Dysruption - Action Analysis

Rishitosh Kumar Singh

Arizona State University

May 7, 2025

Contents

Chapter 1

Introduction

This report presents the work conducted over the semester on a project focused on analyzing decision-making strategies within critical infrastructure networks through interactive simulation games. The primary objective was to design and develop autonomous bots capable of playing two online strategy games—*Break it Bad* and *Save Our System (Scenario 1)*—and to evaluate their performance by recording and analyzing their actions and corresponding scores.

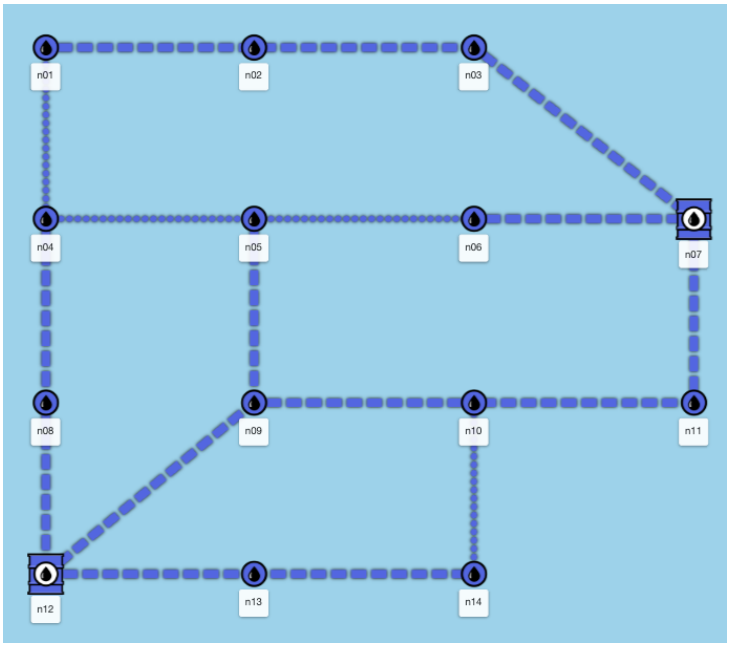

Figure 1.1: Game network with two supply nodes, and 12 demand nodes with "has flow", "no flow" edges.

Both games simulate a fuel transportation network composed of supply nodes, demand nodes, and connecting pipelines. In *Break it Bad*, the objective is to disrupt the network as effectively as possible using a limited number of actions. Success is measured by the cost inflicted on the system through strategically damaging pipelines and cutting off supply routes. In contrast, *Save Our System* challenges players to maintain network stability in the face of potential disasters. The objective in this game is to mitigate risk, prepare for unpredictable events, and ensure uninterrupted fuel flow at minimal cost

The project aimed to:

- Develop autonomous bots to play both games using rule-based strategies.
- Record the sequence of actions taken and the resulting scores.
- Analyze how different types of actions influence system performance under various conditions.

To accomplish these objectives, object-oriented programming principles were applied to structure the bots, ensuring modularity, extensibility, and ease of testing. Additionally, a logging system was developed to capture data from each gameplay session. This report outlines the design process, implementation logic, analysis methods, and the insights gained through experiments with different bot strategies.

Chapter 2

Game and Bot Framework

There are a total of 10 games available for users to play, but this semester the focus was on two of them, as discussed in chapter 1. To ensure that the project and modular bot framework are extensible, the codebase was designed to accommodate various strategies and gameplay logic. The project is structured around a base class, `DysruptionGame`, which encapsulates shared functionalities such as browser login, map interaction, state inspection, and common game utilities. This base class is further extended by two specialized subclasses: `BreakItBad` and `SaveOurSystem`, each implementing game-specific strategies. This design facilitates easy swapping of strategies and analysis of performance data across multiple simulation runs, while also enabling the addition of new games in the future for analysis. More details are provided in appendix B.

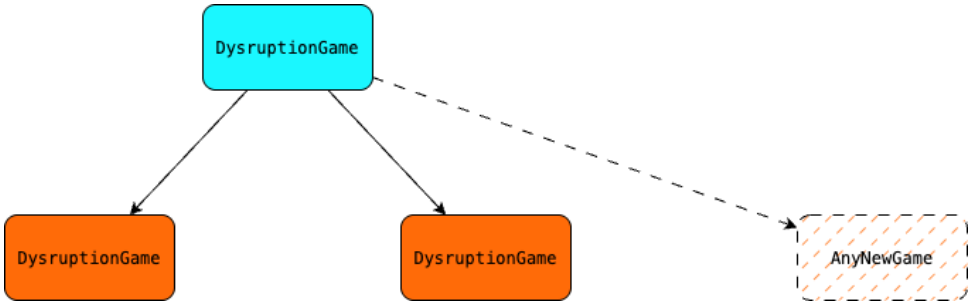

Figure 2.1: Class diagram illustrating the inheritance structure of the bot framework. Any new game can be added by extending the `DysruptionGame` class.

