

# Kiet Group of Institutions



## Problem Statement-Noughts and Crosses with Alpha-Beta Pruning

Submitted to-Mr Abhishek Shukla

Submitted by-

Name-Prakhar Tiwari

Class Roll no-65

University roll no-202401100400138

Branch-CSEAIML

Section-B

# Introduction

## The Game: Noughts and Crosses (Tic-Tac-Toe)

The game of Tic-Tac-Toe (also known as Noughts and Crosses) is played on a 3x3 grid. Two players take turns marking a cell on the grid with their respective symbols:

- **Player X** (human) uses the symbol 'X'.
- **Player O** (AI) uses the symbol 'O'.

The objective of the game is to get three of your symbols (X or O) in a row—either horizontally, vertically, or diagonally.

## The Problem: AI (Player O) vs Human (Player X)

The challenge in this case is creating an AI (for Player O) that can make intelligent decisions and play optimally using a technique called **Minimax** with **Alpha-Beta Pruning**.

## Key Concepts:

### 1. Minimax Algorithm:

The **Minimax algorithm** is a decision-making algorithm used in two-player games like Tic-Tac-Toe. The goal is to simulate every possible game state and find the optimal move for the current player by assuming that both players play optimally.

Here's how it works:

- **Maximizing Player** (AI, Player O) wants to maximize their chances of winning.

- **Minimizing Player** (Human, Player X) wants to minimize the AI's chances of winning.

The algorithm simulates all possible moves and assigns scores to the board positions:

- Positive scores indicate a win for the maximizing player (AI).
- Negative scores indicate a win for the minimizing player (human).
- A score of 0 indicates a draw.

## 2. Alpha-Beta Pruning:

**Alpha-Beta Pruning** is an optimization technique for the Minimax algorithm. It helps cut down the number of nodes that need to be explored by "pruning" branches that are unlikely to be chosen.

- **Alpha** is the best value the maximizer can guarantee so far (the best score that the AI can achieve).
- **Beta** is the best value the minimizer can guarantee so far (the best score that the human player can achieve).

Whenever we encounter a branch where the score is worse than the current alpha or beta, we can stop exploring that branch since it will not affect the final decision.

# Methodology

## 1. The Problem:

The task is to create an AI (Player O) that can play Tic-Tac-Toe optimally against a human player (Player X). The AI should make decisions using a strategy that guarantees the best possible outcome (win, or at least draw). To achieve this, we need to evaluate all possible future game states and choose the best move for the AI at each point in the game.

## 2. Game Rules Recap:

- **Players:** There are two players: Player X (human) and Player O (AI).
- **Objective:** Players take turns to place their respective symbols ('X' or 'O') on a 3x3 grid. The objective is to get three of your symbols in a row (horizontally, vertically, or diagonally).

The AI needs to:

- Maximize its own chances of winning (maximize its score).
- Minimize the human player's chances of winning (minimize the human player's score).

## 3. How the AI Should Play:

To make intelligent decisions, the AI needs to simulate all possible moves it could make, as well as the possible moves the human might make in response. It will "think ahead" by

considering every possible future game state and evaluating the outcomes.

The **Minimax algorithm** is used to simulate all possible future game states. The AI then chooses the best possible move by selecting the one that leads to the most favourable outcome.

#### **4. Minimax Algorithm:**

The **Minimax algorithm** is a classic algorithm used in two-player games like Tic-Tac-Toe. The idea is to recursively simulate all possible future moves and assign scores to these positions. Here's how it works:

- **Maximizing Player (AI):** The AI tries to maximize its score (the best possible outcome for Player O).
- **Minimizing Player (Human):** The human tries to minimize the AI's score (the best possible outcome for Player X).

The **Minimax Algorithm** proceeds as follows:

1. It starts at the current board state.
2. It simulates all possible moves for the current player (AI or human).
3. For each possible move, it recursively evaluates the resulting board state until the game ends (either a win, loss, or draw).
4. The algorithm assigns a score based on the outcome of the game:

- Positive score (e.g., +10) if the AI wins.
  - Negative score (e.g., -10) if the human wins.
  - Zero score (0) if the game ends in a draw.
5. The AI chooses the move that maximizes its score, and the human chooses the move that minimizes the AI's score.

## 5. Alpha-Beta Pruning:

**Alpha-Beta Pruning** is an optimization technique applied to the **Minimax algorithm** to reduce the number of nodes that need to be explored. It "prunes" (cuts off) branches of the game tree that will not affect the final decision. This reduces the search space, making the algorithm faster.

