Tic-Tac-Toe

A PROJECT FILE
for
INTRODUCTION TO AI (AI201B)
Session (2024-25)

Submitted by

Kiran Yadav
202410116100102
Dolly Prajapati
202410116100069
Khushi Kumari
2024101161000101
Gulshan Kumari
202410116100078

Submitted in partial fulfilment of the
Requirements for the Degree of

MASTER OF COMPUTER APPLICATION

Under the Supervision of
Mr. APOORV JAIN
Assistant Professor

Submitted to

DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206
(APRIL- 2025)

TABLE OF CONTENT

1. Introduction	**3**
2. Methodology	**4-5**
3. Code	**6-8**
4. Output	**9**
5. Conclusion	**10**

INTRODUCTION

Tic-Tac-Toe is a well-known two-player strategy game played on a 3×3 grid. The game is simple yet offers deep strategic elements, making it a popular choice for demonstrating artificial intelligence (AI) techniques. The primary objective is for a player to align three of their symbols (‘X’ or ‘O’) in a row, column, or diagonal before their opponent does. If no player achieves this and the board is filled, the game ends in a draw.

This project focuses on developing an **AI-powered Tic-Tac-Toe solver** that enables a human player to compete against an intelligent artificial opponent. The AI is designed to make optimal moves using the **Minimax Algorithm with Alpha-Beta Pruning**, ensuring strategic decision-making and efficient gameplay.

The system follows a structured approach, starting with **board representation and user input handling**, ensuring a smooth interaction between the player and the AI. The game continuously evaluates the board state, checking for win conditions or a draw after each move. The **AI decision-making process** involves analyzing all possible moves and selecting the most optimal one, preventing the human player from winning while maximizing its chances of success.

A key feature of this implementation is the **Alpha-Beta Pruning technique**, which optimizes the Minimax Algorithm by eliminating unnecessary computations, making the AI more efficient and responsive. The game operates in a **loop until a winner is determined or the board is full**, after which the player is given the option to restart or exit.

METHODOLOGY

The Tic-Tac-Toe solver follows a structured approach to determine the best possible moves and play the game efficiently. The methodology consists of the following key steps:

1. Board Representation and Display

- The game board is represented as a **3x3 grid** using a **2D list**, where each cell can contain an 'X', 'O', or remain empty.
- After every move, the board is displayed to reflect the latest state of the game, allowing players to see their progress.
- A clean and user-friendly display format ensures clarity and ease of interaction.

2. Player Turns and Input Handling

- The **human player (O)** makes a move by entering the **row and column indices (0-2)** to select a position.
- Input validation is implemented to ensure:
  - The entered values fall within the valid range.
  - The selected cell is **unoccupied**, preventing overwrites.
- If an invalid move is entered, the program prompts the user to re-enter a valid choice.

3. Win and Draw Detection

- After each move, the game checks for a winning condition:
  - If any **row, column, or diagonal** contains the same symbol ('X' or 'O'), that player wins.
  - If all cells are filled **without a winner**, the game results in a **draw**.
- The program efficiently evaluates board states to ensure immediate detection of game-ending conditions.

4. AI Decision-Making Using the Minimax Algorithm

- The AI opponent (playing as 'X') uses the **Minimax Algorithm**, a recursive decision-making strategy that evaluates all possible future moves.
- The AI assumes:
  - It should **maximize** its score (trying to win).
  - The human opponent will **minimize** the AI's score (trying to win themselves).
- The algorithm assigns scores to different outcomes:
  - **+10** if the AI **wins**.
  - **-10** if the **human wins**.
  - **0** for a **draw**.
- **Alpha-Beta Pruning** is integrated into the Minimax Algorithm to improve efficiency by eliminating unnecessary calculations, significantly reducing computational time.

5. AI Move Selection

- The AI evaluates **all possible moves** using the Minimax Algorithm.
- It selects the move that yields the **highest possible score**, ensuring an optimal decision.
- Once the AI determines the best move, it places its **'X'** symbol in the chosen position and updates the board accordingly.

6. Game Loop and User Interaction

- The game runs in a continuous loop until one of the following occurs:
  - A **player wins**, triggering the game to display the winner.
  - The board **fills up completely**, resulting in a **draw**.
- At the end of the game, the player is given the option to:
  - **Restart the game** for another round.
  - **Exit the program** if they choose to stop playing.