2.1 Bot Action Logics

For each game, the bot was designed with a set of action logics to choose from. The bot can be configured with various decision logics, referred to as "intelligence levels," which differ for each game. These intelligence levels are outlined as follows:

- **Break It Bad**:
  - **not_dumb**: Acts similarly to a typical user, although the bot does not have the ability to determine which action will yield the highest score.
  - **dumb**: Takes actions without considering their effectiveness, including moves that a regular user would typically avoid, resulting in no improvement in score.

- **Save Our System**:
  - **only_maintain**: Always performs the maintain action on the selected edge if the bot randomly decides to act.
  - **only_replace**: Always performs the replace action on the selected edge if the bot randomly decides to act.
  - **only_repair**: Always performs the repair action on the selected edge if the bot randomly decides to act.
  - **weighted_random**: Chooses an action based on the condition of the edge (FCI) and the available budget:
    * If the FCI ≥ 99.6 and the cost of maintenance is within the budget, performs the maintain action. *(Rationale: At this level, the pipeline is nearly perfect, so maintaining it helps prolong its optimal condition, preventing future degradation.)*
    * If $92 \leq$ FCI < 99.6 and the cost of repair is within the budget, performs the repair action. *(Rationale: This range indicates that the pipeline is functioning but showing signs of wear. Repairs can restore it to a more robust state without incurring the high costs of replacement.)*
    * If FCI < 92 and the cost of replacement is within the budget, performs the replace action. *(Rationale: A low FCI suggests significant wear or damage. Replacing the pipeline ensures long-term reliability, especially when incremental repairs are not sufficient.)*

The bot's decision-making process, as illustrated in fig. 2.2, is structured as a series of logical steps that guide its actions during gameplay. The flowchart provides a clear visual representation of how the bot evaluates the game state, selects an action, and executes it.

