EAST WEST UNIVERSITY

10/5/2023

Mini Project CSE366 - 03

Game 1: Tic Tac Toe

Prepared For

Redwan Ahmed Rizvee
Department of Computer Science & Engineering
EAST WEST UNIVERSITY

Prepared By

Sanjida Akter (2020-1-60-222)
Tarek Rahman (2020-1-60-218)
MD. Farhad Billah (2020-2-60-213)
Israth Jahan (2020-1-60-044)

Code:

```
#Copy() in Python Programming is a method that is used on objects to
create copies of them.
import copy

# Constants are variables whose value is not expected to change during the
execution of a program.
PLAYER_X = 'X'#(user)
PLAYER_O = 'O'#(AI)
EMPTY_CELL = '-'

# Represents a game state
class GameState:
    def __init__(self, board, last_move, next_player):
        # 5x5 list representing the game board
        self.board = board
        # Tuple representing the row and column of the last move made
        self.last_move = last_move
        # Player who will make the next move ('X' or 'O')
        self.next_player = next_player

    # There are 25 blocks where we can put our values ('x' and 'O').
    def get_possible_moves(self):
        moves = []
        for i in range(5):
            for j in range(5):
                if self.board[i][j] == EMPTY_CELL:
                    moves.append((i, j))
        return moves

    #Applies a move to the current state and returns a new state
    def apply_move(self, move):
        # creating a new board using 'copy' module. The deepcopy() method
is used to create a new copy of the board that is independent of the
original board.
        new_board = copy.deepcopy(self.board)
        #that represent the row and column where the player wants to place
their symbol ('x' or 'o') on the game
        new_board[move[0]][move[1]] = self.next_player
        #It creates a new copy of the game board, updates it with the new
move, and returns a new GameState object representing the new state of the
game.
        return GameState(new_board, move,
get_next_player(self.next_player))


# Returns the player who will make the next move
def get_next_player(player):
    # play user next is AI nad play Ai then next is user.
    return PLAYER_O if player == PLAYER_X else PLAYER_X



# Returns True if the given row is a winning row
def is_winning_row(row, player):
    return ''.join(row).count(player * 3) > 0 or \
           ''.join(row[0:3]).count(player * 3) > 0 or \
           ''.join(row[1:4]).count(player * 3) > 0 or \
           ''.join(row[2:5]).count(player * 3) > 0 or \
           row[0] == row[1] == row[2] == player or \
           row[1] == row[2] == row[3] == player or \
           row[2] == row[3] == row[4] == player or \
           row[0] == row[1] == row[2] == row[3] == player \
           or row[1] == row[2] == row[3] == row[4] == player

# Returns True if the given column is a winning column
def is_winning_column(board, col, player):
    column = [board[row][col] for row in range(5)]
    return ''.join(column).count(player * 3) > 0 or \
           ''.join(column[0:3]).count(player * 3) > 0 or \
           ''.join(column[1:4]).count(player * 3) > 0 or \
           ''.join(column[2:5]).count(player * 3) > 0 or \
           column[0] == column[1] == column[2] == player or \
           column[1] == column[2] == column[3] == player or \
           column[2] == column[3] == column[4] == player or \
           column[0] == column[1] == column[2] == column[3] == player or \
           column[1] == column[2] == column[3] == column[4] == player


def is_winning_diagonal(board, player):
    #left to right
    if ''.join([board[i][i] for i in range(5)]).count(player*3) > 0:
        return True
    #2 upper
    if ''.join([board[i][j] for i,j in
zip(range(4),range(1,5))]).count(player*3) > 0:
        return True
    #1 upper
    if ''.join([board[i][j+2] for i,j in
zip(range(4),range(0,3))]).count(player*3) > 0:
        return True
    #2 down
    if ''.join([board[i+2][j] for i,j in
zip(range(0,3),range(0,5))]).count(player*3) > 0:
        return True
    #right to left
    if ''.join([board[i][4-i] for i in range(5)]).count(player*3) > 0:
        return True
    #2 upperside
    if ''.join([board[i][4-j] for i,j in
zip(range(0,5),range(1,5))]).count(player*3) > 0:
        return True
    #1 upper
    if ''.join([board[i][3-j] for i,j in
zip(range(0,5),range(1,5))]).count(player*3) > 0:
        return True
    #1 down
    if ''.join([board[i][5-j] for i, j in zip(range(1, 5),
range(1,5))]).count(player * 3) > 0:
        return True
    # 2 down
    if ''.join([board[i][j] for i,j in
zip(range(1,5),range(4))]).count(player*3) > 0:
        return True
    # 2 down
    if ''.join([board[i][5 - j] for i, j in zip(range(2, 5), range(1,
5))]).count(player * 3) > 0:
        return True
    # Return False combination is found
    return False



# Returns True if the given player has won the game
def has_player_won(game_state, player):
    board = game_state.board
    #After this operation, last_row and last_col variables hold the row and column of the last move made, respectively. These variables are used in the subsequent lines of the function to check if the player has won the game.