Here's how it works:

- **Alpha** is the best value the maximizing player (AI) can guarantee so far.
- **Beta** is the best value the minimizing player (human) can guarantee so far.

When searching through the game tree:

- If the AI's score at a node is better than a previously explored node (alpha), the AI will discard other branches.
- If the human's score at a node is worse than a previously explored node (beta), the AI will discard other branches.

- This process effectively reduces the number of unnecessary nodes that need to be evaluated.

## **6. Steps in the Code:**

Here's how the solution implements the above approach:

### **1. Board Representation:**

- The Tic-Tac-Toe board is represented as a 3x3 grid, which is a list of lists in Python. Each cell of the board can be empty (' '), contain an 'X' (Player X), or contain an 'O' (Player O).

### **2. Checking for a Winner:**

- We need to check if either player has won the game. This is done by checking all rows, columns, and both diagonals for three matching symbols ('X' or 'O').

### **3. Minimax Function:**

- The minimax function is called recursively to explore all possible game states. It takes in the following parameters:
  - board: The current state of the board.
  - depth: The current depth of recursion (used to prioritize earlier moves).
  - alpha: The best score the AI can guarantee so far.

- beta: The best score the human player can guarantee so far.
- Maximizing player: A Boolean flag indicating whether the current player is the AI (maximizing) or the human (minimizing).
- The function evaluates the board and returns the score of the best possible move for the current player.

#### **4. Best Move Calculation:**

- The AI uses the best move function to determine its optimal move by calling the minimax function for each possible move on the board. The AI selects the move with the highest score (best move).

#### **5. Game Flow:**

- The game alternates between the AI (O) and the human (X).
- The AI calculates its best move using the minimax algorithm, while the human player enters their move manually.
- The game continues until there is a winner or the board is full (resulting in a draw).

#### **7. Game Loop:**

The game loop ensures that the game continues until a player wins or the game ends in a draw:



- The AI (Player O) makes its move using the **Minimax** algorithm.
- The human player (Player X) is prompted to make a move.
- The game checks after each move if someone has won or if the board is full.

## 8. Why Minimax and Alpha-Beta Pruning?

- **Minimax** ensures that the AI plays optimally by evaluating all possible moves and choosing the one that leads to the best outcome.
- **Alpha-Beta Pruning** makes the Minimax algorithm more efficient by eliminating unnecessary calculations and reducing the time complexity.

## 9. Game States:

The game tree for Tic-Tac-Toe can have up to  $9!$  (362,880) possible board configurations, but with **Alpha-Beta Pruning**, we can quickly ignore large parts of the search space, making the AI's decision-making process much faster.

