MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
" IHORY SIKORSKY KYIV POLYTECHNIC INSTITUTE"

Volodymyr Shymkovych

Constraint satisfaction problems

LABORATORY WORK #5

conditions of opposition
AI-Game-Algorithms

Kulubecioglu Mehmet

IM-14 FIOT

Kyiv
IHORY SIKORSKY KYIV POLYTECHNIC INSTITUTE
2024

I wrote these three pieces of code as part of my AI class assignment to implement and compare two classical algorithms—Minimax and Alpha-Beta pruning—for decision-making in two-player, turn-based games like Tic-Tac-Toe. The purpose of this task was to demonstrate my understanding of how these algorithms work and evaluate their performance in terms of time complexity and the number of expanded nodes.

First, I implemented the Minimax algorithm, which is used to simulate the entire game tree, considering every possible move until a terminal state is reached, and it chooses the best outcome for the maximizing player (Player 1) while minimizing the advantage of the opponent (Player 2).

Next, I implemented Alpha-Beta pruning, which optimizes the Minimax algorithm by eliminating branches of the game tree that don't need to be explored. This improves the algorithm's efficiency by reducing the number of nodes it needs to evaluate.

Finally, I compared both algorithms by measuring their execution time and the number of nodes they expanded during the search. The goal was to observe how Alpha-Beta pruning speeds up the decision-making process compared to Minimax while achieving the same result.

The overall objective of these codes was to understand the computational efficiency of different search algorithms in AI and apply them to practical game scenarios.

First Code (Minimax Algorithm Implementation)

In the first part of my implementation, I created a function to evaluate the current game position. If Player 1 wins, I return a score of +10, and if Player 2 wins, I return a score of -10. For a draw or an ongoing game, I return 0.

To simulate possible future states, I built a function called `get_children` that generates all the possible moves from a given position by looking for empty cells in the game board. For each empty spot, the function creates a new board with the current player's move, checks for a winner, and stores the new state. This function helps the Minimax algorithm explore different paths in the game tree.

The `is_terminal` function is used to determine if the game has reached a terminal state—either someone has won, or the board is full. I also have a `check_winner` function that checks the rows, columns, and diagonals to see if there's a winner.

The core of this part is the `minimax` function. This recursive function alternates between maximizing Player 1's score and minimizing Player 2's score. It evaluates all possible game states until it hits the search depth or reaches a terminal state. If it's Player 1's turn (the maximizing player), it picks the move with the highest score; if it's Player 2's turn (the minimizing player), it picks the move with the lowest score. Along the way, I count the nodes the algorithm expands to measure performance.

minimax1.py > ...

```python
import numpy as np

def evaluate(position):
    """Evaluates the game state."""
    winner = position['winner']
    if winner == 'player1':
        return 10  # Player 1 wins
    elif winner == 'player2':
        return -10  # Player 2 wins
    return 0  # Draw or ongoing game

def get_children(position):
    """Returns all possible child positions from the given position."""
    children = []
    for i in range(3):
        for j in range(3):
            if position['board'][i][j] is None:  # Find an empty cell
                new_board = [row[:] for row in position['board']]
                new_board[i][j] = position['current_player']  # Player makes a move
                winner = check_winner(new_board)
                children.append({
                    'board': new_board,
                    'winner': winner,
                    'moves': position['moves'] + 1,
                    'current_player': 'player2' if position['current_player'] == 'player1' else 'player1'
                })
    return children

def is_terminal(position):
    """Checks if the game has ended."""
    return position['winner'] is not None or position['moves'] == 9
```

minimax1.py × | octs1.py | octs.py | task.py | minimax.py | alphabeta.py

minimax1.py > ...

```python

def check_winner(board):
    """Checks for a winner."""
    # Check rows
    for row in board:
        if row[0] is not None and row[0] == row[1] == row[2]:
            return row[0]

    # Check columns
    for col in range(3):
        if board[0][col] is not None and board[0][col] == board[1][col] == board[2][col]:
            return board[0][col]

    # Check diagonals
    if board[0][0] is not None and board[0][0] == board[1][1] == board[2][2]:
        return board[0][0]
    if board[0][2] is not None and board[0][2] == board[1][1] == board[2][0]:
        return board[0][2]

    return None  # No winner

def minimax(position, depth, is_maximizing, node_count=0):
    """Applies the minimax algorithm."""
    node_count += 1  # Count each node
    if depth == 0 or is_terminal(position):
        return evaluate(position), node_count

