INFO 6205 Spring 2022 Project

The Menace

Team members:

Shashank Siripragada, Section 8, NUID: 002193773

Mayannk Kumaar, Section 8, NUID: 001537115

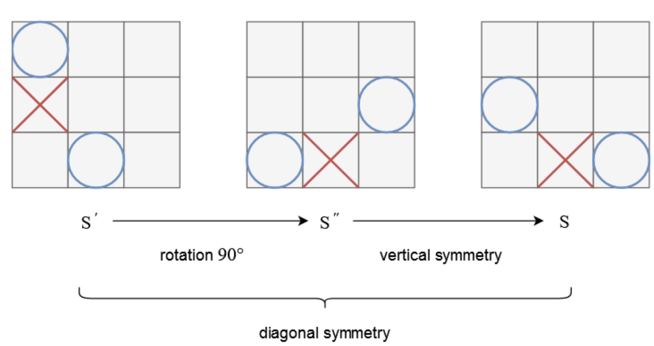
Introduction

Aim:

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

Approach:

- We generate all the possible states of the tic-tac-toe game. We try to reduce the states by checking if one state is equal to some other state after performing rotational steps (flips/ mirror/ rotate) to obtain unique states



- We add indices (beads) which are equally distributed at the beginning to all states achieved after a reduction so that every move is equiprobable
- We train the menace agent against the human agent so that it gets better when it plays a real human
- To train the agent we take several iterations, the probability of ideal moves played by a human agent as input from the user
- At first, the menace agent picks a random index (bead) from the state matching with the empty board state and places 'X'
- The human agent matches the current state with the sample states and picks the best move (index/ bead) given the probability mentioned by the user and places 'O'
- All the moves by menace agents are tracked along with the states which led to these moves.
- If the menace wins the game, we reward the menace by adding the picked index (bead) for each state to the maintained dictionary. Which in return increases the probability of playing that move if this state is achieved later making our chances of winning higher
- If the menace wins the game, we punish the menace by removing the picked index (bead) for each state from the maintained dictionary. Which in return decreases the probability of playing that move if this state is achieved later, we might not play this move and end up losing again
- If the game is drawn, we neither add nor remove indices from the dictionary

Strategy:

When training a menace agent with a human agent, we sample a random value, if it is greater than the given epsilon (1-probability) value the human agent chooses the optimal strategy i.e. picks the best possible move. Else, the human agent picks a random move from the list of possible moves.

Program

Data Structures:

- The **list** is used to store the game state
- **Dictionary** is used to hash the value of the state to possible moves (beads)
- A **tuple** is used to convert a state list to hashing key value in a dictionary
- **Set** is used to find unique possible moves

Functions:

We use the following functions to store and manipulate states.

vflip : Get a vertical flip of a state

rotate90 : rotate a given state by 90 degrees right

next_state : Obtain the next state given the current state and player

remove_duplicates : Remove duplicate states after checking possible flips and rotations

set_possible_moves : Given a state set all possible moves

get_required_action : Given an action, the number of rotations and flips get action after performing given rotations and flips

get_original_action : Given an action, the number of rotations and flips get the Original action

is_win : check for the winner given the board

match_two_states : validate whether two given states are equal or not

check_equal : validate two given states are equal or not after performing rotational steps (flip/mirror/ rotate)

```python
def vflip(state):
    '''
    Get Mirror image (flipped) of a state

    Returns: Flipped state
    '''

    flipped = list(state)

    flipped[2], flipped[5], flipped[8] = state[0], state[3], state[6]
    flipped[0], flipped[3], flipped[6] = state[2], state[5], state[8]

    return flipped

def rotate90(state):
    '''
    Rotate a given state by 90 degrees right

    Returns: Rotated state
    '''
    rotated = list(state)

    rotated[0], rotated[2], rotated[8], rotated[6] = state[6], state[0], state[2], state[8]
    rotated[1], rotated[5], rotated[7], rotated[3] = state[3], state[1], state[5], state[7]

    return rotated

def next_state(cur, next, player):
    '''
    Obtain next state given current state and player
    '''
    # print(cur, '\n', 'len of state', len(cur))
    for i in range(len(cur)):
        for j in range(9):
            next_s = []
            if cur[i][j] == 0:
                next_s = list(cur[i])
                next_s[j] = player
```