CODE

```
import math

PLAYER_X = 'X'  # AI
PLAYER_O = 'O'  # Human
EMPTY = ' '

def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def is_winner(board, player):
    """Checks if a player has won the game."""
    for row in board:
        if all(cell == player for cell in row):
            return True

    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True

    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in
range(3)):
        return True

    return False

def is_draw(board):
    """Checks if the game is a draw."""
    return all(board[row][col] != EMPTY for row in range(3) for col in range(3))

def evaluate(board):
    """Evaluates the board state."""
    if is_winner(board, PLAYER_X):
        return 10
    if is_winner(board, PLAYER_O):
        return -10
    return 0

def minimax(board, depth, is_maximizing, alpha, beta):
    """Minimax algorithm with alpha-beta pruning."""
    score = evaluate(board)

    if score == 10 or score == -10:
```

```
        return score - depth if score == 10 else score + depth

    if is_draw(board):
        return 0

    if is_maximizing:
        max_eval = -math.inf
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_X
                    eval = minimax(board, depth + 1, False, alpha, beta)
                    board[row][col] = EMPTY
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
        return max_eval

    else:
        min_eval = math.inf
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_O
                    eval = minimax(board, depth + 1, True, alpha, beta)
                    board[row][col] = EMPTY
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
        return min_eval

def best_move(board):
    """Finds the best move for the AI."""
    best_val = -math.inf
    move = (-1, -1)

    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_X
                move_val = minimax(board, 0, False, -math.inf, math.inf)
                board[row][col] = EMPTY

                if move_val > best_val:
                    best_val = move_val
                    move = (row, col)

    return move
```

```
def play():
    """Runs a Tic-Tac-Toe game between a human and AI."""
    board = [[EMPTY] * 3 for _ in range(3)]
    print("Tic-Tac-Toe: AI (X) vs. You (O)")
    print_board(board)

    while True:
        row, col = map(int, input("Enter your move (row and column 0-2): ").split())
        if board[row][col] != EMPTY:
            print("Invalid move! Try again.")
            continue

        board[row][col] = PLAYER_O
        if is_winner(board, PLAYER_O) or is_draw(board):
            break

        ai_row, ai_col = best_move(board)
        board[ai_row][ai_col] = PLAYER_X
        print_board(board)

play()
```

OUTPUT

```
Tic-Tac-Toe: AI (X) vs. You (O)
  |   |
---------
  |   |
---------
  |   |
---------
Enter your move (row and column 0-2): 2 0

AI plays:
  |   |
---------
  | X |
---------
O |   |
---------
Enter your move (row and column 0-2): 0 1

AI plays:
X | O |
---------
  | X |
---------
O |   |
---------
Enter your move (row and column 0-2): 2 2
```

```
AI plays:
X | O |
---------
  | X |
---------
O | X | O
---------
Enter your move (row and column 0-2): 0 2

AI plays:
X | O | O
---------
  | X | X
---------
O | X | O
---------
Enter your move (row and column 0-2): 1 0
X | O | O
---------
O | X | X
---------
O | X | O
---------
It's a draw! 🤝
```

Code Explanation:

1.**Importing the Required Library**

- import math

The math module is imported to use math.inf (positive and negative infinity) in the Minimax function.

2. Defining Constants

```
PLAYER_X = 'X'  # AI
PLAYER_O = 'O'  # Human
EMPTY = ''     # Empty cell
```

Defines constants:

- `PLAYER_X`: Represents the AI player using `'X'`.
- `PLAYER_O`: Represents the human player using `'O'`.
- `EMPTY`: Represents an empty space on the board using `' '`.

3. Function to Print the Tic-Tac-Toe Board

```
def print_board(board)

  """Prints the Tic-Tac-Toe board.""

for row in board:

    print(" | ".join(row))

    print("-" * 9)  # Prints a separator line
```

- Iterates through each row in the board and prints it.
- Uses " | ".join(row) to format the board.
- Adds a separator line ("-" * 9) between rows

Example Output:

```
X | O | X

---------

O | X | O

---------

X | O | X
```

4. Function to Check If a Player Has Won

```
def is_winner(board, player):
    """Checks if a player has won the game."""
```

- This function checks if a given player (X or O) has won the game.

```
for row in board:
        if all(cell == player for cell in row):
            return True
```

- Checks if **any row** contains three identical symbols (X or O).

```
for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
```

- Checks if any column contains three identical symbols.

```
if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in
range(3)):
        return True
```

- checks if **either diagonal** contains three identical symbols.
- return False
- If no winning condition is met, return False.

