Final Project Report

Currency Arbitrage Using Directed Graphs

CS 5800
Fall 2022

Presentation

https://youtu.be/Ri-57xs7OnI

Group Members

Ifteda Ahmed-Syed
Leaksmy Heng
Aushee Khamesra
Shivani Patel

Introduction

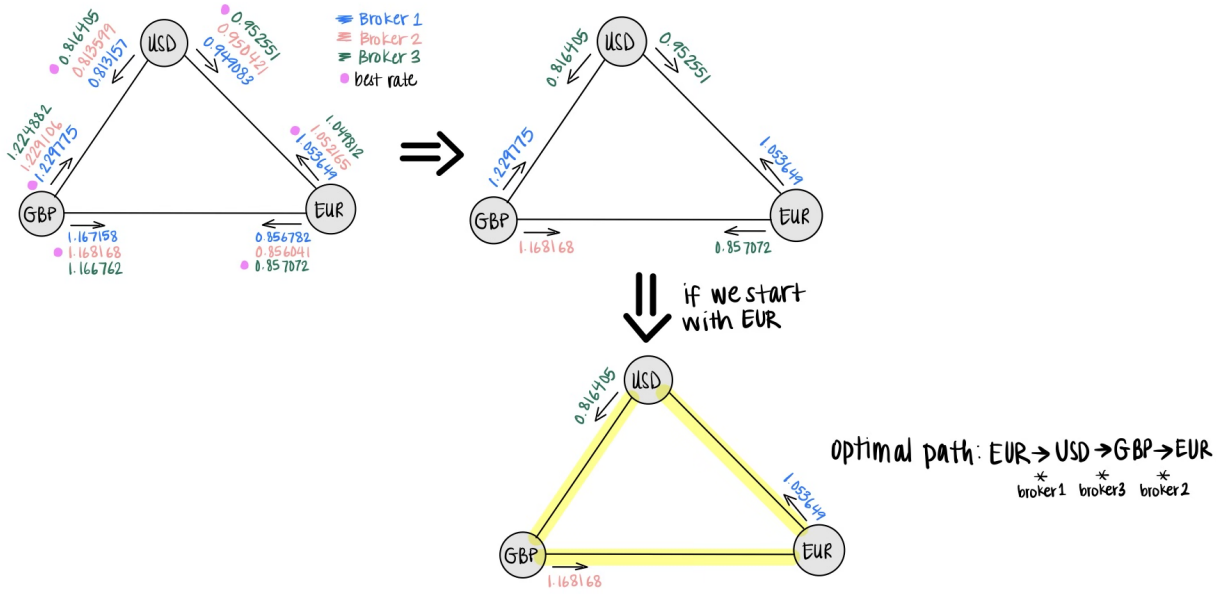
Question

Arbitrage is defined as the buying and selling of a currency or commodity in multiple markets in order to profit from varying market prices. The objective is to sequence transactions in a way that maximizes profit by the end of the process. This process can be represented by a directed graph, where each vertex represents a distinct market and the weight of each directed edge (u, v) represents the exchange rate when trading an item from u to v. With a graphical representation, we can approach the exchange process as a shortest path problem and seek to answer the question: What sequence of trades between a given set of markets will multiply my initial value to yield the maximum profit?

Description & Scope

By illustrating the exchange opportunities as a graph, we are able to approach currency arbitrage as a shortest path problem that can be solved using an algorithm such as Bellman-Ford. Because we are interested in maximizing our initial value, we would modify the chosen algorithm to prefer *maximum*-weighted edges rather than the minimum-weighted edges. This adjustment will trace the optimal path of transactions that will yield the greatest final value among a set of markets.

We plan to analyze how a graph-based approach might support decision-making in currency arbitrage by taking advantage of readily available market information. We will apply our adjusted algorithm to graphical representations of a finite set of markets. We will *not* explore additional risks associated with currency arbitrage or examine financial economic theories, such as the limits to arbitrage. We will walk through currency exchanges among 3 markets as an example. In order to mimic real-world arbitrage, we will create 3 different brokers that provide various rates among the markets. Our algorithm will choose the best rate per currency exchange and then run Bellman-Ford to determine the most profitable exchange path. The output will provide the broker chosen for each transaction as well the respective rate. However, if our algorithm is sufficiently optimized, we will consider adding examples with a greater number of markets.