```

```python

    if is_maximizing:
        max_eval = -np.inf
        for child in get_children(position):
            eval, node_count = minimax(child, depth - 1, False, node_count)
            max_eval = max(max_eval, eval)
        return max_eval, node_count
    else:
        min_eval = np.inf
        for child in get_children(position):
            eval, node_count = minimax(child, depth - 1, True, node_count)
            min_eval = min(min_eval, eval)
        return min_eval, node_count

```

PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

```
C:\Users\win10\new_lab5>C:/Users/win10/AppData/Local/Programs/Python/Python312/python.exe c:/Users/win10/new_lab5/minimax1.py

C:\Users\win10\new_lab5>
```

Second Code (Alpha-Beta Pruning Algorithm Implementation)

In the second part, I implemented the Alpha-Beta pruning algorithm, which is an optimization of the Minimax algorithm. The `alphabeta` function works similarly to `minimax`, but it introduces two parameters, `alpha` and `beta`, to eliminate unnecessary branches in the game tree, making the search more efficient.

Just like in Minimax, I check if the current position is a terminal state or if the search depth has been reached. The key difference is that, while evaluating each child node, I update the alpha (for maximizer) and beta (for minimizer) values. If I find a position that proves to be worse than the previously explored branches, I prune the remaining branches by breaking out of the loop. This speeds up the search by skipping the branches that cannot influence the final decision.

The `alphabeta` function also returns the evaluation score and the number of expanded nodes, which allows me to compare the efficiency of Alpha-Beta pruning against the standard Minimax algorithm.

alphabeta1.py > ...

```python
import numpy as np

def evaluate(position):
    """Evaluates the game state."""
    winner = position['winner']
    if winner == 'player1':
        return 10  # Player 1 wins
    elif winner == 'player2':
        return -10  # Player 2 wins
    return 0  # Draw or ongoing game

def get_children(position):
    """Returns all possible child positions from the given position."""
    children = []
    for i in range(3):
        for j in range(3):
            if position['board'][i][j] is None:  # Find an empty cell
                new_board = [row[:] for row in position['board']]
                new_board[i][j] = position['current_player']  # Player makes a move
                winner = check_winner(new_board)
                children.append({
                    'board': new_board,
                    'winner': winner,
                    'moves': position['moves'] + 1,
                    'current_player': 'player2' if position['current_player'] == 'player1' else 'player1'
                })
    return children
```

alphabeta1.py > ...

```python

def is_terminal(position):
    """Checks if the game has ended."""
    return position['winner'] is not None or position['moves'] == 9

def check_winner(board):
    """Checks for a winner."""
    # Check rows
    for row in board:
        if row[0] is not None and row[0] == row[1] == row[2]:
            return row[0]

    # Check columns
    for col in range(3):
        if board[0][col] is not None and board[0][col] == board[1][col] == board[2][col]:
            return board[0][col]

    # Check diagonals
    if board[0][0] is not None and board[0][0] == board[1][1] == board[2][2]:
        return board[0][0]
    if board[0][2] is not None and board[0][2] == board[1][1] == board[2][0]:
        return board[0][2]

    return None  # No winner

