A Comparative Study of Floyd-Warshall and Bellman-Ford Algorithms in Diverse Arbitrage Scenarios

Johnson Zhu

1 Introduction

1.1 The Origin of Arbitrage

The idea of arbitrage has existed and been taken advantage of in human history for a long time. There is evidence appearing in some of the earliest surviving Sumerian cuneiform tablets from Bronze Age Mesopotamia, millennia prior to the introduction of coinage, illustrating records of ancient Greek merchants utilizing the value discrepancy in two places to make profits. [5] More recent examples of the idea include British merchants' activity in the 17th contrary, where they bought porcelain wares from China and spice from India, and resold them in Britain at a higher price to gain a considerable amount of profits. [1]

1.2 Definition of Arbitrage

In economic terms, the idea of arbitrage, in general, is defined as "the practice of taking advantage of a difference in prices in two or more markets." [2] There are myriad subgenres of arbitrage in the financial markets, and the one associated with exploiting the exchange rate differences between currencies is called currency arbitrage, which is the subgenre this passage is going to focus on.

Currency arbitrage involves the exchange of currencies through an array of different nations, and as the money is exchanged for different currencies, the possibility exists that the absolute value of the money would alter while it is being converted to different countries' currencies due to incongruency in the exchange rates. If the absolute value of the money increases after it is eventually

converted back to the original country's currency, then a currency arbitrage is considered successfully accomplished. The reason that such incongruency in currency exchange could exist is mainly because of the liquidity differences in the countries' financial markets and variations in the speed at which market information is disseminated among participants.

1.3 Social Implication of Arbitrage

For a long period of time, people have been viewing arbitrage activities led by currency arbitrage with a discontent eye because of its speculative nature. However, as the free-market economy outlook and economists like Pareto's theory became more prevalent, the public started to realize arbitrage's ability to mediate economic overbalance. As mentioned in the above section, the arbitrage opportunity is created as the result of market discrepancies, and as arbitrageurs begin to exploit such discrepancies by frequently exchanging currencies, the market flow is stimulated, thus increasing the liquidity, speeding the process by which the market attains a balanced state with equalized demand and supply. In other words, currency arbitrage is considered socially conducive in that it expedites the transformation process of markets from a Pareto-inefficient state, in which the distribution of commodities is uneven and possesses different prices, into a Pareto-efficient state, where the prices of the commodities are unified, yielding healthier market environment. [3] [4]

This shifted attitudes toward currency arbitrage and other similar activities resulted in an increasing number of arbitrageurs in the financial market, and incentivized the efforts of computer scientists to compose several algorithms that aim to efficiently detect arbitrage opportunities in a set of currency markets. Among the algorithms utilized by the arbitrageurs, two of the most welcomed algorithms are the Floyd-Warshall algorithm and the Bellman-Ford algorithm, known for their excellent efficiency and adaptability under various market situations. Yet, due to the difference in the mechanism upon which the algorithms were built, the two algorithms will have advantages over each other, depending on the variation of the number of currency exchange rates and currency in a given financial scenario encountered by the arbitrageurs. Recognizing such volatility of the financial market status and that no study has been done previously to explicitly compare the efficiency of two algorithms in finding arbitrage opportunities under different currency market scenarios, the passage aims to formulate several arbitrage scenarios under which to execute both algorithms to comparatively study and summarize their relative advantage in terms of time efficiency.

2 Introduction and Proofs of Arbitrage Detection Algorithms

2.1 Graphical Transformation of Currency Relationships

The transformation of international currency exchange rate relationships into a weighted directed cyclic graph (WDCG) is a well-acknowledged approach to abstraction in the study of quantitative finance. In this graphical representation, each vertex symbolizes a distinct currency, while the weighted edges from vertex A to vertex B denote the exchange rate from currency A to currency B. The presence of an arbitrage opportunity within the market is indicated by a cycle whose cumulative product of edge weights exceeds one, signifying a net increase in unit currency value following a complete transaction cycle through the involved currencies. The majority, if not all, of the most efficacious algorithms for arbitrage detection, are predicated on this conceptual framework. Notably, the Floyd-Warshall and Bellman-Ford algorithms stand out as the predominant methodologies utilized for this purpose. Before we run the comparative evaluation for the two algorithms under different market situations, it is essential to delve into the mathematical specifies of run-time efficiency and correctness proofs to ensure that we have a sufficient understanding of the characteristics of the two methodologies.

2.2 Examination of Bellman-Ford Algorithm

Initially proposed by Alfonso Shimbel in 1955 and later published by Richard Bellman and Lester Ford Jr. in 1958 and 1956, the Bellman-Ford algorithm is an algorithm that computes the shortest path on a directed graph with also the ability to detect negative loops. [6] The latter ability of the algorithm is essential for the detection of arbitrage, as it could be modified to measure the total weight of the greatest cycle in a graph. For our purpose, an arbitrage opportunity is determined when the greatest loop in the currency exchange rate relationships graph has an accumulative weight exceeding one. Given that the underlying functionality that supports the Bellman-Ford algorithm's capability to detect arbitrage is its ability to detect negative cycles in graph $G(V, E)$, we will first focus on the demonstration and proof of this very functionality.

