Program Structures & Algorithms

Spring 2022 Team Project

The Menace

ERARICA MEHRA

SRI VYSHNAVI KOTHA (002985810)

TANYA SHAH (002988713)

INTRODUCTION

AIM

- Implement “The Menace” by replacing matchboxes with values in a hash table (key will be the state of the game).
- Train the Menace by running games played against “human” strategy, which is based upon optimal strategy.

Choose values for:

- ***Alpha*** (the number of ”beads” in each “matchbox” at the start of the game—may be different for each move: first move, second move, etc.)
- ***Beta*** (the number of ”beads” to add to the “matchbox” in the event of a win)
- ***Gamma*** (the number of ”beads” to take to the “matchbox” in the event of a loss)
- ***Delta*** (the number of ”beads” to add to the “matchbox” in the event of a draw)

Human Strategy - Chooses optimal strategy with probability p*. In the “zone,” choose a random move.

APPROACH

The approach followed in this project is inspired by Reinforcement Learning and Neural Networks. The game definition is defined using Markov Decision Process (MDP), which is a mathematical framework used to model decision-making situations where outcomes are partly controlled and partly random.

Menace is essentially teaching the program to play Tic-tac-toe and training the data to predict the maximum probable move to win and which is played between two players.

Game Types:

1. Menace Vs Random
2. Menace Vs Human

Player Definition - In this approach the game is always started by the Menace (*Player 1*), making ***Crosses*** or ***X*** default playing symbols for the Menace. In both the game types, for Random or Human ***Noughts*** or ***O*** (*Player 2*)is the default symbol.

State Set for Menace contains all the boards for Menace's every move including the initial empty board in the game. This is represented by a 9-character String label in the format "---------" where each value of **"*X*"** & "***O***" are placed in the position of the boards [1-9] and empty values are represented by "**-".**

The Action Set for Menace contains all the available options to play.

Rewards Definitions: (WIN, LOSE, DRAW)

ALPHA - Initial beads assigned - 20

BETA - Beads added when Menace wins - 5

GAMMA - Beads removed when Menace loses- 3

DELTA - Beads added in case Draw - 1

PROGRAM

CODE - Play Method

```
public Game play() throws Exception {

    // get all possible states in a hashmap with key as state/board and Box(initial)
    // as value
    Board board = new Board("---------"); // initial state
/   Reading from training  data and storing in a map
    Map<String, Box> map = new HashMap<>();
    map = FileUtils.readTrainingData();
    List<Board> allOccuredStates = new LinkedList<>();
/   Properties props = loadPropertiesFile("position.properties");
    Set<Integer> cellsBlocked = new HashSet<>();

    Game game = Game.PLAYING;
    for (Map.Entry<String, Box> entry : map.entrySet()) {
        entry.getValue();
    }

    // random number generator
    StringBuilder sb = new StringBuilder();
    while (game == Game.PLAYING) {
        int computerMove = 0;

        if (cellsBlocked.size() == 9) {
            allOccuredStates = draw(game, board, allOccuredStates);
            game = Game.DRAW;
            if (!map.containsKey(board.getLabel()))
                map.put(board.getLabel(), new Box());
            break;
        }

        if (!map.containsKey(board.getLabel())) {
            map.put(board.getLabel(), new Box());
            sb.append(board.getLabel());
            sb.append(System.getProperty("line.separator"));
        }

        if (map.containsKey(board.getLabel())) {
            Box box = map.get(board.getLabel());
            computerMove = getMaxiMumProbableMove(box, cellsBlocked);

            cellsBlocked.add(computerMove);
            if (cellsBlocked.size() == 9) {
                allOccuredStates = draw(game, board, allOccuredStates);
                game = Game.DRAW;
```

```
                if (!map.containsKey(board.getLabel()))
                    map.put(board.getLabel(), new Box());
                break;
            }

        }

        log.info("Computer Move: " + computerMove);
        board.getCompPositions().add(computerMove);

        int humanMove = generatingRandomMove(cellsBlocked);
        cellsBlocked.add(humanMove);
        board.getPlayerPositions().add(humanMove);

        log.info("Human Move   : " + humanMove);
        board.setMoveNumber(humanMove);

        // update board --input current board, + computerMove
        board = gameService.placePos(board, computerMove, Player.MENACE.toString());
        board = gameService.placePos(board, humanMove, Player.RANDOM.toString());

        game = gameService.checkWin(board);
        if (game != Game.PLAYING) {
            board.setLabel(gameService.convertBoardtoLabel(board.getCurrentBoard()));
            log.info("Labels: " + board.getLabel());
            board.setMoveNumber(humanMove);
            allOccuredStates.add(new Board(board.getLabel(), board.getMoveNumber()));
            if (!map.containsKey(board.getLabel()))
                map.put(board.getLabel(), new Box());
            break;
        }

        board.setLabel(gameService.convertBoardtoLabel(board.getCurrentBoard()));
        log.info("Labels: " + board.getLabel());
        board.setMoveNumber(humanMove);

        allOccuredStates.add(new Board(board.getLabel(), board.getMoveNumber()));
    }
    FileUtils.writeToFile(sb.toString(), "results.txt");
```

```
        // maintain the new hashmap
        StringBuilder data = new StringBuilder();
        for (Map.Entry<String, Box> entry : map.entrySet()) {
            data.append(entry.getKey() + "=" + entry.getValue().toString());
            data.append(System.getProperty("line.separator"));
//          writeToProps();
        }
        FileUtils.writeToProps(map);
        FileUtils.writeToFile(data.toString(), "training.txt");

        return game;
    }
```

DATA STRUCTURE AND CLASSES

HashTable - We are loading the trained data from the training.properties in a hashtable. We stored the key value pairs from the training.properties (position.properties for initial rounds) file into a hash table. The key was a String value of the current state of the board. We are using this String value (label) as our hash key since it would be unique for every state. The value of the hash table is a list of beads.