Exchange rate data necessary to build the graphs is readily available via platforms such as Yahoo Finance. We will collect a snapshot of the data for a set of markets on a selected day and apply our modified algorithm to find the optimal combination and order of transactions on that particular day. We recognize that, in practice, transactions are likely to be made over several days and that exchange rates can change several times even over a single day. For the scope of this project, we will not be analyzing the effects of fluctuating rates mid-traversal. We will assume the data is relatively static between the time of collection and the time of processing. In other words, our algorithm will find the most profitable chain of transactions at a given instant. We also recognize that arbitrage processes may limit the total number of transactions over a specific period of time or include limits on the number of times any one market is included in a transaction. While we will not delve into the specifics of these numbers in real-world applications, we may explore different termination points for our algorithm (e.g., terminating before trading with an already visited market vs. terminating after a certain number of total transactions).

Context

Ifteda Ahmed-Syed

Though finance and economy are rooted in mathematics, non-quantifiable factors and forces make these topics very abstract. In my undergraduate studies, I majored in Computer Science and Business with a Business concentration in Finance. A portion of my undergraduate coursework was dedicated to studying the confluence of computer science and business, specifically exploring computational applications and approaches to the field of finance. By viewing finance problems and opportunities from a computer science perspective, I was able to feel less overwhelmed by the abstractness of finance and economics. Code-based analyses and projects helped me feel less daunted by the ambiguity of some more theoretical approaches and gave me more confidence in my understanding of course content. Through this project, I hope to explore more advanced computational and algorithmic approaches to real-world financial/economic opportunities.

Leaksmy Heng

With a Bachelor's degree in Finance Management Information Systems , and 3 years of industry experience working as a software engineer at Dell/EMC; I want to synthesize financial insight with computational skills to create systems that will algorithmically identify & classify emerging financial trends. Therefore, I would like to take this opportunity to explore currency arbitrage. I believe this topic is important to everyone especially during an economic downturn, as it could help identify potential investment opportunities. Through this project, I hope to gain more insight into the complexities, and trade offs of advanced algorithms as it relates to real world financial opportunities.

Aushee Khamesra

Arbitrage is the inefficiency of spot currency rates in the foreign exchange market. Hailing from India currency rates have had a significant impact on my family during oversea travels. In addition to personal lives, this affects businesses as well. Spot rates can be extremely volition and finding an opportunity to arbitrage can save a company millions of dollars on their balance sheet. With mass layoffs occurring an efficient tool can help companies avoid mass layoffs and focus more of their funds towards building value for the company. I unfortunately worked at a company that did not exercise its due diligence in regards to its overseas business operations. This led to a significant tightening of the budget and massive layoffs in its overseas offices. The inefficiency of spot rates occurs because banks will give their own rate based on their calculations. This algorithm could push banks to become more efficient in their spot rates and could lead to the currency market to emulate the efficiency we have come to see in many common US markets. Through this project, I hope that I can learn more about the business and

financial capabilities of algorithms and help companies use the foundations of what we create for currency arbitrage as well as commodities arbitrage.

Shivani Patel

Throughout the past several years, the topic of arbitrage has hit almost every market. With the threat of a recession circulating, more and more people are looking toward non-traditional ways to supplement their income. Topics such as rental arbitrage using Airbnb, cryptocurrency, and most recently, the reselling of event tickets have grown popular. When I declared Data Science as a major during my undergraduate, I gravitated toward business. Many of my courses were focused on less instantaneous rewards such as developing a business plan to build customer relations and using financial data to effectively present product ideas. However, I found that I lacked experience with more financial-heavy topics. Since honing my focus on business and finance, I have passively heard more about currency arbitrage, and as stocks are a newer interest of mine, I am finding that currency arbitrage also interests me. I would love to explore different parts of finance and am excited to learn more about currency arbitrage in depth during this project.

