COMP611 Algorithm Design and Analysis

Assignment 1

Solving Currency Exchange Problems

Assignment Due Date: 16/09/2024

Name: Juchang Kim

Student ID: 22180242

Signature: 김주창

Index

Figures

Tables

1. Introduction

In this assignment, we aim to solve currency exchange problems using shortest path algorithms. The two key tasks are:

1. **Detecting arbitrage opportunities** in a system of currency exchanges using the Bellman-Ford algorithm.
2. **Finding the best conversion rate** between two currencies using the Floyd-Warshall algorithm.

A currency exchange graph is represented as a directed graph where each node corresponds to a currency, and the edges represent exchange rates between currencies. The goal is to analyse such graphs for arbitrage opportunities and identify the best conversion rates.

2. Before Starting Tasks

To complete the tasks in the assignment, we first need to gather real-time currency data, which will serve as the input for various calculations like currency conversion and arbitrage detection. We are using the **api.exchangeratesapi.io** API to retrieve the latest currency exchange rates for six currencies: **NZD (New Zealand Dollar), USD (US Dollar), CAD (Canadian Dollar), AUD (Australian Dollar), GBP (British Pound), and EUR (Euro)**. The currency data will be for the date 27/08/2024, ensuring consistency for testing purposes.

Before running the Python script, you need to install the necessary libraries. You can install the **requests** library by running the following command: "**pip install requests**"

In the first part, we import the necessary Python libraries:

```
1   import requests
2   import math
```

Figure 2.1 Import Python Libraries

requests: This library is used to make HTTP requests to the API to fetch real-time currency data.

math: The math.log() function is used to calculate the natural logarithm of exchange rates for detecting arbitrage opportunities later in the assignment.

Function: get_exchange_rates(api_key, base_currency)

This function retrieves exchange rate data from the **api.exchangeratesapi.io** API.

```python
def get_exchange_rates(api_key, base_currency='EUR'):
    url = 'https://api.exchangeratesapi.io/v1/2024-08-27'
    params = {
        'access_key': api_key,
        'base': base_currency,

        'symbols': 'NZD,USD,CAD,AUD,GBP,EUR'  # Request all available currencies
    }

    response = requests.get(url, params=params)
    data = response.json()

    if response.status_code == 200:
        return data
    else:
        raise Exception(f"Error fetching data: {data.get('error', 'Unknown error')}")
```

Figure 2.2 get_exchange_rates Function

How It Works:

1. **API Key**: The API requires an authentication key (api_key) to access the service. In practice, you would replace the placeholder key with your own.
2. **Base Currency**: The API will return exchange rates based on a specific **base currency**. By default, we set the base currency to **EUR** (Euro), meaning that the exchange rates will be calculated relative to 1 Euro.
3. **URL and Parameters**: The API URL is constructed to fetch the exchange rates for 27/08/2024. The parameters include the base currency and a list of currencies for which we want to retrieve exchange rates.
4. **API Call**: The function sends a **GET** request to the API using the requests.get() function. If the request is successful (status code 200), the response data is returned. Otherwise, an error message is displayed.

Function: create_conversion_table(rates)

This function creates a currency conversion table based on the real-time data fetched from the API.

```python
def create_conversion_table(rates):
    currencies = list(rates['rates'].keys())
    conversion_data = []

    for currency_from in currencies:
        row = []
        for currency_to in currencies:
            if currency_from == currency_to:
                row.append(1.0)
            else:
                rate_from = rates['rates'].get(currency_from, 1)
                rate_to = rates['rates'].get(currency_to, 1)
                row.append((rate_to / rate_from))
        conversion_data.append(row)

    return conversion_data
```

Figure 2.3 create_conversion_table Function

How It Works:

1. **List of Currencies**: The currencies list is created by extracting the currency codes (e.g., NZD, USD) from the data returned by the API.
2. **Conversion Table**: A **2D list** (table) is constructed, where each element represents the exchange rate from one currency to another. For example, the value at position [0][1] in the table might represent the conversion rate from **NZD to USD**.
3. **Self-Conversion**: If the currency_from and currency_to are the same, the conversion rate is set to 1, as converting from a currency to itself should always yield 1.
4. **Other Conversions**: For different currencies, the conversion rate is calculated by dividing rate_to by rate_from. This gives the conversion rate between two currencies.
5. The final table is returned, which can be used in further calculations.

Main Code Execution:

```python
# Execution
api_key = 'a62c4214f8df13af6977b9a35d8f9fbf'  # Replace with your actual API key
try:
    rates = get_exchange_rates(api_key)
    test1 = create_conversion_table(rates)

    # Loop through the 2D array and print each row
    print("Currency Conversion Array for test1:")
    for i in range(len(test1)):
        print(test1[i])

    # Create logarithm scale 2D array
    test1_logscale = [[-math.log(rate) for rate in row] for row in test1]
    print("\nNegative Logarithm Scale Array for test1:")
    for i in range(len(test1_logscale)):
        print(test1_logscale[i])


except Exception as e:
    print(e)
```

Figure 2.4 Main Execution of Currency Conversion Rates

How It Works:

1. **API Call**: The function get_exchange_rates(api_key) is called with an API key to fetch real-time data. The data is stored in the rates variable.
2. **Conversion Table**: The function create_conversion_table(rates) is used to generate the conversion table (a 2D array), which contains the conversion rates between the six currencies (NZD, USD, CAD, AUD, GBP, EUR).
3. **Printing the Data**: The script prints both the **currency conversion table** and the **negative logarithmic scale table**. The latter is used for detecting arbitrage opportunities in further tasks. The -math.log() function is used to calculate the negative logarithm of each exchange rate in the table.

The output of execution with currency conversion rates by API

```
Currency Conversion Array for test1:
[1.0, 0.6242247835816354, 0.8394642329594796, 0.91964164547765O5, 0.4708903966148208, 0.5584629758593209]
[1.6019870186221488, 1.0, 1.3448108037986857, 1.4732539778395084, 0.754360302570778, 0.8946504377077266]
[1.1912359821151182, 0.7435990231304664, 1.0, 1.0955102187445322, 0.5609415840800336, 0.6652611915227097]
[1.0873800734422072, 0.6787695910154445, 0.9128166792875851, 1.0, 0.512036834054254, 0.607261511704662]
[2.1236364283257627, 1.3256264898777264, 1.7827168253892953, 1.9529844993417853, 1.0, 1.1859723194060652]
[1.790629, 1.117755, 1.503169, 1.646737, 0.84319, 1.0]

Negative Logarithm Scale Array for test1:
[-0.0, 0.4712447453812192, 0.1749914085425314, 0.08377120060519222, 0.7531299156232731, 0.5825669547753659]
[-0.47124474538121924, -0.0, -0.2962533368386879, -0.3874735447760027, 0.28188517024205384, 0.11132220939414661]
[-0.1749914085425314, 0.2962533368386878, -0.0, -0.09122020793733915, 0.5781385070807417, 0.40757554623283443]
[-0.08377120060519221, 0.38747354477602713, 0.09122020793733922, -0.0, 0.6693587150180809, 0.4987957541701737]
[-0.7531299156232731, -0.2818851702420539, -0.5781385070807417, -0.669358715018081, -0.0, -0.17056296084790737]
[-0.5825669547753658, -0.11132220939414657, -0.40757554623283443, -0.49879575417017363, 0.17056296084790729, -0.0]
```

Figure 2.5 Output of the Conversion Rates and Logarithm Scale Array

To represent table which is more visible

Currency Conversion Rates Table of Test1

From\To	NZD	USD	CAD	AUD	GBP	EUR
NZD	1	0.6242247835816354	0.8394642329594796	0.9196416454776505	0.4708903966148208	0.5584629758593209
USD	1.6019870186221488	1	1.3448108037986857	1.4732539778395084	0.75436	0.8946504377077266
CAD	1.1912359821151182	0.7435990231304664	1	1.0955102187445322	0.5609415840800336	0.6652611915227097
AUD	1.0873800734422072	0.6787695910154445	0.9128166792875851	1	0.512037	0.607262
GBP	2.1236364283257627	1.3256264898777264	1.7827168253892953	1.9529844993417853	1	1.1859723194060652
EUR	1.790629	1.117755	1.503169	1.646737	0.84319	1

Table 2.1 Currency Conversion Rates Named test1

Negative Logarithm Scale Table of Test1

From\To	NZD	USD	CAD	AUD	GBP	EUR
NZD	0	0.4712447453812192	0.1749914085425314	0.0837712006051922	0.7531299156232731	0.5825669547753659
USD	−0.471244745381219	0	−0.296253336838687	−0.387473544776027	0.2818851702420538 4	0.1113222093941466 1
CAD	−0.174991408542531	0.2962533368386878	0	−0.091220207 9373391	0.5781385070807417	0.4075755462328344 3
AUD	−0.083771200 6051921	0.3874735447760271 3	0.0912202079373392	0	0.6693587150180809	0.4987957541701737
GBP	−0.753129915623273	−0.281885170242053	−0.578138507080741	−0.669358715018081	0	−0.1705629608479 07
EUR	−0.582566954775365	−0.111322209394146	−0.407575546232834	−0.498795754170173	0.1705629608 4790729	0

Table 2.2 Negative Logarithm Scale Table of test1

So, the preparation of tasking of the assignment is done then the next part is detecting arbitrage opportunities.