menace_humanplay.py | menace_selfplay.py | states.py | utils.py

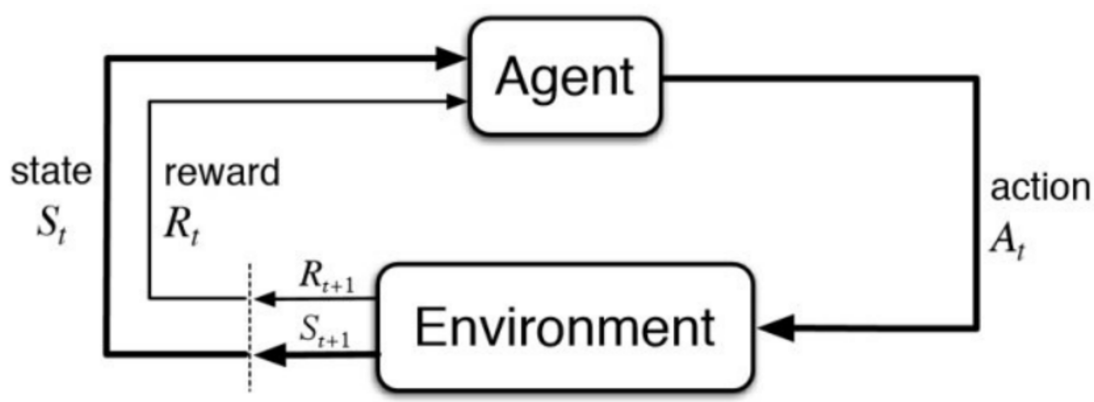
```python
def check_equal(state_1, state_2):
    '''
    Given two states check whether they are equal
    after all possible flips and rotations

    Returns : bool - indicating whether two states or equal
              nrotations - number of rotations to achieve equality
              nflips - number of flips to achieve equality
    '''

    nflips, nrotations = 0, 0
    s1, s2 = state_1, list(state_2)

    if match_two_states(s1, s2) : return True, nrotations, nflips

    ## one rotation
    nrotations += 1
    s2 = rotate90(s2)
    if match_two_states(s1, s2) : return True, nrotations, nflips

    ## two rotations
    nrotations += 1
    s2 = rotate90(s2)
    if match_two_states(s1, s2) : return True, nrotations, nflips

    ## three rotations
    nrotations += 1
    s2 = rotate90(s2)
    if match_two_states(s1, s2) : return True, nrotations, nflips

    ## Flip
    s2 = vflip(list(state_2))
    nflips += 1
    nrotations = 0
    if match_two_states(s1, s2) : return True, nrotations, nflips

    ## one rotation
    nrotations += 1
```

```python

def set_possible_moves(states, alpha=2):
    '''
    Given a state set all possible moves

    Returns : possible_moves (hashmap): possible moves for
              each state in states
    '''
    n = len(states)
    possible_moves = {}

    for i in range(n):
        actions = []
        for j in range(len(states[i])):
            if states[i][j] == 0:
                for alp in range(alpha):
                    actions.append(j)
                    #actions.append(j)

        possible_moves[tuple(states[i])] = actions

    return possible_moves

```

Above are the images of the selected functions to manipulate and store state information.

