Northeastern University
College of Engineering

The Menace

Spring 2022 INFO6205 Project

Team Members

Aishwarya Balyaya

Niul Panvalkar

Shubhang Shah

Instructors

Prof. Robin Hillyard

Date: 04/27/2022

Information Systems Department - Section 01

Introduction:

The design was named MENACE-Tic-Tac-Toe: A pile of matchboxes that contains a number of beads and learned to play tic-tac-toe. Tic-Tac-Toe Board is a 3X3 square shaped grid . It works a bit like a Neural Network. Randomly optimized at the beginning, but after a few subsequent games, it adjusts itself to favorable moves which later succeeds in each increasing further situation.

Aim:

Train the model such that it gradually improves after losing the game based on the probability of its every wrong move. The model will learn from the wrong move taken by it for losing the game so that it can tackle the situation and ensure the next winning game.

Approach:

- Players are given a chance to play to implement their moves
- Aim is to minimize the opponent's benefit being human here and maximize menace benefit. Hence, we used the Min-Max Algorithm here.
- On the basis of the moves taken by the human, the menace model will store the current state.
- After the current state is stored, the random bead selected will denote the next move for menace.
- Menace will be punished if he loses the game for the wrong move taken by it by reducing its move by 1.

- Menace will be rewarded with three extra moves for the move taken by it leading to the winning game.
- A draw will be considered as being positive for the model being trained and improved and moves will neither be reduced nor increased.
- Draw conclusions from observations gained from its moves and hence the model is trained.

Program:

Data Structures & Classes: The data structures and classes are described as follows:

Menace.py:

- train() - The aim of this function is to train menace with random human moves received from random human moves function. Function also keeps track of menace actions based on which we train our menace
- randomHumanMoves() - The aim of this function is to take consideration of empty index on board and take a random move based on that
- menaceTrain() - The aim of this function is to take consideration the list of actions the menace has take
    - If the menace wins the match, all the actions in the list will be rewarded with 3 points
    - If the match is a draw, all the actions in the list will be rewarded with 1 point. In case of draw, we have considered it as a good state
    - If the match was lost by menace, all the actions in the list will be rewarded with -1 so that the move will have a low score

```python
def randomHumanMoves(board: Ternary, player: int = 1) -> Tuple[Ternary, int]:
    board_list = list(board.number)
    actions = np.where(np.array(board_list) == "0")[0]
    action = np.random.choice(actions)
    board_list[action] = str(player)
    return Ternary("".join(board_list)), action


def train(num_rounds=100):
    for r in range(1, num_rounds + 1):
        board = Ternary("0" * 9)
        winner = -1
        turn = 1
        menace_actions = []

        while winner < 0:
            player = turn % 2 + 1
            if player == 2:
                board, action, symmetry_class, _ = menaceMove(board)
                menace_actions.append((action, symmetry_class))
            else:
                board, _ = randomHumanMoves(board)

            turn += 1

            winner = determineWinner(board)
            menaceTrain(menace_actions, winner)

        # Logging
        if r % (num_rounds // 10) == 0:
            action_histogram = {a: len(np.where(np.array(MENACE_MEMORY[0]["0" * 9]) == a)[0]) for a in range(9)}
            print(valuePlotting(Ternary("0" * 9), action_histogram, decimal=False))
```

```python
def menaceTrain(history: List, winner: int, player: int = 2) -> None:
    if len(history) == 5:
        history = history[:-1]

    # Game hasn't ended yet
    if winner < 0:
        return

    if winner not in [player, 0]:
        for action, symmetry_class in history:
            for _ in range(abs(REWARDS["lost"])):
                MENACE_MEMORY[player % 2][symmetry_class].remove(action)

        return

    result = "won" if winner == player else "draw"
    for action, symmetry_class in history:
        MENACE_MEMORY[player % 2][symmetry_class].extend([action] * REWARDS[result])

    return
```

Combinations.py:

- getAllPossibleBoards() - The aim of this function is to get all the possible combinations of the board

```python
def getAllPossibleBoards(player_start: int = 2, exclude_winners: bool = True) -> Tuple[List[List[str]], List[Set[str]]]:
    possibleBoards = [set(), set(), set(), set(), set(), set(), set(), set(), set()]
    boardClasses = [[], [], [], [], [], [], [], [], []]
    possibleBoards[0].add("000000000")
    boardClasses[0].append("000000000")

    for r in range(1, 10):
        player = int((r + player_start) % 2 + 1)
        for s in boardClasses[r - 1]:
            possible_moves = np.where(np.array(list(s)) == "0")[0]
            for a in possible_moves:
                temporaryBoard = list(s)
                temporaryBoard[int(a)] = str(player)
                board = "".join(temporaryBoard)

                # Game end
                if exclude_winners and determineWinner(Ternary(board)) >= 0:
                    continue

                # We allow winning boards, but exclude boards which ended round ago
                if not exclude_winners and determineWinner(Ternary(s)) >= 0:
                    continue

                # If board already exists
                if board in possibleBoards[r]:
                    continue

                boardSymmetries = getBoardSymmetries(board)
                possibleBoards[r].update(boardSymmetries)
                boardClasses[r].append(board)

    return boardClasses, possibleBoards
```

Plot.py:

The helper file that will format the string into a grid

- valuePlotting() - Plots the tic-tac-toe game with values to show probability

```python
def valuePlotting(game: Union[str, Ternary], dictonary: dict, decimal: bool = True) -> str:
    template = (
        " {0} | {1} | {2} \n"
        "-------+-------+------\n"
        " {3} | {4} | {5} \n"
        "-------+-------+------\n"
        " {6} | {7} | {8} \n"
    )

    if not isinstance(game, Ternary):
        board = gameToTernary(game)
    else:
        board = game

    values = []

    for k, a in enumerate(ternaryToXandO(board)):
        if a != " ":
            values.append(f"  {a}  ")
            continue
        if decimal:
            values.append("{0:.3f}".format(round(dictonary[k], 3)))
        else:
            values.append("  {0}  ".format(round(dictonary[k])))

    return template.format(*values)
```

MinMax.py:

```python
def minimax(board: Ternary, depth: int = 0, player: int = 2) -> Tuple[Ternary, int]:
    score, ended = getScore(board, depth, player)
    if score != 0 or ended:
        return board, score

    depth += 1
    scored_actions: Dict = {}
    actions = np.where(np.array(list(board.number)) == "0")[0]
    active_player = len(actions) % 2 + 1

    for a in actions:
        action_board = list(board.number)
        action_board[a] = str(active_player)
        new_board = "".join(action_board)
        _, scored_actions[new_board] = minimax(Ternary(new_board), depth, player)

    if player == active_player:
        max_value = max(scored_actions.values())
        max_boards = [b for b, v in scored_actions.items() if v == max_value]
        return Ternary(np.random.choice(max_boards)), max_value
    else:
        min_value = min(scored_actions.values())
        min_boards = [b for b, v in scored_actions.items() if v == min_value]
        return Ternary(np.random.choice(min_boards)), min_value
```

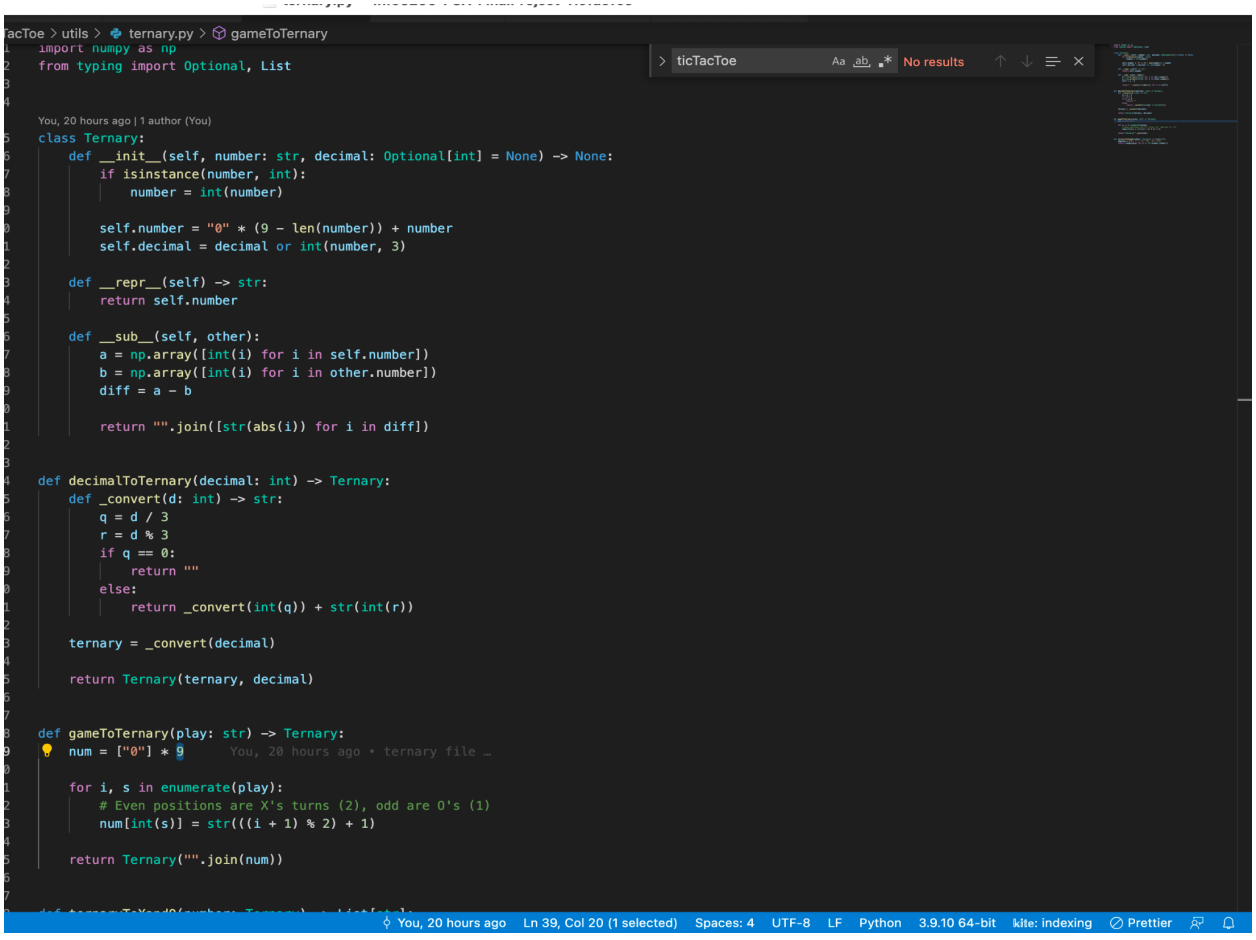
Ternary.py:

Puzzle are ternary numbers because numbers with base 3 are less common. In the tic-tac-toe game we have three states

- Empty state - also represented as 0
- “O” state - also represented as 1
- “X” state - also represented as 2

The aim of this file is to create empty, “X” and “O” state to ternary string. The functions are as follows

- decimalToTernary - Takes a decimal number into the function and change it to ternary
- gameToTernary - Takes the string of moves as input and change it to ternary string of length 9
- ternaryToXandO - The function change the state of empty, “X” and “O” state to ternary string

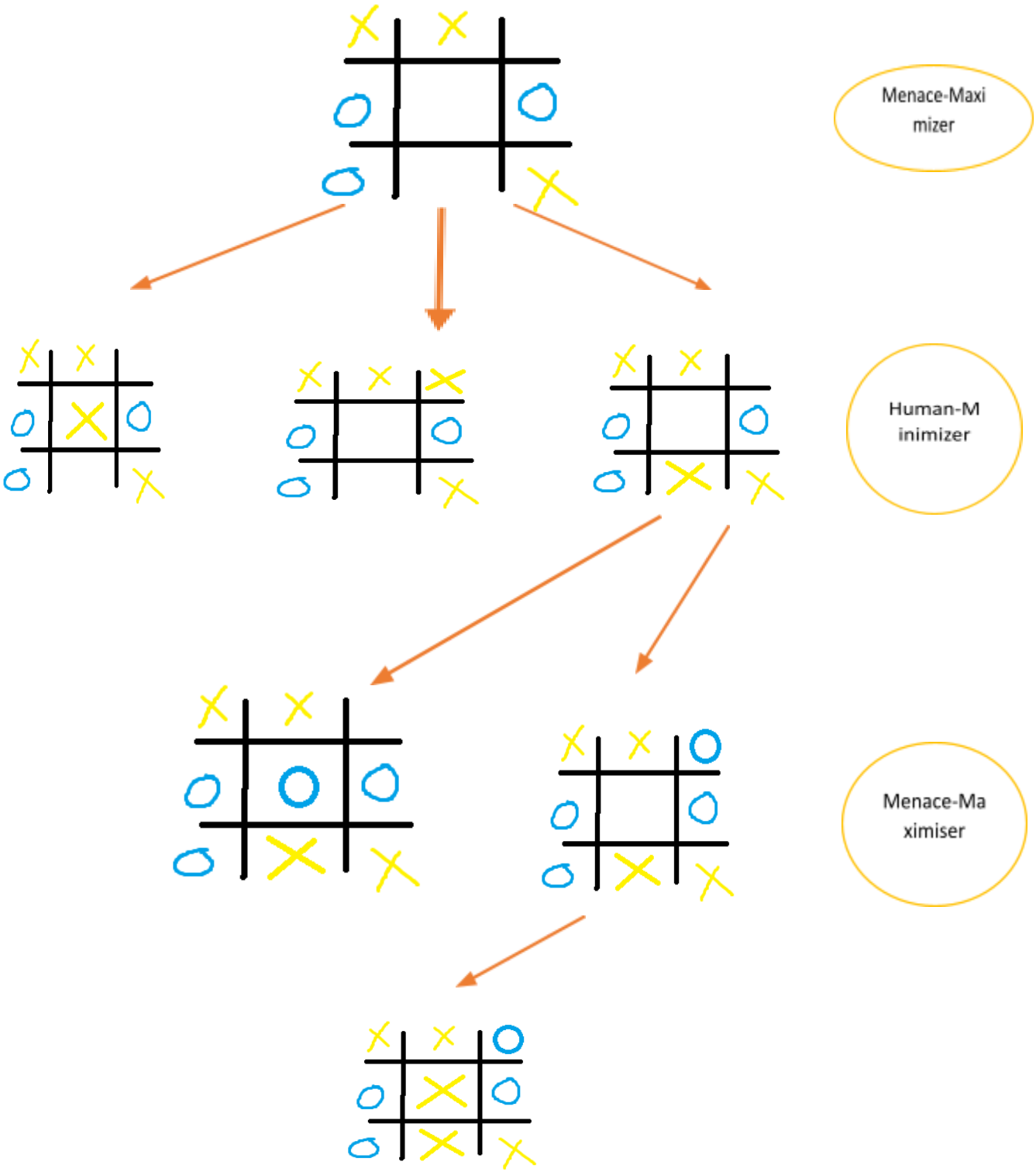


Algorithms :

1. Menace will be player 1 and Human will be player 2. Denoting all the fields by numbers 0-8. We are basically implementing Min-Max Algorithm which delivers an optimal action for the respective player, considering the opponent also to play finest.
2. The algorithm here uses depth-first search, to scan entire down the tree and explore the position depth in the board then backtracks using recursive calls
3. Menace will try to maximize its possible score and the human will try to minimize the possible score for Menace.
4. When the Menace has its turn, it tries to choose moves with the highest possible value. When in human turn, we consider the action to be with the minimal optimum value.
5. Checking the value for every possible move, If menace wins for the particular move, we assign a value of 10 – the distance between the current board and the winning board
6. If a menace loses the game for the particular move, we assign a value of the distance between the current board and the winning board – 10. This is how it will then backtrack.
7. When it proceeds all the way down to each state, it backtracks using recursive calls.

Assumption taken - Opponent which is human in our case will be a perfect and astute player aiming to win and play optimally, whereas human being the opponent is playing the best move he can! That's how Menace will be trained

This is how the Algorithm will look like:

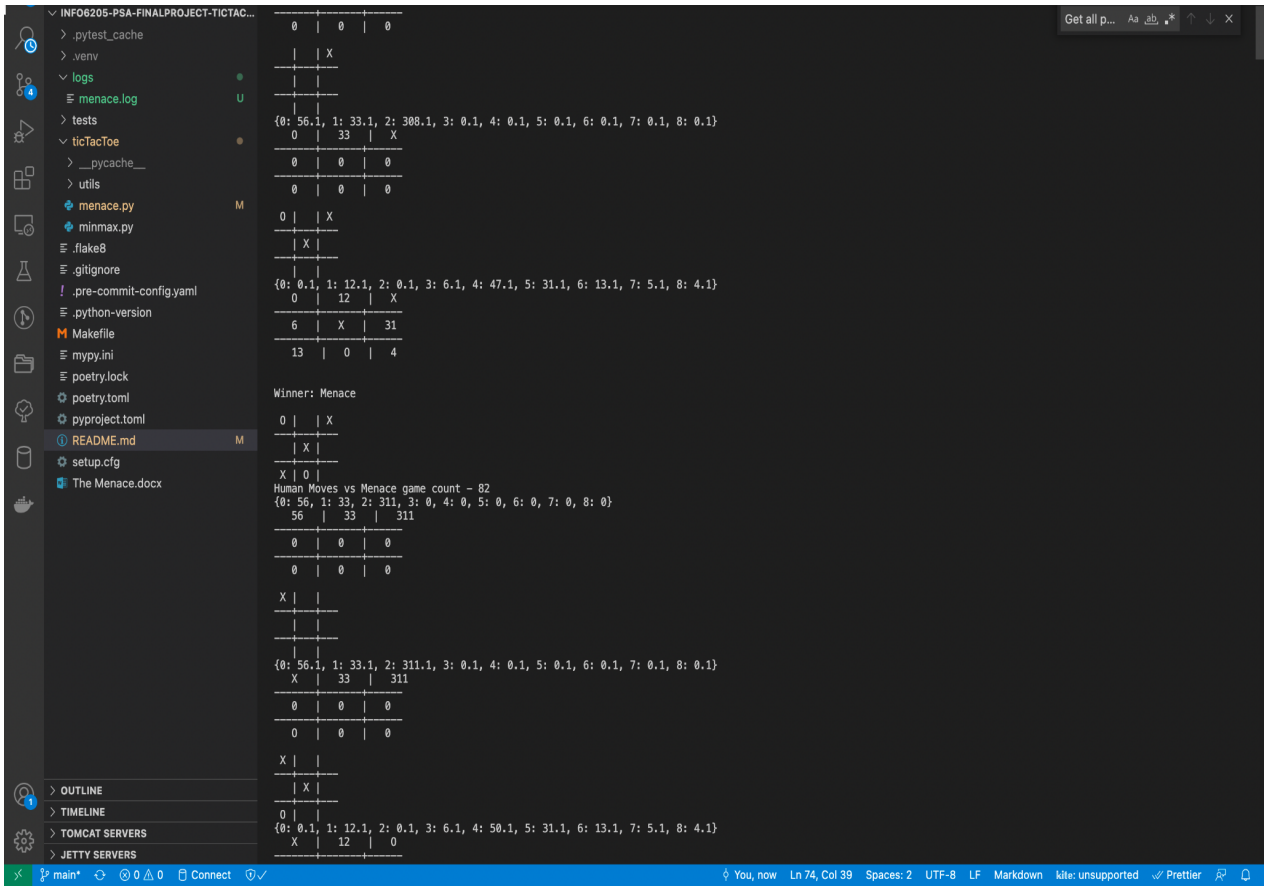


Invariants:

