:* Logarithmic transformation is essential for adapting classical graph algorithms to detect arbitrage opportunities, particularly when dealing with multiplicative factors like exchange rates. Traditional graph algorithms, such as BFS, DFS, Dijkstra, or A-Star, are designed to handle additive relationships, where total path cost is the sum of individual edge weights. However, in arbitrage scenarios where the relationship between edges involves products, such as exchange rates, taking the logarithm of edge weights converts these multiplicative relationships into additive ones [3].

This transformation enables the identification of negative cycles in the graph. By taking the logarithm of each exchange rate, we convert multiplication into addition:

$$log(a \times b \times c) = log(a) + log(b) + log(c) \quad (2)$$

To facilitate the detection of arbitrage, we use negative logs for edge weights:

$$w(C_i, C_j) = - log(R[i][j]) \quad (3)$$

*3) Bellman-Ford Algorithm Utilization:* In the realm of financial arbitrage, the Bellman-Ford algorithm plays a crucial role by identifying negative-weight cycles within graph-based models of financial assets. These cycles are key indicators of arbitrage opportunities. Specifically, a negative-weight cycle suggests that initiating a transaction cycle with a certain amount of currency and following through the designated path will result in a return greater than the initial investment. This confirms the potential for arbitrage [2], [3], [9].

The Bellman-Ford algorithm is utilized in the case of arbitrage detection in the following manner:

*a) Initialize Asset Prices:* Set the price of the source asset (starting currency pair or financial instrument) to zero. Assign infinity as the initial price for all other assets to represent them as initially unreachable.

*b) Relax Assets Rates:* For *V - 1* iterations (where *V* is the number of assets or currency pairs), update the cost of reaching each asset by potentially going through another asset that offers a lower cost route.

*c) Check for Profitable Cycles:* After the main iterations, perform one additional pass over all exchange rates. If any asset's cost can be further reduced, this indicates the presence of a negative-weight cycle, revealing an arbitrage opportunity.

*d) Identify Profitable Trading Loops:* Use the negative-weight cycle detected in the previous step to outline a sequence of trades that result in a net increase in value, confirming the viability of the arbitrage strategy.

## III. Implementation

### A. Programming Language and Tools

For this project's implementation, Python was chosen as the programming language due to its readability, extensive library support, and its strong presence in data analysis and scientific computing. The following libraries are employed in this project:

*a) Requests:* Enables simple HTTP requests to fetch data from external APIs, crucial for real-time financial data retrieval.

*b) NumPy:* Offers extensive support for large, multi-dimensional arrays and matrices, accompanied by a comprehensive set of advanced mathematical functions.

*c) Matplotlib and NetworkX:* Facilitates arbitrage graphs visualization.

*d) Python Standard Libraries:* Math, Time, and Datetime libraries are utilized for mathematical computations, time-related tasks, managing date and time data, respectively.

### B. Data Collection and Preprocessing

Three distinct experiments are conducted within this project to encompass a wide range of scenarios in arbitrage: forex triangular arbitrage, cryptocurrency triangular arbitrage, and cryptocurrency cross-exchange arbitrage.

For the forex triangular arbitrage, dummy data is created, mimicking realistic forex market conditions. This approach does not rely on live data, allowing for controlled experimentation and precise evaluation:

```
currency_names = ['USD', 'EUR', 'GBP', 'JPY']
    rates = np.array([
        [1.0000, 0.8425, 0.7250, 113.50],
        [1.1860, 1.0000, 0.8650, 134.50],
        [1.3800, 1.1555, 1.0000, 148.00],
        [0.0088, 0.0074, 0.0068, 1.0000]
    ])
```

On the other hand, for cryptocurrency-related experiments, real-time data is fetched using the CoinGecko API. This approach ensures that the project covers both hypothetical and real-world scenarios, providing a comprehensive analysis of arbitrage opportunities across different markets. The retrieval of specific cryptocurrency pairs and Bitcoin prices across various exchanges via the CoinGecko API is outlined below:

```
def get_price(coin_id, vs_currency):
    URL =
f"https://api.coingecko.com/api/v3/simple/price?id
s={coin_id}&vs_currencies={vs_currency}"
    try:
        response = requests.get(URL)
        response.raise_for_status()
        data = response.json()
        return data.get(coin_id,
{}).get(vs_currency)
    except requests.RequestException as e:
        print(f"Error fetching data for {coin_id}
in {vs_currency}: {e}")
        return None

def get_bitcoin_price(exchanges):
    URL =
"https://api.coingecko.com/api/v3/coins/bitcoin/ti
ckers"
    exchanges_price = []

    for exchange in exchanges:
        try:
            url = URL.replace("{exchange_id}",
exchange)
            response = requests.get(url)
            response.raise_for_status()
            time.sleep(0.2)

            data = response.json()
            tickers = data.get("tickers", [])

            for ticker in tickers:
                if ticker["base"] == "BTC" and
ticker["target"] == "USD":
                    price = ticker["last"]
                    if (price in exchanges_price):
                        continue
                    print(f"Bitcoin price on
{exchange.capitalize()}: {price} USD")
                    exchanges_price.append(price)
                    break

        except
requests.exceptions.RequestException as e:
            print(f"Error fetching data for
{exchange}: {e}")

    return exchanges_price
```

