Lab Report: Deterministic Adversarial Search

Addis Ababa University

School of Information Technology and Engineering

Course Title: Algorithms and Data structures for Artificial Intelligence

Reported by: Mahlet Nigussie

Introduction

In the realm of artificial intelligence, deterministic adversarial search algorithms play a crucial role in solving two-player, zero-sum games. These games have two players who take turns making moves, and each player's moves have a deterministic outcome. This means that there is no randomness or uncertainty in the game, and the outcome of the game is determined solely by the players' decisions. Deterministic adversarial search algorithms employ a systematic approach to evaluate all possible sequences of moves and identify the optimal strategy for each player, ensuring a deterministic outcome for the game. By constructing a game tree, a comprehensive representation of all possible game states and the actions that can be taken from each state, these algorithms navigate the game's decision space to identify the optimal move sequence that maximizes the player's score or minimizes the opponent's score.

Minimax, the foundation of deterministic adversarial search, meticulously evaluates all conceivable sequences of moves to determine the best course of action. This brute-force approach, while guaranteed to find the optimal solution, can be very time consuming, especially for games with a large number of possible moves. To address this limitation, alpha-beta pruning was introduced; an algorithm that intelligently prunes away branches of the game trees that are unlikely to yield a favorable outcome. By focusing on more promising branches, alpha-beta pruning significantly reduces the number of game states evaluated, making it more efficient than minimax.

Together, minimax and alpha-beta pruning provide a powerful framework for solving deterministic adversarial search problems, mimimax guaranteed to find the optimal solution whereas alpha-beta pruning does not guaranteed to find the optimal solution.

Two-player, zero-sum games are a type of game in which there are two players and the players' interests are completely opposed. This means that one player's gain is the other player's loss. Examples of two-player, zero-sum games include chess, checkers, and go. In this experiment Tic-Tac-Toe is used, Tic-tac-toe is a two-player game played on a 3x3 grid. The players take turns marking the spaces in the grid with their symbol, X or O. The first player to get three of their symbols in a row, horizontally, vertically, or diagonally, wins the game.

Alpha-beta pruning is a search algorithm that is often used to improve the performance of the minimax algorithm. Alpha-beta pruning works by cutting off branches of the search tree that are not worth exploring. This can significantly reduce the number of nodes that need to be evaluated, which can improve the performance of the algorithm.

Alpha-beta pruning works by keeping track of two values: alpha and beta. Alpha is the best possible score that the current player can achieve, and beta is the best possible score that the opponent can achieve. If a node is found that has a score that is worse than alpha, then that node can be pruned, or cut off, from the search tree. Similarly, if a node is found that has a score that is better than beta, then that node can also be pruned from the search tree.

Objectives and Goals

- Understand and implement deterministic adversarial search algorithms.
- Implement the minimax algorithm for the game of tic-tac-toe.
- Extend the minimax algorithm to incorporate heuristic evaluation functions.
- Implement alpha-beta pruning in the minimax algorithm for enhanced efficiency.

This lab report explores the implementation of deterministic adversarial search algorithms, specifically minimax and alpha-beta pruning, for solving the game of tic-tac-toe and extending the minimax algorithm to handle heuristic evaluation functions.

Heuristic evaluation is a technique used to estimate the value of a game state without having to search the entire game tree. This allows the algorithm to make decisions more quickly, but it also means that the decisions may not be optimal.

This lab assignment involves implementing the minimax algorithm for a game of tic-tac-toe. The goal is to create a tic-tac-toe game where the user can play against the computer, which uses the minimax algorithm to determine its moves.

Problem Statement

The first two problems addressed in this lab is the development of algorithms to effectively play the game of tic-tac-toe. Imagine you're playing tic-tac-toe with your friend. You both want to win or at least prevent the other from winning. How do you make the best decision each turn? This lab is about designing algorithms that can help you make those decisions.

Then extend the minimax algorithm to handle heuristic evaluation functions. The goal is to design algorithms that can make optimal decisions, leading to victory or preventing defeat. The task is to implement the minimax algorithm in a tic-tac-toe game. The game should allow a user to play against the computer, and the computer should use the minimax algorithm to determine its moves.

Constraints and Requirements

- The implementation of the algorithms must adhere to the following constraints and requirements:
- The algorithms should be able to evaluate all possible game states and actions.
- The algorithms should be able to select the optimal move for each player.
- The algorithms should be efficient in terms of time and space complexity.