    last_row, last_col = game_state.last_move
    # Check row
    if is_winning_row(board[last_row], player):
        return True
    # Check column
    if is_winning_column(board, last_col, player):
        return True
    # Check diagonals
    if is_winning_diagonal(board, player):
        return True
    return False


# This function get_score is used to determine the score of the given player in the current game_state. The score is used in the game's decision-making process to choose the best possible move for the player.
def get_score(game_state, player):
    #If the player has won, the score returned is 1.
    if has_player_won(game_state, player):
        return 1
    #If the player has lost, the score returned is -1.
    elif has_player_won(game_state, get_next_player(player)):
        return -1
    #Otherwise, the game is still ongoing, and no player has won or lost yet, so the function returns 0.
    else:
        return 0


# Applies backtracking to calculate the minimax score of the given game state
def backtracking(game_state, depth, player):
    # If the maximum search depth has been reached or the game is over, return the score of the current game state for the given player.
    if depth == 0 or game_state.is_game_over():
        return get_score(game_state, player)

    # Get all possible moves for the current game state.
    possible_moves = game_state.get_possible_moves()

    # If it's the turn of the maximizing player (PLAYER_X), enter the
maximizing part of the algorithm.
    if player == PLAYER_X:  # Maximizer
        max_score = float('-inf')
        # For each possible move, apply the move to the current game state
and get the score of the resulting game state using backtracking
recursively with depth-1.
        for move in possible_moves:
            new_game_state = game_state.apply_move(move)
            score = backtracking(new_game_state, depth-1,
get_next_player(player))
            # Update the maximum score if the new score is higher.
            max_score = max(max_score, score)
        return max_score

    # If it's the turn of the minimizing player (PLAYER_O), enter the
minimizing part of the algorithm.
    else:  # Minimizer
        min_score = float('inf')
        # Same as above, but update the minimum score if the new score is
lower.
        for move in possible_moves:
            new_game_state = game_state.apply_move(move)
            score = backtracking(new_game_state, depth-1,
get_next_player(player))
            min_score = min(min_score, score)
        return min_score


import random

# Returns the best move for the AI player using backtracking with a given
depth limit
def get_ai_move(game_state, depth):
    possible_moves = game_state.get_possible_moves()
    if not possible_moves:
        return None

    if random.random() < .8:
        return random.choice(possible_moves)
    best_move = possible_moves[0]
    max_score = float('-inf')
    for move in possible_moves:
        new_game_state = game_state.apply_move(move)
        score = backtracking(new_game_state, depth-1, PLAYER_X)
        if score > max_score:
            max_score = score
            best_move = move
    return best_move


# Displays the current game board
def display_board(board):
    print("  0 1 2 3 4")
    for i in range(5):
        row_str = str(i) + " "
        for j in range(5):
            row_str += board[i][j] + " "
        print(row_str)

# Handles user input and returns a tuple representing the row and column
of the move
def get_user_move():
    while True:
        try:
            row = int(input("Enter row number (0-4): "))
            col = int(input("Enter column number (0-4): "))
            if row < 0 or row > 4 or col < 0 or col > 4:
                raise ValueError()
            break
        except ValueError:
            print("Invalid input. Please enter a number from 0 to 4.")
    return (row, col)


# Main game loop
def play_game():
    # Initialize game state 5x5
    board = [[EMPTY_CELL] * 5 for _ in range(5)]
    game_state = GameState(board, None, PLAYER_X)

    print("Welcome to Tic Tac Toe! You are playing as 'X'.")
    display_board(game_state.board)
    while True:
        # Player X's turn (user)
        if game_state.next_player == PLAYER_X:
            print("Your turn!")
            move = get_user_move()
        # Player O's turn (AI)
        else:
            print("AI's turn...")
            move = get_ai_move(game_state, 3)
            print(f"AI chooses ({move[0]}, {move[1]})")

        game_state = game_state.apply_move(move)
        display_board(game_state.board)

        # Check for end of game
        if has_player_won(game_state, PLAYER_X):
            print("Congratulations, you won!")
            break
        elif has_player_won(game_state, PLAYER_O):
            print("Sorry, you lost.")
            break
        elif not game_state.get_possible_moves():
            print("It's a tie!")
            break
# Start the game
play_game()
```

