COMP 469 ARTIFICIAL INTELLIGENCE

Professor Taehyung (George) Wang.

CSUN

3 X 3 TIC - TAC - TOE

(ALGORITHM)

Team Members:

Pietro Cicciari,
Mario Choto,
Israel Sanchez,
Raj Kumar,
Armon Lee,
Venkat Sairam Ravala

COMP 469 ARTIFICIAL INTELLIGENCE

Professor Taehyung (George) Wang.

Content:

3 X 3 tic-tac-toe problem (fully quoted from homework assignment)

Tic Tac Toe is a two-player game often played on a 3x3 grid. The game involves two players, one using the symbol "X" and the other using "O". The objective is to be the first to form a straight line of three of one's own symbols, either horizontally, vertically, or diagonally

Specific Requirements or Constraints: For this project, we choose a **goal-based agent** to solve the Tic-Tac-Toe problem. This type of agent is good because it focuses on achieving a specific goal, which in this case is either winning the game or preventing the opponent from winning.

PEAS Framework:

- **Performance:** The agent's performance is measured by its ability to win or, at least, not lose the game (Ideally).
- **Environment:** The environment is the 3x3 Tic-Tac-Toe board where the game is played.
- **Actuators:** The actions of the agent include placing an "X" or "O" on the board.
- **Sensors:** The agent uses sensors to see the current state of the board and to check where the opponent has placed their symbol.

Properties of the Task Environment:

- **Fully Observable:** The agent can see the entire board and knows where every symbol is placed.
- **Deterministic:** The outcome of placing a symbol in a specific position is predictable.
- **Sequential:** The game is turn-based, meaning players take turns, so the actions happen in sequence.

- **Static:** The board only changes when a player places a symbol.
- **Discrete:** The board has a fixed number of squares (9), and each move is made in one of these squares.
- **Known:** The rules and the layout of the board are well-known and simple.

Formal Specifications of the Problem:

- **Possible States:** Each state represents the current configuration of symbols on the 3x3 grid.
- **Initial State:** The board is empty with no symbols placed.
- **Goal State(s):** The goal is to have three "X"s or "O"s in a row, column, or diagonal.
- **Actions Available to the Agent:** The agent can place its symbol ("X" or "O") in any empty square.
- **Transition Model:** The state changes when a symbol is placed on the board.
- **Action Cost Function:** Each move is of equal cost; the focus is on winning, not the number of moves.
- **Heuristic Function:** The heuristic will evaluate how close the agent is to winning (e.g., two in a row with an open third space) or to blocking the opponent from winning.

Approach and Design

Explanation of the Chosen Approach:

1. **Minimax Algorithm:**
    - This algorithm is a decision-making tool that looks ahead at all possible moves and their outcomes. It tries to minimize the worst-case scenario (like losing) and maximize the best-case scenario (like winning).

