Verification and Validation Report
An AI-based Approach to Designing Board Games
SE 4G06

Team #6, Board Gamers
Ilao Michael, ilaom
Bedi Hargun, bedih
Dang Jeffrey, dangj12
Ada Jonah, karaatan
Mai Tianzheng, mait6

April 6, 2023

1 Revision History

Date	Version	Notes
March 5th	1.0	Initial Division of Work
March 6th	1.1	NFR Eval finished
March 7th	1.2	Traceability Matricies finished
April 4th	1.3	Rev 1 based on feedback & GitHub issues

Contents

List of Figures

2 Symbols, Abbreviations and Acronyms

~~symbol~~Symbol	~~description~~Description
VnV	Verification and Validation
AI	Artificial Intelligence

3 Terminology and Definitions

This section is expressed in words, not with equations. It provides the meaning of the different words and phrases used in the domain of the problem. The terminology is used to introduce concepts from the world outside of the mathematical model. The terminology provides a real-world connection to give the mathematical model meaning.

This subsection provides a list of terms that are used in the subsequent sections and their meaning, with the purpose of reducing ambiguity and making it easier to correctly understand the requirements:

- **AI Agent**: Refers to the subsystem that has an AI model trained to play the game at hand acting as one of the players of the game.
- **Game Engine**: Refers to the subsystem that is an abstract representation of the actual game as software.
- **Data Visualization**: Refers to the subsystem that visualizes Game Engine and AI Agent logs.
- **Game State**: Refers to the state of the game, which could include player attributes, player score, and game layout. All attributes and characteristics that change throughout a game simulation are stored here.
- **Observation Space**: Refers to the state of the game that is observable to a Game Agent. (Not all information is available to the Game Agents and can vary from agent to agent)
- **Action Space**: Refers to the set of moves an AI Agent can take in a particular Game Engine.

- **Game Designer**: The game designer is the person who is creating the game by coming up with the rules and writing a scenario for the game. The game designer will be the end user of the system to balance the game and improve the design of the game.

4 Functional Requirements Evaluation

4.1 AI Agents Module

4.2 FR Test 1

4.2.1 Test Requirements

- AI Agent responds to the game state
- Display error message

4.2.2 Testing Process

1. Set in *engine.py* a list of size equivalent to the number of players (for the negative case, add an extra player).
2. Run the simulation in the terminal through *main.py*.

4.2.3 Expected Results

- Standard output would be the continuous running of the simulation with no error code.
- Error message displayed in the terminal indicating invalid observation matrix size.

4.2.4 Actual Results

- **Positive Test Case:** Simulation ran accordingly and results of the simulation were displayed in the terminal.
- **Negative Test Case:** Invalid matrix size error displayed in the terminal.

4.2.5 Results Analysis

Both test cases passed as expected.

4.3 FR Test 2

4.3.1 Test Requirements

- AI Agent is exchangable
- Display error message

4.3.2 Testing Process

1. Set each AI agent to an existing policy (set one to a non-existing policy in the negative case) in *ai.py*.
2. Run the simulation in the terminal through *main.py*.

4.3.3 Expected Results

- Standard output would be the continuous running of the simulation with no error code.
- Error message displayed in the terminal indicating an incorrect policy was chosen.

4.3.4 Actual Results

- **Positive Test Case:** Simulation ran accordingly and results of the simulation were displayed in the terminal.
- **Negative Test Case:** Invalid policy name error displayed in the terminal.

4.3.5 Results Analysis

Both test cases passed as expected.

4.4 FR Test 3

4.4.1 Test Requirements

- AI Agent's moves are observable in the environment

4.4.2 Testing Process

1. Run the simulation in the terminal through *main.py*.

4.4.3 Expected Results

- Standard output is all the AI Agent's moves are logged into a log file in the logs folder.

4.4.4 Actual Results

- **Positive Test Case:** Simulation ran accordingly and results of the simulation were displayed in the terminal.

4.4.5 Results Analysis

The test case has passed as expected.

4.5 FR Test 4

4.5.1 Test Requirements

- AI Agent makes valid moves.
- The system checks the validity of the move.

4.5.2 Testing Process

1. Run the simulation in the terminal through *main.py*.

4.5.3 Expected Results

- Standard output is simulation proceeds as normal.

4.5.4 Actual Results

- **Positive Test Case:** Simulation ran accordingly, results of the simulation were displayed in the terminal and a logs file was produced in the logs folder.

4.6 FR Test 5

4.6.1 Test Requirements

- AI Agent takes observation space as input

4.6.2 Testing Process

1. In *test _engine.py*, initialize an engine and an AI agent, then print both the engine's game state and the observation space of the AI agent.
2. Run the simulation in the terminal through *test_engine.py.*
3. Compare both displayed observation space of the agent and game state of the engine.

4.6.3 Expected Results

- The observation space of the AI agent is only what the AI agent is expected to see and not the entire game state of the engine.

4.6.4 Actual Results

- **Positive Test Case:** Test engine was run successfully and the observation space of the test AI agent was limited to the expected scope.

4.6.5 Results Analysis

The test case has passed as expected.

4.7 FR Test 6

4.7.1 Test Requirements

- The simulation is continuous and does not stall on any AI agents' turn

4.7.2 Testing Process

1. Run the simulation with parameters `--` *training-num 1* `--` *test-num 1* in the terminal through *main.py.*

4.7.3 Expected Results

- Simulation should finish in under 5 minutes.

4.7.4 Actual Results

- **Positive Test Case:** Simulation was finished in under 1 minute and ran accordingly, results of the simulation were displayed in the terminal and logs file was produced in the logs folder.

4.7.5 Results Analysis

The test case passed as the simulation finished within the expected time limit.

4.8 Game Engine Module

4.9 FR Test 7

4.9.1 Test Requirements

- Game engine is properly initialized

4.9.2 Testing Process

1. Ensure in *engine.py* all starting game entities is initialized in the function *__init__* (for An Aged Contrived, this would include monuments, players, the map and current monuments)
2. Run the simulation in the terminal through *main.py*.

4.9.3 Expected Results

- Simulation should complete and generate logs of AI agent actions interacting with the game engine.

4.9.4 Actual Results

- **Positive Test Case:** Simulation was finished and a log was generated demonstrating proper AI agent actions.

4.9.5 Results Analysis

The test case has passed.