### C. Graph Construction for Arbitrage Detection

The graph, central to our arbitrage detection algorithm, is constructed using a Python class that encapsulates the complexities of graph theory applied to market rates. Here's an outline of the implementation:

```
class Graph:
    def __init__(self, currency_names, rates):
        self._vertices = len(currency_names)
        self._edges = []
        self._currency_names = currency_names
        self._build(rates, currency_names)

    def add_edge(self, u, v, w):
        if u >= self._vertices or v >=
self._vertices or u < 0 or v < 0:
            raise ValueError("Vertex index out of
bound")
        self._edges.append((u, v, w))

    def _build(self, rates, currency_names):
        for i in range(len(currency_names)):
            for j in range(len(currency_names)):
                if i != j:
                    # Logarithmic conversion
                    self.add_edge(i, j,
-log(rates[i][j]))

    @property
    def vertices(self):
        return self._vertices

    @vertices.setter
    def vectices(self, vertices):
        if vertices < 1:
            raise ValueError("Number of vertices
must be at least 1")
        self._vertices = vertices

    @property
    def edges(self):
        return self._edges

    @property
    def currency_names(self):
        return self._currency_names

    @currency_names.setter
    def currency_names(self, currency_names):
        self._currency_names = currency_names
```

This class incorporates logarithmic conversion as a crucial step, transforming currency exchange rates into weights. This adjustment is vital for working with graph-based algorithms, allowing us to spot negative cycles that indicate potential arbitrage opportunities.

### D. Bellman-Ford Algorithm Implementation

The Bellman-Ford algorithm plays a crucial role in detecting negative cycles within our graph model, which indicate potential arbitrage opportunities. This algorithm methodically updates the estimated costs to reach each vertex, looking for any reductions in cost that occur after all vertices have been processed initially. If such a reduction is found, it signals the presence of a negative cycle, suggesting a profitable arbitrage path. The implementation is outlined as follows:

```
def bellman_ford(graph, src):
    dist = [float('Inf')] * graph.vertices
    dist[src] = 0
    predecessors = [None] * graph.vertices

    # Relax edges V - 1 times
    for _ in range(graph.vertices - 1):
        for u, v, w in graph.edges:
            if dist[u] != float('Inf') and dist[u]
+ w < dist[v]:
                dist[v] = dist[u] + w
                predecessors[v] = u

    # Check for negative-weight cycles
    for u, v, w in graph.edges:
        if dist[u] + w < dist[v]:
            return True, predecessors
    return False, None
```

Arbitrage detection is triggered through the detect_arbitrage function, which employs a variety of helper functions to navigate through the graph, identify viable cycles, and calculate potential profits:

```
def detect_arbitrage(graph):
    for src in range(graph.vertices):
        has_cycle, predecessors =
bellman_ford(graph, src)
        cycle = reconstruct_cycle(predecessors,
src)[::-1]
        if has_cycle and
is_valid_arbitrage_cycle(cycle, graph):
            calculate_profit(cycle, graph)
```

### E. Arbitrage Detection Process

Arbitrage detection begins by constructing a graph with currency symbols and their logarithmic exchange rates as nodes and edges, respectively. The detect_arbitrage function is then called to apply the Bellman-Ford algorithm, searching for negative cycles that indicate potential arbitrage opportunities:

```
graph = Graph(symbols, exchange_rates_matrix)
detect_arbitrage(graph)
```

## IV. Results and Discussion

### A. Arbitrage Detection and Opportunities Detected

*a) Foreign Exchange Triangular Arbitrage:* In this particular analysis, major currencies, such as USD (United States Dollar), EUR (Euro), GBP (British Pound), and JPY (Japanese Yen), are considered. The exchange rates between these currencies were structured into a matrix as presented in Table I. This matrix set the stage for identifying potential triangular arbitrage opportunities within these currencies.

