AI PROJECT DOCUMENTATION

An Intelligent Connect-Four Player using an Alpha-Beta Depth-First algorithm

ID	**الاسم**
20210441	سيف حسام الدين محمد تهامي
20210453	شريف اشرف عوض الباجوري
20210435	سيف الدين اسامة ربيع
20210410	سلمى انور انور عبد العزيز

20210704	مـارتـيـنـا سـلـيـمـان سـامـي
20210422	سـمـا جـمـال مـكـرم عـبـد الـجـلـيـل
20210705	مـارتـيـنـا مـوسـى رسـمـي زكـي

Department of Artificial Intelligence (AI)
Team Members:

PROJECT IDEA & OVERVIEW

Our project idea is to build an artificial Connect-Four player , that can play games against a human opponent. Connect Four is a two-player connection board game, in which the players choose a color and then take turns dropping colored piece into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective of the game *is to be the first to form a horizontal, vertical, or diagonal line of four of one's own pieces. Connect Four is a solved game. The first player can always win by playing the right moves.*

We will implement it using the Alpha-Beta pruning algorithm using multiple heuristic functions that are different and then evaluating the best one.

Our development platform consists of:

- Python 3.11 as our programming language
- Numpy (python library)
- Pygame (python library)

Similar Applications

- ***Connect 4 Online (web)***

https://papergames.io/en/connect4

Main features:

1. *It has 4 modes to play, online, with a friend, with a robot or in a created tournament.*
2. *It offers you the choice of the size of the board (7x6 or 8x7 etc.…)*

3. *It offers you the option to pick the time per turn, and total minutes per player*
4. *It has a leaderboard and a ranking system for the players with the highest score*

- ***Four In A Row (mobile)***

https://apps.apple.com/eg/app/four-in-a-row-classic-games/id604921715

Main features:

1. *Choose between 3 modes of difficulty (easy, medium, hard)*
2. *Play against real humans (multiplayer)*
3. *Challenge friends on the same device (2 players) or against AI*
4. *Colorful themes*
5. *Leaderboard and high score ranking*

- *4 In A Line! Connect 4 by Hasbro (web)*

https://www.mathsisfun.com/games/connect4.html

Main features:

1. *Choose between 4 modes of difficulty (too easy, easy, medium, hard)*
2. *Challenge friends on the same device (2 players) or against AI*
3. *Choose color of the disk*
4. *Name your player*

5. *Scoring system to keep track of match history*

Literature Review:

One of the first areas of artificial intelligence (AI) research was computer chess. Given the potential that people think computers have for the near future, this is not surprising. With these possibilities, it was thought possible to make computers perform tasks that require human intelligence. (Allis, 1988)
Much research has been done over the years in the field of computer chess. Some outcomes are specific to chess, while others have broader implications. Various game tree search techniques are used in many other areas of computer science in general, and artificial

intelligence in particular. Another result of (Allis, 1988) research, it has been observed in the field of computer chess that it is very difficult to develop programs that play chess in a manner like how human play chess. it became clear that the future of computer chess was in full- width—or so-called brute force— searching on fast hardware as (Buro, 2002) suggested.
We are dealing with a two-player zero-sum game with perfect information. The value of such a game is the result of perfect play by both sides and can be found by recursively unrolling all possible continuations from each game state until a known outcome state is reached. Then the minimax rule is used to propagate these resulting values to the initial state. This extended state space is a tree, often called a game tree, where the root of the tree is the initial state, and the leaf nodes are the final states.
The point of game tree search is to insulate oneself from errors in the evaluation function. The standard approach is to grow a full width tree as deep as time allows, and then value the tree as if the leaf evaluations were exact. The alpha-beta algorithm implements this with great computational efficiency (Baum & Smith, 1997).
The focus is on the traditional use of AI by simulating the typical gameplay of Connect-4's classic board game. Connect-4 is classified as a zero-sum adversarial game because one player's advantage becomes the opponent's disadvantage. In the context of AI, the concept of adversarial search is specifically considered. It examines the problems that arise when someone tries to plan ahead in a world where other agents are playing against them. The game is represented as a multi-agent environment.

The Minimax Algorithm

The problem is to design or propose a solution for this connect-4 game. This game can be solved very easily for two players by using matrices or arrays. However, a solution for one player against computer bot (AI) is proposed. A Mini-Max algorithm is used to solve this game. The user is playing against a computer that uses the minimax algorithm to generate the game state. Mini-Max falls into the category of backtracking algorithms. This algorithm has many uses and is used in decisionmaking and game theory to find the optimal move for a player when the opponent is also playing optimally. It is commonly used in two-player games, such as tic-tac-toe and chess, where players take turns making moves. There are two players in mini-max which are called maximizer and minimizer. Maximizer aim to get the highest possible score, while Minimizer do the exact opposite and try to get the lowest possible score. Each board state is assigned a value. In certain states, the board score tends to be positive when Maximizer dominates. If Minimizer dominates in this board state, it tends to be a negative value.

