Optimized Currency Arbitrage Detection Using Graph Theory and Advanced Algorithms

Asray Gopa

June 22, 2024

Abstract

Currency arbitrage involves exploiting differences in exchange rates across different markets to achieve profit. This paper presents a comprehensive approach to detect arbitrage opportunities using advanced algorithms and graph theory. We explore the use of the Bellman-Ford algorithm, the Floyd-Warshall algorithm, dynamic programming principles, and various optimization techniques to enhance performance. The detailed logic, processes, and mathematical underpinnings of the system are discussed, with a focus on efficiency and accuracy. Additionally, we examine the potential for hardware engagement to further accelerate computations and discuss the practical applications of this system in the financial field.

1 Introduction

Currency arbitrage is a financial strategy where traders exploit price discrepancies of the same currency pairs across different markets. The goal is to execute a sequence of currency exchanges starting and ending with the same currency, resulting in a profit. This paper delves into the algorithms and optimizations used to detect such opportunities efficiently.

2 Background and Related Work

Detecting currency arbitrage opportunities is a classic problem in finance and computer science. Traditional approaches often rely on the Bellman-Ford algorithm to detect negative weight cycles in a graph representation of exchange rates. Recent advances have explored the use of parallel processing, asynchronous requests, and all-pairs shortest path algorithms to improve performance.

3 Mathematical Foundations

3.1 Graph Representation

We represent currency exchanges as a directed graph $G = (V, E)$, where each vertex $v \in V$ represents a currency and each edge $(u, v) \in E$ represents an exchange rate from currency

u to currency v. The weight of the edge is given by $-\log(\text{rate})$, transforming the problem into one of finding negative weight cycles.

3.2 Logarithmic Transformation

The use of logarithms transforms the problem into an additive one, which simplifies the detection of arbitrage opportunities. Given an exchange rate r from currency u to currency v, the weight of the edge is $w(u, v) = -\log(r)$. If the sum of the weights in a cycle is negative, then the product of the exchange rates in the cycle is greater than one, indicating an arbitrage opportunity.

3.3 Bellman-Ford Algorithm

The Bellman-Ford algorithm is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph. It is particularly suitable for detecting negative weight cycles, which correspond to arbitrage opportunities.

Algorithm 1 Bellman-Ford Algorithm

```
Initialize d[v] ← ∞ for all v ∈ V
d[source] ← 0
for each vertex i from 1 to |V| − 1 do
  for each edge (u, v) ∈ E do
    if d[v] > d[u] + w(u, v) then
      d[v] ← d[u] + w(u, v)
    end if
  end for
end for
for each edge (u, v) ∈ E do
  if d[v] > d[u] + w(u, v) then
    Negative cycle detected
  end if
end for
```

3.4 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an all-pairs shortest path algorithm that can find shortest paths between all pairs of vertices in a weighted graph. It is particularly efficient for dense graphs.

4 Implementation

4.1 System Architecture

The system is composed of multiple scripts, each leveraging different optimizations:

Algorithm 2 Floyd-Warshall Algorithm

```
Initialize dist[u][v] ← ∞ for all u, v ∈ V
for each vertex u ∈ V do
  for each vertex v ∈ V do
    if u = v then
      dist[u][v] ← 0
    else if (u, v) ∈ E then
      dist[u][v] ← w(u, v)
    end if
  end for
end for
for each vertex k ∈ V do
  for each vertex i ∈ V do
    for each vertex j ∈ V do
      dist[i][j] ← min(dist[i][j], dist[i][k] + dist[k][j])
    end for
  end for
end for
```

- **app.py**: Utilizes the Floyd-Warshall algorithm and multiprocessing for efficient arbitrage detection within a Flask web application.
- **arbitrage.py**: Contains the core logic for fetching exchange rates, detecting arbitrage opportunities, and calculating profits.
- **OtherMethods/using-hardware.py**: Employs asynchronous requests and hardware-optimized operations.
- **OtherMethods/waytoolong.py**: A less optimized version that serves as a baseline for comparison.
- **templates/index.html**: A simple HTML interface for interacting with the Flask application.