Using the arbitrage detection algorithm, several profitable cycles were identified, the details of which are shown in Table II. Notably, the first, second, and fourth arbitrages, despite starting from different currencies (USD, JPY, and GBP respectively), yielded the same profit of 6.51%. This uniformity arises

because these cycles involve the same set of currency exchanges, merely differing in their starting points.

TABLE I
EXCHANGE RATES BETWEEN USD, EUR, GBP, AND JPY PAIRS

| *Currency* | *USD* | *EUR* | *GBP* | *JPY* |
|---|---|---|---|---|
| USD | 1.000 | 0.8425 | 0.7250 | 113.5 |
| EUR | 1.1860 | 1.0000 | 0.8650 | 134.50 |
| GPB | 1.3800 | 1.1555 | 1.0000 | 148.00 |
| JPY | 0.0088 | 0.0074 | 0.0068 | 1.000 |

The second arbitrage detected, which encompasses all four currencies (EUR, USD, JPY, and GBP), produced a distinct profit of 5.77%. This highlights the impact of incorporating a broader range of currencies and the potential for higher returns when the cycle extends through more exchange rates.

TABLE II
FOREIGN EXCHANGE TRIANGULAR ARBITRAGES DETECTED

| *Arbitrage Cycle*[a] | *Profit*[b] |
|---|---|
| USD → JPY → GBP → USD | 6.51% |
| EUR → USD → JPY → GBP → EUR | 5.77% |
| GBP → USD → JPY → GBP | 6.51% |
| JPY → GBP → USD → JPY | 6.51% |

To check the credibility of the arbitrage detected, mathematical proof is provided to verify the real profitability of the instances, specifically for the first and second arbitrages.

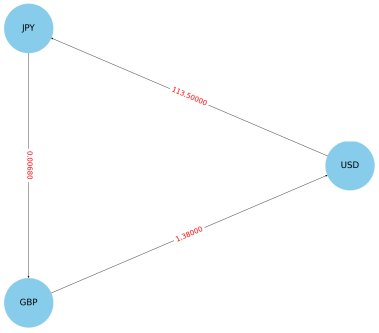


Fig. 13. Graph illustration of the first forex triangular arbitrage detected [Source: Author's visualization using NetworkX and Matplotlib]

The first arbitrage cycle (USD → JPY → GBP → USD) utilizes the following exchange rates: USD to JPY at 113.50, JPY to GBP at 0.0068, and GBP to USD at 1.3800. The followings are the calculation steps, starting with 1 USD:

*1) Convert USD to JPY:* Convert to 113.50 JPY.
*2) Convert JPY to GBP:* Convert to 0.7718 GBP.
*3) Convert GBP to USD:* Convert to 1.0653 USD.

The cycle completes with a return of 1.0651 USD, representing a 6.51% increase from the initial capital.

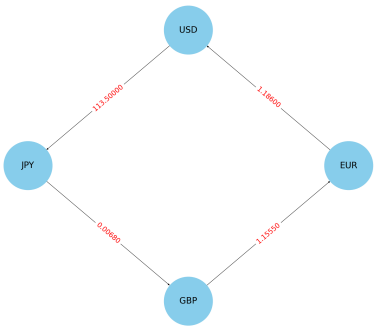


Fig. 14. Graph illustration of the second forex triangular arbitrage detected [Source: Author's visualization using NetworkX and Matplotlib]

The second arbitrage cycle (EUR → USD → JPY → GBP → EUR) utilizes the following exchange rates: EUR to USD at 1.1860, USD to JPY at 113.50, JPY to GBP at 0.0068, and GBP to EUR at 1.1555. The followings are the calculation steps, starting with 1 EUR:

*1) Convert EUR to USD:* Convert to 1.1860 USD.
*2) Convert USD to JPY:* Convert to 134.61 JPY.
*3) Convert JPY to GBP:* Convert to 0.9153 GBP.
*4) Convert GBP to EUR:* Convert to 1.0577 EUR.

The cycle completes with a return of 1.0577 EUR, representing a 5.77% increase from the initial capital.

*b) Cryptocurrency Triangular Arbitrage:* For the cryptocurrency triangular arbitrage experiment, a more realistic approach was adopted by focusing on the current top three cryptocurrencies by market capitalization: Bitcoin (BTC), Ethereum (ETH), and Ripple (XRP). This choice was driven by the significant trading volumes and liquidity that these currencies typically exhibit, which are crucial factors for the viability of arbitrage opportunities.

The experiment involved fetching real-time data, ensuring that the analysis reflects the dynamic nature of the cryptocurrency market. The specific data used in this study was fetched on 2025-01-04 at 18:19:13.344470. This data is comprehensively detailed in Table III, which displays the exchange rates between these three cryptocurrencies.