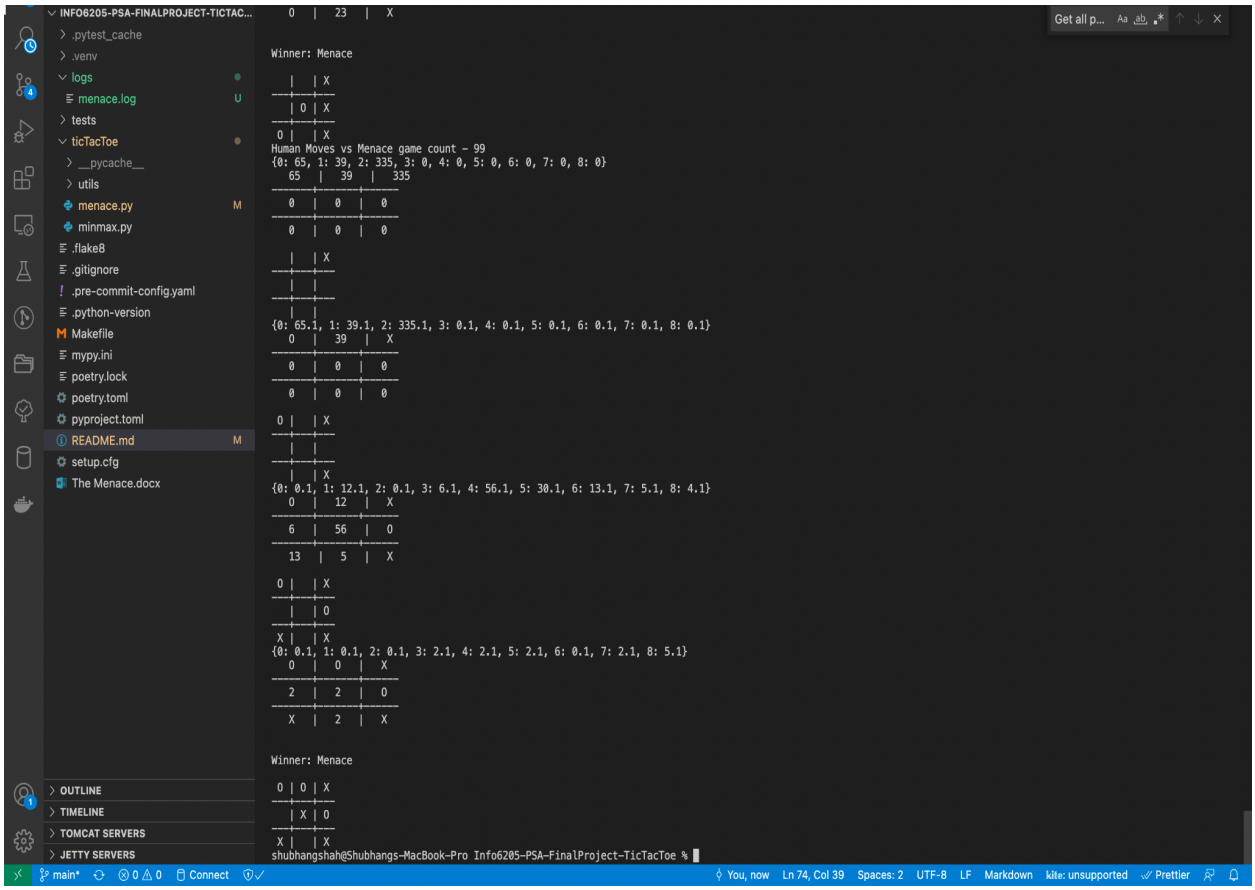
- Object here is invariant a Symmetry group
- Losing of Menace will have to assign the value of depth - 10
- Winning of Menace will have to assign 10 – a depth
- After each subsequent menace turn, humans will have their turn and so for and so forth

Command Line GUI Output with logging timestamp:

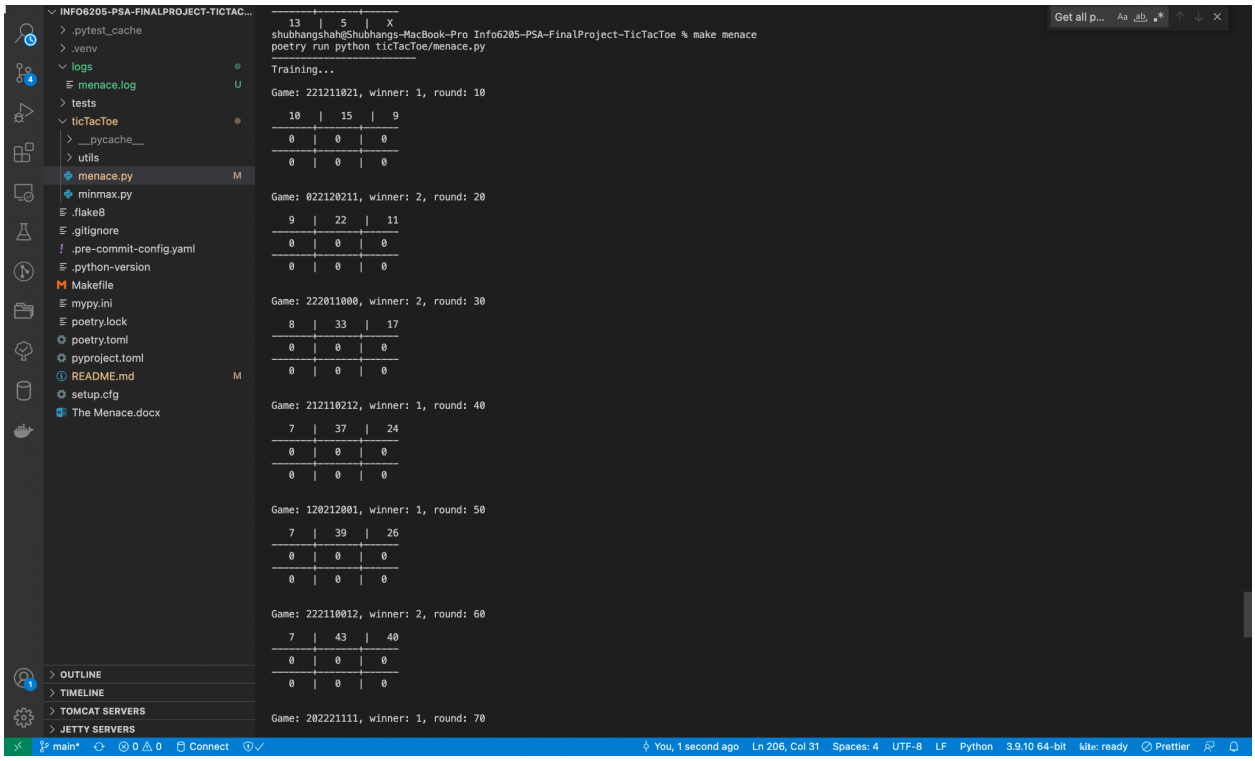
- Menace playing with random human moves



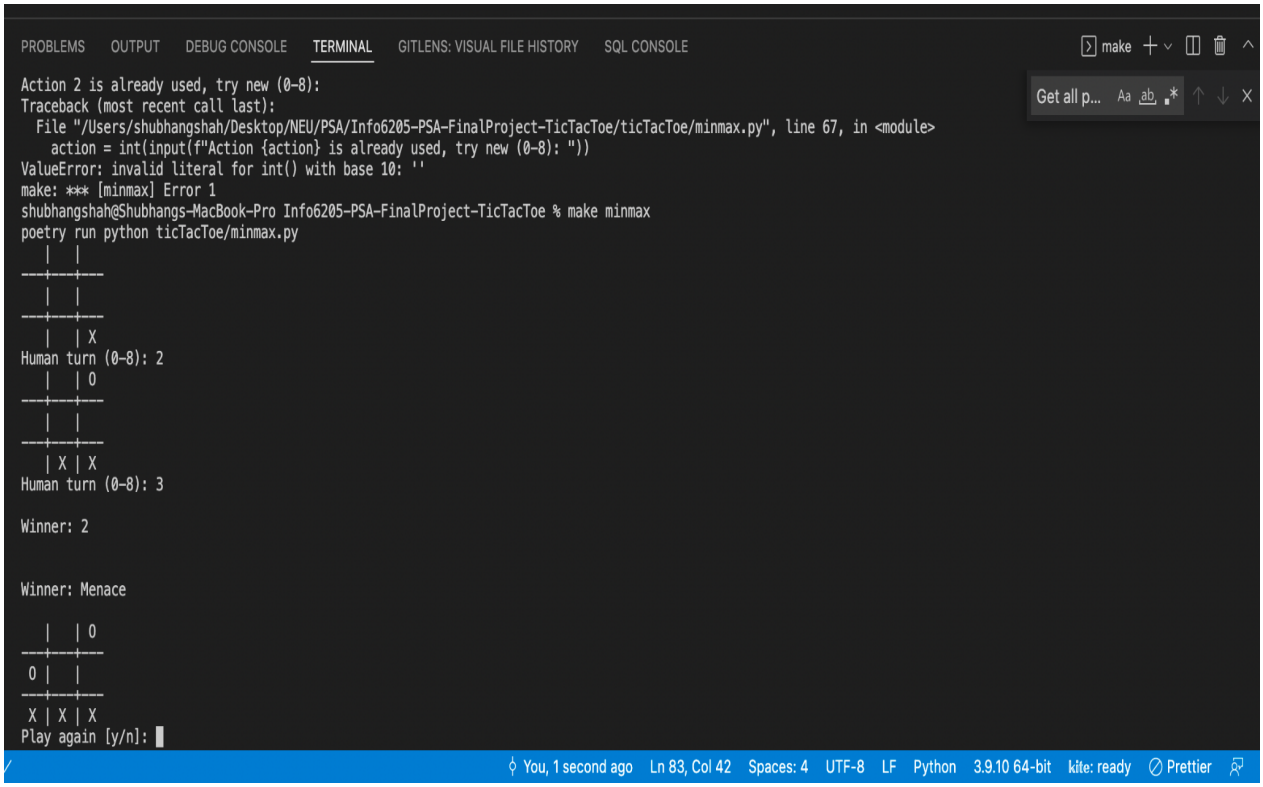
- Menace playing with random human moves - 100 times