PriorityQueue - We generated Menace's move from a priority queue. We calculated the priority based on the frequency of beads in the list. The beads with the maximum frequency were assigned the maximum probability.

Set - We used Set to maintain the number of cells already blocked in the game.

ALGORITHMS

Training Data: For initial training rounds we used a position.properties file that is currently uploaded with the project. This file contained key-value pairs of 304 possible states of the tic tac toe game. Each state was initialized with a list of randomly generated arrays(list of beads). On each win, lose or draw situation, the list of beads gets updated. This is how we trained all possible states.

We used key-value pairs to train the menace against an optimal strategy.

Menace Move : We used Priority Queue to calculate the highest priority move of a state.

Random Move : We used a random number generator to generate a random number(for the human move).

After the list of beads for each state was updated, the resulting data is stored in a new file called training.properties. We used this file to fetch the trained data in the later rounds.

We ran the play method in a loop thousands of times to train the data.

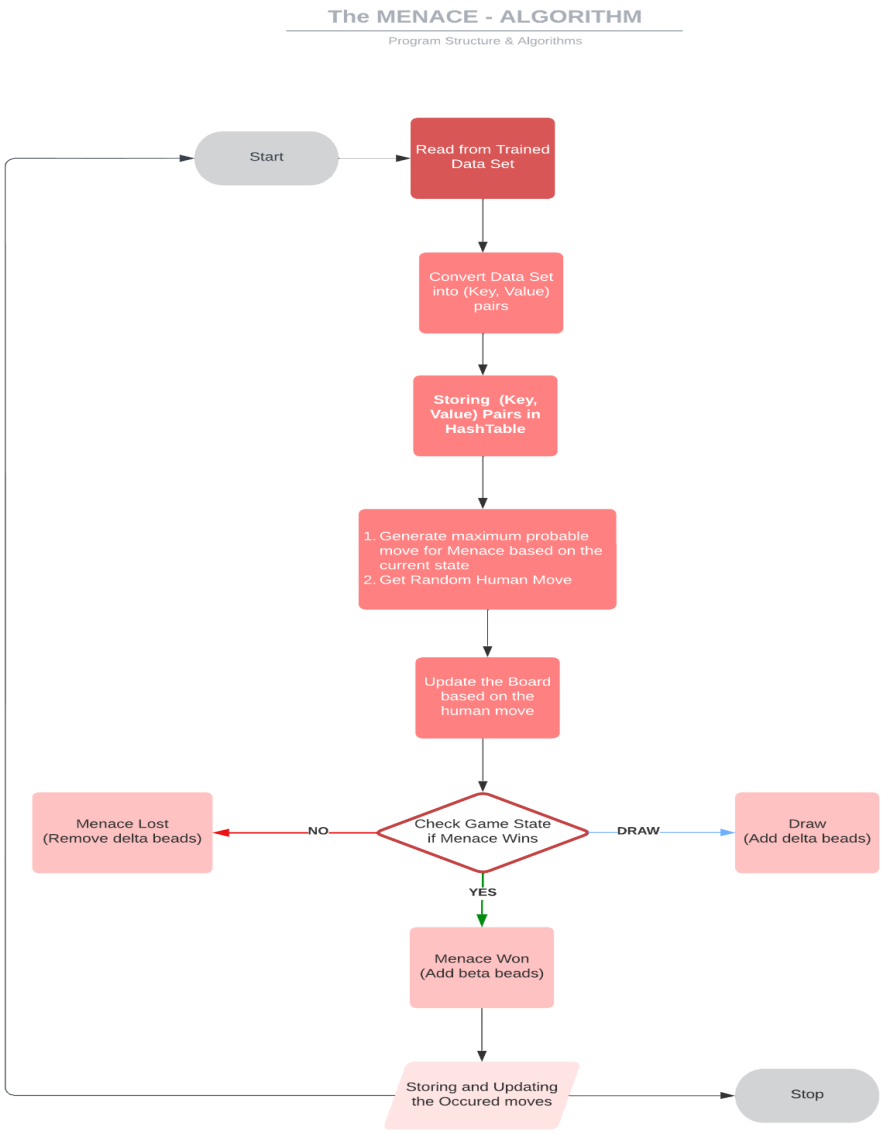
INVARIANTS -

We used Hashtable as a data structure to maintain all the trained states in the tic tac toe game. This is because Hash Table maintains a unique key value for every value in the table. The uniqueness of the hash key in the hashtable acts as the invariant in our algorithm.

