AI Project Report

**An Intelligent Tic-Tac-Toe Player using the Minimax Algorithm, Alpha-Beta Pruning, and Heuristic Functions**

shortURL for code and documentation: <https://shorturl.at/anqPR>

long link:

<https://github.com/Moataz51201/Tic-Tac-Toe_Project/tree/main>

A black and white logo

Description automatically generated

Faculty of Computers and Artificial Intelligence Artificial Intelligence

Fall semester 2023-2024

***Team Members :***

All team members at level 3

Artificial Intelligence project

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **ID** | Total grades(35) |
| **1** | مصطفي محمد حسان احمد (IS) | 20210940 |  |
| **2** | عمر عبد العزيز فؤاد عبد العزيز (IS) | 20210608 |  |
| **3** | جمال احمد محمد عطية (IS) | 20210249 |  |
| **4** | محمد احمد عبد الخالق بيومي (IS) | 20210732 |  |
| **5** | محمد اشرف محمد فراج (IS) | 20210744 |  |
| **6** | علي معتز علي الجندي (IS) | 20210586 |  |
| **7** | معتز اسامه محمد سيد الجبيلي (IT ) | 20210953 |  |

**Reference table**

# CHAPTER 1 : Introduction and Overview

# CHAPTER 2 : Proposed Solution

CHAPTER 3 : Applied Algorithms

CHAPTER 4 : Analysis , discussion and future work.

# CHAPTER 1

# Introduction and Overview

# PROJECT IDEA AND OVERVIEW

The project aims to develop an intelligent Tic-Tac-Toe player that employs advanced algorithms and heuristic functions to make strategic decisions during gameplay. Tic-Tac-Toe is a classic two- player game where opponents take turns marking X or O in a 3x3 grid, aiming to form a line of three of their symbols horizontally, vertically, or diagonally. While Tic-Tac-Toe is a simple game, creating an intelligent player involves implementing algorithms that can explore the game tree efficiently, make optimal moves, and potentially predict the opponent's actions.

We will implement it using the minimax algorithm & Alpha-Beta pruning technique & Heuristic functions.

# SIMILAR APPLICATIONS

# Tic Tac Toe OXO (mobile app)

<https://apps.apple.com/us/app/tic-tac-toe-oxo/id1082462952>

**main features:**

-Single & multiplayer game modes: Single Player vs CPU or Two Player Pass & Play

- Three retro themes: Modern, Chalk, Neon Glow

- Pick from 35 fun avatars: choose your player & your opponent’s profile

- Fun celebratory win screen animations

- Easy to pick up and play

- Multiple difficulty levels for all skill levels

Tic Tac Toe (web)

<https://playtictactoe.org/>

# Introduction

The tic-tac-toe is a unique game. A 3x3 grid is formed by using two vertical and two horizontal lines before the game starts. The players can fill the nine spaces with any two different sign normally crosses (‘X’) and noughts (‘O’) symbols [10]. In this study, we use Visual Studio 2019 software to develop the tic-tac-toe program based on artificial intelligent (AI) heuristic approach, min-max algorithm, and alpha-beta pruning algorithm. When there are no players or less number of players be able to win the game, there is an evidence that the AI approach is effective for tic-tac-toe game development. Figure 1. Tic-Tac-Toe game AI is the simulation of human intelligence processes by machines, especially computer systems. Artificial Intelligence exists when a machine can have human-based skills such as learning, reasoning, and solving problems. Nowadays, there are a lot of applications of AI in different fields. In astronomy, AI can be very useful in solving complex universe problems. In healthcare, AI could give a better and faster diagnosis than humans can be done. AI can also be used for gaming purposes, where AI machines can be used to play strategy games like chess, where the machine can perform the best move in different situations. In this study, AI is proposed for computer gaming where it acts as a challenger (opponent) in tic-tac-toe game.

# REVIWE

**Heuristics in Games**

Heuristics in Games Heuristic search in AI is a technique to solve a problem faster than classic methods or to find an approximate solution when classic methods cannot. This is achieved by trading optimality, completeness, accuracy, or precision for speed. The heuristic search technique can evaluate the available information and makes a decision on which branch to follow. The heuristic technique is capable to produce a solution that is good enough for the problem. Nowadays, many effective heuristics methods has been successfully applied in various problem domains. The heuristic methods are possible to be used in game development as they can help to inspire a creative player experience. In computer games, player want to enjoy the games, and programmer needs to have heuristic skills to guide the software for winning the game. Thus, the program needs to be intelligent in order to make user (player) more exciting with the game.