During this particular cryptocurrency triangular arbitrage experiment, a profitable cycle was detected. Three different cycles were examined: BTC → ETH → XRP → BTC, ETH → XRP → BTC → ETH, and XRP → BTC → ETH → XRP. Remarkably, all three cycles generated an identical profit of 0.04%. This uniformity

in profit results from the cycles incorporating the same currency pairs, albeit in different sequences.

TABLE III
FETCHED RATES OF EACH CRYPTOCURRENCY PAIR

| *Pair*[a] | *Rate*[b] |
|---|---|
| BTC/ETH | 27.236375 |
| BTC/XRP | 40,142 |
| ETH/BTC | 0.03671743 |
| ETH/XPR | 1,474 |
| XRP/BTC | 0.00002492 |
| XRP/ETH | 0.0006784 |

For instance, the arbitrage cycle BTC → ETH → XRP → BTC utilizes the following exchange rates: BTC to ETH at 27.236375 (1 BTC buys 27.236375 ETH), ETH to XRP: 1474 (1 ETH buys 1474 XRP), and XRP to BTC: 2.492e-05 (1 XRP buys 0.00002492 BTC). The followings are the calculation steps, starting with 1 BTC:

*1) Convert BTC to ETH:* Convert to 27.236375 ETH.
*2) Convert ETH to XRP:* Convert to 40142 XRP.
*3) Convert XRP to BTC:* Convert to 1.00004 BTC.

The cycle completes with a return of 1.00004 BTC, representing 0.04% increase from the initial capital.

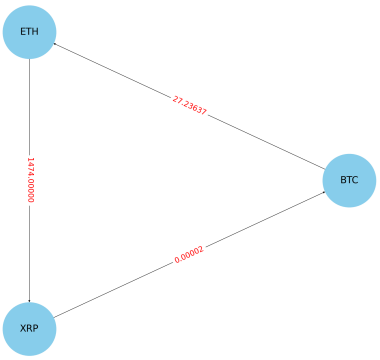


Fig. 15. Graph illustration of the cryptocurrency triangular arbitrage detected [Source: Author's visualization using NetworkX and Matplotlib]

The experiment reveals a modest 0.04% profit from cryptocurrency triangular arbitrage, but real-world conditions usually offer even slimmer margins. This is due to tight market spreads and efficient pricing, where quick arbitrage windows close fast, and transaction fees can wipe out potential profits. Such conditions highlight the importance of precise execution and large transaction volumes to make the most of these fleeting opportunities.

*c) Cross-Market Arbitrage:* This experiment is focused on Bitcoin, using real-time data fetched from four major cryptocurrency exchanges: Binance, Kraken, Coinbase, and Bitstamp. The data as shown in Table IV, gathered on 2025-01-04 at 18:33:51.379468, was used to analyze potential arbitrage opportunities by comparing Bitcoin prices across these platforms.

TABLE IV
BITCOIN'S PRICE ON DIFFERENT EXCHANGES

| *Exchange*[a] | *Price ($)*[b] |
|---|---|
| Binance | 97,746.80 |
| Kraken | 97,735.00 |
| Coinbase | 97,730.34 |
| Bitstamp | 97,740.40 |

The arbitrage detection model identified a specific sequence, Kraken → Bitstamp → Coinbase → Kraken of trades that theoretically should result in profit based on slight price discrepancies between the exchanges (start with 1 BTC):

*1) Trade from Kraken to Bitstamp:* Convert to 1.00006 BTC at a rate of 1.00006.
*2) Trade from Bitstamp to Coinbase:* Convert to 0.99995 BTC at a rate of 0.99995.
*3) Trade from Coinbase to Kraken:* Convert to 1.00000 BTC at a rate of 1.00005.

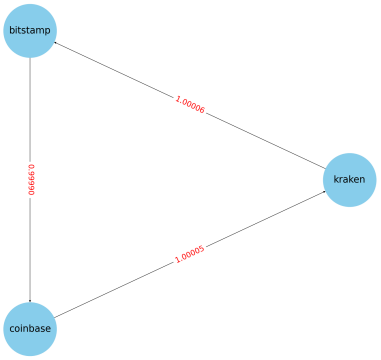


Fig. 16. Graph illustration of the cryptocurrency cross-exchange arbitrage detected [Source: Author's visualization using NetworkX and Matplotlib]

The inherent design of the current arbitrage detection model is structured to always complete a cycle back to the starting exchange, regardless of the profit or loss encountered during the trading sequence. This approach, while effective for identifying price discrepancies between exchanges, systematically returns to the initial starting point, which inherently

neutralizes any accrued profit by the end of the cycle with a 0% profit/loss.

