Software testing project report

Blekinge Tekniska Högskola, Aug 2023

Hammam Dowah - Karam Kirawan - Abdulkarim Dawalibi - Adnan Altukleh

1. Introduction

The objective of this report is to provide an overview of our testing approach for the `minimal_paths.py` and `model.py` files. Additionally, we will detail the configuration of our GitHub Actions setup, which includes testing these files using both Python 3.10 and Python 3.11.

GitHub-link: *https://github.com/adnan32/testing-project*

2. GitHub Actions

Our custom-tailored GitHub Actions workflow automates testing in our project. When there are code changes via push or pull, the workflow steps in for continuous validation. It pays special attention to key test files, test_minimal_paths.py and test_model.py, accommodating different Python versions (3.10 and 3.11) and considering the orjson package presence.

After encountering an error, we made a decision to move our simulation file into the test directory. The reason behind this move was to resolve problems that arose when the simulation file wasn't placed within the test directory. It became clear that the source of the issue was related to a misconfiguration of directories. While we acknowledge the challenge this posed, we faced challenges in trying to tackle it directly within the YAML configuration file. This was due to our limited familiarity with this specific aspect of setting up our workflow.

3. Badges

With the assistance of Codacy, we've achieved an impressive A grade in code quality. The majority of the errors that were identified pertained to improper white spacing, which have now been successfully rectified.

4. Requirements Traceability Matrix

Requirements ID	**Requirements file**	**Requirements description**	**Requirements status**
1	test_model.py	Checks if the number of vertices and specific vertices associated with a hyperedge are correct.	Pass
2	test_model.py	Verifies the count of	Pass

		hyperedges and checks if a specific vertex is associated with a hyperedge.	
3	test_model.py	Ensures that the timing values associated with specific hyperedges are accurate.	Pass
4	test_model.py	Validates that attempting to access hyperedges for an invalid vertex raises an EntityNotFound error.	Pass
5	test_model.py	Validates that attempting to access vertices for an invalid hyperedge raises an EntityNotFound error.	Pass
6	test_model.py	Confirms that the channels associated with a vertex are accurate.	Pass
7	test_model.py	Validates that the participants associated with a channel are accurate.	Pass
8	test_minimal_paths.py	Verifies that the shortest path calculation for a vertex returns the expected results.	Pass
9	test_minimal_paths.py	Validates that the shortest path calculation for a hyperedge returns the expected results.	Pass
10	test_minimal_paths.py	Ensures that the fastest path calculation for a	Pass

		vertex returns the expected results.	
11	test_minimal_paths.py	Validates that the fastest path calculation for a hyperedge returns the expected results.	Pass
12	test_minimal_paths.py	Checks that the path calculation methods for different distance types are not equivalent for various start nodes.	Pass
13	test_minimal_paths.py	Validates that an attempt to perform path calculations on an empty graph raises an EntityNotFound error.	Pass
14	test_model.py	Verifies that the correct number of vertices is returned by the vertices method of the cn instance.	Pass
15	test_model.py	Validates that the correct number of hyperedges is returned by the hyperedges method of the cn instance.	Pass
16	test_model.py	Tests the loading of a communication network from a JSON file and checks that various properties such as the number of participants, channels, vertices, and hyperedges match the expected values.	Pass

5. Tests

5.1 Test minimal_paths.py

We've set up various tests to cover different situations. For instance, we want to check how our code handles different types of distances, like finding the shortest or fastest path. We're also interested in both vertices (individual points) and hyperedges (connections between points) in our communication network.

In each test, we're creating a test communication network with predefined data and timings. We then use the functions single_source_dijkstra_vertices and single_source_dijkstra_hyperedges to calculate the minimal paths. After that, we use the unittest library's assertEqual method to compare the results we've got with the expected outcomes.

We're not only checking the basic cases but also some more complex ones. For example, we're testing if our code treats different starting points in the same way for various distance types. And we're also making sure our code handles errors gracefully. For instance, we're testing how it reacts when dealing with an empty graph.

5.2 Test test_model.py

We begin with a series of tests that examine the connections within the network. For example, we are checking if the vertices and hyperedges are correctly identified, and whether the timings associated with specific hyperedges are accurate. Additionally, we validate the ability of the network to identify channels and participants for each vertex and hyperedge.

Moving on, we have a second section of tests specifically tailored to a predefined communication network example. This is important to confirm that our testing approach is consistent and applicable to different scenarios. We again assess the vertices and hyperedges, ensuring their correctness.

Lastly, we evaluate the handling of real data. The tests involve loading network data from a JSON file, a common format for data storage and exchange. By examining the extracted data, we ascertain whether the network's methods can accurately interpret this data format. This is particularly valuable as it allows us to work with real-world data and validate the robustness of our network implementation.

6. *Discussion*

6.1 Testing the Runtime and Memory Performance of Python Applications

The evaluation of software application efficiency relies heavily on runtime and memory performance testing. These tests have a direct influence on user experience and resource allocation. To ensure a thorough examination, the suggested approach is as follows:

a. Dynamic Runtime Testing:

Implementing dynamic runtime testing, which focuses on real-time monitoring of the application's performance during execution. This includes tracking execution times, response times, and transaction throughput. By incorporating real-world scenarios, we can measure how the application performs under varying workloads and user interactions.

b. Resource Consumption Profiling:

Accurate memory usage assessment is crucial for identifying memory leaks and optimizing memory consumption. This approach focuses on profiling the application's resource consumption during runtime. The steps include:

- Memory Profiling: Implement memory profiling tools to track memory allocation and deallocation patterns. This helps identify memory leaks, excessive memory usage, and areas for optimization.

- Heap Analysis: Perform heap analysis to visualize memory allocation patterns, object lifetimes, and potential memory fragmentation issues.

- Memory Limits: Define acceptable memory thresholds based on available system resources and application requirements. This ensures that memory consumption remains within acceptable bounds.

- Memory Leak Detection: Implement automated memory leak detection tests to identify memory leaks as early as possible during development.

By adopting this approach, we ensure that both runtime and memory aspects of the application are thoroughly evaluated. The dynamic runtime testing provides insights into real-world performance, while resource consumption profiling focuses on memory optimization and leak detection.

6.2 Possible improvements

The tests we conducted didn't focus on how the simulation spread information. Instead, they focused on observing how the hypergraph acted in separate, distinct steps. Due to time constraints, we didn't fully grasp the expected outcome of the simulation. As a result, we limited our testing to the hypergraph's mocked data.

For now, considering our limited resources, we could expand testing using generated mocked datasets. This would provide a more accurate assessment of the model's behavior and minimal paths. Alternatively, we could generate dynamic mock data in real-time, based on the current state of the actual simulation.

Regarding GitHub Actions workflow, The GitHub flow is triggered by both pushes and pulls. Currently, it runs all tests in the test folder. However, an improvement could involve configuring GitHub to exclusively run tests on newly added or modified files.

7. Credit

Contributions of each team member during this project and report

Adnan Altukleh: GitHub Actions and Requirements traceability Matrix
Abdulkarim Dawalibi: Badges and Minimal-paths tests
Hammam Dowah: Model.py tests
Karam Kirawan: Minimal-paths tests