Analyzing Foreign Exchange Markets Using Graphs

Tarun Devi, Veer Patel, Ryan Morris

---

1 Abstract

Foreign exchange markets are variable in the sense that exchange rates are set regularly. As countries undergo economic changes, such as inflation, recession, or economic growth, the value of their currencies change accordingly. Sometimes, there are discrepancies between the exchange rates set between different countries. In our case, we are trying to exploit these discrepancies in a form of trading known as *arbitrage* (Mele, 116). Arbitrage will allow us to cycle through a set of currencies and end up with a greater amount of our original currency. We can find this by representing currencies as the vertices of a graph and the exchange rate between currencies as directed edges. After normalizing the weights using a negative logarithm, we use the Bellman-Ford algorithm to look for negative weight cycles, which implies an arbitrage opportunity. We hope to find cycles that allow for a profit.

2 Introduction

In **forex** markets, traders' main incentives are to find relationships between **currencies** such that exchanging them will create a profit. We can represent these markets using graphs and find a profitable method of exchange using algorithms. We can model markets with graphs by assigning each currency a vertex and making each edge between vertices represent the action of the currencies being exchanged. However, currencies can be exchanged forwards and backward (USD $\rightarrow$ BTC, BTC $\rightarrow$ USD). To represent both sides of transactions, we would make each vertex connection have two edges with direction, one going from currency X to currency Y and one from currency Y to currency X.

Each graph should look like a **complete graph**, with **directed edges** and two times the number of edges of a traditional complete graph. This is because each currency in a forex market has n-1 possible trade-ins with

other currencies, where n is the total amount of currencies available to trade in the open market. However, since each currency can be traded the other way, we will have 2(n-1) directed edges for any given currency. Our project will include vertices (currencies) in our forex market graph system.

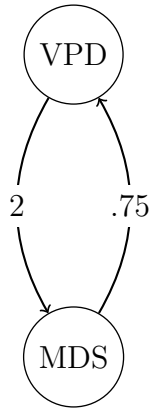
We will use **weighted edges** given that they represent the exchange rate of the related currencies. Rather than using the weights as they are given, we take the negative logarithm of how much 1 unit of the base currency is to how much it is in the destination currency. We do this to have all edges be weighted in an equal form and with a form of magnitude that will allow us to make conversions and calculate cycle weights with regularization. This introduces an additive property of the weights since logarithms convert multiplication to addition. Thus, we use the negative logarithm to express weights, which can be easily reverted using negative exponentiation. Reverting the sum of the weights provides the multiplier for the exchange, specifically opportunity. In other words, it explains how much a currency will multiply after it traverses the cycles of exchanges.

With our project, we are going to make a few core assumptions. Since this project would require monetary resources to fully carry out, specifically getting real-time market data with several currencies, we are going to have to limit our model to have fewer currencies and assume that currencies only make daily updates.

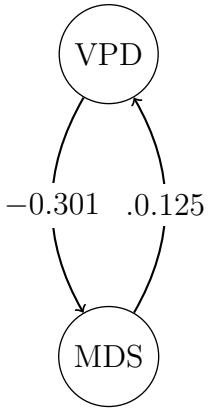
This may seem convoluted, so the following examples explain the principles of finding the cycles without formally implementing the graph traversal algorithms.

3 Small Scale Examples

First, we will observe 2 currencies: the Venerian Pound (VPD) and the Martian Dollar (MDS). In our arbitrary system, we can convert 1 Venerian Pound into 2 Martian Dollars and 1 Martian Dollar into 0.75 Venerian Pounds. Clearly, there is a disparity in the exchange rates that can be exploited. For example, we can start with 10 Venerian Pounds. Converting to Martian Dollars, we have $10 \cdot 2 = 20$ **Martian Dollars. Then, we convert back to VPD:** $20 \cdot 0.75 = 15$ **VPD. We started with 10 Venerian Pounds and ended with 15. This is the core principle of our arbitrage exchange strategy.**



With the normalized weights:



Adding the weights in the normalized graph gives a result of $-.301+.125 = -0.176$**, implying a profitable opportunity since the sums of the weights of the cycle is negative. By reversing the logarithm, we get that the profitability of one cycle is** $10^{-(-0.176)} \approx 1.5$**. By completing one arbitrage cycle, we can multiply our starting money by 1.5x.**

This simple example shows that the exchange rate from currency A into B is always the reciprocal of the exchange rate from currency B into A.