3. Task 1: Detecting Arbitrage Opportunities

- ## Building Algorithm to Detect Arbitrage Opportunities

  - ### Problem Description

    In currency exchange, arbitrage refers to the process of buying and selling currencies to exploit price differences, leading to a profit without any net investment. This can be modelled as detecting a cycle in the currency exchange graph where the product of exchange rates exceeds 1, i.e., $r_{v_0 v_1} \times r_{v_1 v_2} \times \cdots \times r_{v_{k-1} v_0} > 1$

    To detect arbitrage, we transform the exchange rates by taking the negative logarithm of each rate. In this transformed graph, an arbitrage opportunity exists if there is a negative-weight cycle which is less than zero. The Bellman-Ford algorithm is employed to detect such cycles.

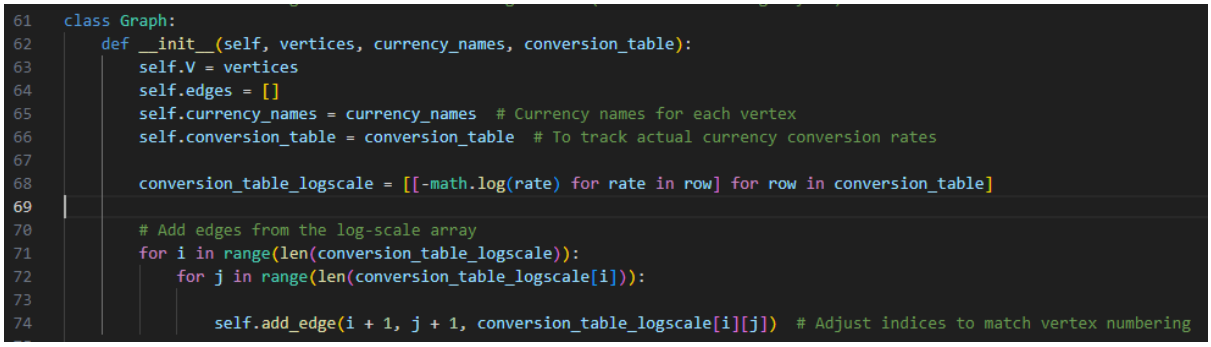
  - ### Solution Approach

1. **Building Bellman-Ford Algorithm In Graph Class**: We initiate graph class which can allow to several definitions. The each definition has function such as initiating graph, adding edges, Bellmen Ford algorithm, print cycle and so on.
2. **Input Data Transformation for Graph Construction**: We define what kind of data will be input and make a graph which has test data.
3. **Test the Algorithm with Test Data and Analysis of Result:** We run the algorithm with test then analyse the outcome of the algorithm

  - ### Implementation

    The Graph class implements the Bellman-Ford algorithm. It stores the currencies as vertices and their exchange rates as directed edges. The find arbitrage function runs the Bellman-Ford algorithm for each vertex and prints any arbitrage cycles found.

    1. Graph Class Definition



```
class Graph:
    def __init__(self, vertices, currency_names, conversion_table):
        self.V = vertices
        self.edges = []
        self.currency_names = currency_names  # Currency names for each vertex
        self.conversion_table = conversion_table  # To track actual currency conversion rates

        conversion_table_logscale = [[-math.log(rate) for rate in row] for row in conversion_table]

        # Add edges from the log-scale array
        for i in range(len(conversion_table_logscale)):
            for j in range(len(conversion_table_logscale[i])):

                self.add_edge(i + 1, j + 1, conversion_table_logscale[i][j])  # Adjust indices to match vertex numbering
```

Figure 3.1 Definition of Graph Class

Purpose: The Graph class represents the currency exchange system, where each **vertex** corresponds to a **currency**, and each **edge** represents the exchange rate between two currencies (converted to a logarithmic scale).

Parameters:

- vertices: The number of vertices (currencies) in the graph.
- currency_names: A list containing the names of the currencies (e.g., ["NZD", "USD", "CAD", "AUD", "GBP", "EUR"]).
- conversion_table: The table of actual exchange rates between currencies used for reference.

Attributes:

- self.V: The total number of vertices (currencies) in the graph.
- self.edges: A list that will store the edges in the form of tuples. Each tuple contains three components:
    - v: The source vertex (currency).
    - u: The target vertex (currency).
    - w: The weight, which represents the negative logarithm of the exchange rate between the two currencies.
- self.currency_names: A list of currency names for each vertex.
- self.conversion_table: A table to track the actual conversion rates between currencies, allowing comparison between conversion rate products and arbitrage opportunities.
- conversion_table_logscale**:** A transformed version of the conversion table where each value is converted into its negative logarithmic scale. This is used for easier calculation of potential arbitrage cycles.

2. **add_edge Function**

```
67      def add_edge(self, v, u, w):
68          self.edges.append((v, u, w))
```

Figure 3.2 add_edge Function

Purpose: Adds a directed edge to the graph between two currencies with the associated weight (w), which is the negative logarithm of the exchange rate.

Parameters:

- v: Source vertex (currency).
- u: Target vertex (currency).
- w: Weight of the edge, representing -log(exchange rate) between currency v and currency u.

3. **find_arbitrage Method**

```
70  def find_arbitrage(self):
71      # Run Bellman-Ford from each vertex
72      for src in range(1, self.V + 1):
73          print(f"Checking arbitrage opportunities starting from: {self.currency_names[src-1]}")
74          self.bellman_ford(src)
75
```

Figure 3.3 find_arbitrage Function

Purpose: Loops through each vertex (currency) in the graph and applies the **Bellman-Ford algorithm** starting from each one to detect arbitrage cycles.

How It Works:

- For each currency (src), the method invokes bellman_ford to look for negative-weight cycles that start and end at the currency.
- It prints the currency name before checking for arbitrage opportunities.

4. **bellman_ford Function**

```
76      def bellman_ford(self, src):
77          # Initialize distances and predecessors
78          d = [float("Inf")] * (self.V + 1)  # Distance array
79          predecessor = [-1] * (self.V + 1)  # To track the path
80
81          d[src] = 0
82
83          # Relax edges |V| - 1 times
84          for _ in range(self.V - 1):
85              for v, u, w in self.edges:
86                  if d[v] + w < d[u]:
87                      d[u] = d[v] + w
88                      predecessor[u] = v
89
90          # Check for negative-weight cycles
91          for v, u, w in self.edges:
92              if d[v] + w < d[u]:
93                  self.print_cycle(predecessor, u, src)
94                  return  # Arbitrage found, no need to continue
95
96          print("There is no arbitrage opportunity")
```

Figure 3.4 bellmen_ford Function

Purpose: The Bellman-Ford algorithm is used to find the shortest path between a source vertex (src) and all other vertices. It also detects negative-weight cycles (which represent arbitrage opportunities).

Key Steps:

1. **Initialization**:
    - d: Distance array initialized with infinity (Inf). The distance to the source (src) is set to 0.
    - predecessor: Tracks the previous vertex in the shortest path.
2. **Relaxation**: In each iteration, it updates the distance to each vertex by checking all edges. The edges are relaxed **V-1 times** (where V is the number of vertices), ensuring the shortest paths are found.
3. **Cycle Detection**: After all edges have been relaxed, the code checks for negative-weight cycles. If d[v] + w < d[u], then a negative cycle exists, which represents an arbitrage opportunity. The method calls print_cycle to print the details of the cycle.