Analysis

Data Collection

To perform currency arbitrage, we gathered conversion rates from x-rates. The main currencies we chose to focus on were the US Dollar (USD), Euro (EUR), and the British Pound (GBP). For the purpose of this project we will be using data from 3 different days to represent 3 separate brokers. Since Arbitrage is done using concurrent rates from different brokers, the reality would be done based on the simultaneous rates at a single time. The data for each broker was stored in a three by three matrix as shown below.

Broker 1 (data collected on 12/1)

	USD	GBP	EUR
USD	1	0.813157	0.949083
GBP	1.229775	1	1.167158
EUR	1.053649	0.856782	1

Broker 2 (data collected on 12/2)

	USD	GBP	EUR
USD	1	0.813599	0.950421
GBP	1.229106	1	1.168168
EUR	1.052165	0.856041	1

Broker 3 (data collected on 12/3)

	USD	GBP	EUR
USD	1	0.816405	0.952551
GBP	1.224882	1	1.166762
EUR	1.049812	0.857072	1

Rates from x-rates.com

Naive Approach

Please see Appendix Item A for the naive implementation using Python.

A graph of the naive approach has the available market as the node and the value exchange between markets as the directed edge. We denote those in a dictionary as shown in Appendix A.

For the algorithm of the naive approach, we only consider two parameters:

1. graph: the nodes and edges value as shown in Figure A
2. source: the market or currency with which the user begins

In order to implement arbitrage using a naive approach, we need to ensure that all the cycles in the graph are detected. To do so, we create a function all listed_all_cycle that takes three parameters including graph, node and tracking node with the tracking node as the currency which the user begins with. We implement this function using a stack data structure with LIFO order. First, we create a node_checker variable to keep track of the nodes that we have visited. The visited node is stored in the form of a list with a tuple as the elements see Appendix Item A. We check if there are elements in node_checker, if there are, we grab the main_node and the last path the graph has visited using the .pop() method. Then we detect the cycle by checking if the last element of the last path is the same as the tracking node. Return the path through the yield method if the cycle is detected. If not, we use the main_node to check if that node has any child_node in the graph. Loop through the child_node to see if it is already in the path. If it is, continue; otherwise, append the child_node to the latest_path.

Once the cycles are detected. We now could find the arbitrage opportunities by looping through the cycles, accessing the weight of each edge in the cycles, and calculating the arbitrage opportunities through multiplying each edge in each cycle. The arbitrage opportunities results are stored in a list. The maximum value of the arbitrage opportunities resulting in the list will be detected and checked against the original value of one. If it is higher, we will return a tuple in the form of the cycle that leads to the arbitrage opportunity, each edge value and the profit margin.

Even though a naive method does provide a solution to find the opportunities, there are some limitations to it especially around the time complexity. Bellman Ford algorithm is implemented further down the paper and it provides a better time complexity.

Runtime & Space Complexity

Identifying the cycles in the graph and calculating each cycle are the main operations. To detect the cycles, we use depth first search and stack data structure. We use node_checker as a stack to check for each distinct node that is inherited directly from the main node (the head currency). In each distinct node, we traverse through all of their child nodes using DFS stack data structure. Hence, the time complexity for this takes O(b^m) where b is the number of nodes or brokers and m is the maximum depth of any broker.

The algorithm uses the following data structure.

Structure Name	**Description**	**Space Complexity (Worst Case)**
Inputs		
graph	Dictionary: brokers and their values	bn^2 (b brokers, each having a matrix of size n^2)
currency_head	String: the market or currency with which the user begins	N (number of vertices = number of markets)
list_head	List: a list of currency_head in case of multiple brokers. See Appendix Item A for example.	N (number of elements in list_head)
Structures used within algorithm		
ori_cycle_path	List: stored all of the cycle in the graph in the form of [(), (), ()..]	Exponential time because we are using a dense graphs with per every 2 nodes, there are cycles (transaction could go back and forth)

node_checker	List: build a stack data structure using this list	n
all_transaction	Dictionary: store all cycles in a path, all the value in each cycle, the calculation for the arbitrage opportunity in the cycle	Exponential time as we store all cycles in the path

In conclusion, the worst space complexity of this algorithm is an exponential time. This occurs in the checking for cycles method. Because the graph that we are using for arbitrage is like a wheel graph but a dense wheel graph in which each vertex is interconnected.

