Solving Currency Exchange Problems

Hayden Richard-Marsters : 21152003
{ qjn4504 } @autuni.ac.nz
Auckland University of Technology
October 2025

Abstract

This project implements an automated system for detecting and analysing currency exchange arbitrage opportunities using graph-based algorithms. Exchange rates are modelled as a directed weighted graph where each currency represents a node and each conversion rate an edge. To detect arbitrage, the Bellman-Ford algorithm is applied in the transformed -log(rate) space, allowing negative cycle detection to signify profitable arbitrage loops. An additional profit threshold is introduced to suppress false positives arising from rounding errors and floating-point imprecision, aligning the model with realistic market constraints. For completeness the system also computes optimal conversion paths using the same graph representation when no arbitrage is present. The solution includes robust input validation, modular design, logging, and structured output consistent with specifications outlined in the brief. Testing against multiple synthetic and real-world datasets, implementing a live exchange rate data pipeline through exchangeratesapi.io. The code is available at { https://github.com/Hayden-RM/currency-arb }.

1 Introduction

The objective of this assignment is to design and implement a computational system capable of detecting currency exchange arbitrage opportunities and determining optimal conversion paths between currencies. In international financial markets, exchange rates fluctuate continuously across multiple trading platforms. These fluctuations may occasionally lead to inconsistencies in quoted rates, creating temporary opportunities for profit through a process known as arbitrage.

Arbitrage refers to the act of exploiting price differentials across different markets to achieve a risk-free profit (Chen). In the context of currency exchange, arbitrage occurs when a trader can convert one currency through a sequence of intermediary currencies and ultimately return to the original currency with an increased amount. For example, 1 USD can be exchanged for 0.9 EUR, 1 EUR for 130 JPY, and 130 JPY for 1.02 USD, a trader would achieve a profit gain of 2% through the cycle USD -> EUR -> JPY -> USD, thereby realising an arbitrage profit.

The project is structured around two principle computational tasks:

1. **Arbitrage detection** – determining the existence of profitable currency exchange cycled within a given set of exchange rates; and
2. **Best Conversion Path Computation** – identifying the most advantageous conversion route between two specified currencies when no arbitrage opportunities are present.

The Bellman-Ford algorithm was selected as the core analytical method due to its suitability for graphs containing negative edge weights and its ability to detect negative cycles, which directly correspond to arbitrage opportunities when exchange rates are transformed using negative logarithmic function. Compared with the Floyd-Warshall algorithm, Bellman-Ford provides a superior computational efficiency for single-source detection and greater interpretability in this application.

To validate the system, both real-time exchange rate data (sourced from the ExchangeRates API) and synthetically generated test cases were utilised. The use of real-time data demonstrates the model's applicability to practical market conditions, while the test cases ensure functional correctness, numerical stability, and robustness across diverse input configurations. Collectively, these components establish a reliable framework for automated arbitrage detection and exchange rate optimisation in dynamic currency markets.

2 Problem Formulation & Algorithm Design

2.1 Graph Representation of Currency Exchange

The currency exchange market can be effectively represented as a directed weighted graph $G = (V, E)$, where each vertex $v_i \in V$ corresponds to a unique currency, and each directed edge $e_{ij} \in E$ represents the exchange rate from R_{ij} from currency i to currency j.

R_{ij} = rate at which one unit of currency i can be converted into currency j

This representation naturally captures the directional nature of exchange rates, as $R_{ij} \neq R_{ij}$ in most cases.

To enable arbitrage detection using shortest-path algorithms, the rates are transformed into a cost representation by applying the negative natural logarithm:

$$W_{ij} = -\ln(R_{ij})$$

This transformation converts the multiplicative relationship of currency conversions into an additive one, allowing standard graph algorithms to operate correctly. In this form, an arbitrage opportunity corresponds to a negative-weight cycle, because a product of rates greater than implies a negative cycle in the log-transformed graph:

$$R_{ii_1} * R_{ii_2} * \dots * R_{ii_n} > 1$$

Which, after transformation, produces a total negative path cost in the -log space.

Illustrative Example
For three currencies *A, B, C:*

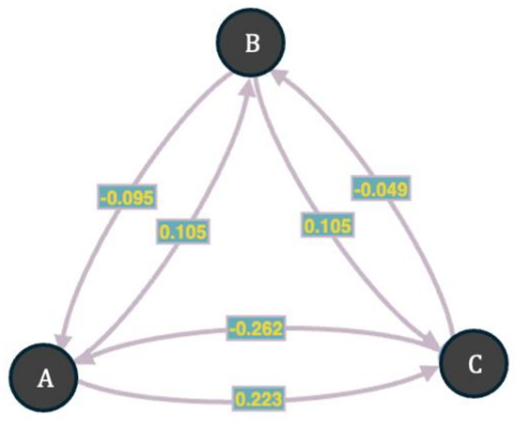
From/To	**A**	**B**	**C**
A	1.0	0.9	0.8

B	1.1	1.0	0.9
C	1.3	1.05	1.0

The corresponding transformed adjacency matrix $W = -\ln(R)$ is:

From/To	**A**	**B**	**C**
A	0.000	0.105	0.223
B	0.095	0.000	0.105
C	-0.262	-0.049	0.000

To illustrate as a visual directed graph showing nodes A, B, C and the respective weighted edges:



2.2 Arbitrage Detection

An arbitrage opportunity exists if and only if there is a negative-weight cycle in the transformed graph W. The Bellman-Ford algorithm is well suited to this track because it relaxes all edges iteratively and can identify negative cycles reachable from a given source vertex.

