INFO 6205 Spring 2022 Project

Menace

**Introduction**

Here in this project we have implemented a Bot called as Menace which can play TicTacToe and it learns how to play it using reinforcement learning. We have implemented similar logic to the YouTube video snippet provided in the project description.

**Aim**

To implement “The Menace” which can play TicTacToe and draw conclusions.

**Approach**

-Instead of match boxes in the real Menace , In our program we have used HashMap with board state as a value in String format and Arraylist of Integers moves on the board as value.

-We have also used another Hashmap<String,Integer> which stores the value of move played for each current board state of the game till the game reaches an end state.

-I have written another menace and an 0.9 best move probability optimal human player.So,that other menace can train the menace to win in the states in which human optimal player cannot.

-I just add the moves that created the menace to win or make a draw some number of times (depending who the menace is playing )into the Matchboxes HashTable or menace training data for each boardstate keys value matching the current game hashtable.

**Program(**code for my UI is on github)

**Game class**

public class Game extends Thread {

public static Logger logger = LoggerFactory.getLogger(Game.class);

public static int[][] board;

public static int turns;

public static int iterationsForMenaceVsMenace = 200000;

public static int iterationsForMenaceVsOptimal = 10000;

public static boolean gameEnded;

public static String player2;

// state of the game either winnine, loosing, or draw or in progress

public static String gameState() {

// check col

for (int i = 0; i < 3; i++) {

int count\_1\_ = 0;

int count\_minus1\_ = 0;

for (int j = 0; j < 3; j++) {

if (board[j][i] == 1) {

count\_1\_++;

}

if (board[j][i] == -1) {

count\_minus1\_++;

}

}

if (count\_1\_ == 3) {

return "1 has won";

}

if (count\_minus1\_ == 3) {

return "-1 has won";

}

}

// check row

for (int i = 0; i < 3; i++) {

int count\_1\_ = 0;

int count\_minus1\_ = 0;

for (int j = 0; j < 3; j++) {

if (board[i][j] == 1) {

count\_1\_++;

}

if (board[i][j] == -1) {

count\_minus1\_++;

}

}

if (count\_1\_ == 3) {

return "1 has won";

}

if (count\_minus1\_ == 3) {

return "-1 has won";

}

}

// check diag

int count\_1\_ = 0;

int count\_minus1\_ = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (i == j) {

if (board[i][j] == 1) {

count\_1\_++;

}

if (board[i][j] == -1) {

count\_minus1\_++;

}

}

}

}

if (count\_1\_ == 3) {

return "1 has won";

}

if (count\_minus1\_ == 3) {

return "-1 has won";

}

// check anti diag

count\_1\_ = 0;

count\_minus1\_ = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (i + j == 2) {

if (board[i][j] == 1) {

count\_1\_++;

}

if (board[i][j] == -1) {

count\_minus1\_++;

}

}

}

}

if (count\_1\_ == 3) {

return "1 has won";

}

if (count\_minus1\_ == 3) {

return "-1 has won";

}

// check draw

if (turns == 9) {

return "its a draw";

} else {

return "the game is still being played";

}

}

public static void playMenaceVsMenace() {

Board gameBoard = Board.getObj();

Menace1.gamesDraw = 0;

Menace1.gamesWon = 0;

Menace1.gamesLost = 0;

for (int i = 0; i < iterationsForMenaceVsMenace; i++) {

gameBoard.reset();

turns = 0;

gameEnded = false;

board = new int[3][3];

while (!gameEnded) {

if (gameState().equals("the game is still being played")) {

if (turns % 2 == 0) {

logger.info("Menace1 moves");

// System.out.println("Menace1 moves");

if (i >= iterationsForMenaceVsMenace - 1) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

logger.error(ex.toString());

}

}

Menace1.menace1Play();

// if (i >= iterationsForMenaceVsMenace - 1)

// logger.info(Arrays.deepToString(board));

System.out.println(Arrays.deepToString(board));

} else if (turns % 2 == 1) {

logger.info("Menace2 moves");

if (i >= iterationsForMenaceVsMenace - 1) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

// Logger.getLogger(Menace1.class.getName()).log(Level.SEVERE, null, ex);

logger.error(ex.toString());

}

}

Menace2.menace2Play();

logger.info(Arrays.deepToString(board));

// System.out.println(Arrays.deepToString(board));