4.2 Data Fetching and Structuring

Exchange rates are fetched from an API and structured into a graph representation. Asynchronous requests are used to reduce latency.

```
import requests
import math
import logging
from collections import defaultdict
from decimal import Decimal, getcontext
import os
from dotenv import load_dotenv
```

```
# Load environment variables from .env file
load_dotenv()

# Set precision for Decimal calculations
getcontext().prec = 50

# Logging configuration
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s
- %(message)s')

# Read API key from environment variable
API_KEY = os.getenv('API_KEY')
BASE_URL = 'https://v6.exchangerate-api.com/v6/'

# Transaction fee as a percentage (e.g., 0 for no fees, 0.5 for 0.5% fees)
TRANSACTION_FEE_PERCENT = Decimal(0)

def get_exchange_rates(api_key):
    try:
        url = f'{BASE_URL}{api_key}/latest/USD'
        response = requests.get(url)
        response.raise_for_status()
        data = response.json()
        if 'conversion_rates' in data:
            return data['conversion_rates']
        else:
            logging.error("Conversion rates not found in response")
            return None
    except requests.exceptions.RequestException as e:
        logging.error(f"Error fetching exchange rates: {e}")
        return None

def structure_exchange_rates(rates):
    structured_data = {}
    currencies = list(rates.keys())
    for i in range(len(currencies)):
        for j in range(len(currencies)):
            if i != j:
                curr1 = currencies[i]
                curr2 = currencies[j]
                rate = Decimal(rates[curr2]) / Decimal(rates[curr1])
                structured_data[(curr1, curr2)] = rate
    return structured_data
```

4.3 Graph Creation and Cycle Detection

The graph is created by applying logarithmic transformation to the exchange rates. Arbitrage cycles are detected using the Bellman-Ford and Floyd-Warshall algorithms.

```
def create_graph(data):
    graph = defaultdict(list)
    for (curr1, curr2), rate in data.items():
        rate_with_fee = apply_transaction_fee(rate)
        graph[curr1].append((curr2, -math.log(rate_with_fee)))
    return graph

def find_arbitrage(graph):
    cycles = []
    nodes = list(graph.keys())
    with Pool(cpu_count()) as pool:
        results = pool.map(bellman_ford, [(graph, node) for node in nodes])
    for has_cycle, predecessors, source in results:
        if has_cycle:
            for n in nodes:
                cycle = reconstruct_cycle(predecessors, source, n)
                if cycle and len(cycle) > 3 and cycle not in cycles:
                    cycles.append(cycle)
    return cycles
```

4.4 Optimizations

- **Asynchronous Requests**: Using `aiohttp` for non-blocking HTTP requests.
- **Parallel Processing**: Leveraging `multiprocessing` and `concurrent.futures` for parallel execution.
- **Early Stopping**: Modifying the Bellman-Ford algorithm to stop early if no updates are made.

5 Efforts to Enhance Time Efficiency

5.1 Asynchronous Requests

One of the primary bottlenecks in fetching exchange rates is network latency. To mitigate this, we utilized asynchronous HTTP requests using the `aiohttp` library. This allowed us to fetch data in parallel, significantly reducing the time required to gather exchange rates.

```
import aiohttp
import asyncio
```

```
async def fetch_rate(session, url):
    async with session.get(url) as response:
        return await response.json()

async def get_exchange_rates(api_key):
    url = f'{BASE_URL}{api_key}/latest/USD'
    async with aiohttp.ClientSession() as session:
        response = await fetch_rate(session, url)
        if 'conversion_rates' in response:
            return response['conversion_rates']
        else:
            logging.error("Conversion rates not found in response")
            return None
```

5.2 Parallel Processing

To handle the computational load of the Bellman-Ford algorithm across multiple nodes, we employed parallel processing using the `multiprocessing` module. By distributing the workload across multiple CPU cores, we achieved a significant speedup.

```
def find_arbitrage(graph):
    cycles = []
    nodes = list(graph.keys())
    with Pool(cpu_count()) as pool:
        results = pool.map(bellman_ford, [(graph, node) for node in nodes])
    for has_cycle, predecessors, source in results:
        if has_cycle:
            for n in nodes:
                cycle = reconstruct_cycle(predecessors, source, n)
                if cycle and len(cycle) > 3 and cycle not in cycles:
                    cycles.append(cycle)
    return cycles
```

5.3 Early Stopping in Bellman-Ford Algorithm

We modified the Bellman-Ford algorithm to include an early stopping condition. If no distance updates occur during an iteration, the algorithm terminates early, reducing unnecessary computations.

5.4 Efficient Data Structures

We utilized adjacency lists and matrices for efficient representation and manipulation of graph data. This not only reduced memory usage but also improved the speed of graph operations.

Algorithm 3 Optimized Bellman-Ford Algorithm with Early Stopping

```
Initialize d[v] ← ∞ for all v ∈ V
d[source] ← 0
for each vertex i from 1 to |V| − 1 do
    updated ← False
    for each edge (u, v) ∈ E do
        if d[v] > d[u] + w(u, v) then
            d[v] ← d[u] + w(u, v)
            updated ← True
        end if
    end for
    if not updated then
        break
    end if
end for
for each edge (u, v) ∈ E do
    if d[v] > d[u] + w(u, v) then
        Negative cycle detected
    end if
end for
```

```python
def create_graph(data):
    graph = defaultdict(list)
    for (curr1, curr2), rate in data.items():
        rate_with_fee = apply_transaction_fee(rate)
        graph[curr1].append((curr2, -math.log(rate_with_fee)))
    return graph
```

6 Time Efficiency and Dynamic Programming

6.1 Time Complexity

The time complexity of the Bellman-Ford algorithm is $O(VE)$, where V is the number of vertices and E is the number of edges. The Floyd-Warshall algorithm has a time complexity of $O(V^3)$. While Floyd-Warshall is more suitable for dense graphs, Bellman-Ford is often more efficient for sparse graphs.