Next, we will observe a case with 3 currencies. We will use Jupiter Dirhams (JDS), Saturn Shillings (SSS), and Neptunian Rupees

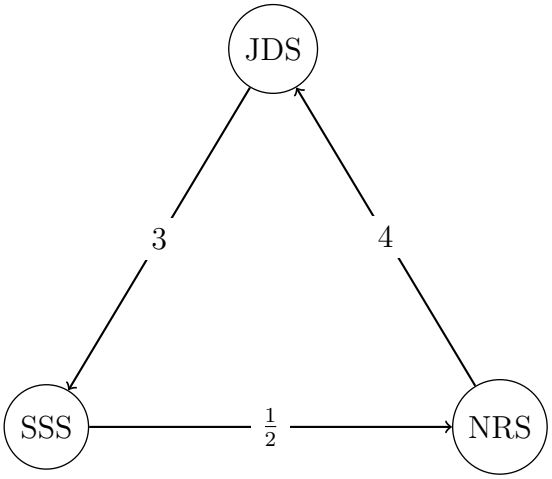
(NRS). We will set the following conversion rates:

$$1 \textbf{ JDS} = 3 \textbf{ SSS}$$

$$1 \textbf{ SSS} = 0.5 \textbf{ NRS}$$

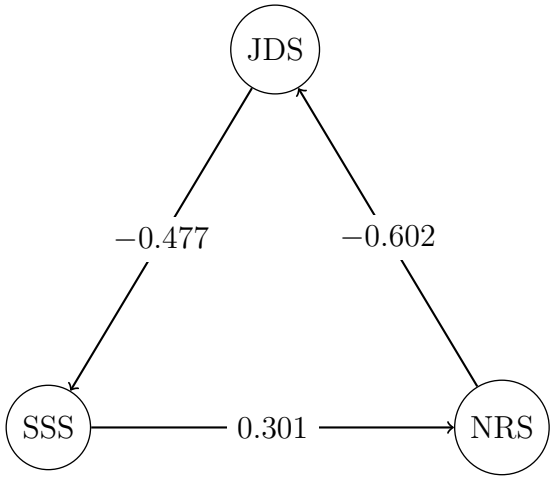
$$1 \textbf{ NRS} = 4 \textbf{ SSS}$$

Using similar reasoning as with 2 currencies, we can start with 1 JDS. We then convert to 3 SSS. We can convert the 3 SSS to 1.5 NRS, with the 0.5 conversion rate. Finally, we take the 1.5 NRS and convert it back to SSS, leaving us with 6 JDS when we started with 1 JDS. The following graphical representation illustrates the same idea.



The graph does not include half of the edges for simplicity purposes, but in the computer representation, there will be an extra edge between each set of vertices with a weight equal to the reciprocal of the existing directed vertex.

Normalized Edge weights (Using Negative Logarithms).



Add the weights of the 3-cycle, starting at JDS and returning back to JDS. The cycle weight is $-0.477 + .301 - 0.602 = -0.778$, and $10^{-(-0.778)} \approx 6$, meaning 1 JDS becomes 6 JDS after the cycle is completed.

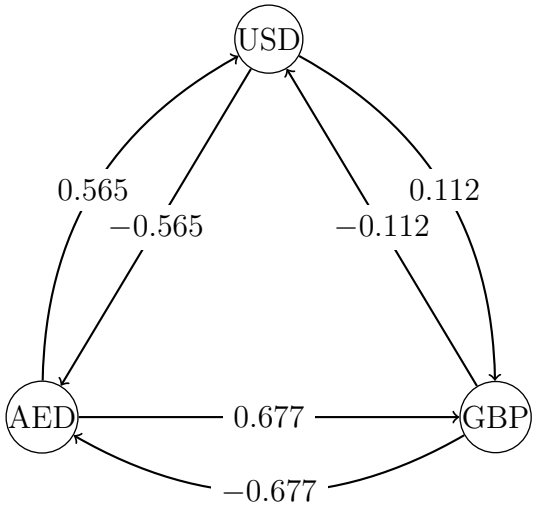
Finally, we will give an example with 3 real currencies: USD, AED, and GBP. For real currency, The conversion rate from A $\rightarrow$ B is the reciprocal of B$\rightarrow$ A. The current conversion rates (as of 11/04/2024) are as follows .

$$1 \textbf{ USD} = 3.6725 \textbf{ AED}$$

$$1 \textbf{ GBP} = 1.2955 \textbf{ USD}$$

$$1 \textbf{ AED} = 0.2102 \textbf{ GBP}$$

Following our trivial calculation, we can see how real currencies are much more balanced than our artificial examples. 1 USD can be converted into 3.6725 AED, which can be converted into 0.7719595 GBP, which is then reconverted into 1.00 USD. This demonstration will demonstrate the nature of balanced cycles with edge weights based on the negative logarithm of exchange rates.



