

INFO 6205 Spring 2022 Project

Playing Tic-Tac-Toe with Menace

Authors: Pranshu Dixit, Aditya Kinare, Shreya Patil

NUID: 001585095, 001095881, 002193245

Introduction

In this project, we implement “The Menace” by replacing matchboxes with values in a hash table (the key will be the state of the game). We train the Menace by running games played against the “human” strategy, which is based upon the optimal strategy.

Aim

Train the Menace by running games played against the “human” strategy, which is based upon optimal strategy (see Tic-tac-toe).

Chosen values are:

- $\text{Alpha} = (\text{total number of empty positions on board} + 2)/2$
- $\text{Beta} = 3$
- $\text{Gamma} = 1$
- $\text{Delta} = 1$

Application Demo Video

You can view the video in the link which explains a walkthrough of the application and how to run it on your local machines.

Video Link: <https://youtu.be/LozkpJE-yGQ>

Program

Data Structures & classes

- Array Lists
- Hash table

List of Classes:

- PlayingTicTacToeApplication.java (Main Class):
 - This is the main class that starts the server. It calls the train_menace() function and trains the menace first so that the user doesn't have to wait for training the menace as it takes more time with a GUI interface.
- GameService
- GameController
 - The endpoints are defined here. They interact with the Javascript UI.
- Gameplay, Game
- TrainMenace
 - This class has the train_menace() function which trains the menace and keeps track of the number of losses, wins, and draws in a game by the menace and it will train the menace accordingly.
 - For each win, it awards 3 beads for every move.
 - For each draw, it awards 1 bead for every move.
 - For each loss, it removes 1 bead for every move.
- TrainedMenacePlayerStorage
- Board
- MenacePlayer:
 - This class keeps track of the game states in match_boxes and moves played in every game in moves_played.

List of Built-in Classes:

- Java.awt.Color
- java.awt.Graphics
- java.awt.Graphics2d
- java.awt.Shape
- java.awt.geom.Ellipse2d
- java.awt.Dimension
- java.awt.event.ActionEvent
- java.awt.event.ActionListener
- java.util.Random, java.util.HashMap
- java.util.ArrayList, java.util.Iterator
- java.util.List, java.util.Map, java.util.Observable
- java.util.Observer
- javax.swing.BoxLayout
- javax.swing.JButton
- javax.swing.JFrame
- javax.swing.JPanel
- javax.swing.JSplitPane.

Algorithm

- We use Reinforcement Learning Algorithm to train the menace.
- To train the menace, we initially play 2 new menaces with each other for roughly 100000 games.
- Then we combine the states of both the menaces to make a super menace and take a new menace and train it with the combined menace. We call this menace the learning menace
- The super menace plays the game with a winning probability of 0.8.
- The learning menace is saved after 3 different game timestamps which we then use to define our difficulty levels.
- The easy-level menace is saved after 100 games. The probability of winning is approximately 0.3
- The medium-level menace is saved after 1000 games. The probability of winning is approximately 0.6.
- The hard level menace is saved after 100000 games.

Invariants

- As Player X plays first, in every game

number of moves by Player X \geq number of moves by Player O

Flow Charts (UI Flow)