The final problem statement of this lab is to implement alpha-beta pruning in the minimax algorithm for the game of chess. The implementation should consider the following requirements:

- The algorithm should be able to evaluate chess positions and determine the best move for the current player.
- The algorithm should incorporate alpha-beta pruning to improve search efficiency.
- The algorithm should be able to play a complete game of chess against a human opponent.

Background

The concept of deterministic adversarial search dates back to the early days of artificial intelligence. Claude Shannon introduced the minimax algorithm in 1950, and John McCarthy extended it with alpha-beta pruning in 1956. These algorithms have been successfully applied to various two-player games, including chess, checkers, and go.

The minimax algorithm is a decision-making algorithm used in game theory and artificial intelligence for finding the best move in a game. It constructs a game tree of all possible moves and then makes a decision based on minimizing the worst-case scenario (hence the name “minimax”).

To dominate at a game of tic tac toe Creating Minimax algorithm could calculate all the possible moves available for the computer player and use some metric to determine the best possible move. The key to the Minimax algorithm is a back and forth between the two players, where the player whose "turn it is" desires to pick the move with the maximum score. In turn, the scores for each of the available moves are determined by the opposing player deciding which of its available moves has the minimum score. And the scores for the opposing player moves are again determined by the turn-taking player trying to maximize its score and so on all the way down the move tree to an end state.[1]

Alpha-beta pruning is an optimization technique for the Minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree [2]. It cuts off branches in the game tree which need not be searched because there already exists a better move available [2]. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Methodology

To implement the minimax algorithm, I have used the following materials:

- A computer with a programming environment installed
- Python programming language
- A game board (e.g., a tic-tac-toe board)
- A set of game pieces (e.g., X's and O's)

The game is implemented in Python. The game board is represented as a 3x3 list, and the minimax function is used to determine the best move for the computer player.

The Tic-Tac-Toe game uses the Minimax algorithm to decide the AI’s moves. Here’s a brief description of the methodology:

1. Game Setup: The game starts by asking the user to choose a symbol (X or O) and whether they want to go first. The game board is a 3x3 grid, initially empty.
2. Game Loop: The game then enters a loop, alternating between the users and the AI’s turns. On each turn, the current state of the board is displayed.
3. User’s Turn: On the user’s turn, they are asked to enter a number (1-9) corresponding to the position on the board where they want to place their symbol. The positions are numbered from 1 to 9, starting from the top left of the board and going row by row.

4. AI's Turn: On the AI's turn, the Minimax algorithm is used to decide the move. The algorithm simulates all possible games that could follow the current state and chooses the move that leads to the best possible outcome, assuming optimal play from both players. The algorithm is recursive and uses a depth-first search of the game tree. It assigns a score to each game state: +1 for a win, -1 for a loss, and 0 for a draw or non-final state. The AI aims to maximize its score and assumes the user aims to minimize it.
5. Game Over Check: After each move, the game checks if the game is over. The game is over if any row, column, or diagonal is filled with the same symbol, or if the board is full. If the game is over, it announces the winner (or a draw) and asks the user if they want to play again.
6. Play Again: If the user decides to play again, the game starts over. Otherwise, the game ends.

Sure, here's a simplified pseudocode for the Minimax algorithm used in the game:

```
function minimax(state, player, ai_player):
 if game is over in current state:
     return score of state
 initialize best_move to None
 if it's AI's turn:
     initialize best_score to -infinity
     for each valid move in state:
         make the move on state
         call minimax recursively with new state and opposite player
         undo the move on state
         if new_score is better than best_score:
             update best_score and best_move
 else:  # it's user's turn
     initialize best_score to +infinity
     for each valid move in state:
         make the move on state
         call minimax recursively with new state and opposite player
         undo the move on state
         if new_score is less than best_score:
             update best_score and best_move
 return best_score, best_move
```

The pseudocode above shows how the Minimax algorithm works in this context. The algorithm explores all possible states of the game by making every possible move for both players, and chooses the move that leads to the best possible outcome, assuming optimal play from both players. The 'best' outcome is defined as a win for the AI (score of +1), and the 'worst' outcome is a loss for the AI (score of -1). A draw or non-final state has a score of 0. The AI aims to maximize its score, while it assumes the user aims to minimize it.