Given a graph $G(V, E)$ with its weight function $w : E \to \mathbb{R}$ and a source vertex s, the Bellman-Ford algorithm is able to find the shortest path of all the vertices to the source vertex $\delta(s, v)$ and return true, otherwise false, should there exist a negative cycle in the graph. Bellman-Ford algorithm achieves these functionalities by continually applying an operation named relax operation upon each edge in the graph $|V| - 1$ times, updating the estimated shortest distance $v.d$

between source point s and every vertex $v \in V$ until $v.d = \delta(s, v)$. Initially, the estimated shortest distance $v.d$ of every $\{v \in V \mid v \neq s\}$ is initialized to positive infinity, which ensures that if there exists a vertex u unattainable from s, $u.d$, after the complete execution of the algorithm, will remain its initial value of positive infinity. We name this property as the *unattainable vertex property*. In addition, $s.d$ is set to 0. Here is the pseudocode representation of the procedure:

Algorithm 1: Bellman-Ford (G, s, w)

Input: input parameters $G(V, E)$, s, w

Output: A boolean result of whether a negative loop exists in G

```
1  //graph G(V, E) is given in the form of an adjacency list
2  INITIALIZE-SINGLE-SOURCE (G, s)
3  for i = 1 to |V| − 1 do
4      for each vertex u ∈ V do
5          for each vertex v ∈ G.Adj[u] do
6              RELAX(u, v, w)
7          end
8      end
9  end
10 for each vertex u ∈ V do
11     for each vertex v ∈ G.Adj[u] do
12         if v.d > u.d + w(u, v) then
13             return FALSE
14         end
15     end
16 end
17 return TRUE
```

Here the RELAX operation is defined as such:

Algorithm 2: RELAX(u, v, w)

```
1  if v.d > u.d + w(u, v) then
2      v.d = u.d + w(u, v)
```

2.2.1 Correctness Proof of Bellman-Ford Algorithm

As mentioned above, the detection of arbitrage opportunities involves the detection of negative cycles in the money exchange rate graph. Thus, we will delve into the correctness of the Bellman-Ford algorithm. We will start with the proof of Bellman-Ford algorithm's ability to correctly calculate the single source shortest path from every vertex to source vertex s, then based on the understanding we will proceed to prove Bellman-Ford algorithm's ability to detect the presence of negative cycles in a graph.

Let procedure Bellman-Ford execute upon the graph $G(V, E)$ with source point s and the weight function $w : E \to \mathbb{R}$, we have to prove that if the procedure returns TRUE, it means for every vertex $v \in V$ there is $v.d = \delta(s, v)$ and if a negative cycle presents in the graph that is attainable from s, the procedure will return FALSE.

Lemma 1: *Given a directed graph $G(V, E)$ with source point s and the weight function $w : E \to \mathbb{R}$, if a negative cycle is not reachable from source point s in G, then after $|V| - 1$ rounds of execution of the For loop from line 3 to line 9 in the aforementioned pseudocode, we have $v.d = \delta(s, v)$ for every $v \in V$ reachable from s.*

Proof: For any vertex u reachable from s we set $p = \langle v_0, v_1, ..., v_k \rangle$ as an arbitrary shortest path from $v_0 = s$ to $v_k = u$, because this path is a simple path according to the definition of shortest path [7], the path p has at most $|V| - 1$ edges. This gives us the property that $k \leq |V| - 1$. The For loop from line 3 to line 9 in the Bellman-Ford pseudocode relaxes all $|E|$ edges for $|V| - 1$ times, and at the ith iteration for $i = 1, 2, ..., k$, there exists edge (v_{i-1}, v_i) being relaxed. In the first iteration, as all the edges are being relaxed and $s.d = \delta(s, s)$ is set to 0, we are able to calculate all the shortest paths consisting only one edge $p = \langle v_0 = s, v_1 \rangle$ through the relationship $\delta(s, v_1) = v_1.d = \min(v_1.d, s.d + w(s, v))$ yielded by the relax operation. In the second iteration, all shortest path routes consist of two edges $p = \langle v_0 = s, v_1, v_2 \rangle$ will be found, given the relaxation of edge (v_1, v_2), which yields $\delta(s, v_2) = \min(v_2.d, v_1.d + w(v_1, v_2))$. The process can then be inductively generalized to show that all shortest distance route $p = \langle v_0, v_1, ..., v_i \rangle$ will be found owning to the relaxation of the edge (v_{i-1}, v_i) after the ith iteration. According to the inequality we established earlier stating $k \leq |V| - 1$, the longest shortest path route will consist of no more than $|V| - 1$ edges, and thus, we are guaranteed to attain the single source shortest path for all vertexes in graph G after $|V| - 1$ iterations.