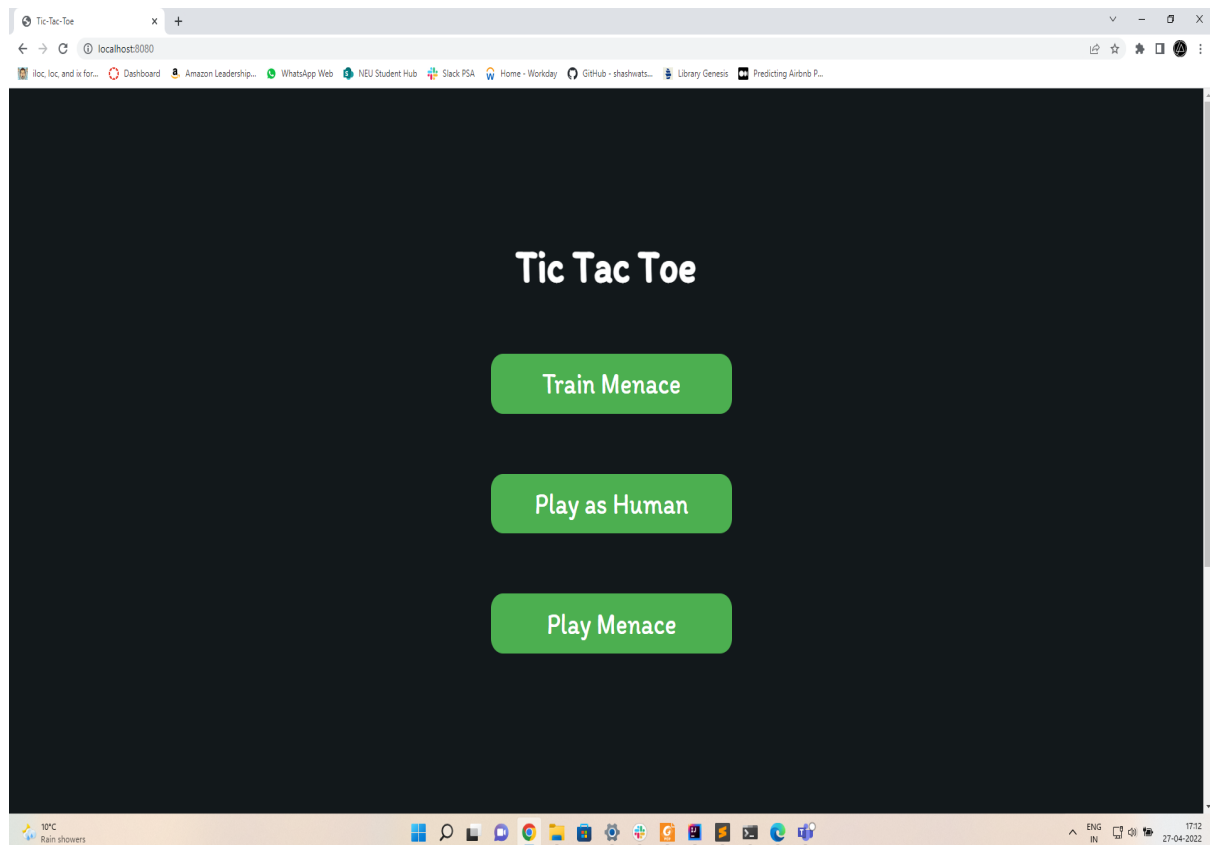


Figure 1.1: First Page

When the application runs, the first page opened three options are available for the user to choose:

1. Watch the menace train itself
2. Play as a human against a menace
3. Play a menace against another menace

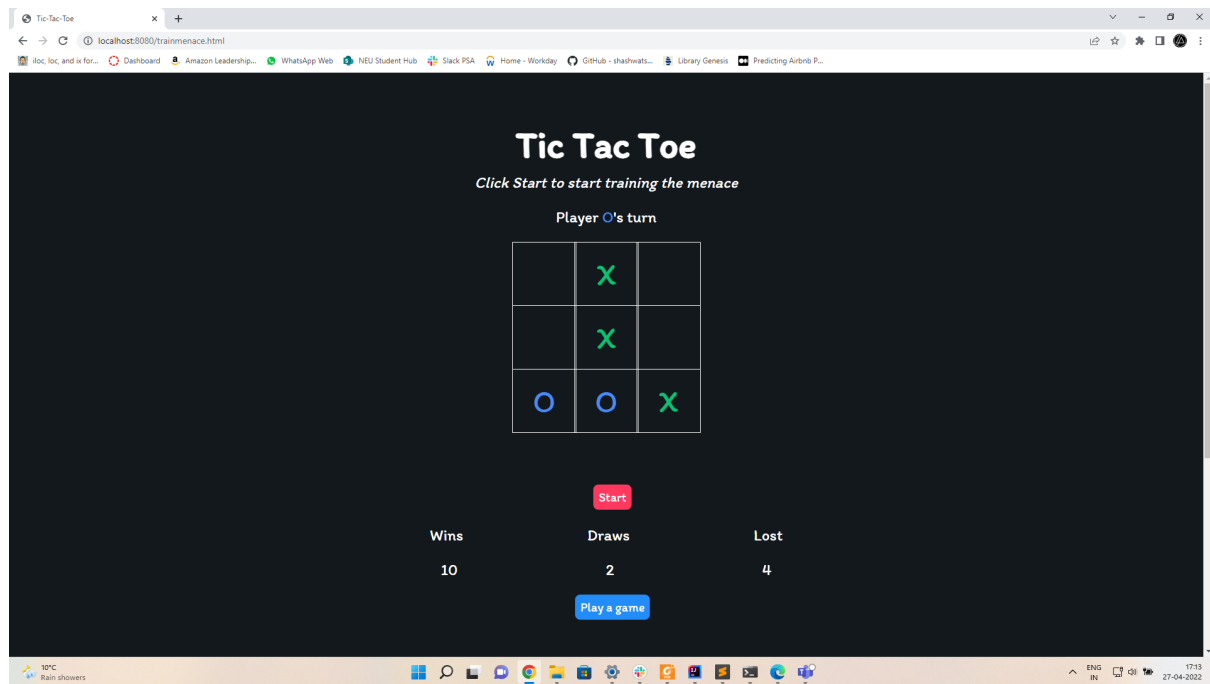


Figure 1.2: Watching the menace getting trained

- Here the user watches the Menace get trained, with the logs of Wins, Draws, and Losses.