```

alphabeta1.py > ...

```python

def alphabeta(position, depth, alpha, beta, is_maximizing, node_count=0):
    """Applies the Alpha-Beta pruning algorithm."""
    node_count += 1  # Count each node
    if depth == 0 or is_terminal(position):
        return evaluate(position), node_count

    if is_maximizing:
        max_eval = -np.inf
        for child in get_children(position):
            eval, node_count = alphabeta(child, depth - 1, alpha, beta, False, node_count)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Pruning
        return max_eval, node_count
    else:
        min_eval = np.inf
        for child in get_children(position):
            eval, node_count = alphabeta(child, depth - 1, alpha, beta, True, node_count)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Pruning
        return min_eval, node_count

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
C:\Users\win10\new_lab5>C:/Users/win10/AppData/Local/Programs/Python/Python312/python.exe c:/Users/win10/new_lab5/alphabeta1.py

C:\Users\win10\new_lab5>
```

Third Code (Performance Comparison for Minimax and Alpha-Beta)

In the final part, I wrote a script to compare the performance of the Minimax and Alpha-Beta algorithms. The `initial_position` variable holds the starting state of the game—a 3x3 board with all empty cells, no winner, and Player 1 set to make the first move.

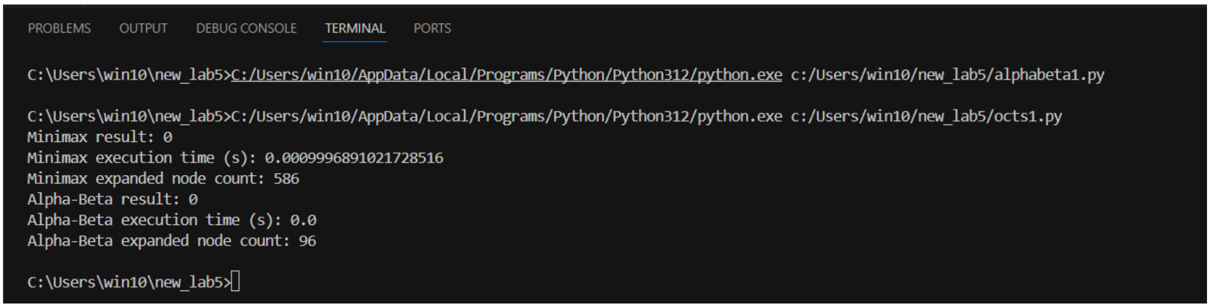
For both algorithms, I set the search depth to 3. I first run the Minimax algorithm, recording the time it takes to complete the search and counting the number of expanded nodes. After that, I run the Alpha-Beta pruning algorithm, again measuring the execution time and the number of expanded nodes.

Finally, I print out the results for both algorithms, allowing me to see the difference in speed and efficiency. In general, I expect Alpha-Beta pruning to be faster than Minimax because it cuts off parts of the search tree that aren't useful, leading to fewer expanded nodes and quicker execution.