Graphical Approach

Please see Appendix Item B for the enhanced, graph-based implementation using Python.

A graph of this problem would include each available market as a vertex and the exchange rate between market u to market v as a directed edge from vertex u to vertex v of weight equal to the exchange rate.

In our enhanced approach, we are able to consider rates from different brokers, so long as their respective matrices are arranged in the same order. The overall algorithm expects three inputs:

1) rates: dictionary of broker names and their respective rates matrices
2) markets: a list of markets or currencies in the same order as the above matrices
3) source: the market or currency with which the user begins

Creating a graph based on all rates from brokers would force us to either create vertices as clustered market-broker associations or have more edges than necessary for the algorithm. For each transaction from market u to market v, there are x number of possible exchange rates, where x equals the number of brokers. This means there are x directed edges in each direction between any two vertices.

|E| = number of brokers * (number of markets(number of markets - 1))

In order to reduce the density of the graph, we use an auxiliary function to select the best rate among the brokers for each transaction. This function selects the greatest exchange rate between each pair of vertices and outputs a single matrix (Python DataFrame) which represents a complete graph: one edge in each direction per pair of vertices. It also returns a dictionary of all edges in the output graph, along with the broker whose rate was greatest for that transaction.

The next step of the algorithm is to create a matrix based on the negative log of each rate. This step is essential because it allows us to truly apply Bellman-Ford as a *shortest*-path algorithm and later detect negative-weighted cycles to find arbitrage opportunities. We also initialize data vessels to track the following:

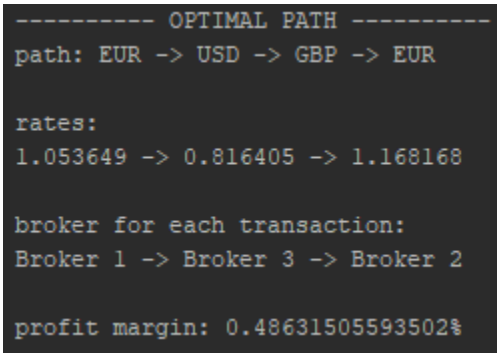
1) log_rates: matrix of -log rates
2) distance: shortest distance from the source to all other vertices
3) predecessor: the predecessor of each vertex in the shortest path
4) max_profit: product of exchange rates in optimal path
5) reverse_optimal_path: vertices in optimal path in reverse order
6) reverse_optimal_edges: edges: edges in optimal path in reverse order

The first traversal of the graph closely mimics the Bellman-Ford algorithm using a single source. Each edge is relaxed based on whether it offers a greater exchange rate (more negative log) than the shortest path accumulated so far. Upon finding a potential log weight that is less than the existing weight, both the predecessor of the current vertex and the minimum distance to the current vertex are updated. By the end of these iterations, we will have found all shortest paths from the given source, as well as the order of vertices. Each shortest path will have at most |V| - 1 edges.

A second traversal of the graph also closely follows Bellman-Ford, as it seeks to detect negative-weighted cycles. Because we have taken the -log of each rate, a negative-weighted cycle represents a path that would infinitely reduce the value if not terminated. In context, this would infinitely *increase* our profit if not terminated. Negative-weighted cycles can be detected by determining if any edges can be relaxed on this second iteration - that is, detecting any shorter distances than the minimum distances we have already discovered.

Upon detecting a negative cycle, we follow the chain of predecessors starting from the current vertex to determine the complete order of markets in the arbitrage opportunity. We have elected to set conditions that allow the path accumulation to terminate upon reaching the source or reaching a market that has already been visited. The original, un-logged exchange rates are used to determine the percent change of the current path. If the change is greater than the current maximum profit, the optimal profit, path, and edges are updated.

By this point, if there is an arbitrage opportunity in the graph (profit > 0), then its information will have been recorded in max_profit, reverse_optimal_path, and reverse_optimal_edges. The remainder of the algorithm functions to print the information in a presentable manner. As seen in the image below, the information stored for each path in profitable_paths allow us to show not only the order of markets in the arbitrage opportunity, but also the rates for that order of transactions, as well as the broker that provides each chosen rate. The total profit margin is also shown as a percent.