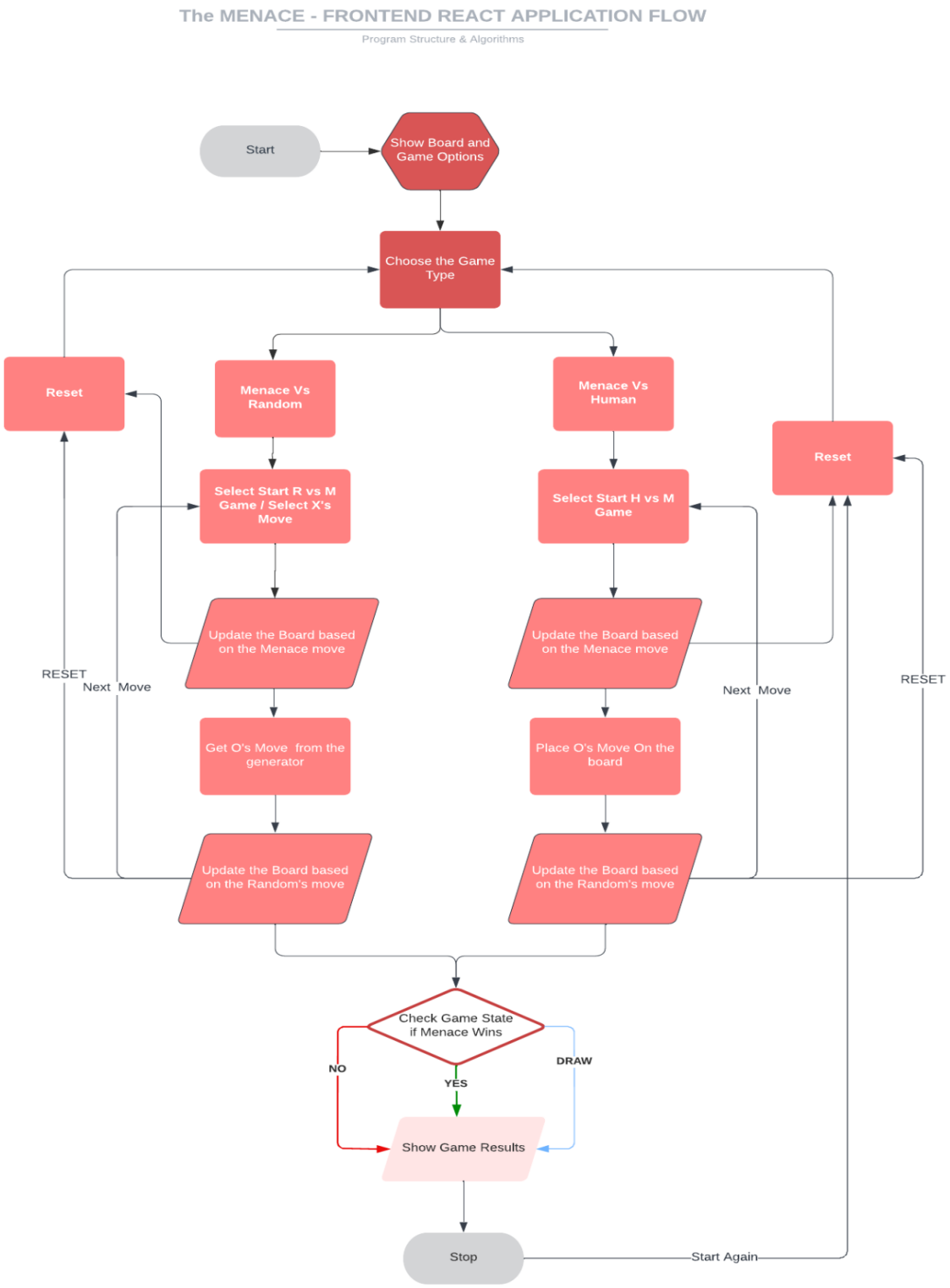
Another invariant in the algorithm is that the algorithm maintains a game state throughout the flow of the program. It starts with the game state ‘PLAYING’ and stops the game when the state changes to either ‘DRAW’, ‘WIN’, ‘LOSE’.

