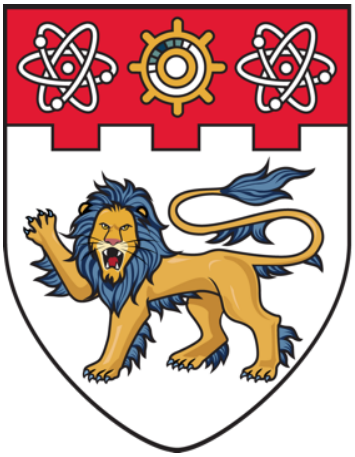

NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

NANYANG TECHNOLOGICAL UNIVERSITY

EE4483 ARTIFICIAL INTELLIGENCE AND DATA MINING

INDIVIDUAL PROJECT II

CHOONG LI QING

U1423070 J

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

YEAR 2017/2018

Contents

Game Play

Tic-tac-toe is a game where two players taking turns marking a 3-by-3 grid with either ‘X’ or ‘O’. The goal of the player is to place three of their marks in a row – horizontally, vertically, or diagonally. The program is written with the intention of playing the perfect game and the following strategies are implemented:

1. Win: If player has 2 in a row, play 3rd to win
2. Block: If opponent has 2 in a row, play 3rd to block
3. Fork: Create opportunity where player has 2 threats to win
4. Blocking Fork:
    1. Create two in a row to force the opponent into defending
       i.e. if "X" has two opponent and "O" is centre, "O" must not play corner
    2. If the opponent can fork, player should block the fork
5. Centre: Player marks centre (as first move), but does not make difference for equally perfect players
6. Opposite corner: If opponent is in corner, play opposite corner
7. Empty corner: Play in a corner square
8. Empty side: Play in middle square on any of the 4 sides

Reference 1 Tic-Tac-Toe Strategy[1]

Heuristic Search

In this program, the best-first algorithm was used as it is relatively simple to implement and less complex in complexity measurement. The heuristic cost considered in this search was the cost of “winning” or “losing”, instead of the idea of point calculation cost.

[1] https://en.wikipedia.org/wiki/Tic-tac-toe

Search Strategy & Complexity

As mentioned in reference 1, Tic-Tac-Toe is a relatively simple game where most people tend to play in a single-stepped mind. This means that the program only must consider the single-step heuristic cost.

Strategy Explanation

Win Combinations

To win in tic-tac-toe, the player must take turns in placing their markers on the table such that they will attain 3-in-a-row.