Pseudocode:

```
#initialize pygame and set yp game environment
initialize pygame
set up screen with WIDTH and HEIGHT
define colors (WHITE, BLACK, RED, GREEN, BLUE, GREY)
define board size and figure sizes (SQUARE_SIZE, CIRCLE_RADIUS, CIRCLE_WIDTH, CROSS_WIDTH)

#initialize the game board
create a 3x3 matrix (board) with all cells set to 0 (empty)

#function that draws the grid lines
def draw_lines(color=WHITE):
    #draw horizontal and vertical lines to create a 3x3 grid
    for i from 1 to BOARD_ROWS-1:
        draw horizontal line at position SQUARE_SIZE * i
        draw vertical line at position SQUARE_SIZE * i

#function that draws x and o on the board
def draw_figures():
    #goes over each cell in the board
    for row from 0 to BOARD_ROWS-1:
        for col from 0 to BOARD_COLUMNS-1:
            if board[row][col] == 1:
                #draw for player x
                draw cross lines at (col, row)
            elif board[row][col] == 2:
                #draw for player o
                draw circle at (col, row)

#fuhnction to mark square on board
def mark_square(row, col, player):
    #set board cell at (row, col) to the player's number (1 for x, 2 for o)
    board[row][col] = player

#function that checks if a square is open
def available_square(row, col):
    #return true if the cell is empty, otherwise return false
    return board[row][col] == 0

#function to check of board is full
def is_board_full(check_board=board):
    # Return True if there are no empty cells left on the board, otherwise False
    return not any cell is 0 in check_board

#checks for win
def check_win(player, check_board=board):
    #checks all rows, columns, and diagonals for a win
    for col from 0 to BOARD_COLUMNS-1:
        if all cells in column col are player:
            return true
    for row from 0 to BOARD_ROWS-1:
        if all cells in row row are player:
            return true
    if all cells in diagonal from top-left to bottom-right are player:
        return true
    if all cells in diagonal from top-right to bottom-left are player:
        return true
    return false

#minmax algorithm
def minmax(minmax_board, depth, is_maximizing):
    #base cases for algorithm
    if check_win(2, minmax_board):
        return 1  #ai wins
    elif check_win(1, minmax_board):
        return -1  #player wins
    elif is_board_full(minmax_board):
        return 0  #draw
```

```
    if is_maximizing:
        #maximize the score for ai
        best_score = -infinity
        for each empty cell in minmax_board:
            mark cell with AIs move
            score = minmax(minmax_board, depth + 1, false)
            unmark cell
            best_score = max(score, best_score)
        return best_score
    else:
        #minimize the score for player
        best_score = infinity
        for each empty cell in minmax_board:
            mark cell with player's move
            score = minmax(minmax_board, depth + 1, true)
            unmark cell
            best_score = min(score, best_score)
        return best_score

#function that will find best move for ai
def find_best_move():
    best_move = (-1, -1)
    best_value = -infinity
    for each empty cell in board:
        mark cell with AI's move
        move_value = minmax(board, 0, false)
        unmark cell
        if move_value > best_value:
            best_value = move_value
            best_move = cell coordinates
    return best_move

#display messages on screen
def display_message(message, color):
    create font object
    render message text with specified color
    get text position
    draw text on screen at the text position
    update display to show the message

#game loop
def main():
    initialize game state
    draw initial grid
    set board to empty
    set player's turn to true
    set game-over flag to false

    while true:
        #reset the game for a new round
        reset board
        draw grid
        update display

        player_turn = true
        game_over = false

        while not game_over:
            for event in event queue:
                if event is quitted:
                    exit the game
```

```
                if event is mousebuttondown and it is player's turn and game is not over:
                    get mouse position
                    convert mouse position to board cell coordinates
                    if selected cell is empty:
                        mark the cell with player's move
                        check for win or draw
                        if win or draw:
                            game_over = true
                        else:
                            switch turn to ai

                if it is AI's turn and game is not over:
                    find the best move using minmax
                    mark the cell with AI's move
                    check for win or draw
                    if win or draw:
                        game_over = true
                    switch turn to player

                draw updated board and figures
                update display

            if game_over:
                clear screen with gray background
                redraw grid and figures
                display game result message (win or draw)
                wait for 2 seconds

                #restart game
                clear screen
                redraw grid

if __name__ == "__main__":
    run main function
```