Minimax Problems

Mini-max algorithm has some defects and limitations, the player has a lot of choices, and the branching factor of Connect 4 games are huge therefore, it slows down as you go deeper. On average, the branching factor for Connect-4 tends to approach 7. This means that 7 subtrees are created per turn. Another drawback of the minimax algorithm is that each board state must be accessed twice. First time to find its children, second time to evaluate the heuristic value (utility value). Also, it is used only for games where the game tree is finite

The Alpha-Beta Algorithm

A proposed solution to this is to try and optimize the minimax algorithm, and that can be done by applying the Alpha-Beta algorithm. Alpha-beta is the most common way to search game trees in adversarial board games such as chess, checkers, and Othello. It's much more efficient than a simple brute-force minimax search because it allows you to prune large parts of the game tree while still ensuring the correct game tree values. However, the number of nodes visited by the algorithm increases exponentially as the search depth increases.
This obviously limits the scope of the search, as the game program must satisfy external time constraints. often you have only a few minutes to make a decision. In general, the more forward-looking the program, the better the quality of the play.
Alpha-beta pruning is not a new algorithm. It is basically an advancement or technique of the minimax algorithm that reduces the time complexity of the minimax algorithm by half and speeds up the AI bot's decision time for one move.

The Algorithm Details

Alpha is the best value that the maximizer currently can guarantee at that level or above, while Beta is the best value that the minimizer currently can guarantee at that level or above.
Initially, both players start with the worst possible value, so alpha is assigned a negative infinity value and beta is assigned a positive infinity value. If a particular branch of a particular node is chosen, the minimum possible score that the minimizing player can get may be less than the maximum possible score that the maximizing player can definitely get, Beta is below Alpha, in this case, the parent node should not select this node, as the parent node's

score will be worse. So, you don't need to investigate other branches of this node.

Algorithm is as follows:

1. *First check the current node if it is a leaf or terminal node return its value.*
2. *If not, then if it is the max's turn then check each node for maximum value by using mini-max algorithm.*
3. *If the best value is greater than the value for min player (i.e., alpha>=best) then break the loop and return best value/move.*
4. *Now, check the minimum values for the min player by using mini-max algorithm and then check if beta>=alpha breaks loop and return best move for min.*
5. *Implementing minimax search using alpha-beta pruning is relatively straightforward. A normal depth-first search with limited depth is used.*

Time complexity is:

- *Worst case: The worst case complexity of alpha-beta will be the same as the minimax best case complexity i.e. $O(b^d)$, where b is branching factor and d is depth.*
- *Best case: The best case will be when all the moves are optimal, then the pruning can be done and alpha-beta can prune almost half node so complexity is $O(b^d/2)$, It means the alpha-beta algorithm can search more than half nodes than mini- max algorithm in the same time, which makes it faster.*

Implementing Algorithm on The Connect-4 Game:

connect-4 is a game with 7x6 dimensions, when the game begins, the board fills from the bottom and play continues until the player connects four pieces or elements or all the blocks on the board are filled, resulting in a draw.
It has 3 states, Win (1), Tie (0), Lose (-1).
First, the mini-max algorithm is implemented and then converted into alpha-beta prune to make the game more efficient. Game tree of this game will consist of 7 branches from the root node. As the game progresses the game

tree will expand and almost every node in tree will have 7 branches. So, the complexity of game will increase with the depth of the game.
The application of heuristic function is to coordinate robot agents in adversarial environment. After the AI exhaustively evaluates the tree to some depth (steps to think ahead), it evaluates the board using a heuristic function. Eventually, as the end of the game approaches, the AI finds win-loss conditions limited to the specified depth and plays perfectly .