Algorithm:

Ideally, 2 player zero-sum games (Tic-Tac-Toe, Chess, Go, etc.) use a min-max algorithm, in which it keeps on playing and exploring all states with outcomes win/ lose/ draw and selects the path with the maximum reward. But here we will be using reinforcement learning where our menace agent will be trained for a certain number of iterations against a human-like agent with probability as input from a user. If the probability is not given we take the default probability of human agent optimal moves as 0.7 (1 – epsilon{0.3})



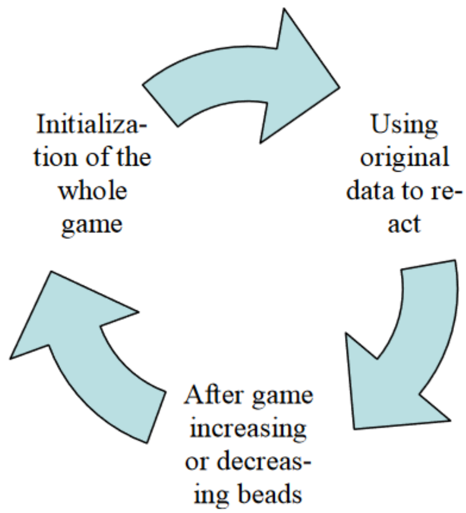


Figure 3. Process of the game

Invariants:

- The state of the board (S) would remain unchanged by any rotation or symmetry
- At any given point in the game, the difference between the number of ‘X’ & ‘O’ on board will never be greater than 1.