Code Implementation

```
import pygame
import numpy as np
import sys

try:
    pygame.init()
except pygame.error as e:
    print(f"Failed to initialize Pygame: {e}") #error handling
    sys.exit()

WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)
GREY = (180, 180, 180)
GREEN = (0, 255, 0)

WIDTH = 300
HEIGHT = 300
LINE_WIDTH = 5
BOARD_ROWS = 3
BOARD_COLUMNS = 3
SQUARE_SIZE = WIDTH // BOARD_COLUMNS
CIRCLE_RADIUS = SQUARE_SIZE // 3
CIRCLE_WIDTH = 15
CROSS_WIDTH = 25

try:
    screen = pygame.display.set_mode((WIDTH, HEIGHT))
    pygame.display.set_caption('Tic Tac Toe Minmax')
except pygame.error as e:
    print(f"Failed to set up display: {e}") #error handling
    pygame.quit()
    sys.exit()

screen.fill(BLACK)

board = np.zeros((BOARD_ROWS, BOARD_COLUMNS))

def draw_lines(color=WHITE):
    for i in range(1, BOARD_ROWS):
        pygame.draw.line(screen, color, (0, SQUARE_SIZE * i), (WIDTH, SQUARE_SIZE * i), LINE_WIDTH)
        pygame.draw.line(screen, color, (SQUARE_SIZE * i, 0), (SQUARE_SIZE * i, HEIGHT), LINE_WIDTH)

def draw_figures():
    for row in range(BOARD_ROWS):
        for column in range(BOARD_COLUMNS):
            if board[row][column] == 1:
                pygame.draw.line(screen, RED, (column * SQUARE_SIZE + SQUARE_SIZE // 4, row * SQUARE_SIZE + SQUARE_SIZE // 4),
                                 (column * SQUARE_SIZE + 3 * SQUARE_SIZE // 4, row * SQUARE_SIZE + 3 * SQUARE_SIZE // 4), CROSS_WIDTH)
                pygame.draw.line(screen, RED, (column * SQUARE_SIZE + SQUARE_SIZE // 4, row * SQUARE_SIZE + 3 * SQUARE_SIZE // 4),
                                 (column * SQUARE_SIZE + 3 * SQUARE_SIZE // 4, row * SQUARE_SIZE + SQUARE_SIZE // 4), CROSS_WIDTH)
            elif board[row][column] == 2:
                pygame.draw.circle(screen, BLUE, (column * SQUARE_SIZE + SQUARE_SIZE // 2, row * SQUARE_SIZE + SQUARE_SIZE // 2),
                                   CIRCLE_RADIUS, CIRCLE_WIDTH)
```

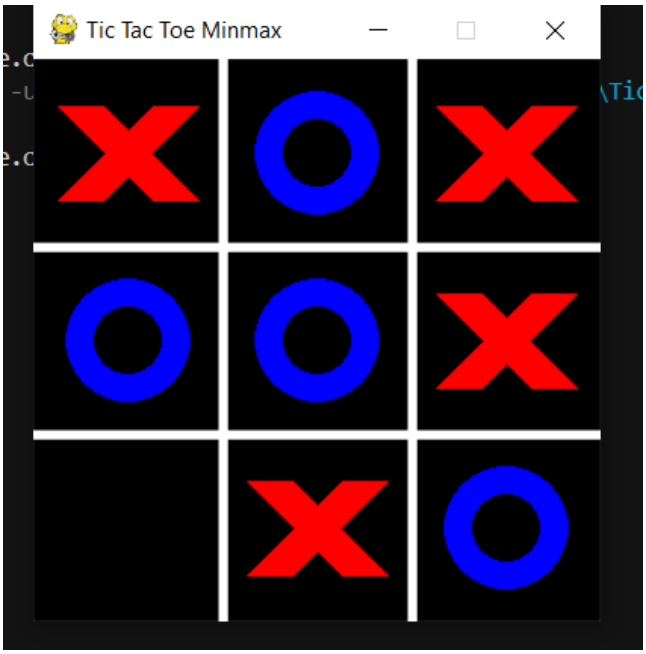
```
def mark_square(row, column, player):
    board[row][column] = player

def available_square(row, column):
    return board[row][column] == 0

def is_board_full(check_board=board):
    return not np.any(check_board == 0)

def check_win(player, check_board=board):
    for column in range(BOARD_COLUMNS):
        if np.all(check_board[:, column] == player):
            return True
    for row in range(BOARD_ROWS):
        if np.all(check_board[row, :] == player):
            return True
    if np.all(np.diag(check_board) == player):
        return True
    if np.all(np.diag(np.fliplr(check_board)) == player):
        return True
    return False

def minmax(minmax_board, depth, is_maximizing):
    if check_win(2, minmax_board):
        return 1
    elif check_win(1, minmax_board):
        return -1
    elif is_board_full(minmax_board):
        return 0

    if is_maximizing:
        best_score = -float('inf')
        for row in range(BOARD_ROWS):
            for column in range(BOARD_COLUMNS):
                if minmax_board[row][column] == 0:
                    minmax_board[row][column] = 2
                    score = minmax(minmax_board, depth + 1, False)
                    minmax_board[row][column] = 0
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for row in range(BOARD_ROWS):
            for column in range(BOARD_COLUMNS):
                if minmax_board[row][column] == 0:
                    minmax_board[row][column] = 1
                    score = minmax(minmax_board, depth + 1, True)
                    minmax_board[row][column] = 0
                    best_score = min(score, best_score)
        return best_score

def find_best_move():
    best_move = (-1, -1)
    best_value = -float('inf')
    for row in range(BOARD_ROWS):
        for column in range(BOARD_COLUMNS):
            if board[row][column] == 0:
                board[row][column] = 2
                move_value = minmax(board, 0, False)
                board[row][column] = 0
                if move_value > best_value:
                    best_value = move_value
                    best_move = (row, column)
    return best_move
```

```
def display_message(message, color):
    font = pygame.font.Font(None, 50)
    text = font.render(message, True, color)
    text_rect = text.get_rect(center=(WIDTH // 2, HEIGHT // 2))
    screen.blit(text, text_rect)
    pygame.display.update()

def main():
    global board
    draw_lines()
    pygame.display.update()

    while True:
        board = np.zeros((BOARD_ROWS, BOARD_COLUMNS))
        draw_lines()
        pygame.display.update()

        player_turn = True
        game_over = False

        while not game_over:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
                    sys.exit()

                if event.type == pygame.MOUSEBUTTONDOWN and player_turn and not game_over:
                    mouseX = event.pos[0] // SQUARE_SIZE
                    mouseY = event.pos[1] // SQUARE_SIZE
                    if available_square(mouseY, mouseX):
                        mark_square(mouseY, mouseX, 1)
                        if check_win(1):
                            game_over = True
                        elif is_board_full():
                            game_over = True
                        else:
                            player_turn = False

                if not player_turn and not game_over:
                    move = find_best_move()
                    if move != (-1, -1):
                        mark_square(move[0], move[1], 2)
                        if check_win(2):
                            game_over = True
                        elif is_board_full():
                            game_over = True
                    player_turn = True

                draw_figures()
                pygame.display.update()

            if game_over:
                screen.fill(GREY)
                draw_lines()
                draw_figures()
                if check_win(1):
                    display_message("Player X wins!", GREEN)
                elif check_win(2):
                    display_message("Player O wins!", RED)
                else:
                    display_message("It's a draw!", WHITE)
                pygame.time.wait(2000)

                screen.fill(BLACK)
                draw_lines()
                pygame.display.update()

if __name__ == "__main__":
    main()
```