To enhance the clarity and formality of our proof regarding the Bellman-Ford algorithm's capacity to identify negative cycles, we formally delineate a negative cycle as follows:

Assume that graph G contains a cycle reachable from the source vertex s, we define the cyclic route as $c = \langle v_0, v_1, ..., v_k \rangle$ where we have $v_0 = v_k$ due to the nature of a cycle. A negative cycle is a cyclic route that satisfies this inequality:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \tag{1}$$

By *Lemma 1*, if no negative cycle presents in the graph, every vertex $v \in V$ will have $v.d = \delta(s, v)$ or $v.d = \infty$. This, along with the property of the relax operation we demonstrated earlier, allows us to reach the corollary that after the full execution of the Bellman-Ford procedure, we will have the following relationship for every edge $e = (u, v) \in E$:

$$v.d = \delta(s, v) \leq \delta(s, u) + w(u, v) \tag{2}$$

Because of this, the if statement at line 12 in *Algorithm 1* will not return FALSE, and consequently, the algorithm is guaranteed to return TRUE. Then, we assume that there exists a negative loop $c = \langle v_0, v_1, ..., v_k \rangle$ presenting in graph G where $v_0 = v_k$, we may prove the negative cycle detection of the algorithm utilizing the proof of contradiction.

We first posit that procedure Bellman-Ford returns TRUE, which implies that for $i = 1, 2, ..., k$, we have $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$. We then apply some algebraic transformations:

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

Because $v_0 = v_k$, we have $\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$ which gives us $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$, which contradicts inequality (1) that we established earlier. Thus, the Bellman-Ford procedure will return FALSE, should there exist a negative loop in graph G, which concludes our proof.

2.2.2 Time Complexity Proof of Bellman-Ford Algorithm

According to *Lemma 1*, it takes a maximum of $|V| - 1$ iterations to find the shortest path for every vertex in the graph $G(V, E)$. If in the Vth iteration, the shortest path of a node can still be decreased, it means that there exists a negative cycle attainable from source vertex s. For every iteration, we have to traverse through every edge $e \in E$ and apply the relax operation on each of them, which requires $O(1)$ time to execute. Therefore, the overall time complexity of determining if a negative cycle attainable from a given source vertex s exists is $O(|V||E|)$. However, because our goal is to determine whether arbitrage opportunities exist in a given currency exchange rate relationship graph, which, through reduction, could be transformed into the problem of determining if a negative loop exists in a graph. This requires us to conduct the single-source shortest path finding operation for every vertex in the graph G. As a result, we have the final time complexity of $O(|V|^2|E|)$ for detecting arbitrage opportunities in the given graph. Nevertheless, this process could be optimized by constructing a "super vertex" that connects to every other vertex in the graph with zero-weighted edges, allowing us to detect the existence of negative cycles without iteratively executing the procedure upon every vertex in the graph. By taking this approach, the $O(|V|^2|E|)$ time complexity can be transformed a $O\left(|V+1|(|E|+|V|)\right)$ complexity.

2.3 Examination of Floyd-Warshall Algorithm

Floyd-Warshall algorithm is a shortest path algorithm that employs the idea of dynamic programming. The original form of the algorithm was first proposed by Stephen Warshall in 1962, aiming to find the transitive closure of a graph. [8] The algorithm was later modified and published by Robert Folyd in the same year, in its commonly known form of finding the all-pair shortest path in a directed weighted graph. [9] The central notion of Floyd-Warshall algorithm in finding the shortest path lies in comparing and finding the minimal between the length of two kinds of suboptimal shortest paths, the one that includes the intermediate point k and the one that does not. More specifically, given a graph $G(V, E)$ with $v \in V = \{1, 2, ..., n\}$ and weight function $w : E \to \mathbb{R}$, the algorithm proposed the following relationship:

$$\begin{cases} d_{i,j}^{(1)} = w_{i,j} \\ d_{i,j}^{(k+1)} = \min\left\{d_{i,j}^{(k)}, d_{i,k}^{(k)} + d_{k,j}^{(k)}\right\} \end{cases} \tag{3}$$

Where $d_{i,j}^{(k)}$ denotes the shortest path from vertex i to vertex j that contains intermediate vertexs only from the vertex set $I_k = \{1, 2, ..., k-1\} \subset V$, where $1 \leq k \leq n$. Especially, when $k = 1$, there is $I_1 = \varnothing$, in which case no intermediate vertex is contained in the route and we simply

take $w(i, j)$ as the shortest distance between vertex i and j. Applying relationship (3), the Folyd-Warshall algorithm calculates the distance of shortest path $d_{i,j}$ for every $i, j \in V$ recursively based on previous shortest paths that contain less available intermediate vertexes, until the procedure attains the shortest path that incorporates every available intermediate vertex v in the graph, in other words, the value of $d_{i,j}^{(n)}$. Here is the pseudocode representation of the process:

Algorithm 3: Floyd–Warshall (A, n)

Input: adjacency matrix A with each element $a_{i,j} = w(i, j)$, and number of vertex n