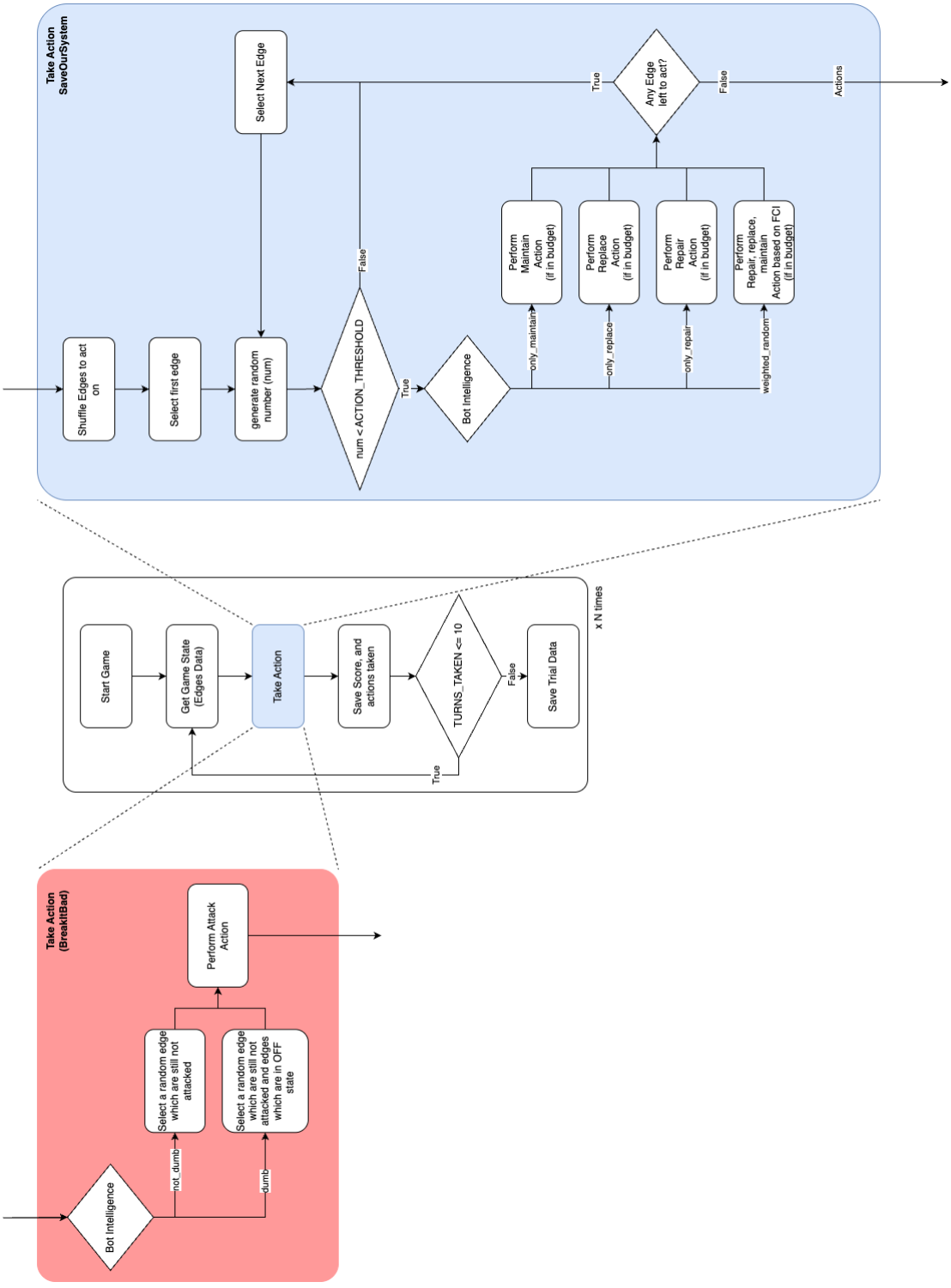

Figure 2.2: Flowchart illustrating the decision-making logic of the bots for both games. The central white box is implemented in the `bot.py` and `bot-savepursystem.py` files, while the other two boxes are implemented within their respective game classes in `src/game.py`.

Chapter 3

Result and Analysis

3.1 Break It Bad

To evaluate the performance of the bot in the *Break It Bad* game, the bot was run in `not_dumb` intelligence mode for approximately 500 game simulations. In each simulation, the bot performed up to 5 moves, and the resulting score was recorded after each move. The objective was to analyze the progression of scores with each action and to identify the move sequences that resulted in the highest disruption scores.

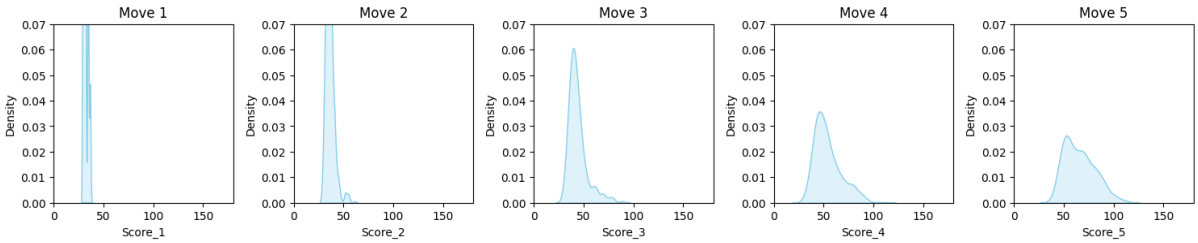

Figure 3.1: KDE plot showing the distribution of scores after each move over 500 simulations (maximum of 5 moves shown). Higher density towards the right indicates increasing score trends.

As shown in fig. 3.1, the KDE plots illustrate how the score distribution changes with each move. Initially, the scores are tightly concentrated, but as the game progresses, the distributions broaden and shift toward higher values, indicating increasing disruption. By Move 4 and Move 5, the distributions become significantly wider and start to exhibit bi-modal characteristics.

To gain deeper insights into the bot's decision-making process, the sequence of moves leading to the highest score after 5 moves was extracted. The maximum score achieved was **113**. The corresponding move sequence is presented in table 3.1, where each entry represents an edge that was attacked during the game. The edges are denoted as pairs of nodes, indicating the targeted connections.