Input and Output Section

Sample Inputs and Corresponding Outputs:

- **Sample Input:**
    - The game starts with an empty 3x3 Tic-Tac-Toe board. The player clicks on a square to place an 'X'.
- **Sample Output:**
    - The computer, using the Minimax algorithm, places an 'O' in a strategic position after the player's move.
    - If the player wins, the game displays "Player X wins!".
    - If the computer wins, the game displays "Player O wins!".
    - If the game ends in a draw, the game displays "It's a draw!".
- **Screenshot Output:**



Explanation of Input Processing and Output Generation:

1. **Input Processing:**
    - The game “listens” for mouse clicks from the player. When the player clicks on a square, the program checks if the square is available.
    - If the square is available, the player’s move ('X') is marked on the board.
    - The program then checks if this move has resulted in a win or if the board is full (resulting in a draw). If neither is true, it becomes the computer’s turn.
2. **Output Generation:**
    - On the computer’s turn, the Minimax algorithm is used to find the best possible move for the computer ('O').
    - The computer places its 'O' in the chosen square, and the board is updated.
    - After the computer’s move, the program checks if the computer has won or if the game is a draw.
    - Depending on the outcome, a message is displayed on the screen, such as "Player X wins!", "Player O wins!", or "It's a draw!".

Test Cases:

- **Standard Cases:**
    - **Case 1:** The game starts with an empty board. The player clicks on the center square, and the computer responds by placing an 'O' in a corner.
    - **Case 2:** The player has placed several 'X's, and the computer strategically places 'O's to block the player from winning.

- **Edge Cases:**
    - **Case 3:** The board is nearly full with only one move left. The computer should make the best move to either win or force a draw.
    - **Case 4:** The player is one move away from winning. The computer should block this move to prevent the loss.

Expected Results for Each Test Case:

- For each test case, the computer will make a move that either blocks the player from winning, attempts to win, or forces a draw. The game should correctly identify and display the outcome based on the moves made by both the player and the computer.