Output: matrix $D^{(n)}$ with each element $d_{i,j}^{(n)}$ denoting the shortest distance between vertex i and j

```
D^(0) = W
for k = 1 to n do
    set D^(k) = (d_{i,j}^(k)) as a new n × n matrix
    for i = 1 to n do
        for j = 1 to n do
            d_{i,j}^(k) = min{d_{i,j}^(k−1), d_{i,k}^(k−1) + d_{k,j}^(k−1)}
        end
    end
end
return D^(n)
```

2.3.1 Correctness Proof of Floyd-Warshall Algorithm

To prove the correctness of Floyd–Warshall algorithm in computing the shortest path for all vertex pairs, it is important to show that $d_{i,j}^{(k)}$ truly signifies the length of the shortest route between vertex i and j, considering intermediate vertexes only from set $I_k = \{1, 2, ..., k-1\}$. It will then be trivial to establish the corollary that $d_{i,j}^{(n)} = \delta(i, j)$. Once this capability of the Floyd–Warshall algorithm to accurately derive the shortest path matrix $D^{(n)}$ is proven, we can then proceed to demonstrate how $D^{(n)}$ can be employed to detect the existence of a negative cycle in G.

Lemma 2: *In the afromentioned recursive equation (3), $d_{i,j}^{(k)}$ is the length of the shortest route between vertex i and j that contains only intermediate vertexs from set $I_k = \{1, 2, ..., k-1\}$.*

Proof: The recursive relation presented in equation (3) lends itself to verification via mathematical induction. Consider the base case where $k = 1$. Here, $d_{i,j}^{(1)}$ intuitively represents the length of the shortest path when the set of intermediate vertices is I_1, implying that the sole path between

vertices i and j, devoid of intermediate vertices, is the direct edge (i, j), which has a length denoted by $w(i, j) = d_{i,j}^{(1)}$. We thus inductively consider the situation wherein $k + 1$. Specifically, there are two scenarios for the shortest path from vertex i to j with all intermediate vertex $v_i \in I_k$:

(a) The shortest path does not include intermediate vertex v_k.

(b) The shortest path includes intermediate vertex v_k.

In scenario (a), the length of the shortest path does not get updated with the introduction of vertex v_k to set I_{k+1}, since the path does not benefit from the inclusion of v_k, and thus $d_{i,j}^{(k+1)} = d_{i,j}^{(k)}$. On the other hand, in scenario (b), the introduction of the intermediate vertex v_k to set I_{k+1} results in a shorter path length. This means the shortest path $i \overset{d_{i,j}}{\rightsquigarrow} j$ can be broken down into two constituent shortest paths intersecting at v_k. In other words, we can have $i \overset{d_{i,k}}{\rightsquigarrow} k \overset{d_{k,j}}{\rightsquigarrow} j$. This implies that the updated shortest path length $d_{i,j}^{(k+1)}$ is the sum of the lengths $d_{i,k}^{(k)}$ and $d_{k,j}^{(k)}$. Consequently, because we calculate $d_{i,j}^{(k+1)}$ by acquiring the lesser value yielded from scenario (a) and (b), we can thereby ascertain that $d_{i,j}^{(k+1)}$ will be the length of the shortest path given that all intermediate vertex $v_i \in I_{k+1}$.

Thus, by *Lemma 2*, $d_{i,j}^{(n)}$ is the length of the shortest route between vertex i and j that contains intermediate points from $I_n = V$, meaning this is the length of the shortest path between vertex i and j with every vertex in the graph available for use, which is the definition of $\delta(i, j)$. Therefore, we reach the conclusion that $d_{i,j}^{(n)} = \delta(i, j)$, validating Floyd–Warshall algorithm's correctness in returning the shortest path matrix $D^{(n)}$.

For the proof of Floyd–Warshall algorithm's ability to detect negative cycles in graph G, we recall inequality (1). It shows that as one traverses through the negative cyclic route, the length of the shortest path - which is defined by the aggregation of the edge weights - will always be able to decrease, in that the total weight of the negative cycle route is less than 0. Thus, if during the execution of the For loop at line 2 in *Algorithm 3*, we obtain the result of $d_{i,i}^{(k+1)} < d_{i,i}^{(k)} = 0$ for any vertex $i \in V$, we can conclude the existence of negative loops in graph G, because the distance from any vertex to the vertex per se should not be able to decrease after the initialization of the algorithm, save there presents a negative cycle in the graph G. Consequently, the Floyd–Warshall algorithm can be modified to the form demonstrated in the following pseudocode to detect negative cycles:

Algorithm 4: Floyd–Warshall (A, n)

Input: adjacency matrix A with each element $a_{i,j} = w(i, j)$, and number of vertices n

Output: A boolean result indicating the existence of negative cycles in G

$D^{(0)} = A$
for $k = 1$ *to* n **do**
  set $D^{(k)} = (d^{(k)}_{i,j})$ as a new $n \times n$ matrix
  for $i = 1$ *to* n **do**
    for $j = 1$ *to* n **do**
      $d^{(k)}_{i,j} = \min\{d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}\}$
    end
    if $d^{(k)}_{i,i} < d^{(k-1)}_{i,i}$ **then**
      return TRUE
    end
  end
end
return FALSE

2.3.2 Time Complexity Proof of Floyd-Warshall Algorithm

As demonstrated in the pseudocode, there are three For loops nested together that compose the major body of the procedure, with each loop iterating through every node. This yields an overall time complexity of $O(|V|^3)$.