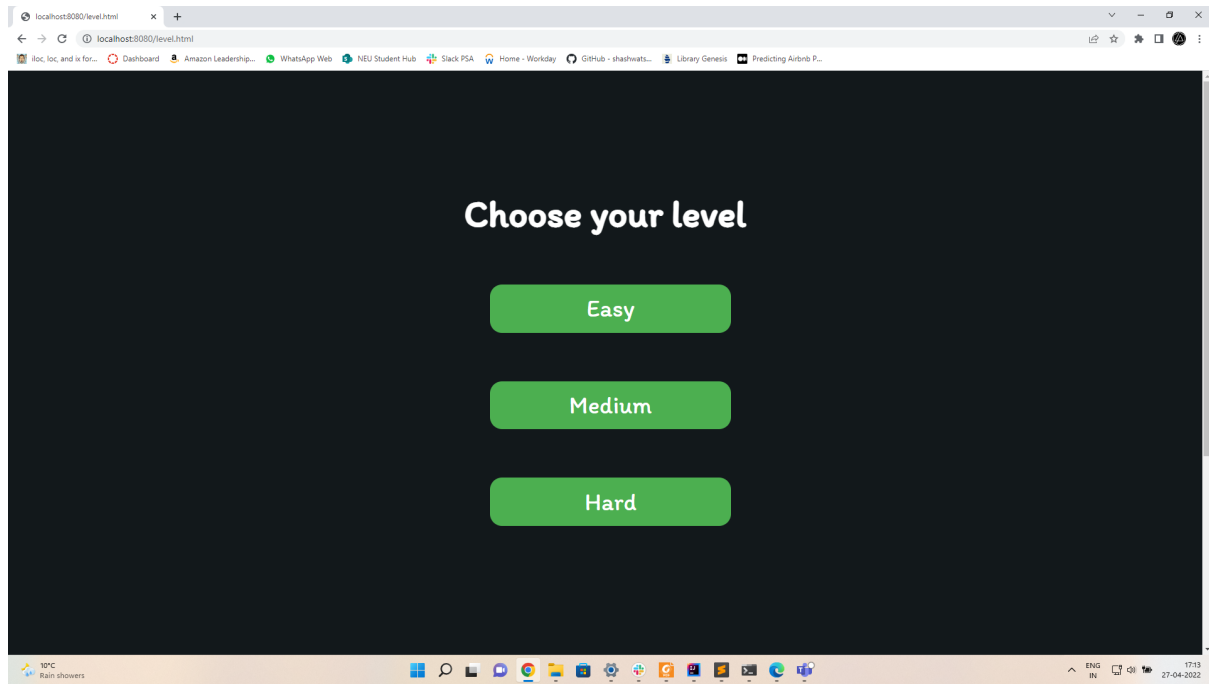


Figure 1.3: Choose level

- If the user chooses to play against the menace, the application asks to choose the level.
- The level here refers to the number of steps Menace has been trained for.
- At the easy level, the menace has been trained for fewer steps, and at the hard level, the menace has been trained for more steps.
- The probability of making the optimal move increases as the level increases.

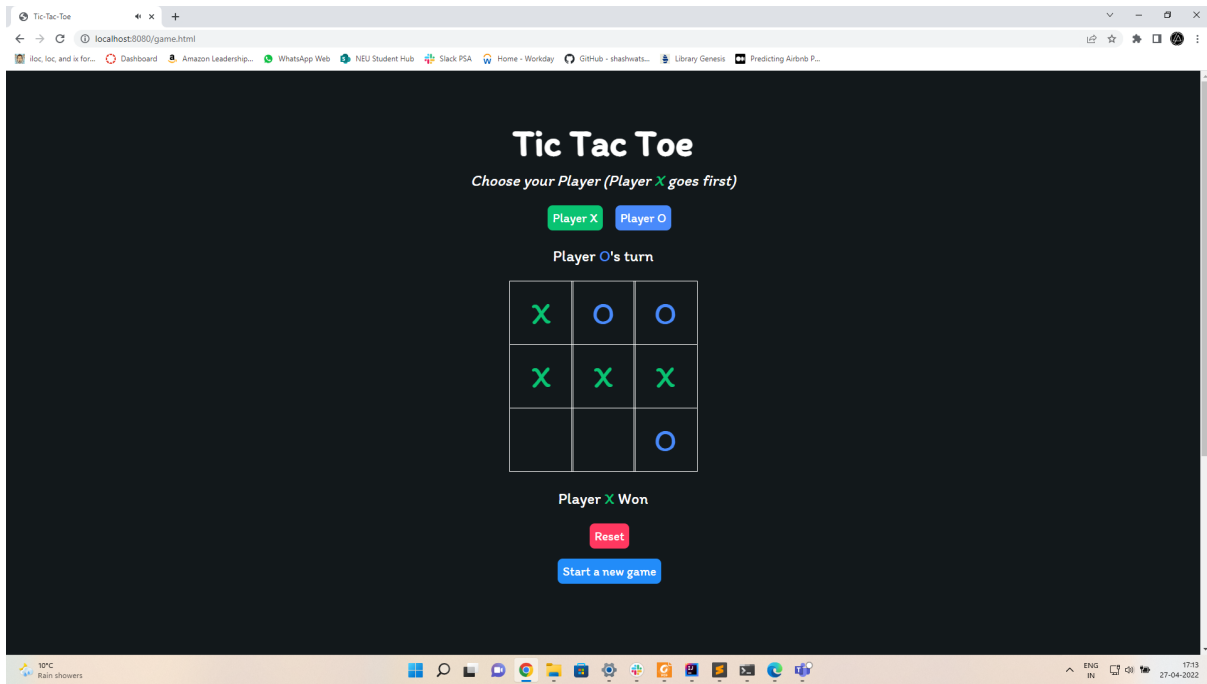


Figure 1.4: Play a game against Menace (Player X wins)

