**ASSIGNMENT HELP**

**MANUAL**



SUBMITTED

TO

VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY, PUNE

FOR THE SKILL AND COMPETENCY EVALUATION OF

ARTIFICIAL INTELLIGENCE [CAUA31201]

IN

**CSE AI DEPARTMENT**

BY

**Vedant Rakesh Mukhekar**

**Class: T.Y. BTech Division: A Batch: A2**

**Batch Teacher**

**Dr. ANURADHA YENKIKAR.**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **SR. NO.** | **CONTENTS** | **PAGE NO.** |
| **1** | **PROBLEM STATEMENT** | **4-5** |
| **2** | **LIBRARY USED** | **5-6** |
| **3** | **THEORY** | **6-7** |
| **4** | **METHODOLOGY** | **7-14** |
| **5** | **ADVANTAGES & DISADVANTAGES** | **14-15** |
| **6** | **WORKING** | **16-17** |
| **7** | **DIAGRAM** | **17-18** |
| **8** | **CONCLUSION** | **18-19** |

### Problem Statement

The **Minimax algorithm** is a decision-making algorithm used in two-player games such as Tic-Tac-Toe, Chess, and Connect Four. The goal of this algorithm is to determine the best possible move for a player, assuming that both players play optimally.

The problem is to implement the **Minimax algorithm** for a simple two-player game (such as Tic-Tac-Toe), where one player tries to maximize their score (MAX), and the opponent tries to minimize the score (MIN). The algorithm explores all possible moves from the current state, evaluates them, and chooses the optimal move.

### Libraries Used

* **Python Standard Libraries**:
  + No external libraries are required for the basic implementation.

Optional libraries for GUI or visualization:

* **Pygame**: For creating a graphical interface for the game (optional).

### Theory

The **Minimax algorithm** is a recursive algorithm used for decision-making in game theory. It operates under the assumption that:

* The **MAX** player tries to maximize their score.
* The **MIN** player (the opponent) tries to minimize the score.

The algorithm recursively evaluates each possible move in the game tree, determining the best move for the MAX player by assuming that the opponent will also play optimally to minimize the score.

#### Key Concepts:

* **Maximizer (MAX)**: The player who tries to get the highest possible score.
* **Minimizer (MIN)**: The opponent, who tries to minimize the score.
* **Game Tree**: A tree where each node represents a state of the game. The root node represents the current state, and the children represent possible future states based on the player's moves.
* **Terminal State**: A state where the game ends (win, loss, or draw).

#### Minimax Recursion:

* The **Minimax algorithm** evaluates the best move by looking ahead and simulating all possible moves by both players.
* It assigns a score to each terminal state (win = +1, loss = -1, draw = 0) and recursively propagates these scores back through the game tree.

### Methodology

1. **Define the Game**:
   * Choose a simple two-player game such as Tic-Tac-Toe.
   * Define the game board and the rules for winning.
2. **Implement the Minimax Algorithm**:
   * Create a recursive function to evaluate all possible moves.
   * If the current state is a terminal state (win, loss, or draw), return the corresponding score.
   * If it’s the MAX player's turn, choose the move that maximizes the score. If it’s the MIN player’s turn, choose the move that minimizes the score.
3. **Integrate User Interaction**:
   * Allow players to make moves and display the game state after each turn.
   * Use the Minimax algorithm to determine the best move for the MAX player.

### Advantages & Disadvantages

* **Advantages**:
  + Provides optimal moves assuming both players play perfectly.
  + Simple to understand and implement for basic games.
* **Disadvantages**:
  + The time complexity can be high (O(b^d)), where b is the branching factor and d is the depth of the tree.
  + Inefficient for games with a large search space unless optimizations like alpha-beta pruning are implemented.

### Working Example (Python Code)

python

Copy code

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board):

for row in board:

if row[0] == row[1] == row[2] != " ":

return row[0]

for col in range(3):

if board[0][col] == board[1][col] == board[2][col] != " ":

return board[0][col]

if board[0][0] == board[1][1] == board[2][2] != " ":

return board[0][0]

if board[0][2] == board[1][1] == board[2][0] != " ":

return board[0][2]

return None

def is\_draw(board):

return all(cell != " " for row in board for cell in row)

def minimax(board, depth, is\_maximizing):

winner = check\_winner(board)

if winner == "X":

return 1

elif winner == "O":

return -1

elif is\_draw(board):

return 0

if is\_maximizing:

best\_score = float('-inf')

for i in range(3):

for j in range(3):

if board[i][j] == " ":

board[i][j] = "X" # MAX player's move

score = minimax(board, depth + 1, False)

board[i][j] = " "

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == " ":

board[i][j] = "O" # MIN player's move

score = minimax(board, depth + 1, True)

board[i][j] = " "

best\_score = min(score, best\_score)

return best\_score

def find\_best\_move(board):

best\_score = float('-inf')

move = (-1, -1)

for i in range(3):

for j in range(3):

if board[i][j] == " ":

board[i][j] = "X" # MAX player's move

score = minimax(board, 0, False)

board[i][j] = " "

if score > best\_score:

best\_score = score

move = (i, j)

return move

# Example usage

board = [

[" ", " ", " "],

[" ", " ", " "],

[" ", " ", " "]

]

while True:

print\_board(board)

if is\_draw(board):

print("It's a draw!")

break

row, col = find\_best\_move(board)

board[row][col] = "X" # MAX player's move

if check\_winner(board) == "X":

print\_board(board)

print("Player X wins!")

break

print\_board(board)

if is\_draw(board):

print("It's a draw!")

break

# Assume the MIN player is making a random move

import random

while True:

row, col = random.randint(0, 2), random.randint(0, 2)

if board[row][col] == " ":

board[row][col] = "O" # MIN player's move

break

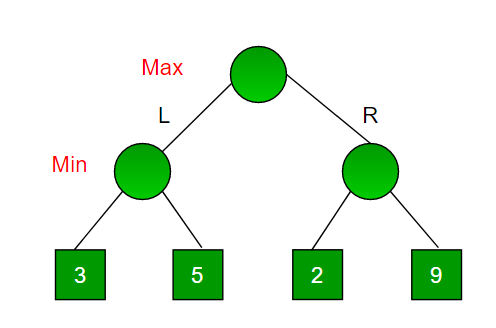
if check\_winner(board) == "O":

print\_board(board)

print("Player O wins!")

break

### Diagram



### Conclusion

The **Minimax algorithm** is a fundamental technique in artificial intelligence for game playing. It efficiently determines the best move by evaluating all possible outcomes of a game, assuming optimal play from both sides. This implementation provides a basic structure for a two-player game and can be extended to more complex games with additional features like alpha-beta pruning for improved efficiency.