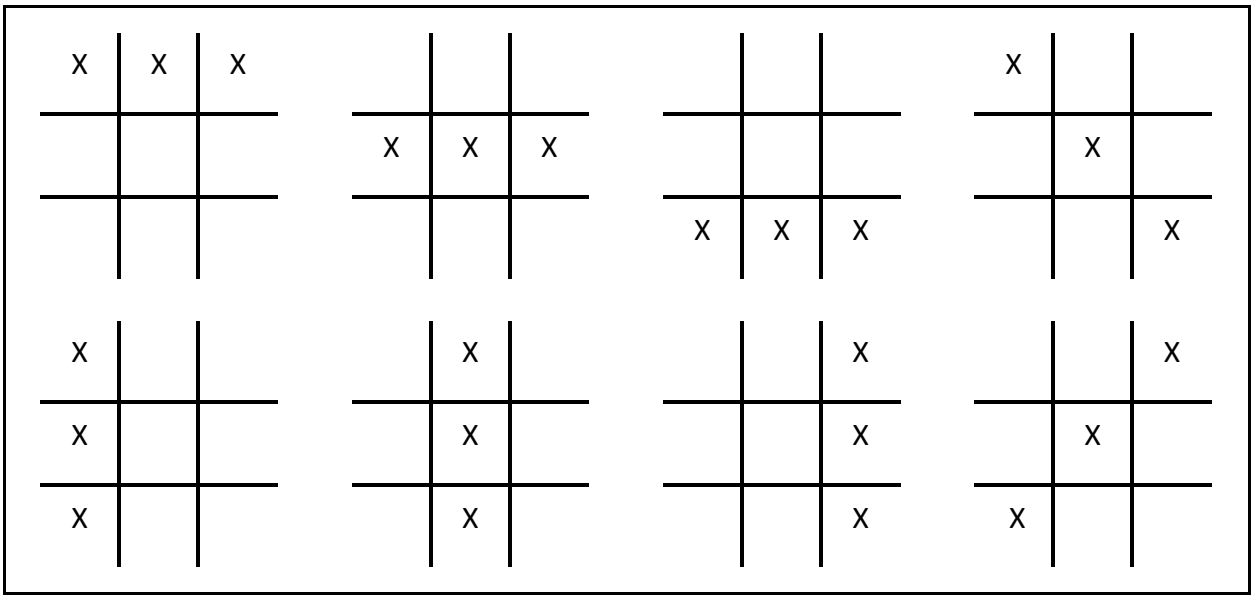

Figure 1 All possible combinations

Win Condition

```
for move in range(1, 10):
        hold = holdBoard(board)
        # Check if can win in next move
        if isEmpty(hold, move):
            setMove(hold, move, aiLetter) # Winning Turn
            if isWin(hold, aiLetter):
                return move
```

Code 1 Winning Turn

The program must first consider the best move to win whereby the AI has already placed two markers on the spaces. (Code 1) calls the program to iterate through the whole board and check if there are empty spaces. If there are empty spaces, it will create a copy of the board and place

a mark there. The program will then check the copied board against the winning combinations and return the position of the marker if a winning combination is found.

Block Condition

```
    for move in range(1, 10):
        hold = holdBoard(board)
        if isEmpty(hold, move):
            setMove(hold, move, playerLetter) # Blocking Turn
            if isWin(hold, playerLetter):
                return move
```

Code 2 Block Turn

(Code 2) is programmed in a similar manner to (code 1), however, it considers the turn whereby the program is unable to win, but it can block the opponent's move.

Both (code 1) and (code 2) is positioned first of priority due to the heuristic cost if "winning" over "losing".

Centre Condition

```
    # Check Center Empty
    if isEmpty(board, 5):
        return 5
```

Code 3 Centre Turn

(Code 3) handles the first plausible move the program can take. This is due to the possibility of the program getting a higher probability of getting a winning combination in that position. However, it might not make a difference if the opponent is equally intellectual and can place a 'fork' combination.

Empty Corner Condition

```
    # If Center empty, take corner
    move = ranChoice(board, [1, 3, 7, 9])
    if move != None:
        return move
```

Code 4 Empty Corner Turn

(Code 4) considers whereby the opponent has placed a centre marker, the program will take any of the corners available. It also considers the probability of attaining a 'fork' combination should the program plays on its second turn.

Empty Side Condition

```
# If center and corners are taken, no block or win
    return ranChoice(board, [2, 4, 6, 8])
```

Code 5 Empty Side Turn

(Code 5) fills the board should there be no possible winning or losing combination. It will fill the board to attain a 'tie'.

Complexity

Unlike other heuristic searches, like minmax and alpha-beta, this search runs at a worst time complexity of n as it runs through all possibilities of its single-step before executing the required move. In terms of space complexity, while it does not require much memory due to its singular-step algorithm, there is the consideration of creating multiple copies of the board to test the winning combinations. This leads to a worst case of N in space complexity.

Advantages & Limitations

It is cleaner and does not require large memory space to run the program at its best. However, there might not be a possibility for the program to win at its best as it does not see further than 2 steps. Due to its narrow-minded step algorithm, it is at most able to attain a 'tied' game result with a human player. Ergo, there is a high possibility that the user will be able to win the program.

Winning Situation

This program is written in the situation that the AI is as equally intelligent as the normal human being. However, it is possible for the program to win should the user make a mistake or have a lower intellectual skill level.