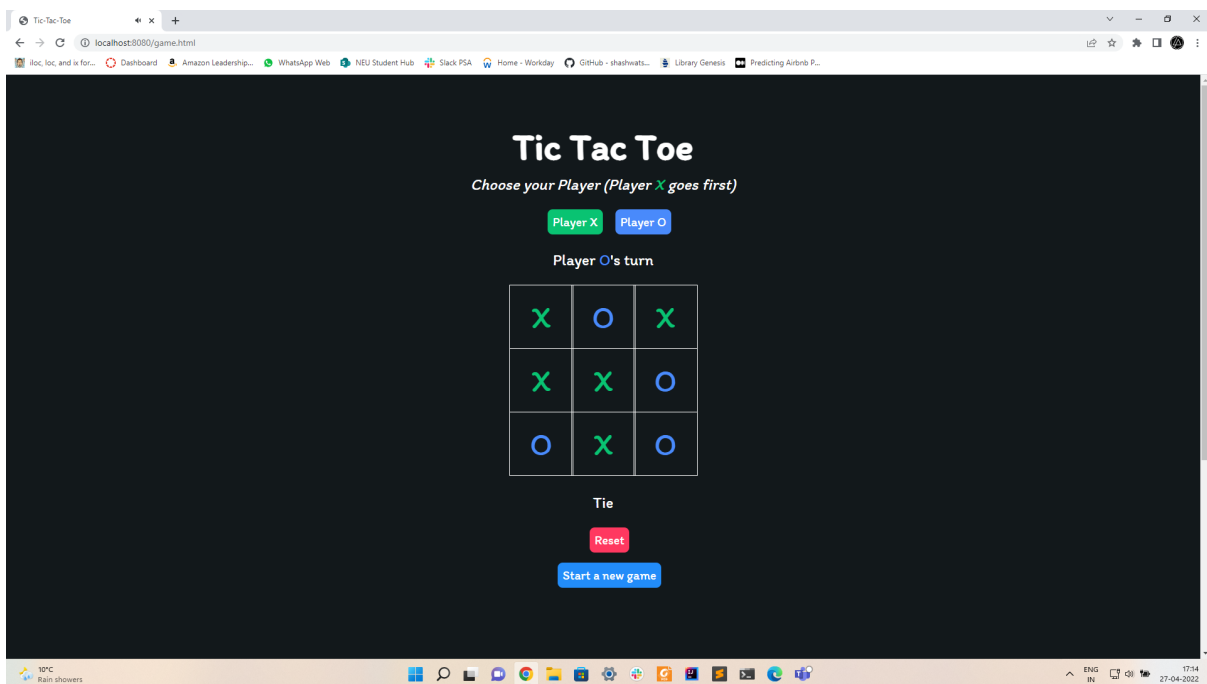


Figure 1.5: Play a game against Menace (Match is a tie)

- The user is asked to play as Player X or Player O. The Player X plays the first move.
- The game result is displayed.

