**Minimax algorithm**

**Code:**

import math

# Initialize the board

board = [' ' for \_ in range(9)]

# Print the board

def print\_board():

    for i in range(3):

        print('|'.join(board[i\*3:(i+1)\*3]))

        if i < 2:

            print('-' \* 5)

# Check for a winner

def check\_winner(brd, player):

    win\_conditions = [

        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows

        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # cols

        [0, 4, 8], [2, 4, 6]              # diagonals

    ]

    return any(all(brd[i] == player for i in cond) for cond in win\_conditions)

# Check for tie

def is\_full(brd):

    return all(spot != ' ' for spot in brd)

# Minimax algorithm

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

    if check\_winner(brd, 'O'):

        return 1

    if check\_winner(brd, 'X'):

        return -1

    if is\_full(brd):

        return 0

    if is\_maximizing:

        best\_score = -math.inf

        for i in range(9):

            if brd[i] == ' ':

                brd[i] = 'O'

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

                brd[i] = ' '

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

        return best\_score

    else:

        best\_score = math.inf

        for i in range(9):

            if brd[i] == ' ':

                brd[i] = 'X'

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

                brd[i] = ' '

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

        return best\_score

# Get the best move for AI

def best\_move():

    best\_score = -math.inf

    move = -1

    for i in range(9):

        if board[i] == ' ':

            board[i] = 'O'

            score = minimax(board, 0, False)

            board[i] = ' '

            if score > best\_score:

                best\_score = score

                move = i

    return move

# Main game loop

def play\_game():

    print("Tic-Tac-Toe (You = X, AI = O)")

    print\_board()

    while True:

        # Player move

        move = int(input("Enter your move (0-8): "))

        if board[move] != ' ':

            print("Invalid move, try again.")

            continue

        board[move] = 'X'

        print\_board()

        if check\_winner(board, 'X'):

            print("You win!")

            break

        if is\_full(board):

            print("It's a tie!")

            break

        # AI move

        ai\_move = best\_move()

        board[ai\_move] = 'O'

        print("\nAI played:")

        print\_board()

        if check\_winner(board, 'O'):

            print("AI wins!")

            break

        if is\_full(board):

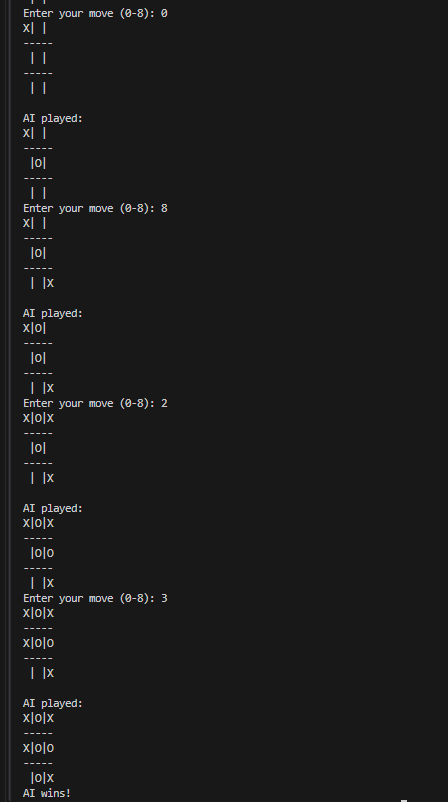
            print("It's a tie!")

            break

# Run the game

play\_game()

**Output:**



**Alpha-Beta Pruning**

**Code:**

import math

# Initialize the board

board = [' ' for \_ in range(9)]

# Print the board

def print\_board():

    for i in range(3):

        print('|'.join(board[i\*3:(i+1)\*3]))

        if i < 2:

            print('-' \* 5)

# Check for a winner

def check\_winner(brd, player):

    win\_conditions = [

        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows

        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # cols

        [0, 4, 8], [2, 4, 6]              # diagonals

    ]

    return any(all(brd[i] == player for i in cond) for cond in win\_conditions)

# Check if board is full (draw)

def is\_full(brd):

    return all(spot != ' ' for spot in brd)