**Minimax Algorithm**

Minimax algorithm is a kind of backtracking algorithm that is used in decision theory. It uses game theory, decision theory, statistics and philosophy to find the optimal move for a player, assuming that the opponent also plays optimally. It is commonly being used in two-player turn-based games such as Tic-Tac-Toe, Chess, etc. The player needs to fulfil two conditions in order to win a game. Firstly, the player needs to maximize the chance to win the game. Secondly, player needs to minimize the opponent’s winning chance. The principle of the minimax search algorithm is to find the optimal path to minimize the maximum possible loss. Two possible results, + shows for computer wins, and - for computer loses. The steps of the Minimax search algorithm are summarized below.

1. Construct the complete game tree.
2. Evaluate scores for leaves using the evaluation function.
3. Back-up scores from leaves to root, considering the player type:

. For max player, select the child with the maximum score.

. For min player, select the child with the minimum score.

1. At the root node, choose the node with maximum value and perform the corresponding move.

**Alpha-Beta Pruning**

Alpa-Beta Pruning algorithm is an optimization algorithm for the minimax algorithm. It reduces the computational time by a huge factor. It allows faster search and even goes into deeper levels in the game tree. It will cut off branches of game trees that does not require to be searched when there is a better movement exists [5]. Alpha-beta pruning seeks to reduce the number of nodes that needs to be evaluated in the search tree by the minimax algorithm. The alpha cut-off process shown in Figure 3, node C (MIN) cannot be more than 1 since node D returns 1. Node B with value of 4 will not search the remaining children of node C, as node A will certainly pick node B over node C for the max node. The remaining children can be aborted if alpha beta, for both the alpha cut-off and beta cut-off [12]. The steps of the Alpha-Beta Pruning algorithm fundamentally is summarized as below.

1. Search down the tree to the given depth.
2. Once reaching the bottom, calculate the evaluation for this node.(i.e. it's utility)
3. Backtrack, propagating values and paths iv. Attain the minimum score of the Alpha.

4. **Development**

This section discuss the development of the tic-tac-toe games using artificial intelligence heuristics. Firstly, we perform pre-production process include rules-based study, game prototype and prototype testing. Secondly, we perform production process including programming and game design. Finally, we perform post-production process including maintenance and testing [6,7,9].

In the first phase, pre-production, we produce a proper prototype for game interface including gameplay ideas and features. It acts as a proof of concept and to test ideas, by modifying some of the game features. Later, we take the prototype testing result to produce the real game interface. This process allows us to gain more understanding about the final interface and the features to let users be more attracted by our game. In the second phase, production, we focused on two parts: implementation of heuristic concept in the source code and provide more detailed design for the game. In the programming part, we provide a proper AI for perfect game [12,13]. Computer (opponent of game) needs to recognize game’s difficulty level for all possible Then, we produce a design that enables the program fix with game rules. We develop a design for sound effects where it is important to give impression for the game’s delivery [8]. Then, we design an attractive theme for the game and find right elements to suit the game theme. We also apply background image for each of the scene. In the third phase, we perform post-production development process including maintenance and testing. Maintenance is very important next to game production, where it is necessary for programmer to recheck the source code and fix all the bugs produced after executing the game design. Then, testing is a process to test the efficiency of the game discussed in next section.

**Conclusion**

This paper has discussed and developed the tic tac toe game using artificial intelligence heuristic approach. It was proved that min-max and alpha-beta pruning algorithms have extraordinary talent in giving a great performance for tic-tac-toe game.

**References**

[1] Russel S, Norvig P. Artificial intelligence: a modern approach. Pearson Education Limited; 2013

[2] Pearl J. Intelligent search strategies for computer problem solving. Addision Wesley. 1984.

[3] Stuart R, Peter N. Artificial intelligence: a modern approach. 2003

[4] Moret BM. Decision trees and diagrams. ACM Computing Surveys (CSUR). 1982 Dec 1;14(4):593-623.

[5] Fullerton T. Game design workshop: a playcentric approach to creating innovative games. CRC press; 2014 Mar 5

# CHAPTER 2

# Proposed Solution

Agent functionalities:

Player interaction: players able to take turns marking X OR O on the 3X3 grid board

Game State Evaluation: there is a mechanism to evaluate the current state of the game, considering factors like potential wins, losses, and draws.

Game Result Display: Display the result of each game, indicating whether it's a win, loss, or draw.

Restart Game: Allow players to restart the game after it concludes.

Tic Tac Toe Game Use Case Description:

Use Case Name: Tic Tac Toe Game

Actors:

Player 1 (inherits from Player)

Player 2 (inherits from Player)

Events:

Start Game(UC1):

Description: The Player actor initiates a new game.

that by default, it initiates in single mode.

