Dysruption - Action Analysis

Rishitosh Kumar Singh

May 1, 2025

1 Introduction

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

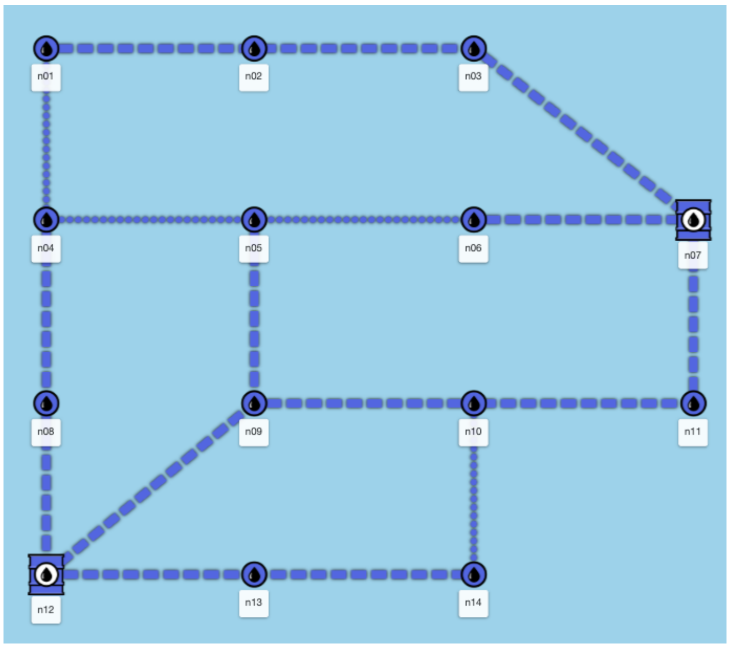

Figure 1: Game network with two supply nodes, and 12 demand nodes with "has flow", "no flow" edges.

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

The project aimed to:

- Build 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 achieve this, object-oriented programming principles were used to structure the bots, making them modular, extensible, and easy to test. A logging system was also developed to capture data from each gameplay session. This report details the design process, implementation logic, analysis methods, and the insights gained through experiments with different bot strategies.

2 Game and Bot Framework

There are a total of 10 games that a user can play, but this semester I only worked on two of them as discussed in section 1. So, to make the project and modular bot framework extensible, I designed the codebase to be adaptable to various strategies and gameplay logic. The project is structured around a base class `DysruptionGame`, which encapsulates shared behavior such as browser login, map interaction, state inspection, and common game utilities. This base class is extended by two specialized subclasses: `BreakItBad` and `SaveOurSystem`, each implementing game-specific strategies. This design allows for easy swapping of strategies and analysis of performance data across multiple simulation runs and also makes it easy to add new games in the future for their analysis. The more details are given in appendix B.

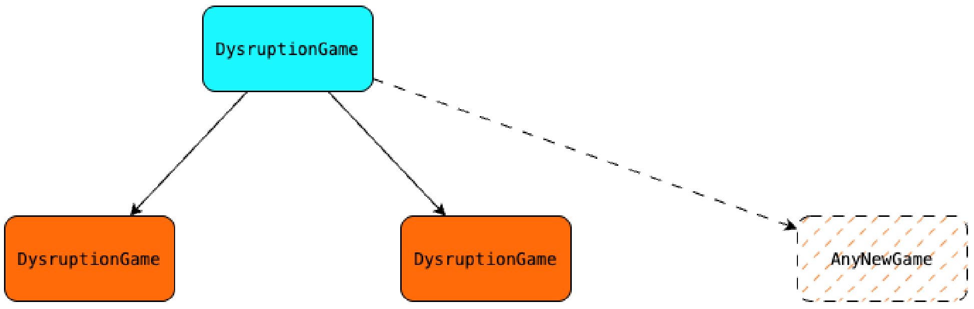

Figure 2: Class diagram illustrating the inheritance structure of the bot framework. Any new game can be added by extending the `DysruptionGame` class.

3 Bot Action Logics

For each game, I have given the bot a set of action logics to choose from. The bot can be configured with various decision logics, referred to as "intelligence levels" different for each games. These include:

- **Break It Bad**:
  - **not_dumb**: Will take action as a normal user, but the bot still doesn't know which action will give the highest score.

  - **dumb**: The bot will act as dumb, meaning it can take action which a normal user will never take which will give it no boost in score.