3.2 Save Our System

To assess different bot strategies in the *Save Our System* game, all four intelligence levels from section 2.1 were tested, with approximately $20 - 30$ trials conducted for each

Move	Edge Attacked
1	`7 <-> 3`
2	`12 <-> 8`
3	`7 <-> 6`
4	`11 <-> 7`
5	`5 <-> 9`

Table 3.1: Sequence of moves leading to the highest score in *Break It Bad*. Each entry `x <-> y` represents an attack on the pipeline (edge) connecting node `x` to node `y`.

strategy. The bot's actions and resulting scores were recorded after each game step. The best scores for each strategy are summarized in table 3.2. The results from all scenarios were then aggregated and analyzed to determine the best-performing strategy based on the highest final score achieved.

Strategy	Max Score Achieved
only_maintain	19
only_repair	15
only_replace	20
weighted_random	**28**

Table 3.2: Summary of the maximum scores achieved by different strategies in the *Save Our System* game.

Move	Score	Actions Taken
1	3	MAINTAIN: 10↔11, 7↔6, 10↔14, 14↔13, 2↔1
2	6	MAINTAIN: 4↔1, 11↔7, 7↔6
3	9	MAINTAIN: 14↔13, 12↔8, 9↔12, 2↔1, 11↔7
4	12	REPAIR: 9↔10, REPLACE: 2↔3, MAINTAIN: 2↔1, REPAIR: 5↔9, MAINTAIN: 14↔13
5	15	REPAIR: 6↔5, MAINTAIN: 11↔7, REPLACE: 4↔5
6	18	REPAIR: 5↔9, REPAIR: 10↔14, MAINTAIN: 11↔7
7	21	REPAIR: 12↔8, REPAIR: 9↔12, REPAIR: 13↔12
8	24	REPAIR: 7↔3, REPAIR: 9↔12, REPAIR: 2↔3, REPAIR: 10↔14, REPAIR: 10↔11
9	27	REPLACE: 14↔13, REPAIR: 7↔6, REPAIR: 11↔7, REPAIR: 13↔12
10	28	REPLACE: 4↔5, REPAIR: 4↔1, REPAIR: 10↔11, REPAIR: 11↔7

Table 3.3: Score progression of the best trial for the *Save Our System* game. The table displays the score after each move and the corresponding actions taken. The actions are categorized as MAINTAIN, REPAIR, or REPLACE. Each entry `x <-> y` represents an action performed on the pipeline (edge) connecting node `x` to node `y`.

Table 3.3 highlights how the best move sequence in the `weighted_random` strategy, in which the bot dynamically chooses actions based on the Failure Condition Index (FCI)

of each pipeline segment and the available budget. This strategy outperformed other fixed-action methods, achieving a maximum score of 28. The success of this approach lies in its ability to balance maintenance, repair, and replacement actions according to the pipeline's condition.

The effectiveness of the weighted random strategy can be attributed to its adaptive nature. By continuously assessing the condition of each pipeline and weighing the costs of potential actions, the bot optimizes resource allocation. This balanced approach is crucial in managing critical infrastructure, where maintaining optimal performance with limited resources is a constant challenge.

In the early game (moves 1-3), the bot primarily performed maintenance actions to preserve high-FCI pipelines, preventing premature deterioration and ensuring cost-effective long-term stability. As the game progressed (moves 4-7), the bot adapted by prioritizing repairs when FCI dropped below 99.6, effectively restoring moderately degraded pipelines without incurring replacement costs. In the late game (moves 8-10), the bot focused on replacing severely degraded pipelines (FCI < 92) when repairs became inefficient, ensuring that critical failures were addressed in a budget-optimized manner.

Chapter 4

Conclusion and Future Work

In conclusion, the project provided a valuable learning experience, resulting in the development of a modular bot framework that can be extended to incorporate additional games in the future if needed. For this semester, the development of the bots for the *Break It Bad* and *Save Our System* games was completed successfully, with both bots functioning as expected. The bots demonstrated the ability to play the games using different strategies and intelligence levels as specified in the project requirements.

For future work, the *Save Our System* game bot can be further enhanced by incorporating reinforcement learning (RL) to improve decision-making. Instead of predefining strategies such as "only_maintain" or "only_replace," the bot could autonomously learn optimal moves through continuous gameplay. Using RL, the bot would explore various actions, and after each iteration, it would learn which moves yield the best rewards. After several hundred iterations, the bot would develop a robust policy to optimize gameplay, eliminating the need for manually defining strategies and allowing the bot to adapt to the game's random environment.