Logs

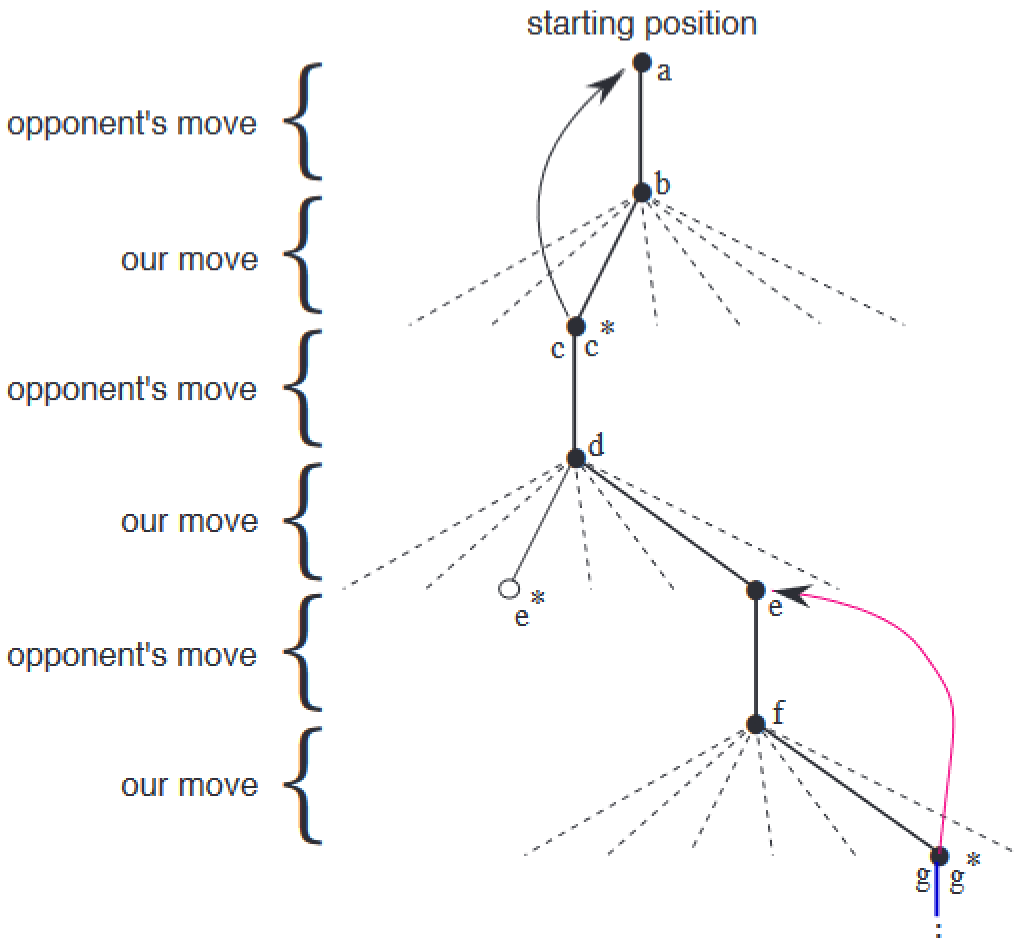
Game Log: The game log contains information regarding the initialization of the states and also state reduction information.

```
2022-04-27 20:40:00,923 INFO No of first possible move states: 3
2022-04-27 20:40:00,923 INFO #######################################################
2022-04-27 20:40:00,923 INFO No. of next possible move states: 24
2022-04-27 20:40:00,923 INFO Reduced no. of states after eliminating duplicates: 12
2022-04-27 20:40:00,923 INFO #######################################################
2022-04-27 20:40:00,923 INFO No. of next possible move states: 84
2022-04-27 20:40:00,955 INFO Reduced no. of states after eliminating duplicates: 38
2022-04-27 20:40:00,955 INFO #######################################################
2022-04-27 20:40:00,955 INFO No. of next possible move states: 228
2022-04-27 20:40:01,111 INFO Reduced no. of states after eliminating duplicates: 108
2022-04-27 20:40:01,111 INFO #######################################################
2022-04-27 20:40:01,111 INFO No. of next possible move states: 540
2022-04-27 20:40:02,023 INFO Reduced no. of states after eliminating duplicates: 174
2022-04-27 20:40:02,023 INFO #######################################################
2022-04-27 20:40:02,023 INFO No. of next possible move states: 696
2022-04-27 20:40:03,545 INFO Reduced no. of states after eliminating duplicates: 228
2022-04-27 20:40:03,545 INFO #######################################################
2022-04-27 20:40:03,545 INFO No. of next possible move states: 684
2022-04-27 20:40:05,036 INFO Reduced no. of states after eliminating duplicates: 174
2022-04-27 20:40:05,036 INFO #######################################################
2022-04-27 20:40:05,036 INFO No. of next possible move states: 348
2022-04-27 20:40:05,427 INFO Reduced no. of states after eliminating duplicates: 89
2022-04-27 20:40:05,427 INFO #######################################################
2022-04-27 20:40:05,427 INFO No. of next possible move states: 89
2022-04-27 20:40:05,443 INFO Reduced no. of states after eliminating duplicates: 23
2022-04-27 20:40:05,458 INFO #######################################################
```

Training Log: The training log contains information on training runs when the MENACE agent plays against Human-agent where MENACE_1 is the menace agent and MENACE_2 is a human agent. We log the two opponents, the epsilon value, and the outcome of the game.