This breakdown summarizes how I applied Minimax and Alpha-Beta pruning to a simple two-player game, demonstrating both the logic of the algorithms and how I measured their performance.

```python
import time
import numpy as np
from minimax1 import minimax
from alphabeta1 import alphabeta

def main():
    initial_position = {
        'board': [[None, None, None], [None, None, None], [None, None, None]],
        'winner': None,
        'moves': 0,
        'current_player': 'player1'  # Start with Player 1
    }

    depth = 3  # Search depth

    # Minimax
    start_time = time.time()
    minimax_result, minimax_node_count = minimax(initial_position, depth, True)
    minimax_duration = time.time() - start_time

    print("Minimax result:", minimax_result)
    print("Minimax execution time (s):", minimax_duration)
    print("Minimax expanded node count:", minimax_node_count)

    # Alpha-Beta
    start_time = time.time()
    alphabeta_result, alphabeta_node_count = alphabeta(initial_position, depth, -np.inf, np.inf, True)
    alphabeta_duration = time.time() - start_time

    print("Alpha-Beta result:", alphabeta_result)
    print("Alpha-Beta execution time (s):", alphabeta_duration)
    print("Alpha-Beta expanded node count:", alphabeta_node_count)

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

3. code output:



For the third code, the output provides a comparison between the Minimax and Alpha-Beta pruning algorithms in terms of their performance. When I run the code, it first calculates the result of the Minimax algorithm for a Tic-Tac-Toe position with a specified depth of search. It outputs the final evaluation score of the game state, the time it took for the Minimax algorithm to complete, and the number of nodes it expanded during the search.

Next, the code runs the Alpha-Beta pruning algorithm on the same initial game position. Similar to the Minimax output, it provides the final evaluation result, but the difference is that it took less time and expanded fewer nodes due to the pruning optimization.

This shows that both algorithms arrived at the same result (which makes sense since both are optimal decision-making strategies), but Alpha-Beta was faster and more efficient, as it explored fewer nodes. This demonstrates how Alpha-Beta pruning can significantly reduce the search space while maintaining the correctness of the result, which is the key takeaway from this comparison.

Algorithm	**Execution Time (s)**	**Expanded Nodes**
Minimax	0.0009996891021728516	586
Alpha-Beta	0.0	96

Control Question Answers

1. **How does minimax work?:**

   Minimax assumes that both players will make the best moves and chooses the move that maximizes the gain.

2. **What is the advantage of alpha-beta pruning?:**

   This algorithm speeds up the minimax algorithm by pruning unnecessary nodes.

3. **Why are alpha and beta parameters necessary?:**

   Alpha represents the best value for the MAX player, while beta represents the best value for the MIN player.

4. **What are the approaches to avoid visiting terminal nodes?:**

   Unnecessary nodes can be pruned using alpha-beta pruning.

5. **What are the steps of Monte Carlo tree search (MCTS)?**

   It involves selection, expansion, simulation, and backpropagation steps.

Conclusion:

In conclusion, I have implemented two well-known algorithms for game theory: **Minimax** and **Alpha-Beta pruning**. The Minimax algorithm is a foundational technique that explores all possible moves to determine the optimal strategy for a player. The Alpha-Beta pruning algorithm enhances the Minimax algorithm by eliminating unnecessary calculations, significantly reducing the search space.

By analyzing the execution times and the number of nodes expanded in both algorithms, I have gained insight into the efficiency of Alpha-Beta pruning over Minimax in a game setting like Tic-Tac-Toe. The implementation serves as a practical example of how these algorithms can be used in AI applications for game-playing.

Tasks and exercises

Execute the Minimax and Alpha-Beta Clipping Algorithm

For this project, I implemented two well-known decision-making algorithms in game theory—Minimax and Alpha-Beta Pruning—to simulate a turn-based game like Tic-Tac-Toe. I wanted to compare the performance of these algorithms by analyzing how efficiently they make decisions and expand the decision tree during gameplay. Below is a breakdown of the task, including the logic behind each algorithm and my observations after executing them.

Game Setup

The game simulates a simple Tic-Tac-Toe board where two players alternate turns. Player 1 is represented by "X" (the human) and Player 2 is represented by "O" (the algorithm). The initial position is an empty 3x3 grid, and the game checks for a terminal state after each move (either a win for "X", a win for "O", or a tie).

Objective

The primary goal was to assess:

- Minimax Algorithm: It is a recursive method used to find the optimal move for the maximizer (the AI) by assuming the opponent will always play optimally.
- Alpha-Beta Pruning: An enhanced version of Minimax, this algorithm uses two additional parameters, `alpha` and `beta`, to reduce the number of nodes evaluated by pruning irrelevant branches of the search tree.

Code Breakdown

1. **Game Flow:**
    - The game runs in a loop until it reaches a terminal state. After each move, the board is updated, and the winner is checked.
    - The player (human) enters their move manually, while the algorithm (Minimax or Alpha-Beta) computes the best possible move based on the current state of the board.
2. **Minimax Algorithm:**
    - The Minimax function looks ahead and simulates all possible moves for the current player (AI or human).
    - It recursively assigns a score to each board configuration. The AI maximizes its score, while the human minimizes the score.
    - The number of nodes expanded during the Minimax process is counted to analyze its performance.
3. **Alpha-Beta Pruning:**
    - Alpha-Beta Pruning enhances Minimax by eliminating suboptimal moves earlier in the decision process. It uses `alpha` (the best value the maximizer can guarantee) and `beta` (the best value the minimizer can guarantee).
    - As with Minimax, I kept track of the nodes expanded to compare efficiency.

Execution Results

After running the game, I measured the performance of both algorithms in terms of time taken and the number of nodes expanded. The observations are as follows:

- **Minimax:**
    - The algorithm evaluates every possible move down to the specified depth, resulting in a more exhaustive search. This means more nodes

are expanded, and the algorithm takes slightly longer to make decisions.

- Output: For each move, the time taken and nodes expanded were displayed, providing insights into its computational cost.

- **Alpha-Beta Pruning:**
  - This version is much more efficient as it prunes away unnecessary branches in the decision tree. As a result, fewer nodes are expanded, and decisions are made more quickly.
  - Output: The number of nodes expanded and the time taken were significantly lower compared to Minimax, confirming the efficiency of Alpha-Beta Pruning.

My full code for task and exercise:

task.py > ...