Since the project follows a modular architecture, developing an RL-based bot for the *Save Our System* game would be relatively straightforward. The bot framework's extensible design allows for the seamless integration of new games and strategies, enabling further exploration of advanced decision-making algorithms in future projects.

Chapter 5

Acknowledgements

The author would like to express gratitude to Dr. Thomas Seager for providing the opportunity to contribute to his research project. His guidance and insights were instrumental in shaping the direction of the work and ensuring its successful completion. Appreciation is also extended to Mazin Abdel Magid for providing detailed walkthroughs of the project and its requirements, which greatly facilitated the development process.

Appendix A

Repository Structure

This chapter provides an overview of the repository structure along with a brief description of each file and folder. It also includes guidelines on how to use the repository and where to make changes if needed.

```
/
├── config
│   ├── breakitbad.py
│   └── saveoursystem1.py
├── src
│   ├── game.py
│   └── utils
│       ├── const.py
│       ├── Edge.py
│       ├── metrics.py
│       └── Node.py
├── bot.py
├── bot-saveoursystem.py
└── requirements.txt
```

The details of the files and folders are as follows:

- **config/**: Contains configuration files for each game that the bot reads. These files include settings necessary for the bot's operation. Each game has its own configuration file, allowing users to adjust parameters such as the number of trials, the game URL, and the bot's intelligence level.
  - `break_it_bad.py`: Configuration file for the `BreakItBad` game.
  - `save_our_system.py`: Configuration file for the `SaveOurSystem` game.
- **src/**: Contains the source code for the bot framework and game-specific implementations.
  - `game.py`: The main game file where the base game class and the derived classes (`BreakItBad` and `SaveOurSystem`) are defined. New games can be added by defining them in this file.
  - `utils/`: Contains utility files such as data classes for storing node and edge data, as well as constants used within the games.

- **bot.py**: The bot script for the `BreakItBad` game. To run the bot, execute the command `python3 bot.py` in the terminal.
- **bot-saveoursystem.py**: The bot script for the `SaveOurSystem` game. To run the bot, execute the command `python3 bot-saveoursystem.py` in the terminal.
- `requirements.txt`: A file listing the Python dependencies required to run the project. This file can be used to set up a Python environment for the project.

Appendix B

Detailed Bot Framework Design

Class Design

The codebase is structured around a base class, `DysruptionGame`, which encapsulates common functionalities such as browser login, map interaction, state inspection, and utility functions used across different games. This base class is extended by two specialized subclasses:

- `BreakItBad`: Implements logic to attack pipelines with the goal of maximizing disruption.
- `SaveOurSystem`: Focuses on maintenance and repair actions while operating under a constrained budget.

Each subclass overrides and extends the functionality of the base class to implement game-specific strategies, including decision-making processes for action selection and score retrieval.

Class: DysruptionGame

The `DysruptionGame` class serves as the foundational blueprint for automating interactions within the simulation games. It abstracts repetitive tasks such as navigating the game user interface, extracting the current map state, and executing actions. Below is a summary of its core components and functions:

- **Constructor (__init__)**: Initializes the game by setting the URL, intelligence strategy, and configuration flags. It also launches the game and prepares storage for nodes and edges.
- **reset_game(first_run=False)**: Opens the game in a browser and waits for key UI elements to load. If it is the first run, it waits for the login prompt; otherwise, it waits for the introductory banner.
- **initialize_edges_nodes()**: Prepares empty lists to store node and edge objects before parsing the game state.
- **login()**: Enters the username in the designated field and submits the login form to access the game environment.

- **start_game()**: Waits for the introductory banner and map buttons to load, then clicks through to initiate the game.
- **activate_map_read_mode()**: Activates read mode by selecting the map option, allowing the bot to read and collect edge and node information.
- **find_edges()**: Identifies all visible pipeline edges on the map and retrieves their visual properties (such as flow direction and status) to initialize them as `Edge` objects.
- **find_edges_state()**: Inspects each edge individually to extract additional metadata from the inspector panel, including attributes like temperature, flow, and the Failure Condition Index (FCI).
- **find_nodes()**: Detects all nodes on the map, classifying them as either supply or demand based on their visual appearance.
- **find_nodes_state()**: Gathers detailed information for each node, including attributes such as name, penalty, demand, or shortfall, by examining the inspector panel.
- **random_action()**: Returns `True` or `False` based on a predefined probability threshold, allowing the bot to make probabilistic decisions.
- **action()**: Ensures the game's user interface is ready before performing the next action, adding a controlled delay between bot moves.
- **quit()**: Closes the browser session in a clean and orderly manner.