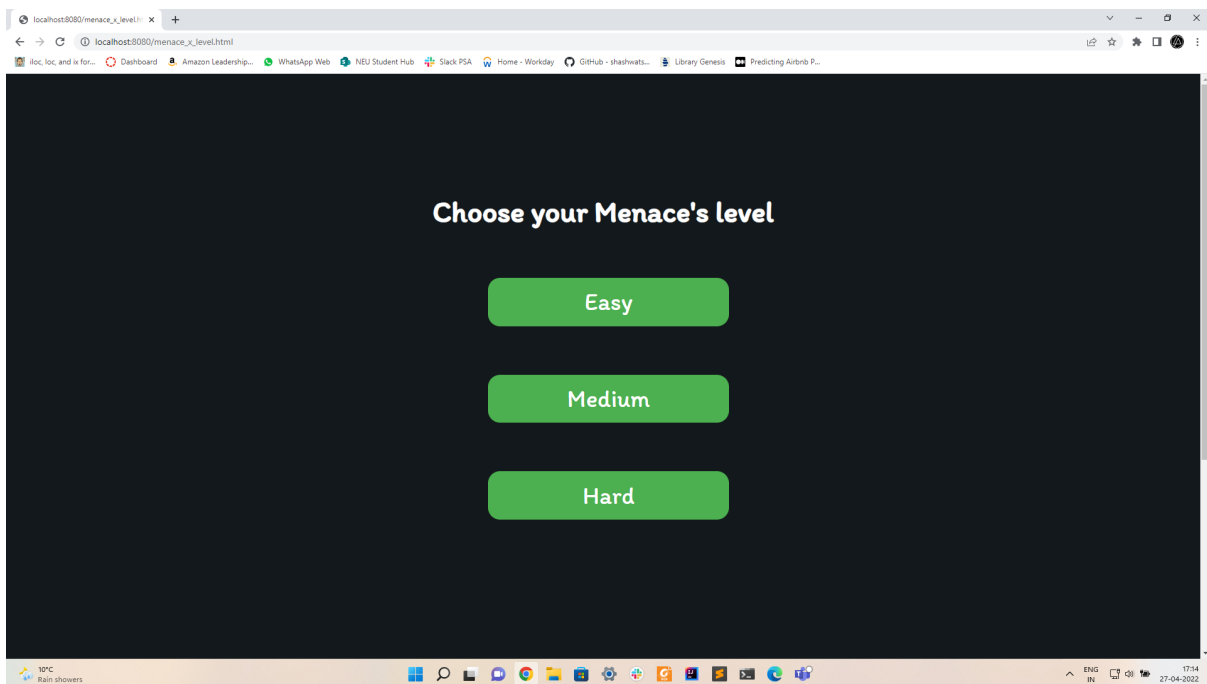


Figure 1.6: Choose the User's menace level

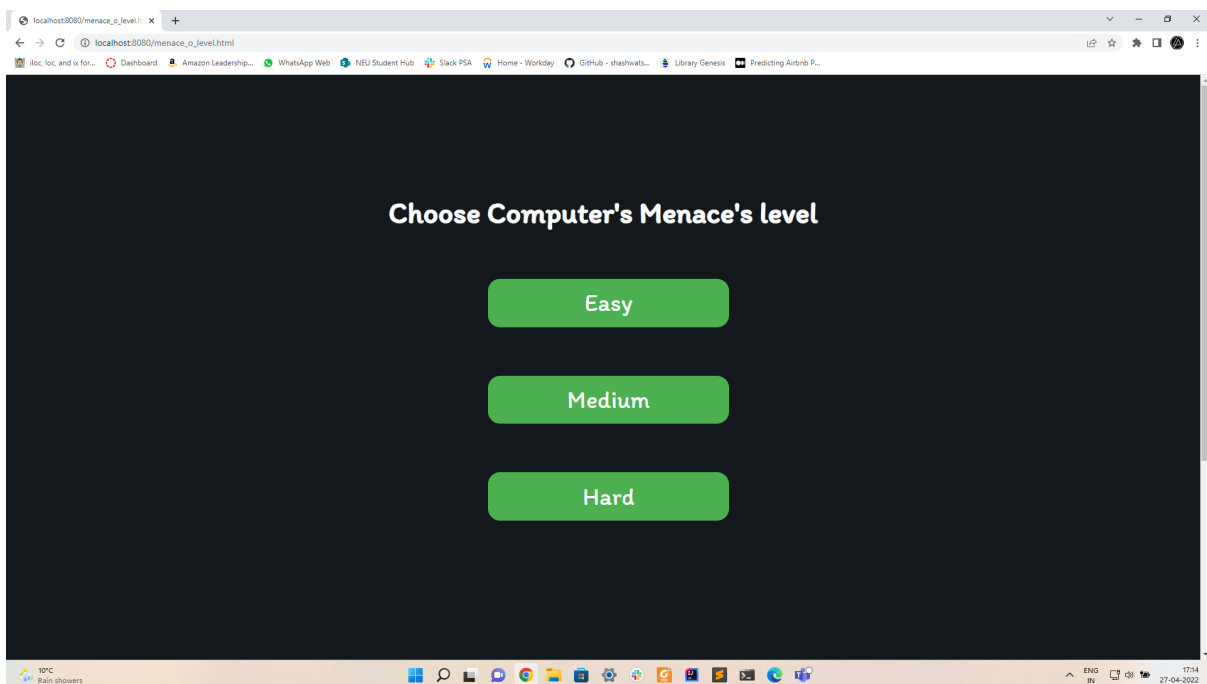


Figure 1.7: Choose Computer's menace level

- If the user chooses to play a menace against another menace, the user is asked to choose the user's and computer's menace level

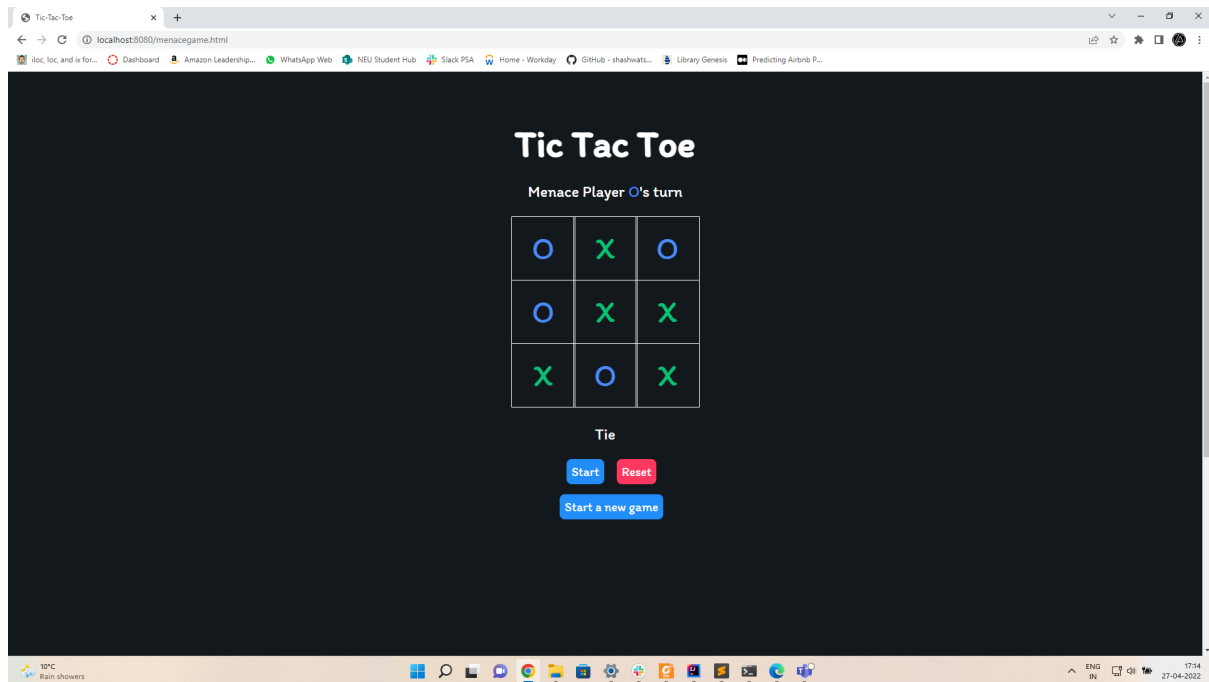
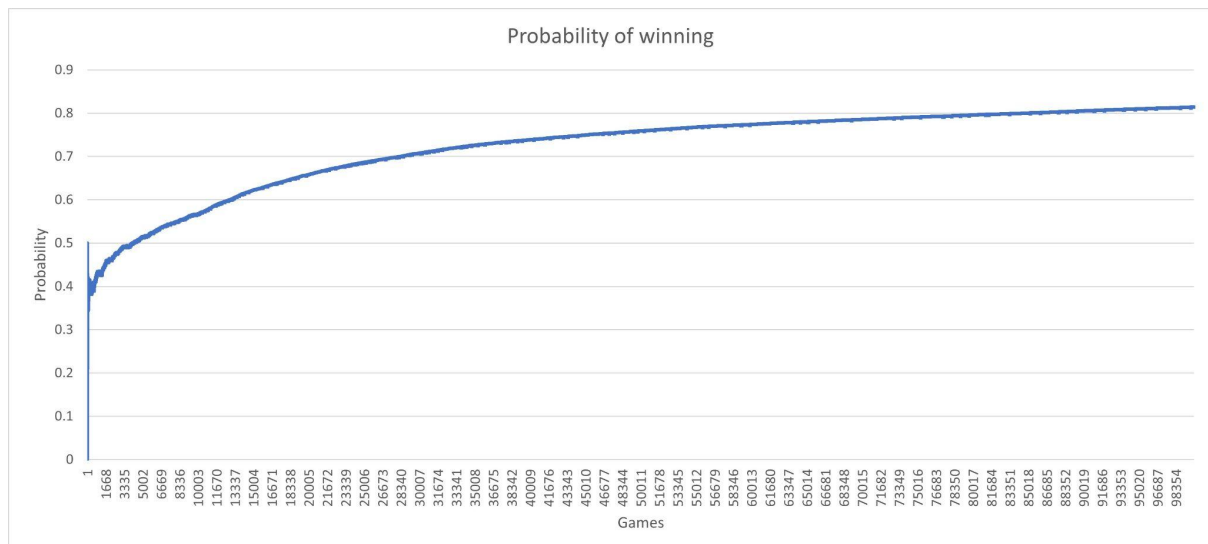