Performance Analysis

Completeness:

- The Minimax algorithm is complete for Tic-Tac-Toe. This means it will always find the best move possible given the current state of the game.

Cost Optimality:

- The Minimax algorithm is optimal in Tic-Tac-Toe. It always makes the move that gives the best possible outcome for the computer, whether that's a win or a draw.

Time Complexity:

- The time complexity of the Minimax algorithm for Tic-Tac-Toe is manageable because the board is small (3x3). The algorithm looks at all possible moves, which is feasible for a game of this size.

Space Complexity:

- The algorithm requires space to store all possible game states as it decides on the best move. In Tic-Tac-To, this space requirement is also manageable due to the small board size.

Discussion of Trade-offs:

- The main trade-off with the Minimax algorithm is between time and decision accuracy. because the algorithm takes time to analyze all possible moves, but it guarantees that the move that it makes is the best one. In larger, more complex games, this process can be way to slow, but in Tic-Tac-Toe, it works efficiently.

Error Handling

Explanation of Error Handling:

In our implementation of the Tic-Tac-Toe game, we have incorporated error handling to ensure that the program can easily manage any unexpected issues that may come up, usually during the initialization and setup.

1. **Initialization Error Handling:**

○ When the program starts, it attempts to initialize the Pygame library, which is needed for creating the game's graphical interface. To handle any potential errors during this process, we use a `try-except` block. If the initialization fails (for example, due to a system-specific issue or a missing Pygame installation), the program catches the error, prints an informative message, and exits safely.

Example:

python

Copy code

```
try:

    pygame.init()

except pygame.error as e:

    print(f"Failed to initialize Pygame: {e}")

    sys.exit()
```

2. **Display Setup Error Handling:**

- After initializing Pygame, the program sets up the game window where the Tic-Tac-Toe board will be displayed. Similar to the initialization process, setting up the display could encounter errors (for example due to hardware limitations or incorrect configurations). To manage this possible case, another `try-except` block is used around the display setup code. If an error occurs, the program will again catch the error, provide a descriptive message, and then exit.

Example:

python

Copy code

```
try:

    screen = pygame.display.set_mode((WIDTH, HEIGHT))

    pygame.display.set_caption('Tic Tac Toe Minmax')

except pygame.error as e:

    print(f"Failed to set up display: {e}")

    pygame.quit()

    sys.exit()
```

Examples of Error Scenarios and Their Handling:

- **Failed Pygame Initialization:**
    - If Pygame fails to initialize, the program prints a message like "Failed to initialize Pygame: error details" and exits. This prevents the program from continuing in an unstable state.
- **Failed Display Setup:**

- If the display window cannot be created, the program will output "Failed to set up display: error details" and quit Pygame before exiting, ensuring that resources are cleaned up properly.

Optimization

Optimization Techniques Applied: One common optimization for Minimax is using **alpha-beta pruning**, which reduces the number of states the algorithm needs to evaluate by cutting off branches that won't affect the final decision.

Comparison with Other Possible Approaches: Minimax is compared to simpler strategies like random placement or basic rule-based strategies, which are faster but less accurate.

Assumptions and Limitations

Assumptions Made During Implementation:

- The board is always a 3x3 grid.
- The computer always plays optimally, meaning it always makes the best possible move.
- The opponent may not always play optimally.

Known Limitations of the Solution:

- **Scalability:** The Minimax algorithm works well for Tic-Tac-Toe, but for larger, more complex games, it might be too slow without optimizations like alpha-beta pruning.
- **Opponent Behavior:** The algorithm assumes the opponent might make mistakes, but it always plays perfectly by itself.

Results and Conclusion

Summary of Results:

- The Minimax algorithm performs very well in Tic-Tac-To, consistently always finding the best possible moves. It can either win or draw every game, depending on the opponent's moves. During testing, it showed that it is a strong strategy for a game like Tic-Tac-Toe.

Final Thoughts on Effectiveness and Efficiency:

- The Minimax algorithm is a good choice for Tic-Tac-Toe because it guarantees the best possible outcome, whether that's a win or a draw. While it is computationally heavier than simpler strategies, the small size of the Tic-Tac-Toe board makes this a pretty easy game to handle. For more complex games, the algorithm might need optimizations like alpha-beta pruning to perform efficiently.