---

5. Function to Check If the Game is a Draw

```
def is_draw(board):
    """Checks if the game is a draw."""
    return all(board[row][col] != EMPTY for row in range(3) for col in range(3))
```

- Returns True if **all** cells are filled (not EMPTY), meaning the board is full and no one has won.

---

6. Function to Evaluate the Board

```
def evaluate(board):
    """Evaluates the board state."""
    if is_winner(board, PLAYER_X):
        return 10
    if is_winner(board, PLAYER_O):
```

```
        return -10
    return 0
```

- Assigns **+10 if AI (X) wins**, **-10 if the human (O) wins**, and **0 if neither**.

---

7. Minimax Algorithm with Alpha-Beta Pruning

```
def minimax(board, depth, is_maximizing, alpha, beta):
    """Minimax algorithm with alpha-beta pruning."""
```

- This function **recursively explores** all possible moves to determine the best one.

```
    score = evaluate(board)
```

- Calls evaluate() to check if the game has been won or lost.

```
    if score == 10 or score == -10:
        return score - depth if score == 10 else score + depth
```

- If AI wins, returns 10 - depth (depth reduces score to favor quicker wins).
- If human wins, returns -10 + depth (depth reduces penalty to favor slower losses).

```
    if is_draw(board):
        return 0
```

- If it's a draw, return 0.

Minimax for AI (Maximizing Player)

```
    if is_maximizing:
        max_eval = -math.inf
```

- AI is maximizing, so initialize max_eval to negative infinity.

```
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
```

- Loops through all empty cells to explore possible moves.

```
                    board[row][col] = PLAYER_X
                    eval = minimax(board, depth + 1, False, alpha, beta)
                    board[row][col] = EMPTY
```

- AI makes a move, calls minimax() **recursively**, then undoes the move (**backtracking**).

```
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
```

```
                break
```

- Updates max_eval, **prunes** unnecessary recursion if beta ≤ alpha.

Minimax for Human (Minimizing Player)

```
    else:
        min_eval = math.inf
```

- The human is minimizing, so initialize min_eval to **positive infinity**.

```
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
```

- Loops through empty cells to explore moves.

```
                    board[row][col] = PLAYER_O
                    eval = minimax(board, depth + 1, True, alpha, beta)
                    board[row][col] = EMPTY
```

- Human makes a move, calls minimax() **recursively**, then undoes the move (**backtracking**).

```
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
```

- Updates min_eval, **prunes** unnecessary recursion if beta ≤ alpha.

---

8. Function to Find the Best Move

```
def best_move(board):
    """Finds the best move for the AI."""
    best_val = -math.inf
    move = (-1, -1)
```

- Initializes best_val to negative infinity and move to an invalid position.

```
    for row in range(3):
        for col in range(3):
            if board[row][col] == EMPTY:
```

- Iterates through all empty cells.

```
                board[row][col] = PLAYER_X
                move_val = minimax(board, 0, False, -math.inf, math.inf)
                board[row][col] = EMPTY
```

- Simulates a move, calls minimax(), and undoes the move.

```
if move_val > best_val:
    best_val = move_val
    move = (row, col)
```

- Updates best_val and move if a better move is found.

---

9. Game Loop

```
def play():
    """Runs a Tic-Tac-Toe game between a human and AI."""
    board = [[EMPTY] * 3 for _ in range(3)]
```

- Initializes an **empty** 3x3 board.

```
    while True:
        row, col = map(int, input("Enter your move (row and column 0-2): ").split())
```

- Takes input from the human.

```
        if board[row][col] != EMPTY:
            print("Invalid move! Try again.")
            continue
```

- Ensures the move is valid.

```
        board[row][col] = PLAYER_O
```

- Updates the board with the human's move.