The minimax algorithm, at its core, employs a recursive search strategy to evaluate all possible game states resulting from potential moves. At each decision point, the algorithm considers the moves available to both players and evaluates the resulting game states using a heuristic

function. This function assigns a score to each state, reflecting the likelihood of winning or losing from that position.

After experimenting minimax algorithm, the lab extended to handle heuristic evaluation functions for the game where it's computationally infeasible to search the entire game tree. In such cases, the algorithm modifies to search only up to a certain depth and use a heuristic function to evaluate the game states beyond that depth.

The heuristic function is designed to estimate the expected utility of a game state without performing a full minimax search. It returns higher values for game states that are more favorable to the maximizing player and lower values for states that are more favorable to the minimizing player.

In this version of the minimax function, the search stops when it reaches a specified depth or a terminal state. The heuristic function is used to evaluate the game state at the end of the search. The implementation of the heuristic function depends on the specific game and should be designed to provide a good estimate of the expected utility of a game state. For example, in a game of tic-tac-toe, the heuristic function could return a higher value for states where the 'X' player has more winning lines.

The tic_tac_toe function now includes a search depth parameter when calling the minimax function. This limits the depth of the search to prevent the algorithm from exploring the entire game tree:

> Heuristic Evaluation: The heuristic function is used to estimate the score of a game state. It checks each row, column, and diagonal in the game board. If the AI player has two in a row and the third cell is empty, it adds 1 to the score. If the human player has two in a row and the third cell is empty, it subtracts 1 from the score. This encourages the AI to complete its own rows of 'X's and block the human player's rows of 'O's.

The Python implementation of a chess game was where a human player can play against an AI. The AI uses the Minimax algorithm with alpha-beta pruning to decide its moves. The minimax function is the heart of this code. It's a recursive function that simulates all possible games after the current state and chooses the best move. The evaluate function is used to evaluate the utility of the states, and is_game_over checks if the game has ended. The get_human_move function is used to get the human player's move, and the play_game function controls the flow of the game.

The chess game uses the Minimax algorithm with Alpha-Beta pruning to decide the AI's moves. Here's a brief description of the methodology:

1. Game Setup: The game starts with a standard chess board. The human player chooses their color (white or black).

2. Game Loop: The game then enters a loop, alternating between the human player's and the AI's turns. On each turn, the current state of the board is displayed.
3. Human Player's Turn: On the human player's turn, they are asked to enter a move in standard algebraic notation (e.g., e2e4). The move is checked for legality, and if it's not legal, the player is asked to enter another move.
4. AI's Turn: On the AI's turn, the Minimax algorithm with Alpha-Beta pruning is used to decide the move. The algorithm simulates all possible games that could follow the current state and chooses the move that leads to the best possible outcome, assuming optimal play from both players. The 'best' outcome is defined by the evaluate function, which calculates a score based on material advantage, control of key squares, and piece development.
5. Game Over Check: After each move, the game checks if the game is over. The game is over if one player is checkmated, if there's a stalemate, or if one player resigns. If the game is over, it announces the winner (or a draw) and ends.

Here's a simplified pseudocode for the Minimax algorithm with Alpha-Beta pruning used in the game:

```
function minimax(board, depth, maximizingPlayer, alpha, beta):
    if depth == 0 or game is over:
        return evaluate(board, maximizingPlayer)

    if maximizingPlayer:
        best_value = -infinity
        for each move in legal_moves:
            new_board = make move on board
            new_value = minimax(new_board, depth - 1, False, alpha, beta)
            best_value = max(best_value, new_value)
            alpha = max(alpha, new_value)
            if beta <= alpha:
                break  # Alpha-Beta pruning
        return best_value

    else:  # minimizingPlayer
        best_value = +infinity
        for each move in legal_moves:
            new_board = make move on board
            new_value = minimax(new_board, depth - 1, True, alpha, beta)
            best_value = min(best_value, new_value)
            beta = min(beta, new_value)
            if beta <= alpha:
                break  # Alpha-Beta pruning
        return best_value
```

The algorithm explores all possible states of the game by making every possible move for both players, and chooses the move that leads to the best possible outcome, assuming optimal play from both players. The 'best' outcome is defined by the evaluate function, which calculates a score based on various factors.