Mathematical Condition for Arbitrage:

$$\exists \text{ cycle } C = (v_1, v_2, \dots, v_k) \text{ such that } \sum_{(i,j)\in C} W_{ij} < 0$$

Algorithm Overview:

1. Transform exchange rates R_{ij} into weights $W_{ij} = -\ln(R_{ij})$.
2. Initialise distance $d[v] = \infty$ for all vertices, and $d[s] = 0$ for the source vertex.
3. Relax all edges $|V| - 1$ times:
   $$\text{if } d[v_j] > d[v_i] + W_{ij}, \text{then } d[v_j] = d[v_i] + W_{ij}$$
4. Perform one additional relaxation pass to check for improvements. If any distance can still be updated, a negative cycle exists implying an arbitrage opportunity.

Pseudocode:

```
FUNCTION detect_arbitrage(R, eps = 1e-12, tol = 1e-10, profit_tol = 1e-3):
    // Transform exchange rates to negative log weights
    W = rates_to_neglog_weights(R, eps)
    n = LENGTH(R)

    best_cycle = None
    best_profit = 1.0

    // Try each currency as potential starting point for arbitrage
    FOR s = 0 TO n-1:
        // Run Bellman-Ford to find negative cycles from source s
        (dist, pred, cycle) = bellman_ford_single_source(W, s, tol)

        // Skip if no cycle found from this source
        IF cycle == None:
            CONTINUE

        // Normalize cycle format: remove duplicate start/end node if present
        IF LENGTH(cycle) >= 2 AND cycle[0] == cycle[-1]:
            cycle = cycle[0:-1]  // Remove last element

        // Validate cycle has at least 2 nodes (meaningful arbitrage)
        IF LENGTH(cycle) < 2:
            CONTINUE

        // Calculate actual profit multiple from original rates
        profit = _cycle_profit(R, cycle)

        // Apply economic threshold to filter insignificant profits
        IF profit > (1.0 + ABS(profit_tol)) AND profit > best_profit:
            best_profit = profit
            best_cycle = cycle

    // Return results based on whether profitable arbitrage was found
    IF best_cycle == None:
        RETURN (False, None, None)
    ELSE:
        RETURN (True, best_cycle, best_profit)
END FUNCTION
```

2.3 Best Conversion Rate

When no arbitrage is detected, the system computes the best conversion path between two given currencies. This is achieved by applying the same shortest-path relaxation logic used in Bellman-Ford, without checking for negative cycles. In the transformed -log domain, the shortest path corresponds to the maximum product of rates in the original space.

Pseudocode:

```
FUNCTION best_conversion(R, s, t, eps = 0.0, tol = 1e-12, exact_from_R = False):
    n = LENGTH(R)

    // Validate input indices
    IF s < 0 OR t < 0 OR s >= n OR t >= n:
        RETURN (None, None)

    // Transform exchange rates to negative log weights
    W = _rates_to_neglog_weights(R, eps)

    // Run Bellman-Ford to find shortest paths
    (dist, pred) = _bellman_ford_sssp(W, s, tol)

    // Check if target is reachable
    IF dist[t] >= INFINITY:
        RETURN (None, None)

    // Attempt to reconstruct path using predecessor chain
    path = _reconstruct_path_pred(pred, s, t)

    // Fallback: if no path found, use direct edge
    IF path == None:
        RETURN (R[s][t], [s, t])

    // Secondary fallback: if path is invalid, try distance-consistent reconstruction
    IF LENGTH(path) < 2 AND s != t:
        path = _reconstruct_path_consistent(dist, W, s, t, tol)
    // Final validation of reconstructed path
    IF LENGTH(path) < 2 AND s != t:
        RETURN (None, None)

    // Calculate final conversion rate
    IF exact_from_R:
        rate = _path_product(R, path)  // Multiply actual rates along path
    ELSE:
        rate = EXP(-dist[t])  // Convert back from log space

    RETURN (rate, path)
END FUNCTION
```

Path Reconstruction:

The optimal path is reconstructed by traversing the predecessor list pred[] backward from the target vertex until the source vertex is reached. This produces the ordered path representing the sequence of currency exchanges.

Example:

Given the matrix above, the best conversion from A to C follows the path C ← B ← A, and when reconstructed forward: A → B → C, producing an effective rate of $R_{A \to C}^{best} = R_{AB} * R_{BC} = 0.9 * 0.9 = 0.81$.

In the absence of a negative cycle, this path represents the most efficient legitimate exchange sequence.

2.4 Algorithmic Complexity & Empirical Analysis

Theoretical Time Complexity

The core computational procedure of this system is the Bellman-Ford algorithm, used for detecting negative-weight cycles in the transformed exchange-rate graph. Given a graph *G = (V, E)* where each currency is represented by a vertex v_i and every directed exchange rate corresponds to an edge e_{ij}, the algorithm performs iterative relaxation over all edges.