Runtime & Space Complexity

The time complexity of the algorithm is determined by three major sequential operations. The first major operation involves traversing each broker's respective rates matrix in order to create a single complete graph that selects the greatest rate for each pair of markets. This is done in a triply-nested for-loop, giving a complexity of $O(b * n^2)$ where b equals the number of brokers and n equals the number of markets (vertices).

The next major operation iterates over the graph to relax its edges n - 1 times. This is also completed in a triply-nested for-loop, giving a complexity of $O((n-1) * n^2)$.

The final major operation detects negative cycles. This is completed using a double-nested for-loop that contains a while loop. This operation runs at most n^3 times.

Each of the three major operations have a polynomial - specifically cubic - upper bound. Other operations, such as initializing lists/dictionaries or printing results are completed in at most linear time and do not exist within other loops. Therefore, the time complexity of the algorithm is determined by the three major cubic operations. As these operations are sequential, the runtime of the algorithm is **$O(n^3)$**.

The algorithm relies on various data vessels. As mentioned above, the algorithm uses the following inputs and data structures.

Structure Name	**Description**	**Space Complexity (Worst Case)**
Inputs		
rates	dictionary{string: float[][]}. brokers and their respective matrices of rates	bn^2 (b brokers, each having a matrix of size n^2)
markets	list of string. markets or currencies	N (number of vertices = number of markets)
source	string. market or currency with which the user begins	1
Structures used within algorithm		
edges	dictionary{tuple(string, string): string}. every directed edge and the broker whose rate was greatest for that transaction	$n^2 - n$ ($\|E\| = \|V\|(\|V\| - 1) = \|V\|^2 - \|V\|$)
log_rates	DataFrame. matrix of -log rates	n^2
distance	dictionary{string: float}. shortest distance from the source to all other vertices	n

predecessor	dictionary{string: string}. predecessor of each vertex in the shortest path	n
max_profit	float. profit of optimal path	1
reverse_optimal_path	string[]. vertices in optimal path in reverse order	n + 1 (source occurs twice; all other vertices occur at most once)
reverse_optimal_edges	string[]. edges in optimal path in reverse order	n (|reverse_optimal_path| - 1)

The space complexity of the algorithm is determined by the number of markets (vertices) n and the number of brokers b. Based on the above table, the complexity would be dominated by the input parameter rates, giving the algorithm a space complexity of **$O(bn^2)$**.

Conclusion

Comparison of The Algorithms

Algorithm	**Space Complexity**	**Time Complexity**
Naive	Exponential	O(b^m)
Bellman-Ford	O(bn^2)	O(n^3)

Correctness

We tested both algorithms for one broker and multiple brokers as shown in the image below. As the result shows, there is a limitation while using Naive approach with multiple brokers. The maximum arbitrage opportunities were not detected.

Single Broker: Data

	USD	EUR	GBP	CAD
USD	1.000000	0.951054	0.816910	1.344334
EUR	1.051380	1.000000	0.858838	1.413512
GBP	1.224123	1.164376	1.000000	1.645841
CAD	0.743863	0.707458	0.607501	1.000000

Figure A1: Bellman-Ford Algorithm

```
---------- OPTIMAL PATH ----------
path: USD -> GBP -> EUR -> CAD -> USD
rates: 0.81691 -> 1.164376 -> 1.413512 -> 0.743863
profit margin: 0.013796820168443524%
```

Figure A2: Naive Algorithm

```
------------ Optimal Path ------------------
path: USD -> GBP -> EUR -> CAD -> USD
rates: 0.81691 -> 1.164376 -> 1.413512 -> 0.743863
profit margin: 0.013796820168443524%
```

Multiple Broker: Data

Broker 1 (data collected on 12/1)

	USD	GBP	EUR
USD	1	0.813157	0.949083
GBP	1.229775	1	1.167158
EUR	1.053649	0.856782	1

Broker 2 (data collected on 12/2)

	USD	GBP	EUR
USD	1	0.813599	0.950421
GBP	1.229106	1	1.168168
EUR	1.052165	0.856041	1

Broker 3 (data collected on 12/3)