Further, because no complex data structures are used in the Floyd-Warshall algorithm, it may have a smaller time constant, despite the fact that its time complexity seems to be larger than that of Bellman-Ford algorithm as the scale of the graph grows larger. This makes it important for us to delve into the nuance performance difference between the two algorithms under various scenarios through the means of experimentation to determine their relative advantages.

2.4 Reduction from Negative Cycle Detection to Arbitrage Detection

To demonstrate how an arbitrage opportunity detection problem can be transformed into a negative cycle detection problem, we recall the graphical representation of the currency exchange rates described at the beginning of the section. Mathematically, we denote the graph as $R(V, E)$ and its weight function as $w : E \to \mathbb{R}$. Specifically, we have the exchange rate from currency A

to currency B represented in the weight of the edge $w(A, B)$. The arbitrage opportunity exists, if there presents a cyclic route $c = \langle v_0, v_1, ..., v_k \rangle$ such that the product of the cycle weight exceeds one. We demonstrate the condition with the following inequality:

$$\prod_{i=1}^{k} w(v_i, v_{i-1}) > 1 \tag{4}$$

By taking the reciprocal on both sides of the inequality, we can transform the inequality into this:

$$\prod_{i=1}^{k} \frac{1}{w(v_i, v_{i-1})} < 1 \tag{5}$$

After that, we can take the logarithm on both sides of the inequality (5) to turn the multiplication operation into addition:

$$\sum_{i=1}^{k} \log\left(\frac{1}{w(v_i, v_{i-1})}\right) < \log(1) = 0 \tag{6}$$

This is in identical form to the negative cycle inequality (1) we established earlier, and thus we have shown that the negative cycle detection problem is in essence the same as an arbitrage opportunity detection problem.

3 Comparative Analysis of the Algorithm Efficiency

3.1 Methodology

3.1.1 Density of Directed Graphs

As the proofs in the former section indicate, the time efficiency for the Bellman-Ford and Floyd–Warshall algorithm to detect arbitrage opportunities in a graph is $O\left((|V| + 1)\left(|E| + |V|\right)\right)$ and $O\left(|V|^3\right)$ respectively. Mathematically, although the Bellman-Ford algorithm seems to possess a lower quadratic growth time complexity after the "super vertex" optimization, owing to the discrepancies in the two algorithm implementations, such mathematical complexity gap could be offset by many nuance factors, like the higher level data structures utilized in the implementation of the Bellman-Ford algorithm.

Therefore, uncertainty exists for us to attempt to predict the performance difference between two algorithms under actual scenarios, making an experiment inspection imperative. In reality,

owing to the variability in the amount of currency exchange rate data available, we may obtain currency exchange rate graphs with diverse vertex-to-edge ratios. As indicated by the time complexity of the two algorithms, significant performance differences might emerge due to different distributions of $|V|$ and $|E|$, we deem it effective for our algorithm comparison goal to delineate a few representative arbitrage scenarios based on different ratios. To achieve this, we will resort to the definition given in discrete math regarding the density of the graphs. The density of a directed graph $G(V, E)$ is defined as [11]:

$$D(G) = \frac{|E|}{2\binom{|V|}{2}} = \frac{|E|}{|V|\left(|V|-1\right)} \tag{7}$$

A graph G is considered a dense graph, if its density $D(G)$ exceeds $\frac{1}{2}$. Conversely, graph G is considered a sparse graph if its density $D(G)$ is less than $\frac{1}{2}$. However, to maximize the contrast between the performance of the two algorithms in facilitating our comparative analysis, we will take the two definitions to their extremity. That is, every dense graph we construct will have a density close to one, and every sparse graph will be generated with its $|E|$ close to that of $|V|$, making its density $D(G)$ resembling that of a tree.

3.1.2 Scenarios and Exchange Rate Graph Construction

With the definition established, we will formulate three scenarios in which to execute both algorithms to achieve an effective evaluation of the relative algorithm performance advantages under different arbitrage circumstances. In the first scenario, we will construct dense graphs with the number of vertexes ranging from 5 - 30 nodes. In the second scenario, we will construct a sparse graph with the number of vertexes falling into an identical range from 5 - 30 nodes. In the third scenario, we will construct a dense graph with the number of vertexes ranging from 110 - 140 nodes. Eventually, in the fourth scenario, we will construct a sparse graph with nodes number ranging from 110 - 140. We will then demonstrate the performance difference between the two algorithms, as well as the trend of run time change as the number of vertexes increases under different scenarios, utilizing a line diagram, with its x-axis denoting the number of vertexes and y-axis denoting the execution time entailed.