The overall idea

The overall idea here is that the heuristic function can guide the AI to an advantageous position. A function is implemented to calculate the value of the board according to the placement of the pieces on the board. This function is often called the evaluation function, also known as the heuristic function. The basic idea behind the heuristic function is to assign a high value to the board for maximizer's turn and a lower value to the board for minimizer's turn [(Nasa, Didwania, Maji, & Kumar, 2018)].
In the actual algorithm it is minimax()'s responsibility to calculate the most optimal move for the AI to play, with current board and the search depth as parameters. The two other parameters are alpha and beta. It is to be kept in mind that the AI is the maximizer and human is the minimizer. The first thing that is done is the calculation of the score (value) of the present board. Next check if the current state is a terminal state or not. The next step is to create multiple game states for the next iteration or stage. For each game state created call the minimizer function. The minimizer function is structurally same as the maximizer function. The board score is computed first. Then it is checked if the game state is a terminal state or not. A loop is run to generate the next iteration of game states. For each game state generated the maximizing move for the AI is calculated. The principle of alpha beta pruning is applied to reduce the number of game states that needs to be checked. This

function returns the minimum score of the human player [(Nasa, Didwania, Maji, & Kumar, 2018)].
The heuristic function calculates one parameter, namely computer points. Next, two variables are initialized whose job is to save the winning position of the respective player and the maximum score. The next step is to determine score through the amount of available chips. It is done through incrementing computer points whenever the particular field of the game state is empty or not. The final stage is to check if the current state is a winning condition or not [(Nasa, Didwania, Maji, & Kumar, 2018)].
The actual function that calculates the value of game state calculates six parameters, namely vertical points, horizontal points, centre points, diagonal one point, diagonal two points and the final points. Next a nested loop is run for each column, where we rate the column according to the function and the points to the respective column. The same procedure is carried out in calculating the horizontal points and the centre points and the two diagonal points. The final points are nothing but the summation of the former five points [(Nasa, Didwania, Maji, & Kumar, 2018)].

CONCLUSION

In this paper, we implemented a Connect-4 game using two algorithms and examined their comparison. The first algorithm is the minimax algorithm, and the second algorithm is one is Mini-Max with Alpha-Beta Pruning, an optimized version of the Mini-Max algorithm. The study found that at the same level of difficulty, the two algorithms behave very differently in terms of the number of iterations performed and the time taken with alpha-beta pruning taking much less time and performing very few iterations than minimax to generate the game state [(Nasa, Didwania, Maji, & Kumar, 2018)].
Despite many evaluation and search improvements, programs still show weaknesses in the opening phase stemming from a lack of strategic planning. To mitigate this problem, programs utilize opening books in which move sequences or positions together with moves are stored [4].
Today, many game-playing programs are attached to servers, playing against human players and other programs 24 hours a day. In order to prevent repeated losses, it therefore has become necessary for programs to update

their opening books automatically without human intervention [(Buro, 2002)].
In multi-game matches players are facing simple but effective playing strategies that cannot be met by the well-known game-tree search techniques alone. Perhaps the most obvious and simple one is the following: "If you have won a game, try it the same way next time". A player with no learning mechanism and no move randomization follows this strategy, but is also a victim of it, because he does not deviate and therefore can lose games twice in the same way. To avoid this, the player must find reasonable move alternatives. He can do so passively by copying opponent's moves when colors are reversed. This elegant method lets the opponent show you your own errors, so you can play the opponent's winning moves next time by yourself. In this way, even an otherwise stronger opponent can be compromised, because—roughly speaking—eventually he is playing against himself. Thus, copying moves makes it necessary to come up with good move alternatives actively. To do so, a player must understand his winning chances after deviations from known lines [(Buro, 2002)].
These basic requirements of a skilled match strategy lead directly to an algorithm for guiding opening book play based on mini-max search. The procedure builds a game tree from played variations—starting with the initial game position—and labels the leaves depending on the particular game outcomes. Moreover, in each interior node the algorithm evaluates all moves not played so far and adds the edge and node corresponding to the heuristically best move together with its evaluation to the tree. Given such a tree, it is easy to guide the opening book play by propagating leaf evaluations to the root using the minimax algorithm and extending lines by expanding minimax leaves. This algorithm avoids losing the same way and explores new variations [(Buro, 2002)].
Good human players are conducting a highly selective look-ahead search in which they only rarely miss decisive variations. On the other hand, the original minimax algorithms waste most of their time by analyzing irrelevant lines. In the presence of good evaluation functions, selective Alpha-Beta searches can approximate the very focused human search behavior. However, programs still must search many more nodes in order to come up with decisions of comparable quality. Future research on better evaluation schemes, selective search, and planning will benefit from machine learning advances and certainly result in a decreased search effort [(Buro, 2002)].

References

Rivest, R. L. (1987). Game Tree Searching By Min/Max Approximation. Artificial Intelligence, 34(1), 77-96.

Nasa, R., Didwania, R., Maji, S., & Kumar, V. (2018). Alpha-beta pruning in mini-max algorithm- an optimized approach for a connect-4 game. Int. Res. J. Eng. Technol, 1637-1641.