	USD	GBP	EUR
USD	1	0.816405	0.952551
GBP	1.224882	1	1.166762
EUR	1.049812	0.857072	1

Figure B1: Bellman-Ford Algorithm

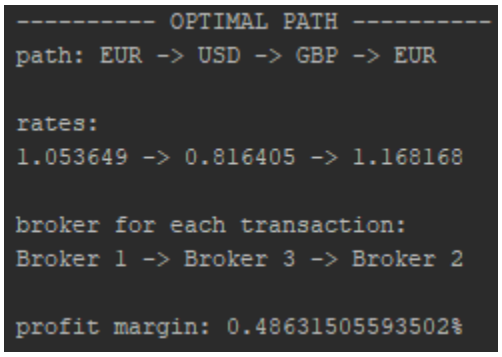
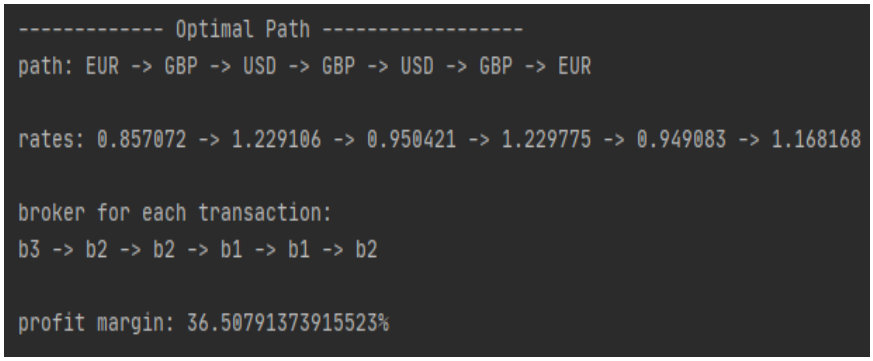


Figure B2: Naive Algorithm



Weaknesses, Limitations, & Avenues for Future Research

As we discussed earlier in the report, some limitations that we faced was the lack of availability when it came to spot rates. More visibility and access would allow us to expand our data to a greater broker network. For this model to have real world applications, the algorithm would need to work in the present. If we had more time we could develop a real time implementation that would allow the algorithm to notify the user of a real time opportunity and more importantly act upon that opportunity without needing explicit user permission, reducing human error or delay. Another limitation is the cost of arbitrage. In our project we factor the profit through arbitrage but we do not consider the broker fees as well as any other costs that may be associated such as short term capital gains tax rates for the user. A time restraint also limited us from gathering a more complete data set with more examples of arbitrage to test the viability of the algorithm.

Future research could include research into the arbitrage of other areas where there can be such as commodities, fixed income, or even sports betting. We can also look into faster shortest-path algorithms. Because arbitrage is done in real time, the speed of the algorithm is a key factor in determining the best rates and making exchanges in a timely manner is essential. Additionally, when working with large scale data, a major issue is finding a way to standardize data. A large portion of our problem, if expanded to include more data points and brokers, will be focused on data wrangling.

Ifteda Ahmed-Syed

As someone who has a generally risk-averse approach to finance, I appreciated the opportunity to explore computational approaches to finance that this project provided. In completing this project, there was a fair amount of contextual research our group had to complete in order to understand arbitrage and certain finance concepts. I do not doubt that this background knowledge will be valuable in the future. From a computer science standpoint, I appreciated the chance to consciously create an algorithmic solution and compare options. In my work as a software engineer, developing solutions is something I do every day. This assignment has helped me be more aware of the efficiency of my code and possibly identify a related concept.

Leaksmy Heng

This project enables me to explore more about the real world opportunities in the financial sector. It excites me because I am able to incorporate what I've learnt in class mainly using graphs to detect cycles by implementing DFS and store the data in a stack data structure.This has helped me enhance my knowledge in data structure and algorithm. I am more confident in using graphs and implementing them in any other frameworks now. In the future, I am planning on building more on top of what I have now, creating an API to scrap the real world currency exchange rate and develop a Flask or Django app for the user interface.