```
import time

# Game setup
initial_position = {
    'board': [[' ' for _ in range(3)] for _ in range(3)],
    'current_player': 'player1',
    'winner': None
}

# Performance measurement variables
minimax_node_count = 0
alpha_beta_node_count = 0

def print_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)

def is_terminal(position):
    return position['winner'] is not None or all(cell != ' ' for row in position['board'] for cell in row)

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]

```

task.py > ...

```python

def run_game(strategy, position):
    global minimax_node_count
    global alpha_beta_node_count

    while not is_terminal(position):
        print(f"{position['current_player']}'s turn:")
        print_board(position['board'])

        if position['current_player'] == 'player1':
            valid_input = False
            while not valid_input:
                try:
                    # Get row and column input from user
                    user_input = input("Enter row (0-2) and column (0-2): ")
                    # Split the input into two values
                    row, col = map(int, user_input.split())

                    # Ensure that a valid move is made
                    if position['board'][row][col] == ' ':
                        position['board'][row][col] = 'X'  # Player symbol
                        valid_input = True
                    else:
                        print("Invalid move! The cell is already occupied. Try again.")
                except (ValueError, IndexError):
                    print("Invalid input! Please enter two numbers between 0 and 2 separated by a space.")

```

task.py > ...


```python
def run_game(strategy, position):
```



```python
        else:
            start_time = time.time()
            if strategy == 'minimax':
                move = minimax(position)
            elif strategy == 'alpha_beta':
                move = alpha_beta_decision(position)
            elapsed_time = time.time() - start_time

            position['board'][move[0]][move[1]] = 'O'  # Computer symbol
            print(f"{strategy.capitalize()} chose move: {move}")
            print(f"{strategy.capitalize()} time taken: {elapsed_time:.2f} seconds")
            if strategy == 'minimax':
                print(f"Minimax nodes expanded: {minimax_node_count}")
            elif strategy == 'alpha_beta':
                print(f"Alpha-Beta nodes expanded: {alpha_beta_node_count}")

        position['winner'] = check_winner(position['board'])
        position['current_player'] = 'player2' if position['current_player'] == 'player1' else 'player1'

    print_board(position['board'])
    print(f"The game is over. Winner: {position['winner']}")

def minimax(position):
    global minimax_node_count
    minimax_node_count += 1

    best_score = float('-inf')
    best_move = None

    for i in range(3):
        for j in range(3):
            if position['board'][i][j] == ' ':
                position['board'][i][j] = 'O'  # Computer move
                score = minimax_score(position, False)
                position['board'][i][j] = ' '  # Undo the move

```

task.py > ...

