**Project Title**: Noughts and Crosses with Alpha-Beta Pruning  
 **Author**: Roushan Kumar  
 **Instructor**: Mayank Lakhotia  
 **Course**: Introduction to AI  
 **Date**: 11/03/2025

**Overview**:  
 Noughts and Crosses, also known as Tic-Tac-Toe, is a popular two-player game where players alternately mark spaces on a 3x3 grid. The goal is to get three marks in a horizontal, vertical, or diagonal row. In this project, we focus on solving the game using an Artificial Intelligence (AI) technique called **Alpha-Beta Pruning**.

**Objective**:  
 The main objective of this project is to develop an AI agent that plays Noughts and Crosses (Tic-Tac-Toe) optimally. The AI will use the **Minimax algorithm** to evaluate possible moves, combined with **Alpha-Beta Pruning** to enhance the efficiency of the search process and reduce the number of nodes evaluated in the game tree.

**Significance**:  
 Alpha-Beta Pruning is a well-known optimization technique for the Minimax algorithm, which cuts off branches of the game tree that don’t need to be explored. This technique helps make the AI agent more efficient by avoiding unnecessary calculations, enabling faster decision-making.

### **3. Methodology**

The methodology outlines the approach used in this project, focusing on the implementation of Alpha-Beta Pruning for the Noughts and Crosses game.

#### **1. Game Representation:**

* **Board State**: The game is played on a 3x3 grid, represented by a list of lists in Python. Each cell can either be empty, contain "X" (player 1), or contain "O" (player 2).
* **Game Moves**: Players take turns, and the AI must decide its best move at each step.

#### **2. Minimax Algorithm:**

The **Minimax** algorithm is used to evaluate the possible outcomes of each move. It simulates every possible move on the board and selects the one that maximizes the AI's chances of winning while minimizing the opponent's chances. The algorithm works by exploring all possible moves and assigning a score to each outcome:

* **Maximizing Player (AI)**: The AI seeks to maximize the score.
* **Minimizing Player (Opponent)**: The opponent seeks to minimize the AI's score.

#### **3. Alpha-Beta Pruning:**

Alpha-Beta Pruning is an enhancement to the Minimax algorithm. It eliminates branches of the game tree that do not need to be evaluated, thus speeding up the decision-making process:

* **Alpha**: The best value found so far for the maximizing player.
* **Beta**: The best value found so far for the minimizing player.
* If at any point the value of a branch becomes worse than the current alpha or beta, that branch is pruned (i.e., it is not explored further).

#### **4. Algorithm Implementation:**

* **Base Case**: If the game is over (win, loss, or draw), the algorithm returns the score of the board (positive for a win, negative for a loss, and 0 for a draw).
* **Recursive Case**: The algorithm recursively explores all possible moves and uses Alpha-Beta Pruning to decide the optimal move for the AI.

#### **5. Decision-Making Process:**

* The AI evaluates all possible moves for its turn and selects the one that provides the best outcome. Alpha-Beta Pruning ensures that the AI evaluates fewer possible moves, which improves its performance.

### **Code Implementation**

Below is the Python implementation of the Noughts and Crosses game with Alpha-Beta Pruning:

import math

# Initialize the board

board = [' '] \* 9 # 3x3 grid

# Check if a player has won

def check\_winner(board, player):

win\_conditions = [

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

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

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

]

for condition in win\_conditions:

if board[condition[0]] == board[condition[1]] == board[condition[2]] == player:

return True

return False

# Evaluate the board: 1 for AI win, -1 for player win, 0 for draw or ongoing

def evaluate(board):

if check\_winner(board, 'X'):

return 1

elif check\_winner(board, 'O'):

return -1

return 0

# Check if the board is full (no empty spaces)

def is\_full(board):

return ' ' not in board

# Get available moves (empty spaces)

def get\_available\_moves(board):

return [i for i in range(9) if board[i] == ' ']

# Minimax algorithm with Alpha-Beta Pruning

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

score = evaluate(board)

# Return score if the game is over

if score != 0:

return score

if is\_full(board):

return 0

if is\_maximizing:

best = -math.inf

for move in get\_available\_moves(board):

board[move] = 'X' # AI's move

best = max(best, minimax(board, depth + 1, alpha, beta, False))

board[move] = ' ' # Undo move

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = math.inf

for move in get\_available\_moves(board):

board[move] = 'O' # Player's move

best = min(best, minimax(board, depth + 1, alpha, beta, True))

board[move] = ' ' # Undo move

beta = min(beta, best)

if beta <= alpha:

break

return best

# Find the best move for the AI

def find\_best\_move(board):

best\_val = -math.inf

best\_move = -1

for move in get\_available\_moves(board):

board[move] = 'X' # AI's move

move\_val = minimax(board, 0, -math.inf, math.inf, False)

board[move] = ' ' # Undo move

if move\_val > best\_val:

best\_val = move\_val

best\_move = move

return best\_move

# Print the board in a readable format

def print\_board(board):

for i in range(0, 9, 3):

print(board[i:i+3])

# Main game loop

def play\_game():

while True:

print\_board(board)

# AI's move

print("AI's move (X):")

best\_move = find\_best\_move(board)

board[best\_move] = 'X'

# Check for AI win or draw

if check\_winner(board, 'X'):

print\_board(board)

print("AI wins!")

break

if is\_full(board):

print\_board(board)

print("It's a draw!")

break

# Player's move (assuming player is 'O')

print("Your move (O):")

player\_move = int(input("Enter a move (0-8): "))

if board[player\_move] == ' ':

board[player\_move] = 'O'

else:

print("Invalid move. Try again.")

continue

# Check for Player win or draw

if check\_winner(board, 'O'):

print\_board(board)

print("You win!")

break

if is\_full(board):

print\_board(board)

print("It's a draw!")

break

# Start the game

play\_game()

### **Screenshots and Output**

Below are some screenshots showing the game in action and the AI making its optimal moves using Alpha-Beta Pruning.