5. print_cycle Function

```python
    def print_cycle(self, predecessor, start, source):
        # To find the negative-weight cycle path
        cycle = []
        visited = set()
        current = start

        # Follow predecessors to identify the cycle
        while current not in visited:
            visited.add(current)
            current = predecessor[current]

        # Reconstruct the cycle
        cycle_start = current
        current = cycle_start

        # Include source vertex at the beginning and end to show full cycle
        cycle.append(source)
        while True:
            cycle.append(current)
            current = predecessor[current]
            if current == cycle_start:
                cycle.append(current)
                break

        cycle.append(source)  # Append source again to close the cycle

        # remove the duplicated vertex in the cycle
        i = 0
        while i < len(cycle) - 1:
            if cycle[i] == cycle[i + 1]:
                del cycle[i + 1]  # Remove the duplicate by index
            else:
                i += 1  # Only increment if no removal to avoid skipping elements
```

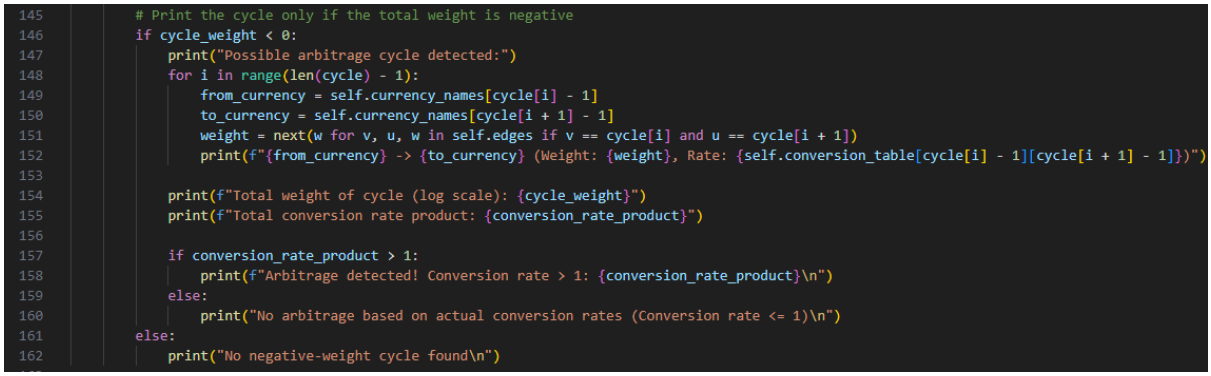
Figure 3.5 Parts of print_cycle Function

Purpose: The print_cycle method reconstructs and prints the arbitrage cycle found by the Bellman-Ford algorithm.

How It Works:

1. **Cycle Reconstruction**: It follows the predecessor array to trace back the cycle starting from the detected vertex (start). This allows the algorithm to form a list of vertices involved in the arbitrage.
2. **Source Inclusion**: The method ensures that the **source vertex** is included at the start and end of the cycle, showing the full loop.
3. **Removing Duplicates**: The code ensures that no duplicated vertices are included in the cycle.

6. **Cycle Weight Calculation and Output In print_cycle Function**



```
        # Print the cycle only if the total weight is negative
        if cycle_weight < 0:
            print("Possible arbitrage cycle detected:")
            for i in range(len(cycle) - 1):
                from_currency = self.currency_names[cycle[i] - 1]
                to_currency = self.currency_names[cycle[i + 1] - 1]
                weight = next(w for v, u, w in self.edges if v == cycle[i] and u == cycle[i + 1])
                print(f"{from_currency} -> {to_currency} (Weight: {weight}, Rate: {self.conversion_table[cycle[i] - 1][cycle[i + 1] - 1]})")

            print(f"Total weight of cycle (log scale): {cycle_weight}")
            print(f"Total conversion rate product: {conversion_rate_product}")

            if conversion_rate_product > 1:
                print(f"Arbitrage detected! Conversion rate > 1: {conversion_rate_product}\n")
            else:
                print("No arbitrage based on actual conversion rates (Conversion rate <= 1)\n")
        else:
            print("No negative-weight cycle found\n")
```

Figure 3.6 Cycle Weight Calculation and Output in print_cycle Function

Purpose: Once an arbitrage cycle is identified (i.e., a negative-weight cycle), this section of the code calculates the total weight of the cycle and checks whether there is an arbitrage opportunity. A negative total weight (in log scale) indicates an arbitrage opportunity, but to confirm it, the actual product of the conversion rates along the cycle must also be greater than 1.

Cycle Detection: If a negative-weight cycle is detected, the print_cycle function prints the details of the arbitrage cycle, including**:**

- **From and To Currencies:** The currencies involved in each step of the cycle.
- **Weight:** The negative logarithm of the exchange rate between the currencies.
- **Actual Conversion Rate:** The real conversion rate between the currencies (fetched from self.conversion_table).

Total Weight Calculation:

- **Log Scale Weight**: The total log scale weight of the cycle is printed to confirm the presence of a negative-weight cycle.

Conversion Rate Product:

- **Conversion Rate Product:** The actual product of the conversion rates along the arbitrage cycle is calculated.
- If the product is greater than 1, the system prints a message confirming the arbitrage opportunity.
- If the product is less than or equal to 1, it prints that no arbitrage exists based on the actual conversion rates.

- ## Task1: Testing to Detect Arbitrage Opportunities with Bellmen Ford Algorithm (First Test)

  - ### Input First Test Case

    To run the functions, the test need to be input and the test case is used the data set from the Before Starting Task section.

    Currency Conversion Rates Table for Test of the Test

```
print("Task1 - Detect Arbitrage with Bellman-Ford Algorithms (Test arbitrage cycle)\n")
# Example usage
test1 = [[1.0, 0.6242247835816354, 0.8394642329594796, 0.9196416454776505, 0.4708903966148208, 0.5584629758593209],
        [1.6019870186221488, 1.0, 1.3448108037986857, 1.4732539778395084, 0.754360302570778, 0.8946504377077266],
        [1.1912359821151182, 0.7435990231304664, 1.0, 1.0955102187445322, 0.5609415840800336, 0.6652611915227097],
        [1.0873800734422072, 0.6787695910154445, 0.9128166792875851, 1.0, 0.512036834054254, 0.607261511704662],
        [2.1236364283257627, 1.3256264898777264, 1.7827168253892953, 1.9529844993417853, 1.0, 1.1859723194060652],
        [1.790629, 1.117755, 1.503169, 1.646737, 0.84319, 1.0]]

currency_names = ["NZD", "USD", "CAD", "AUD", "GBP", "EUR"]
```

Figure 3.7 Currency Conversion Rates Array for Test

First, the graph is constructed by parameters of number of vertices which are number of currencies, currency names, and the currency conversion rate which is test1.

```
graph_test1 = Graph(6, currency_names, test1)

# Find arbitrage opportunities
graph_test1.find_arbitrage()
```

Figure 3.8 Input First Test in the Graph Class

Lastly, the graph using the function which is find_arbitrage(), it will show the result.