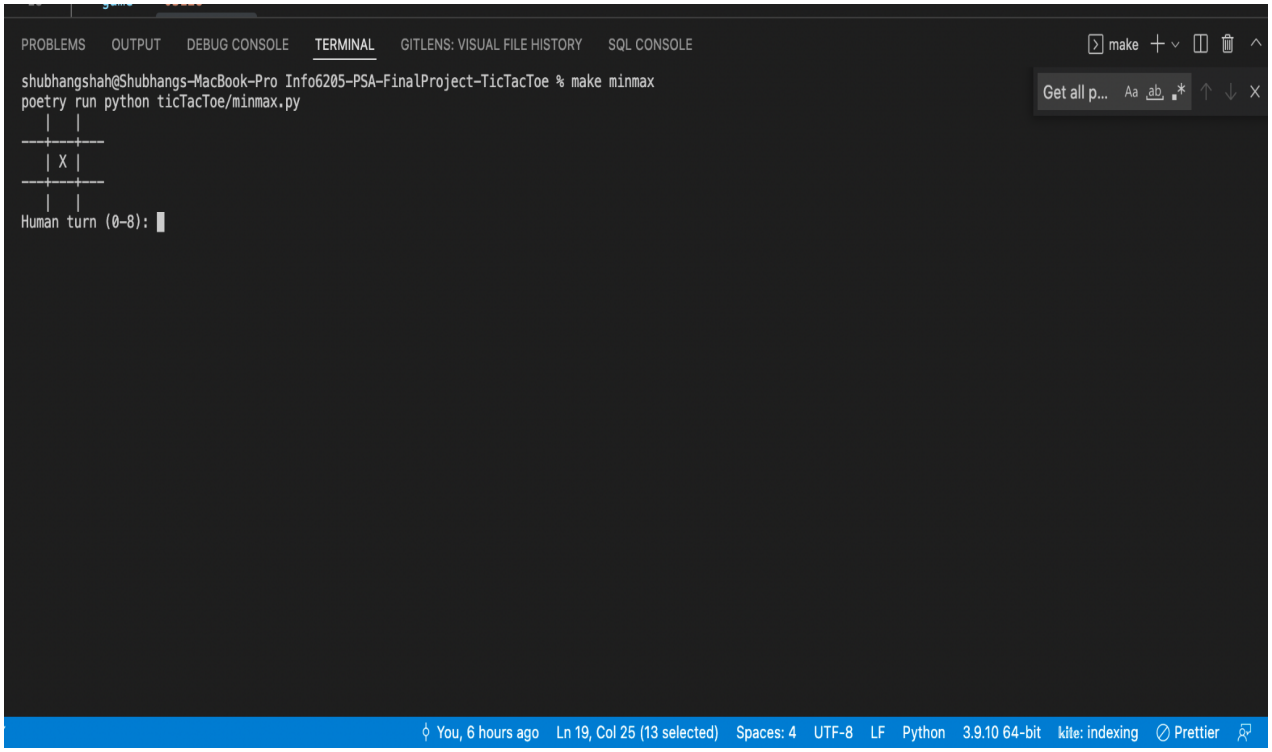


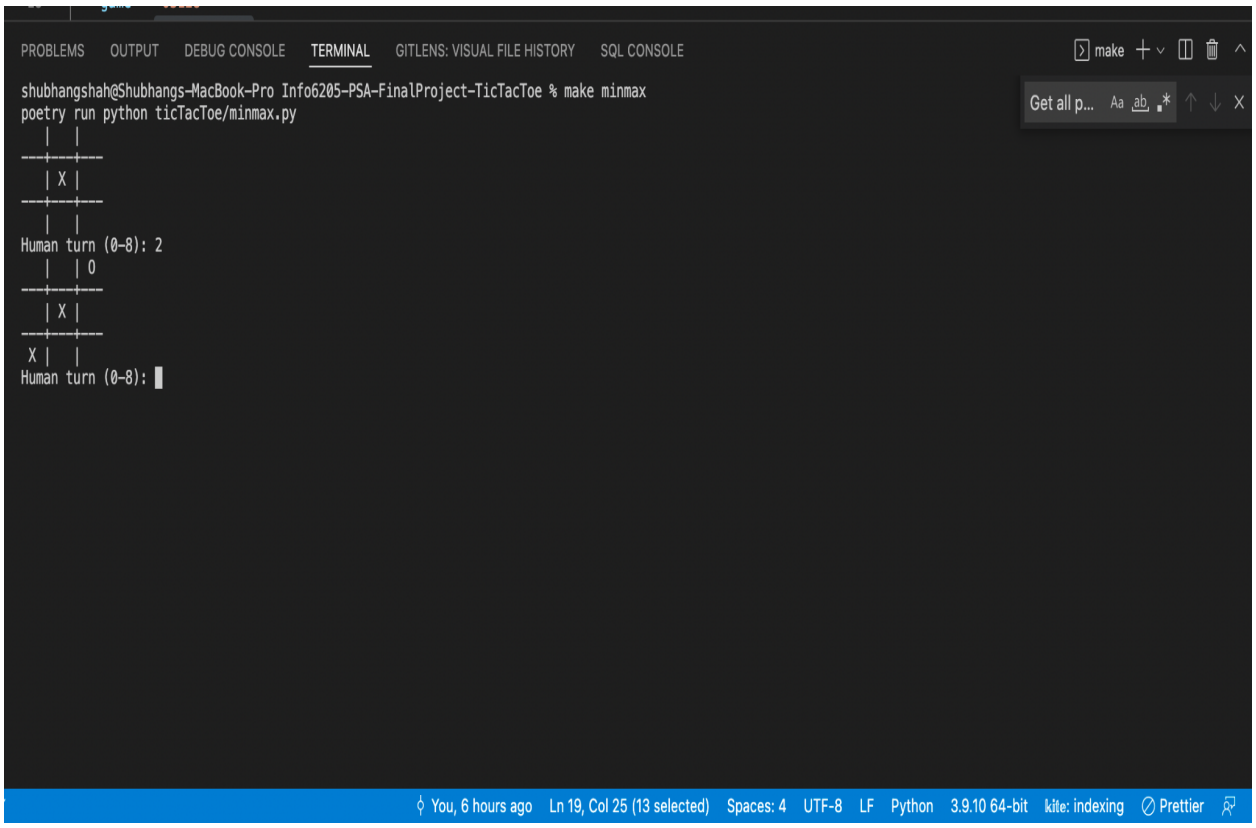
- Training of Menace



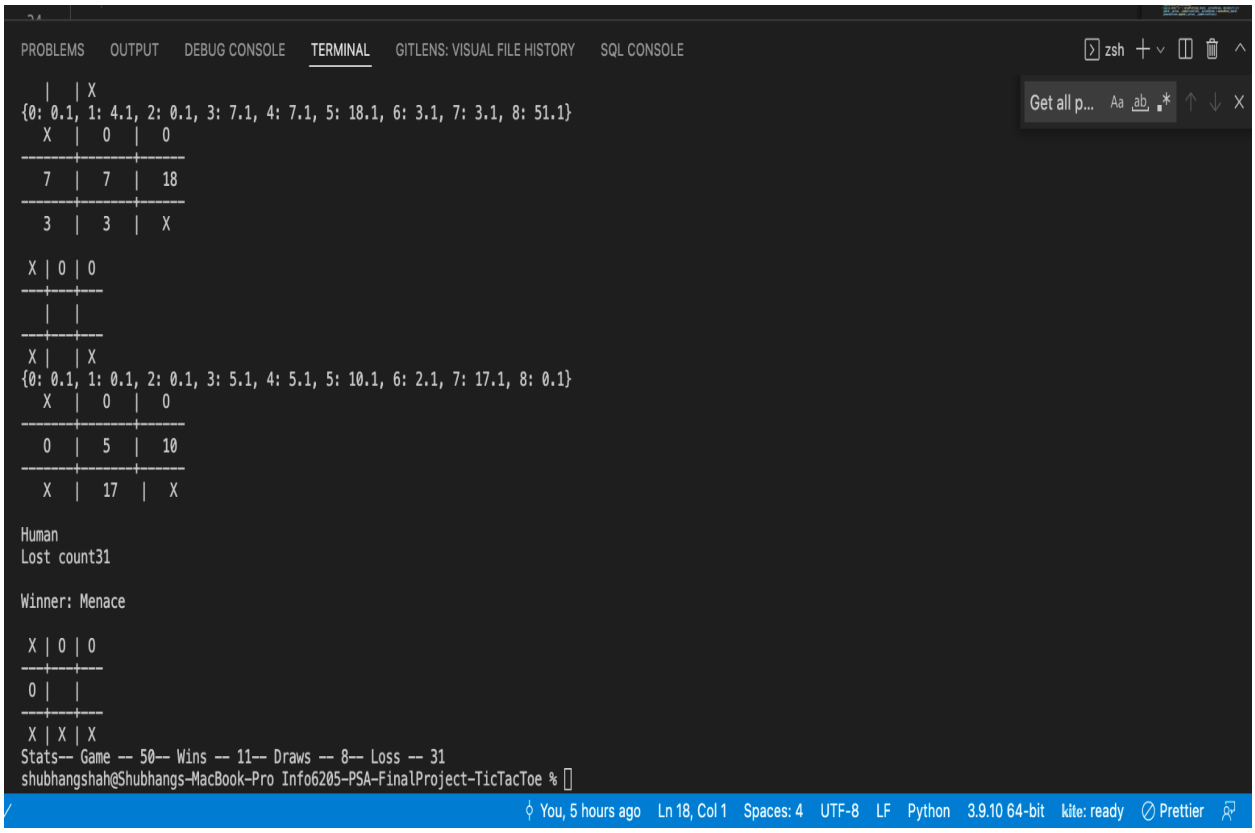
- Human playing with Menace



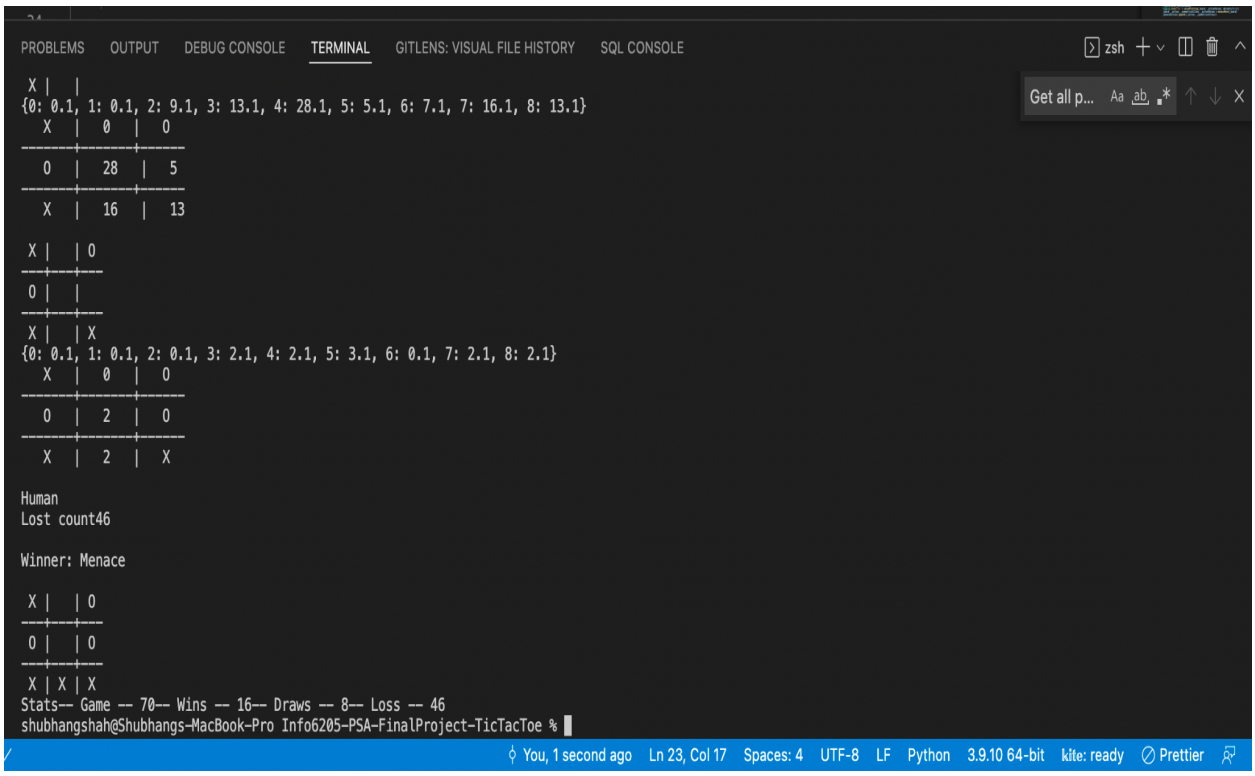

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
GITLENS: VISUAL FILE HISTORY
SQL CONSOLE
make
Get all p...
shubhangshah@Shubhangs-MacBook-Pro Info6205-PSA-FinalProject-TicTacToe % make minmax
poetry run python ticTacToe/minmax.py
| |
---+---+---
| X |
---+---+---
| |
Human turn (0-8):
You, 6 hours ago
Ln 19, Col 25 (13 selected)
Spaces: 4
UTF-8
LF
Python