To ensure that our evaluation reflects the performance of the algorithms in reality, all data used to construct the currency exchange rate graphs are selected from a real-life dataset that recorded the exchange rate of 150 currencies from the year 2021 to 2024 [10]. Because every currency exchange rate provided in the dataset is normalized in terms of euro, denoted as $r_{i,e}$ the actual exchange rate

from two arbitrary currencies A to B, which is denoted as $r_{a,b}$, will be found by applying this calculation:

$$r_{a,b} = \frac{r_{a,e}}{r_{b,e}} \tag{8}$$

In the experimentation, we will be utilizing method (8) to construct edges in the currency exchange rate graph, in that it allows us to control the density of the graph with great flexibility. The input of the program will begin with a number $\boldsymbol{n}$ indicating the number of currencies included in the graph, and following up are the abbreviations of each currency. After this, another integer $\boldsymbol{m}$ is presented in the input text to indicate the number of edges that will be constructed in the graph, and following which are $\boldsymbol{m}$ lines of statements defining the weight of the edges, each in the format of **currency_A** $\boldsymbol{r_{a,b}}$ **currency_B**. To illustrate, the input given to the graph generation program to create a dense exchange rate graph with 3 currencies will look like what is demonstrated below:

```
3
AUD
GBP
EUR
6
AUD 1.8591 GBP
GBP 0.5378 AUD
AUD 1.5772 EUR
EUR 0.634 AUD
GBP 0.8483 EUR
EUR 1.1787 GBP
```

3.1.3 Visualization Explaination

To take into account the variability presented during the actual execution of the algorithms, we will run the two algorithms comparatively in every graph provided in each scenario 30 times and utilize the statistics of the execution results to construct our line diagram. Visually, the execution results of the two algorithms will be represented by two lines with distinct colors in the plot. Specifically, the mean execution time of the algorithm of the 30 trials will be represented by a point in the algorithm's corresponding line in the diagram, and the standard deviation of the 30 run time results will be signified by an error bar extending from every node in the line. For implementation, Python scripts will be used for the construction of the diagrams.

3.2 Experiment Result Demonstration

3.2.1 Scenario I: Dense Graph with 5 - 30 nodes

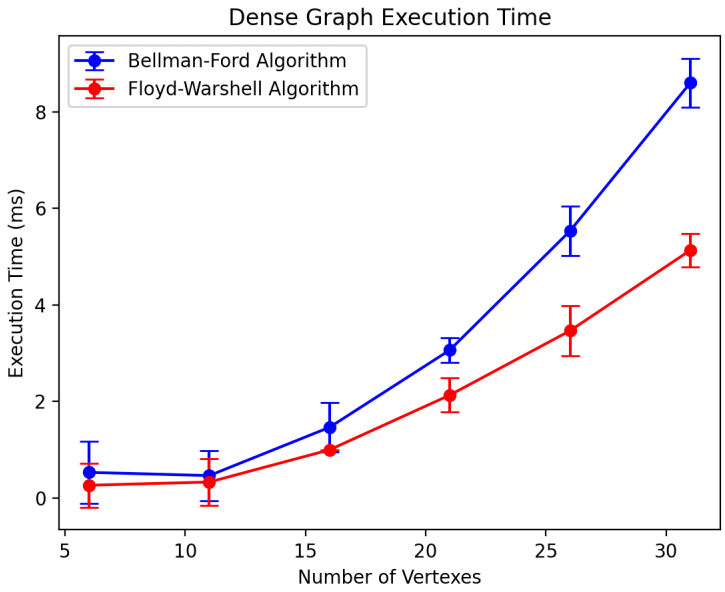

Fig 1: Dense Graph with 5-30 Nodes

As we can observe from the graph, the execution time difference remains relatively small until the number of vertexes surpasses 10, and the performance gap seems to enlarge as the number of vertexes increases. A possible explanation for this performance trend originates from the utilization of higher-level data structures employed in Bellman-Ford's implementation. In the dense graph scenario, we have $|E| \approx |V|^2$, and this may render the actual time complexity of the Bellman-Ford algorithm to a cubic growth rate, similar to that of the Floyd–Warshall algorithm. Nevertheless, despite the fact that both algorithms seem to take a similar amount of steps to solve the problem according to their algorithmic complexity, operations of the adjacency list utilized by the Bellman-Ford algorithm would yield higher constant time costs during the actual execution. Consequently, this can render the Bellman-Ford algorithm less efficient compared to the Floyd–Warshall algorithm when put into practice.