  - ### Output of the First Test case

    The output shows all currencies if there is an arbitrage opportunity or not. If the negative cycle is found, represents the pathway, sum of negative weight cycle, total conversion rate, check the arbitrage condition which are negative weight cycle is less than zero and total conversion rate is more than 1. If the two conditions do not meet at least one of them, the result is no arbitrage opportunity.

```
Task1 - Detect Arbitrage with Bellman-Ford Algorithms (First Test)

Checking arbitrage opportunities starting from: NZD
No negative-weight cycle found

Checking arbitrage opportunities starting from: USD
Possible arbitrage cycle detected:
USD -> EUR (Weight: 0.11132220939414661, Rate: 0.8946504377077266)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.84319)
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.1859723194060652)
EUR -> USD (Weight: -0.11132220939414657, Rate: 1.117755)
Total weight of cycle (log scale): -4.163336342344337e-17
Total conversion rate product: 1.0
No arbitrage based on actual conversion rates (Conversion rate <= 1)

Checking arbitrage opportunities starting from: CAD
No negative-weight cycle found

Checking arbitrage opportunities starting from: AUD
Possible arbitrage cycle detected:
AUD -> GBP (Weight: 0.6693587150180809, Rate: 0.512036834054254)
GBP -> CAD (Weight: -0.5781385070807417, Rate: 1.7827168253892953)
CAD -> EUR (Weight: 0.407575546232834 43, Rate: 0.6652611915227097)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.84319)
GBP -> AUD (Weight: -0.669358715018081, Rate: 1.9529844993417853)
Total weight of cycle (log scale): -1.1102230246251565e-16
Total conversion rate product: 0.9999999999999999
No arbitrage based on actual conversion rates (Conversion rate <= 1)

Checking arbitrage opportunities starting from: GBP
Possible arbitrage cycle detected:
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.1859723194060652)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.84319)
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.1859723194060652)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.84319)
Total weight of cycle (log scale): -1.6653345369377348e-16
Total conversion rate product: 1.0
No arbitrage based on actual conversion rates (Conversion rate <= 1)

Checking arbitrage opportunities starting from: EUR
Possible arbitrage cycle detected:
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.84319)
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.1859723194060652)
Total weight of cycle (log scale): -8.326672684688674e-17
Total conversion rate product: 1.0
No arbitrage based on actual conversion rates (Conversion rate <= 1)
```

Figure 3.9 Output of The First Test of Task1

- Key Results and Their Analysis

1. **Checking Arbitrage Opportunities Starting from NZD:**
    - **No negative-weight cycle found:**

        - The algorithm did not detect any negative-weight cycles starting from NZD, meaning there is no arbitrage opportunity from this currency based on the given exchange rates.

2. **Checking Arbitrage Opportunities Starting from USD:**
    - **Possible Arbitrage Cycle Detected:**
        - Cycle: USD → EUR → GBP → EUR → USD
        - Weights:
            - USD → EUR: 0.111322 (log scale), Rate: 0.8947
            - EUR → GBP: 0.170562, Rate: 0.8432
            - GBP → EUR: -0.170563, Rate: 1.186
            - EUR → USD: -0.111322, Rate: 1.1178
        - Total Weight: -4.16e-17 (log scale), which is very close to 0.
        - Conversion Rate Product: 1.0.
        - Conclusion: Although the total weight is negative (indicating a theoretical arbitrage opportunity), the product of the actual conversion rates is exactly 1, meaning that no real arbitrage opportunity exists. The round-trip conversion returns the exact starting value, not a profit.
3. **Checking Arbitrage Opportunities Starting from CAD:**
    - **No negative-weight cycle found:**
        - No arbitrage opportunity was detected when starting from CAD, as no negative-weight cycle was found.
4. **Checking Arbitrage Opportunities Starting from AUD:**
    - **Possible Arbitrage Cycle Detected:**
        - Cycle: AUD → GBP → CAD → EUR → GBP → AUD
        - Weights:
            - AUD → GBP: 0.669359 (log scale), Rate: 0.512
            - GBP → CAD: -0.578139, Rate: 1.7827
            - CAD → EUR: 0.407576, Rate: 0.6653
            - EUR → GBP: 0.170563, Rate: 0.8432

        - GBP → AUD: -0.669359, Rate: 1.953
    - Total Weight: -1.11e-16 (log scale), which is negative but close to 0.
    - Conversion Rate Product: 0.9999999999999999 (approximately 1).
    - Conclusion: While the total weight is negative, suggesting the possibility of arbitrage, the conversion rate product is nearly 1.0, meaning no profitable arbitrage opportunity exists based on the actual conversion rates.

5. **Checking Arbitrage Opportunities Starting from GBP:**
    - **Possible Arbitrage Cycle Detected:**
        - Cycle: GBP → EUR → GBP → EUR → GBP
        - Weights:
            - GBP → EUR: -0.170563, Rate: 1.186
            - EUR → GBP: 0.170563, Rate: 0.8432
        - Total Weight: -1.665e-16 (log scale), which is negative.
        - Conversion Rate Product: 1.0.
        - Conclusion: Despite detecting a negative-weight cycle, the actual conversion rate product is 1.0, indicating no arbitrage opportunity exists in practice.
6. **Checking Arbitrage Opportunities Starting from EUR:**
    - **Possible Arbitrage Cycle Detected:**
        - Cycle: EUR → GBP → EUR
        - Weights:
            - EUR → GBP: 0.170563, Rate: 0.8432
            - GBP → EUR: -0.170563, Rate: 1.186
        - Total Weight: -8.33e-17 (log scale), slightly negative.
        - Conversion Rate Product: 1.0.
        - Conclusion: Similar to previous cases, despite detecting a negative-weight cycle, the actual conversion rate product is exactly 1.0, meaning no arbitrage exists.

7. Summary of Result:

- **No Arbitrage Detected in Practice:** In all cases, while some negative-weight cycles were detected (which theoretically indicate arbitrage), the actual conversion rate product was 1.0 or very close to 1.0, meaning that no arbitrage opportunity exists in practice based on the provided exchange rates.
- **Floating-Point Precision:** The small negative log scale values (e.g., -4.16e-17) and the conversion rate product being 1.0 suggest that the cycles are so close to being neutral that floating-point precision might play a role. In real-world applications, conversion rates slightly above or below 1.0 are critical, but in this scenario, no clear arbitrage profit can be identified.

- ## Task1: Testing to Detect Arbitrage Opportunities with Bellmen Ford Algorithm (Second Test)

  - ### Input Second Test Case

    The first test case shows there is no arbitrage opportunity in the test case, the currency conversion rates are edited as rounded up in each rate to make different result.

```
print("Task 1. Detect Arbitrage with Bellman-Ford Algorithms (Second Test)")

test2 = [[round(test1[i][j], 4) for j in range(len(test1[i]))] for i in range(len(test1))]

test2_logscale = [[-math.log(test1[i][j]) for j in range(len(test2[i]))] for i in range(len(test2))]

# Loop through the 2D array and print each row
print("Currency Conversion Array (test2):")
for i in range(len(test2)):
    print(test2[i])

# Loop through the 2D array and print each row
print("\nNegative Logarithm Scale (test2_logscale):")
for i in range(len(test2_logscale)):
    print(test2_logscale[i])
```

Figure 3.10 Input Second test in the Graph Class

Then the negative logarithm scale array is used same function with first test. Also, to check the second test data, it prints once all the data.

```
Task 1. Detect Arbitrage with Bellman-Ford Algorithms (Second Test)
Currency Conversion Array (test2):
[1.0, 0.6242, 0.8395, 0.9196, 0.4709, 0.5585]
[1.602, 1.0, 1.3448, 1.4733, 0.7544, 0.8947]
[1.1912, 0.7436, 1.0, 1.0955, 0.5609, 0.6653]
[1.0874, 0.6788, 0.9128, 1.0, 0.512, 0.6073]
[2.1236, 1.3256, 1.7827, 1.953, 1.0, 1.186]
[1.7906, 1.1178, 1.5032, 1.6467, 0.8432, 1.0]

Negative Logarithm Scale (test2_logscale):
[-0.0, 0.4712447453812192, 0.17499140085425314, 0.08377120060519222, 0.7531299156232731, 0.5825669547753659]
[-0.47124474538121924, -0.0, -0.2962533368386879, -0.3874735447760027, 0.28188517024205384, 0.11132220939414661]
[-0.17499140085425314, 0.2962533368386878, -0.0, -0.09122020793733915, 0.5781385070807417, 0.40757554623283443]
[-0.0837712006051921, 0.38747354477602713, 0.09122020793733922, -0.0, 0.6693587150180809, 0.49879575417017363]
[-0.753129915623273, -0.2818851702420539, -0.5781385070807417, -0.669358715018081, -0.0, -0.17056296084790737]
[-0.5825669547753658, -0.11132220939414657, -0.40757554623283443, -0.49879575417017363, 0.17056296084790729, -0.0]
```

Figure 3.11 Printed the Second Test Data

Currency Conversion Rates Table of Second Test

From\To	NZD	USD	CAD	AUD	GBP	EUR
NZD	1	0.6242	0.8395	0.9196	0.4709	0.5585
USD	1.602	1	1.3448	1.4733	0.7544	0.8947
CAD	1.1912	0.7436	1	1.0955	0.5609	0.6653
AUD	1.0874	0.6788	0.9128	1	0.512	0.6073