Whereas this behavior particularly suits the model for triangular arbitrage within a single market this model shows limitations when applied to cross-market arbitrage, where opportunities might be better seized by not returning to the starting point but rather by strategically ending trades at points of maximum profitability. To address this and enhance its effectiveness for cross-market arbitrage, the model needs a fundamental redesign, such as profit-taking strategy, dynamic decision making, and conditional cycle completion.

### *B. Limitations and Challenges*

Although the Bellman-Ford algorithm and graph theory are powerful tools for detecting arbitrage opportunities in financial markets, their real-world application is faced with challenges and limitations. Timing is crucial in trading, with arbitrage opportunities typically lasting only a short time due to the rapid pace of financial markets..

Transaction fees also significantly impact the profitability of arbitrage strategies. Each trade incurs costs that vary by trading platform, transaction volume, and the financial instruments used. These fees can quickly erode profits from arbitrage, particularly if the strategy requires multiple transactions to complete a cycle as the algorithm suggests.

Market liquidity is also vital. Without sufficient volume, trading activity might influence market prices and compromise arbitrage opportunities. These challenges emphasize the need for real-time data processing, fast execution, and detailed cost-benefit analysis.. Therefore, whereas these tools can detect potential opportunities, their practical application requires careful consideration of operational and market realities.

## V. Conclusion

The Bellman-Ford Algorithm and the graph-based model has proven effective for arbitrage detection, particularly in triangular arbitrage, and can also be adapted for cross-market arbitrage with further refinements. Whereas the model efficiently uncovers arbitrage opportunities, implementing it successfully in real trading scenarios poses significant challenges. These challenges include the crucial timing of trades, significant transaction fees, and the necessity for robust market liquidity and stability. Moreover, the fast-paced nature of financial markets requires advanced technology for quick and effective execution. Despite these obstacles, with strategic modifications and improvements, the model has great potential to exploit arbitrage opportunities across various market conditions.

## VI. Acknowledgment

The author expresses heartfelt gratitude to God Almighty for granting the strength, perseverance, and opportunity to successfully complete this paper. The author also wishes to express profound gratitude to Dr. Ir. Rinaldi, M.T., the lecturer for the IF1220 Discrete Mathematics course, for his unwavering guidance, inspiration, and motivation throughout his time teaching the students.

## VII. Appendix

The complete source code used for the arbitrage detection, including real-time data fetching, arbitrage algorithm implementation, and visualization tools, is available on GitHub. Access the code repository here: GitHub Repository Link

## References

[1] J. Fernando, "Arbitrage," *Investopedia*, Mar. 20, 2023. https://www.investopedia.com/terms/a/arbitrage.asp (accessed Dec. 30, 2024).

[2] R. Andrew Martin, "Graph algorithms and currency arbitrage, part 1 · Reasonable Deviations," *reasonabledeviations.com*, May 02, 2019.https://reasonabledeviations.com/2019/03/02/currency-arbitrage-graphs/ (accessed Dec. 30, 2024).

[3] A. Yarkov, "Algorithmic Alchemy: Exploiting Graph Theory in the Foreign Exchange," *DEV Community*, Oct. 05, 2023. https://dev.to/optiklab/algorithmic-alchemy-exploiting-graph-theory-in-the-foreign-exchange-399k (accessed Dec. 30, 2024).

[4] R. Munir. "Graf (Bag. 1)", 2023. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf (accessed: Dec. 30, 2024).

[5] R. Munir. "Graf (Bag. 2)", 2023. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf (accessed: Dec. 30, 2024).

[6] "Shortest Path Algorithm Tutorial with Problems," *GeeksforGeeks*, Nov. 02, 2023. https://www.geeksforgeeks.org/shortest-path-algorithms-a-complete-guide/#types-of-shortest-path-algorithms (accessed Dec. 30, 2024).

[7] Amgad Madkour, W. G. Aref, F. Zhang, M. F. Abdel, and Saleh Basalamah, "A Survey of Shortest-Path Algorithms," *arXiv (Cornell University)*, May 2017, doi: https://doi.org/10.48550/arxiv.1705.02044.

[8] "Detect a negative cycle in a Graph | (Bellman Ford)," *GeeksforGeeks*, Oct. 12, 2017. https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/ (accessed Dec. 31, 2024).

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press/McGraw-Hill, 2001

## Statement

I hereby declare that this paper is my own work, not a paraphrase or translation of someone else's paper, and not plagiarism.

Bandung, 5 January 2025

Nathaniel Jonathan Rusli
13523013