3.2.2 Scenario II: Sparse Graph with 5 - 30 nodes

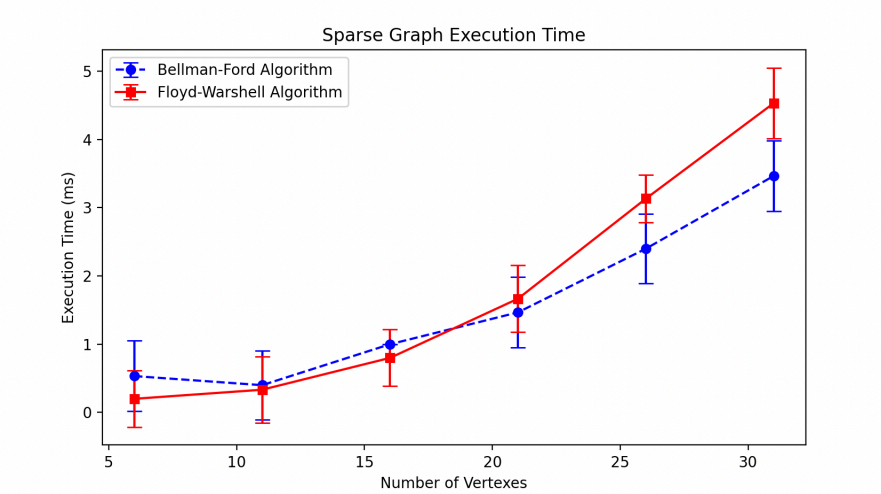

Fig 2: Sparse Graph with 5-30 Nodes

As observed in Fig 2, there are no significant advantages for either Bellman-Ford or Floyd-Warshall algorithm in terms of the execution time of both algorithms. Despite the seemingly relative advantage for Bellman-Ford algorithm during the circumstances under 20 nodes and that for Floyd−Warshall algorithm under circumstances with 20 - 30 nodes yielded by the lines, the actual advantage is not absolute, given the heavy overlap of the error bars of two lines. This ambiguous performance advantage may be produced due to the small difference in polynomial functions' output for small values of input variables. It is worth noticing that as the number of nodes reaches 30, the advantage of the Bellman-Ford algorithm in comparison with Floyd−Warshall seems to grow larger, signifying a potentially better relative efficiency under sparse graphs with a lower number of nodes.

3.2.3 Scenario III: Dense Graph with 110 - 140 nodes

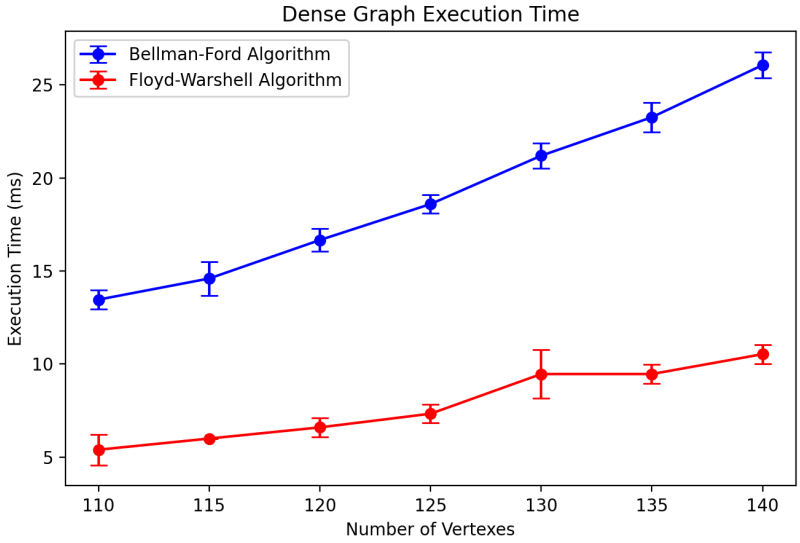

Fig 3: Dense Graph with 110 - 140 Nodes

At higher numbers of nodes, the performance difference seems to grow more significant. With a higher overall number of operations required to detect arbitrage opportunities, the performance discrepancy resulting from greater constant operations in Bellman-Ford Algorithms's implementation is further enlarged. This allows us to quite easily conclude that Floyd–Warshall Algorithm possesses a higher relative execution efficiency under dense graph situations.

3.2.4 Scenario IV: Sparse Graph with 110 - 140 nodes

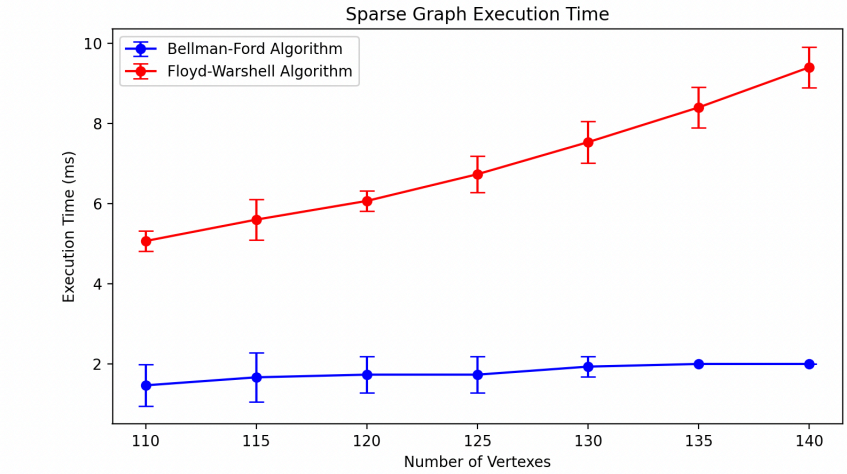

Fig 4: Sparse Graph with 110-140 Nodes

As the graph transitions from dense to sparse, the comparative efficiency appears to shift in favor of the Bellman-Ford Algorithm. This paper posits that the primary cause of this differential performance is rooted in the intrinsic algorithmic complexities of the two procedures. The Bellman-Ford algorithm, exhibiting quadratic complexity, contrasts with the cubic complexity inherent to the Floyd–Warshall algorithm. In scenarios where $|E| \approx |V|$, the operational requirements of the Bellman-Ford algorithm are reduced, thereby translating its algorithmic efficiency into a tangible performance advantage during the actual execution.