}

} else {

logger.info(Arrays.deepToString(board));

logger.info(gameState());

if (gameState().equals("1 has won")) {

Menace1.incrementWins();

} else if (gameState().equals("-1 has won")) {

Menace1.incrementLosses();

} else if (gameState().equals("its a draw")) {

Menace1.incrementDraws();

}

gameEnded = true;

}

if (!gameEnded) {

turns++;

}

}

Menace1.putDataFromCurrentToMatchBoxes();

Menace2.putDataFromCurrentToMatchBoxes();

turns = 0;

}

}

public static void playMenaceVsOptimal() {

Board gameBoard = Board.getObj();

Random rand = new Random();

Menace1.gamesDraw = 0;

Menace1.gamesWon = 0;

Menace1.gamesLost = 0;

for (int i = 0; i < iterationsForMenaceVsOptimal; i++) {

gameBoard.reset();

turns = 0;

gameEnded = false;

int value;

int[] move = new int[2];

// System.out.println("iteration number: " + i);

board = new int[3][3];

// System.out.println(Arrays.deepToString(board));

while (!gameEnded) {

if (gameState().equals("the game is still being played")) {

if (turns % 2 == 0) {

logger.info("Menace1 moves");

// System.out.println("Menace1 moves");

if (i >= iterationsForMenaceVsOptimal - 1) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

logger.error(ex.toString());

// Logger.getLogger(Menace1.class.getName()).log(Level.SEVERE, null, ex);

}

}

Menace1.menace1Play();

logger.info(Arrays.deepToString(board));

// System.out.println(Arrays.deepToString(board));

} else if (turns % 2 == 1) {

logger.info("Optimal Player plays");

// System.out.println("optimal player plays");

value = Optimal.TicTacMove(board, -1, move);

if (i >= iterationsForMenaceVsOptimal - 1) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

logger.error(ex.toString());

// Logger.getLogger(Menace1.class.getName()).log(Level.SEVERE, null, ex);

}

}

int tempVal = rand.nextInt(10);

if (tempVal != 9) {

changeBoard(move[0], move[1], -1);

} else {

int row = rand.nextInt(3);

int col = rand.nextInt(3);

while (board[row][col] != 0) {

row = rand.nextInt(3);

col = rand.nextInt(3);

}

changeBoard(row, col, -1);

}

logger.info(Arrays.deepToString(board));

// System.out.println(Arrays.deepToString(board));

}

} else {

logger.info(Arrays.deepToString(board));

logger.info(gameState());

// System.out.println(gameState());

gameEnded = true;

if (gameState().equals("1 has won")) {

Menace1.incrementWins();

} else if (gameState().equals("-1 has won")) {

Menace1.incrementLosses();

} else if (gameState().equals("its a draw")) {

Menace1.incrementDraws();

}

}

if (!gameEnded) {

turns++;

}

}

Menace1.putDataFromCurrentToMatchBoxesO();

}

}

public static void playMenaceVsHuman() {

Board gameBoard = Board.getObj();

gameBoard.reset();

board = new int[3][3];

gameEnded = false;

turns = 0;

Menace1.menace1Play();

turns++;

}

public static void humanMakeMove(int row, int col) {

// to avoid unintended moves

if (turns % 2 == 1 && !gameEnded) {

if (gameState().equals("the game is still being played")) {

if (board[row][col] == 0) {

changeBoard(row, col, -1);

turns++;

}

}

}

}

public static void resetBoard() {

player2 = "";

board = new int[3][3];

Board.getObj().reset();

}

public static void changeBoard(int row, int col, int val) {

Board gameBoard = Board.getObj();

board[row][col] = val;

gameBoard.updateCell(row, col, val);

// call function to modify the game board;

}

@Override

public void run() {

if (Game.player2 == "human") {

// if(Thread.currentThread().isAlive()){

// Thread.currentThread().stop();

// }

Game.playMenaceVsHuman();

} else if (Game.player2 == "optimal") {

// if(Thread.currentThread().isAlive()){

// Thread.currentThread().stop();

// }

Game.playMenaceVsOptimal();

} else if (Game.player2 == "menace") {

// if(Thread.currentThread().isAlive()){

// Thread.currentThread().stop();

// }

Game.playMenaceVsMenace();