FLOW CHARTS

ALGORITHM FLOW CHART



FRONTEND FLOWCHART

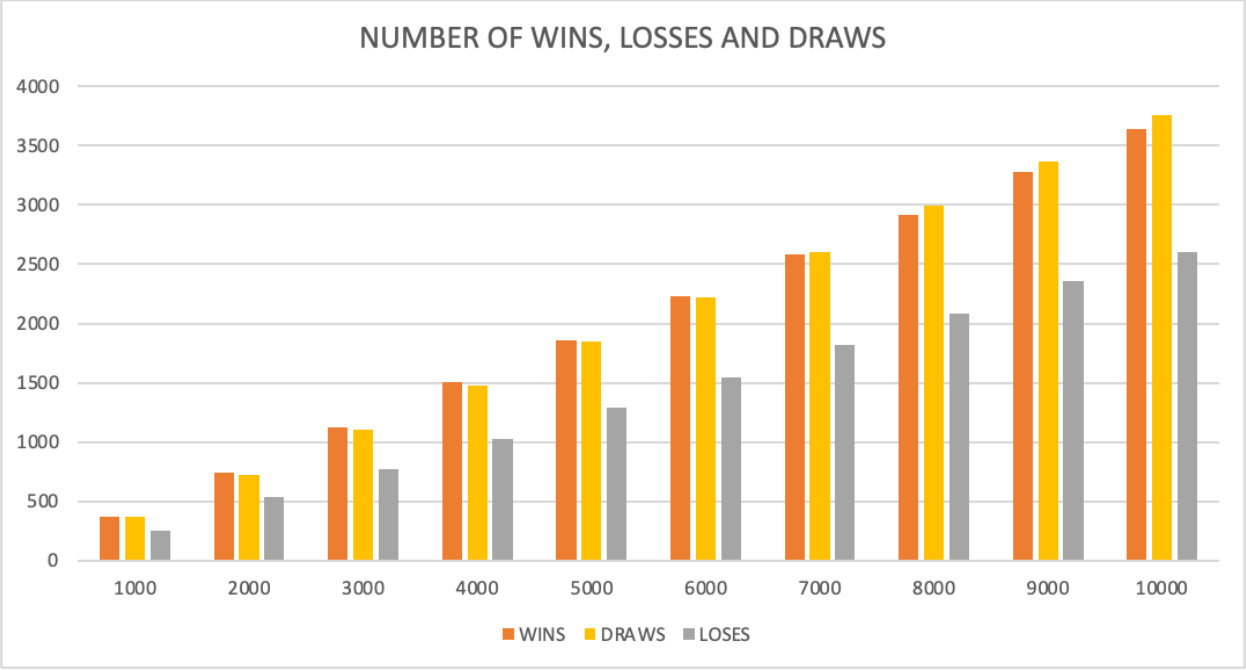


OBSERVATIONS & GRAPHICAL ANALYSIS

Graphs

The below graph depicts the number of WINS, LOSSES and DRAWS for 10,000 plays. It is observed that the number of draws and wins are more when compared to the losses.

Number of Rounds	WINS	LOSES	DRAWS
1000	373	254	373
2000	743	536	721
3000	1125	768	1107
4000	1502	1026	1472
5000	1857	1292	1851
6000	2229	1547	2224
7000	2581	1818	2601
8000	2920	2083	2997
9000	3278	2353	3369
10000	3635	2607	3758



Winning Combinations as defined in our project -

```
List<Integer> topRow = Arrays.asList(1, 2, 3);
List<Integer> midRow = Arrays.asList(4, 5, 6);
List<Integer> botRow = Arrays.asList(7, 8, 9);
List<Integer> leftCol = Arrays.asList(1, 4, 7);
List<Integer> midCol = Arrays.asList(2, 5, 8);
List<Integer> rightCol = Arrays.asList(3, 6, 9);
List<Integer> diag1 = Arrays.asList(1, 5, 9);
List<Integer> diag2 = Arrays.asList(3, 5, 7);
```

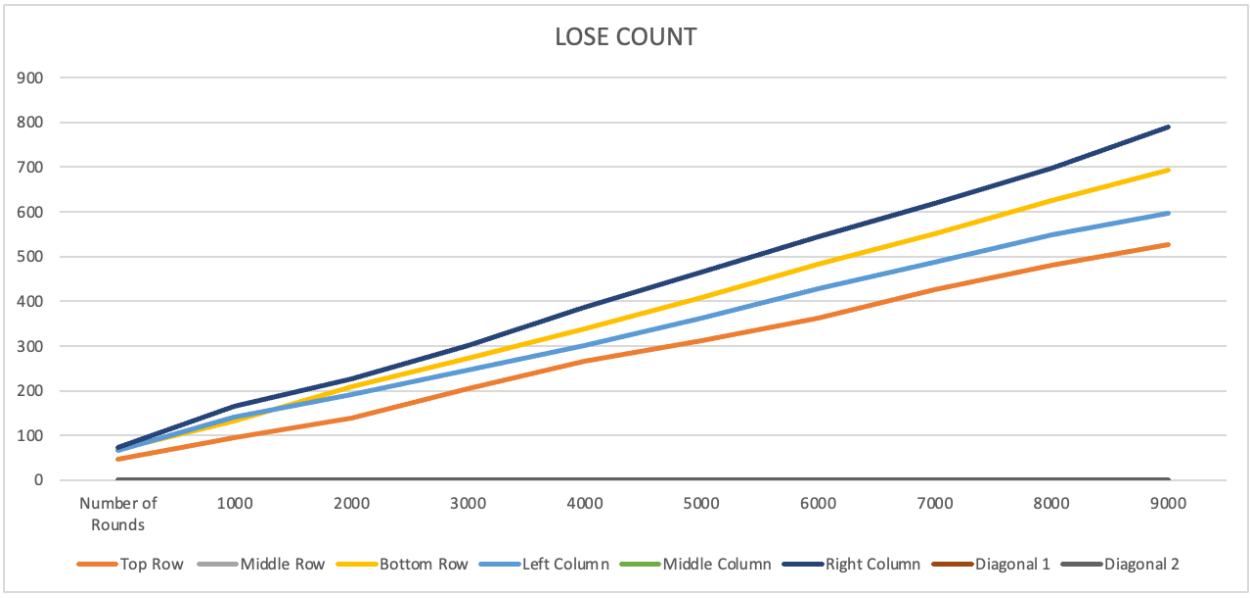
The below data represents the number of wins or losses per 1000 rounds for every above mentioned combination.