GBP	2.1236	1.3256	1.7827	1.953	1	1.186
EUR	1.7906	1.1178	1.5032	1.6467	0.8432	1

Table 3.1 Currency Converion Rates Table of Test2

Negative Logarithm Scale Table of Second Test

From\To	NZD	USD	CAD	AUD	GBP	EUR
NZD	0	0.4712447453812192	0.1749914085425314	0.0837712006051921	0.7531299156232731	0.5825669547753659
USD	−0.4712447453812192	0	−0.2962533368386879	−0.3874735447760270	0.2818851702420539	0.1113222090939414661
CAD	−0.1749914085425314	0.2962533368386878	0	−0.0912202079373391	0.5781385070807417	0.4075755462328344
AUD	−0.0837712006051921	0.3874735447760271	0.0912202079373392	0	0.6693587150180809	0.4987957541701737
GBP	−0.7531299156232731	−0.2818851702420539	−0.5781385070807417	−0.6693587150180809	0	−0.1705629608479073
EUR	−0.5825669547753658	−0.1113222090939414657	−0.4075755462328344	−0.4987957541701737	0.1705629608479072	0

Table 3.2 Negative Logarithm Scale Table of Test2

Then, the number of the currencies, currency conversion rate array are input as parameters. When the graph initialised, weight of edges are input in the new graph then run the find_arbitrage() function.

```
200   # Create the Graph with Second Test
201   graph_test2 = Graph(6, currency_names, test2)
202
203
204   # Find arbitrage opportunities
205   print("\n")
206   graph_test2.find_arbitrage()
```

Figure 3.12 Setting up the new Graph Class and run the find_arbitrage Function

- ## Output of the Second Test case

The output same frame with the first test. If there is a negative weight cycle. It shows the pathway, sum of the negative weight cycle, total conversion rate. When the sum of negative weight cycle is less 0 and total conversion rate is more than 1. It defines that there is an arbitrage opportunity. Then every currency run it and checked.

```
Checking arbitrage opportunities starting from: NZD
No negative-weight cycle found

Checking arbitrage opportunities starting from: USD
Possible arbitrage cycle detected:
USD -> EUR (Weight: 0.11132220939414661, Rate: 0.8947)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.8432)
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.186)
EUR -> USD (Weight: -0.11132220939414657, Rate: 1.1178)
Total weight of cycle (log scale): -4.163336342344337e-17
Total conversion rate product: 1.000130863367232
Arbitrage detected! Conversion rate > 1: 1.000130863367232

Checking arbitrage opportunities starting from: CAD
No negative-weight cycle found

Checking arbitrage opportunities starting from: AUD
Possible arbitrage cycle detected:
AUD -> GBP (Weight: 0.6693587150180809, Rate: 0.512)
GBP -> CAD (Weight: -0.5781385070807417, Rate: 1.7827)
CAD -> EUR (Weight: 0.407575546232834 43, Rate: 0.6653)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.8432)
GBP -> AUD (Weight: -0.669358715018081, Rate: 1.953)
Total weight of cycle (log scale): -1.1102230246251565e-16
Total conversion rate product: 0.9999967535035269
No arbitrage based on actual conversion rates (Conversion rate <= 1)

Checking arbitrage opportunities starting from: GBP
Possible arbitrage cycle detected:
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.186)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.8432)
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.186)
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.8432)
Total weight of cycle (log scale): -1.6653345369377348e-16
Total conversion rate product: 1.0000704012390398
Arbitrage detected! Conversion rate > 1: 1.0000704012390398

Checking arbitrage opportunities starting from: EUR
Possible arbitrage cycle detected:
EUR -> GBP (Weight: 0.17056296084790729, Rate: 0.8432)
GBP -> EUR (Weight: -0.17056296084790737, Rate: 1.186)
Total weight of cycle (log scale): -8.326672684688674e-17
Total conversion rate product: 1.0000352
Arbitrage detected! Conversion rate > 1: 1.0000352
```

Figure 3.13 The Output of the find_arbitrage with Second Test

- ## Key Results and Their Analysis:

1. **Checking Arbitrage Opportunities Starting from NZD**:
    - **Result**: No negative-weight cycle found.
    - **Conclusion**: No arbitrage opportunity was detected from NZD. The exchange rates from NZD to other currencies do not form a cycle that results in a profitable arbitrage opportunity.
2. **Checking Arbitrage Opportunities Starting from USD**:
    - **Cycle**: USD → EUR → GBP → EUR → USD.
    - **Weights** and **Rates**:
        - USD → EUR: Weight = 0.111322, Rate = 0.8947.
        - EUR → GBP: Weight = 0.170563, Rate = 0.8432.
        - GBP → EUR: Weight = -0.170563, Rate = 1.186.
        - EUR → USD: Weight = -0.111322, Rate = 1.1178.
    - **Total Weight (Log Scale)**: -4.16e-17 (slightly negative).
    - **Total Conversion Rate Product**: 1.000130863367232 (slightly greater than 1).
    - **Conclusion**:
        - The negative total weight indicates the presence of a theoretical arbitrage opportunity.
        - The conversion rate product is slightly greater than 1 (1.00013), which confirms the presence of a small but real arbitrage opportunity starting from USD. This suggests that converting USD → EUR → GBP → EUR → USD would yield a small profit.
3. **Checking Arbitrage Opportunities Starting from CAD**:
    - **Result**: No negative-weight cycle found.
    - **Conclusion**: No arbitrage opportunity was detected starting from CAD. The conversion rates involving CAD do not form a profitable cycle.
4. **Checking Arbitrage Opportunities Starting from AUD**:
    - **Cycle**: AUD → GBP → CAD → EUR → GBP → AUD.
    - **Weights** and **Rates**:

        - AUD → GBP: Weight = 0.669359, Rate = 0.512.
        - GBP → CAD: Weight = -0.578139, Rate = 1.7827.
        - CAD → EUR: Weight = 0.407576, Rate = 0.6653.
        - EUR → GBP: Weight = 0.170563, Rate = 0.8432.
        - GBP → AUD: Weight = -0.669359, Rate = 1.953.
    - **Total Weight (Log Scale)**: -1.11e-16 (slightly negative).
    - **Total Conversion Rate Product**: 0.9999967535035269 (slightly less than 1).
    - **Conclusion**:
        - Despite the negative total weight indicating a potential arbitrage, the actual conversion rate product is slightly less than 1 (0.999996), which suggests that no arbitrage exists in practice. The round-trip conversion does not yield a profit and essentially returns the same value.
5. **Checking Arbitrage Opportunities Starting from GBP**:
    - **Cycle**: GBP → EUR → GBP → EUR → GBP.
    - **Weights** and **Rates**:
        - GBP → EUR: Weight = -0.170563, Rate = 1.186.
        - EUR → GBP: Weight = 0.170563, Rate = 0.8432.
    - **Total Weight (Log Scale)**: -1.665e-16 (slightly negative).
    - **Total Conversion Rate Product**: 1.0000704012390398 (slightly greater than 1).
    - **Conclusion**:
        - The negative total weight suggests a theoretical arbitrage opportunity.
        - The total conversion rate product of 1.00007 indicates a very small arbitrage opportunity, where converting GBP → EUR → GBP → EUR → GBP results in a slight profit.
6. **Checking Arbitrage Opportunities Starting from EUR**:
    - **Cycle**: EUR → GBP → EUR.
    - **Weights** and **Rates**:

        - EUR → GBP: Weight = 0.170563, Rate = 0.8432.
        - GBP → EUR: Weight = -0.170563, Rate = 1.186.
    - **Total Weight (Log Scale)**: -8.33e-17 (slightly negative).
    - **Total Conversion Rate Product**: 1.0000352 (slightly greater than 1).
    - **Conclusion**:
        - The negative total weight indicates an arbitrage opportunity.
        - The total conversion rate product of 1.0000352 indicates a small arbitrage opportunity, where converting EUR → GBP → EUR results in a slight profit.

Arbitrage Opportunities Detected: The algorithm successfully detected arbitrage opportunities for cycles starting from USD, GBP, and EUR. The conversion rate products in these cases were slightly greater than 1, indicating the presence of small arbitrage opportunities.

No Arbitrage from NZD and CAD: No profitable cycles were found when starting from NZD or CAD, indicating that there are no arbitrage opportunities from these currencies.