In this example, it is clear that any cycle has weights adding up to 0. This implies that there are no arbitrage opportunities in this set of currencies. If a negative weight cycle is detected, a profitable opportunity exists. This is only possible because exchange rates are set periodically and are not standard. There could be many small discrepancies in values that lead to profitable (negative weight) cycles.

4 Code Implementation

In this section, we will provide an overview of the implementation of the graph and algorithms using Python.

```
class Graph:
    def __init__(self, vertices, index_to_currency):
        self.V = vertices
        self.graph = []
        self.index_to_currency = index_to_currency

    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])


    def BellmanFord(self, src):
```

```
        dist = [float("Inf")] * self.V
        parent = [-1] * self.V
        dist[src] = 0


        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    parent[v] = u

        # Check for negative-weight cycles
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains a negative weight cycle.")
                self.printNegativeCycle(parent, v)
                return


        self.printArr(dist)
```

To detect negative cycles, the Bellman-Ford Algorithm uses relaxation. Approximations for the correct distance between 2 vertices are repeatedly replaced by better solutions until optimal (minimum weight) paths are obtained. If a path from a vertex to itself is found with a weight less than 0, then a negative weight cycle exists. In our use case, this implies an arbitrage opportunity. This is the connection between our textbook and the project. The Bellman-Ford Algorithm uses weighted, directed graphs (as covered in *Applied Combinatorics* (Applied Combinatorics 245).

The algorithm will calculate the shortest distance from all vertices from a source vertex unless a negative weight cycle is detected. If such a cycle is detected, then that is reported rather than the weights.

To implement this by hand, we do the following. Pick a source. Then, for every edge x-y: If D(x) > D(y) + weight of edge xy, then update D(y), setting D(y) = D(x) + weight of edge xy. Once complete, then do the algorithm again. If any edges are updated, then a negative cycle exists.

To implement this using Python, we loop through the list of vertices and adjust the minimum weights of paths between vertices until optimal paths are found.

```
def BellmanFord(self, src):
        dist = [float("Inf")] * self.V
        parent = [-1] * self.V
        dist[src] = 0


        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    parent[v] = u

        # Check for negative-weight cycles
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains a negative weight cycle.")
                self.printNegativeCycle(parent, v)
                return
```

The above is the implementation of the Bellman-Ford Algorithm in our use case. We first complete the algorithm to find the lowest path weights. Then, we loop through each edge again to find the presence of any cycles, as described above.

To obtain the selected currencies, we use a loop of API calls for each currency pair.

```
def buildGraphForSelectedCurrencies(api_key, selected_currencies):
currency_to_index = {currency: idx for idx, currency in enumerate(s
```