In the dense case – where all currencies can be exchanged directly with each other:

$$|E| = n^2$$

and Bellman-Ford runs for $n - 1$ iterations.
Hence, the total time complexity is:

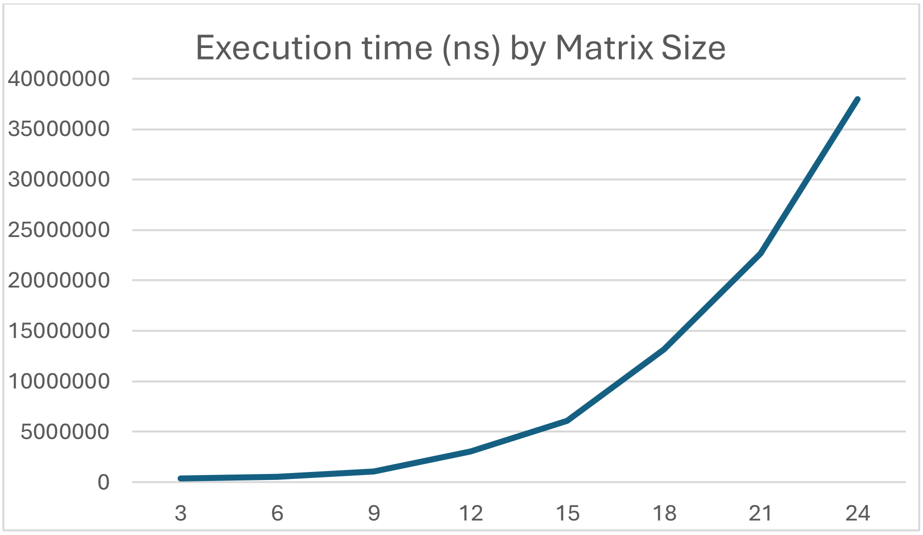
$$T(n) = O(V * E) = O(n^3)$$

This cubic relationship is comparable to Floyd-Warshall's complexity; however, Bellman-Ford operates on a per-source basis and terminates early when no further relaxations occur, making it empirically faster for smaller or sparse graphs. Moreover, Bellman-Ford supports negative0cycle detection, which directly corresponds to arbitrage loops, without requiring all pairs shortest paths.

Empirical Time Analysis

Empirical performance testing was conducted using synthetic matrices of increasing dimension n $(3, 6, \dots, 24)$. Each test case was generated under consistent computational conditions and measured using Python's time.perf_counter_ns().

Matrix Size (n)	**3**	**6**	**9**	**12**	**15**	**18**	**21**	**24**
Execution time (ns)	358250	494834	1039333	3040791	6086917	13122375	22696792	37980875



Observations:

- The empirical data exhibits an approximately cubic growth pattern, consistent with the $O(n^3)$ theoretical bound.

- Between $n = 3$ and $n = 24$, execution time increases by roughly two orders of magnitude, reflecting the n^3 scaling behaviour.
- Slight super-linear deviations at larger n values arise from Python's interpreter overhead and cache inefficiencies in nested loops.
- The smallest instances $(n \leq 6)$ terminate early due to the absence of negative cycles, confirming early-stopping efficiency.

3 Implementation

3.1 Environment Setup

Language: Python 3.9.10

Interpreter/Tools:
- IDE: VS Code
- Vitural Environment: python -m venv venv ; dependencies pinned in { requirements.txt }

Libraries:
- **Core:** standard library only for algorithms.
- **I/O & API:** requests (HTTP), python-dotenv (load API key).
- **Logging:** logging (built-in)

API provider (live data): exchangeratesapi.io (api layer)

3.2 Code Structure & Program Functionality

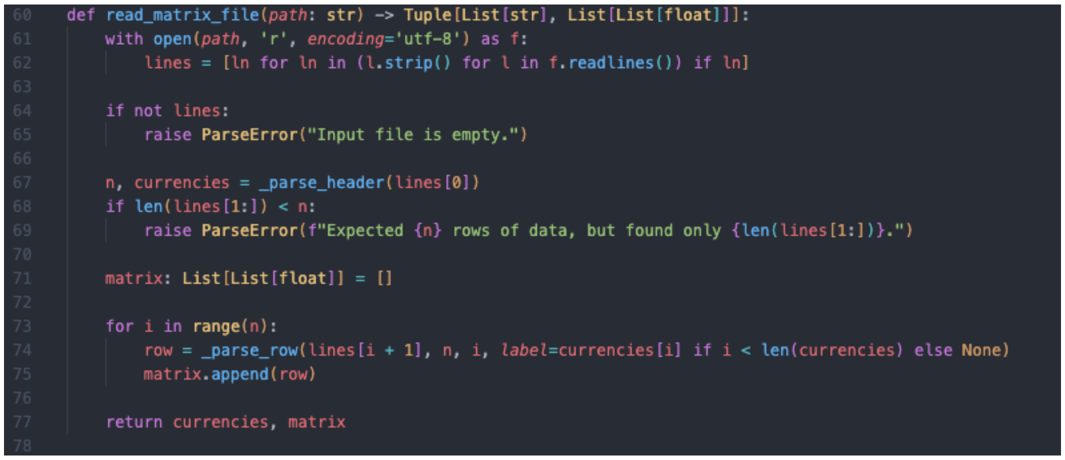
```
logs/
        debug.log
scripts/
        bruteforce_checker.py
src/
        providers/
                exchangerates_api.py
                normalize.py
        __init__.py
        arbitrage.py
        bellman_ford.py
        best_rate.py
        dump_all_arbitrage.py
        execute_utils.py
        io_parsing.py
        logging_conf.py
        runner.py
        test_runners.py
        transform.py
tests_benchmarks/
tests_extra/
tests_fetched_api/
tests_given/
.env
.gitattributes
.gitignore
license
main.py                 # Run from here
readme.md
requirements.txt
```

Core Functionality:

src/io_parsing.py – **input validation & parsing**

- Reads files of the form: n, C1, C2, … followed by n rows of n positive floats.
- Raises ParseError with user-friendly messages (count mismatch, row length, non-numeric, non-positive rate).