Conclusion and Findings with 2 Tests with Arbitrage Opportunities

After running two tests using the Bellman-Ford algorithm to detect arbitrage opportunities, the results provide insights into how theoretical and practical arbitrage scenarios differ. The first test used real-time exchange rates, while the second test involved rounding the rates to detect arbitrage more effectively. Here's an integrated conclusion based on the two tests:

- The **first test** with real exchange rates did not reveal practical arbitrage opportunities, with conversion rate products equal to or less than 1, even though negative-weight cycles were detected theoretically.
- The **second test**, using rounded exchange rates, detected minor arbitrage opportunities (conversion rate products slightly greater than 1) in cycles involving currencies like **USD, GBP, and EUR**.
- Overall, while **theoretical arbitrage** can be detected, practical arbitrage remains elusive in real-time data due to small margins. The rounding of rates in **Test 2** introduced slight arbitrage opportunities, highlighting the delicate balance and precision required in currency exchange markets.

4. Task 2: Finding the Best Conversion Rate

- ## Building Algorithm to Find Best Conversion Rate

  - ### Problem Description

    The goal is to find the best conversion rate between two currencies. This can be achieved by considering both direct exchange rates and possible routes via intermediate currencies. The problem is modeled as finding the shortest path in the currency exchange graph.

  - ### Solution Approach

1. **Build Floyd-Warshall Algorithm**: This algorithm computes the shortest paths between all pairs of currencies, allowing us to determine the best conversion rate between any two currencies. Additionally, there are more definition which help to tracking path way, calculating conversion rates and so on.
2. **Data Transformation And Run the Test data**: The data need to be inserted in the algorithm and then run it.
3. **Conversion Path and Print the result**: If there is the shortest path is not the direct path, it represents of the pathway, weight, total conversion rate via intermediate node (currency)
4. **Analysis of the output:** it need to be discuss why it has shortest path which is not the direct path how to prove it.

  - ### Implementation

1. **Floyd-Warshall Algorithm Definition**

   Purpose: The floyd_warshall_with_path function implements the Floyd-Warshall algorithm, which finds the shortest paths between all pairs of vertices in a graph. In the context of currency exchange, the vertices represent currencies, and the edges represent the negative logarithms of the exchange rates. This function helps to identify the best conversion path between two currencies, and it tracks the intermediate steps for reconstructing the optimal path.

```python
# Task 2. Finding Best Conversion Rate with Floyd-Warshall Algorithms

def floyd_warshall_with_path(graph):
    n = len(graph)

    # Initialize distance and next matrices
    dist = [[float('inf')] * n for _ in range(n)]
    next_node = [[-1] * n for _ in range(n)] # To tracking the path

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0  # Distance to itself is 0
            elif graph[i][j] != 0:
                dist[i][j] = graph[i][j]
                next_node[i][j] = j  # Initialize next node to j if there is an edge

    # Main loop for the Floyd-Warshall algorithm
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    next_node[i][j] = next_node[i][k]  # Update the next node in the path

    return dist, next_node
```

Figure 4.1 Definition of the floyd_warshall_with_path Function

Parameters:

- **graph**:
  - A 2D array (or adjacency matrix) representing the graph, where graph[i][j] is the cost (or weight) of converting from currency i to currency j. The weights are typically the negative logarithms of the exchange rates.

Attributes:

- **n**:
  - The number of vertices (currencies) in the graph, derived from the dimensions of the input graph.
- **dist**:
  - A 2D array that stores the shortest path distances between each pair of vertices. It is initialized with infinity (float('inf')) for all pairs except for self-loops (distance from a vertex to itself), which are set to 0.
- **next_node**:
  - A 2D array used to store the next vertex along the shortest path from vertex i to vertex j. This allows for path reconstruction later. It is initialized with -1 to indicate that no path exists initially.

Key Functionality:

1. **Initialization of Distance and Next Matrices:**

   The dist matrix is initialized such that dist[i][j] is set to infinity for all pairs, except for the diagonal where dist[i][i] is set to zero (since the distance from a vertex to itself is zero). The next_node matrix is initialized with -1 to indicate that no path has been found yet.

2. **Setting Initial Distance and Path Information:**
   For every edge in the input graph, the corresponding entry in the dist matrix is updated to the weight of the edge, and the next_node[i][j] is set to j, meaning the path from vertex i to vertex j directly leads to j.

3. **Main Loop for the Floyd-Warshall Algorithm:**
   This triple nested loop iterates over all pairs of vertices (i, j) and checks whether including an intermediate vertex k would result in a shorter path from i to j. If so, the function updates the dist[i][j] with the shorter path and sets next_node[i][j] to indicate that the path from i to j now includes the vertex k.

4. **Return Value:**
   The function returns two matrices:
   - **dist**: The shortest path distances between every pair of vertices.
   - **next_node**: The next vertex in the shortest path from any vertex i to vertex j, which is used for path reconstruction.

2. Path Reconstruction Using get_path Function

Purpose: The get_path function reconstructs the shortest path between two vertices (currencies) in a graph using the next_node matrix, which is generated from the Floyd-Warshall algorithm. It ensures the retrieval of the optimal sequence of exchanges between two currencies, allowing for a clear understanding of the path taken to achieve the best conversion rate.

```
235  def get_path(next_node, start, end):
236      # Reconstruct the path from start to end using the next_node matrix
237
238      path = [start]
239      visited = set()  # To detect cycles
240
241      while start != end:
242          if start in visited:  # Cycle detection
243              return None  # Cycle detected; no valid path
244          visited.add(start)
245
246          start = next_node[start][end]
247
248          path.append(start)
249
250      return path
```

Figure 4.2 Definition of get_path Function

Parameters:

- **next_node**:
  A 2D matrix that stores the next vertex along the shortest path between each pair of vertices. It was generated during the Floyd-Warshall algorithm's execution.
- **start**:
  The starting vertex (currency) in the path.
- **end**:
  The target vertex (currency) in the path.

Attributes:

- **path**:
  A list that stores the sequence of vertices (currencies) from the start to the end. It begins with the starting vertex and is populated as the function traverses through the path.
- **visited**:
  A set used to detect cycles. It stores the vertices that have already been visited in the current traversal. If a vertex is visited more than once, the function assumes a cycle is detected, which indicates no valid path exists.

Key Functionality:

1. **Initialization:**
   The function begins by initializing the path list with the starting vertex, and the visited set is used to keep track of vertices already traversed to detect cycles.
2. **Path Traversal and Cycle Detection:**
   The main loop runs until the starting vertex becomes the target vertex (start != end). During each iteration:
   - If the current vertex has already been visited, a cycle is detected, and the function returns None, indicating that no valid path exists.
   - The current vertex is added to the visited set to ensure that no cycle will occur in subsequent iterations.
   - The start is updated to the next vertex in the path using the next_node matrix (next_node[start][end]), and this vertex is appended to the path list.
3. **Return Value:**
   Once the traversal is complete, and the path from start to end is fully constructed, the function returns the path list, which contains the sequence of vertices (currencies) from the starting vertex to the target vertex.

3. best_conversion_rate Function Definition

```
252  def best_conversion_rate(n, rates, source_index, target_index):
253      # Convert rates to weights using logarithms
254      weights = [[-math.log(rates[i][j]) for j in range(n)] for i in range(n)]
255
256      # Apply the Floyd-Warshall algorithm with path tracking
257      dist, next_node = floyd_warshall_with_path(weights)
258
259      path = get_path(next_node, source_index, target_index)
260
261      # Direct conversion rate
262      direct_rate = rates[source_index][target_index]
263
264      # Compute detoured conversion rate by multiplying rates along the path
265      if path:
266          rate = 1.0
267          sum_of_logs = 0.0  # To store the sum of negative logarithms
268          for i in range(len(path) - 1):
269              rate *= rates[path[i]][path[i + 1]]
270              sum_of_logs += weights[path[i]][path[i + 1]]  # Sum of negative logarithms
271      else:
272          rate = direct_rate  # Fallback if no detoured_rate path found
273          sum_of_logs = weights[source_index][target_index]
274
275      # Determine the best path
276      if rate > direct_rate:
277          path = get_path(next_node, source_index, target_index)
278          path_names = [currencies[i] for i in path] if path else None
279      else:
280          path_names = None
281
282      return direct_rate, rate, path_names, sum_of_logs, weights, path
```

Figure 4.3 Definition of best_conversion_rate Function

Purpose:

The best_conversion_rate function is designed to calculate and compare the direct conversion rate and the best conversion rate via intermediate currency between two currencies which are source and target using the Floyd-Warshall algorithm. It returns both the direct conversion rate and the optimal conversion rate via intermediate currencies (if any), along with the path taken and the logarithmic scale comparison of both rates.