Aushee Khamesra

A close peer of mine is in the financial services industry. He explains how algorithms are becoming more popular due to how they can create profits without in a more fast paced environment. During the course of this project I was able to learn more about how lucrative algorithms can be in the financial industry. It has opened a realm open for my future career path as well as enhancing my knowledge of algorithms outside of the classroom. This project has also opened my thinking to more ways that I can introduce to my current employer. For example, as a pharmaceutical company, data is extremely important and classifying that data can be tedious and time consuming. I believe implementing certain divide and conquer algorithms could speed up the process and lead to further progress for the company towards finding a fruitful clinical drug.

Shivani Patel

This project gave me an opening to explore the financial side of data science. While I generally enjoy algorithms and finding efficient ways to solve problems, this project had the added benefit of having a clear real-world application. A family member of mine performs arbitrage in the form of electricity, and while I knew his job entailed buying and selling electricity,, I did not understand how it could produce profits. He is currently working with a development team to produce an algorithm similar to the scope of this project; they are looking to find an efficient way to look at thousands of broker rates in real time and narrow down profitable transactions. I am excited to discuss this with him further and to better understand the challenges that arise when working with larger scale data in this field. I would like to transition to the financial industry

Appendix

Item A. Naive Algorithm

```
def turn_dict_dict_to_dict_list(currency_arbitrage_graph):
    new_dict = {}
    for node in currency_arbitrage_graph:
        new_dict[node] = list(currency_arbitrage_graph[node].keys())
    return new_dict


def listed_all_cycle(graph: dict, node: str, tracking_node: list) -> list:
    graph = turn_dict_dict_to_dict_list(graph)
    node_checker = [(node, [])]
    cycle_path = []

    while node_checker:
        # checking for the last node and latest_path
        main_node, latest_path = node_checker.pop()
        if latest_path:
            # if the last element of the latest_path is the same as the
tracking_node
            # -> cycle is detected hence append that latest_path to cycle_path
            if main_node in tracking_node:
                cycle_path.append(latest_path)
                yield latest_path

        # if there are no cycle detected, use the main node to check in depth (dfs
structure)
        for child_node in graph[main_node]:
            if child_node not in latest_path:
                node_checker.append((child_node, latest_path + [child_node]))
    print(len(cycle_path))
    return cycle_path


def arbitrage_currency(graph, currency_head, list_head):
    all_transaction = {'path': [], 'cur_val': [], 'value': []}
    counter = 0
    ori_cycle_path = listed_all_cycle(graph, currency_head, list_head)

    # loop through the cycle path and calculate the value
    for cycle_path in ori_cycle_path:
        all_transaction['path'].append(cycle_path)
        all_transaction['cur_val'].append([])
        prev_node = currency_head
        result = 1

        for node_in_cycle in cycle_path:

all_transaction['cur_val'][counter].append(str(graph[prev_node][node_in_cycle]))
            result = result * graph[prev_node][node_in_cycle]
            prev_node = node_in_cycle

        counter += 1

        all_transaction['value'].append(result)
```

```
    max_value = max(all_transaction['value'])
    # check if there are arbitrage opportunities exist
    if max_value >= 1:
        find_max_value_position = all_transaction['value'].index(max_value)
        profit_margin = all_transaction["value"][find_max_value_position] - 1
        # this means there is only 1 broker
        if len(list_head) == 1:
            # printing the path
            print('------------- Optimal Path ------------------')
            print('path: ' + currency_head, '->', ' ->
'.join(all_transaction['path'][find_max_value_position]))
            print('\nrates: ' + ' ->
'.join(all_transaction['cur_val'][find_max_value_position]))
            print(f'\nprofit margin: {profit_margin}\n')

        else:
            # printing the path
            print('------------- Optimal Path ------------------')
            transaction_path = []
            transaction_broker = []
            for transaction in all_transaction['path'][find_max_value_position]:
                split_transaction = transaction.split('_')
                transaction_path.append(split_transaction[0])
                transaction_broker.append(split_transaction[1])
            print('path: ' + currency_head.split('-')[0], '->', ' ->
'.join(transaction_path))
            print('\nrates: ' + ' ->
'.join(all_transaction['cur_val'][find_max_value_position]))
            print('\nbroker for each transaction:\n' + ' ->
'.join(transaction_broker))
            print(f'\nprofit margin: {profit_margin}')

        return all_transaction['path'][find_max_value_position],
all_transaction['cur_val'][find_max_value_position],
all_transaction['value'][find_max_value_position]

    return 'No arbitrage opportunities is detected'
```