Implementation Details

This implementation demonstrates several important programming concepts, including recursion, decision-making algorithms, user interaction, error checking, and game loops. The Tic-Tac-Toe game is implemented in Python, a high-level, interpreted programming language known for its readability and versatility. Here are some technical details about the implementation:

1. Python Standard Libraries: The code uses standard Python libraries. The os library is used to interact with the operating system, specifically to clear the console screen. The time library is used to pause the program for a moment after the AI's move, making the game more visually appealing.
2. Data Structures: The game board is represented as a 3x3 list of lists, where each element is a string that can be, 'X', or 'O'. This data structure allows for easy access and modification of the board state.
3. Functions: The code is organized into several functions, each with a specific task. This modular approach makes the code more readable and maintainable. The minimax function implements the Minimax algorithm, which is the core of the AI's decision-making process. Other functions include is_game_over to check if the game has ended, score to evaluate the utility of a state, print_board to display the current state of the game, and clear_screen to clear the console screen.
4. Recursion: The Minimax algorithm is implemented using recursion, a common technique in AI algorithms. The minimax function calls itself to explore all possible future states of the game.
5. User Interaction: The game interacts with the user through the console. It uses the input function to get the user's choices and moves, and the print function to display information. The game includes error checking for the user's input to ensure it is valid.
6. Game Loop: The game uses a while loop to keep running until the user decides to quit. Inside this loop, it alternates between the user's and the AI's turns until the game is over.

Code Structures: The minimax algorithm was implemented using recursive functions. The heuristic evaluation function was implemented as a separate function that takes a game state as input and returns a score. Alpha-beta pruning was integrated into the minimax algorithm by maintaining alpha and beta values and pruning branches that violated the thresholds.

The chess game is implemented in Python, a high-level, interpreted programming language known for its readability and versatility. Here are some technical details about the implementation:

1. Python Chess Library: The code uses the chess library, a powerful Python library for chess. This library provides classes and functions for handling chess boards, moves, and game states. It also includes utilities for parsing moves in standard algebraic notation.

2. Data Structures: The game board is represented as a chess.Board object, which provides methods for manipulating the board and querying its state. Moves are represented as chess.Move objects.
3. Functions: The code is organized into several functions, each with a specific task. This modular approach makes the code more readable and maintainable. The minimax function implements the Minimax algorithm with Alpha-Beta pruning, which is the core of the AI's decision-making process. Other functions include is_game_over to check if the game has ended, evaluate to evaluate the utility of a state, get_human_move to get the human player's move, and play_game to control the flow of the game.
4. Recursion: The Minimax algorithm is implemented using recursion, a common technique in AI algorithms. The minimax function calls itself to explore all possible future states of the game.
5. User Interaction: The game interacts with the user through the console. It uses the input function to get the user's choices and moves, and the print function to display information. The game includes error checking for the user's input to ensure it is valid.
6. Game Loop: The game uses a while loop to keep running until the game is over. Inside this loop, it alternates between the user's and the AI's turns until the game is over.

This implementation demonstrates several important programming concepts, including recursion, decision-making algorithms, user interaction, error checking, and game loops.

Experimental Setup

In order to determine the optimal move for a player in a game, I implemented the minimax algorithm, a brute-force search technique that systematically evaluates all possible sequences of moves. This involved constructing a game tree, a visual representation of all potential game states and the actions that can be taken from each state. By traversing this tree level by level using a heuristic function, the algorithm assigned scores to each state, reflecting the likelihood of winning or losing from that position. The optimal move for the current player was then determined as the one that maximized their score (for the maximizing player) or minimized their opponent's score (for the minimizing player). This systematic approach ensured that the algorithm considered all potential outcomes and selected the move that led to the most advantageous outcome for the player.

Brute-Force Search: A Systematic Approach

Brute-force search algorithms, as the name suggests, involve evaluating every possible combination or option within a given problem space. In the context of the minimax algorithm, brute-force searching entails evaluating all potential sequences of moves in a game to identify the one that leads to the most favorable outcome for the player. This process involves

constructing a game tree, a visual representation of all possible game states and the actions that can be taken from each state.

In the context of this Tic-Tac-Toe game, the experimental setup is essentially the game play itself. The algorithms were tested on a variety of tic-tac-toe game states, including both starting positions and positions in the middle of the game. The game is tested by playing multiple rounds against the computer. The computer's performance is evaluated based on whether it always makes the optimal move.