See full read_matrix_file function below:

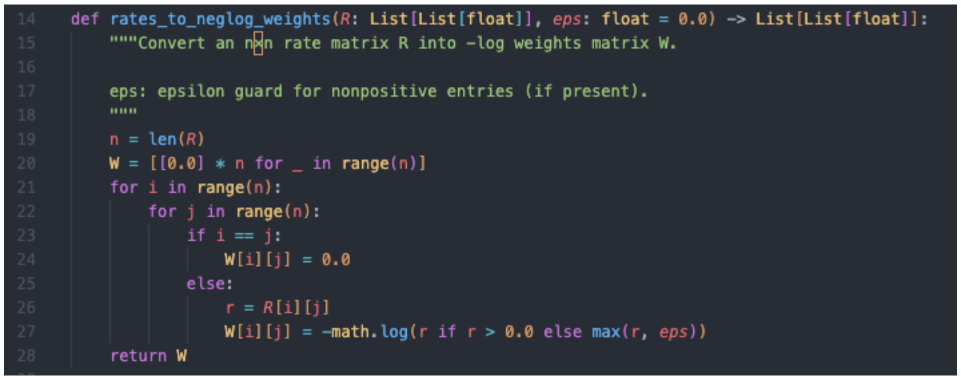


```python
def read_matrix_file(path: str) -> Tuple[List[str], List[List[float]]]:
    with open(path, 'r', encoding='utf-8') as f:
        lines = [ln for ln in (l.strip() for l in f.readlines()) if ln]

    if not lines:
        raise ParseError("Input file is empty.")

    n, currencies = _parse_header(lines[0])
    if len(lines[1:]) < n:
        raise ParseError(f"Expected {n} rows of data, but found only {len(lines[1:])}.")

    matrix: List[List[float]] = []

    for i in range(n):
        row = _parse_row(lines[i + 1], n, i, label=currencies[i] if i < len(currencies) else None)
        matrix.append(row)

    return currencies, matrix
```

src/transform.py – **Rate → weight transform**

- Implements the standard arbitrage transform $W_{ij} = -\ln R_{ij}$.
- Diagonal fixed to 0.0; optional eps guards only non-positive inputs.

See full rates_to_neglog_weights function below:

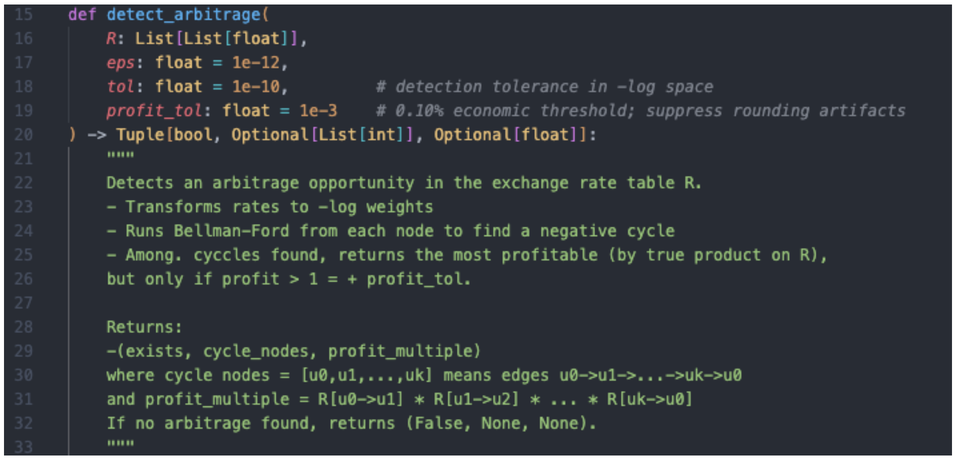


```python
def rates_to_neglog_weights(R: List[List[float]], eps: float = 0.0) -> List[List[float]]:
    """Convert an n×n rate matrix R into -log weights matrix W.

    eps: epsilon guard for nonpositive entries (if present).
    """
    n = len(R)
    W = [[0.0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i == j:
                W[i][j] = 0.0
            else:
                r = R[i][j]
                W[i][j] = -math.log(r if r > 0.0 else max(r, eps))
    return W
```

src/arbitrage.py – **Task 1: Arbitrage detection**

- Converts R to W = -log R.
- Runs Bellman-Ford from every source; collects any cycles found.
- Computes true profit on R and reports the best cycle that exceeds an economic threshold (default profit_tol = 0.1%) to suppress round/fee artifacts.