Item B. Enhanced Algorithm

```
import sys
import numpy as np
import pandas as pd
from itertools import combinations


broker_1 = [
    [1, 0.813157, 0.949083],
    [1.229775, 1, 1.167158],
    [1.053649, 0.856782, 1]
]
broker_2 = [
    [1, 0.813599, 0.950421],
    [1.229106, 1, 1.168168],
    [1.052165, 0.856041, 1]
]
broker_3 = [
    [1, 0.816405, 0.952551],
    [1.224882, 1, 1.166762],
    [1.049812, 0.857072, 1]
]

input_markets = ['USD', 'GBP', 'EUR']
brokers = {"Broker 1": broker_1, "Broker 2": broker_2, "Broker 3": broker_3}


def choose_best_rates(broker_rates, markets):
    if len(broker_rates) is 0:
        print("No rates given.")
        return

    rates = broker_rates.values()[0]
    comb = list(combinations(markets, 2))
    comb += [(item[1], item[0]) for item in comb]
    edges = dict.fromkeys(comb, broker_rates.keys()[0])

    for broker in broker_rates.keys()[1:]:
        for row in range(len(broker_rates[broker])):
            for col in range(len(broker_rates[broker][row])):
                broker_rate = broker_rates[broker][row][col]
                if broker_rate > rates[row][col]:
                    rates[row][col] = broker_rate
                    edge = (markets[row], markets[col])
                    edges[edge] = broker

    return pd.DataFrame(rates, index=markets, columns=markets), edges


def algorithm(rates, markets, source):
    rates, edges = choose_best_rates(rates, markets)

    log_rates = rates.applymap(lambda x: -np.log(x))

    distance = dict.fromkeys(rates.index, sys.maxsize)
    distance[source] = rates.index.get_loc(source)

    predecessor = dict.fromkeys(rates.index, -1)

    max_profit = 0
```

```
    reverse_optimal_path = []
    reverse_optimal_edges = []

    for i in range(len(markets) - 1):
        for start, row in log_rates.iterrows():
            for end, weight in row.iteritems():
                potential_weight = distance[start] + weight
                if potential_weight < distance[end]:
                    distance[end] = potential_weight
                    predecessor[end] = start

    for start, row in log_rates.iterrows():
        for end, weight in row.iteritems():
            potential_weight = distance[start] + weight
            if potential_weight < distance[end]:  # negative cycle detected
                reverse_negative_cycle = [source, end, start]
                reverse_negative_weights = [rates.loc[end][source],
rates.loc[start][end]]
                while predecessor[start] not in reverse_negative_cycle:

reverse_negative_weights.append(rates.loc[predecessor[start]][reverse_negative_cycle
[-1]])
                    reverse_negative_cycle.append(predecessor[start])
                    start = predecessor[start]
                if predecessor[start] is source:

reverse_negative_weights.append(rates.loc[predecessor[start]][reverse_negative_cycle
[-1]])
                    reverse_negative_cycle.append(predecessor[start])
                    profit = np.prod(reverse_negative_weights) - 1
                    if profit > max_profit:
                        max_profit = profit
                        reverse_optimal_path = reverse_negative_cycle
                        reverse_optimal_edges = reverse_negative_weights

    if max_profit <= 0:
        print("No arbitrage opportunities for given rates and source will lead to
profit.")
    else:
        path = " -> ".join(market for market in reverse_optimal_path[::-1])
        edge_path = " -> ".join(str(market) for market in
reverse_optimal_edges[::-1])
        start = reverse_optimal_path[-1]
        broker_list = []
        for end in reverse_optimal_path[-2::-1]:
            broker_list.append(edges[(start, end)])
            start = end
        broker_path = " -> ".join(str(broker) for broker in broker_list)

        print("---------- OPTIMAL PATH ----------")
        print("path: " + path + "\n")
        print("rates:\n" + edge_path + "\n")
        print("broker for each transaction:\n" + broker_path + "\n")
        print("profit margin: " + '{:.14%}'.format(max_profit))


algorithm(brokers, input_markets, "EUR")
```