## CODE

```
import math

# Constants for the players
PLAYER_X = 'X'
PLAYER_O = 'O'
EMPTY = ' '

# The board size (3x3)
BOARD_SIZE = 3

# Check if the current player has won
def is_winner(board, player):
    # Check rows, columns, and diagonals
    for row in range(BOARD_SIZE):
        if all([board[row][col] == player for col in range(BOARD_SIZE)]):
            return True
    for col in range(BOARD_SIZE):
        if all([board[row][col] == player for row in range(BOARD_SIZE)]):
            return True
    if all([board[i][i] == player for i in range(BOARD_SIZE)]):
        return True
    if all([board[i][BOARD_SIZE - 1 - i] == player for i in range(BOARD_SIZE)]):
        return True
```

```
    if all([board[i][BOARD_SIZE - 1 - i] == player for i in
range(BOARD_SIZE)]):
```

```
        return True
```

```
    return False
```

```
# Check if the board is full
```

```
def is_full(board):
```

```
    return all(board[row][col] != EMPTY for row in range(BOARD_SIZE)
for col in range(BOARD_SIZE))
```

```
# Minimax with Alpha-Beta Pruning
```

```
def minimax(board, depth, alpha, beta, maximizing_player):
```

```
    if is_winner(board, PLAYER_X):
```

```
        return 10 - depth
```

```
    if is_winner(board, PLAYER_O):
```

```
        return depth - 10
```

```
    if is_full(board):
```

```
        return 0
```

```
    if maximizing_player: # Maximizing for PLAYER_X
```

```
        max_eval = -math.inf
```

```
        for row in range(BOARD_SIZE):
```

```
            for col in range(BOARD_SIZE):
```

```
    if board[row][col] == EMPTY:
        board[row][col] = PLAYER_X
        eval = minimax(board, depth + 1, alpha, beta, False)
        board[row][col] = EMPTY
        max_eval = max(max_eval, eval)
        alpha = max(alpha, eval)
        if beta <= alpha:
            break
    return max_eval
else: # Minimizing for PLAYER_O
    min_eval = math.inf
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == EMPTY:
                board[row][col] = PLAYER_O
                eval = minimax(board, depth + 1, alpha, beta, True)
                board[row][col] = EMPTY
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
    return min_eval
```

```
# Find the best move for the AI (PLAYER_X)
```

```
def best_move(board):
```

```
    best_value = -math.inf
```

```
    move = None
```

```
    for row in range(BOARD_SIZE):
```

```
        for col in range(BOARD_SIZE):
```

```
            if board[row][col] == EMPTY:
```

```
                board[row][col] = PLAYER_X
```

```
                move_value = minimax(board, 0, -math.inf, math.inf, False)
```

```
                board[row][col] = EMPTY
```

```
                if move_value > best_value:
```

```
                    best_value = move_value
```

```
                    move = (row, col)
```

```
    return move
```

```
# Print the board in a human-readable format
```

```
def print_board(board):
```

```
    for row in range(BOARD_SIZE):
```

```
        print(" | ".join(board[row]))
```

```
        if row < BOARD_SIZE - 1:
```

```
            print("-" * 5)
```

```
# Play a game
```

```
def play_game():  
    board = [[EMPTY for _ in range(BOARD_SIZE)] for _ in  
range(BOARD_SIZE)]  
    print("Noughts and Crosses (Tic-Tac-Toe) Game\n")  
    print_board(board)
```

```
while True:
```

```
    # Player O's (AI) turn  
    print("\nAI (O)'s turn:")  
    row, col = best_move(board)  
    board[row][col] = PLAYER_O  
    print_board(board)
```

```
    if is_winner(board, PLAYER_O):
```

```
        print("\nAI (O) wins!")  
        break
```

```
    if is_full(board):
```

```
        print("\nIt's a draw!")  
        break
```

```
    # Player X's (Human) turn
```

```
    print("\nPlayer (X)'s turn:")
```

```
    while True:
```

```

try:
    row, col = map(int, input(f"Enter row and column (0-2) for X:
").split())

    if board[row][col] == EMPTY:
        board[row][col] = PLAYER_X
        break
    else:
        print("Cell already occupied. Try again.")
except (ValueError, IndexError):
    print("Invalid input. Please enter row and column values
between 0 and 2.")

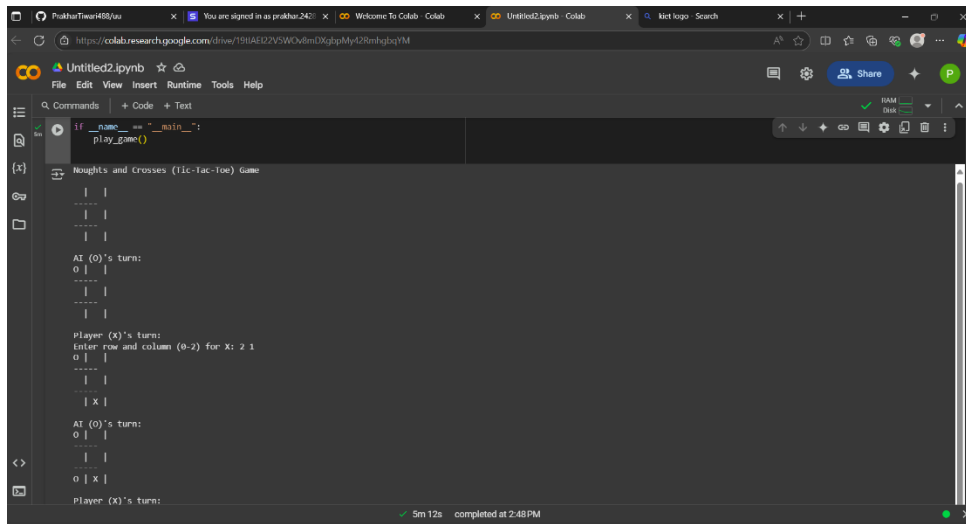
print_board(board)

if is_winner(board, PLAYER_X):
    print("\nPlayer (X) wins!")
    break
if is_full(board):
    print("\nIt's a draw!")
    break

if __name__ == "__main__":
    play_game()

```

# OUTPUT Screenshot



Colab notebook titled 'Untitled2.ipynb' showing the initial state of a Tic-Tac-Toe game. The code defines a 3x3 board and the AI's turn. The output shows the board and the AI's turn.