}

}

}

**Data Structures & classes**

**Menace1 class**

public class Menace1 {

public static Map<String, ArrayList<Integer>> matchBoxes = new HashMap<>();

public static Map<String, Integer> currentGameMoves = new HashMap<>();

public static int gamesWon = 0;

public static int gamesLost = 0;

public static int gamesDraw = 0;

public static int winRewards=5;

public static int winRewardsO=15;

public static int drawRewards=2;

public static int drawRewardsO=5;

public static int punishment=-1;

public static int punishmentO=-1;

public static void menace1Play() {

// create a key for inserting into matchboxes

String boardState = Arrays.deepToString(Game.board);

Random rand = new Random();

matchBoxes.putIfAbsent(boardState, new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9)));

int choose\_one = rand.nextInt(matchBoxes.get(boardState).size());

while (true) {

Integer board\_location = matchBoxes.get(boardState).get(choose\_one);

int[] temp = convertIntToBoardLoc(board\_location);

if (Game.board[temp[0]][temp[1]] == 0) {

Game.changeBoard(temp[0], temp[1], 1);

currentGameMoves.put(boardState, board\_location);

return;

} else {

choose\_one = rand.nextInt(matchBoxes.get(boardState).size());

}

}

}

public static int[] convertIntToBoardLoc(int board\_location) {

int[] temp = new int[2];

switch (board\_location) {

case 1:

temp[0]=0;

temp[1]=0;

break;

case 2:

temp[0]=0;

temp[1]=1;

break;

case 3:

temp[0]=0;

temp[1]=2;

break;

case 4:

temp[0]=1;

temp[1]=0;

break;

case 5:

temp[0]=1;

temp[1]=1;

break;

case 6:

temp[0]=1;

temp[1]=2;

break;

case 7:

temp[0]=2;

temp[1]=0;

break;

case 8:

temp[0]=2;

temp[1]=1;

break;

case 9:

temp[0]=2;

temp[1]=2;

break;

default:

break;

}

return temp;

}

public static void putDataFromCurrentToMatchBoxes() {

if (Game.gameState().equals("1 has won")) {

for (Map.Entry<String, Integer> e : currentGameMoves.entrySet()) {

for (int i = 0; i < winRewards; i++) {

matchBoxes.get(e.getKey()).add(e.getValue());

}

}

}

if (Game.gameState().equals("its a draw")) {

for (Map.Entry<String, Integer> e : currentGameMoves.entrySet()) {

for (int i = 0; i < drawRewards; i++) {

matchBoxes.get(e.getKey()).add(e.getValue());

}

}

}

currentGameMoves.clear();

}

public static void putDataFromCurrentToMatchBoxesO() {

if (Game.gameState().equals("1 has won")) {

for (Map.Entry<String, Integer> e : currentGameMoves.entrySet()) {

for (int i = 0; i < winRewardsO; i++) {

matchBoxes.get(e.getKey()).add(e.getValue());

}

}

}

if (Game.gameState().equals("its a draw")) {

for (Map.Entry<String, Integer> e : currentGameMoves.entrySet()) {

for (int i = 0; i < drawRewardsO; i++) {

matchBoxes.get(e.getKey()).add(e.getValue());

}

}

}

currentGameMoves.clear();

}

public static void incrementWins(){

gamesWon++;

GameBoard.getObj().updateWinCount(gamesWon);

}

public static void incrementLosses(){

gamesLost++;

GameBoard.getObj().updateLossCount(gamesLost);

}

public static void incrementDraws(){

gamesDraw++;

GameBoard.getObj().updateDrawCount(gamesDraw);

}

}

**Menace2 class**

public class Menace2 {

public static Map<String, ArrayList<Integer>> matchBoxes = new HashMap<>();

public static Map<String, Integer> currentGameMoves = new HashMap<>();

public static int gamesWon;

public static int gamesLost;

public static int gamesDraw;

public static int winRewards=5;

public static int drawRewards=2;

public static int punishment=-1;

public static void menace2Play() {

String boardState = Arrays.deepToString(Game.board);

Random rand = new Random();

matchBoxes.putIfAbsent(boardState, new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9)));

if(matchBoxes.get(boardState).isEmpty()){

matchBoxes.put(boardState,new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9)));

}

int choose\_one = rand.nextInt(matchBoxes.get(boardState).size());

while (true) {

Integer board\_location = matchBoxes.get(boardState).get(choose\_one);

int[] temp = convertIntToBoardLoc(board\_location);

if (Game.board[temp[0]][temp[1]] == 0) {

Game.changeBoard(temp[0], temp[1], -1);

currentGameMoves.put(boardState, board\_location);

return;