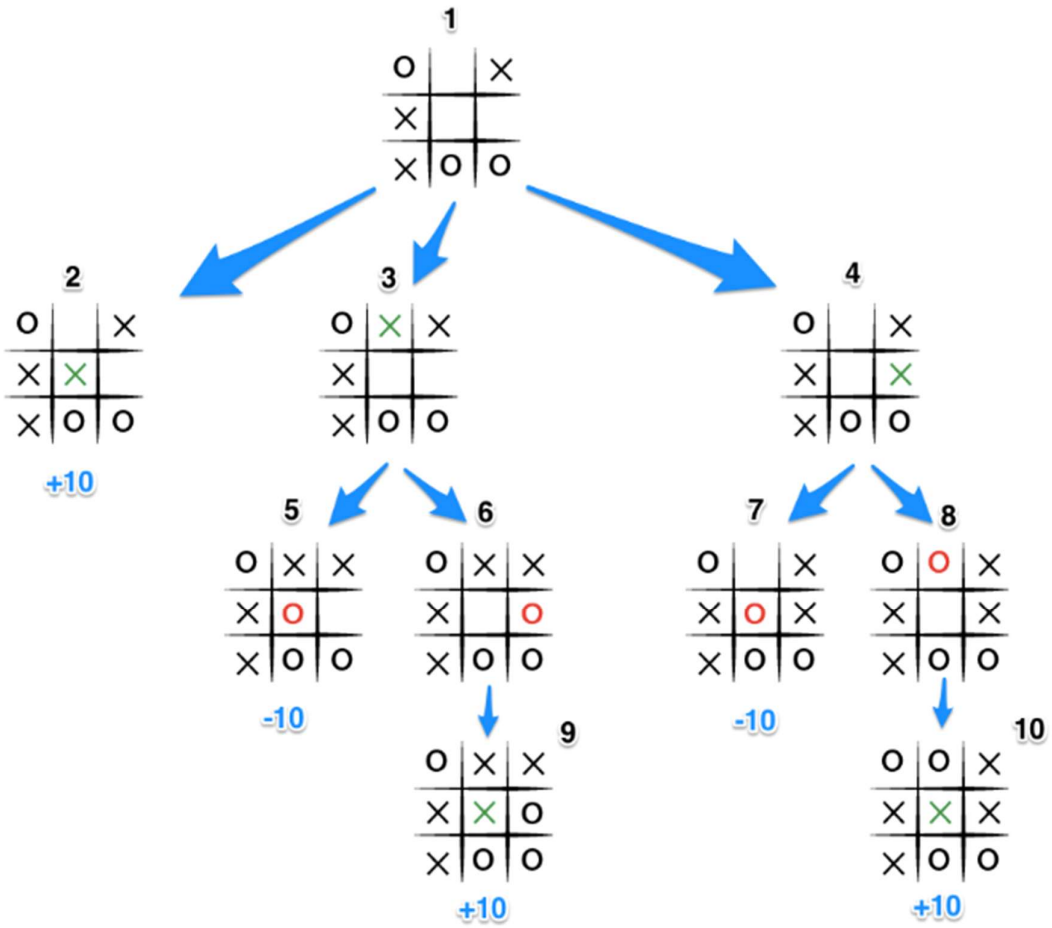
```
        ai_row, ai_col = best_move(board)
        board[ai_row][ai_col] = PLAYER_X
```

- AI calculates and makes the best move.

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

- Runs the game when executed.