- **Save Our System**:
  - **only_maintain**: Only perform the maintain action on the selected edge if the bot randomly decides to take action.
  - **only_replace**: Only perform the replace action on the selected edge if the bot randomly decides to take action.
  - **only_repair**: Only perform the repair action on the selected edge if the bot randomly decides to take action.
  - **weighted_random**: If the bot randomly decides to take action, then if edge's:
    * FCI ≥ 99.6 and the cost of maintaining it is within the budget, perform the maintain action
    * $92 \leq \text{FCI} < 99.6$ and the cost of repairing it is within the budget, perform the repair action
    * FCI < 92 and the cost of replacing the edge is within the budget, perform the replace action

4 Result and Analysis

4.1 Break It Bad

To evaluate the performance of the bot in the *Break It Bad* game, I ran the bot in `not_dumb` intelligence mode for approximately 500 game simulations. In each simulation, the bot executed a series of up to 5 moves, and after every move, the resulting score was recorded. The goal was to analyze how the score evolved with each action and to identify the move sequences that led to the highest disruption scores.

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

To further understand the bot's decision-making, I extracted the sequence of moves that led to the highest score after 5 moves. The maximum score achieved was **113**. The corresponding sequence of moves is reported in table 1. Each entry in the table represents an edge that was attacked during the game. The edges are represented as pairs of nodes, indicating the connections that were targeted.

4.2 Save Our System

To evaluate different bot strategies in the *Save Our System* game, I tested all four intelligence levels from section 3, with approximately 20 trials for each and the bot's actions and resulting scores were recorded after each game step. The best scores for each strategy are reported in table 2. Then the results of all the scenario is aggregated and analyzed and I identified the best-performing strategy based on the highest final score achieved.

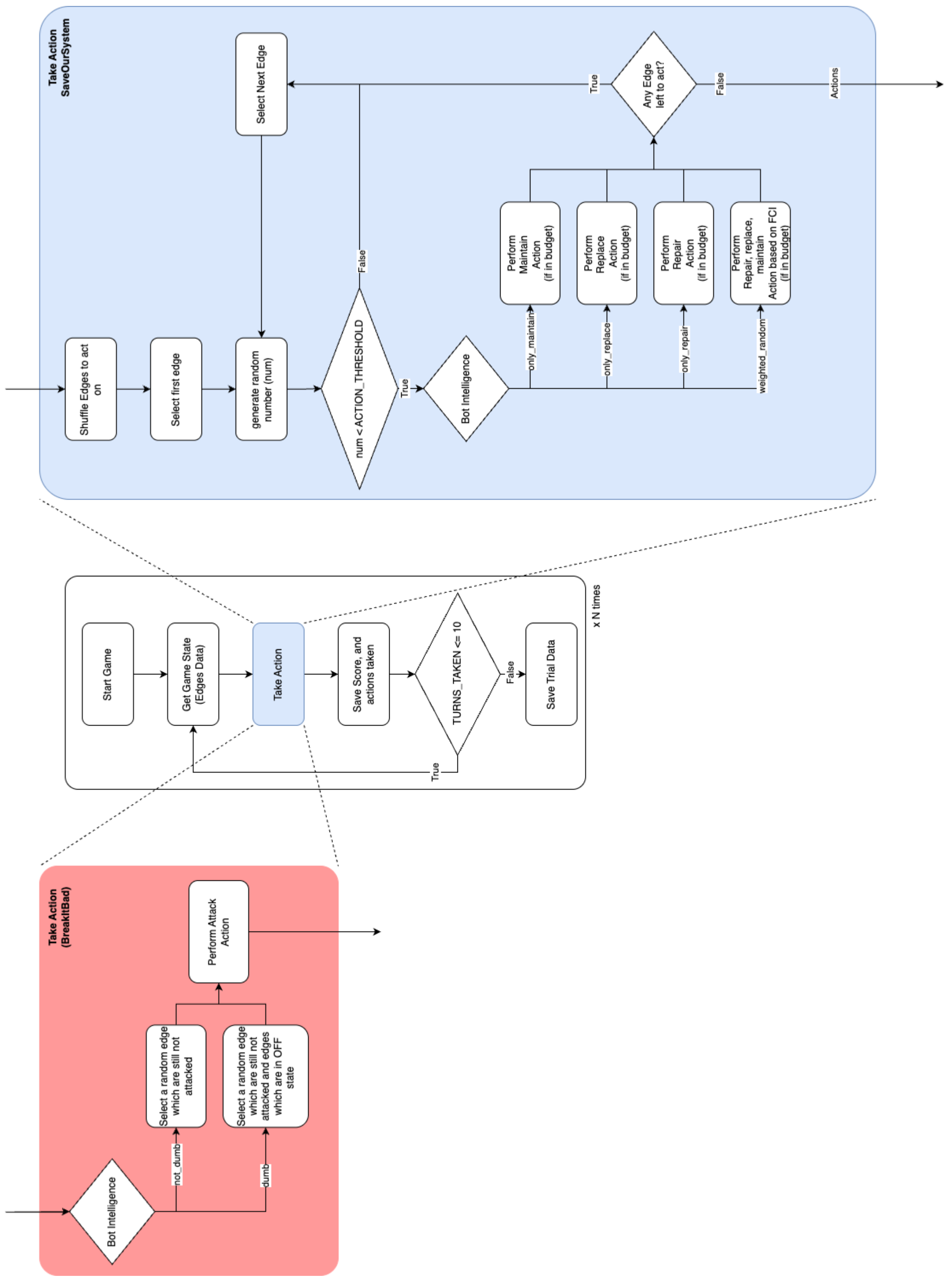
Take Action SaveOurSystem
Shuffle Edges to act on
Select first edge
generate random number (num)
num < ACTION_THRESHOLD
True
False
Bot Intelligence
Select Next Edge
only_maintain
only_replace
only_repair
weighted_random
Perform Maintain Action (if in budget)
Perform Replace Action (if in budget)
Perform Repair Action (if in budget)
Perform Repair, replace, maintain Action based on FCI (if in budget)
Any Edge left to act?
True
False
Actions
Start Game
Get Game State (Edges Data)
Take Action
Save Score, and actions taken
TURNS_TAKEN <= 10
True
False
Save Trial Data
x N times
Take Action (BreakItBad)
Bot Intelligence
not_dumb
dumb
Select a random edge which are still not attacked
Select a random edge which are still not attacked and edges which are in OFF state
Perform Attack Action