4.10 FR Test 8

4.10.1 Test Requirements

- AI Agent makes progress towards an end-game state

4.10.2 Testing Process

1. Initialize a game engine with a game state with an almost completed game in the *test_engine.py.*
2. Assign a very high reward to AI when they complete the reward.
3. Run the test simulation in the terminal through *test_engine.py.*

4.10.3 Expected Results

- Simulation should complete and AI should finish with a very high reward value.

4.10.4 Actual Results

- **Positive Test Case:** Simulation was finished, the terminal displayed a high reward on the AI agent and the log file displayed the reward.

4.10.5 Results Analysis

The test case has passed.

4.11 FR Test 9

4.11.1 Test Requirements

- Game state changes accordingly to AI Agent moves

4.11.2 Testing Process

1. Initialize a game engine with a game state and an AI Agent in the *test_engine.py*.

2. Initialize a game turn on the AI Agent ~~.~~(energy fill in the case of An Age Contrived).

3. Assess the game state of the game engine after the move has been completed. (see if the first monument was filled by the AI agent's action)

4.11.3 Expected Results

- Game state after AI action is changed accordingly based on the AI action.

4.11.4 Actual Results

- **Positive Test Case:** Game engine state was properly changed after the AI action (The game engine's monument was filled properly by the AI Agent's energy fill action).

4.11.5 Results Analysis

The test case has passed.

4.12 FR Test 10

4.12.1 Test Requirements

- Action list should be consistent throughout the simulation

4.12.2 Testing Process

1. In *env.py*, store and display the available action list for every AI agent in the terminal.

2. Check and throw an exception if there are any discrepancies in the action list during the simulation runtime.

4.12.3 Expected Results

- Simulation should continue running without exception errors and all displayed action lists should be consistent throughout the simulation.

4.12.4 Actual Results

- **Positive Test Case:** Simulation continued running and all AI Agents' available action lists remained consistent throughout the simulation.

4.12.5 Results Analysis

The test case has passed.

4.13 FR Test 11

4.13.1 Test Requirements

- AI Agent and Engine are consistent with each other

4.13.2 Testing Process

1. Initialize a game engine with a game state and an AI agent *test_engine.py*. (empty monument will have all unfilled tiles)
2. Initialize a given game move through the AI Agent and display the game state afterwards. (fill monument with Invertible tile)
3. Run the test simulation in the terminal through *test_engine.py* repeatedly (at least three times).
4. Compare each game state after the same AI Agent move was sent to the engine.

4.13.3 Expected Results

- All game states after the AI Agent move remains consistent.

4.13.4 Actual Results

- **Positive Test Case:** Each trial's game engine end-game state showed the monument filled with the AI Agent's Invertible tile.

4.13.5 Results Analysis

The test case has passed.

4.14 FR Test 12

4.14.1 Test Requirements

- Game engine error handles incorrect data types correctly

4.14.2 Testing Process

1. Initialize a game engine with a game state and an AI agent *test_engine.py*.
2. Initialize a game move with incorrect data types through the AI Agent.

4.14.3 Expected Results

- Game engine displays to the terminal that a data type provided by the agent is incorrect and the simulation does not continue running.

4.14.4 Actual Results

- **Negative Test Case:** Data type error was thrown by the engine and simulation was halted.

4.14.5 Results Analysis

The test case has passed.

4.15 FR Test 13

4.15.1 Test Requirements

- Game engine error handles illegal moves correctly

4.15.2 Testing Process

1. Initialize a game engine with a game state and an AI agent *test_engine.py*.
2. Initialize an illegal game move through the AI Agent.

4.15.3 Expected Results

- Game engine displays to the terminal that an illegal move was made by the agent and the simulation does not continue running.

4.15.4 Actual Results

- **Negative Test Case:** Dead step exception was thrown by the game engine and the simulation did not complete successfully.

4.15.5 Results Analysis

The test case has passed.

4.16 Data Visualizer Module

4.17 FR Test 14

4.17.1 Test Requirements

- The data visualizer is able to read the JSON log file

4.17.2 Testing Process

1. A simulation log JSON file is created after running the AI simulation.
2. The data visualizer then is directed to read to the directory/location of the log file.
3. The data visualizer is executed.

4.17.3 Expected Results

- The JSON log file is read successfully by the data visualizer and data is displayed on web application.

4.17.4 Actual Results

- **Positive Test Case:** The data is successfully read by the visualizer and the web application successfully displays the according data in the JSON file.

4.17.5 Results Analysis

The test case has passed.

4.18 FR Test 15

4.18.1 Test Requirements

- The data visualizer is able to read the JSON log file

4.18.2 Testing Process

1. Run *npm install* in the */src/visualization_v1/* folder.
2. A simulation log JSON file is created after running the AI simulation.
3. The data visualizer then is directed by putting the log JSON file into */src/visualization_v1/data/* folder and renaming to the file to *game.json*.
4. The data visualizer is executed with *npm start* and accessed with *http://localhost:3000/*.

4.18.3 Expected Results

- The JSON log file is read successfully by the data visualizer and data is displayed on the web application.

4.18.4 Actual Results

- **Positive Test Case:** The data is successfully read by the visualizer and the web application successfully displays the according data in the JSON file.

4.18.5 Results Analysis

The test case has passed.

4.19 FR Test 16

4.19.1 Test Requirements

- Data visualizer about to select specific data points

4.19.2 Testing Process

1. Run *npm install* in the */src/visualization_v1/* folder.
2. A simulation log JSON file is created after running the AI simulation.
3. The data visualizer then is directed by putting the log JSON file into */s-rc/visualization_v1/data/* folder and renaming to the file to *game.json*.
4. The data visualizer is executed with *npm start* and accessed with *http://localhost:3000/*.
5. On the data visualizer page, try all possible numbers of simulations displayed at once on the page.

4.19.3 Expected Results

- The JSON log file is read successfully by the data visualizer and data is displayed on the web application. Additionally, the data displayed changes accordingly based on the amount of data points selected by the user.

4.19.4 Actual Results

- **Positive Test Case:** The data is successfully read by the visualizer and the web application successfully displays the according data in the JSON file. The data populates on the web page according to the number of simulations selected.