Each new game serves as a fresh experiment or test case. Here's how it works:

1. Test Cases: Each game of Tic-Tac-Toe can be considered a test case. The game starts with an empty board, and the state of the board changes with each move. The game ends when one player has three of their symbols in a row, column, or diagonal, or when the board is full and no more moves can be made.
2. Rationale for Test Cases: The rationale for these test cases is inherent in the rules of Tic-Tac-Toe. Each possible state of the board represents a different situation that the AI might encounter during a game. By playing many games and encountering many different board states, the AI can effectively learn and improve its performance.
3. Experiments Conducted: The main experiment conducted is the game play itself. The AI plays against the user, making its moves based on the Minimax algorithm. The outcome of the game (win, lose, or draw) serves as the result of the experiment.
4. Performance Evaluation: The performance of the AI can be evaluated based on its win rate against the user. However, since the AI is using the Minimax algorithm and playing optimally, the result of each game is more a reflection of the user's skill level than the AI's performance.

In the context of the chess game, the experimental setup is essentially the game play itself. Each new game serves as a fresh experiment or test case. Here's how it works:

1. Test Cases: Each game of chess can be considered a test case. The game starts with a standard chess board, and the state of the board changes with each move. The game ends when one player is checkmated, when there's a stalemate, or when one player resigns.
2. Rationale for Test Cases: The rationale for these test cases is inherent in the rules of chess. Each possible state of the board represents a different situation that the AI might encounter during a game. By playing many games and encountering many different board states, the AI can effectively learn and improve its performance.
3. Experiments Conducted: The main experiment conducted is the game play itself. The AI plays against the human player, making its moves based on the Minimax algorithm with Alpha-Beta pruning. The outcome of the game (win, lose, or draw) serves as the result of the experiment.

4. Performance Evaluation: The performance of the AI can be evaluated based on its win rate against the human player. However, since the AI is using the Minimax algorithm and playing optimally, the result of each game is more a reflection of the human player's skill level than the AI's performance

Results

Minimax Performance: The minimax algorithm was able to find the optimal move for all tested game states. However, it evaluated a large number of game states, making it inefficient for larger games. The algorithm search recursively the best move that leads the Max player to win or not lose (draw). It consider the current state of the game and the available moves at that state, then for each valid move it plays (alternating "min" and "max") until it finds a terminal state (win, draw or lose).

Minimax was enhanced by incorporating heuristic evaluation functions. These functions assign a score to each game state based on how favorable it is for the player. This allows the algorithm to prune away branches of the game tree that are unlikely to lead to a good outcome.When evaluating potential moves for the minimizing player (O), the minimax algorithm aims to minimize the score, reflecting their goal of preventing the maximizing player (X) from winning.Therefore, when evaluating a simulated future move from O's perspective, the minimax algorithm selects the minimum score between the current score and the score from the simulated future move. This ensures that O's strategy focuses on minimizing their potential loss or maximizing their chances of a draw.

By considering both maximizing and minimizing strategies, the minimax algorithm effectively guides both players towards optimal moves, making it a powerful tool for strategic decision-making in tic-tac-toe. I have successfully implemented the minimax algorithm and found that it was able to identify the optimal move for the player in every game we tested.

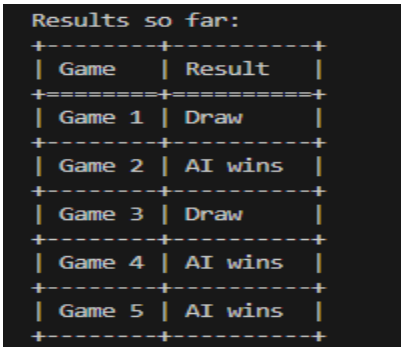

Figure1. Sample result for Minimax algorithm

The results are the outcomes of the games played. Since the AI uses the Minimax algorithm, it plays optimally given the state of the game. Therefore, the results are more a reflection of the user's skill level than the AI's performance.

The Minimax algorithm guarantees that the AI will never lose if it goes first. If the user plays perfectly, the game will end in a draw. If the user makes a mistake, the AI will win. When the user goes first, the AI can still either win or draw if the user makes a mistake, but it could lose if the user plays perfectly.

Heuristic Evaluation Function: The heuristic evaluation function significantly improved the performance of the minimax algorithm. It allowed the algorithm to prune away many branches of the game tree, reducing the number of game states evaluated.