See snippet below:



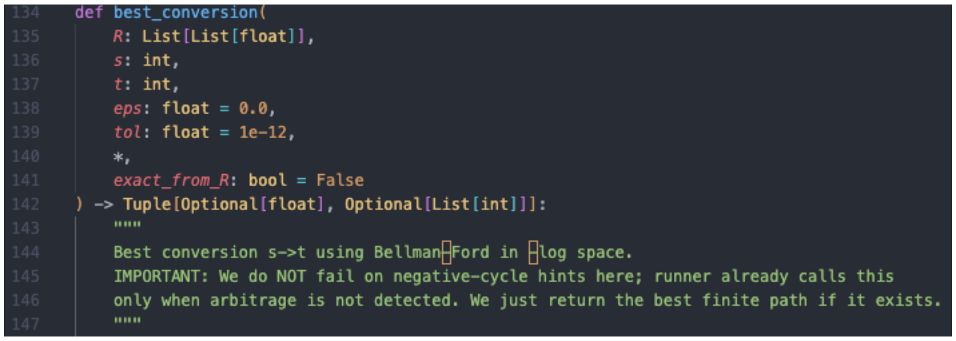
```python
def detect_arbitrage(
    R: List[List[float]],
    eps: float = 1e-12,
    tol: float = 1e-10,         # detection tolerance in -log space
    profit_tol: float = 1e-3    # 0.10% economic threshold; suppress rounding artifacts
) -> Tuple[bool, Optional[List[int]], Optional[float]]:
    """
    Detects an arbitrage opportunity in the exchange rate table R.
    - Transforms rates to -log weights
    - Runs Bellman-Ford from each node to find a negative cycle
    - Among. cyccles found, returns the most profitable (by true product on R),
    but only if profit > 1 = + profit_tol.

    Returns:
    -(exists, cycle_nodes, profit_multiple)
    where cycle nodes = [u0,u1,...,uk] means edges u0->u1->...->uk->u0
    and profit_multiple = R[u0->u1] * R[u1->u2] * ... * R[uk->u0]
    If no arbitrage found, returns (False, None, None).
    """
```

src/best_rate.py – **Task 2: Best Conversion (when no arbitrage exists)**

- Runs Bellman-Ford (BF) single-source shortest path (SSSP) in – log space to obtain dist and pred.
- Reconstructs the path; returns rate = exp(-dist[t]) and node path.

- Includes a small fallback ([s, t]) to avoid “N/A” in dense matrices.

See snippet below:

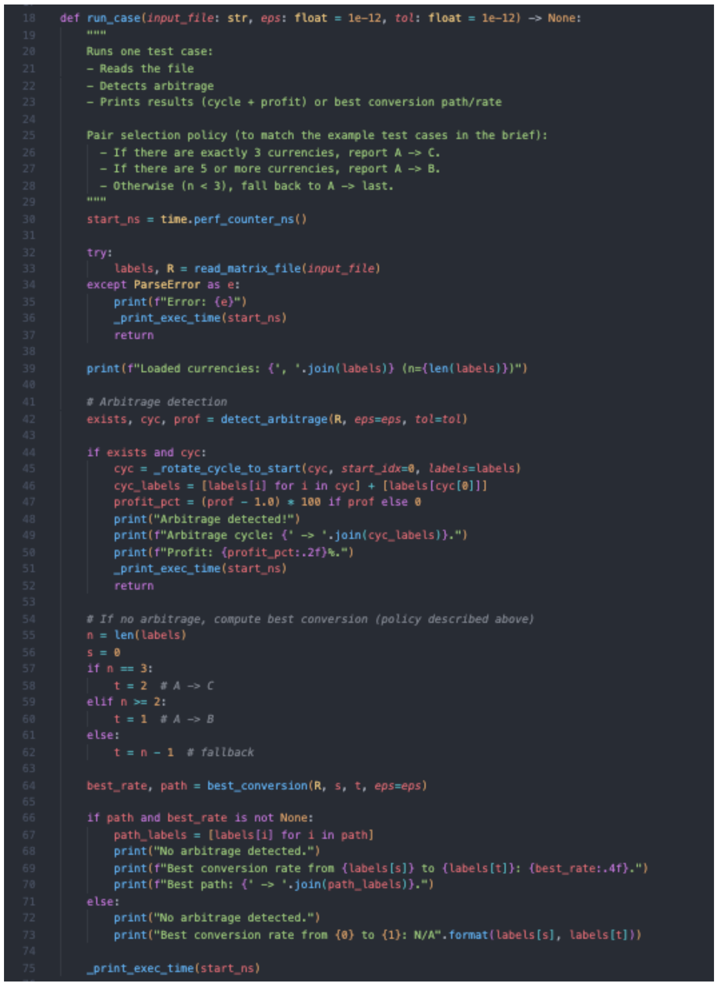


```python
def best_conversion(
    R: List[List[float]],
    s: int,
    t: int,
    eps: float = 0.0,
    tol: float = 1e-12,
    *,
    exact_from_R: bool = False
) -> Tuple[Optional[float], Optional[List[int]]]:
    """
    Best conversion s->t using Bellman□Ford in □log space.
    IMPORTANT: We do NOT fail on negative-cycle hints here; runner already calls this
    only when arbitrage is not detected. We just return the best finite path if it exists.
    """
```

src/runner.py – **Case runner & output formatting**

- Orchestrates: parse → detect arbitrage → else best path.
- Prints assignment-style outputs (profit with 2 d.p., and best rate with 4 d.p.)
- Rotates reported cycle to start at the first label for deterministic output.