4.19.5 Results Analysis

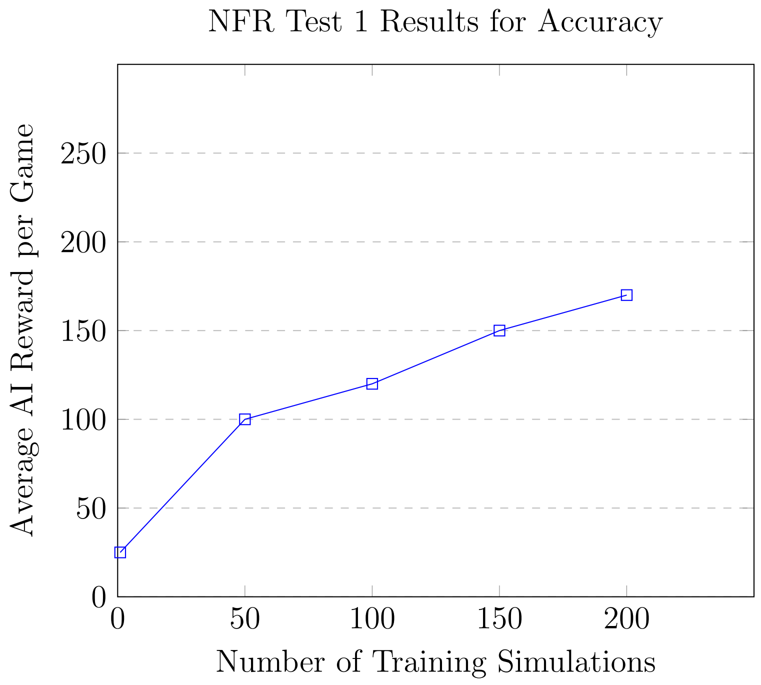
The test case has passed.

5 Nonfunctional Requirements Evaluation

5.1 Accuracy

To test accuracy of the system, win rates were compared between AI's using models from that had different levels of training, and checking how their win rates compared (against an AI that only takes random moves) and how long the model was trained for. The expected conclusion was that the longer the model was trained for the higher win rate it would have. The simulation was ~~ran~~ run for a total of 200 times, comparing win rates at intervals of 50 simulations. The higher the AI reward, the more points the AI scored during the game. After running these simulations, the AI's rewards and improved after each interval of training, so we can draw the conclusion that the system became more accurate with more training.

- NFR Test 1: **Pass**



5.2 Usability

To test the usability of the system, we invited the AI Research Professors and Game Designers to manually test our data visualizers and determine if they can easily navigate and comprehend the data from the charts. We specified all features of our data visualizers and identify how long it takes for the testers to become comfortable with using the tool. The result is that the AI Research Professor and game designers familiarize the manipulation of the data visualizer and remember the workflow of the data visualizer within

20 mins which is significantly less than our expected time for new users to learn how data visualizers.

- NFR Test 2: **Pass**

5.3 Modularity

To test the modularity of the system, AI Agents should be able to be swapped with ease and the system should function correctly. AI Agents policies (IE. strategies), are generated as files after a successful execution of the system, the system was ~~ran~~ run twice with different parameters and saved each policy separately. Then the policies were then given as input to the system and was able to function correctly.

- NFR Test 2: **Pass**

5.4 Portability

To test portability of the system, we tested on different Operating Systems (Mac OS and Windows) and checked if the libraries were able to be installed and if the system could be executed. After clearing all existing libraries and installing a fresh version of Python, the system was able to be installed and executed on both Mac OS and Windows.

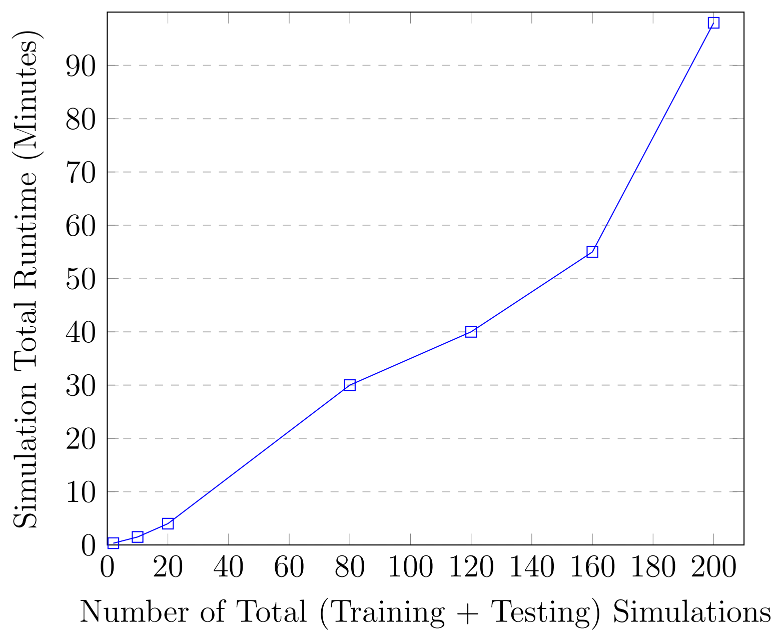
- NFR Test 4: **Pass**
- NFR Test 5: **Pass**
- NFR Test 6: **N/A**

5.5 Performance

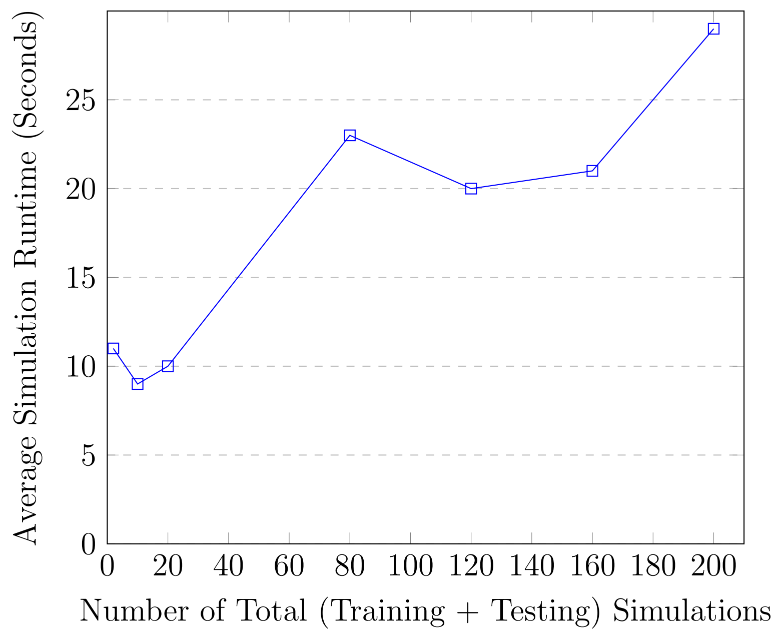
To test performance of the system, the system was executed with different number of simulations and timed the length of the runtime. The results were plotted and to ensure the system passes the test, the average runtime per simulation must be less than 5 minutes. The average was well below the targeted of 5 minutes.