Parameters:

- **n**:
  The total number of currencies (vertices) in the system.
- **rates**:
  A 2D matrix containing the exchange rates between each pair of currencies.
- **source_index**:
  The index of the source currency for which the conversion rate is being calculated.

- **target_index**:
  The index of the target currency to which the source currency will be converted.

Attributes:

- **weights**:
  A 2D matrix where each entry is the negative logarithm of the corresponding exchange rate in the rates matrix. This transformation allows for easier detection of the best conversion path.
- **dist, next_node**:
  The dist matrix stores the shortest paths between all pairs of currencies in terms of negative logarithms, and the next_node matrix stores the next vertex (currency) to visit in the shortest path. These are calculated using the floyd_warshall_with_path function.
- **path**:
  A list that stores the sequence of currencies in the optimal roundabout path (if one exists) from the source currency to the target currency. It is obtained by using the get_path function, which reconstructs the path from the next_node matrix.
- **direct_rate**:
  The direct conversion rate between the source and target currencies. This is directly fetched from the rates matrix.
- **rate**:
  The total conversion rate for the roundabout path. If a valid path exists, the function multiplies the rates along this path. If no valid path exists, rate is set to the direct conversion rate.
- **sum_of_logs**:
  A sum of the negative logarithms of the exchange rates along the roundabout path. This is used to compare the roundabout conversion rate with the direct conversion rate in terms of logarithmic scale.

Key Functionality:

1. **Convert Exchange Rates to Logarithms:**
   The function begins by transforming the rates matrix into its logarithmic scale (stored in weights). This transformation allows for the detection of negative cycles (arbitrage opportunities) and facilitates the calculation of detoured paths.

2. **Apply Floyd-Warshall Algorithm:**
   The Floyd-Warshall algorithm is applied to the weights matrix to compute the

shortest paths between all currency pairs, updating the dist and next_node matrices.

3. **Compute Direct and Detoured Conversion Rates:**
   - **Direct Conversion Rate**: The conversion rate between the source and target currencies is retrieved directly from the rates matrix.
   - **Detoured Conversion Rate**: If a valid path exists (determined by get_path), the function calculates the detoured conversion rate by multiplying the rates along this path. The sum of the negative logarithms of the rates along the path is also computed.
4. **Compare Rates and Return Results:**
   If the detoured conversion rate is greater than the direct rate, the path taken and corresponding detoured conversion rate are returned. Otherwise, the function returns the direct rate and indicates that no detoured path offers a better rate.

Return Values:

- **direct_rate**:
  The direct conversion rate from the source to the target currency.
- **rate**:
  The best conversion rate (either direct or detoured).
- **path_names**:
  The list of currency names in the path of the detoured conversion (if any).
- **sum_of_logs**:
  The sum of negative logarithms of the rates in the best path, used for comparison.
- **weights**:
  The matrix of negative logarithms of the exchange rates.
- **path**:
  The reconstructed path as a sequence of currency indices.

- ## Task 2: Testing to Find the Best Conversion Rate with Floyd-Warshall Algorithms (First Test – No Arbitrage Opportunity)

  - ### Input First Test Case

    In this step, the first test for task 2 is used the test1 which is used in the first test in task 1. This test data show that there is **no arbitrage opportunity** and real time API data source.

```
# Task 2. Testing To Find Best Conversion Rate with Floyd-Warshall Algorithms (First Test)
print("Task 2. Finding Best Conversion Rate with Floyd-Warshall Algorithms (First Test)\n")

# Example usage
source_currency = 'NZD'  # Currency A
target_currency = 'USD'  # Currency C

# Example array
test1 = [[1.0, 0.6242247835816354, 0.8394642329594796, 0.9196416454776505, 0.4708903966148208, 0.5584629758593209],
         [1.6019870186221488, 1.0, 1.3448108037986857, 1.4732539778395084, 0.754360302570778, 0.8946504377077266],
         [1.1912359821151182, 0.7435990231304664, 1.0, 1.0955102187445322, 0.5609415840800336, 0.6652611915227097],
         [1.0873800734422072, 0.6787695910154445, 0.9128166792875851, 1.0, 0.512036834054254, 0.607261511704662],
         [2.1236364283257627, 1.3256264898777264, 1.7827168253892953, 1.9529844993417853, 1.0, 1.1859723194060652],
         [1.790629, 1.117755, 1.503169, 1.646737, 0.84319, 1.0]]

currencies = ["NZD", "USD", "CAD", "AUD", "GBP", "EUR"]

source_index = currencies.index(source_currency)
target_index = currencies.index(target_currency)

direct_rate, detoured_rate, path_names, sum_of_logs, weights, path = best_conversion_rate(len(test1), test1, source_index, target_index)
```

Figure 4.4 Setting Up test 1 Data, Source Currency, Target Currency and Run the best_conversion_rate Function

In this case there are two currencies are determined as source currency and target currency. To find the best conversion rate, the two currencies need to be set and the function run to find the shortest path.

After running the best_conversion_rate function, it will return direct_rate, detoured_rate, path_names, sum_of_logs, weights, path values.

  - ### Printing the output

    There are several print functions to show that if there is a shortest pathway. If so, it also print the return value of the best_conversion_rate function. If there is no shortest pathway via intermediate currency, printing detoured and direct conversion is same.

```
307  if path_names:
308      print(f"Direct conversion rate from {source_currency} to {target_currency}: {direct_rate}")
309      print(f"Detoured conversion rate from {source_currency} to {target_currency}: {detoured_rate}")
310      print(f"Best conversion path via detoured conversion: {' -> '.join(path_names)}")
311      print(f"The conversion rate via intermediate currency is better than the direct conversion rate.")
312      # Print the negative logarithms for each step in the detoured path
313      print("\nNegative logarithms along the detoured path:")
314      for i in range(len(path) - 1):
315          from_currency = currencies[path[i]]
316          to_currency = currencies[path[i + 1]]
317          log_value = weights[path[i]][path[i + 1]]
318          print(f"{from_currency} -> {to_currency}: {log_value}")
319
320      print(f"\nSum of negative logarithms: {sum_of_logs}")
321      print(f"\nDirect conversion of logarithm scale: {-math.log(direct_rate)}")
322  else:
323      print(f"Direct conversion rate from {source_currency} to {target_currency}: {direct_rate}")
324      print("There is no shortest path via intermediate currencies.")
```

Figure 4.5 Printing the Result of the best_conversion_rate Function for First Test

If the shorted path is defined, show the sequence of currency conversion, list of negative logarithms of each conversion from source to target, sum of logarithm scale of the path and logarithm scale of direct conversion rate to compare with detoured logarithm scale.

- ## Output of the First Test Case

```
Task 2. Finding Best Conversion Rate with Floyd-Warshall Algorithms (First Test)

Direct conversion rate from NZD to USD: 0.62422478358163 54
There is no shortest path via intermediate currencies.
```

Figure 4.6 The Output of the First test of Task 2

the code is trying to find the best conversion rate between two currencies, in this case, from NZD (New Zealand Dollar) to USD (United States Dollar), using the Floyd-Warshall Algorithm. The algorithm looks for the best possible exchange rate, whether it's a direct conversion or by passing through intermediate currencies (also called a detoured conversion).

Detoured Conversion Rate:

- The algorithm then checks if a better conversion rate exists by using other currencies as intermediates. This is done using the Floyd-Warshall algorithm, which looks for the shortest (or in this case, the most favorable) path from NZD to USD by potentially converting via other currencies like CAD, AUD, GBP, or EUR.

However, the algorithm can not find a better conversion rate than direct conversion. So, there is no shortest path via intermediate currencies.

- ## Analysis of the result and observation

In this test, the objective was to find the **best conversion rate** between two currencies—**NZD** (New Zealand Dollar) and **USD** (United States Dollar)—either directly or by using intermediate currency conversions (detoured conversion). The Floyd-Warshall algorithm was applied to the exchange rate data to identify both the **direct conversion rate** and any **better conversion path through intermediate currencies**.