By consolidating these core functionalities, the class serves as a reliable and reusable foundation for developing bots for multiple games, enabling efficient extension and adaptation to new game environments.

Class: BreakItBad

The `BreakItBad` class extends the base `DysruptionGame` class to implement logic specific to the *Break it Bad* game scenario. In this game, the objective is to maximize disruption within the fuel distribution network using a limited number of moves.

Key components of the class include:

- **Constructor (__init__)**: Initializes the inherited attributes and configurations specifically for the `BreakItBad` game.
- **initialize_edges_nodes()**: Overrides the base method to additionally set up separate lists for attacked edges, not attacked edges, and edges that are in an off state.
- **get_score()**: Retrieves the current score from the scoreboard element on the game page using Selenium.
- **attack()**: Selects an edge to attack based on the chosen intelligence strategy—either randomly or through a targeted approach—and performs a simulated click on the selected edge.

- **action()**: Initiates an attack using the `attack()` method and then calls the base class's `action()` method to maintain the game loop.
- **is_game_over()**: Determines whether the game has concluded by checking if all attackable edges have already been used.
- **filter_edges()**: Categorizes edges into distinct groups (not attacked, stopped, attacked) based on their current status to facilitate strategic decision-making.
- **deactivate_map_read_mode()**: Disables map read mode and activates attack mode by interacting with the corresponding user interface button.
- **get_state()**: Aggregates all logic to update the current map state, including reading nodes, edges, and their properties. Implements retries to ensure robustness in state retrieval.

This class encapsulates all behavior specific to the *BreakItBad* game and provides a structured approach to implementing bot actions aimed at maximizing disruption within the game environment.

Class: SaveOurSystem

The `SaveOurSystem` class is a subclass of `DysruptionGame` that implements logic for a defensive gameplay scenario, where the objective is to maintain and restore a critical fuel network while operating under a constrained budget. The class supports multiple action types—repair, replace, and maintain—each associated with distinct costs and outcomes.

Key components of the class include:

- **Constructor (__init__)**: Inherits all base parameters from `DysruptionGame` and initializes edge categories specific to the *Save Our System* game.
- **initialize_edges_nodes()**: Extends the base method to also set up lists for operational and destroyed edges, categorizing them based on their current state.
- **get_score()**: Retrieves the current performance score displayed in the game UI.
- **get_budget()**: Reads the current available budget from the game interface, which is crucial for determining the feasibility of actions.
- **perform_action(edge, action_button_selector)**: Clicks the appropriate button (repair, replace, or maintain) and applies the chosen action to the selected edge.
- **action()**: Implements strategy-specific decision logic. Depending on the selected intelligence mode, the bot chooses the most suitable action while adhering to budget constraints. In the `weighted_random` mode, the bot evaluates the edge's health (FCI) before making a decision.
- **is_game_over()**: Serves as a placeholder for the game-ending condition (inherited from the base class), although it is not directly linked to the specific game logic.
- **filter_edges()**: Categorizes edges into operational and destroyed based on their failure condition index (FCI), aiding in the prioritization of maintenance and repair.

- **get_state()**: Gathers and consolidates the current game state, including node and edge detection, edge status extraction, and classification.

This class facilitates the simulation of various maintenance strategies and evaluates their effectiveness in maintaining network stability despite failures and budget limitations.

bot.py and bot-saveoursystem.py

The `bot.py` and `bot-saveoursystem.py` files serve as the entry points for running the bots in their respective games. These files instantiate the game class objects and execute the main game loop. The primary loop structure is illustrated in the center white box of the fig. 2.2 flowchart.

Appendix C

How to Run the Project

To run the project, follow these steps:

- Clone the repository to your local machine.
- Navigate to the project directory.
- Download `chromedriver` and place it in the project directory. Update the `config/breakitbad.py` and `config/saveoursystem1.py` files with the path to the `chromedriver`.
- Install the required dependencies using:

```
pip install -r requirements.txt
```

- Modify the configuration files in the `config` folder as needed. The configuration files are:
  - `break_it_bad.py`: Configuration file for the `BreakItBad` game.
  - `save_our_system.py`: Configuration file for the `SaveOurSystem` game.
- Run the bot for the desired game:

```
python3 bot.py
```

  or

```
python3 bot-saveoursystem.py
```