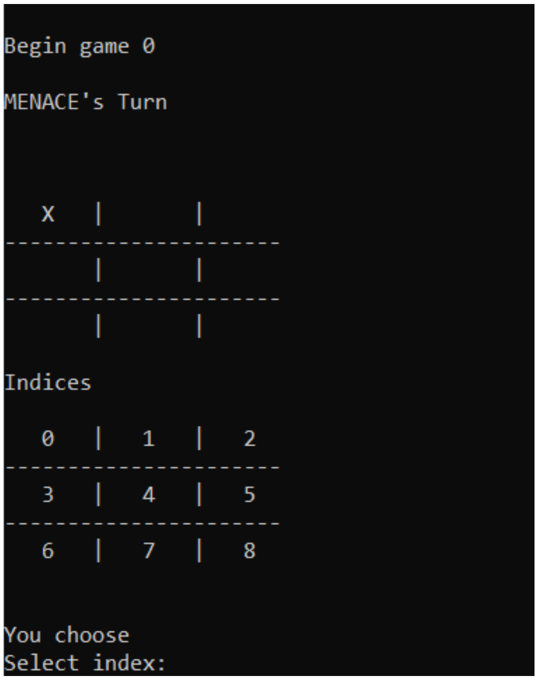
```
2022-04-27 19:27:12,182 INFO Training Game 0 Time: 2022-04-27 19:27:12.182141
2022-04-27 19:27:12,182 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,197 INFO Won: MENACE_1
2022-04-27 19:27:12,226 INFO Training Game 1 Time: 2022-04-27 19:27:12.226488
2022-04-27 19:27:12,226 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,232 INFO Draw
2022-04-27 19:27:12,232 INFO Training Game 2 Time: 2022-04-27 19:27:12.232591
2022-04-27 19:27:12,232 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,263 INFO Draw
2022-04-27 19:27:12,263 INFO Training Game 3 Time: 2022-04-27 19:27:12.263852
2022-04-27 19:27:12,263 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,263 INFO Won: MENACE_1
2022-04-27 19:27:12,297 INFO Training Game 4 Time: 2022-04-27 19:27:12.297534
2022-04-27 19:27:12,297 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,318 INFO Won: MENACE_1
2022-04-27 19:27:12,362 INFO Training Game 5 Time: 2022-04-27 19:27:12.362840
2022-04-27 19:27:12,362 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,383 INFO Draw
2022-04-27 19:27:12,383 INFO Training Game 6 Time: 2022-04-27 19:27:12.383791
2022-04-27 19:27:12,383 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,396 INFO Draw
2022-04-27 19:27:12,396 INFO Training Game 7 Time: 2022-04-27 19:27:12.396831
2022-04-27 19:27:12,396 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,396 INFO Won: MENACE_1
2022-04-27 19:27:12,428 INFO Training Game 8 Time: 2022-04-27 19:27:12.428092
2022-04-27 19:27:12,428 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,445 INFO Won: MENACE_1
2022-04-27 19:27:12,478 INFO Training Game 9 Time: 2022-04-27 19:27:12.478756
2022-04-27 19:27:12,478 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,494 INFO Won: MENACE_2
2022-04-27 19:27:12,529 INFO Training Game 10 Time: 2022-04-27 19:27:12.529369
2022-04-27 19:27:12,529 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,545 INFO Won: MENACE_1
2022-04-27 19:27:12,586 INFO Training Game 11 Time: 2022-04-27 19:27:12.586654
2022-04-27 19:27:12,586 INFO menace_agent_prob: 1 human_agent_prob: 0
2022-04-27 19:27:12,596 INFO Won: MENACE_1
2022-04-27 19:27:12,627 INFO Training Game 12 Time: 2022-04-27 19:27:12.627805
2022-04-27 19:27:12,627 INFO menace_agent_prob: 1 human_agent_prob: 0
```

Flow

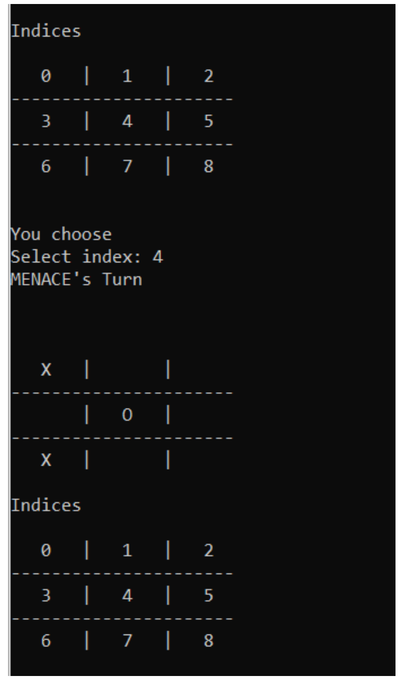


UI

After selection mode **'play '** the code initializes a command-line based game where MENACE **(X)** goes first. Then we are prompted to choose the remaining indices on the board (0 to 8). Our move is represented by **(O).** Our selected index is then displayed on the board.