- NFR Test 7: **Pass**

NFR Test 7 Results for Performance (Total)



NFR Test 7 Results for Performance (Average)



5.6 Extensibility

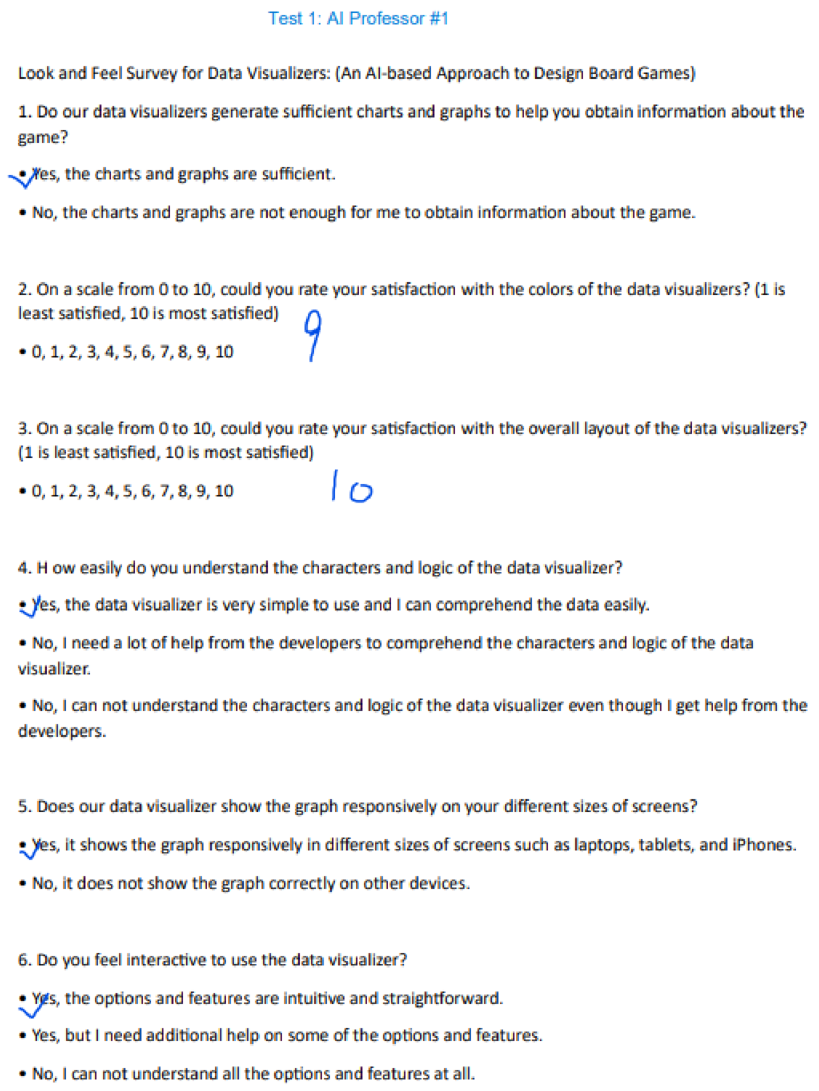
To test extensibility, development time was measured in how long it took to add in new game mechanic. Mechanics varied depending on complexity, but on average it took less than 2 days of development time to develop a new feature, although testing and integration into the entire caused the total time to be over 2 days, causing this test to fail. To further evaluate extensibility of the system, integration and testing time should only be measured since development of mechanics is not a good measurement of the system as a whole, only the game engine.

- NFR Test 7: **Fail**

5.7 Look and Feel

To test the look and feel of the data visualizer, we invited the game designers and stakeholders to test our data visualizers and conduct a look and feel survey. There are six questions that ask them how they feel about the data visualizer's colors, designs, interactions, and diversity. According to the survey's findings, our users are happy with the data visualizer's user interface and enjoy using it to comprehend data.

- NFR Test 9: **Pass**



Test 1: AI Professor #1

Look and Feel Survey for Data Visualizers: (An AI-based Approach to Design Board Games)

1. Do our data visualizers generate sufficient charts and graphs to help you obtain information about the game?

- [✓ Yes, the charts and graphs are sufficient.]
- No, the charts and graphs are not enough for me to obtain information about the game.

2. On a scale from 0 to 10, could you rate your satisfaction with the colors of the data visualizers? (1 is least satisfied, 10 is most satisfied)

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 9

3. On a scale from 0 to 10, could you rate your satisfaction with the overall layout of the data visualizers? (1 is least satisfied, 10 is most satisfied)

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 10

4. H ow easily do you understand the characters and logic of the data visualizer?

- [✓ Yes, the data visualizer is very simple to use and I can comprehend the data easily.]
- No, I need a lot of help from the developers to comprehend the characters and logic of the data visualizer.
- No, I can not understand the characters and logic of the data visualizer even though I get help from the developers.