} else {

choose\_one = rand.nextInt(matchBoxes.get(boardState).size());

}

}

}

public static int[] convertIntToBoardLoc(int board\_location) {

int[] temp = new int[2];

switch (board\_location) {

case 1:

temp[0]=0;

temp[1]=0;

break;

case 2:

temp[0]=0;

temp[1]=1;

break;

case 3:

temp[0]=0;

temp[1]=2;

break;

case 4:

temp[0]=1;

temp[1]=0;

break;

case 5:

temp[0]=1;

temp[1]=1;

break;

case 6:

temp[0]=1;

temp[1]=2;

break;

case 7:

temp[0]=2;

temp[1]=0;

break;

case 8:

temp[0]=2;

temp[1]=1;

break;

case 9:

temp[0]=2;

temp[1]=2;

break;

default:

break;

}

return temp;

}

public static void putDataFromCurrentToMatchBoxes() {

if (Game.gameState().equals("-1 has won")) {

for (Map.Entry<String, Integer> e : currentGameMoves.entrySet()) {

for (int i = 0; i < winRewards; i++) {

matchBoxes.get(e.getKey()).add(e.getValue());

}

}

}

if (Game.gameState().equals("its a draw")) {

for (Map.Entry<String, Integer> e : currentGameMoves.entrySet()) {

for (int i = 0; i < drawRewards; i++) {

matchBoxes.get(e.getKey()).add(e.getValue());

}

}

}

currentGameMoves.clear();

}

}

**Optimal class(for optimal(0.9)human player)**

public class Optimal {

public static int EMPTY = 0;

public static int BOARD\_SIZE = 3;

public static int TicTacMove(int[][] board, int whosemove, int[] best\_place) {

int i;

int j;

int[] place = new int[2];

int best;

int value;

// See if anybody has won

value = TicTacFinal(board);

if (value! = EMPTY) {

best place[0] = -1;

return value;

}

// Try each available move and maximize (or minimize)

best = -2 \* whosemove; // anything will beat this

for (i = 0; i < BOARD\_SIZE; i = i + 1) {

for (j = 0; j < BOARD\_SIZE; j = j + 1) {

if (board[i][j] == EMPTY) {

// Try moving here, see its value, then undo move

int[][] temp=new int[3][3];

for (int t = 0; t < 3; t++) {

for (int s = 0; s < 3; s++) {

temp[t][s] = board[t][s];

}

}

temp[i][j] = whosemove;

value = TicTacMove(temp, -whosemove, place);

temp[i][j] = EMPTY;

// See if value is better than what we have

if ((whosemove < 0 && value < best)

|| (whosemove > 0 && value > best)) {

best = value;

best\_place[0] = i;

best\_place[1] = j;

}

if (best == whosemove) {

// Then I found a guaranteed win; take it.

return best;

}

}

}

}

if (best == -2 \* whosemove) {

// No empty spaces are on board; it's a cat's game.

best\_place[0] = -1;

return 0;

}

return best;

}

// TicTacFinal

//

// returns the code of the player that has won in the board, or EMPTY

// if neither has won.

public static int TicTacFinal(int[][] board) {

int i;

int j;

int k;

// See if any row is filled by a player

for (i = 0; i < BOARD\_SIZE; i = i + 1) {

k = board[i][0];

if (k != EMPTY) {

for (j = 1; j < BOARD\_SIZE; j = j + 1) {

if (board[i][j] != k) {

break;

}

}

if (j == BOARD\_SIZE) {

// then k has filled row

return k;

}

}

}

// See if any column is filled by a player

for (j = 0; j < BOARD\_SIZE; j = j + 1) {

k = board[0][j];

if (k != EMPTY) {

for (i = 1; i < BOARD\_SIZE; i = i + 1) {

if (board[i][j] != k) {

break;

}

}

if (i == BOARD\_SIZE) {

// then k has filled column

return k;

}

}

}

// See if \ diagonal is filled by a player

k = board [0][0];

if (k != EMPTY) {

for (i = 0; i < BOARD\_SIZE; i = i + 1) {

if (board[i][i] != k) {

break;

}

}

if (i == BOARD\_SIZE) {

// then k has filled \ diagonal

return k;

}

}

// See if / diagonal is filled by a player

k = board[0][BOARD\_SIZE - 1];

if (k != EMPTY) {

for (i = 0; i < BOARD\_SIZE; i = i + 1) {

if (board[i][BOARD\_SIZE - 1 - i] != k) {

break;

}

}

if (i == BOARD\_SIZE) {

// then k has filled / diagonal

return k;

}

}

return EMPTY;

}

}

**Datastructures**

We have used HashMap<String,ArrayList<Integer>> for match boxes.