Figure 3: Flowchart illustrating the decision-making logic of the bots for both games. The center white box is implemented in the `bot.py` and `bot-savepursystem.py` files. The other two boxes are implemented in their respective game classes in `src/game.py`.

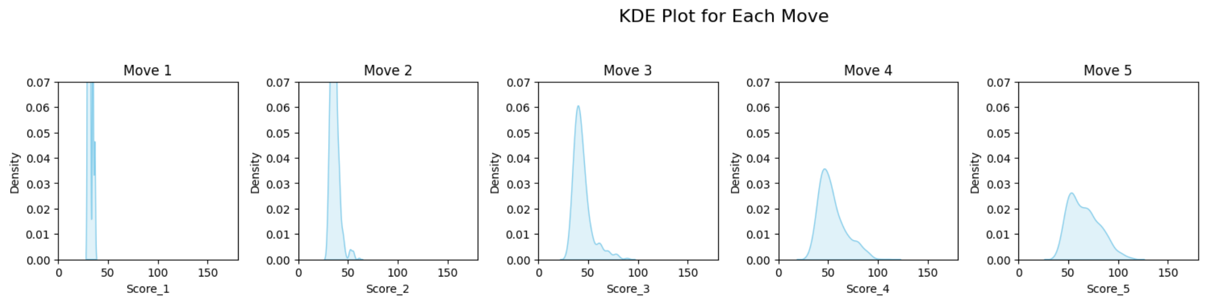

Figure 4: KDE plot showing the distribution of scores after each move over 500 simulations (max 5 moves shown). Higher density towards the right indicates increasing score trends.

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

Table 1: Sequence of moves leading to the highest score in *Break It Bad*. Here, each entry `x <-> y` represents an attack on the pipeline (edge) connecting node `x` to node `y`.

Table 3 highlights how the best move in `weighted_random` strategy progressively improved the score with a balanced mix of maintenance, repair, and replacement actions. The score increase aligns with strategically targeting weakened or high-impact pipelines, showcasing this strategy's adaptability.

