**3. Apply a Wide Variety of Testing Techniques and Compute Test Coverage and Yield**

Since the website node is closed, my original ilp code is no longer usable, so I can only modify it a lot to use local data. This data is placed in the root directory of the shared file folder.

**3.1 Range of Techniques**

In our App, its early foundational methods (such as determining position and direction) have already undergone unit testing as part of the coursework. These tests have been refined and validated, ensuring that these modules are optimized to provide a reliable foundation for subsequent testing. At this stage, the focus of testing has shifted to verifying the overall functionality of the program and the core pathfinding algorithm, encompassing the complete workflow from order processing to route planning. Through systematic testing, we ensure that the program operates correctly in real-world scenarios and meets the requirements while validating the core algorithms and the interactions between modules.

I implemented four types of tests to ensure the system's stability. You can find the test with the same name in the “test” folder:

Functional Test  
The purpose of functional testing is to verify whether the system's core functional modules operate according to design requirements, ensuring that input data is correctly parsed and generates the expected output. This test primarily focuses on the accuracy of order processing and route planning functionalities.

In functional tests, we validate the correctness of order data, including verifying the total price of orders, restaurant matching, and the generation of delivery routes. The test scenarios include processing valid orders, testing different starting and ending locations, and simulating erroneous data to check error handling mechanisms. The results of these tests help ensure the stability and reliability of the system’s core functionalities.

Pipeline Test  
Pipeline testing aims to verify the end-to-end workflow, ensuring that the interaction and data transfer between different modules function seamlessly.

During testing, we load order information from local data files and pass them through various processing stages of the system. The final output files (e.g., flightpath-\*.json) are checked to ensure they meet the expected format and content. Typical test scenarios include processing a batch of orders, validating whether the generated routes are correct, and ensuring that paths avoid no-fly zones when necessary. Pipeline testing ensures the overall workflow integrity and usability of the system.

Stress Test  
The goal of stress testing is to evaluate the system's performance under high-load conditions, ensuring that it maintains reasonable response times and stability while handling large volumes of orders.

In this test, we gradually increase the number of orders (e.g., 10, 100, 1000, 10000) and observe processing times and resource consumption. Under stress testing, we analyze the system's bottlenecks when processing large-scale orders, including CPU and memory usage, and assess its robustness under high concurrency. The results of stress tests help identify system performance limits and optimize efficiency.

Smoke Test  
Smoke testing is designed to quickly verify the basic availability of key functionalities, ensuring that no critical defects are introduced after code changes and that the system can operate correctly with minimal input.

This test is mainly used for regression testing after functionality updates or system deployments, quickly executing a series of critical function checks such as order input, path calculation, and no-fly zone avoidance to confirm that the system is functioning as expected. Smoke tests focus on whether the entire system "can run" rather than the detailed correctness of logic, serving as an initial validation step in the development workflow.

**3.2 Evaluation Criteria for the Adequacy of the Testing**

The adequacy of the testing process was assessed using key metrics such as statement coverage, branch coverage, and line coverage. The goal was to achieve high levels of code coverage across critical components, ensuring comprehensive validation of the system’s functionality. Based on the JaCoCo code coverage report:

Overall project coverage:

90% class coverage and 81% line coverage, indicating that most of the project's logic was exercised through tests.

Module-level coverage:

record package: Achieved 100% class and line coverage, ensuring that all records used within the application were fully tested.

App class: Covered 90% of methods and 77% of lines, validating core functionalities such as data processing and workflow execution.

Astar class: Achieved 75% method coverage and 86% line coverage, reflecting thorough testing of the pathfinding logic with some edge cases needing further attention.

CardDeserializer class: Reached 100% method and line coverage, indicating complete verification of credit card information processing.

LngLatHandler class: Attained 83% method and line coverage, ensuring robustness in coordinate handling operations.

order class: Achieved 100% method coverage and 77% line coverage, verifying the order processing logic comprehensively.

point class: Attained 72% method coverage and 79% line coverage, suggesting the need for additional test cases to cover corner cases.

test package: Currently lacks coverage, with 0% methods and lines covered, highlighting the need for more test implementations.

These results indicate that the core components of the system, such as data handling and pathfinding, have been well-tested. However, efforts should be made to improve test coverage for lower-covered modules such as point and to include more unit tests within the test package.

**文本

描述已自动生成**

Evaluation Criteria for the Adequacy of the Testing

**3.3 Results of Testing**

The testing process provided valuable insights into the system's functionality and performance. The Astar.findpath algorithm demonstrated its capability to successfully avoid no-fly zones and compute valid paths in scenarios where feasible routes existed, proving its robustness and efficiency. In cases where a path was not possible, the system handled the situation appropriately by returning null or an empty path, reflecting proper error handling mechanisms.

Several tests revealed specific edge cases that affected path generation efficiency, especially when navigating through dense no-fly zones or when the start or end locations were in close proximity to obstacles. These findings highlighted potential areas for algorithmic optimization, particularly in terms of reducing computational overhead and improving real-time response.

The code coverage analysis, conducted using JaCoCo, indicated that core components, including order processing and pathfinding, were comprehensively tested. The highest coverage was achieved in essential modules such as CardDeserializer and order, which attained 100% method coverage, while other critical modules like Astar reached 86% line coverage, indicating thorough testing of the core logic.

Furthermore, the structural tests confirmed that all key execution paths were covered, ensuring the correctness of the internal logic. The robustness of the order validation process was affirmed through test cases simulating various invalid inputs, including incorrect card details and pricing mismatches. Throughout the testing phase, no critical defects were detected in the primary functional workflows.

文本

描述已自动生成文本

描述已自动生成

文本

描述已自动生成

**3.4 Evaluation of the Results**

The overall results of the testing demonstrate the strengths of the system, particularly in its ability to efficiently handle pathfinding across complex scenarios involving multiple no-fly zones. The algorithm consistently generated optimal paths in the majority of tested conditions, showcasing its robustness and correctness. Additionally, the system's validation mechanisms ensured that erroneous orders were correctly identified and handled, contributing to overall reliability.

Despite the strong performance, the testing process identified several areas for improvement, particularly in optimizing pathfinding when dealing with highly obstructed environments. The performance evaluation showed that as the number of obstacles increased, the computational time and memory usage grew significantly, highlighting the need for further optimizations in terms of scalability.

Future improvements should focus on enhancing the algorithm's efficiency when handling larger datasets and more complex scenarios. Additionally, the test coverage should be extended to less-covered modules, such as point, to ensure complete validation of all system components.

Overall, the testing results confirm that the system is well-prepared for deployment in real-world applications, with potential for further optimization to enhance scalability and performance under extreme conditions.