We have used HashMap<String,Integer> for current game moves.

We have used 2d Array to store Board of size 3x3.

**Algorithm**

**Menace play algorithm**

When menace gets a chance to play it first checks if the game is in progress

If the game is in progress then menace checks if there is a similar state the matchboxes hashmap.

Else do not play .

If similar state does not exist then initialize new key in mathchboxes hashmap for key as boardstate with all the moves 1 to 9 and pick a move at random from it and then if that move is not empty on the board try to pick another move at random till you find an empty board move.

If similar boardstate exists then pick a random move from the values in the Arraylist for the key as boardstate in the matchboxes Hashmap.

Store the moves played in the current game in the Hashmap of current game moves.

Now when the current game ends and returns a state of either Draw or a Win we add the moves played in the game into the matchboxes Hashmap n and m number of times where n>m so that for the next game there is a high probability for Menace to play that move rather that the move that I did not win with.

**Optimal human play algorithm (0.9 probability of an optimal move)**

If the game is in terminal state, then if menace wins then 1 if human wins the -1 if draw 0;

If the player is maximizing player then value of the move is some max neg value.

For each child state that means possible next states of the board

Value =max (value ,optimalhumanplay(childboardstate,depth-1,false(meaning minimizing player))

Return value after the recursion completes.

Else if the player is minimizing player, then value is equal to some max positive value.

Now for each child node meaning next state of board

Value=min (val, optimalhumanplay(childboardstate,depth-1,true(meaning maximizing player))

Return that value after the recursion completes.

Now store the child node returning the value and that is the optimal move.

Now to introduce the 0.9 probability of playing optimal move I just generate a random number with bound 10 and then if the generated random number is 9 then human optimal player plays a random move making probability of random move 0.1.