Multiplayer (UC2):

Description: Multiplayer mode is an extended event, indicating that players can choose to engage in multiplayer mode

Restart the Game(UC3):

Description: Players can choose to restart the game

Choose Move(UC4):

Description: Players (either Player 1 or Player 2) choose a move during the game.

Chick Statue(UC5):

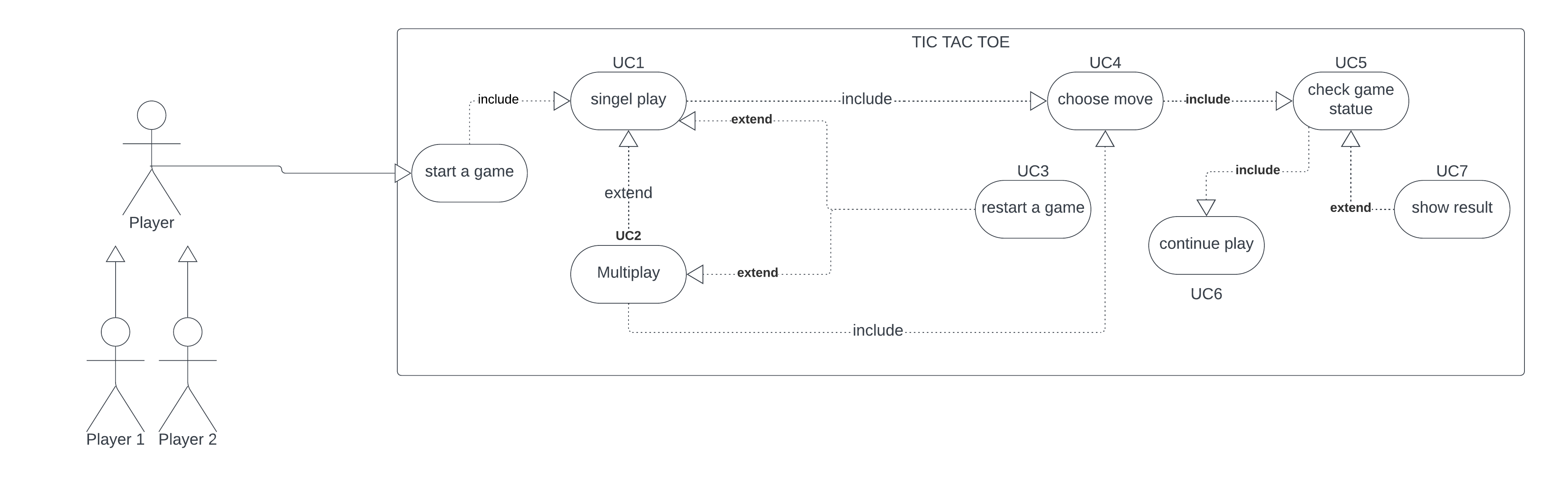
Implement logic to check for a winning condition after each move. This involves checking if any row, column, or diagonal has three matching symbols (X or O).

Continue play (UC6):

Description: If neither a win nor a draw has occurred after a move, the game continues, allowing the next player to make a move.

Show Result (UC7):

Description: Display the result of the game



CHAPTER 3

Applied Algorithms

First Algorithm : MINIMAX

O's Level (Maximizing Player):

At each O node, the algorithm evaluates all possible moves O can make (placing an "O" chip).

For each move, it simulates X's response by creating a child node representing the resulting board state.

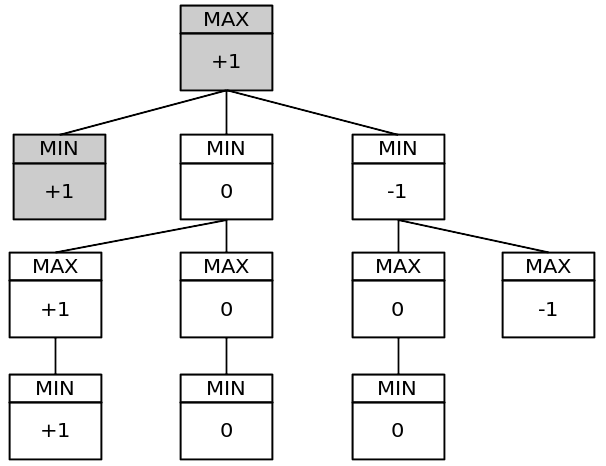
It then recursively calls the minimax algorithm on each child node (minimizing level).

Finally, it assigns a score to the current O node, typically based on how favorable the best outcome from its child nodes is for O (e.g., +1 for a winning move, 0 for a tie).