# Minimax with Alpha-Beta Pruning

def minimax\_ab(brd, depth, is\_maximizing, alpha, beta):

    if check\_winner(brd, 'O'):

        return 1

    if check\_winner(brd, 'X'):

        return -1

    if is\_full(brd):

        return 0

    if is\_maximizing:

        max\_eval = -math.inf

        for i in range(9):

            if brd[i] == ' ':

                brd[i] = 'O'

                eval = minimax\_ab(brd, depth + 1, False, alpha, beta)

                brd[i] = ' '

                max\_eval = max(max\_eval, eval)

                alpha = max(alpha, eval)

                if beta <= alpha:

                    break  # Beta cutoff

        return max\_eval

    else:

        min\_eval = math.inf

        for i in range(9):

            if brd[i] == ' ':

                brd[i] = 'X'

                eval = minimax\_ab(brd, depth + 1, True, alpha, beta)

                brd[i] = ' '

                min\_eval = min(min\_eval, eval)

                beta = min(beta, eval)

                if beta <= alpha:

                    break  # Alpha cutoff

        return min\_eval

# Best move using alpha-beta pruning

def best\_move\_ab():

    best\_score = -math.inf

    move = -1

    for i in range(9):

        if board[i] == ' ':

            board[i] = 'O'

            score = minimax\_ab(board, 0, False, -math.inf, math.inf)

            board[i] = ' '

            if score > best\_score:

                best\_score = score

                move = i

    return move

# Main game loop

def play\_game():

    print("Tic-Tac-Toe (You = X, AI = O)")

    print\_board()

    while True:

        # Player move

        try:

            move = int(input("Enter your move (0-8): "))

            if board[move] != ' ':

                print("Invalid move, try again.")

                continue

        except (ValueError, IndexError):

            print("Invalid input. Enter a number between 0 and 8.")

            continue

        board[move] = 'X'

        print\_board()

        if check\_winner(board, 'X'):

            print("You win!")

            break

        if is\_full(board):

            print("It's a tie!")

            break

        # AI move

        ai\_move = best\_move\_ab()

        board[ai\_move] = 'O'

        print("\nAI played:")

        print\_board()

        if check\_winner(board, 'O'):

            print("AI wins!")

            break

        if is\_full(board):

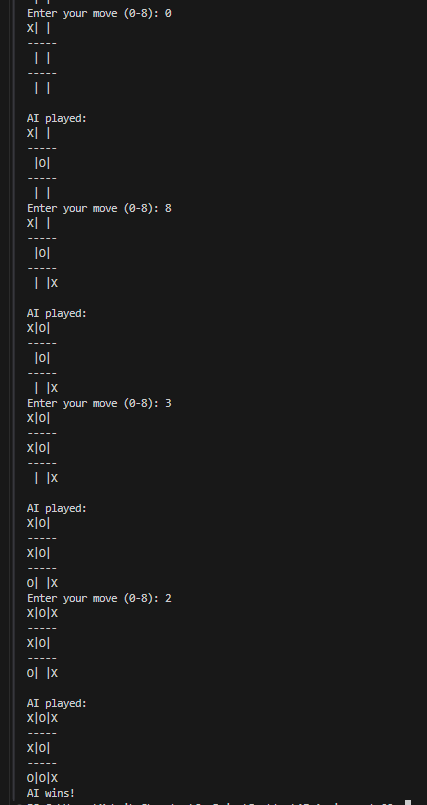
            print("It's a tie!")

            break

# Run the game

play\_game()

**Output:**



**Comparison**

**1. Search Space Explored**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Standard Minimax** | **Alpha-Beta Pruning** |
| Nodes Explored | Explores all possible moves in the game tree | Skips branches that cannot influence the outcome |
| Efficiency | Less efficient due to exhaustive search | More efficient by eliminating unnecessary nodes |
| Time Complexity | O(b^d), where *b* is branching factor and *d* is depth | O(b^(d/2)) in the best case (with perfect ordering) |

**2. Performance in Practice**

|  |  |  |
| --- | --- | --- |
| **Factor** | **Standard Minimax** | **Alpha-Beta Pruning** |
| Game Response Speed | Slower due to evaluating all options | Faster due to reduced evaluations |
| Suitability for Deep Trees | Limited effectiveness | Highly suitable for deeper game trees |
| Memory Usage | Higher due to larger call stack | Lower due to fewer recursive calls |

**3. Application Example: Tic-Tac-Toe**

* Standard Minimax may evaluate up to 150+ nodes per move.
* Alpha-Beta Pruning can reduce the number of evaluated nodes to approximately 30–40, depending on the move ordering.

**Summary of Comparison**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Standard Minimax** | **Alpha-Beta Pruning** |
| Speed | Slower | Faster |
| Optimization | Not optimized | Prunes unneeded branches |
| Real-Time Game Suitability | Less suitable | Highly suitable |
| Depth Reach within Same Time | Shallower | Deeper |
| Optimality | Maintains optimality | Maintains optimality |