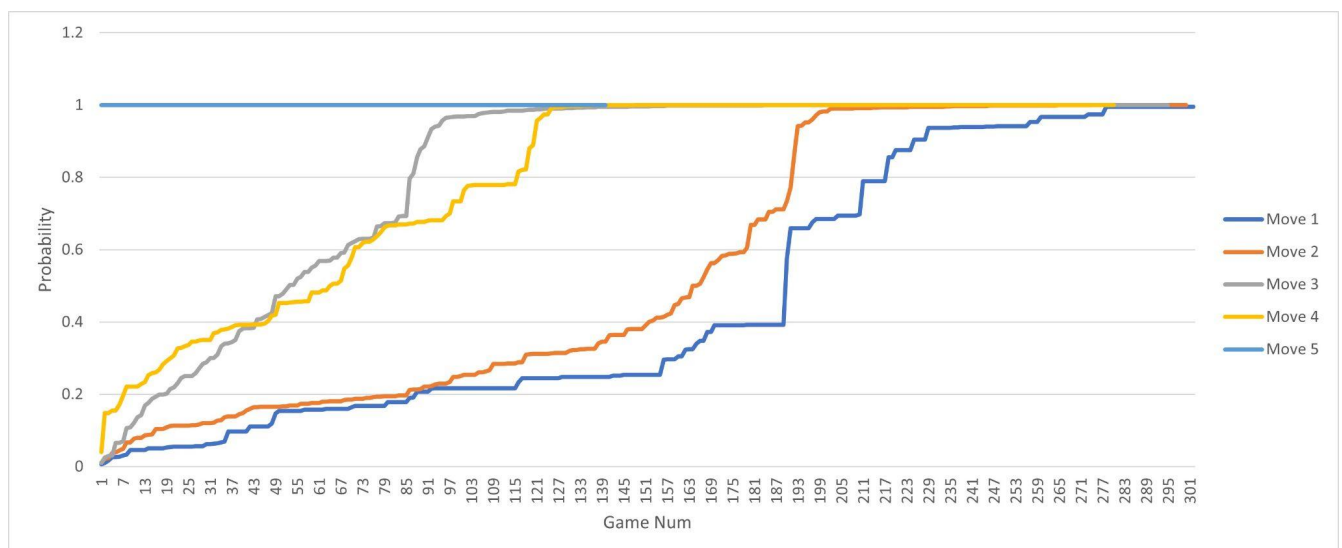
Figure 1.8: A game between 2 Menaces is played

Observations & Graphical Analysis

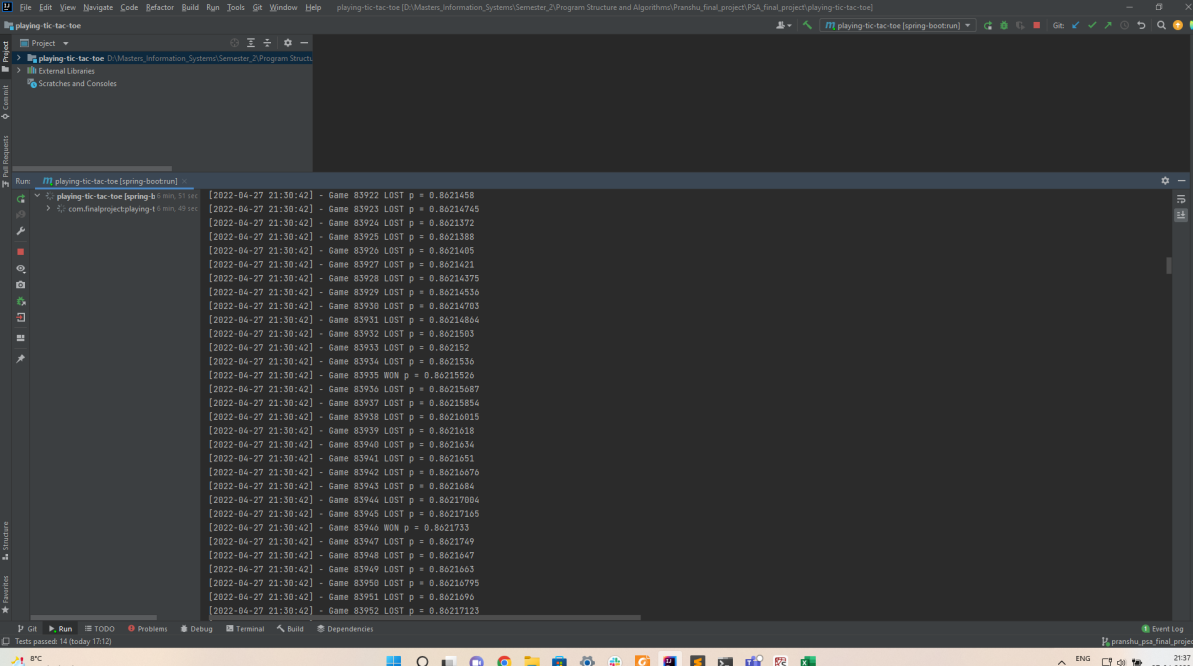
- The probability of the menace winning the next game increases as it plays more games. The plot confirms the same.



- The probability of playing the optimal move increases as the game progresses. We played 100 games to prove this with the opponent of Menace having a $p = 0.8$ of winning the game.



