**

**COMP611 – ADA**

**Software Assignment Report**

|  |  |
| --- | --- |
| **Experiment Date** | **16 September 2024** |
| **Submitted Date** | **16 September 2024** |
| **Name(s)/ID(s)** | **Jeri Camacho 22180058** |
| **Project title** | **Currency Exchange Analyzer** |
| **Project objectives** | Detect arbitrage opportunities and find the best conversion rate between currencies using dynamic programming algorithms. |
| **Configurations and settings** | **Use of JUnit 4 for testing**  **VSCode for coding environment with ‘Test Runner for Java’ extension** |
| **Methods** | **Bellman-Ford** and **Floyd-Warshall** algorithms were used to detect arbitrage opportunities and find optimal currency conversion rates. |
| **Implementation steps/flowchart** | 1. Start 2. Input Currency data and exchange rates 3. Create Currency exchange graph 4. Convert exchange rates to log graph 5. Task 1: Detect Arbitrage (Bellman-Ford algo) output: Arbitrage detected or not 6. Task 2: Best Conversion Rate (Floyd-Warshall) output: Best conversion rate and path 7. End |
| **Datasets** | Data used in experiment was real-world currency and real-time exchange rates sourced from <https://www.bloomberg.com/markets/currencies/cross-rates>  The rates were used to simulate real-life scenarios to test both arbitrage detection and also searching for best conversion rates. This diverse set of exchange rates allowed for test cases that had scenarios both with and without arbitrage opportunities helping increase the robustness and realism of the experiment |
| **Input (image, file, etc.)** | **Test Case 8 Example Input Code:** |
| **Output (image, file, etc.)** | **Test 1 Test Cases Results:**  **Test 2 Test Cases Results:** **A screenshot of a computer  Description automatically generated** |
| **Testing steps** |  **Setup the Test Environment:**   * Install and configure JUnit 4.13.2 and Hamcrest Core 1.3 for unit testing. * Make sure the project is setup correctly in the IDE (VS Code) with the necessary .jar files in the lib directory. * Prepare the settings.json file to enable the test runner in VS Code.    **Write Unit Tests for Task 1 (Arbitrage Detection):**   * **Test Case 1: No Arbitrage (Five Currencies):**   + Use synthetic exchange rates that have no arbitrage opportunities.   + Verify that the ArbitrageDetector class correctly identifies that no arbitrage exists. * **Test Case 2: Arbitrage Exists (Five Currencies):**   + Use synthetic exchange rates where an arbitrage opportunity is present.   + Verify that the ArbitrageDetector class identifies and prints the arbitrage cycle. * **Test Case 3: Real-World Exchange Rates (No Arbitrage):**   + Use real-world exchange rates from Bloomberg to ensure no arbitrage is detected. * **Test Case 4: Real-World Exchange Rates (Arbitrage Exists):**   + Alter real-world rates hypothetically to introduce an arbitrage opportunity and verify its detection.    **Write Unit Tests for Task 2 (Finding the Best Conversion Rate):**   * **Test Case 5: Direct Conversion (Five Currencies):**   + Test scenarios where a direct exchange exists between two currencies.   + Verify the output to ensure the BestConversionRateFinder class returns the direct conversion rate. * **Test Case 6: Intermediate Conversion (Five Currencies):**   + Create scenarios where the best conversion path includes intermediate currencies.   + Verify that the BestConversionRateFinder identifies the correct sequence of conversions. * **Test Case 7: Real-World Exchange Rates (Direct Conversion):**   + Utilize current real-world exchange rates and verify the direct conversion rate is calculated accurately. * **Test Case 8: Real-World Exchange Rates (Intermediate Conversion):**   + Use real-world exchange rates to find the best conversion rate via intermediate conversions.    **Run All Tests:**   * Run all the test cases in both Task1TestCases.java and Task2TestCases.java using the JUnit test runner in VS Code. * Check the output for each test to ensure they pass as expected.    **Validate Outputs:**   * Make sure the console outputs match the expected results for each test. * Confirm that test assertions correctly validate the behavior of the methods under test. |
| **Bugs or problems** | 1. **JUnit Library Setup Issues:**  * One of the main problems was failing test units due to missing JUnit libraries. Initially, the required .jar files (junit-4.13.2.jar and hamcrest-core-1.3.jar) were not correctly added to the lib directory. * As a result, the test classes could not be compiled or executed, throwing errors such as "package org.junit does not exist" and "cannot find symbol: class Test." * This issue was resolved by ensuring the correct versions of JUnit and Hamcrest Core were included in the project's lib folder.  1. **VS Code Settings Misconfiguration:**  * Another challenge was the incorrect setup of the settings.json file in VS Code. Without the proper configuration, the test runner was unable to detect and run the test files. * This caused errors when attempting to execute the test cases, which prevented the validation of code functionality. * The problem was fixed by setting up the settings.json file appropriately to recognize the JUnit test framework within the project.   These issues delayed testing process and made it hard to check if code was accurate but once the library and settings were corrected the tests were able to run smoothly. |
| **Result analysis** | **Test Table of Results:** |
| **Conclusion/Reflection** | **Strengths**  One of the strengths of this project was learning to use dynamic programming algorithms like Bellman-Ford and Floyd-Warshall. These algorithms were crucial in detecting arbitrage opportunities and finding the best currency conversion rates. By incorporating real-world exchange rates into the program, the software became more practical and gave insight into how actual currency exchanges work. This also helped in understanding how slight differences in rates can lead to significant financial outcomes.  **Weaknesses**  A key weakness I encountered was dealing with the precision of floating-point numbers. Even small rounding errors could lead to incorrect detection of arbitrage opportunities, which required setting appropriate tolerance levels in the tests. Another challenge was setting up the testing environment correctly. At first, I had issues with JUnit not working properly because the required JAR files were missing or the settings were not configured correctly. This caused some tests to fail unexpectedly, which was frustrating to troubleshoot.  **What I learned**  Through this project, I learned the importance of thorough testing and setup in software development. It taught me how vital it is to configure the development environment correctly to avoid errors that are not directly related to the code itself. I also learned how to think critically about how minor changes in data, such as exchange rates, can have a big impact on the results. Overall, this project helped me improve my programming and problem-solving skills, especially when it comes to working with real-world data and complex algorithms. |
| **References** | <https://www.bloomberg.com/markets/currencies/cross-rates> <https://mvnrepository.com/artifact/junit/junit/4.13.2> |

**Note**: Max two pages

**Appendix**: Source code with comments and line numbers