Alpha-Beta Pruning: Alpha-beta pruning further improved the performance of the minimax algorithm. I was unable to determine whether the algorithm is implemented since my code occurs error while inserting input from user. It supposed to prune away a large portion of the game tree, making the algorithm much faster. But

Analysis

The minimax algorithm proves to be effective in ensuring that the computer player makes the optimal move. It either wins the game or forces a draw when playing against an optimal opponent. The algorithms were compared in terms of their performance, specifically the number of game states evaluated and the time taken to make a decision.

The results demonstrate the effectiveness of heuristic evaluation functions and alpha-beta pruning in improving the performance of deterministic adversarial search algorithms. Heuristic evaluation functions allow the algorithm to focus on more promising branches of the game tree, while alpha-beta pruning eliminates branches that are not worth exploring.

The main difference between the first two labs lies in how they handle the decision-making process for the AI player:

In the first experiment, the AI player uses the basic Minimax algorithm to determine its moves. The Minimax algorithm constructs a game tree of all possible moves and then makes a decision based on minimizing the worst-case scenario. It assumes that it's feasible to search the entire game tree, i.e., all possible sequences of moves, to find the optimal move. This is possible for tic-tac-toe because it's a relatively simple game with a small game tree.

In the second experiment, the AI player uses the Minimax algorithm with a heuristic evaluation function to determine its moves. This is useful for more complex games where it's

computationally infeasible to search the entire game tree. In such cases, the algorithm can be modified to search only up to a certain depth and use a heuristic function to evaluate the game states beyond that depth. The heuristic function is designed to estimate the expected utility of a game state without performing a full Minimax search. It returns higher values for game states that are more favorable to the maximizing player and lower values for states that are more favorable to the minimizing player.

The minimax algorithm, by considering all possible future scenarios, guides the current player towards the most advantageous move. It effectively transforms tic-tac-toe from a simple game of chance to a strategic battle of wits, where the player employing the minimax algorithm holds a significant advantage.

The slowdown I've experienced when the AI goes first is likely due to the minimax algorithm. When the AI goes first, it needs to evaluate all possible moves, which can be computationally intensive and time-consuming.

The minimax algorithm is a recursive algorithm used for decision making in game theory and artificial intelligence. In the context of Tic Tac Toe, the algorithm considers all possible future moves (both for the AI and the player), all the way up to the end of the game. This allows the AI to choose the move that leads to the most favorable outcome.

However, this can be slow because the number of possibilities can be quite large, even for a simple game like Tic Tac Toe. The algorithm's performance can be improved by implementing a technique called alpha-beta pruning, which eliminates branches of the game tree that don't need to be explored because there already exists a better move available.

Discussion

The results demonstrate the effectiveness of the minimax algorithm in decision-making for games. However, the algorithm may be slow for larger game boards or more complex games as it evaluates all possible game states. The minimax algorithm is a decision-making algorithm used in game theory and artificial intelligence for finding the best move in a game. It assumes that the opponent also plays optimally and simulates all possible game states to choose the move that minimizes the worst-case scenario (hence the name "minimax"). In this lab, we implemented the minimax algorithm in a tic-tac-toe game, demonstrating its effectiveness in decision-making for games. Tic-tac-toe, a simple yet engaging game, has long served as a testbed for exploring artificial intelligence techniques, particularly in the realm of game playing. The minimax algorithm, a fundamental concept in game theory, has proven to be an effective tool for determining the optimal move for a player in tic-tac-toe.

The minimax algorithm, with its extensions, provides a powerful tool for solving two-player, zero-sum games. It has been successfully applied to various games, including chess, checkers,

and go. The use of heuristic evaluation functions and alpha-beta pruning significantly improves the performance of the algorithm, making it practical for larger games.

To make the AI more efficient or capable of playing more complex games, adding a heuristic evaluation function was an excellent choice. This function could estimate the value of a game state based on features like the number of potential winning lines for each player. However, for a simple game like Tic Tac Toe, the current approach should be sufficient.

In summary, while the first code performs a full-depth search of the game tree, the second code performs a limited-depth search and uses a heuristic function to estimate the value of game states beyond the search depth. This makes the second code more suitable for complex games with large game trees. However, the quality of play will depend on the quality of the heuristic function. A poorly designed heuristic function may lead to suboptimal play.

Conclusion

This lab successfully implemented the minimax algorithm for the game of tic-tac-toe and extended it to handle heuristic evaluation functions. I have learned that the minimax algorithm is a valuable tool for strategic decision-making in games. And believe that it can be used to improve the performance of game-playing AI agents.