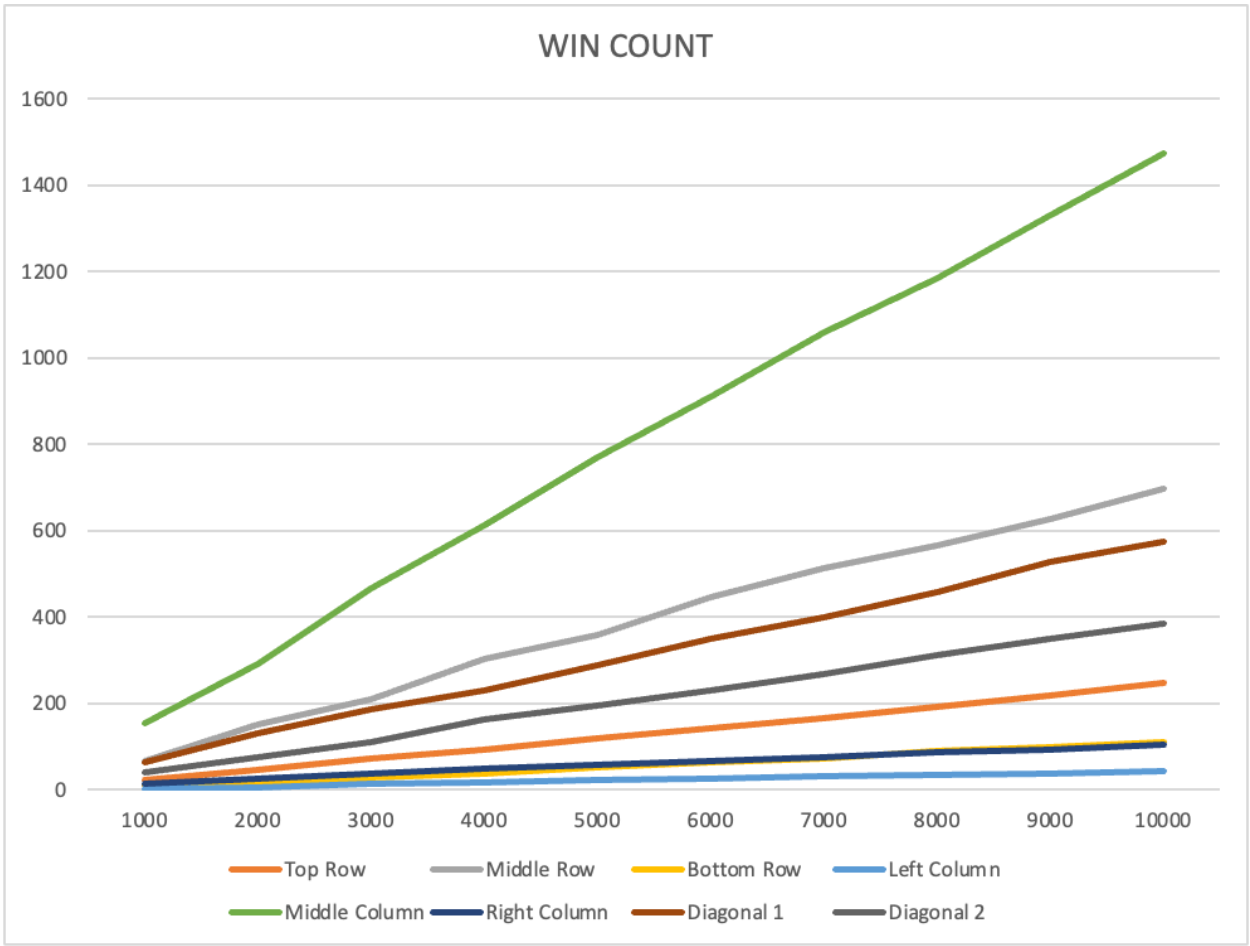
LOSE COUNT

Number of Rounds	Top Row	Middle Row	Bottom Row	Left Column	Middle Column	Right Column	Diagonal 1	Diagonal 2
1000	46	0	69	66	0	73	0	0
2000	95	0	133	142	0	166	0	0
3000	140	0	210	192	0	226	0	0
4000	205	0	273	247	0	301	0	0
5000	267	0	338	301	0	386	0	0
6000	312	0	408	362	0	465	0	0
7000	363	0	483	428	0	544	0	0
8000	426	0	551	488	0	618	0	0
9000	480	0	626	549	0	698	0	0
10000	526	0	694	597	0	790	0	0