3.9.10 64-bit
kite: indexing
Prettier


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
GITLENS: VISUAL FILE HISTORY
SQL CONSOLE
make
Get all p...
shubhangshah@Shubhangs-MacBook-Pro Info6205-PSA-FinalProject-TicTacToe % make minmax
poetry run python ticTacToe/minmax.py
| |
---+---+---
| X |
---+---+---
| |
Human turn (0-8): 2
| | O
---+---+---
| X |
---+---+---
X | |
Human turn (0-8):
You, 6 hours ago
Ln 19, Col 25 (13 selected)
Spaces: 4
UTF-8
LF
Python
3.9.10 64-bit
kite: indexing
Prettier

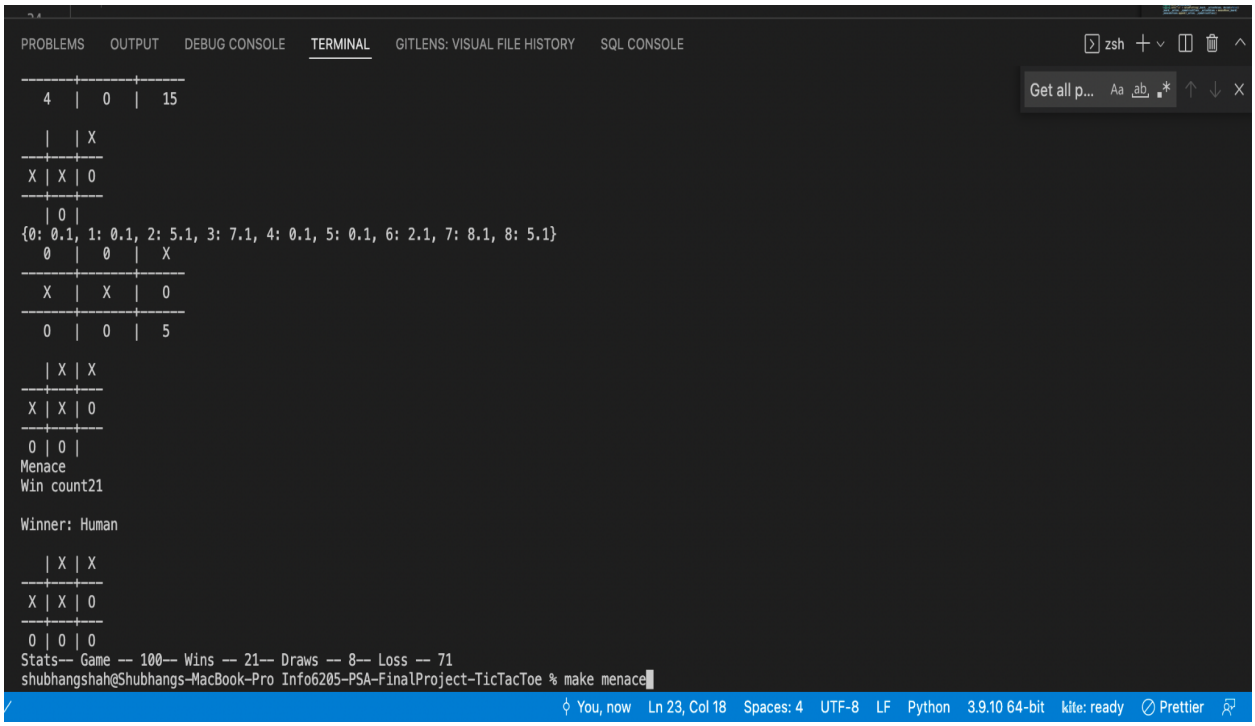
- Statistics of the game when ran 50 times