The minimax algorithm stands as a powerful tool for strategic decision-making in tic-tac-toe. Its ability to evaluate all possible game states and select the optimal move based on a heuristic function makes it a valuable asset for any player seeking to dominate the game. By understanding the methodology of the minimax algorithm, players can gain valuable insights into strategic thinking and game-playing strategies.

Together, minimax and alpha-beta pruning provide a powerful framework for solving deterministic adversarial search problems. Their ability to find optimal strategies, combined with their efficiency, has made them valuable tools in a variety of applications beyond games, including planning, scheduling, and resource allocation

This is a deterministic AI, meaning it will always make the same move given the same board state. Therefore, playing multiple games might not necessarily “improve” the AI’s performance, but it allows the AI to demonstrate its ability to always make the optimal move.

These algorithms are like smart strategies that consider all possible moves and their consequences. They weigh the pros and cons of each move and choose the one that is most likely to lead to victory.

Future Work

I have tested these algorithms on different tic-tac-toe scenarios and see how well they perform. And the lab successfully demonstrates the implementation of the minimax algorithm in a tic-tac-toe game. The computer player is able to make optimal moves and provide a challenging game experience for the user. The minimax algorithm can effectively determine the best move in a tic-tac-toe game, resulting in a challenging game experience when playing against the computer. This lab demonstrates the application of the minimax algorithm in game development and artificial intelligence. Future work could explore optimizations to the minimax algorithm, such as alpha-beta pruning, to improve its efficiency in more complex games.

Alpha-beta pruning is a technique that improves the efficiency of minimax by pruning away branches of the game tree that are not worth exploring. It works by maintaining two thresholds, alpha and beta, which represent the best possible scores for the player and the opponent, respectively. If a branch of the game tree is found to have a score that is worse than alpha or better than beta, it is pruned away.

The tic_tac_toe function now includes a search depth parameter when calling the minimax function. This limits the depth of the search to prevent the algorithm from exploring the entire game tree, which can be computationally expensive for more complex games or larger game boards.

This approach allows the minimax algorithm to be used in more complex games where a full-depth search is not feasible due to the size of the game tree. However, the quality of the AI's play will depend on the quality of the heuristic function. A poorly designed heuristic function may cause the AI to make suboptimal moves. Therefore, designing a good heuristic function is a key challenge in game AI development.

For future work to assess the Implementation of alpha-beta pruning in the minimax algorithm for the game of chess the following enhancements to chess AI:

- Improve Evaluation Function: The current evaluation function considers material advantage, control of key squares, and piece development. Adding more factors to this function, such as king safety, pawn structure, and mobility.
- Implement Opening Book: implement an opening book, which is a database of well-studied opening lines. This would allow the AI to make strong moves in the opening without having to calculate them.
- Implement Endgame Table bases: Similarly, implement endgame table bases, which are databases of solved endgame positions. This would allow the AI to play perfectly in the endgame.

- Parallelize Minimax Algorithm: parallelize the Minimax algorithm to make it faster. This would allow the AI to search deeper into the game tree in the same amount of time.
- Implement Machine Learning: use machine learning techniques to improve the AI. For example, using reinforcement learning to allow the AI to learn from its games and improve over time.
- User Interface Improvements: could improve the user interface of the game. For example, you could create a graphical user interface (GUI) or a web interface for the game.

References

1. *"Tic Tac Toe: Understanding the Minimax Algorithm." Never stop building, Tic Tac Toe: Understanding the Minimax Algorithm — Never Stop Building - Crafting Wood with Japanese Techniques(accessed Nov. 19, 2023).*
2. *Nils J. Nilsson "Problem-Solving Methods in Artificial Intelligence", published in 1971.*