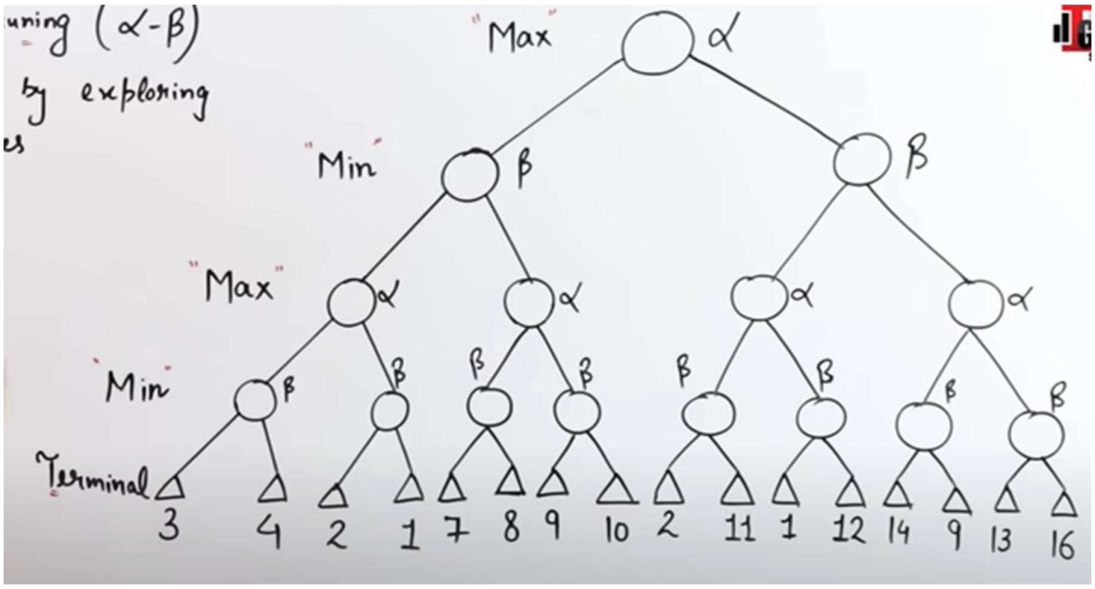
How Minimax Algorithm Works

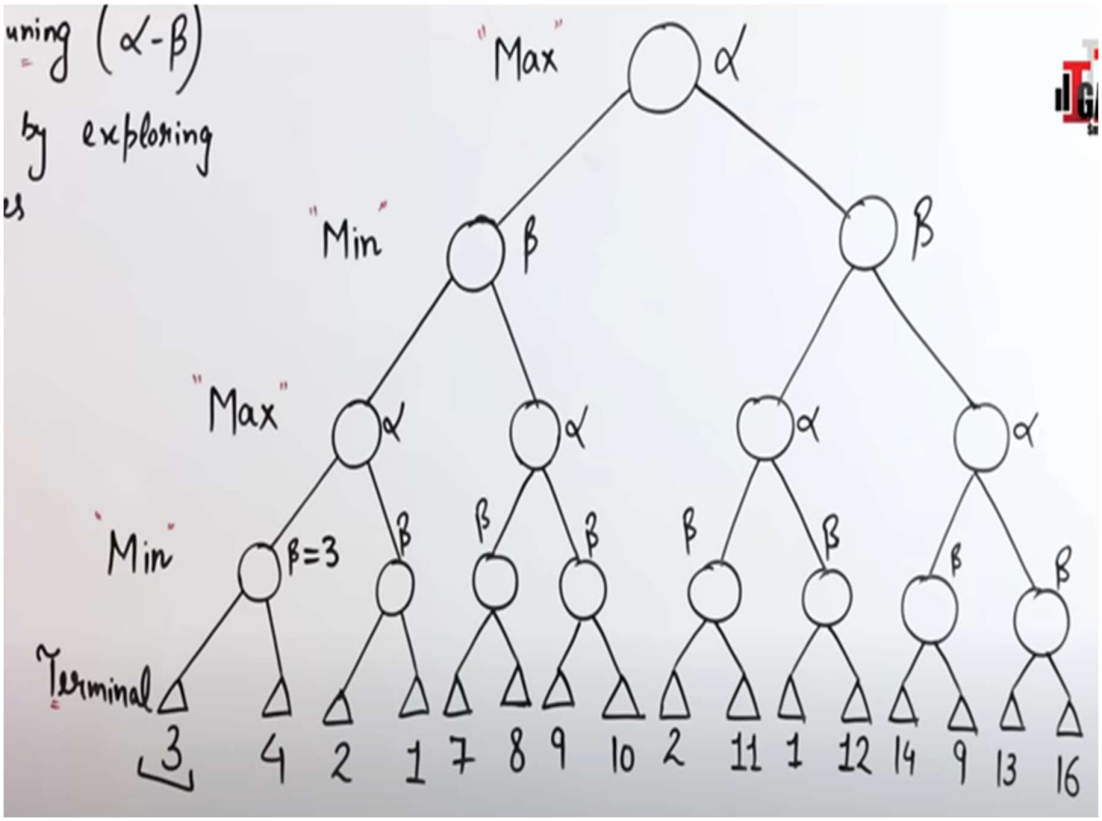


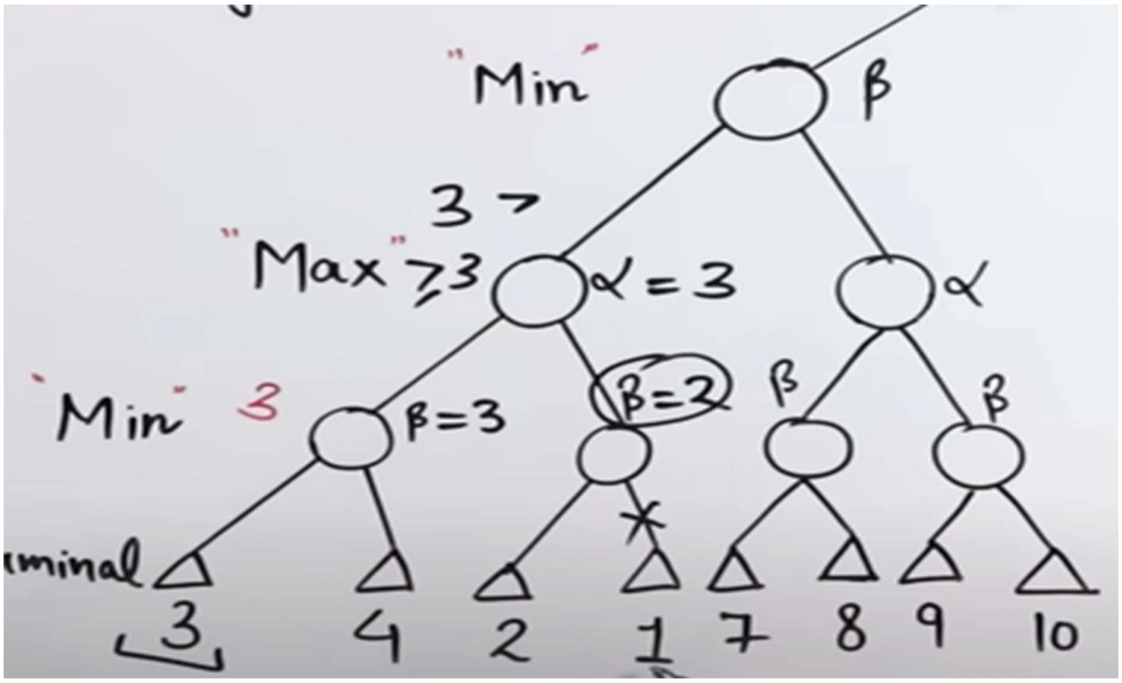
Explanation:

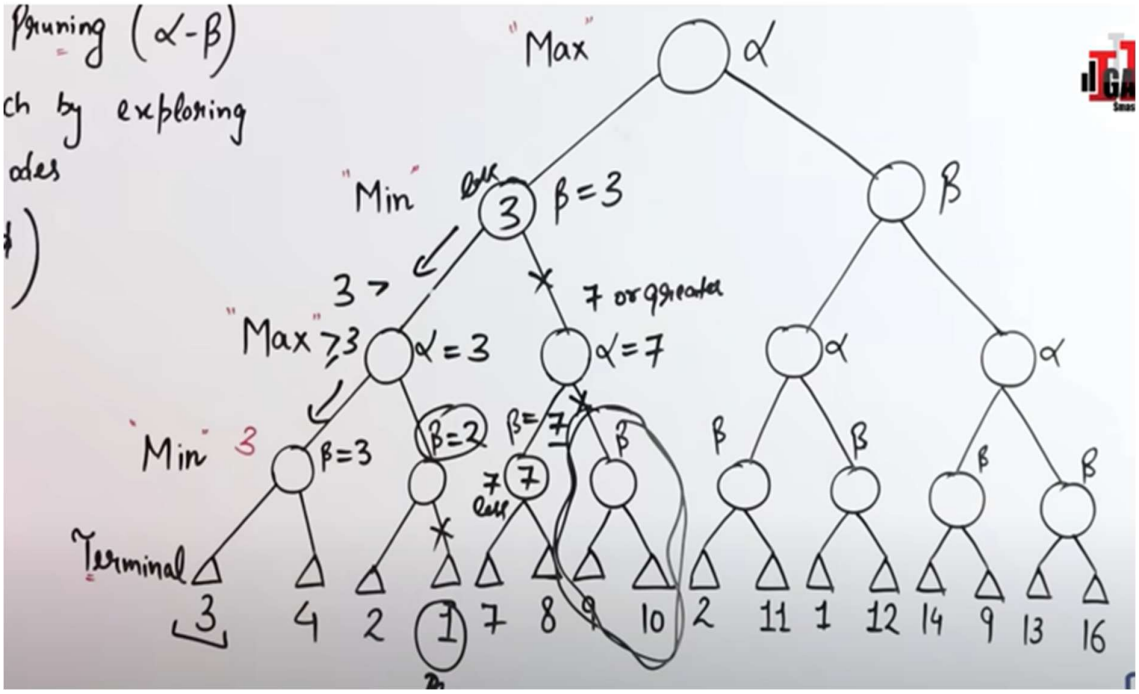
- It's X's turn in state 1. X generates the states 2, 3, and 4 and calls minimax on those states.
- State 2 pushes the score of +10 to state 1's score list, because the game is in an end state.
- State 3 and 4 are not in end states, so 3 generates states 5 and 6 and calls minimax on them, while state 4 generates states 7 and 8 and calls minimax on them.
- State 5 pushes a score of -10 onto state 3's score list, while the same happens for state 7 which pushes a score of -10 onto state 4's score list.
- State 6 and 8 generate the only available moves, which are end states, and so both of them add the score of +10 to the move lists of states 3 and 4.
- Because it is O's turn in both state 3 and 4, O will seek to find the minimum score, and given the choice between -10 and +10, both states 3 and 4 will yield -10.
- Finally the score list for states 2, 3, and 4 are populated with +10, -10 and -10 respectively, and state 1 seeking to maximize the score will chose the winning move with score +10, state 2.