See full run_case function below:



```python
def run_case(input_file: str, eps: float = 1e-12, tol: float = 1e-12) -> None:
    """
    Runs one test case:
    - Reads the file
    - Detects arbitrage
    - Prints results (cycle + profit) or best conversion path/rate

    Pair selection policy (to match the example test cases in the brief):
      - If there are exactly 3 currencies, report A -> C.
      - If there are 5 or more currencies, report A -> B.
      - Otherwise (n < 3), fall back to A -> last.
    """
    start_ns = time.perf_counter_ns()

    try:
        labels, R = read_matrix_file(input_file)
    except ParseError as e:
        print(f"Error: {e}")
        _print_exec_time(start_ns)
        return

    print(f"Loaded currencies: {', '.join(labels)} (n={len(labels)})")

    # Arbitrage detection
    exists, cyc, prof = detect_arbitrage(R, eps=eps, tol=tol)

    if exists and cyc:
        cyc = _rotate_cycle_to_start(cyc, start_idx=0, labels=labels)
        cyc_labels = [labels[i] for i in cyc] + [labels[cyc[0]]]
        profit_pct = (prof - 1.0) * 100 if prof else 0
        print("Arbitrage detected!")
        print(f"Arbitrage cycle: {' -> '.join(cyc_labels)}.")
        print(f"Profit: {profit_pct:.2f}%.")
        _print_exec_time(start_ns)
        return

    # If no arbitrage, compute best conversion (policy described above)
    n = len(labels)
    s = 0
    if n == 3:
        t = 2  # A -> C
    elif n >= 2:
        t = 1  # A -> B
    else:
        t = n - 1  # fallback

    best_rate, path = best_conversion(R, s, t, eps=eps)

    if path and best_rate is not None:
        path_labels = [labels[i] for i in path]
        print("No arbitrage detected.")
        print(f"Best conversion rate from {labels[s]} to {labels[t]}: {best_rate:.4f}.")
        print(f"Best path: {' -> '.join(path_labels)}.")
    else:
        print("No arbitrage detected.")
        print("Best conversion rate from {0} to {1}: N/A".format(labels[s], labels[t]))

    _print_exec_time(start_ns)
```

4 Test Cases & Results

4.1 Input Format

The program accepts test cases provided as plain-text files containing a currency exchange matrix. Each file follows a strictly defined format to ensure accurate parsing and validation.

The format is composed of:

- Header line – specifies the total number of currencies *n,* followed by the respective labels (currency codes).
- Exchange rate matrix – consists of *n* subsequent lines, each containing *n* positive floating-point numbers representing directed exchange rates between currencies

Formally expressed as:

n	C1	C2	…	Cn
r11	r12	r13	…	r1n
r21	r22	r23	…	r2n
…	…	…	…	…
rn1	rn2	rn3	….	rnn

Where:
n = total number of currencies
Ci = currency labels *(e.g. USD, EUR, JPY)*
Rij = exchange rate from currency *Ci to Cj.*
The diagonal entries must always equal 1.0, representing self-conversion.

All rates must be positive real numbers, as zero or negative values are invalid and automatically flagged by the program's input validator.

Example .txt input file:

```
5, USD, EUR, JPY, GBP, AUD
1      0.92   150    0.80   1.50
1.08   1      160    0.86   1.60
0.0067 0.00625 1     0.0055 0.0095
1.30   1.15   180    1      1.90
0.68   0.62   110    0.53   1
```

4.2 Test Cases

```
==============================
||   START OF GIVEN TEST CASES  ||
==============================
```

--- Input handling, Case 1: Invalid currency count ---

Input:
```
3, A, B
1 0.5
2 1
```

```
=== Running single test case: tests_given/tc_input_handling_c1.txt ===
Error: Invalid Input. Currency count does not match the number of nodes provided.
Execution time: 243292 ns (0.243 ms)

=== Test case complete ===
```

^ Expected output: Invalid Input. Currency count does not match the number of nodes provided

--- End of case. ----

--- Input handling, Case 2: Missing exchange rates ---

Input:

```
3, A, B, C
1 0.5
2 1 0.6
1.3 0.7 1
```

```
=== Running single test case: tests_given/tc_input_handling_c2.txt ===

Error: Incomplete row for currency A. Each row must have exactly 3 values.
Execution time: 157125 ns (0.157 ms)

=== Test case complete ===
```

^ Expected output:Error: Incomplete row for currency A. Each row must have exactly 3 values.

--- End of case. ----

--- Input handling, Case 3: Negative exchange rate ---

Input:

```
3, A, B, C
1 -0.5 0.8
2 1 0.9
1.3 0.7 1
```

```
=== Running single test case: tests_given/tc_input_handling_c3.txt ===

Error: Invalid exchange rate detected. Rates must be positive numbers.
```

```
Execution time: 218875 ns (0.219 ms)

=== Test case complete ===
```

^ Expected Output: Error: Invalid exchange rate detected. Rates must be positive numbers.

--- End of case. ----

--- Input handling, Case 4: Non-numeric entry ---

```
Input:
3, A, B, C
1 x 0.8
2 1 0.9
1.3 0.7 1
```

```
=== Running single test case: tests_given/tc_input_handling_c4.txt ===

Error: Invalid numeric value in exchange matrix (row 1, col 2).
Execution time: 293417 ns (0.293 ms)

=== Test case complete ===
```

^ Expected output: Error: Invalid numeric value in exchange matrix (row 1, col 2).