WIN COUNT

Number of Rounds	Top Row	Middle Row	Bottom Row	Left Column	Middle Column	Right Column	Diagonal 1	Diagonal 2
1000	21	66	11	2	154	14	64	41
2000	47	151	19	4	292	25	129	76
3000	72	208	29	14	467	38	186	111
4000	93	304	37	17	611	50	229	161
5000	120	357	51	22	769	56	287	195
6000	143	445	62	26	910	65	348	230
7000	165	514	73	31	1058	75	398	267
8000	191	564	90	35	1185	87	456	312
9000	219	627	99	38	1329	92	526	348
10000	248	696	110	43	1474	104	575	385


LOSE COUNT
900
800
700
600
500
400
300
200
100
0
Number of Rounds
1000
2000
3000
4000
5000
6000
7000
8000
9000
Top Row
Middle Row
Bottom Row
Left Column
Middle Column
Right Column
Diagonal 1
Diagonal 2


WIN COUNT
1600
1400
1200
1000
800
600
400
200
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Top Row
Middle Row
Bottom Row
Left Column
Middle Column
Right Column
Diagonal 1
Diagonal 2

RESULTS & MATHEMATICAL ANALYSIS

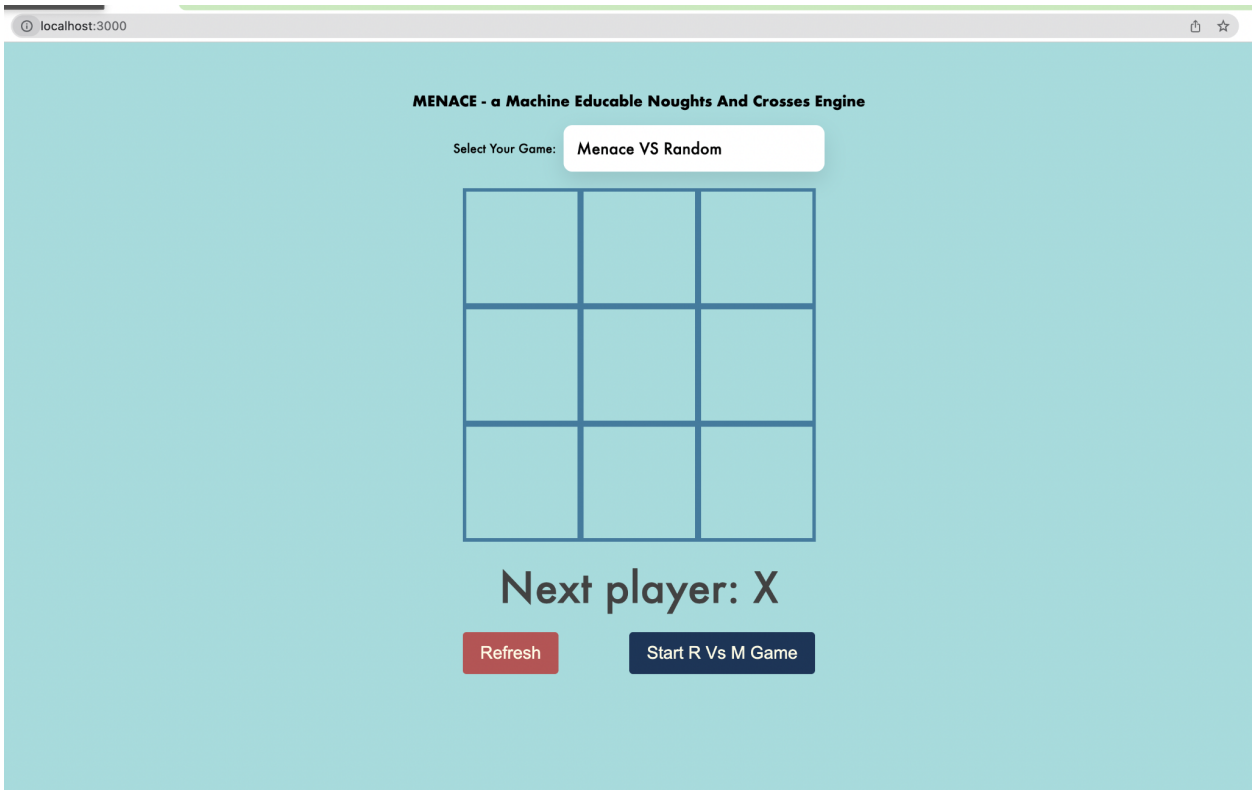
Probability function that defines the transition probability from M1 to M2 by taking the action M1*M2*a —>[0,1]. In the approach used, M2 depends on O's turn. Whenever Menace takes an action, The system return a new state after O's turn.

Every possible state has it's own probability which depends on every O's move.