X's Level (Minimizing Player):

Similar to O's level, but the algorithm aims to minimize O's potential gain.

It assigns scores to X nodes based on the worst outcome for O among its child nodes.

4. Choosing the Best Move: ultimately chooses the move from its level that leads to the highest score (maximizing its chances of winning). 

totally , the diagram provides a good overview of how the minmax algorithm works for Tic-Tac-Toe (when the root case has 3 possible moves). it is important to note that the minmax algorithm can be computationally expensive

A diagram of a diagram

Description automatically generated

More specific diagram for minimax alogrithem

A diagram of a diagram

Description automatically generated

This image showed an overview of ALL game tree

**Pseudocode(minimax)**

1. function minimax (board, maximizingPlayer):
2. if the game is over:
3. return the utility value of the current board.
4. if maximizingPlayer:
5. bestScore = -infinity
6. for each possible move:
7. score = minimax(resultingBoard, False)
8. bestScore = max(bestScore, score)
9. return bestScore
10. else:
11. bestScore = +infinity
12. for each possible move:
13. score = minimax(resultingBoard, True)
14. bestScore = min(bestScore, score)
15. return bestScore

Second algorithm: alpha-beta

Here's how the alpha-beta pruning algorithm works, as shown in the diagram:

The algorithm starts by exploring all possible moves for the current player (usually denoted as Max).

For each of these moves, the algorithm then considers all possible moves that the opponent (usually denoted as Min) could make in response.

As the algorithm explores the game tree, it keeps track of two values: alpha and beta.

Alpha is the highest score that Max is guaranteed to achieve, based on the moves explored so far.

Beta is the lowest score that Min can achieve, based on the moves explored so far.

If the algorithm reaches a node where the score for Max is already greater than or equal to beta, it can stop exploring that branch of the tree. This is because the opponent will never choose a move that leads to a worse score for themselves than beta.

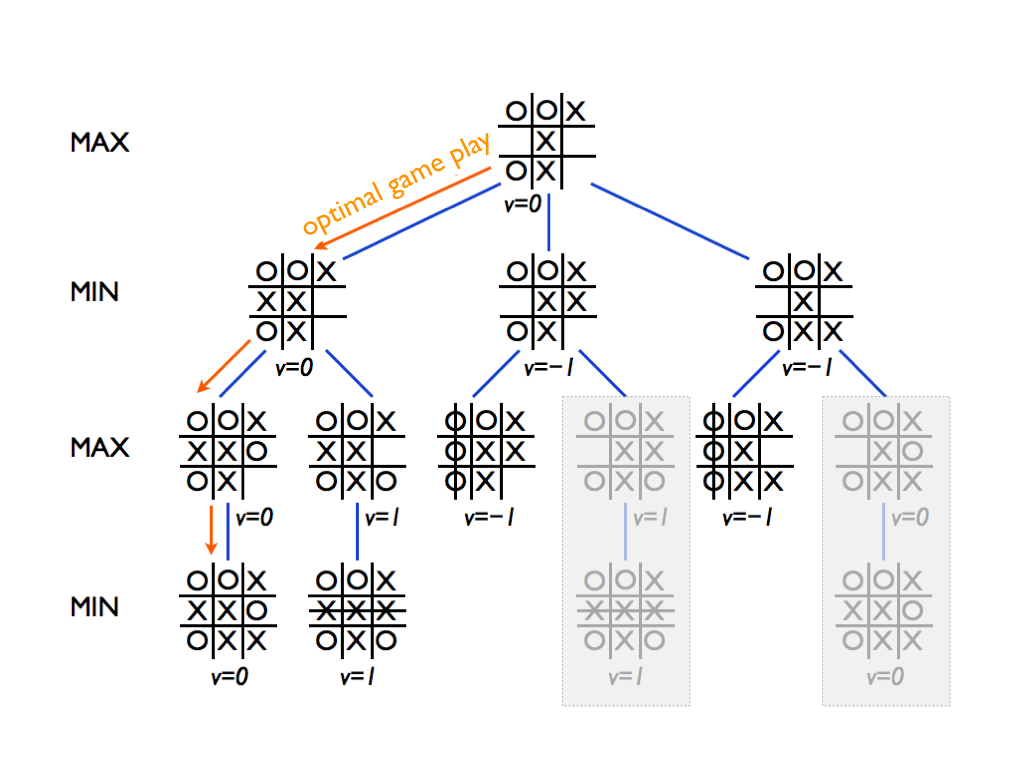
Similarly, if the algorithm reaches a node where the score for Min is less than or equal to alpha, it can stop exploring that branch of the tree. This is because Max will never choose a move that leads to a worse score for themselves than alpha.