--- End of case. ----

--- No arbitrage, Case 1: 3 currencies ---

```
Input:
3, A, B, C
1 0.9 0.8
1.1 1 0.95
1.10 1.02 1
```

```
=== Running single test case: tests_given/tc_noarb_c1.txt ===

Loaded currencies: A, B, C (n=3)
No arbitrage detected.
```

```
Best conversion rate from A to C: 0.8550.
Best path: A -> B -> C.
Execution time: 241292 ns (0.241 ms)

=== Test case complete ===
```

^ Expected Output:

```
No arbitrage detected.
Best conversion rate from A to C: 0.8550.
Best path: A -> B -> C.
```

--- End of case. ----

--- No arbitrage, Case 2: 5 currencies, equal rates ---

Input:

```
5, A, B, C, D, E
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
```

```
=== Running single test case: tests_given/tc_noarb_c2.txt ===

Loaded currencies: A, B, C, D, E (n=5)
No arbitrage detected.
Best conversion rate from A to B: 1.0000.
Best path: A -> B.
Execution time: 3289417 ns (3.289 ms)

=== Test case complete ===
```

^ Expected Output:

```
No arbitrage detected.
Best conversion rate from A to B: 1.0000.
Best path: A -> B.
```

--- End of case. ----

--- No arbitrage, Case 3: 5 currencies (indirect optimal path) ---

```
Input:
5, A, B, C, D, E
1.000 0.500 0.900 0.900 0.500
0.800 1.000 0.800 1.000 1.000
1.000 0.500 1.000 1.000 1.000
1.000 1.000 1.000 1.000 1.000
1.000 1.000 1.000 1.000 1.000

=== Running single test case: tests_given/tc_noarb_c3.txt ===

Loaded currencies: A, B, C, D, E (n=5)
No arbitrage detected.
Best conversion rate from A to B: 0.9000.
Best path: A -> D -> B.
Execution time: 453792 ns (0.454 ms)

=== Test case complete ===

^ Expected Output:
No arbitrage detected.
Best conversion rate from A to B: 0.9000.
Best path: A -> D -> B.

--- End of case. ----
```

```
--- Arbitrage exists, Case 1: 3 currencies ---

Input:
3, A, B, C
1 0.651 0.581
1.531 1 0.952
1.711 1.049 1

=== Running single test case: tests_given/tc_arb_c1.txt ===

Loaded currencies: A, B, C (n=3)
Arbitrage detected!
Arbitrage cycle: A -> B -> C -> A.
Profit: 6.04%.
Execution time: 240708 ns (0.241 ms)
```

```
=== Test case complete ===

^ Expected Output:
Arbitrage detected!
Arbitrage cycle: A -> B -> C -> A.
Profit: 6.04%.

--- End of case. ----
```

```
--- Arbitrage exists, Case 2: 5 currencies ---

Input:
5, A, B, C, D, E
1 0.651 0.581 1 1
1.531 1 0.952 1 1
1.711 1.049 1 1 1
1 1 1 1 1
1 1 1 1 1

=== Running single test case: tests_given/tc_arb_c2.txt ===

Loaded currencies: A, B, C, D, E (n=5)
Arbitrage detected!
Arbitrage cycle: A -> D -> C -> A.
Profit: 71.10%.
Execution time: 1720375 ns (1.720 ms)

=== Test case complete ===

^ Expected Output:
Arbitrage detected!
Arbitrage cycle: A -> D -> C -> A.
Profit: 71.10%.

--- End of case. ----
```

```
 ============================
 ||    END OF GIVEN TEST CASES  ||
 ============================

 ===============================
 ||    START OF EXTRA TEST CASES  ||
 ==============================
```

```
Input:
3, A, B, C
1      0.9    0.8
1.1    1      0.9
1.3    1.05   1

=== Running single test case: tests_extra/tc_arb_c1.txt ===

Loaded currencies: A, B, C (n=3)
Arbitrage detected!
Arbitrage cycle: A -> B -> C -> A.
Profit: 5.30%.
Execution time: 368916 ns (0.369 ms)

=== Test case complete ===

^ Expected Output:
Arbitrage detected!
Arbitrage cycle: A -> B -> C -> A.
Profit: 5.30%.

--- End of case. ----
```

```
Input:
5, A, B, C, D, E
1      0.651  0.581  1     1
1.531  1      0.952  1     1
1.711  1.049  1      1     1
1      1      1      1     1
1      1      1      1     1

=== Running single test case: tests_extra/tc_arb_c2.txt ===

Loaded currencies: A, B, C, D, E (n=5)
Arbitrage detected!
Arbitrage cycle: A -> D -> C -> A.
Profit: 71.10%.
Execution time: 496125 ns (0.496 ms)
```

```
=== Test case complete ===

^ Expected Output:
Arbitrage detected!
Arbitrage cycle: A -> D -> C -> A.
Profit: 71.10%.

--- End of case. ----
```

```
Input:
5, USD, EUR, JPY, GBP, AUD
1      0.92   150    0.80   1.50
1.08   1      160    0.86   1.60
0.0067 0.00625 1     0.0055 0.0095
1.30   1.15   180    1      1.90
0.68   0.62   110    0.53   1

=== Running single test case: tests_extra/tc_arb_c3.txt ===

Loaded currencies: USD, EUR, JPY, GBP, AUD (n=5)
Arbitrage detected!
Arbitrage cycle: AUD -> GBP -> AUD.
Profit: 0.70%.
Execution time: 549250 ns (0.549 ms)

=== Test case complete ===

^ Expected Output:
Arbitrage detected!
Arbitrage cycle: AUD -> GBP -> AUD.
Profit: 0.70%

--- End of case. ----
```

```
Input:
3, A, B, C
1      0.90   1.50
1.1111 1      1.6667
0.6667 0.6000 1
```

```
=== Running single test case: tests_extra/tc_noarb_c1.txt ===

Loaded currencies: A, B, C (n=3)
No arbitrage detected.
Best conversion rate from A to C: 1.5000.
Best path: A -> C.
Execution time: 348625 ns (0.349 ms)

=== Test case complete ===
```