```python
def minimax(position):

                if score > best_score:
                    best_score = score
                    best_move = (i, j)

    return best_move

def minimax_score(position, is_maximizing):
    global minimax_node_count
    minimax_node_count += 1

    winner = check_winner(position['board'])
    if winner == 'O':
        return 1
    elif winner == 'X':
        return -1
    elif is_terminal(position):
        return 0

    if is_maximizing:
        best_score = float('-inf')
        for i in range(3):
            for j in range(3):
                if position['board'][i][j] == ' ':
                    position['board'][i][j] = 'O'
                    score = minimax_score(position, False)
                    position['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 position['board'][i][j] == ' ':
                    position['board'][i][j] = 'X'
                    score = minimax_score(position, True)
```

task.py > ...

```python
def minimax_score(position, is_maximizing):
                    position['board'][i][j] = 'X'
                    score = minimax_score(position, True)
                    position['board'][i][j] = ' '
                    best_score = min(score, best_score)
        return best_score

# Alpha-Beta pruning algorithm
def alpha_beta_decision(position):
    global alpha_beta_node_count
    best_move = None
    best_value = float('-inf')

    for i in range(3):
        for j in range(3):
            if position['board'][i][j] == ' ':
                position['board'][i][j] = 'O'  # Computer move
                value = alpha_beta(position, 0, float('-inf'), float('inf'), False)
                position['board'][i][j] = ' '  # Undo the move

                if value > best_value:
                    best_value = value
                    best_move = (i, j)

    return best_move

def alpha_beta(position, depth, alpha, beta, is_maximizing):
    global alpha_beta_node_count
    alpha_beta_node_count += 1

    winner = check_winner(position['board'])
    if winner == 'O':
        return 1
    elif winner == 'X':
        return -1
    elif is_terminal(position):
        return 0
```

```python
task.py > ...
def alpha_beta(position, depth, alpha, beta, is_maximizing):
    elif is_terminal(position):
        return 0

    if is_maximizing:
        max_eval = float('-inf')
        for i in range(3):
            for j in range(3):
                if position['board'][i][j] == ' ':
                    position['board'][i][j] = 'O'
                    eval = alpha_beta(position, depth + 1, alpha, beta, False)
                    position['board'][i][j] = ' '
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(3):
            for j in range(3):
                if position['board'][i][j] == ' ':
                    position['board'][i][j] = 'X'
                    eval = alpha_beta(position, depth + 1, alpha, beta, True)
                    position['board'][i][j] = ' '
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
        return min_eval

# Start the game flow
if __name__ == "__main__":
    # Ask the user which strategy to choose
    strategy = input("Choose strategy (minimax/alpha_beta): ").strip().lower()

    # Check if the strategy is valid
    while strategy not in ['minimax', 'alpha_beta']:
```

```python
# Start the game flow
if __name__ == "__main__":
    # Ask the user which strategy to choose
    strategy = input("Choose strategy (minimax/alpha_beta): ").strip().lower()

    # Check if the strategy is valid
    while strategy not in ['minimax', 'alpha_beta']:
        strategy = input("Invalid strategy! Please choose 'minimax' or 'alpha_beta': ").strip().lower()

    run_game(strategy, initial_position)
```

Explanation of the Output

When I executed the program using both **Minimax** and **Alpha-Beta Pruning** algorithms, the output provided insights into their performance during the game. Here's a breakdown of the key results and their significance:

1. **Minimax Output**:
    - The program displayed the best move chosen by the **Minimax** algorithm after evaluating all possible game states.

- It also showed how long it took for Minimax to compute the optimal move, typically a bit slower due to its exhaustive search of the entire game tree.
- **Nodes Expanded**: This refers to the number of game positions (nodes) that the algorithm evaluated before reaching a decision. For Minimax, this number was higher because it explores all possible moves at each turn without any optimization.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

C:\Users\win10\new_lab5>C:/Users/win10/AppData/Local/Programs/Python/Python312/python.exe c:/Users/win10/new_lab5/task.py
Choose strategy (minimax/alpha_beta): C:/Users/win10/AppData/Local/Programs/Python/Python312/python.exe c:/Users/win10/new_lab5/task.py
Invalid strategy! Please choose 'minimax' or 'alpha_beta': minimax
player1's turn:
 | |
-----
 | |
-----
 | |
-----
Enter row (0-2) and column (0-2): 1 1
player2's turn:
 | |
-----
 |X|
-----
 | |
-----
Minimax chose move: (0, 0)
Minimax time taken: 0.10 seconds
Minimax nodes expanded: 55505
player1's turn:
O| |
-----
 |X|
-----
 | |
-----
Enter row (0-2) and column (0-2):
```