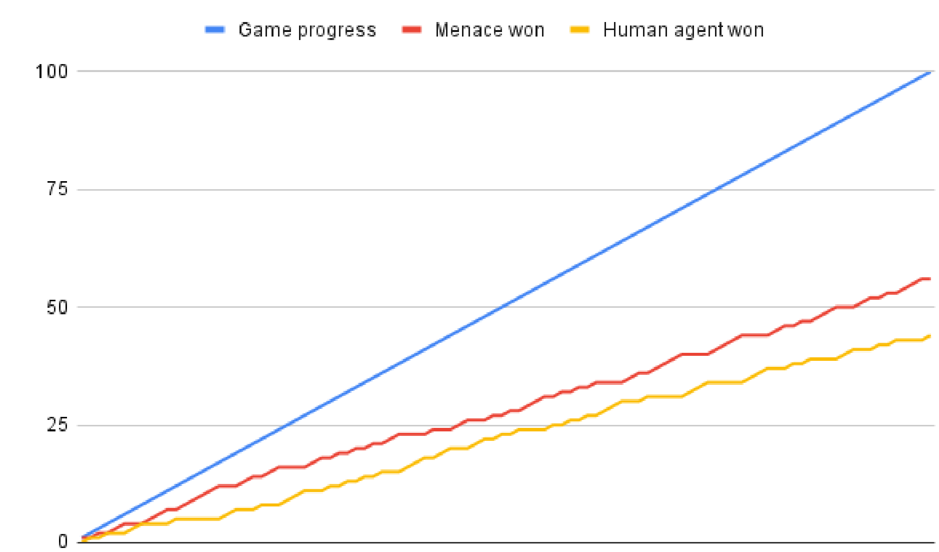
```
Begin game 0

MENACE's Turn


  X  |     |
---------------------
     |     |
---------------------
     |     |

Indices

  0  |  1  |  2
---------------------
  3  |  4  |  5
---------------------
  6  |  7  |  8


You choose
Select index:
```



```
Indices

  0  |  1  |  2
----------------------
  3  |  4  |  5
----------------------
  6  |  7  |  8

You choose
Select index: 4
MENACE's Turn


  X  |     |
----------------------
     |  O  |
----------------------
  X  |     |

Indices

  0  |  1  |  2
----------------------
  3  |  4  |  5
----------------------
  6  |  7  |  8
```

Observations & Graphical Analysis

As we pick an optimal human strategy with a probability of P* and plot the number of games where the menace agent played against the human agent. We observe that the chances of menace winning increases as we play more games



Testcases

tests.py

```python
import unittest

from states import vflip, rotate90, next_state, remove_duplicates
from states import set_possible_moves, get_required_action, get_original_action
from states import is_win, match_two_states, check_equal

class TestMenaceStateMethods(unittest.TestCase):

    def test_invariant(self):
        '''
        Test for checking invariant that difference in
        count of X's and O's on board should be <= 1
        '''
        state = [1,2,2,0,0,1,0,0,0]
        self.assertTrue(abs(state.count(1) - state.count(2))<=1)


    def test_vlip(self):
        '''
        Test for Vertical flip of a state
        '''
        self.assertEqual(vflip([1,0,0,0,0,0,0,0,0]), [0,0,1,0,0,0,0,0,0])

    def test_rotate90(self):
        '''
        Test for rotate state by 90 degrees right
        '''
        self.assertEqual(rotate90([1,2,2,0,0,1,0,0,0]), [0, 0, 1, 0, 0, 2, 0, 1, 2])

    def test_next_state(self):
        '''
        Test for getting next state given current state and a move
        '''
        self.assertEqual(next_state([[1,2,1,2,1,2,1,0,0]], [], 2), [[1, 2, 1, 2, 1, 2, 1, 2, 0], [1, 2, 1, 2, 1, 2, 1, 0, 2]])

    def test_remove_duplicates(self):
        '''
        Test for removing duplicate states
```