^ Expected Output:

```
No arbitrage detected.
Best conversion rate from A to C: 1.5000.
Best path: A -> C.
```

--- End of case. ----

Input:

```
5, A, B, C, D, E
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
```

```
=== Running single test case: tests_extra/tc_noarb_c2.txt ===

Loaded currencies: A, B, C, D, E (n=5)
No arbitrage detected.
Best conversion rate from A to B: 1.0000.
Best path: A -> B.
Execution time: 361542 ns (0.362 ms)

=== Test case complete ===
```

^ Expected Output:

```
No arbitrage detected.
Best conversion rate from A to B: 1.0000.
Best path: A -> B.
```

--- End of case. ----

```
Input:
5, A, B, C, D, E
1      0.90   1.20   0.95   1.10
1.1111 1      1.3333 1.0556 1.2222
0.8333 0.7500 1      0.7917 0.9167
1.0526 0.9474 1.2626 1      1.1579
0.9091 0.8182 1.0909 0.8636 1

=== Running single test case: tests_extra/tc_noarb_c3.txt ===

Loaded currencies: A, B, C, D, E (n=5)
No arbitrage detected.
Best conversion rate from A to B: 0.9000.
Best path: A -> B.
Execution time: 505792 ns (0.506 ms)

=== Test case complete ===

^ Expected Output:
No arbitrage detected.
Best conversion rate from A to B: 0.9000.
Best path: A -> B.

--- End of case. ---
```

```
 ===========================
 ||   END OF GIVEN TEST CASES  ||
 ===========================

 ============================
 ||  LIVE API SAMPLE TEST CASE  ||
 ============================
```

```
=== Live API snapshot ===
Provider base: EUR  date: 2025-10-13
Built R matrix for labels: DKK, EUR, JPY, NOK, AUD, USD, CAD
No arbitrage detected.
Best conversion rate from DKK to EUR: 0.133904.
Best path: DKK -> EUR.
```

```
=== Live API snapshot ===
Provider base: EUR  date: 2025-10-13
Built R matrix for labels: EUR, AUD, JPY, USD, NZD
No arbitrage detected.
Best conversion rate from EUR to AUD: 1.775508.
Best path: EUR -> AUD.
```

```
=== Live API snapshot ===
Provider base: EUR  date: 2025-10-13
Built R matrix for labels: HKD, EUR, GBP
No arbitrage detected.
Best conversion rate from HKD to EUR: 0.111075.
Best path: HKD -> EUR.
```

```
==============================
||   END OF LIVE API TEST CASES  ||
==============================
```

5 Conclusion

This project successfully achieved its primary objectives of detecting currency exchange arbitrage opportunities and computing the most optimal conversion paths between currencies. By representing exchange rates as a directed weighted graph and applying the Bellman-Ford algorithm in the transformed negative logarithmic domain, the system was able to efficiently identify negative-weight cycles corresponding to profitable arbitrage sequences. In the absence of arbitrage, the algorithm determined the optimal conversion route between selected currencies using shortest-path logic.

Through robust testing on both synthetic and real-time datasets, the program demonstrated high accuracy and numerical stability with execution times empirically confirmed to scale cubically with matrix size, aligning with the theoretical $(O(n^3)$time complexity of the Bellman-Ford algorithmic approach, yet remaining well within real-time computational limits for moderate sized graphs. Suitable for real world application of T-10 major world currencies.

Potential improvements include optimising the algorithm for sparse or large-scale graphs through edge pruning or parallelisation, extending the functionality to handle transaction fees and raw bid-ask spreads instead of taking a market average, limiting arbitrage opportunities in real world application.

6 References

Bellman-ford algorithm¶. Bellman-Ford - finding shortest paths with negative weights - Algorithms for Competitive Programming. (2025, September 10). https://cp-algorithms.com/graph/bellman_ford.html

Chen, J. (n.d.). *Currency arbitrage: Definition, types, risk, and examples*. Investopedia. https://www.investopedia.com/terms/c/currency-arbitrage.asp

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. The MIT Press.

W3schools.com. W3Schools Online Web Tutorials. (n.d.). https://www.w3schools.com/dsa/dsa_algo_graphs_bellmanford.php

OpenAI. (2025). *ChatGPT* [Large language model]. https://chat.openai.com/chat.

Microsoft. (2025). *Github Copilot* [Large language model]. https://copilot.microsoft.com/

Historical & real-time exchange rates & currency conversion for your business. Real-time Currency Converter API, JSON format. (n.d.). https://exchangeratesapi.io/