Appendix – Code

```
# Import
import random

# Functions
def start():
    print "Welcome to Tic-Tac-Toe!"
    print "The board will be displayed in the similar format of your numberpad"
    print "You are encouraged to refer to that"

def playAgain():
    print "Thanks for playing!"
    print "Play Again? (Y/N)"
    again = raw_input().upper()

    if again == 'Y':
        return True
    else:
        return False

def drawBoard(board):
    # Follows the numberpad keys
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('-----------')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('-----------')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])

def isWin(board, letter):
    # Check Win Conditions
    # 8 Win conditions
    # b - board, l - player letter
    return((board[7] == letter and board[8] == letter and board[9] == letter) or
           (board[4] == letter and board[5] == letter and board[6] == letter) or
           (board[1] == letter and board[2] == letter and board[3] == letter) or
           (board[7] == letter and board[5] == letter and board[3] == letter) or
           (board[7] == letter and board[4] == letter and board[1] == letter) or
           (board[8] == letter and board[5] == letter and board[2] == letter) or
           (board[9] == letter and board[6] == letter and board[3] == letter) or
           (board[9] == letter and board[5] == letter and board[1] == letter))

def holdBoard(board):
    # For System Checking
    boardCopy = []
```

```
    for i in board:
        boardCopy.append(i)
    return boardCopy

def isEmpty(board, move):
    return board[move] == ' '

def isPlayerMove(board):
    move = ' '
    arr = '1 2 3 4 5 6 7 8 9'.split()
    while move not in arr or not isEmpty(board, int(move)):
        print "Your turn. (Select from 1 to 9)"
        move = raw_input()
        if isEmpty(board, int(move)) == False:
            print "Invalid move. It has been occupied"
    return int(move)

def setMove(board, move, letter):
    board[move] = letter

def isFull(board):
    for i in range(1, 10):
        if isEmpty(board, i):
            return False
    return True

# Computer's Turn
def aiTurn(board, aiLetter, playerLetter):
    """
    The "Perfect" Strategy
    1. Win: If player has 2 in a row, play 3rd to win
    2. Block: If oppo has 2 in a row, play 3rd to block
    3. Fork: Create opportunity where player has 2 threats to win
    4. Blocking Fork:
        1. Create two in a row to force the oppo into defending
            ie. if "X" has two oppo and "O" is center, "O" must not play corner
        2. If the oppo can fork, player should block the fork
    5. Center: Player marks center (as first move), but does not make difference
for
        equally perfect players
    6. Opposite corner: If oppo is in corner, play oppo corner
    7. Empty corner: Play in a corner square
    8. Empty side: Play in middle square on any of the 4 sides
    """
```

```
    for move in range(1, 10):
        hold = holdBoard(board)
        # Check if can win in next move
        if isEmpty(hold, move):
            setMove(hold, move, aiLetter) # Winning Turn
            if isWin(hold, aiLetter):
                return move

    for move in range(1, 10):
        hold = holdBoard(board)
        if isEmpty(hold, move):
            setMove(hold, move, playerLetter) # Blocking Turn
            if isWin(hold, playerLetter):
                return move

    # Check Center Empty
    if isEmpty(board, 5):
        return 5

    # If Center empty, take corner
    move = ranChoice(board, [1, 3, 7, 9])
    if move != None:
        return move

    # If center and corners are taken, no block or win
    return ranChoice(board, [2, 4, 6, 8])

# Choosing Random from list
def ranChoice(board, moves):
    possibleMoves = []
    for i in moves:
        if isEmpty(board, i):
            possibleMoves.append(i)

    if len(possibleMoves) != 0:
        return random.choice(possibleMoves)
    else:
        return None

# Main
start()
while True:
    print "Do you wish to go first? (Y/N)"
    goFirst = raw_input().upper()
    # Player One plays X, player Two plays O
```

```
if goFirst == 'Y':
    playerLetter = 'X'
    aiLetter = 'O'
    turn = 'player'
else:
    aiLetter = 'X'
    playerLetter = 'O'
    turn = 'computer'
board = [' '] * 10 #ignoring index 0
drawBoard(board)

isPlaying = True

while isPlaying:
    if turn == 'player':
        # Player's turn
        move = isPlayerMove(board)
        setMove(board, move, playerLetter)
        drawBoard(board)

        if(isWin(board,playerLetter)):
            #Check if Win
            print "You won!"
            isPlaying = False

        else:
            if(isFull(board)):
                #Check if Tie
                print "It's a tie!"
                isPlaying = False
            else:
                #Handover turn
                turn = 'computer'


    if turn == 'computer':
        # Compuer's turn

        move = aiTurn(board, aiLetter, playerLetter)
        setMove(board, move, aiLetter)
        drawBoard(board)

        if(isWin(board,aiLetter)):
            #Check if Win
            print "You lost!"
```

```
                isPlaying = False
            else:
                if(isFull(board)):
                    #Check if Tie
                    print "It's a tie!"
                    isPlaying = False
                else:
                    #Handover turn
                    turn = 'player'

    if not playAgain():
        break
```