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

# Naughts and Crosses (Tic-Tac-Toe) Game

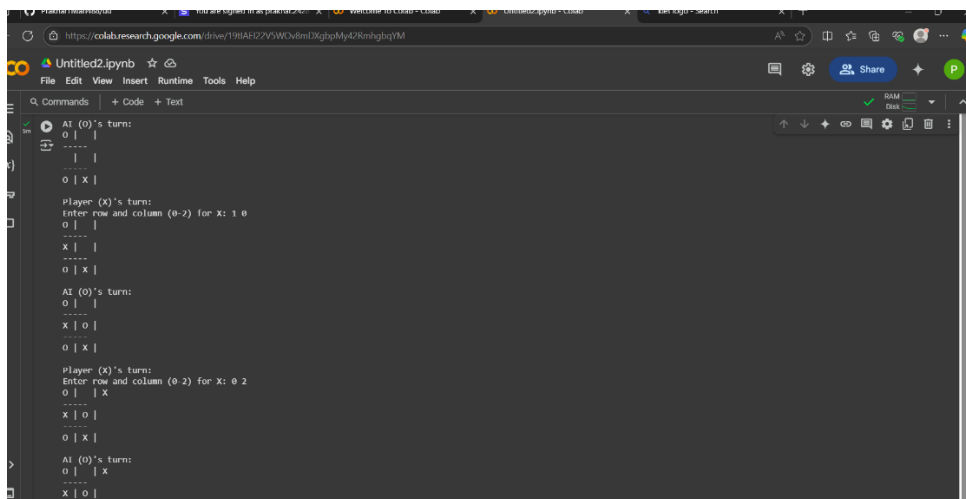
# Initial Board
board = [
    [' ', ' ', ' '],
    [' ', ' ', ' '],
    [' ', ' ', ' ']
]

# AI (O)'s turn
print("AI (O)'s turn:")
display_board(board)

# Player (X)'s turn
print("Player (X)'s turn:")
print("Enter row and column (0-2) for X: 2 1")
row, col = 2, 1
board[row][col] = 'X'
display_board(board)

# AI (O)'s turn
print("AI (O)'s turn:")
display_board(board)
```

5m 12s completed at 2:48 PM



Colab notebook titled 'Untitled2.ipynb' showing the game progress. The code continues to show the AI's turn and the player's turn. The output shows the board and the AI's turn.

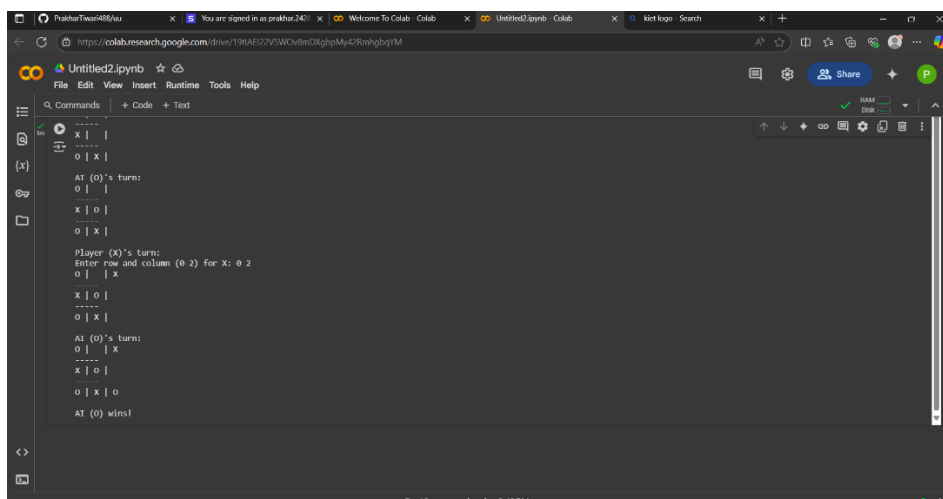
```
AI (O)'s turn:
display_board(board)

Player (X)'s turn:
Enter row and column (0-2) for X: 1 0
row, col = 1, 0
board[row][col] = 'X'
display_board(board)

AI (O)'s turn:
display_board(board)

Player (X)'s turn:
Enter row and column (0-2) for X: 0 2
row, col = 0, 2
board[row][col] = 'X'
display_board(board)

AI (O)'s turn:
display_board(board)
```



Colab notebook titled 'Untitled2.ipynb' showing the game ending with AI winning. The code continues to show the AI's turn and the player's turn. The output shows the board and the AI's turn.

```
AI (O)'s turn:
display_board(board)

Player (X)'s turn:
Enter row and column (0-2) for X: 0 2
row, col = 0, 2
board[row][col] = 'X'
display_board(board)

AI (O)'s turn:
display_board(board)

AI (O) wins!
```

5m 12s completed at 2:48 PM