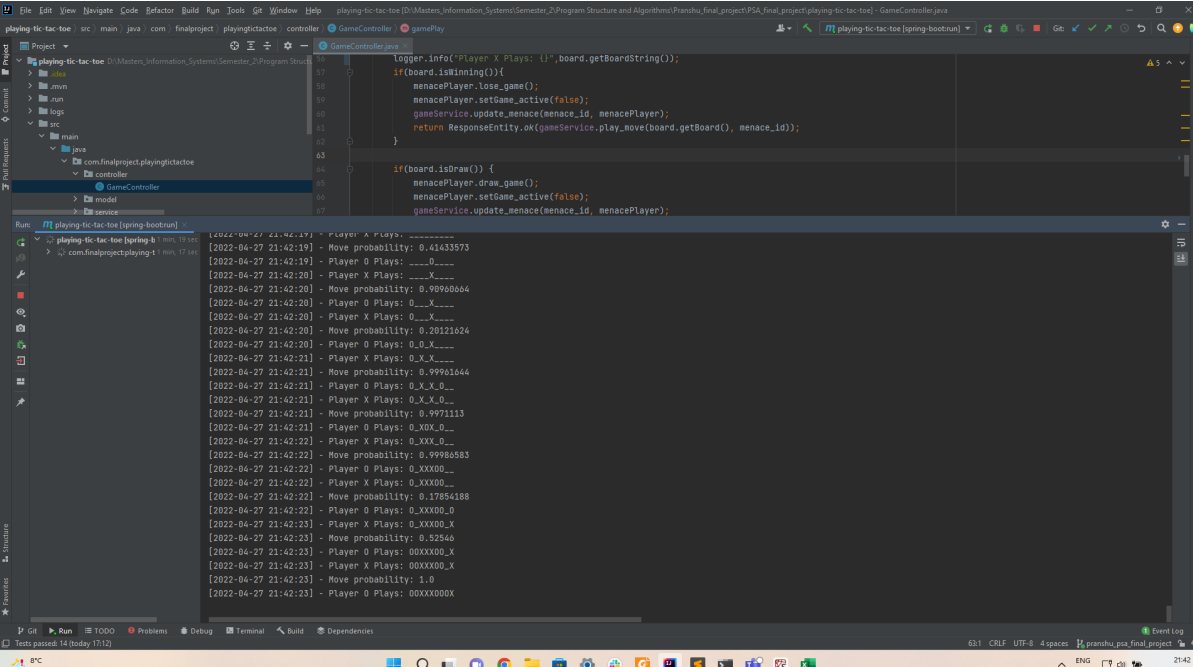
Logging



The screenshot shows an IDE window titled "playing-tic-tac-toe" with a project structure on the left. The main editor displays a log of game results for "playing-tic-tac-toe (spring-bootrun)". The log entries are as follows:

```
[2022-04-27 21:30:42] - Game 83922 LOST p = 0.8621458
[2022-04-27 21:30:42] - Game 83923 LOST p = 0.86216765
[2022-04-27 21:30:42] - Game 83924 LOST p = 0.8621372
[2022-04-27 21:30:42] - Game 83925 LOST p = 0.8621388
[2022-04-27 21:30:42] - Game 83926 LOST p = 0.8621405
[2022-04-27 21:30:42] - Game 83927 LOST p = 0.8621421
[2022-04-27 21:30:42] - Game 83928 LOST p = 0.86214375
[2022-04-27 21:30:42] - Game 83929 LOST p = 0.86214536
[2022-04-27 21:30:42] - Game 83930 LOST p = 0.86216703
[2022-04-27 21:30:42] - Game 83931 LOST p = 0.86214864
[2022-04-27 21:30:42] - Game 83932 LOST p = 0.8621503
[2022-04-27 21:30:42] - Game 83933 LOST p = 0.862152
[2022-04-27 21:30:42] - Game 83934 LOST p = 0.8621536
[2022-04-27 21:30:42] - Game 83935 WON p = 0.86215526
[2022-04-27 21:30:42] - Game 83936 LOST p = 0.86215687
[2022-04-27 21:30:42] - Game 83937 LOST p = 0.86215854
[2022-04-27 21:30:42] - Game 83938 LOST p = 0.86216015
[2022-04-27 21:30:42] - Game 83939 LOST p = 0.8621618
[2022-04-27 21:30:42] - Game 83940 LOST p = 0.8621634
[2022-04-27 21:30:42] - Game 83941 LOST p = 0.8621651
[2022-04-27 21:30:42] - Game 83942 LOST p = 0.86216676
[2022-04-27 21:30:42] - Game 83943 LOST p = 0.8621684
[2022-04-27 21:30:42] - Game 83944 LOST p = 0.86217004
[2022-04-27 21:30:42] - Game 83945 LOST p = 0.86217165
[2022-04-27 21:30:42] - Game 83946 WON p = 0.8621733
[2022-04-27 21:30:42] - Game 83947 LOST p = 0.8621749
[2022-04-27 21:30:42] - Game 83948 LOST p = 0.8621647
[2022-04-27 21:30:42] - Game 83949 LOST p = 0.8621663
[2022-04-27 21:30:42] - Game 83950 LOST p = 0.86216795
[2022-04-27 21:30:42] - Game 83951 LOST p = 0.8621696
[2022-04-27 21:30:42] - Game 83952 LOST p = 0.86217123
```