6.2 Dynamic Programming Principles

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. Both the Bellman-Ford and Floyd-Warshall algorithms use dynamic programming principles to compute shortest paths and detect negative weight cycles.

6.3 Memory Efficiency

The space complexity of the Bellman-Ford algorithm is $O(V)$ for the distance and predecessor arrays. The Floyd-Warshall algorithm requires $O(V^2)$ space to store the distance matrix. Efficient memory management and data structures, such as adjacency lists and matrices, are crucial for handling large graphs.

7 Hardware Engagement

7.1 Parallel Processing

Utilizing multi-core processors can significantly speed up computations. By parallelizing the Bellman-Ford algorithm using Python's `multiprocessing` module, we can distribute the workload across multiple CPU cores.

```
def find_arbitrage(graph):
    cycles = []
    nodes = list(graph.keys())
    with Pool(cpu_count()) as pool:
        results = pool.map(bellman_ford, [(graph, node) for node in nodes])
    for has_cycle, predecessors, source in results:
        if has_cycle:
            for n in nodes:
                cycle = reconstruct_cycle(predecessors, source, n)
                if cycle and len(cycle) > 3 and cycle not in cycles:
                    cycles.append(cycle)
    return cycles
```

7.2 GPU Acceleration

Graphics Processing Units (GPUs) can further accelerate computations, especially for dense graphs. Libraries such as NVIDIA's CUDA or OpenCL can be used to implement parallel algorithms on GPUs.

8 Applications in the Financial Field

8.1 Real-Time Arbitrage Detection

Real-time detection of arbitrage opportunities can provide traders with actionable insights to make profitable trades. By continuously monitoring exchange rates and running the detection algorithms, the system can alert traders to potential arbitrage opportunities.

8.2 Algorithmic Trading

Algorithmic trading involves using computer algorithms to execute trades at high speed and volume. The arbitrage detection system can be integrated into trading algorithms to automatically execute trades when arbitrage opportunities are detected.

8.3 Risk Management

Currency arbitrage strategies can be used as part of a broader risk management framework to hedge against currency risk. By exploiting price discrepancies, traders can mitigate the impact of adverse currency movements on their portfolios.

8.4 Market Efficiency

Arbitrage opportunities indicate inefficiencies in the market. By exploiting these opportunities, traders help to correct price discrepancies, contributing to overall market efficiency.

9 Conclusion

This paper presents a comprehensive approach to currency arbitrage detection, utilizing advanced algorithms and optimizations. The combination of graph theory, asynchronous processing, and parallel execution enables efficient and accurate detection of arbitrage opportunities. The system's potential applications in real-time trading, algorithmic trading, risk management, and market efficiency highlight its significance in the financial field.

10 Future Work

Future work includes exploring more sophisticated transaction fee models, integrating real-time market data, and applying machine learning techniques to predict arbitrage opportunities. Additionally, further optimization techniques, such as GPU acceleration and distributed computing, can be investigated to enhance performance.

References

[1] Ji-Mei Shen, Zhehu Yuan, Y. Jin, "AlphaMLDigger: A Novel Machine Learning Solution to Explore Excess Return on Investment," *arXiv (Cornell University)*, 2022.

[2] Otabek Sattarov, Azamjon Muminov, Cheol Won Lee, Hyun Kyu Kang, Ryum-Duck Oh, Junho Ahn, Hyung Jun Oh, Heung Seok Jeon, "Recommending Cryptocurrency Trading Points with Deep Reinforcement Learning Approach," *Applied Sciences*, 2020.

[3] Dymitr Ruta, "Automated Trading with Machine Learning on Big Data," *2014*.

[4] J. Li, Chen Zheng, Yang Chao, "Integrating Tick-level Data and Periodical Signal for High-frequency Market Making," *2023*.

[5] Johannes Stübinger, Lucas Schneider, "Statistical Arbitrage with Mean-Reverting Overnight Price Gaps on High-Frequency Data of the SP 500," *University of Erlangen-Nürnberg*, 2019.

[6] R. Preston McAfee, Sergei Vassilvitskii, "An overview of practical exchange design," *2012*.

[7] Bijesh Dhyani, Pushkar Nigam, Ankit Kumar, Abhishek Kumar, K Venkatesan, V D Ambeth Kumar, "Arbitrage-Type Trade Using Correlation Analysis," *Graphic Era Hill University, Savitribai Phule University*, 2021.