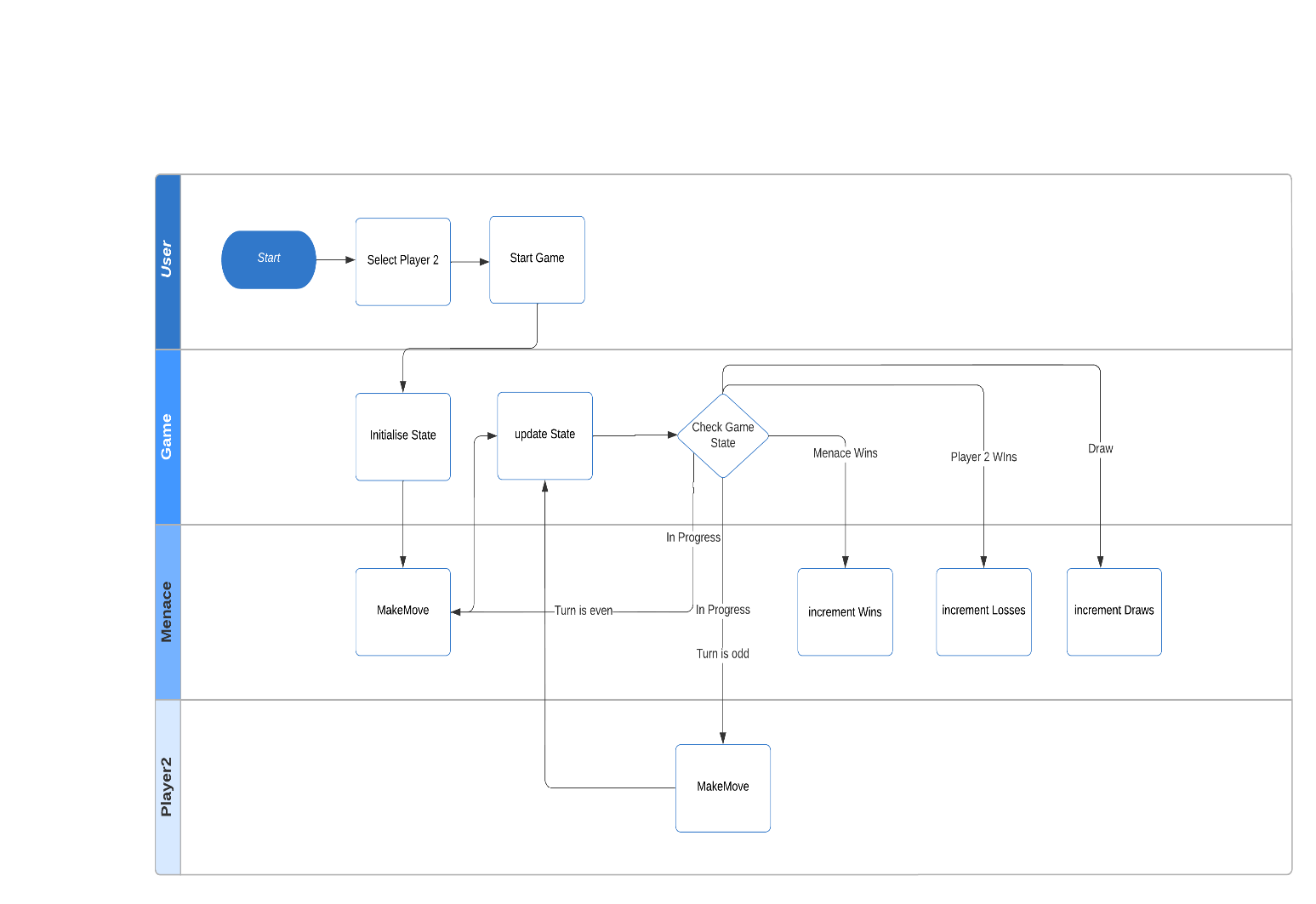
**Invariants**

The turn would never exceed 9.

Number of beads in the HashMap would always be greater than 0;

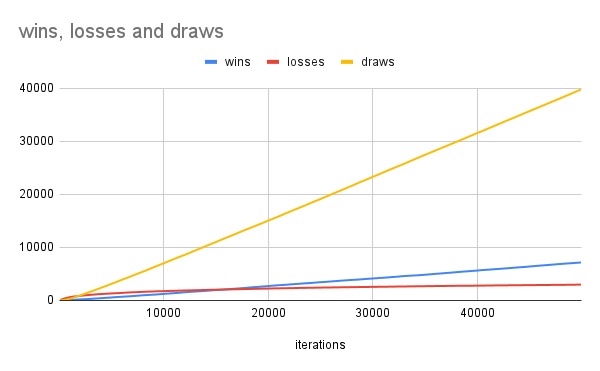
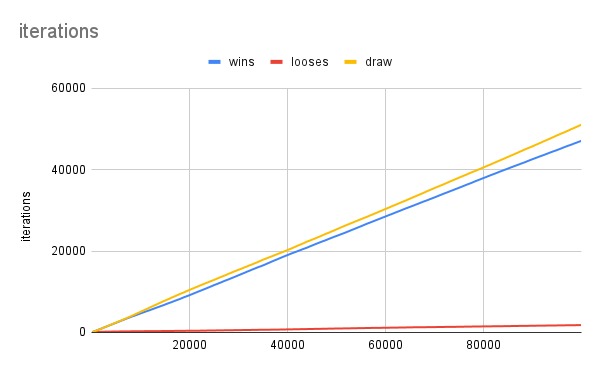
Minimum Number of games required for menace to win are 5.