5 Conclusion and Future Work

To conclude, the project was a great learning experience in which I have created a bot framework that can be further extended to add more games in the future, if needed. For this semester the development of BreakItBad and SaveOurSystem game bots is complete and they are working as expected. The bots were able to play the games with different strategies and intelligence level as per the given requirements.

For future work, The SaveOurSystem game bot can be improved to get more information about the best moves to make. This can be done by using reinforcement learning.

Strategy	Max Score Achieved
only_maintain	19
only_repair	15
only_replace	20
weighted_random	**22**

Table 2: Summary of maximum scores achieved by different strategies in the *Save Our System* game.

Move	Score	Actions Taken
1	3	MAINTAIN: 11↔7, 6↔5, 14↔13, 2↔1, 5↔9, 4↔1, 12↔8, 10↔14
2	6	MAINTAIN: 5↔9, 14↔13, 13↔12, 9↔10, 4↔1
3	7	MAINTAIN: 9↔12, 10↔14, 4↔1
4	10	REPAIR: 7↔3, REPLACE: 12↔8, MAINTAIN: 9↔10, 10↔14, 14↔13, 6↔5
5	11	MAINTAIN: 10↔14, 14↔13, REPAIR: 7↔3
6	14	REPLACE: 11↔7, MAINTAIN: 10↔14, REPAIR: 8↔4
7	17	REPAIR: 5↔9, 2↔1, MAINTAIN: 10↔14
8	20	MAINTAIN: 10↔14, REPLACE: 14↔13, REPAIR: 2↔3
9	21	REPAIR: 2↔1, REPAIR: 7↔3
10	22	REPAIR: 6↔5, 9↔12, MAINTAIN: 14↔13, 10↔14, REPAIR: 12↔8

Table 3: Score Progression of Best Trial for *Save Our System* Game. The table shows the score after each move and the actions taken. The actions are categorized as MAINTAIN, REPAIR, or REPLACE. Here, each entry `x <-> y` represents an attack on the pipeline (edge) connecting node `x` to node `y`.

Using this approach, we don't have to device the strategy for the bot like "only_maintain", "only_replace", etc. The bot will keep on playing the games, and will explore each and every move. After every iteration the bot will learn which move will give him best reward. After several hundred iterations, the bot will learn the best policy to play the game and then we can judge the bot which action to take. This approach will be best for this game as the this game environment is random, and to get fruitful results, we don't have to define the strategy for the bot but let the bot explore the game and learn from it.

6 Acknowledgements

I would like to thank Dr. Thomas Seager for giving me an opportunity to assist in his research project. His guidance and inputs helped shape the direction of the work and ensured its successful completion. I would also like to thank Mazin Abdel Magid for explaining the project and it's requirements in detail.

Appendix

A Repository Structure

Here I will give the structure of the repository and a brief description of each file and folder. And how to use, and where 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 the configuration files for each game that the bot reads. It contains some settings that the bot needs to run. Each game have its own configuration file. In this the user can change number of trials to run, 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/**: Source code for the bot framework and game-specific implementations.
  - `game.py`: This is main game file where the base game class and BreakItBad, SaveOurSystem derived game classes are defined. If any new game needs to be added, it can be added in this file.
  - `utils/`: This folder contains the utility files like data classes to save Node and Edges data, and some constants used in game.
- **bot.py**: The bot file for BreakItBad game. To run this game simply run it in ternminal using `python3 bot.py` command.
- **bot-saveoursystem.py**: The bot file for BreakItBad game. To run this game simply run it in ternminal using `python3 bot-saveoursystem.py` command.
- `requirements.txt`: List of Python dependencies required to run the project. It can be used to create python environment for this project.

B Detailed Bot Framework Design

Class Design

The codebase is centered around a base class `DysruptionGame`, which encapsulates shared behavior such as browser login, map interaction, state inspection, and common game utilities. This base class is extended by two specialized subclasses:

- `BreakItBad`: Implements logic to attack pipelines and maximize disruption.
- `SaveOurSystem`: Focuses on maintenance and repair actions under a constrained budget.

Each subclass overrides and extends the behavior of the base class to implement game-specific strategies, including action selection and score retrieval.