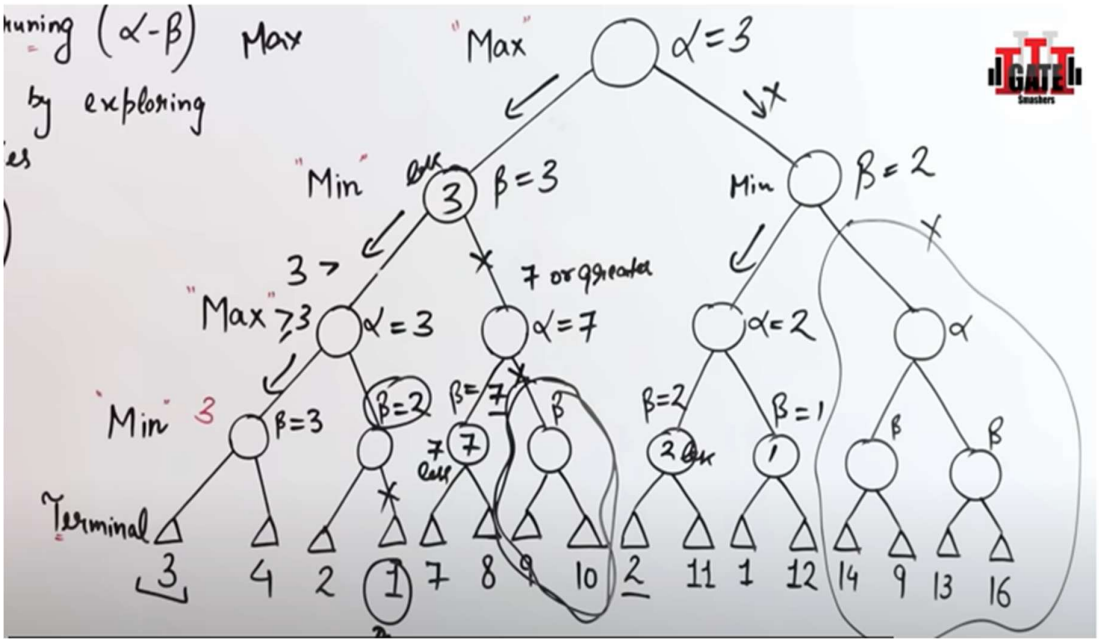
How Alpha-Beta Pruning Works






"Min"
β
3 >
"Max" ≥3
α=3
α
"Min" 3
β=3
β=2
β
β
minal
3
4
2
1
7
8
9
10


Pruning (α-β)
ch by exploring
odes
"Max"
α
"Min"
3
β=3
β
3 >
7 or greater
"Max" ≥3
α=3
α=7
α
α
"Min" 3
β=3
β=2
β=7
β
β
β
β
β
7
7
Terminal
3
4
2
1
7
8
9
10
2
11
1
12
14
9
13
16