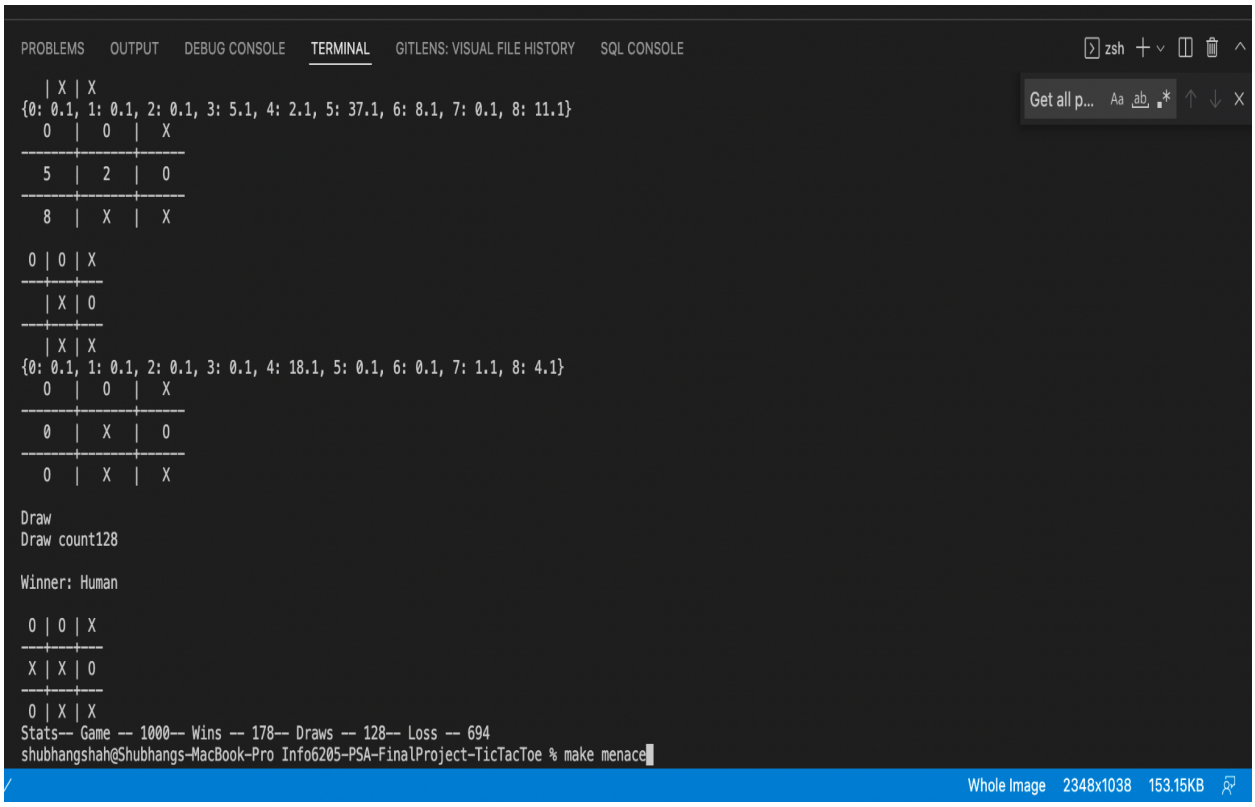
- Statistics of the game when ran 70 times



- Statistics of the game when ran 100 times

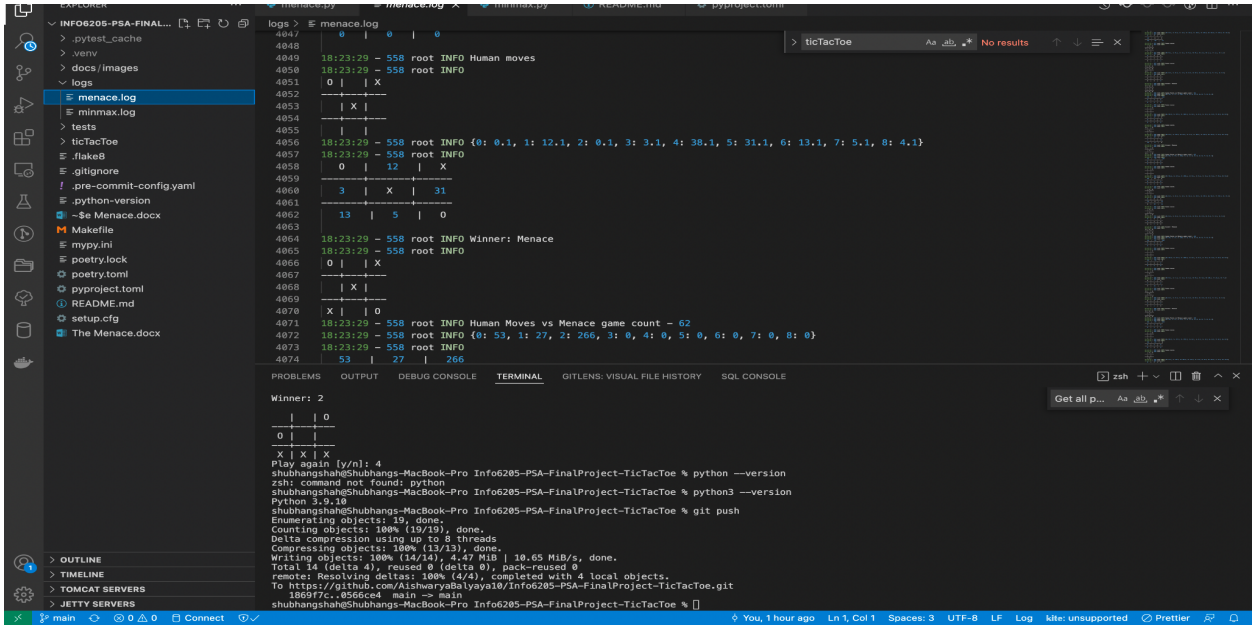


- Statistics of the game when ran 1000 times

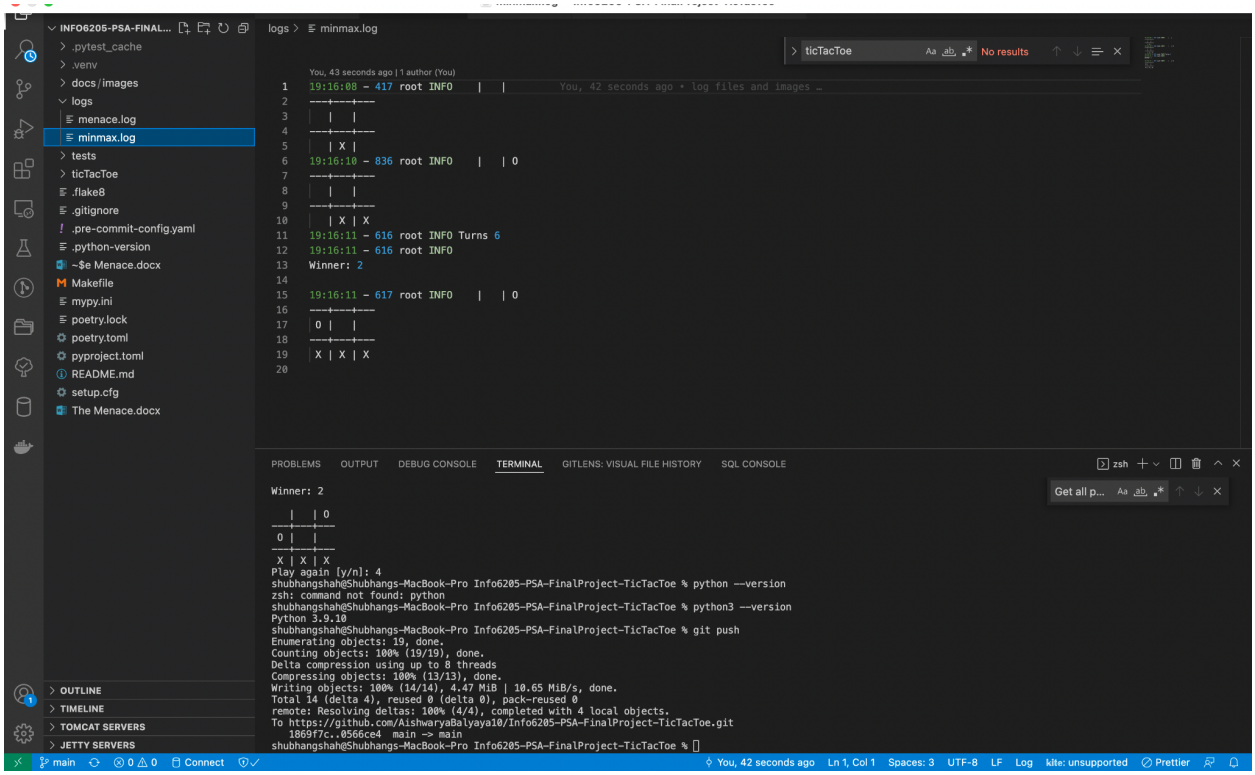


Logging Files:

- Menace.log



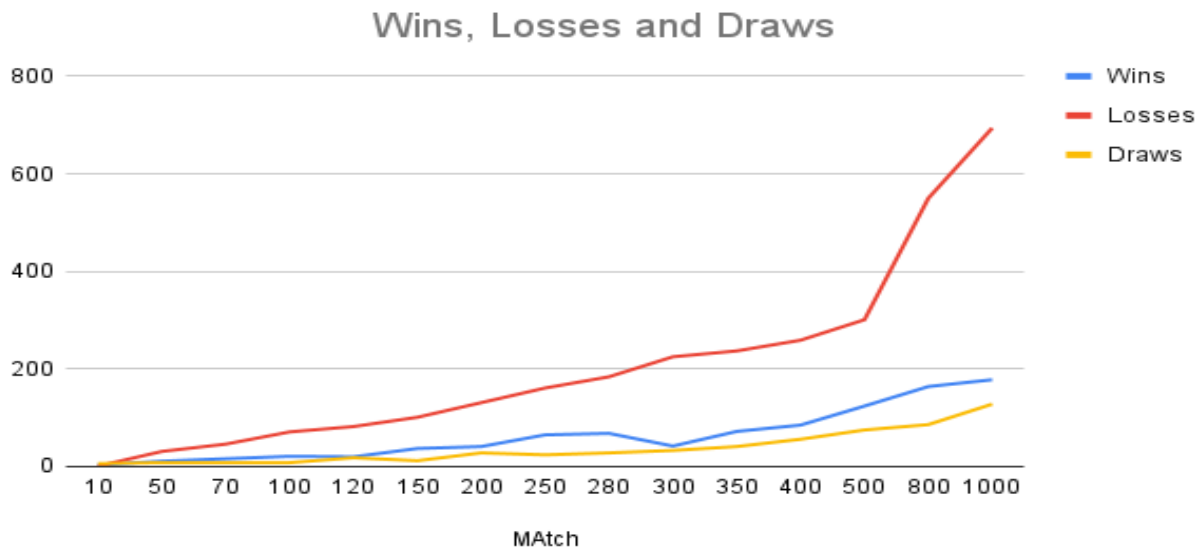
- Minmax.log



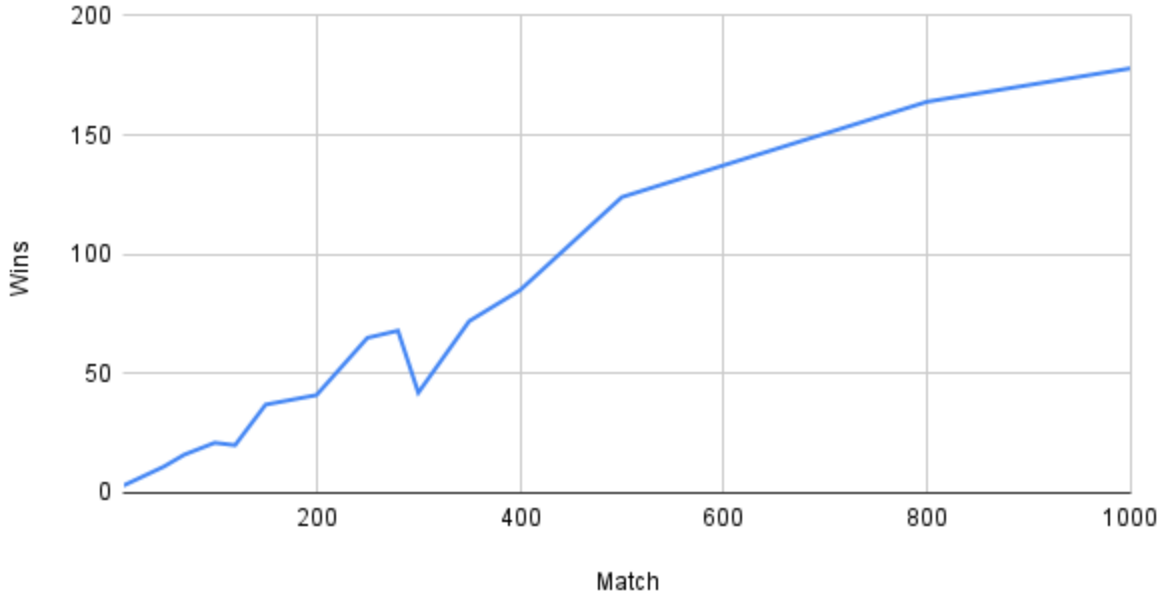
Menace vs Random Human Plays:

Match	Wins	Draws	Losses
10	3	6	1
50	11	8	31
70	16	8	46
100	21	8	71
120	20	18	82
150	37	12	101
200	41	28	131
250	65	24	161
280	68	28	184
300	42	33	225
350	72	41	237
400	85	56	259
500	124	75	301
800	164	86	550
1000	178	128	694

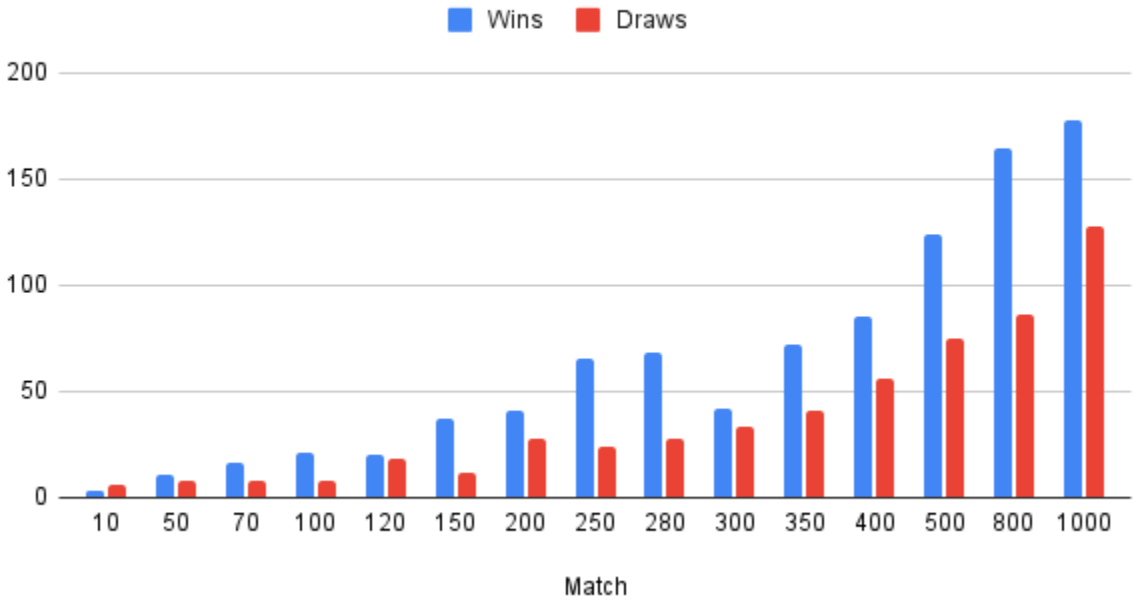
Graphical Analysis:



Wins vs. Match



Wins and Draws



Results & Mathematical Analysis:

As we can see from the above graph and excel we can conclude that the menace is gradually increasing the probability to win.

At the start of the game, a tree can cover all the possible outcomes. Once all the possible moves are stored, participants can take the best optimal move based on the tree. Thus, increasing the chance of wins and draws.

The upper bound for the number of positions will be $3^9 = 19638$, as a game can either have "O", "X" and blank positions. In a 3×3 grid the number of positions that is possible is $9! = 362880$. Thus to possibly, fill the whole grid with 5 "X" and 4 "0" the number of ways will be $9C5$ ($9\ choose\ 5$) = 126

In Tic-Tac-Toe, the shape of the matrix board is square hence more inclined towards the symmetry group of the square.

Dividing the square board matrix into 4 axis as t_x, t_y, t_{AC} & t_{BD} where AC & BD are diagonal axis of the board and we rotate the entire grid about *90° r, 180° r2 and 270° r3* where r is element. This will give us the same board.

Rotating the board 90° to the right, hence, 4 field maps back to 4, 0 to 0, 1 to 3, 2 to 6, 3 to 1, 5 to 7, 6 to 2, 7 to 5 and 8 to 8. This is how we get rotations using all possible elements' permutations.

Please see below to see all the possible rotations and reflections in our board. The cases covered are rotation to 0 (initial state), 90, 180, 270. Also, the reflections which are taken into consideration are the reflection on X axis, Y axis, AC axis and BD axis.

Element	Permutation	Rotation Cycle
e	(0, 1, 2, 3, 4, 5, 6, 7, 8)	0
r	(0, 3, 6, 1, 4, 7, 2, 5, 8)	90
r^2	(6, 3, 0, 7, 4, 1, 8, 5, 2)	180
r^3	(6, 7, 8, 3, 4, 5, 0, 1, 2)	270
t_x	(8, 7, 6, 5, 4, 3, 2, 1, 0)	Reflection on x axis
t_y	(8, 5, 2, 7, 4, 1, 6, 3, 0)	Reflection on y axis
t_{AC}	(2, 5, 8, 1, 4, 7, 0, 3, 6)	Reflection on AC axis
t_{BD}	(2, 1, 0, 5, 4, 3, 8, 7, 6)	Reflection on BD axis

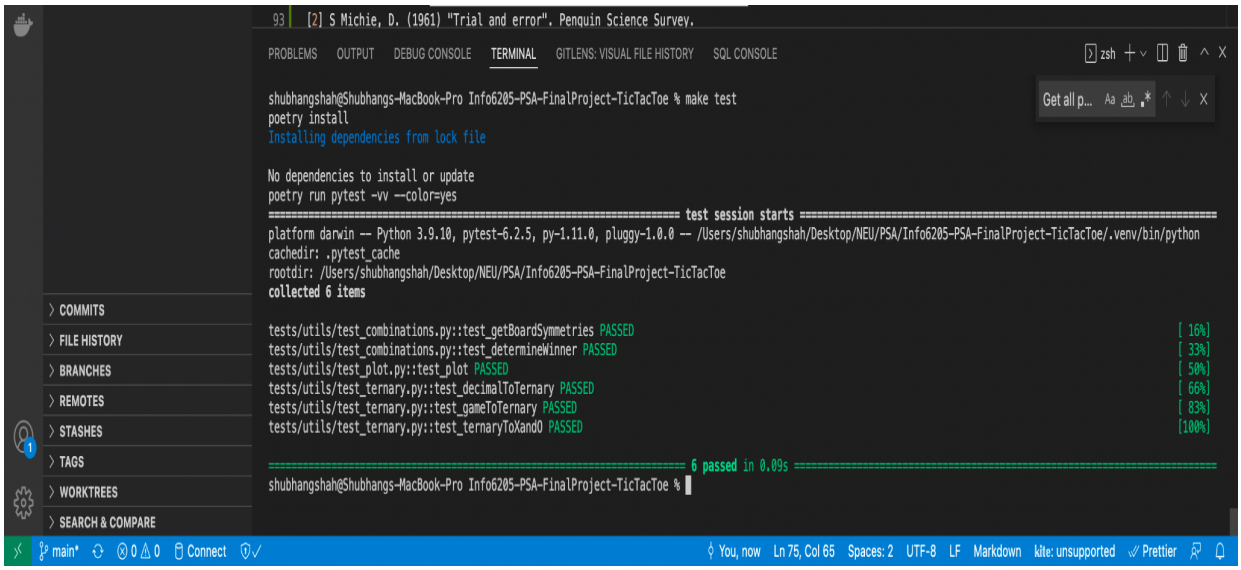
As seen from the above table we conclude that we chose 4 as the origin and hence that's why the rest of the numbers go clockwise all around.