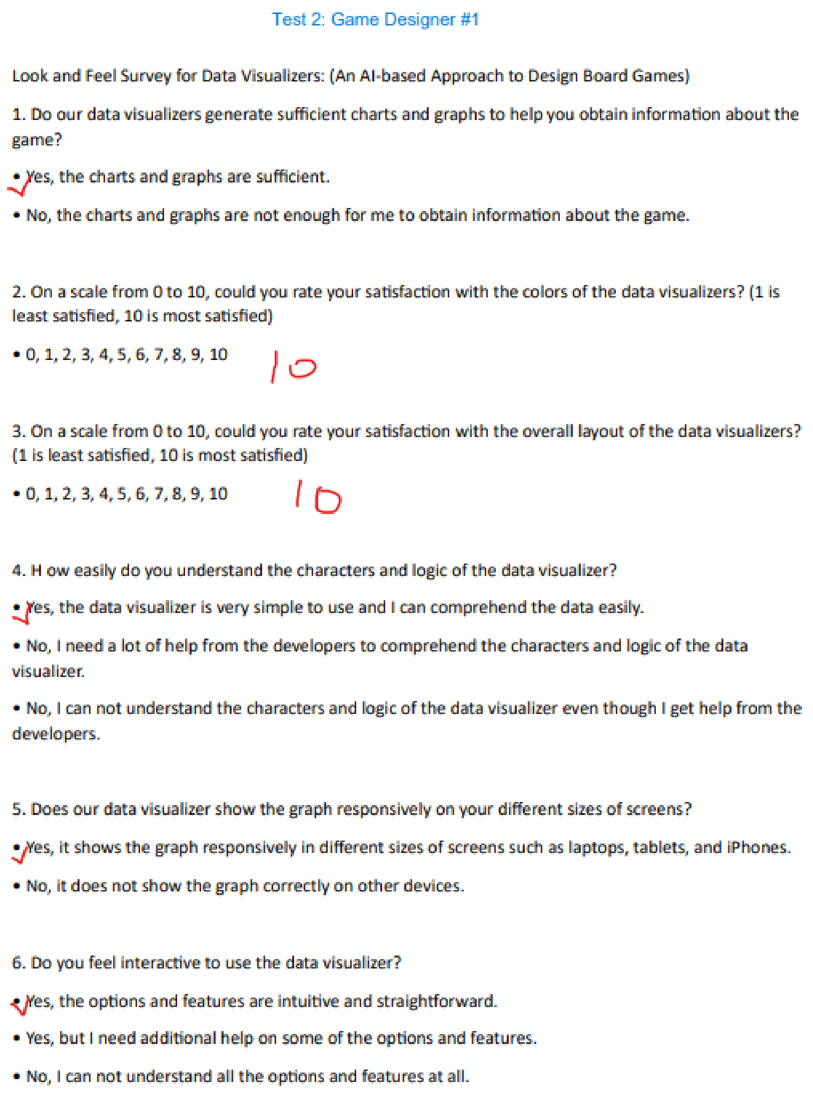
5. Does our data visualizer show the graph responsively on your different sizes of screens?

- [✓ Yes, it shows the graph responsively in different sizes of screens such as laptops, tablets, and iPhones.]
- No, it does not show the graph correctly on other devices.

6. Do you feel interactive to use the data visualizer?

- [✓ Yes, the options and features are intuitive and straightforward.]
- Yes, but I need additional help on some of the options and features.
- No, I can not understand all the options and features at all.

Figure 1: One of the survey from AI Professor



Test 2: Game Designer #1

Look and Feel Survey for Data Visualizers: (An AI-based Approach to Design Board Games)

1. Do our data visualizers generate sufficient charts and graphs to help you obtain information about the game?

- [✓ Yes, the charts and graphs are sufficient.]
- No, the charts and graphs are not enough for me to obtain information about the game.

2. On a scale from 0 to 10, could you rate your satisfaction with the colors of the data visualizers? (1 is least satisfied, 10 is most satisfied)

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 10

3. On a scale from 0 to 10, could you rate your satisfaction with the overall layout of the data visualizers? (1 is least satisfied, 10 is most satisfied)

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 10

4. H ow easily do you understand the characters and logic of the data visualizer?

- [✓ Yes, the data visualizer is very simple to use and I can comprehend the data easily.]
- No, I need a lot of help from the developers to comprehend the characters and logic of the data visualizer.
- No, I can not understand the characters and logic of the data visualizer even though I get help from the developers.

5. Does our data visualizer show the graph responsively on your different sizes of screens?

- [✓ Yes, it shows the graph responsively in different sizes of screens such as laptops, tablets, and iPhones.]
- No, it does not show the graph correctly on other devices.

6. Do you feel interactive to use the data visualizer?

- [✓ Yes, the options and features are intuitive and straightforward.]
- Yes, but I need additional help on some of the options and features.
- No, I can not understand all the options and features at all.

Figure 2: One of the survey from Game Designer

6 Unit Testing

6.1 AI Game Engine Unit Tests

These functions are under a test class, so self refers to a child of a TestCase class. They use PyTest libraries to run the unit tests and automate them to a degree.

- **Unit Test 1 - FR 7 Test (4.9)**: Check whether the game engine initialized correctly.

  Engine has multiple functions that initializes it and goes back and forth between the AI module and game engine to make sure the AI agents makes choices to initialize the game board. If there is any errors in the initialization, this variable remains False and appropriate exceptions are raised by the game engine.

```
        def check_engine_initilization(self):
            engine = Engine()
            self.assertTrue(engine.is_initialed)
```

- **Unit Test 2 - FR 7 Test (4.9)**: Check whether the AI agents are initialized correctly in the game engine.

```
        def check_engine_initilization(self):
            engine = Engine()
            NUM_PLAYERS = 5 #different games can have different players
            self.assertTrue(len(engine.agents) == NUM_PLAYERS)
```

- **Unit Test 3 - FR1 Test (4.2)**: Check whether the AI agents respond to game engine with correct turn type.

```
        def check_engine_initilization(self):
            engine = Engine()
            current_agent = engine.get_current_agent()
            self.assertTrue(current_agent.turn.turn_type == engine.turn.turn_type)
```

- **Unit Test 4 - FR2 Test (4.3)**: Check whether the new AI agent respond to game engine with correct turn type.

```
        def check_engine_initilization(self):
            engine = Engine()
            new_agent = Player("agent_name", "
    default_ai_policy")
            engine.agents[5] = new_agent
            self.assertTrue(engine.agents[5].turn.
    turn_type == engine.turn.turn_type)
```

- **Unit Test 5 - FR3 Test (4.4)**: Check whether the AI Agent's moves are observable in the environment. We do this by checking the action_list in the game engine has action objects inside which are logged to JSON with their English explanations.

```
        def check_engine_initilization(self):
            engine = Engine()
            current_agent = engine.get_current_agent()
            self.assertTrue(len(engine.get_action_list(
    current_agent)) > 0)
```

- **Unit Test 6 - FR5 Test (4.6)**: Check whether the AI Agent takes observation space as input. Unit test cannot 100% test this due to the AI library's input system. However, we can check whether the observation space object is initialized and can check a few key variables to see whether the game engine actually creates a observation space.

```
        def check_engine_initilization(self):
            engine = Engine()
            observation_space = engine.
    get_observation_state()
            self.assertTrue(observation_space != None and
     len(observation_space.map) > 0)
```