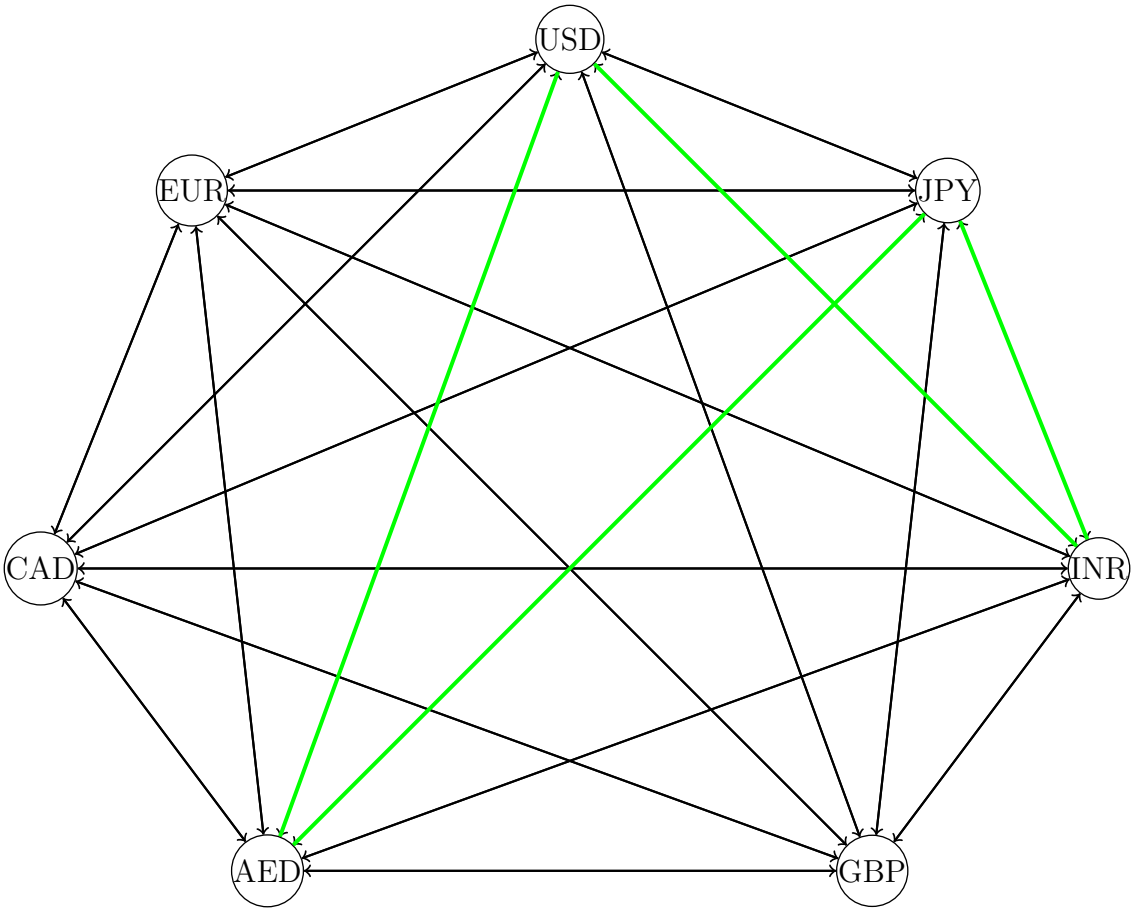
```
    index_to_currency = {idx: currency for currency, idx in currency_to
    num_currencies = len(selected_currencies)


    g = Graph(num_currencies, index_to_currency)


    for base_currency in selected_currencies:

        url = f"https://v6.exchangerate-api.com/v6/{api_key}/latest/{ba
        response = requests.get(url)
        data = response.json()

        if "conversion_rates" not in data:
            print(f"Error fetching data for base currency {base_currency
            continue

        rates = data["conversion_rates"]


        for target_currency, rate in rates.items():
            if target_currency != base_currency and target_currency in

                weight_direct = -math.log(rate)
                weight_reverse = math.log(rate)
                u = currency_to_index[base_currency]
                v = currency_to_index[target_currency]
                g.addEdge(u, v, weight_direct)
                g.addEdge(v, u, weight_reverse)

    return g, currency_to_index
```

We take a list of currencies as the input. Then, we call our API to obtain the appropriate rates, starting from the base currency. We add each of these edges with their corresponding weights to the graph structure so that the Bellman-Ford Algorithm can be completed.

This is the main framework for our program. There are some utility functions to print the appropriate cycles and take input, and those will be

available in the code appendix.

5 Results



This is what a complete graph of seven Currencies (USD, EUR, JPY, GBP, CAD, AED, INR) would look like. We ran these seven currencies through our algorithm given data from our API (Roughly on November 10th at 12 pm) and found that a sequence of trades highlighted in green ($USD -> AED -> JPY -> INR -> USD$) was profitable and had weight of -0.00014178, meaning that if we traded 1 USD at 12 pm on Nov. 10th into the cycle of trades highlighted we would have made 1.000142 USD.