By pruning these branches, the alpha-beta pruning algorithm can significantly reduce the number of nodes that need to be explored, making the search more efficient.

A diagram of a diagram

Description automatically generated

This algorithm is used to improve the efficiency of minimax search in two-player zero-sum games like tic tac toe and others



On the left side of the tree, it starts with the prompt “Is the center marked?”.

If the center is marked, the decision tree branches into two possibilities:

“Is there an open corner?”

If there is an open corner, the best move is to take the corner.

If there are no open corners, the best move is to take an open edge.

If the center is not marked, the best move is to take the center.

The decision tree continues to branch out in this way, considering all possible moves that could be made by both players and recommending the best move for the current player based on the potential outcomes.

Overall, the decision tree provides a basic strategy for playing Tic-Tac-Toe. However, it is important to note that Tic-Tac-Toe is a relatively simple game with a limited number of possible moves. More complex games, such as Chess or Go, would require much more sophisticated decision trees to be effective.

Here are some additional details about the decision tree in the image:

The squares on the board are labeled with numbers to represent the different moves that can be made.

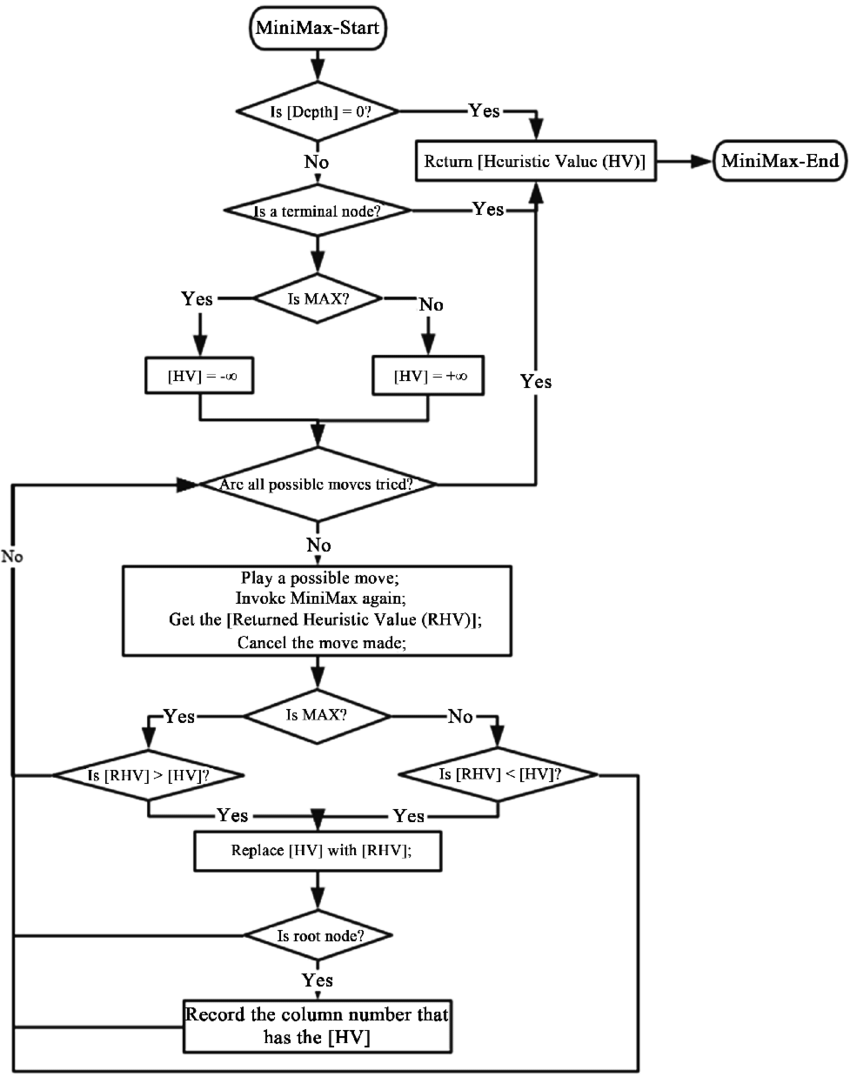
The values at the end of each branch represent the expected outcome of the game for the current player if they make the move at the beginning of the branch. A higher value means that the outcome is more favorable for the player.

The decision tree assumes that the opponent is playing rationally and trying to win the game.