**Flow Charts (inc. UI Flow)**

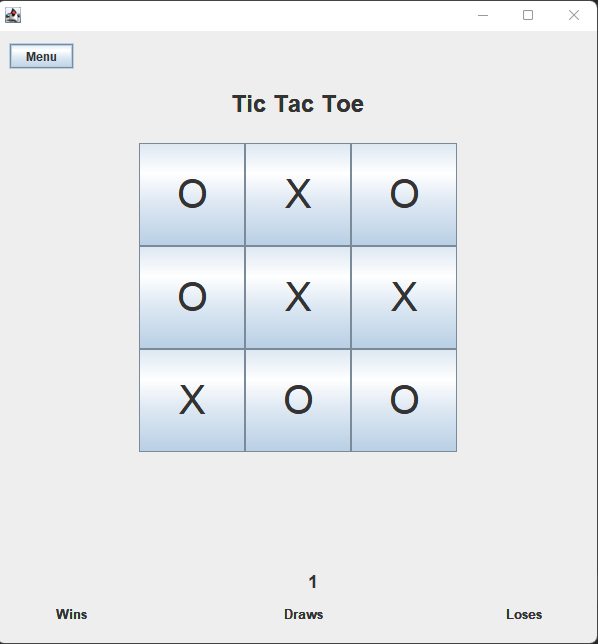


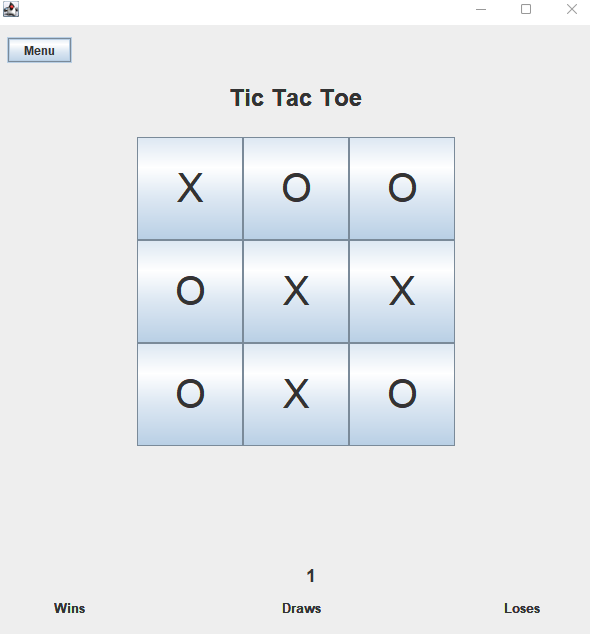
**Observations & Graphical Analysis**

-have observed that when I train my menace with an optimal human player it plays randomly for some boardstates or plays for a draw on some board states that do not reach or are hard to reach in which opponent does not play an optimal move but the moves played by opponent is not bad.So that’s why I train my menace again with an untrained menace2 so that I has some or good data to try for a win in those situations.

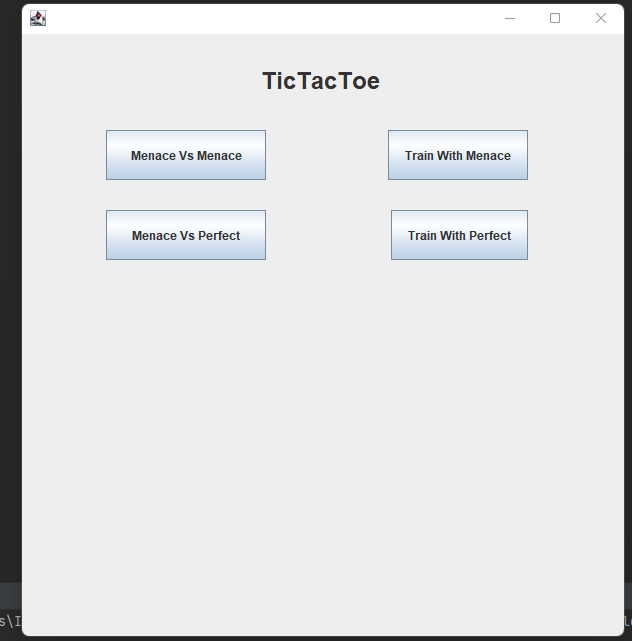


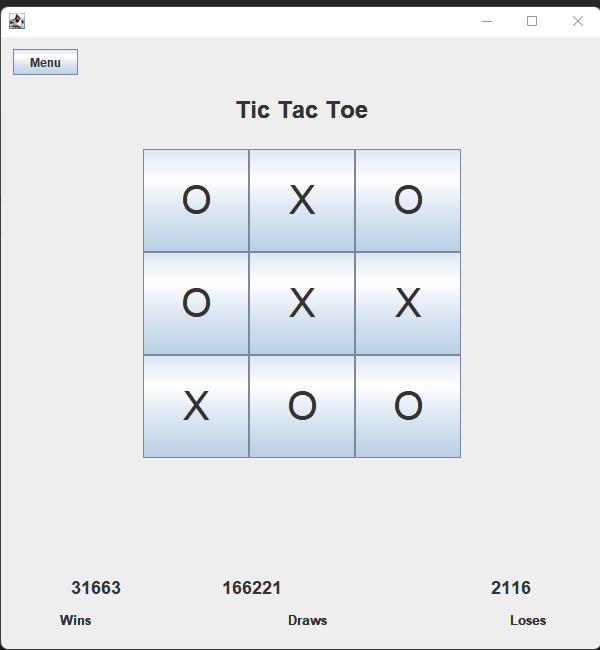
**Results & Mathematical Analysis(data on wins and looses in my in my repo)**