tests.py

```python
        self.assertEqual(next_state([[1,2,1,2,1,2,1,0,0]], [], 2), [[1, 2, 1, 2, 1, 2, 1, 2, 0], [1, 2, 1, 2, 1, 2, 1, 0, 2]])

    def test_remove_duplicates(self):
        '''
        Test for removing duplicate states
        from states list after flip and rotation
        '''
        self.assertEqual(remove_duplicates([[1,0,0,0,0,0,0,0,0], [0,0,1,0,0,0,0,0,0]]), [[1,0,0,0,0,0,0,0,0]])

    def test_set_possible_moves(self):
        '''
        Test for obtaining all possible moves
        '''
        self.assertEqual(set_possible_moves([[1,0,0,0,0,0,0,0,0]], 2),
                         {(1, 0, 0, 0, 0, 0, 0, 0, 0):
                         [1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8]})

    def test_get_required_action(self):
        '''
        Test for getting required state given current state,
        number of rotations and flips
        '''
        self.assertEqual(get_required_action(0, 3, 0), 6)

    def test_get_original_action(self):
        '''
        Test for getting back the original state,
        number of rotations and flips done and current state
        '''
        self.assertEqual(get_original_action(6, 3, 0), 0)

    def test_is_win(self):
        '''
        Test for checking who won given a board state
        '''
        self.assertEqual(is_win([1,2,2,2,1,1,2,2,1]), 1)

    def test_match_two_states(self):
        '''
        Test for matching two states index by index
        '''
        self.assertEqual(match_two_states([1,2,2,0,0,1,0,0,0], [1,2,2,0,0,1,0,0,0]), True)

    def test_check_equal(self):
        '''
        Test to check whether two states are equal
        after all possible flips and rotations
        '''
        self.assertEqual(check_equal([1,0,0,0,0,0,0,0,0], [0,0,1,0,0,0,0,0,0]), (True, 3, 0))

if __name__ == '__main__':
    unittest.main()
```

test_invariant (self) : Test for invariant where difference between number of ‘X’ & ‘O’ should not be greater than 1

test_vlip (self) : Test the vertical flip of a state

test_rotate90 (self) : Test for rotate state by 90°

test_next_state (self) : Test whether next state can be achieved from current state after giving a defined move

test_remove_duplicates (self) : Test for excluding duplicates after performing rotational steps (flip/ mirror/ rotate)

test_set_possible_moves (self) : Test for getting all possible moves for that state

test_get_required_action (self) : Test for a particular index to get the position after certain number of flips and rotations

test_get_original_action (self) : Test for a particular index to get the position before certain number of flips and rotations

test_is_win (self) : Test for checking the winner by providing the final board

test_match_two_states (self) : Test for validating whether two given states are equal or not

test_check_equal (self) : Test for validating whether two given states are equal or not after performing rotational steps (flip/ mirror/ rotate)

Testcase Output:

```
...........
----------------------------------------------------------------------
Ran 11 tests in 0.001s

OK
```

Conclusion

- The accuracy of moves menace makes which leads to win or draw improves as we increase the number of our iterations while training the menace agent.
- Another factor on which the accuracy of menace depends is the start number of beads along with the change in the quantity of reward/ punishment beads (alpha, beta, gamma).
- If we want better results we can use an **epsilon-decreasing strategy** which is exploiting the epsilon instead of exploring and decreasing it until it becomes 0, As epsilon decreases our probability of menace winning increases

  NOTE: epsilon = 1 - probability

References

https://towardsdatascience.com/lets-beat-games-using-a-bunch-of-code-part-1-tic-tac-toe-1543e981fec1

https://twice22.github.io/tictactoe/

http://incompleteideas.net/book/bookdraft2018mar16.pdf