We were able to identify several profitable cycles using our implementation and are unable to show them all here. Since the cycles that are profitable change on a day-to-day basis, there is no sense in displaying any that we

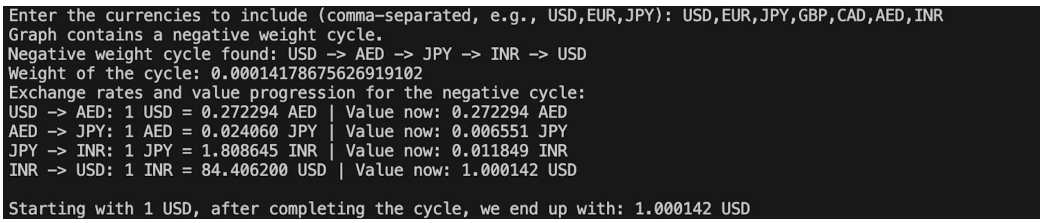


Figure 1: Output from algorithm as described

found, as trying the same cycle the next day may not lead to profitable cycles.

6 Conclusions

This study explores the graph-based implementation of arbitrage opportunities in foreign exchange markets. Using a representation using weighted edges, we created a graph out of a chosen foreign exchange market. Our results showed that negative cycles can be found depending on the current values for exchange rates. The profitability of the found cycles depends heavily on the magnitude of the value of the negative cycle.

Theoretically, our approach is sound, but there are some practical limitations. Future work would involve real-time exchange rate data and the optimization of algorithms for larger sets of currencies. We would also analyze fees and other areas that were ignored for the sake of simplicity.

6.1 Limitations

There are some limitations to our particular implementation. When we convert weights to negative logs, we are often left with floating point cutoffs and small approximations, which can accumulate into larger errors and even affect the profitability of the algorithm, given that the margins are already small. This means that the algorithm could miss or falsely count a negative weight cycle and could result in a loss of profit or opportunity for profit. Another limitation is that with more currencies we add to our graph, the run time and computation process becomes larger and larger, and because of this time delay and with the fact that exchange rates are constantly changing,

by the time we would have found the cycle for the trade, the trade's profitability may have collapsed because the rates would have changed at that moment. This is because arbitrage opportunities typically have a time interval in which traders can pursue a profit (Akram, Rime, and Sarno, 251). This is why we would need something that also executes the trades in a trading platform as we look for cycles in the graph every instant to minimize time error delay between execution of the trades and the timespan of which the trade is profitable.

Profitable Cycle Weight

The profitability of a cycle is determined by the magnitude of the negative weight. The larger the magnitude, the more profitable the cycle. Some cycles have a very small magnitude, so the profitability per unit of currency will be low. Thus, to make a profit, large amounts of currency must be exchanged through several currencies, which may be impractical. Another consideration is that there may be fees for exchanging currencies, depending on the method of transfer. The cycle must be profitable to a certain magnitude to overcome the fees.

API Limitations

The API that we chose updates currencies daily, but exchange rates change rapidly throughout the day. To accurately determine which cycles are actually profitable, an API that provides real-time data by the minute would be required.

6.2 Bellman-Ford Specifics

The Bellman-Ford algorithm can solve the same problems as Dijkstra's algorithm, but takes slower to solve these same problems. However, this algorithm can handle the case of negative weights. This algorithm can detect negative cycles in such graphs. This is important because there is no notion of a shortest path between two vertices if a negative cycle exists. Any path can be made shorter simply by traversing the negative cycle.

The algorithm proceeds using relaxation, where existing approximations for distances are repeatedly improved by traversing new edges. The algorithm is slow, however, running in $O(|V| \cdot |E|)$ time, compared to Dijkstra's algorithm, which runs in $O((|V|+|E|) \cdot \log |V|)$. Our case will always take the worst time complexity since the number of edges on our complete digraph is maximum.

References

[1] Mitchel T. Keller and William T. Trotter (1986) *Applied Combinatorics.*

[2] Bellman-Ford Algorithm *https://en.wikipedia.org/wiki/Bellman-Ford_algorithm.*

[3] Python Graph Implementation *https://www.python.org/doc/essays/graphs/*

[4] Exchange Rate API *https://www.exchangerate-api.com/docs/standard-requests*

[5] Mele, Marco. "On the Inefficient Markets Hypothesis: Arbitrage on the Forex Market." International Journal of Economics 9.2 (2015): 111-122.