Class: DysruptionGame

The `DysruptionGame` class serves as the foundational blueprint for automating interactions with the simulation games. It abstracts repetitive tasks such as navigating the game UI, extracting map state, and performing 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 the node/edge storage.
- **reset_game(first_run=False)**: Opens the game in a browser and waits for key UI elements to load. If it's the first run, it waits for the login prompt; otherwise, it waits for the intro banner.
- **initiliaze_edges_nodes()**: Initializes empty lists to hold node and edge objects before parsing the game state.
- **login()**: Fills in the username field and submits the login form to enter the game environment.
- **start_game()**: Waits for the intro banner and map buttons to load, then clicks through to start the game.
- **activate_map_read_mode()**: Enables read mode by clicking the map selector, which allows edge and node information to be read.
- **find_edges()**: Finds all visible pipeline edges on the map and reads their visual properties (e.g., flow direction and status) to initialize them as `Edge` objects.
- **find_edges_state()**: Clicks each edge to extract additional metadata from the inspector panel such as temperature, flow, and FCI (Failure Condition Index).
- **find_nodes()**: Detects all nodes on the map and classifies them as supply or demand based on visual styles.

- **find_nodes_state()**: Extracts detailed data for each node (like name, penalty, demand, or shortfall) by reading the inspector panel.
- **random_action()**: Returns `True` or `False` based on a threshold probability. Useful for making bots act probabilistically.
- **action()**: Waits for the game's UI to be ready before allowing the next action. Used to introduce pacing between bot moves.
- **quit()**: Shuts down the browser session cleanly.

By consolidating this core functionality, the class serves as a reliable and reusable foundation for building bots for multiple games.

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 disrupt the fuel distribution network as much as possible within a limited number of moves.

Key components of the class include:

- **Constructor (__init__)**: Initializes the inherited attributes and settings for the BreakItBad game.
- **initiliaze_edges_nodes()**: Overrides the base method to additionally prepare separate lists for attacked, not attacked, and off-state edges.
- **get_score()**: Extracts the current score from the scoreboard element on the page using Selenium.
- **attack()**: Depending on the selected intelligence strategy, chooses an edge to attack—either randomly or selectively—and executes a simulated click on it.
- **action()**: Executes an attack via the `attack()` function and then calls the base class's `action()` method to continue the game loop.
- **is_game_over()**: Checks whether the game is complete by verifying if all available attackable edges have been used.
- **filter_edges()**: Sorts all edges into categories (not attacked, stopped, attacked) based on their current status for better decision-making.
- **deactivate_map_read_mode()**: Switches off map read mode and enables attack mode by clicking the corresponding UI button.
- **get_state()**: Combines all logic to refresh the current map state, including reading nodes, edges, and their properties. Includes retries to ensure robustness.

This class encapsulates all behavior specific to the BreakItBad game and provides a structured approach to implementing bot behavior for disruptive gameplay.

Class: SaveOurSystem

The `SaveOurSystem` class is a subclass of `DysruptionGame` and implements logic for the defensive gameplay scenario, where the player must maintain and restore a critical fuel network under a constrained budget. It includes multiple types of actions—repair, replace, and maintain—each with different costs and effects.

Key components of the class include:

- **Constructor (__init__)**: Inherits all base parameters and initializes edge categories for this game.
- **initiliaze_edges_nodes()**: Extends the base method to additionally set up lists for operational and destroyed edges.
- **get_score()**: Extracts the current performance score displayed in the game.
- **get_budget()**: Reads the current available budget from the game UI, which is needed to decide on allowable actions.
- **perform_action(edge, action_button_selector)**: Clicks the appropriate repair, replace, or maintain button, then applies the action on the selected edge.
- **action()**: Implements strategy-specific logic. Depending on the intelligence mode, the bot chooses which action to perform while staying within the budget. The `weighted_random` mode evaluates edge health (FCI) before deciding.
- **is_game_over()**: Placeholder check for game-ending condition (inherited from base), though not directly tied to game logic in this case.
- **filter_edges()**: Classifies edges as destroyed or operational based on their failure index, used to prioritize repairs.
- **get_state()**: Aggregates all state-gathering steps: node and edge detection, edge detail extraction, and classification.

This class enables simulation of different maintenance strategies and their effectiveness in keeping the network operational despite failures and budget constraints.

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. They create the objects of game classes, and execute the main game loop. The main loop can be visualized in the fig. 3 flowchart in the center white box.

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 same directory as the project, and update config/breakitbad.py and config/saveoursystem1.py files with the path to the chromedriver.
- Install the required dependencies using:

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

- Change the configuration files in the `config` folder as per your requirements. 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
```