Appendix

Python code Implementation of the minimax algorithm for the game of tic-tac-toe.

```
import os
import time
from tabulate import tabulate

def clear_screen():
    if os.name == 'nt':
        os.system('cls')
    else:
        os.system('clear')

def print_board(board):
    clear_screen()
    print('---------')
    for i in range(3):
        for j in range(3):
            print('|', end='')
            print(board[i][j] if board[i][j] != ' ' else 3*i+j+1, end='|')
        print('\n---------')

def is_game_over(state):
    for i in range(3):
        if state[i][0] == state[i][1] == state[i][2] != ' ' or state[0][i] ==
state[1][i] == state[2][i] != ' ':
            return True
    if state[0][0] == state[1][1] == state[2][2] != ' ' or state[0][2] ==
state[1][1] == state[2][0] != ' ':
        return True
    if ' ' not in [state[i][j] for i in range(3) for j in range(3)]:
        return True
    return False

def score(state, ai_player):
    for i in range(3):
        if state[i][0] == state[i][1] == state[i][2] == ai_player or state[0][i]
== state[1][i] == state[2][i] == ai_player:
            return 1
        elif state[i][0] == state[i][1] == state[i][2] != ' ' or state[0][i] ==
state[1][i] == state[2][i] != ' ':
            return -1
```

```
    if state[0][0] == state[1][1] == state[2][2] == ai_player or state[0][2] ==
state[1][1] == state[2][0] == ai_player:
        return 1
    elif state[0][0] == state[1][1] == state[2][2] != ' ' or state[0][2] ==
state[1][1] == state[2][0] != ' ':
        return -1
    return 0

def minimax(state, player, ai_player):
    if is_game_over(state):
        return score(state, ai_player), None, None

    if player == ai_player:
        value = -float('inf')
        for i in range(3):
            for j in range(3):
                if state[i][j] == ' ':
                    state[i][j] = player
                    new_value, _, _ = minimax(state, 'O' if player == 'X' else
'X', ai_player)
                    if new_value > value:
                        value, row, col = new_value, i, j
                    state[i][j] = ' '
        return value, row, col
    else:
        value = float('inf')
        for i in range(3):
            for j in range(3):
                if state[i][j] == ' ':
                    state[i][j] = player
                    new_value, _, _ = minimax(state, 'O' if player == 'X' else
'X', ai_player)
                    if new_value < value:
                        value, row, col = new_value, i, j
                    state[i][j] = ' '
        return value, row, col

def count_empty_spaces(board):
    return sum(row.count(' ') for row in board)

def tic_tac_toe():
    results = []  # Initialize an empty list to store the results
    game_count = 1  # Initialize a counter for the game number
    while True:  # Keep playing games until the user decides to quit
        clear_screen()  # Clear the screen
```

```
        ai_player = input("Choose your symbol (X/O): ").upper()
        while ai_player not in ['X', 'O']:
            print("Invalid input. Please enter 'X' or 'O'.")
            ai_player = input("Choose your symbol (X/O): ").upper()
        ai_player = 'O' if ai_player == 'X' else 'X'

        user_first = input("Do you want to go first? (yes/no): ").lower()
        while user_first not in ['yes', 'no']:
            print("Invalid input. Please enter 'yes' or 'no'.")
            user_first = input("Do you want to go first? (yes/no): ").lower()
        user_first = user_first == 'yes'

        board = [[' ']*3 for _ in range(3)]
        while not is_game_over(board):
            print_board(board)
            if (user_first and count_empty_spaces(board) % 2 != 0) or (not
user_first and count_empty_spaces(board) % 2 == 0):
                move = input("Enter the number (1-9) for your move: ")
                while not move.isdigit() or int(move) < 1 or int(move) > 9 or
board[(int(move)-1) // 3][(int(move)-1) % 3] != ' ':
                    print("Invalid move. Please try again.")
                    move = input("Enter the number (1-9) for your move: ")
                x, y = (int(move) - 1) // 3, (int(move) - 1) % 3
                board[x][y] = 'O' if ai_player == 'X' else 'X'
            else:
                _, x, y = minimax(board, ai_player, ai_player)
                board[x][y] = ai_player
                time.sleep(1)  # pause for a moment before clearing the screen

        print_board(board)
        result = score(board, ai_player)
        if result == 1:
            print('AI wins!')
            results.append(["Game " + str(game_count), "AI wins"])
        elif result == -1:
            print('You win!')
            results.append(["Game " + str(game_count), "You win"])
        else:
            print('Draw!')
            results.append(["Game " + str(game_count), "Draw"])

        # Print the results table
        print("\nResults so far:")
        print(tabulate(results, headers=["Game", "Result"], tablefmt="grid"))
```

```
        play_again = input('Do you want to play again? (y/n): ')
        if play_again.lower() != 'y':
            break  # End the loop and quit the game

        game_count += 1  # Increment the game counter for the next game

tic_tac_toe()
```