[6] Q. Farooq Akram, Dagfinn Rime, Lucio Sarno, Arbitrage in the foreign exchange market: Turning on the microscope,Journal of International Economics,Volume 76, Issue 2,2008, Pages 237-253, ISSN 0022-1996,

7 Appendix

Code implementation at *github.com/tarundevi/forex-arbitrage*

```
import requests
import math

class Graph:
    def __init__(self, vertices, index_to_currency):
        self.V = vertices
        self.graph = []
        self.index_to_currency = index_to_currency

    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    def printNegativeCycle(self, parent, start):
        cycle = []
        cycle_weight = 0
        current = start
        visited = set()

        # Traverse the parent chain to get the cycle
        while current not in visited:
            visited.add(current)
            cycle.append(current)  # Append currency index
            prev = parent[current]
            # Find the edge weight from prev to current
            for u, v, w in self.graph:
                if u == prev and v == current:
                    cycle_weight += w
                    break
            current = prev

        # Backtrack to find the actual start of the cycle
        cycle_start = current
        cycle = cycle[cycle.index(cycle_start):] + [cycle_start]
# Complete the cycle
```

```
        # Print the cycle and its total weight
        print("Negative weight cycle found:", "->
--------".join(self.index_to_currency[idx] for idx in cycle))
        print("Weight of the cycle:", cycle_weight)

        # Print the exchange rates and simulate the currency
        conversion
        self.printExchangeRates(cycle)

    def printExchangeRates(self, cycle):
        print("Exchange rates and
--------value progression for the negative cycle:")
        initial_value = 1.0
        value = initial_value

        for i in range(len(cycle) - 1):
            u = cycle[i]
            v = cycle[i + 1]
            currency_u = self.index_to_currency[u]
            currency_v = self.index_to_currency[v]

            # Find the weight for the edge u -> v
            for edge_u, edge_v, weight in self.graph:
                if edge_u == u and edge_v == v:
                    exchange_rate = math.exp(-weight)
                    value *= exchange_rate
                    print(f"{currency_u} -> {currency_v}: 1
--------------------{currency_u} = {exchange_rate:.6f}
--------------------{currency_v} | Value now: {value:.6f}
--------------------{currency_v}")
                    break


        final_currency = self.index_to_currency[cycle[0]]
        print(f"\nStarting with 1 {final_currency}, after
--------completing the cycle, we end up with: {value:.6f}
--------{final_currency}")
```

```
    def BellmanFord(self, src):
        dist = [float("Inf")] * self.V
        parent = [-1] * self.V
        dist[src] = 0


        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    parent[v] = u

        # Check for negative-weight cycles
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains a negative weight cycle.")
                self.printNegativeCycle(parent, v)
                return


        self.printArr(dist)


    def printArr(self, dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print("{0}\t\t{1}".format(self.index_to_currency[i],
            dist[i]))



def buildGraphForSelectedCurrencies(api_key, selected_currencies):
    currency_to_index = {currency: idx for idx, currency in
    enumerate(selected_currencies)}
index_to_currency = {idx: currency for currency, idx in
currency_to_index.items()}
    num_currencies = len(selected_currencies)
```

```
    g = Graph(num_currencies, index_to_currency)


    for base_currency in selected_currencies:

        url = f"https://v6.exchangerate-api.com/v6/{api_key}/latest/
--------{base_currency}"
        response = requests.get(url)
        data = response.json()

        if "conversion_rates" not in data:
            print(f"Error fetching data for base currency
------------{base_currency}")
            continue

        rates = data["conversion_rates"]


        for target_currency, rate in rates.items():
            if target_currency != base_currency and
            target_currency in selected_currencies:

                weight_direct = -math.log(rate)
                weight_reverse = math.log(rate)
                u = currency_to_index[base_currency]
                v = currency_to_index[target_currency]
                g.addEdge(u, v, weight_direct)
                g.addEdge(v, u, weight_reverse)

    return g, currency_to_index



if __name__ == "__main__":
    api_key = "API_KEY"
```

```
    selected_currencies = input("Enter the currencies to include
----(comma-separated, e.g., USD,EUR,JPY): ").split(",")
    selected_currencies = [currency.strip().upper() for
    currency in selected_currencies]


    g, currency_to_index = buildGraphForSelectedCurrencies(api_key,
    selected_currencies)


    if selected_currencies[0] in currency_to_index:
        g.BellmanFord(currency_to_index[selected_currencies[0]])
    else:
        print(f"{selected_currencies[0]} not found in currency index")
```