Evaluation of testing

Limitations

The testing process for this portfolio, while robust in many areas, has some notable omissions and deficiencies. For R1, unit testing was extensively performed for validating individual components (credit card validation, restaurant validation and order validation), but integration testing was limited. Specifically, the interaction between validation and further processes such as file generation and pathfinding was not comprehensively tested due to time constraints. For R2, system-level testing covered edge cases for no-fly zones and high-density configurations but did not account for irregularly shaped or dynamically changing no-fly zones. For R3, performance testing focused on average runtime but did not assess how the system scales with concurrent requests or unexpected spikes in load.

Additionally, while system-level testing covered important edge cases for no-fly zones and runtime performance, some edge cases, such as extremely high-density configurations or irregular user-defined no-fly zones, were not exhaustively addressed due to time constraints. Creating more detailed simulation tests could have improved the robustness of the code as the simulation tests that I created were limited to a few orders with only several no fly zones as I couldn't manually create lots of them. Even though I created a function that generated no fly zones, it was a simple one that didn't create varied sized no fly zones. For better testing, creating an order generator would have alos been beneficial to further test robustness of the system, as having a lot of orders would have tested the system under high intensity.

Further, doing more integration testing and in a lot more detail, could've helped

Future improvement could focus on:

- Expanding integration testing
- Including automated tools for load and stress testing
- Simulating real-world dynamic scenarios such as changing no-fly zones in real-time

R1

I had extensively tested this requirement and from the results seen from the table and tests. I ran tests with coverage and as can be seen from the table, my system got 100% for class, method and line coverage for the credit card information check, pizza information check and restaurant information check classes. This indicates that all logical paths in these components were executed during testing, confirming that the system handles both standard and edge cases effectively. The unit tests focused on validating individual fields, such as the format of credit card information, the validity of pizza orders (price and quantity constraints), and restaurant attributes (menu consistency and opening hours). These tests made sure that any invalid data would be identified early in the validation pipeline, increasing the system's reliability.

The high coverage metrics for these classes indicate strong alignment with the planned testing approach, aligning with the reliability and robustness of this requirement. However, integration tests between these validation components and the downstream processes, such as pathfinding and file generation, were more limited. This could leave potential gaps in verifying the seamless flow of valid orders through the entire system.

Class	Coverage wanted	Coverage achieved
Credit card information check	100 %	100%

Restaurant information check	100 %	100 %
Pizza information check	100 %	100 %

R2

Testing for R2 demonstrated a strong focus on edge cases and varying densities of no-fly zones, as evidenced by the Number of Pathfinding Failures vs. No-Fly Zone Density analysis and performance tests. The tests incorporated both synthetic data and realistic scenarios, ensuring a robust evaluation of the pathfinding algorithm's ability to generate valid and optimal routes under constraints. I tested R2 using simulation tests and to eliminated unnecessary test cases based on the requirements, I used category partitioning to do that. This left me with a certain set of test cases that I ran on my code. I varied the amount of no fly-zones by using a generator (though very simplified), to provide the specified number of no-fly zones in a given shape and size. This way I was able to test the A* algorithm calculation and route to see if the path was always found in the 60 sec runtime constraint. The results show that the path was found in most cases, however the amount of times the path wasn't found in the given timeframe increased with the amount of no-fly zones.

The achieved coverage for AStarFlightPathSolver was 100% for classes and methods, but line coverage reached 72%, as some error-handling pathways were not executed. Specifically, the untested lines pertain to the condition where a neighbouring node is already visited, but a newly calculated tentativeG value is lower than the existing g value for that neighbour. This condition, which optimises path recalculations by updating the neighbour's parent and heuristic values, was not triggered during testing, which is a limitation of the testing that could be addressed with more time. Overall the coverage indicates that the system's robustness was well-tested for general cases but could benefit from additional focus on rare edge cases and untested paths. It could be beneficial for system robustness to introduce additional system-level tests for seamless integration between validation, pathfinding, and file generation processes.

Class	Coverage wanted (class)	Coverage achieved (class)	Coverage wanted (method)	Coverage achieved (method)	Coverage wanted (line)	Coverage achieved (line)
A star flight path solver	100 %	100%	100%	100%	100%	72%

R3

The testing process for R3 primarily focused on performance testing under various conditions. I evaluated the runtime using synthetic data and real-world API responses to analyse how the system performed under both situations. This testing approach revealed that the system follows the runtime constraint in all cases, with average runtimes well below the threshold. For example, performance testing showed an average runtime of 415.5 ms across different no-fly zone configurations with synthetic data and an average runtime of 1008.4 ms across different no-fly zone configurations through incorporating the REST API service as well.

I further went on to test the effect of different amounts of no-fly zones on performance which showed that the runtime was still way below the threshold in all cases. The time did have an increase at 25+ no fly zones and this could be tested further with stress testing under a significantly larger amount of the no-fly zones.

Results compared to target

The achieved coverage metrics indicate that testing targets were largely met, with class and method coverage exceeding the targets. The classes that got 0% in coverage tests most surround the API requirement which wasn't tested for this project due to time constraints such as the GetClient class.

Statistical Analysis:

- **Pass Rate Analysis for performance:** Across 40 test cases, 40 succeeded, yielding a success rate of 100%
- **Runtime Variability:** Performance testing with synthetic data showed an average runtime of 415.5 ms with a standard deviation of 42 ms across varying no-fly zone configurations
- **Failures Analysis:** Failures primarily occurred with high densities of no-fly zones, highlighting areas where the algorithm could be optimised for better performance under stress
- **Line Coverage Deficiency:** Analysis of untested lines revealed a specific scenario in the AStarFlightPathSolver where already visited neighbouring nodes were not revisited under lower cost conditions
- **Runtime Correlation:** A regression analysis indicated a direct correlation between the number of no-fly zones and average runtime ($r = 0.92$), confirming the impact of increased complexity on performance

Future Improvements

To improve coverage and testing effectiveness:

- Implement automated testing frameworks for generating and running edge cases dynamically
- Extend system-level testing to cover rare edge cases, including dynamically changing no-fly zones
- Improve integration testing to make sure seamless data flow and interaction between components
- Use coverage analysis tools to identify and refactor unused or hard-to-reach code, to increase line coverage
- Expand performance testing to include concurrent load scenarios, so that the system sticks to the runtime constraint under various conditions