- **Unit Test 7 - FR8 Test (4.10)**: Check whether the AI Agent makes progress towards an end-game state. We can check this by looking at the AI agent's reward function and see whether it is greater than 0. Which indicates that the AI actually did moves to earn points to end the game eventually.

```
        def check_engine_initilization(self):
            engine = Engine()
            current_agent = engine.get_current_agent()
            self.assertTrue(current_agent.reward > 0)
```

- **Unit Test 8 - FR9 Test (4.11)**: Check whether the Game state changes accordingly to AI Agent moves. This is hard to check in unit test level since multiple major sub-systems are involved in changing the game state with the actions. But we can check the AI Agent's observation space is different after the action to infer that there was a change in the game state.

```
        def check_engine_initilization(self):
            engine = Engine()
            current_agent = engine.get_current_agent()
            initial_observation_space = current_agent.
    get_observation_space()
            engine.play_turn() #agent takes action here
            new_observation_space = current_agent.
    get_observation_space()
            self.assertTrue(initial_observation_space !=
    new_observation_space)
```

- **Unit Test 9 - FR10 Test (4.12)**: Check whether the Action list should be consistent throughout the simulation. We can check this by comparing the action list between different turns.

```
        def check_engine_initilization(self):
            engine = Engine()
            initial_action_list = engine.get_action_list
    ()
            engine.play_turn() #a turn played
            new_action_list = engine.get_action_list()
            self.assertTrue(initial_action_list ==
    new_action_list)
```

6.2 Data Visualization Unit Tests

The following tests cover the functions that were created to transform the JSON file received from the game engine module into formats that were needed for the graphs used in this module.

Note: JSON Object array refers to an object array with all data from input file, or any modified object array that is return from the the following functions: getAllData, getAllDataExEnd, getDataWithMergedActions. Excluding the first test, all other functions help with **FR15 Test**, and **FR16 Test**.

Test ID	**Function Name**	**Inputs**	**Output**	**Result**
DV1 - maps to **FR14 Test (4.17)**	getAllData	None	JSON object array of the file generated from the Game Engine module	Pass
DV2	getAllData- Ex-End	None	JSON object array with only the data needed for the graphs	Pass
DV3	getDataWith-MergedActions	None	JSON object array that merges the action and action_details fields into one	Pass
DV4	getSimulation-Data	JSON Object Array, Number of Simulations	Array of Simulations	Pass
DV5	getPlayerData	Object with 1 Simulation data, Player ID	Object with Simulation data for the specified player	Pass
DV6	getFrequency-Map	None	Object Array with each object having action name and frequency set to 0	Pass

DV7	getCountMap	None	Object Array with each object having action name and count set to 0	Pass
DV8	getFrequencyMapForPlayer	JSON Object Array with merged actions, Number of Simulations, Player ID	Frequency Map Object for the specified player for the specified simulations	Pass
DV9	getCountMapForPlayer	JSON Object Array with merged actions, Number of Simulations, Player ID	Count Map Object for the specified player for the specified simulations	Pass
DV10	getAllNonZeroActions	Frequency/Count Map of a Player	Frequency/Count Map that only has non-zero frequencies/counts	Pass
DV11	getScores	JSON Object Array with merged actions, Number of Simulations	Object with final scores data for the number simulations specified	Pass
DV12	getPlayers	JSON Object Array	Array with sequence of Player IDs	Pass

DV13	getSimulations	JSON Object Array	Array with sequence of simulation IDs	Pass
DV14	getNumberOfMoves	JSON Object Array with merged actions, Number of Simulations, Player ID	Number of moves for the player specified for the number of simulations specified	Pass
DV15	getMap	JSON Object Array with merged actions, Number of Simulations, Player ID	Frequency map for specified player's actions for the number of simulations specified, 2D array of action names where each index is the move number	Pass

7 Changes Due to Testing

Based on the testing, many small bugs and edge cases were discovered in the game engine modules and were fixed. Also, the AI training parameters were improved to meet requirements of runtime and efficiency. Apart from the automated and unit testing, we have also had feedback from our supervisors on data visualizer and training process. Before talking to our supervisors to strategies on AI training, we were using the Apple M1 Pro MacBooks to train the AI agent and we weren't meeting the required training levels. We have tried to improve the training parameters using the manual tests results but weren't able to reach our target goals. Then after discussing it with our supervisors, we got access to Compute Canada supercomputer through their sponsorship and increase the training speed. Then our NFR performance tests started to pass and our system started to show more logical outputs since the AI Agents were smart enough to take logical actions. For data visualizer, we have started with python libraries but after getting supervisor and other user feedback, we have quickly realized that the python did not provide enough graph types to provide meaningful tools to analyze the data generated through our system. So, we have decided to build a web app using React and Apache graphing libraries. At the end, users and supervisor feedback indicated that the visualizer version 1 (React version) was more aligned with our NFRs than visualizer version 0 (python version). Also, based on the feedback, we have made our action space human readable by including English descriptions and tags to them. We have categorized actions which helped with the folder structure in the code and also assigning categories to colour code each action category on the visualizer which improved the usability of our system significantly.

- **NFR Test 1**: Changed total simulations to 200, due to length of runtime for 1000 simulations after game engine grew more complex (each simulation took longer to run)
- **NFR Test 6**: Test was not completed as the system was able to be ran on developer's PC's and the computing cluster was not needed.
- **NFR Test 7**: Test was changed to also measure performance against more than 1 simulation and ensuring that the average was less than 1 simulation per 5 minutes

- **FR Test 17**: Related feature was removed after the scope was redefined for the project, thus this test was no longer necessary for the project.

8 Automated Testing

8.1 AI Agents Game Engine Modules

Since the AI libraries and game engine is written in Python, we have used PyTest libraries to write unit tests. Unit tests described above can run automatically by adding the -test argument (custom made argument in our game engine to handle running all the unit tests) when running the simulation as follow:

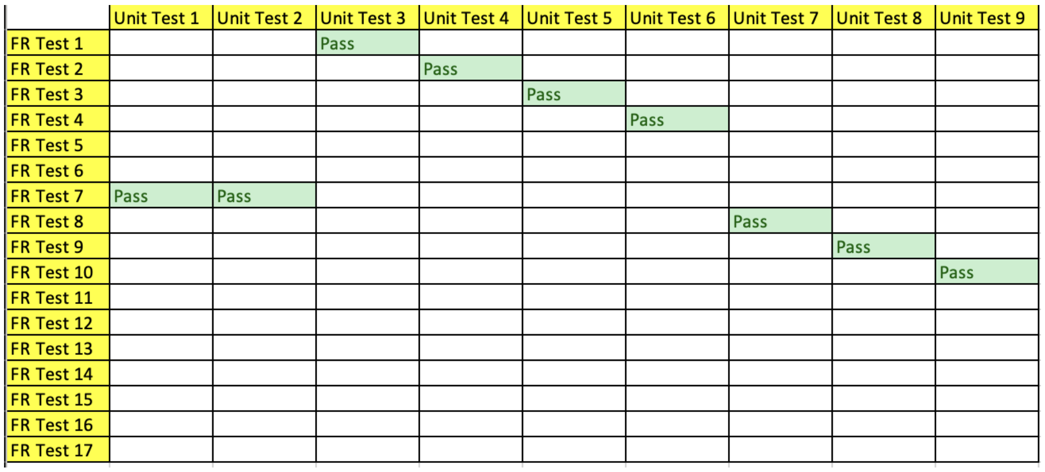
python main.py -test

8.2 Data Visualizaton Module

As this module is developed with JavaScript, we used a popular JavaScript testing framework called Jest for the unit testing. We created mock data as inputs for all the functions and then used their matcher functions like toEqual to validate the results. We also used the code coverage functionality in Jest to measure the code coverage of the tests on the functions. Code Coverage Metrics can be viewed in the Code Coverage Metrics section.

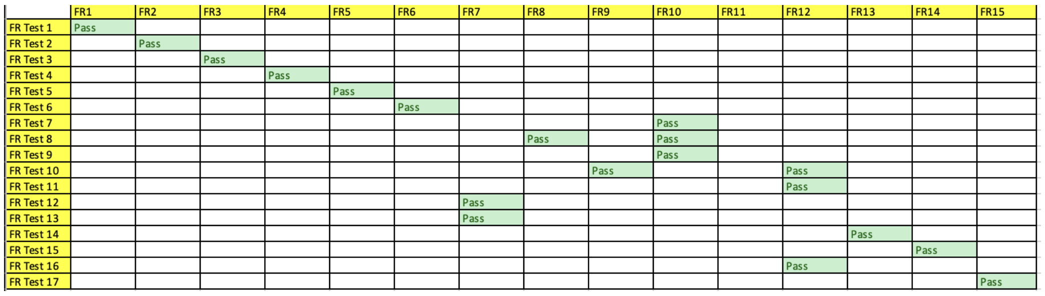
9 Trace to Requirements

Full requirement details can be found on the SRS, under sections 6 and 7. Traceability Matrix below.



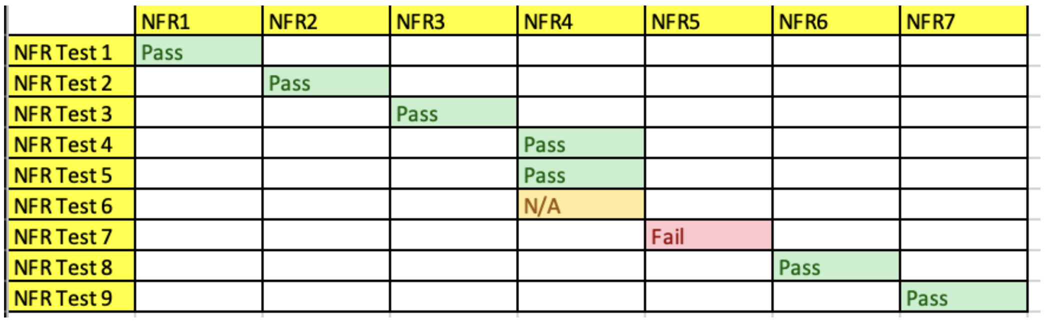
	Unit Test 1	Unit Test 2	Unit Test 3	Unit Test 4	Unit Test 5	Unit Test 6	Unit Test 7	Unit Test 8	Unit Test 9
FR Test 1			Pass						
FR Test 2				Pass					
FR Test 3					Pass				
FR Test 4						Pass			
FR Test 5									
FR Test 6									
FR Test 7	Pass	Pass							
FR Test 8							Pass		
FR Test 9								Pass	
FR Test 10									Pass
FR Test 11									
FR Test 12									
FR Test 13									
FR Test 14									
FR Test 15									
FR Test 16									
FR Test 17									

Figure 3: Unit Tests to Functional Requirement Tests Matrix



	FR1	FR2	FR3	FR4	FR5	FR6	FR7	FR8	FR9	FR10	FR11	FR12	FR13	FR14	FR15
FR Test 1	Pass														
FR Test 2		Pass													
FR Test 3			Pass												
FR Test 4				Pass											
FR Test 5					Pass										
FR Test 6						Pass									
FR Test 7										Pass					
FR Test 8								Pass		Pass					
FR Test 9										Pass					
FR Test 10									Pass			Pass			
FR Test 11												Pass			
FR Test 12							Pass								
FR Test 13							Pass								
FR Test 14													Pass		
FR Test 15														Pass	
FR Test 16												Pass			
FR Test 17															Pass