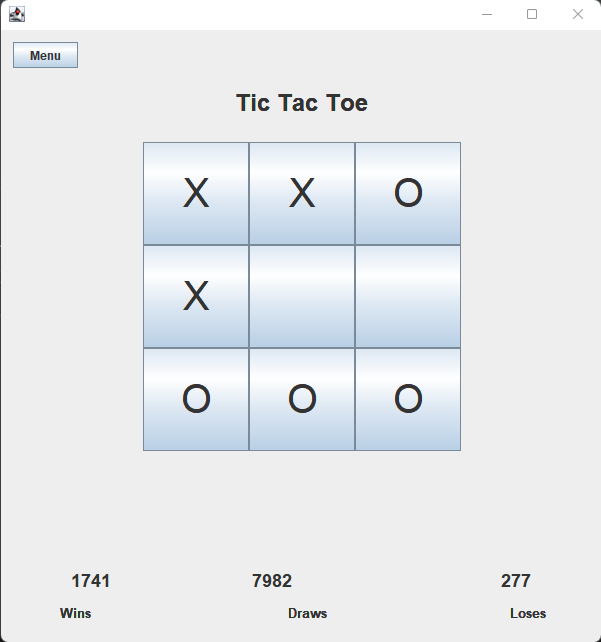
Menace 1 vs Menace 2 screen shot

Menace 1 vs Optimal human player

Menu



Menace training with menace



**Menace training with human optimal player**

For Menace training with optimal player.I have given the last 50 iterations wins ,losses and draws in a txt files in my repo .

For Menace training with Menace 2. I have given the last 50 iterations wins ,losses and draws in a txt files in my repo.

I have used these to make some conclusions.

**Testcases**

public class Menace1Test {

public Menace1Test() {

}

@BeforeEach

public void setUp() {

Game.board = new int[3][3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

Game.board[i][j] = 0;

}

}

Game.turns = 0;

}

/\*\*

\* Test of convertIntToBoardLoc method, of class Menace1.

\*/

@Test

public void testConvertIntToBoardLoc() {

System.out.println("convertIntToBoardLoc");

int board\_location = 2;

int[] expResult = new int[] { 0, 1 };

int[] result = Menace1.convertIntToBoardLoc(board\_location);

assertArrayEquals(expResult, result);

}

@Test

public void testMenace1Play() {

Game.board[0][0] = 1;

Game.board[0][1] = -1;

Game.turns = 2;

Menace1.menace1Play();

int count = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (Game.board[i][j] == 1)

count++;

}

}

assertEquals(count, 2);

}

/\*\*

\* Test of incrementWins method, of class Menace1.

\*/

@Test

public void testIncrementWins() {

System.out.println("incrementWins");

int prevCount = Menace1.gamesWon;

Menace1.incrementWins();

assertEquals(Menace1.gamesWon, prevCount + 1);

}

/\*\*

\* Test of incrementLosses method, of class Menace1.

\*/

@Test

public void testIncrementLosses() {

System.out.println("incrementLosses");

int prevCount = Menace1.gamesLost;

Menace1.incrementLosses();

assertEquals(Menace1.gamesLost, prevCount + 1);

}

/\*\*

\* Test of incrementDraws method, of class Menace1.

\*/

@Test

public void testIncrementDraws() {

System.out.println("incrementDraws");

int prevCount = Menace1.gamesDraw;

Menace1.incrementDraws();

assertEquals(Menace1.gamesDraw, prevCount + 1);

}

}

**Conclusion**

After looking at the data generated menace maintains less than 10 percent probability of loosing after about 9000 iterations with the human(0.9)optimal player.

So even if the menace is trained with the optimal player I have found it important doing some tests that its is necessary for menace to play another menace which starts out with making random moves and learns from its experience so that menace can easily learn to play when averagely good moves are thrown at it and it can try to win.

I have also found that menace learns to fork and plays some really good moves after being trained with optimal player and another unlearned menace which uses reinforcement learning to learn similarly to out menace bot.

**References**

- <https://en.wikipedia.org/wiki/Minimax>

- <https://youtu.be/TtisQ9yZ2zo?t=2210>