Figure 1.9: Logging Menace getting trained



The screenshot shows an IDE window titled "playing-tic-tac-toe" with a project structure on the left. The main editor displays the code for "GameController.java". The code is as follows:

```
logger.info("Player X Plays: {}", board.getBoardString());
if (board.isWinning()) {
    menacePlayer.lose_game();
    menacePlayer.setGame_active(false);
    gameService.update_menace(menace_id, menacePlayer);
    return ResponseEntity.ok(gameService.play_move(board.getBoard(), menace_id));
}

if (board.isDraw()) {
    menacePlayer.draw_game();
    menacePlayer.setGame_active(false);
    gameService.update_menace(menace_id, menacePlayer);
}
```

The log shows the following moves and probabilities:

```
[2022-04-27 21:42:19] - Player X Plays: -----
[2022-04-27 21:42:19] - Move probability: 0.41635573
[2022-04-27 21:42:19] - Player O Plays: ____O____
[2022-04-27 21:42:20] - Player X Plays: ____X____
[2022-04-27 21:42:20] - Move probability: 0.90960664
[2022-04-27 21:42:20] - Player O Plays: ____X____
[2022-04-27 21:42:20] - Player X Plays: ____X____
[2022-04-27 21:42:20] - Move probability: 0.20121624
[2022-04-27 21:42:20] - Player O Plays: ____O_X____
[2022-04-27 21:42:21] - Player X Plays: ____X____
[2022-04-27 21:42:21] - Move probability: 0.99961644
[2022-04-27 21:42:21] - Player O Plays: ____X_X_O____
[2022-04-27 21:42:21] - Player X Plays: ____X_X_O____
[2022-04-27 21:42:21] - Move probability: 0.9971113
[2022-04-27 21:42:21] - Player O Plays: ____O_X_O____
[2022-04-27 21:42:22] - Player X Plays: ____XXX_O____
[2022-04-27 21:42:22] - Move probability: 0.99985583
[2022-04-27 21:42:22] - Player O Plays: ____XXXO_O____
[2022-04-27 21:42:22] - Player X Plays: ____XXXO_O____
[2022-04-27 21:42:22] - Move probability: 0.17854188
[2022-04-27 21:42:22] - Player O Plays: ____XXXO_O____
[2022-04-27 21:42:23] - Player X Plays: ____XXXO_X____
[2022-04-27 21:42:23] - Move probability: 0.92846
[2022-04-27 21:42:23] - Player O Plays: ____OXXXO_O_X____
[2022-04-27 21:42:23] - Player X Plays: ____OXXXO_O_X____
[2022-04-27 21:42:23] - Move probability: 1.0
[2022-04-27 21:42:23] - Player O Plays: ____OXXXO_OX____
```

Figure 1.10: Logging each player's moves with move probability

Testcases

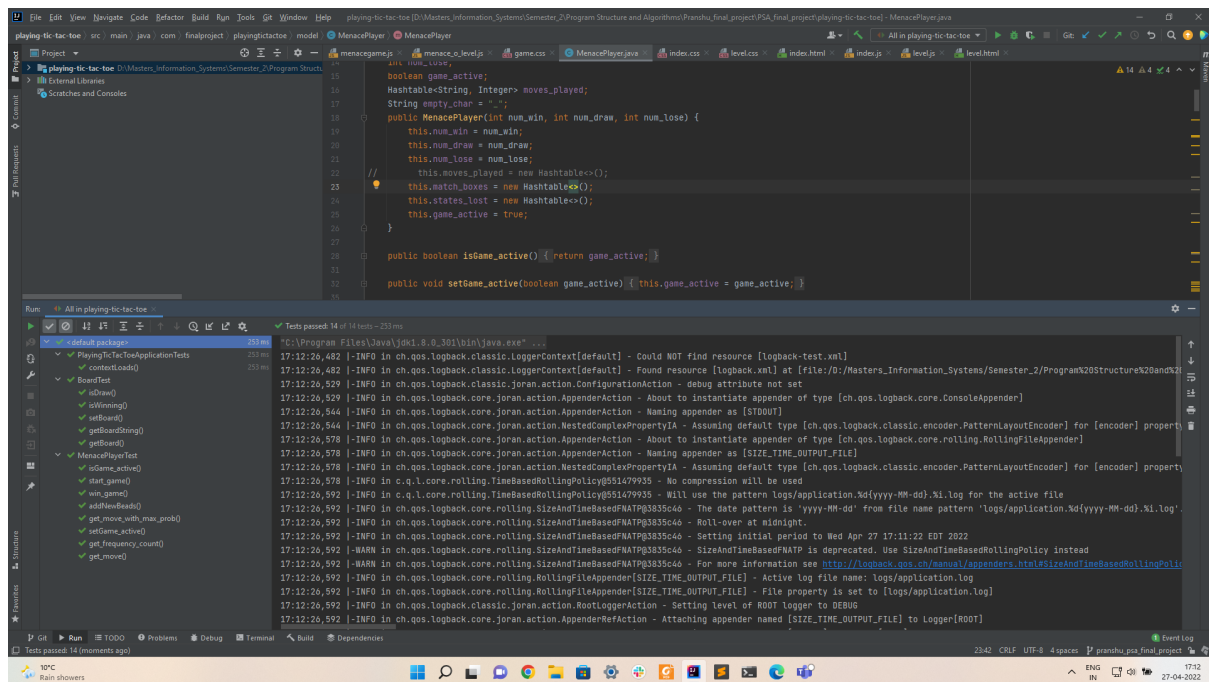


Figure 1.11: Unit Test Cases Passed

Conclusion

- In this project, we train the menace by running games played against the “human” strategy, which is based upon optimal strategy. The winning probability of our menace after training for 100000 games is 0.8. As our menace plays more games, its winning probability increases.

References

1. <https://spring.io/guides/tutorials/rest/>
2. <https://www.youtube.com/watch?v=TtisQ9yZ2zo&t=2210s>
3. <https://www.twilio.com/blog/create-rest-apis-java-spring-boot>