**Pseudocode (alpha beta)**

1. function alpha\_beta(node, depth, alpha, beta, maximizing\_player)
2. if depth == 0 or node is a terminal node
3. return the heuristic value of node
4. if maximizing\_player
5. max\_eval = negative infinity
6. for each child of node
7. eval = alpha\_beta(child, depth - 1, alpha, beta, False)
8. max\_eval = max(max\_eval, eval)
9. alpha = max(alpha, eval)
10. if beta <= alpha
11. break
12. return max\_eval
13. else
14. min\_eval = positive infinity
15. for each child of node
16. eval = alpha\_beta(child, depth - 1, alpha, beta, True)
17. min\_eval = min(min\_eval, eval)
18. beta = min(beta, eval)
19. if beta <= alpha
20. break
21. return min\_eval

MINIMAX WITH HEURISTIC FUNCTIONS



while minimax without heuristics guarantees optimal decisions, it can be computationally expensive. Minimax with heuristics sacrifices optimality for efficiency, making it more suitable for scenarios where the complete search is not feasible. The choice between the two depends on the specific requirements of the application and the available computational resources.

### **Pseudocode for Minimax Algorithm with Heuristic Function**

1. function minimax(board, maximizingPlayer, heuristic\_function):
2. if game\_over(board) or depth\_limit\_reached:
   1. return evaluate(board, heuristic\_function)
3. if maximizingPlayer:
   1. bestScore = -INFINITY
   2. for each possible move in board:
      1. if move is legal:
      2. board.apply(move)
      3. score = minimax(board, False, heuristic\_function)
      4. board.undo\_move(move)
      5. bestScore = max(bestScore, score)
   3. return bestScore
4. else:
   1. bestScore = +INFINITY
   2. for each possible move in board:
      1. if move is legal:
      2. board.apply(move)
      3. score = minimax(board, True, heuristic\_function)
      4. board.undo\_move(move)
      5. bestScore = min(bestScore, score)
   3. return bestScore
5. function evaluate(board, heuristic\_function):
6. return heuristic\_function(board)
7. function game\_over(board):
8. return check\_for\_win(board) or check\_for\_draw(board)
9. function check\_for\_win(board):
10. # Check if any player has won
11. # Return true if yes, else false
12. function check\_for\_draw(board):
13. # Check if the game is a draw
14. # Return true if yes, else false

A diagram of a game

Description automatically generated

### **Pseudocode for Minimax Algorithm with Heuristic and Symmetric Reduction:**

1. function minimax(board, depth, maximizingPlayer):
2. if board is a terminal node:
3. return heuristic\_evaluation(board, maximizingPlayer)
4. if maximizingPlayer:
5. maxEval = -infinity
6. for each valid move in available\_moves(board):
7. make move in board
8. eval = minimax(board, depth + 1, False)
9. undo move in board
10. maxEval = max(maxEval, eval)
11. return maxEval
12. else: # minimizingPlayer
13. minEval = +infinity
14. for each valid move in available\_moves(board):
15. make move in board
16. eval = minimax(board, depth + 1, True)
17. undo move in board
18. minEval = min(minEval, eval)
19. return minEval
20. function heuristic\_evaluation(board, maximizingPlayer):
21. if maximizingPlayer and board is a winning state for 'O':
22. return 1
23. else if not maximizingPlayer and board is a winning state for 'X':
24. return -1
25. else if maximizingPlayer and board is a winning state for 'X':
26. return -1
27. else:
28. return 0
29. function available\_moves(board):
30. return list of empty positions in the board
31. function generate\_symmetric\_states(board):
32. symmetric\_states = [board]
33. add horizontal reflection of the board to symmetric\_states
34. add vertical reflection of the board to symmetric\_states
35. add diagonal reflection of the board to symmetric\_states
36. return symmetric\_states

CHAPTER 3

Analysis , discussion and future work

Analysis of the results, what are the insights?

Pseudocode for Minimax Algorithm:

Insights:

The minimax algorithm correctly follows the standard structure for a recursive search.

The base case checks for terminal states and returns the utility value.

The algorithm explores all possible moves for both players.

Pseudocode for Alpha-Beta Pruning:

Insights:

The alpha-beta pruning algorithm effectively prunes branches of the search tree based on the values of alpha and beta.

It uses a depth-first search approach and can significantly reduce the number of nodes explored.

Pseudocode for Minimax with Heuristic Function:

Insights:

The minimax algorithm with a heuristic function uses the heuristic to evaluate non-terminal states.

It applies the heuristic to estimate the desirability of a board state without exploring all possible moves fully.

Pseudocode for Minimax with Symmetry Reduction and Heuristic Reduction:

Insights:

The pseudocode includes a minimax algorithm with symmetry reduction, where symmetric states are generated and considered.

Heuristic reduction is applied by using a heuristic\_evaluation function for terminal states instead of fully exploring the subtree.