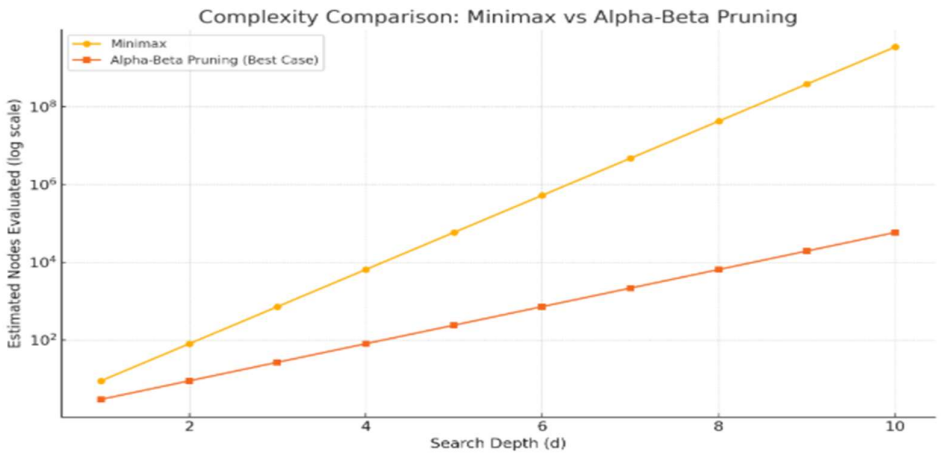

uning (α-β) Max
by exploring
es
"Max"
α=3
"Min"
3
β=3
Min
β=2
3 >
7 or greater
"Max" ≥3
α=3
α=7
α=2
α
"Min" 3
β=3
β=2
β=7
β
β=2
β=1
β
β
Terminal
3
4
2
1
7
8
9
10
2
11
1
12
14
9
13
16

Time Complexity: Minimax vs Alpha-Beta Pruning

Algorithm	Time Complexity (Worst Case)	Reason
Minimax	O(b^d)	Explores every node in the game tree (brute-force).
Alpha-Beta Pruning	O(b^(d/2)) (Best case)	Prunes away irrelevant branches, reducing total nodes.

Where:

- **b = branching factor (number of legal moves per turn)**
- **d = depth of the game tree (number of moves left until the game ends)**



CONCLUSION

This project successfully implements an **AI-powered Tic-Tac-Toe solver**, demonstrating the application of **artificial intelligence in strategic decision-making**. Through the integration of the **Minimax Algorithm with Alpha-Beta Pruning**, the AI is capable of analyzing potential game states, predicting the best possible moves, and responding intelligently to human actions. By optimizing decision-making through Alpha-Beta Pruning, the AI is able to eliminate unnecessary computations, significantly improving efficiency while maintaining an optimal strategy.

The structured methodology adopted in this project ensures a smooth and interactive gameplay experience. The board representation, user input validation, game state evaluation, and AI-driven move selection work seamlessly together, enabling an engaging and competitive game environment. Players can challenge the AI, observe its strategic approach, and experience how artificial intelligence is applied in decision-making scenarios.

Beyond its technical implementation, this project holds **educational, practical, and research significance**. From an educational standpoint, it serves as an **introductory model for AI in gaming**, helping learners understand key concepts such as **game trees, heuristic evaluation, and algorithmic optimization**. Practically, the project demonstrates how **decision-making algorithms can be optimized for real-world applications**, which is particularly relevant in fields such as **robotics, autonomous systems, and competitive AI gaming**. Additionally, this project provides a foundational study in **game theory and AI problem-solving**, offering a stepping stone for more advanced AI-driven applications.

The insights gained from this project can be extended to more complex **strategic games such as Chess, Connect Four, and Go**, where AI plays an increasingly critical role. Furthermore, the Minimax Algorithm and its variations can be adapted for **real-world decision-making problems** beyond gaming, including **automated planning, financial modeling, and cybersecurity strategies**.

In conclusion, this project not only provides a **challenging and interactive AI opponent for Tic-Tac-Toe** but also serves as a **valuable case study in artificial intelligence and game theory**. It highlights the effectiveness of **strategic AI decision-making**, showcasing how **mathematical models and computational algorithms can be used to create intelligent systems**. The success of this project encourages further exploration into AI's role in problem-solving, paving the way for more sophisticated applications in both gaming and real-world decision-making.