Thus by Burnside's Lemma we can prove that the possible state after removing all the rotations and reflection will be 23. This is the case if we have 5 "X". In the similar way if we consider all the cases where the value of X and 0 is from 0 to 5 we will see that there are possible 304 unique states a tic-tac-toe graph can be in.

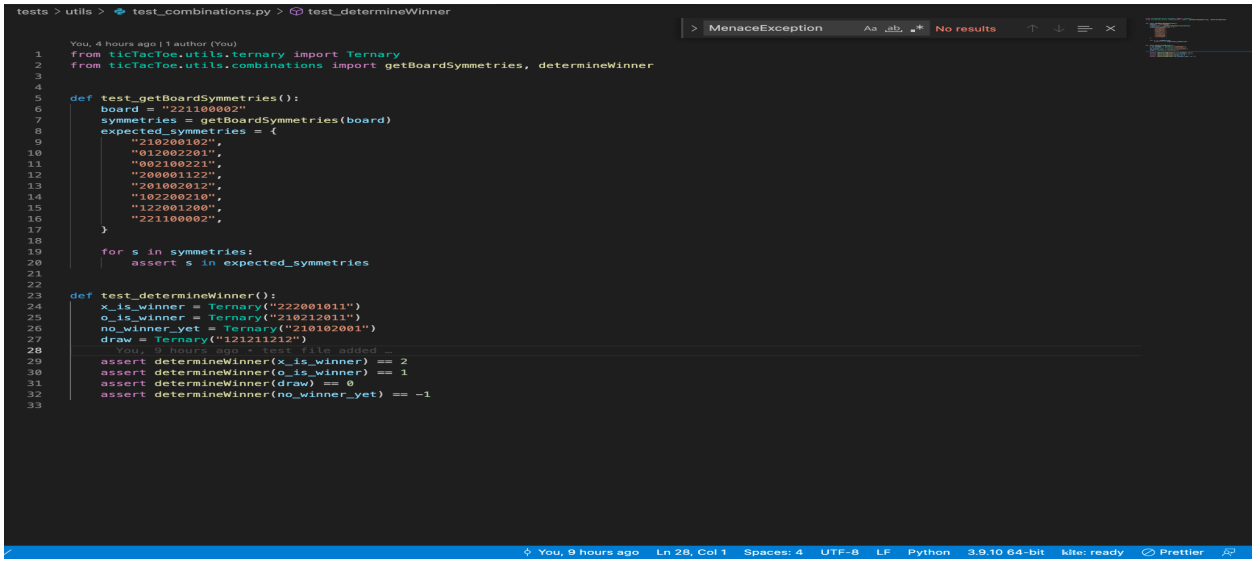
Thus, with the increase in the number of games we can see that the number of wins is also increased.

Test Cases:

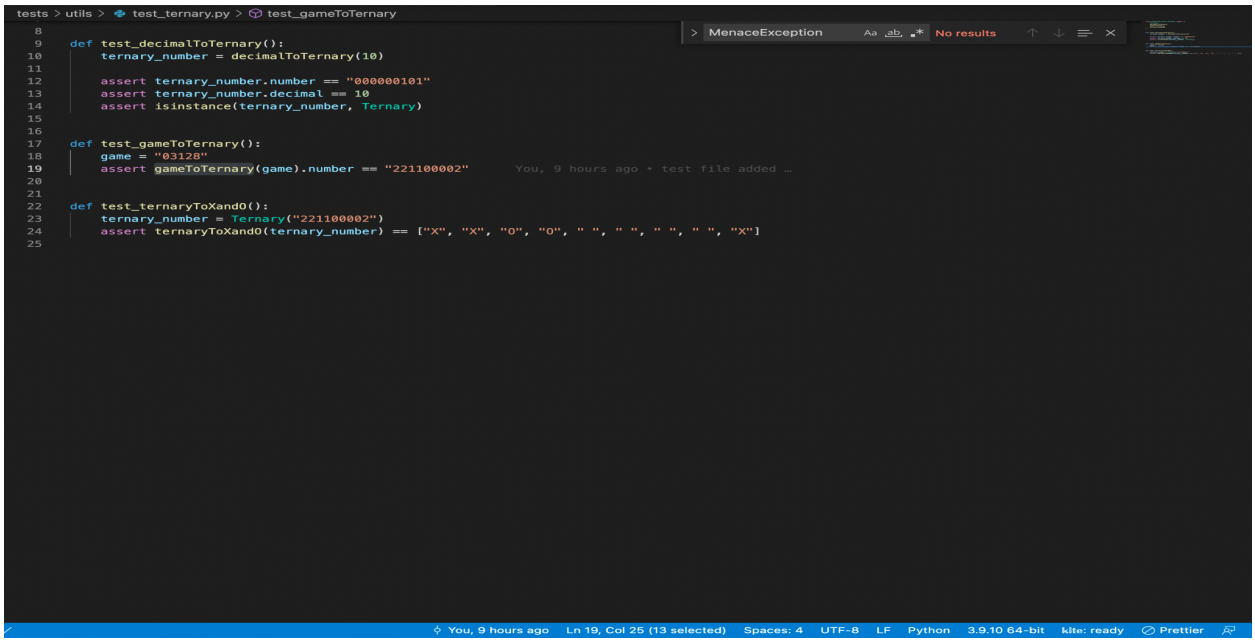
- Pass test cases



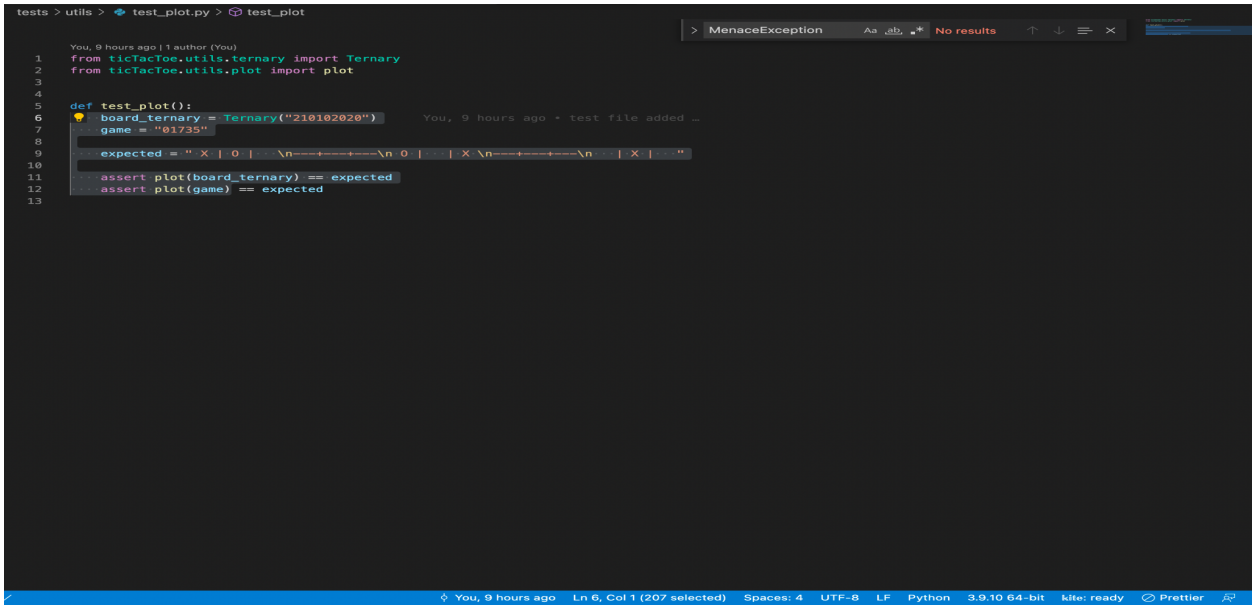
- Combinations test file



- Ternary test file



- Plot test file



Conclusions:

Score values of wins, loss and draws

- The values of win, loss and draw for Menace is 3, -1 and 1
- At the initial move when we do not have all the possible outcomes the chances of Menace losing the game is higher
- With the Menace playing more games we can see from the graphs that the win chances are also increasing

Importances of matches in Menace

- In the beginning of the game we can see that the tree has all the possible outcomes
- To come up with the best move the minmax will give us with the best possible outcome when the Menace has trained with many games
- Thus increasing the number of matches are increasing the chances of our wins and draws as shown in the graph and excel.

References:

- https://www.javatpoint.com/mini-max-algorithm-in-ai
- https://www.mscroggs.co.uk/blog/tags/menace
- http://www.se16.info/hgb/tictactoe.htm
- https://en.wikipedia.org/wiki/Burnside%27s_lemma
- https://levelup.gitconnected.com/minimax-algorithm-explanation-using-tic-tac-toe-game-22668694aa13
- https://en.wikipedia.org/wiki/Matchbox_Educable_Noughts_and_Crosses_Engine#Origin
- https://en.wikipedia.org/wiki/Burnside%27s_lemma