Analysis of the Heuristic Evaluation Function:

Insights:

The heuristic\_evaluation function assigns values based on winning states for 'O' and 'X'.

It correctly returns 1 for a winning state for 'O' (maximizing player) and -1 for a winning state for 'X' (minimizing player).

The function considers symmetric states in the check\_for\_win function, improving efficiency.

COMPARING BETWEEN ADVANTAGES AND DISADVANTAGE BETWEEN EACH ALGORITHM:

1. minimax   
   **Optimal Decision Making:**
   * Minimax guarantees optimal decision-making for both players in a game where the outcome is determined solely by the actions of the players and there is no chance involved.
2. **Completeness:**
   * The algorithm explores all possible moves and their consequences, ensuring that it considers every option available in the game.
3. **Widely Applicable:**
   * Minimax can be applied to a broad range of games, including classic board games like chess, checkers, and Tic Tac Toe.
4. **Mathematically Sound:**
   * The algorithm is based on a sound mathematical foundation, making it a well-defined and rigorous approach to decision-making in games.
5. **Negamax Variant:**
   * The minimax algorithm can be simplified using the negamax variant, which reduces the need for separate functions for maximizing and minimizing players.

### Disadvantages:

1. **Exponential Growth in Complexity:**
   * The number of nodes (game states) explored by the minimax algorithm grows exponentially with the depth of the game tree. In complex games, this can lead to a large number of computations and increased time complexity.
2. **Memory Requirements:**
   * The algorithm may require significant memory resources, especially in games with a large branching factor and depth. Storing and exploring all possible game states can lead to high memory usage.
3. **Not Suitable for Real-Time Games:**
   * Minimax is not well-suited for real-time games or situations where decisions must be made quickly. The exhaustive search can be too time-consuming for applications that demand rapid responses.
4. **Assumes Perfect Knowledge:**
   * Minimax assumes that both players have perfect knowledge of the game, including knowledge of all possible moves and their consequences. In real-world scenarios, this assumption may not hold.
5. **No Consideration of Opponent's Strategy:**
   * The algorithm assumes that the opponent plays optimally. If the opponent deviates from optimal play, the minimax algorithm may not perform as well.
6. **Difficulty Handling Games with Chance:**
   * Minimax is designed for deterministic games, and incorporating chance elements (randomness) into the game makes it more challenging to apply minimax directly.
   * Alpha beta pruning
7. **Reduction in Search Space:**
   * Alpha-beta pruning significantly reduces the number of nodes (game states) that need to be evaluated by pruning branches that cannot influence the final decision. This leads to a substantial reduction in the search space.
8. **Improved Time Complexity:**
   * By eliminating unnecessary evaluations of subtrees, alpha-beta pruning improves the time complexity of the algorithm. This makes it more efficient, especially in games with deep and complex decision trees.
9. **Memory Efficiency:**
   * As fewer nodes need to be stored and evaluated, alpha-beta pruning reduces the memory requirements of the algorithm, making it more memory-efficient compared to the standard minimax algorithm.
10. **Enhanced Applicability to Real-Time Systems:**
    * Alpha-beta pruning is particularly useful in real-time applications or systems where decisions must be made quickly. The reduction in search space contributes to faster decision-making.
11. **Guaranteed Optimal Results:**
    * Alpha-beta pruning retains the optimality of the minimax algorithm. It guarantees the same optimal results as the standard minimax algorithm but achieves them more efficiently.
12. **Easy Implementation:**
    * Alpha-beta pruning can be seamlessly integrated into existing minimax implementations, making it a practical and straightforward optimization to apply.

### Disadvantages:

1. **Implementation Complexity:**
   * While the basic idea of alpha-beta pruning is conceptually simple, its efficient implementation can be challenging. Ensuring correct pruning while managing alpha and beta values requires careful coding.
2. **Assumes Perfect Ordering of Moves:**
   * Alpha-beta pruning assumes that the best moves are considered first in the search. If the ordering of moves is suboptimal, the benefits of pruning may be diminished, and the algorithm may not achieve the maximum reduction in search space.
3. **Limited Applicability to Non-Deterministic Games:**
   * Alpha-beta pruning is most effective in deterministic games with no chance elements. In games with randomness or uncertainty, such as those involving dice rolls, its effectiveness may be reduced.
4. **Dependency on Game Tree Structure:**
   * The efficiency of alpha-beta pruning depends on the structure of the game tree. In certain cases, where the optimal move is located deep in a branch with many nodes, the pruning benefits may be less pronounced.