Figure 4: Functional Requirements to Tests Matrix

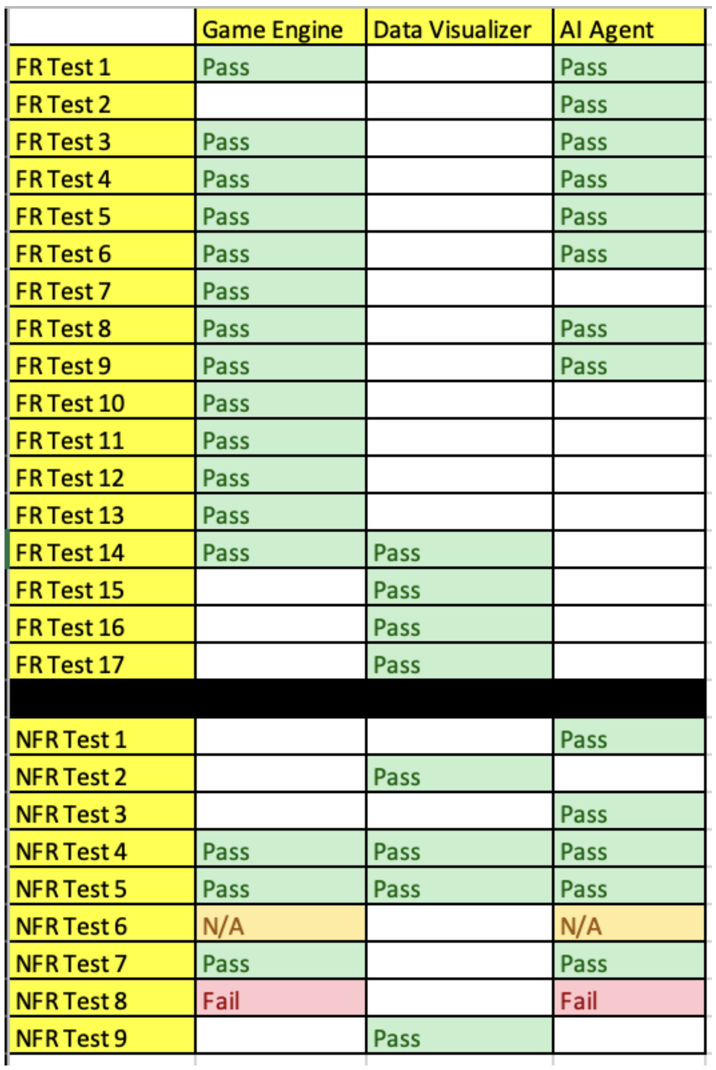


	NFR1	NFR2	NFR3	NFR4	NFR5	NFR6	NFR7
NFR Test 1	Pass						
NFR Test 2		Pass					
NFR Test 3			Pass				
NFR Test 4				Pass			
NFR Test 5				Pass			
NFR Test 6				N/A			
NFR Test 7					Fail		
NFR Test 8						Pass	
NFR Test 9							Pass

Figure 5: Non-Functional Requirements to Tests Matrix

10 Trace to Modules

Full module details can be found on the Design Document.



	Game Engine	Data Visualizer	AI Agent
FR Test 1	Pass		Pass
FR Test 2			Pass
FR Test 3	Pass		Pass
FR Test 4	Pass		Pass
FR Test 5	Pass		Pass
FR Test 6	Pass		Pass
FR Test 7	Pass		
FR Test 8	Pass		Pass
FR Test 9	Pass		Pass
FR Test 10	Pass		
FR Test 11	Pass		
FR Test 12	Pass		
FR Test 13	Pass		
FR Test 14	Pass	Pass	
FR Test 15		Pass	
FR Test 16		Pass	
FR Test 17		Pass	
NFR Test 1			Pass
NFR Test 2		Pass	
NFR Test 3			Pass
NFR Test 4	Pass	Pass	Pass
NFR Test 5	Pass	Pass	Pass
NFR Test 6	N/A		N/A
NFR Test 7	Pass		Pass
NFR Test 8	Fail		Fail
NFR Test 9		Pass	

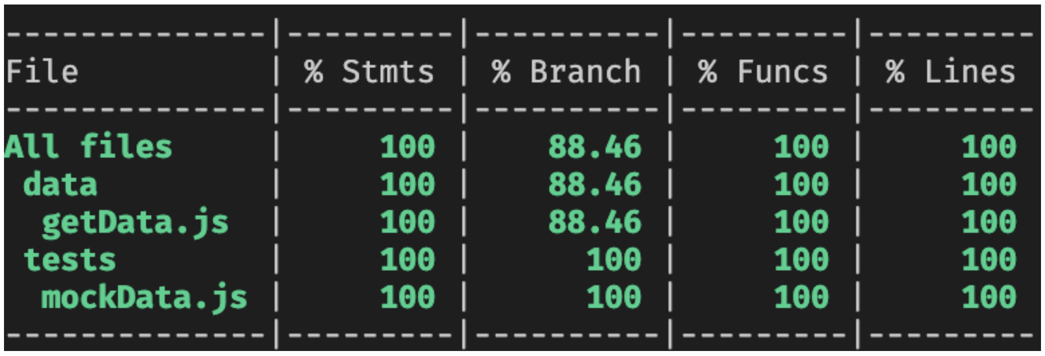
Figure 6: Modules to Tests Matrix

11 Code Coverage Metrics

11.1 AI Agents Game Engine Modules

The unit test coverage is not taken into account since unit tests can only check a very small subset of the game engine due to complex code execution sequence that AI libraries require. So, any code coverage figure would be really low and not really reflect our confidence in the game. Most of our testing is done through looking at the visualization and log files to confirm that the AI outputs are within the game rules and makes sense.

11.2 Data Visualization Module



```
--------------|---------|----------|---------|---------
File          | % Stmts | % Branch | % Funcs | % Lines
--------------|---------|----------|---------|---------
All files     |     100 |    88.46 |     100 |     100
 data         |     100 |    88.46 |     100 |     100
  getData.js  |     100 |    88.46 |     100 |     100
 tests        |     100 |      100 |     100 |     100
  mockData.js |     100 |      100 |     100 |     100
--------------|---------|----------|---------|---------
```

Figure 7: Data Visualization Code Coverage Metrics

With our unit testing, we were able to get 100% statement and function coverage. We achieved 88.46% branch coverage as we were not able to reach a branch in a sorting function which was used to sort some of the objects and arrays. However, as a group we had decided that we were satisfied with achieving $> 80\%$ coverage on statements, branches and functions.

12 Reflection

As a team, we noticed the changes in our verification and validation plan (VnV) and the report had changed in major and minor ways as there was a major difference between writing the plan and report, which would be the finishing of the implementation of our project. Before, planning the verification and validation of our system was very abstract as we were not entirely sure how we would end up implementing the whole system. Currently, we finished the implementation and are able to test directly and have solid test cases to demonstrate the feasible system. The concreteness of the project demonstrates how we have to critically think about what portion of the system we either missed completely in our code coverage or spent too many test cases when it is an obvious piece of code that will self-facilitate itself after implementation on our VnV plan. Furthermore, the focus of the project is the structure of an AI simulation of ANY board game, so many of our test cases are generalized for all board games applied to the system and any unit tests directly related to the game we specifically worked with (An Age Contrived) are omitted from this document. Any specific examples of An Age Contrived for testing are described in brackets in the documents. Another change between the VnV Plan and the VnV Report is the metrics used for testing the Non-Functional Requirements, initially we did not know the performance of our system, so we had a low threshold for how well we wanted it to perform. After building the system, our system performed way higher than expected due to not knowing how the system will run with the given resources.

For next time, when writing our VnV plan we will consider of the possible implementation of the system, rather than abstracting some of the tests. This would result in fewer changes for us during the VnV report portion and lead to greater consistency in the report.