4 Conclusion

4.1 Result Summarization

In this paper, noticing the growing vitality of the arbitrage activity, we proposed a comparative study to analyze the execution efficiencies of two most widely used arbitrage opportunity detection algorithms, the Bellman-Ford algorithm and Floyd–Warshall algorithm, under diverse arbitrage scenarios delineated by the density and the number of vertexes of the currency exchange rate graph. Eventually, through executing the algorithms comparatively on various graphs constructed via realistic currency exchange rate data gathered from online datasets and illustrating the result with an error-bar line diagram, we are able to draw conclusions from the trends manifested. We find that under situations wherein a dense graph is presented, Floyd–Warshall algorithm yields better relative efficiency, presumably due to its complexity independent of the number of edges in the graph, and under situations with a sparse graph, the Bellman-Ford algorithm boasts a higher relative efficiency, which may be owning to its quadratic time complexity.

4.2 Reflection and Vision

Although we have mathematically delved into theoretical discrepancies of the Bellman-Ford and the Floyd–Warshall algorithms and have carried out experiments in an attempt to analyze their comparative advantages under diverse arbitrage circumstances, insufficiency indeed still presents in our study. Firstly, the experiment is run on static data generated from a currency exchange rate dataset, which, albeit reflecting reality to a certain extent, does not guarantee a perfect prediction for the performances of the two algorithms under practical situations. Secondly, both algorithms in this paper have been analyzed in their most fundamental form. In reality, the involvement of technical factors such as multithread computation or currency exchange rate API invocation might alter the performance exhibited in this study, making further analysis imperative still, when it comes

to the practical implementation of arbitrage programs.

Nonetheless, I sincerely hope that the light shed by this study on the comparative strengths of the Bellman-Ford algorithm and the Folyd-Warshall algorithm will enrich our collective knowledge regarding the appropriate application of these algorithms across various scenarios. It is also my genuine hope that the insights provided in this study can invigorate arbitrage practices and energize financial exchanges within the global currency markets to promote a more equilibrated economic landscape. In the end, I extend my deepest thanks for your engagement with this research.

Works Cited

Bellman, Richard. “On a routing problem”. *Quarterly of Applied Mathematics*, vol. 16, Apr. 1958, pp. 87–90. https://doi.org/10.1090/qam/102435.

Bhatia, Ruchi. Currency Exchange Rates. www.kaggle.com/datasets/ruchi798/currency-exchange-rates. Accessed 22 Apr. 2024.

Cormen, Thomas, et al. *Introduction to Algorithms*. Third, Mit P, 2009, pp. 580–642.

Diestel, Reinhard. “Graph Decotnpositions”. *Oxford University Press eBooks*, Sept. 1990. https://doi.org/10.1093/oso/9780198532101.001.0001.

Floyd, Robert W. “Algorithm 97: Shortest path”. *Communications of the ACM*, vol. 5, June 1962, p. 345. https://doi.org/10.1145/367766.368168.

La Porte, Mathieu de. ARBITRAGE : Définition de ARBITRAGE. www.cnrtl.fr/lexicographie/arbitrage. Accessed 23 Mar. 2024.

Poitras, Geoffrey. Arbitrage: Historical Perspectives. 2009. www.sfu.ca/~poitras/EQF_ARB%24%24.pdf.

———. “Origins of arbitrage”. *Financial History Review*, May 2021, pp. 1–28. https://doi.org/10.1017/s0968565021000020.

Rifky, Muhammad, and Indraputra Bariansyah. Arbitrage Strategy using Negative Cycle Detection Algorithm. 2019. informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2018-2019/Makalah/Makalah-Stima-2019-134.pdf. Accessed 12 Apr. 2024.

Warshall, Stephen. “A Theorem on Boolean Matrices”. *Journal of the ACM*, vol. 9, Jan. 1962, pp. 11–12. https://doi.org/10.1145/321105.321107.

Weyl, E. Glen. Is Arbitrage Socially Beneficial? Oct. 2007. papers.ssrn.com/sol3/papers.cfm?abstract_id=1324423.

Xiong, Wei. “Convergence trading with wealth effects: an amplification mechanism in financial markets”. *Journal of Financial Economics*, vol. 62, Nov. 2001, pp. 247–92. https://doi.org/10.1016/s0304-405x(01)00078-2.

[1]: (Poitras)

[2]: (la Porte)

[3]: (Xiong)

[4]: (Weyl)

[5]: (Poitras, “Origins of Arbitrage”)

[6]: (Bellman)

[7]: (Cormen et al. 580–642)

[8]: (Warshall)

[9]: (Floyd)

[10]: (Bhatia)

[11]: (Diestel)