Key Observations:

1. **Direct Conversion Rate:**
    - The direct conversion rate between **NZD** and **USD** was found to be 0.6242247835816354.
    - This indicates that converting directly from NZD to USD would yield approximately **0.6242 USD** for every **1 NZD**.
2. **Detoured Conversion Rate:**
    - The result shows that **there is no shorter path via intermediate currencies**. In other words, no better exchange rate could be achieved by converting NZD to another currency first (like CAD, EUR, or AUD) before converting it to USD.
    - As a result, the **detoured conversion rate** is not available or relevant in this test because there were no paths that offered an improvement over the direct rate.
3. **Assumption of No Arbitrage Opportunity:**
    - The test data assumes **no arbitrage opportunity** exists, meaning there are **no negative weight cycles** in the graph. In economic terms, this suggests that the currency exchange rates are consistent, and there are no ways to exploit inconsistencies in the rates to make a risk-free profit.
    - This assumption aligns with the result: since there is no arbitrage, the **direct conversion rate** is likely the best possible rate, and detoured conversion rates are not able to provide any improvement.

Conclusion:

In this test:

- **Direct conversion** from NZD to USD is the best and only option, providing a rate of **0.6242247835816354**.
- **No better conversion path** was found using intermediate currencies, confirming that the direct rate is optimal in this scenario.
- Since the test data assumes of **no arbitrage opportunity**, the exchange rates are well-balanced, and there are **no negative weight cycles** that could allow for a better rate through multiple currency exchanges.

Thus, the **Floyd-Warshall algorithm** verified that the direct conversion rate is the best available option when there are no arbitrage opportunities in the system.

- ## Task 2: Testing to Find the Best Conversion Rate with Floyd-Warshall Algorithms (Second Test – With Arbitrage Opportunity)

  - ### Input First Test Case

    Unlikely the first test case of the task 2, the second test case of task 1 will be used and this test has arbitrage opportunities.

```
# Task 2. Testing To Find Best Conversion Rate with Floyd-Warshall Algorithms (Second Test)
print("\nTask 2. Finding Best Conversion Rate with Floyd-Warshall Algorithms (Second Test)\n")
# Currency list for mapping
test2 = [[1.0, 0.6242, 0.8395, 0.9196, 0.4709, 0.5585],
         [1.602, 1.0, 1.3448, 1.4733, 0.7544, 0.8947],
         [1.1912, 0.7436, 1.0, 1.0955, 0.5609, 0.6653],
         [1.0874, 0.6788, 0.9128, 1.0, 0.512, 0.6073],
         [2.1236, 1.3256, 1.7827, 1.953, 1.0, 1.186],
         [1.7906, 1.1178, 1.5032, 1.6467, 0.8432, 1.0]]

currencies = ["NZD", "USD", "CAD", "AUD", "GBP", "EUR"]
source_currency = 'NZD'  # Currency A
target_currency = 'USD'  # Currency B

source_index = currencies.index(source_currency)
target_index = currencies.index(target_currency)

direct_rate, detoured_rate, path_names, sum_of_logs, weights, path = best_conversion_rate(len(test2), test2, source_index, target_index)
```

Figure 4.7 Setting Up test 2 Data, Source Currency, Target Currency and Run the best_conversion_rate Function

As mentioned, the test 2 data is rounded up 4 decimal places from the test 1. It makes tiny difference from the test 1 and may occur different result.

The other step of test is same with test 1 of the task 2 and also same with currency names, source currency, target currency, return values.

  - ### Printing the output

```
if path_names:
    print(f"Direct conversion rate from {source_currency} to {target_currency}: {direct_rate}")
    print(f"Detoured conversion rate from {source_currency} to {target_currency}: {detoured_rate}")
    print(f"Best conversion path via detoured conversion: {' -> '.join(path_names)}")
    print(f"The conversion rate via intermediate currency is better than the direct conversion rate.")
    # Print the negative logarithms for each step in the detoured path
    print("\nNegative logarithms along the detoured path:")
    for i in range(len(path) - 1):
        from_currency = currencies[path[i]]
        to_currency = currencies[path[i + 1]]
        log_value = weights[path[i]][path[i + 1]]
        print(f"{from_currency} -> {to_currency}: {log_value}")

    print(f"\nSum of negative logarithms: {sum_of_logs}")
    print(f"\nDirect conversion of logarithm scale: {-math.log(direct_rate)}")
else:
    print(f"Direct conversion rate from {source_currency} to {target_currency}: {direct_rate}")
    print("There is no shortest path via intermediate currencies.")
```

Figure 4.8 Printing the Result of the best_conversion_rate Function for First Test

The print function are all same with the test 1 of the task 2 and make only different test data.

- ## Output of the Second Test Case

```
Task 2. Finding Best Conversion Rate with Floyd-Warshall Algorithms (Second Test)

Direct conversion rate from NZD to USD: 0.6242
Detoured conversion rate from NZD to USD: 0.6242522
Best conversion path via detoured conversion: NZD -> CAD -> USD
The conversion rate via intermediate currency is better than the direct conversion rate.

Negative logarithms along the detoured path:
NZD -> CAD: 0.17494880246454l5
CAD -> USD: 0.29625202313484805

Sum of negative logarithms: 0.4712008255993896

Direct conversion of logarithm scale: 0.47128444914545803
```

Figure 4.9 The Output of the Second Test of Task 2

The algorithms print direct conversion and detoured conversion. Then, the detoured conversion is slightly more than direct conversion which means detoured conversion is more profitable even thought that is small amount. Also, It shows the best conversion path for detoured conversion which is NZD – CAD – USD.

As well as the negative logarithm scales are presented in the detoured path. Then the sum of negative logarithm is 0.4712008255993896 which less than direct conversion of logarithm scale (0.47128444914545803)

- ## Analysis of the result and observation

In the second test of Task 2, we examine the possibility of achieving a **better conversion rate** between **NZD** (New Zealand Dollar) and **USD** (United States Dollar) by considering both direct and detoured (intermediate currency) conversion rates.

Key Observations:

1. **Direct Conversion Rate:**
    - The direct conversion rate from **NZD to USD** is **0.6242**.
    - This means that for every **1 NZD**, you would get approximately **0.6242 USD** through direct conversion.
2. **Detoured Conversion Rate:**
    - The detoured conversion rate from **NZD to USD** is found to be **0.6242522**.

- This rate was achieved by converting NZD to **CAD (Canadian Dollar)** first, and then converting CAD to USD. The conversion path used is: **NZD -> CAD -> USD**.
- The detoured rate of **0.6242522** is slightly better than the direct conversion rate of **0.6242**, confirming that converting through intermediate currencies provides a **marginally better conversion rate**.

3. **Best Conversion Path:**
   - The algorithm determined that the **best conversion path** between NZD and USD is through CAD, as follows: **NZD -> CAD -> USD**.
   - This detoured path provides a small improvement over the direct conversion rate, making it the preferable route.
4. **Negative Logarithms of Detoured Path:**
   - The **negative logarithms** of the exchange rates along the detoured path are calculated:
     - **NZD -> CAD**: 0.1749488024645415
     - **CAD -> USD**: 0.29625202313484805
   - The sum of these negative logarithms is **0.4712008255993896**.
   - This indicates that the detoured path results in a slight improvement over the direct conversion.
5. **Direct Conversion Logarithm Scale:**
   - The logarithmic scale for the direct conversion from **NZD to USD** is **0.47128444914545803**.
   - This value is slightly higher than the sum of the negative logarithms along the detoured path, meaning the detoured path offers a marginally better conversion rate, as a **lower logarithmic sum** is associated with a better conversion rate.

Conclusion:

In this second test, we see that:

- The **detoured conversion path** (NZD -> CAD -> USD) provides a **slightly better rate** (0.6242522) compared to the **direct conversion rate** (0.6242).

- Although the difference in rates is very small, the **detoured conversion** route is the optimal choice for getting the best conversion rate from **NZD to USD**.
- The **sum of negative logarithms** along the detoured path confirms that the detoured rate is marginally better than the direct rate, as reflected by a smaller logarithmic sum compared to the direct conversion's logarithmic scale.

This test illustrates how the **Floyd-Warshall algorithm** can effectively identify the best conversion path, even when the improvement over the direct conversion rate is minimal.