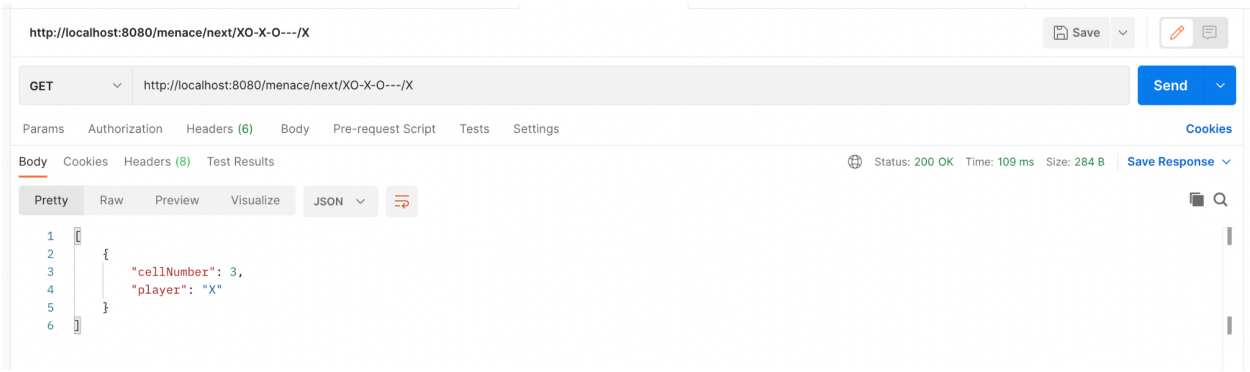
P(M2 = -O- - X - - - - |M1 = - - - - - - - - -, a = 5) = …=⅛

The goal of this project is to create a function that maps a state to the best action that Menace can take.

FRONTEND APPLICATION



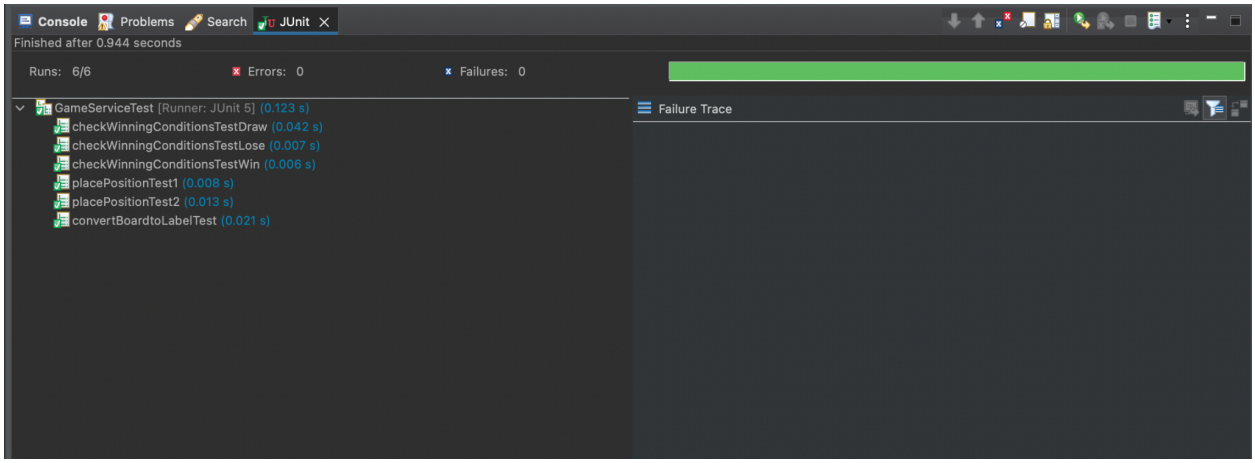
POSTMAN REQUEST TO GET RESPONSE ENTITY FROM THE CONTROLLER



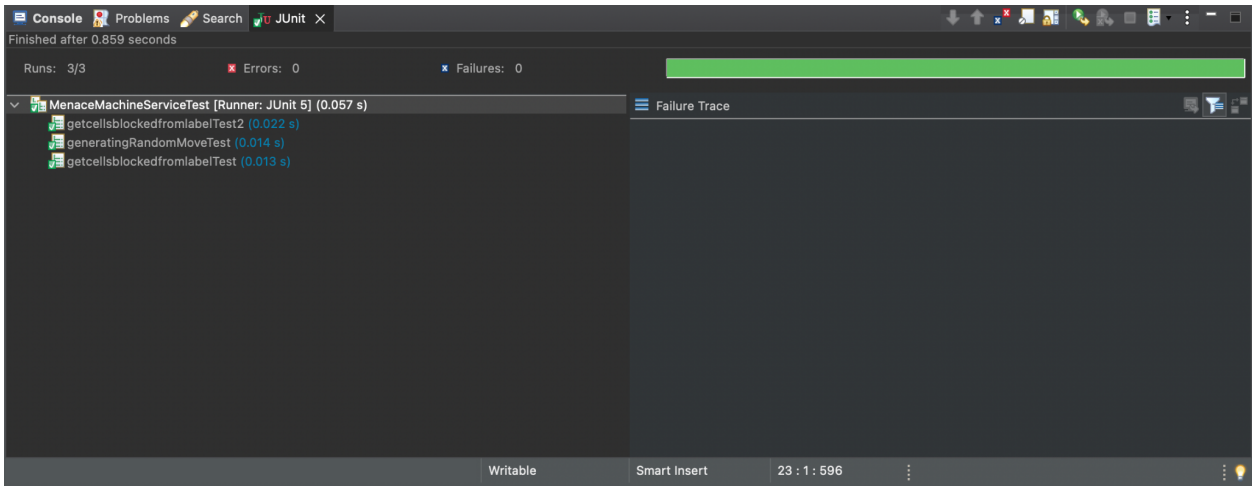
TEST CASES

We have created these unit test cases to check all the methods in this project.