Problem Statement:

Implementing a 2-player game of Tic Tac Toe with a 5 x 5 board. The players will be a human and a computer. The goal is to match symbols in consecutive three cells either vertically or horizontally or diagonally. The computer will use AI to determine its moves.

The game will start with a given initial configuration of the board, including the human player's last move. The game will alternate between the human player and the computer. The computer will make its move based on AI knowledge, and the human player will input their moves.

The game will end when either a player matches their symbols in consecutive three cells or when the board is full. In the case of a full board without a winner, the game will end in a draw.

Formal algorithm writeup and discussion how the problem is approached:

1. The 'GameState' class represents the state of the game at any given moment. It has three attributes: 'board' , 'Last_move' , and 'next_player'. 'board' is 5x5 list representing the game board. 'last_move' is a tuple representing the row and column of the last move made. 'next_player' is the player who will make the next move ('X' or 'O').

2. The 'get_possible_moves' method returns a list of tuples representing the possible moves that can be made on the current board. It does this by iterating through every cell on the board and checking if it is empty.

3. The 'apply_move' method takes a move represented as a tuple and applies it to the current game state. It creates a new copy of the game board using the 'copy' module, updates it with the new move, and returns a new 'GameState' object representing the new state of the game.

4. The ‘get_next_player’ function takes the current player as an argument and returns the player who will make the next move.

5. The ‘is_winning_row’, ’is_winning_column’, and ‘is_winning_diagonal’ functions check if a row, column, or diagonal respectively is a winning combination for a given player.

6. The ‘get_socre’ to calculate the score of a given game state for a specific player. This function is used in the minimax algorithm’s decision-making process to choose the best possible move for the player.

7. The minimax algorithm suing ‘backtracking’ to find the best move for the AI player. This algorithm recursively evaluates all ‘get_possible_game’, ’states’ up to a certain ‘depth’ limit to determine the best move for the AI player. Once the best move is found, it returned to the main game loop, and the game continues.

8. The game loop handles user input and calls the necessary functions to update the game state and check for the end of the game. If the game is over, the loop displays the appropriate message and exits. Otherwise, the loop continues until the game is over.

Discussion:

This is a Tie Tac Teo game in which the user plays against an AI player. The game is represented by 5x5 board with X’s and O’s. The user always goes first as X, and the AI player always goes second as O. The first player to get three in a row, column, left diagonals, right diagonals win the game. If all the spaces on the board are filled and no player has three in a row, column, left diagonals and right diagonals, the game is a draw.

Conclusion:

In conclusion we can say, the tic-tac-toe game is basically for two players. One player plays X and the other plays O. The players take turns placing their marks on a grid of five -by-five cells. If a given player gets three marks in a row horizontally, vertically, or diagonally, then that player wins the game. Tic Tac Toe can be used to promote several cognitive skills including counting and spatial skills, and color and shape identification. In order to make the game unbeatable, it was necessary to create an algorithm that could calculate all the possible moves available for the computer player and use some metric to determine the best possible move. After extensive research it became clear that the Minimax algorithm was right for the job.