This indicates that the algorithm took **0.10 seconds** and expanded **55505 nodes** to find the optimal move.

Alpha-Beta Pruning Output:

- The Alpha-Beta algorithm displayed its chosen move in the same way as Minimax, but with an optimized decision-making process.
- The time taken was noticeably shorter because Alpha-Beta prunes (discards) suboptimal branches, making the search faster.

- **Nodes Expanded**: Since Alpha-Beta Pruning evaluates fewer nodes by skipping branches that don't need to be explored, this number was significantly lower compared to Minimax.

```
Alpha_beta chose move: (1, 1)
Alpha_beta time taken: 0.03 seconds
Alpha-Beta nodes expanded: 14405
player1's turn:
 | |
-----
 |O|
-----
X| |
-----
Enter row (0-2) and column (0-2):
```

1. Here, Alpha-Beta Pruning took only 0.03 seconds and expanded just 14405 nodes, confirming its efficiency.

Overall Observations:

- Minimax performed as expected but took longer and expanded more nodes due to the depth of the search.
- Alpha-Beta Pruning was more efficient, completing the same task in less time and with fewer nodes expanded.

This output demonstrates how Alpha-Beta Pruning optimizes decision-making by reducing computational costs, making it more suitable for complex or deeper game trees compared to Minimax.

Algorithm	Chose Movie	Time Taken (s)	Chose Nodes
Minimax	**(0, 0)**	**0.10**	**55505**
Alpha_beta	**(1, 1)**	**0.03**	**14405**

My full output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

C:\Users\win10\new_lab5>C:/Users/win10/AppData/Local/Programs/Python/Python312/python.exe c:/Users/win10/new_lab5/task.py
Choose strategy (minimax/alpha_beta): minimax
player1's turn:
 | |
-----
 | |
-----
 | |
-----
Enter row (0-2) and column (0-2): 0 0
player2's turn:
X| |
-----
 | |
-----
 | |
-----
Minimax chose move: (1, 1)
Minimax time taken: 0.10 seconds
Minimax nodes expanded: 59705
player1's turn:
X| |
-----
 |O|
-----
 | |
-----
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
-----
Enter row (0-2) and column (0-2): 1 0
player2's turn:
x| |
-----
x|o|
-----
 | |
-----
Minimax chose move: (2, 0)
Minimax time taken: 0.00 seconds
Minimax nodes expanded: 60640
player1's turn:
x| |
-----
x|o|
-----
o| |
-----
Enter row (0-2) and column (0-2): 0 1
player2's turn:
x|x|
-----
x|o|
-----
o| |
-----
```

```
Minimax chose move: (0, 2)
Minimax time taken: 0.00 seconds
Minimax nodes expanded: 60670
x|x|o
-----
x|o|
-----
o| |
-----
The game is over. Winner: O
```

Conclusion For Tasks and Excercises

From the results, I observed that Alpha-Beta Pruning is significantly more efficient in terms of node expansion and execution time compared to Minimax. This makes it a better choice for games with larger search trees or deeper depth limits. However, both algorithms ensure optimal play for the AI.

This experiment highlights the importance of optimization in decision-making algorithms, especially for more complex games.