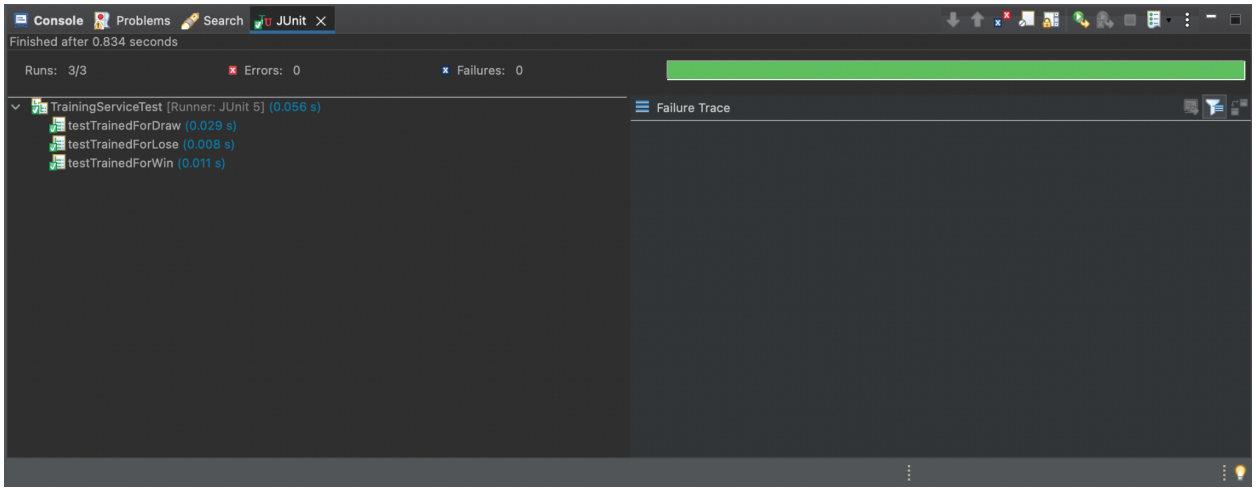
GameServiceTest



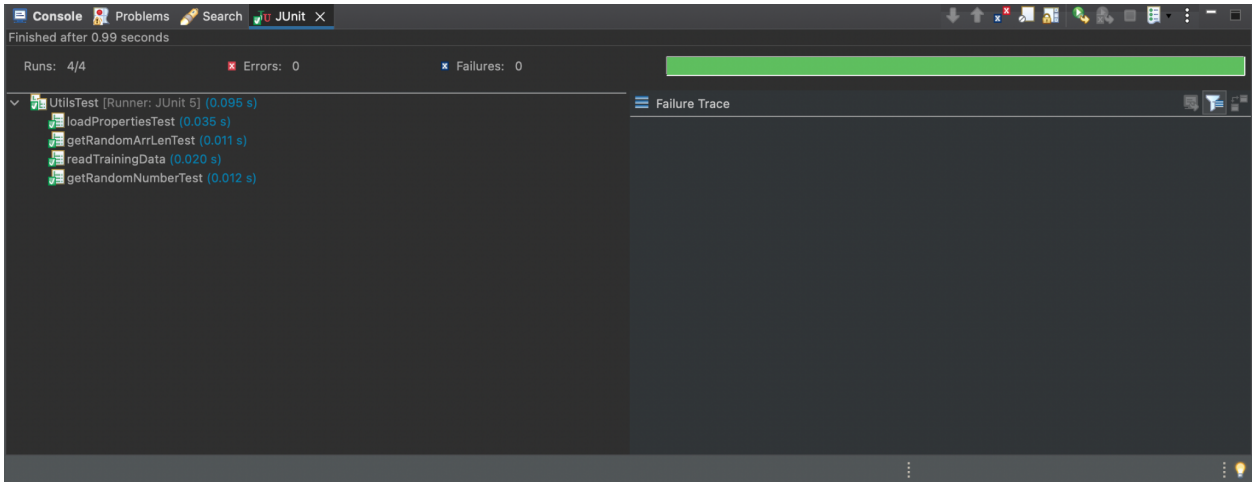
MenaceMachineServiceTest



TrainingServiceTest



UtilsTest



CONCLUSION

In this project we have trained the data by running multiple iterations to store the moves played by human/random and modifying the beads accordingly which lead to the result of Menace winning or drawing the majority of games by iterating through the layouts present in the training data.

REFERENCES

1. https://odsc.medium.com/how-300-matchboxes-learned-to-play-tic-tac-toe-using-menace-35e0e4c29fc
2. https://towardsdatascience.com/reinforcement-learning-and-deep-reinforcement-learning-with-tic-tac-toe-588d09c41dda
3. https://www.atarimagazines.com/v3n1/matchboxttt.html
4. https://chalkdustmagazine.com/features/menace-machine-educable-noughts-crosses-engine/
5. https://www.belloflostsouls.net/2019/03/teaching-304-matchboxes-to-beat-you-at-tic-tac-toe.html