### Advantages:

1. **Faster Decision Making:**
   * The heuristic function provides an estimate of the desirability of a game state without exploring all possible moves, allowing the algorithm to make quicker decisions, especially in cases where the search space is extensive.
2. **Applicability to Complex Games:**
   * In games with large and complex decision trees, a heuristic function allows the algorithm to focus on the most promising moves, avoiding a full exploration of the entire tree. This is particularly beneficial in games with deep and branching possibilities.
3. **Increased Search Depth:**
   * The use of heuristics enables the algorithm to explore deeper into the game tree within a reasonable amount of time, providing a better understanding of the game state and potentially leading to more informed decisions.
4. **Adaptability to Imperfect Information:**
   * Heuristics can help in scenarios where complete information about the game state is unavailable. The algorithm can make reasonable decisions based on the estimated desirability of the states, even if they are not fully explored.

### Disadvantages:

1. **Risk of Inaccurate Evaluation:**
   * Heuristic evaluations are approximations, and there's a risk that they might not accurately represent the true desirability of a game state. Inaccurate heuristics can lead to suboptimal decisions.
2. **Dependency on Heuristic Quality:**
   * The effectiveness of the algorithm heavily relies on the quality of the heuristic function. If the heuristic function is poorly designed or doesn't capture the nuances of the game well, it may not contribute meaningfully to decision-making.
3. **Difficulty in Designing Heuristics:**
   * Designing effective heuristics for complex games can be challenging. Finding a balance between accuracy and computational efficiency is crucial, and it may require domain-specific knowledge.
4. **Potential for Overlooking Optimal Moves:**
   * Depending on the heuristic function and the search strategy, the algorithm may overlook optimal moves in certain situations, leading to suboptimal decisions.
5. **Increased Implementation Complexity:**
   * Incorporating a heuristic function adds complexity to the implementation. Developers need to carefully design, test, and fine-tune the heuristic to ensure it contributes positively to the decision-making process.

- Why did the algorithm behave in such a way?

The algorithm is designed for a two-player game with perfect information, such as Tic-Tac-Toe. The provided code structure is clear, with functions for the minimax algorithm, evaluation using a heuristic function, and checking for game over, wins, and draws.

However, a few points need clarification or potential modifications:

Depth Limit

The code mentions depth\_limit\_reached as a condition to stop further exploration. It's crucial to define and implement this depth limit correctly. The depth limit determines how deep the algorithm explores the game tree, and setting it appropriately is essential for balancing computational resources and decision quality.

Heuristic Function:

The heuristic function (heuristic\_function(board)) is called to evaluate non-terminal states. The quality of the heuristic function directly impacts the algorithm's performance. Ensure that the heuristic function effectively captures the desirability of a board state and aligns with the game's objectives.

Legal Moves:

In the code, it checks if a move is legal before applying it. Make sure the code accurately identifies legal moves to avoid errors in the exploration of the game tree.

Evaluation at Terminal States:

The evaluate function simply calls the heuristic function. It's common to return a specific value (e.g., +INF, -INF, 0) at terminal states without invoking the heuristic. The terminal states should be explicitly handled in the minimax function.

Undoing Moves:

The code uses board.undo\_move(move) to revert changes after evaluating a move. Ensure that the undo operation correctly restores the board state to its previous condition.

INFINITY Constant:

The code uses +INFINITY and -INFINITY to initialize scores. Make sure these constants are appropriately defined in the code or imported from a library.

Handling Game Over

The game\_over function is a crucial part of the algorithm. Make sure it correctly determines if the game is over, either due to a win or a draw.

Future Modifications

Consider adding alpha-beta pruning to further optimize the minimax algorithm by reducing the number of nodes explored.

Experiment with different heuristic functions to improve the algorithm's decision-making efficiency and accuracy.

Fine-tune the depth limit based on the complexity of the game and available computational resources.

What might be the future modifications you’d like to try when solving this problem?

Enhanced Heuristic Functions:

Continuously experiment with and refine your heuristic functions. The quality of the heuristic can significantly impact the algorithm's performance. Consider exploring machine learning techniques to train a heuristic or reinforcement learning for improved decision-making.

Dynamic Difficulty Adjustment:

Implement dynamic difficulty adjustment based on the player's skill level. You can adjust the search depth or modify the heuristic function dynamically to provide a more challenging experience for advanced players while allowing less experienced players to compete.

Optimizations for Early Game Endings:

Optimize the algorithm to recognize early game endings and focus on deepening the search in critical game states. This can improve efficiency, especially in scenarios where the outcome is clear well before reaching the maximum search depth.

shortURL for code and documentation: <https://shorturl.at/anqPR>

long link:

<https://github.com/Moataz51201/Tic-Tac-Toe_Project/tree/main>