Buro, M. (2002). Improving Heuristic Mini-Max Search By Supervised Learning. Artificial Intelligence, 134(1-2), 85-99.

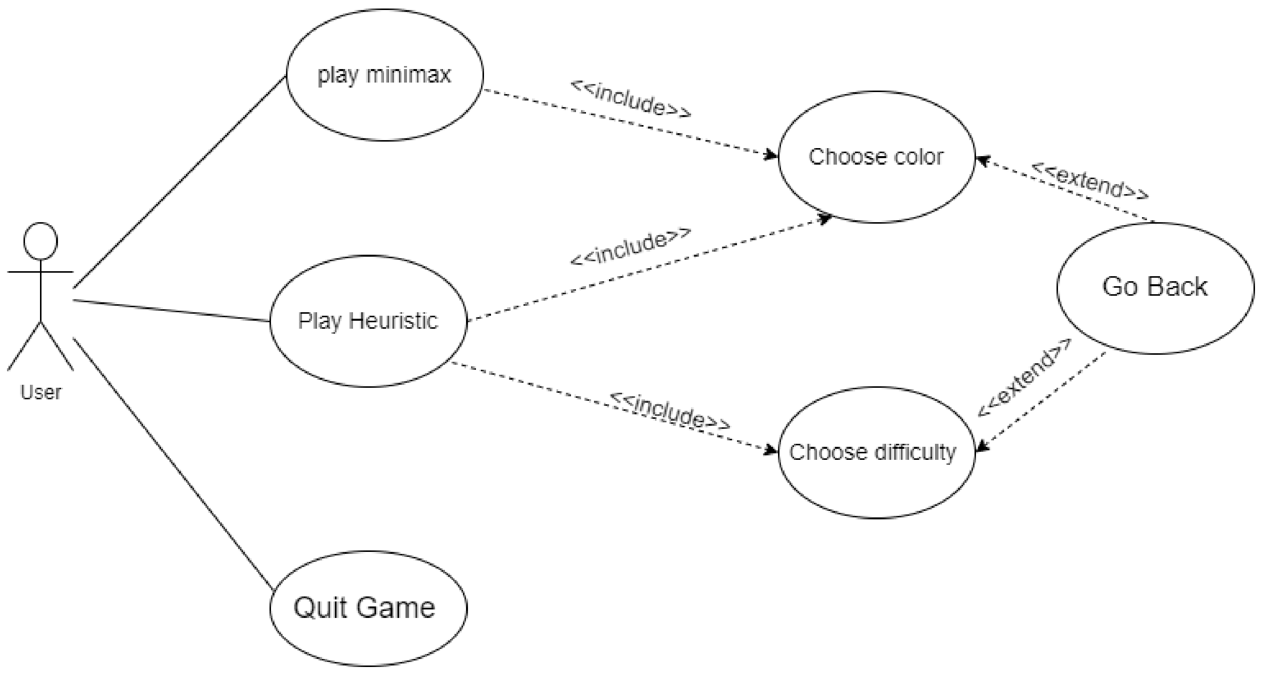
Baum, E. B., & Smith, W. D. (1997). A Bayesian Approach To Relevance In Game Playing. Artificial Intelligence, 97(1-2), 195-242.

Allis, L. V. (1988). A Knowledge-Based Approach of Connect-Four. J. Int. Comput. Games Assoc, 11(4), 165.

Main Functionalities

Our Connect- 4 Game has several features, like:
1-Choosing Algorithm
2-Choosing the color of the pieces
3-Choosing the difficulty of the AI
4-Playing connect-4 with AI

Use-case Diagram of System:



Play with AI description:

Identifier & Name	*User play with ai*
Initiator	*User*
Goal	*User is able to play connect-4 with AI*
Precondition	*None*
Postcondition	*User has to pick his chosen color & difficulty*

Main Success Scenario	*1- User presses play button* *2- User picks his desired color (UC3) 3- User picks the difficulty of AI (UC4) 4- User starts playing the game* *5- After finishing game User returns to Main Menu*
Extensions	*User presses back* • *User goes back to main menu* *User presses back* • *User goes back to Choose Color screen*

Choose Color description:

Identifier & Name	*User choose color*
Initiator	*User*
Goal	*User chooses red or blue as his piece color*
Precondition	*User chose to play game*
Postcondition	*User has to pick difficulty of AI or None if local*
Main Success Scenario	*5- User picks either red or blue*
Extensions	*User presses back* • *User goes back to main menu*

Choose Difficulty description:

Identifier & Name	*User choose difficulty*
Initiator	*User*
Goal	*User chooses easy, medium, or hard as AI difficulty*
Precondition	*User chose to play game and User chose his color*
Postcondition	*User plays game*
Main Success Scenario	*1- User picks either easy, medium, or hard*

Extensions	*User presses back* • *User goes back to choose color screen*

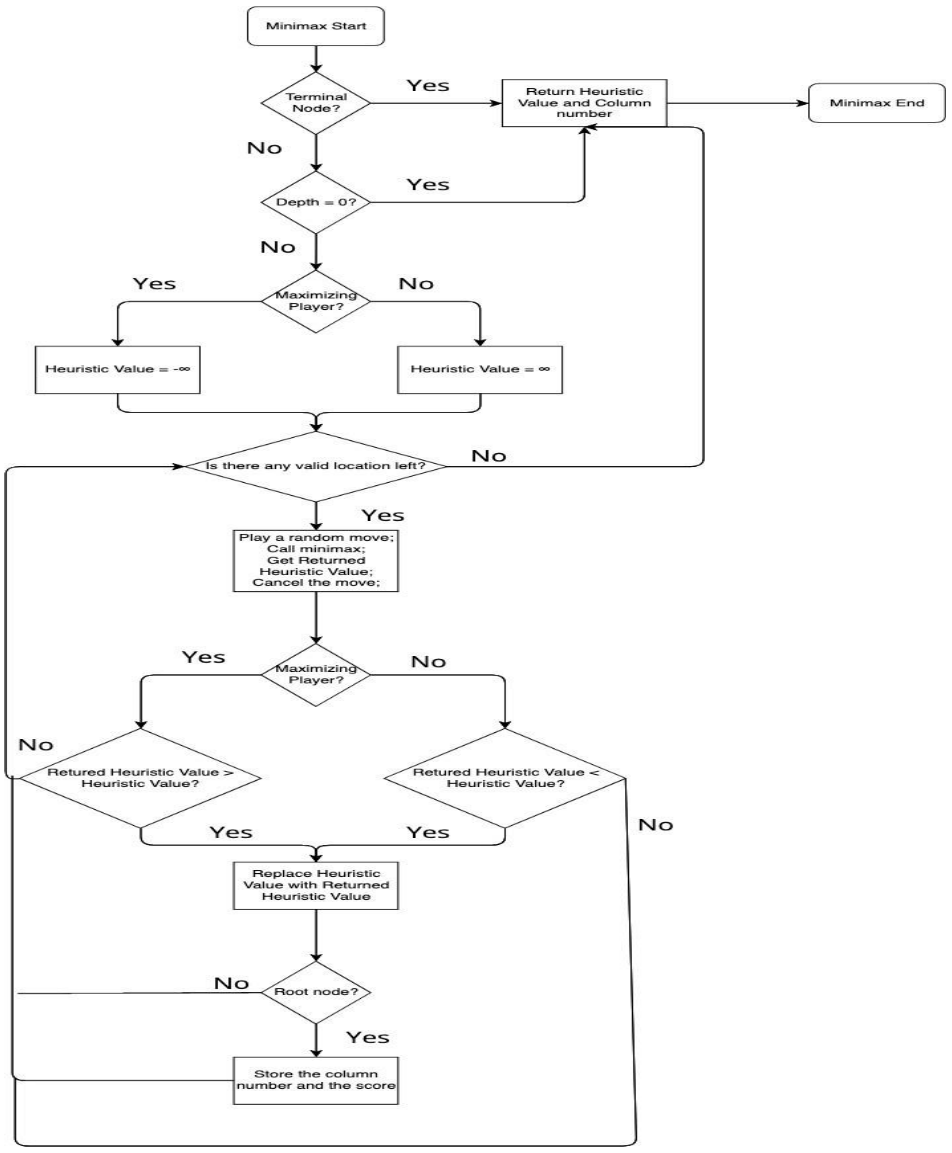
Go Back description:

Identifier & Name	*User go back*
Initiator	*User*
Goal	*User goes back to the previous screen*
Precondition	*User chose to play game*
Postcondition	*User is back at the previous screen*
Main Success Scenario	*1- User presses back* *2- User is back at the previous screen*
Extensions	*None*

Quit Game description:

Identifier & Name	*UC5 quit game*
Initiator	*User*
Goal	*Game closes successfully*
Precondition	*None*
Postcondition	*Program is terminated successfully*
Main Success Scenario	*1- User presses quit* *2- Game window is closed and terminated*
Extensions	*None*

Implementing Minimax Algorithm

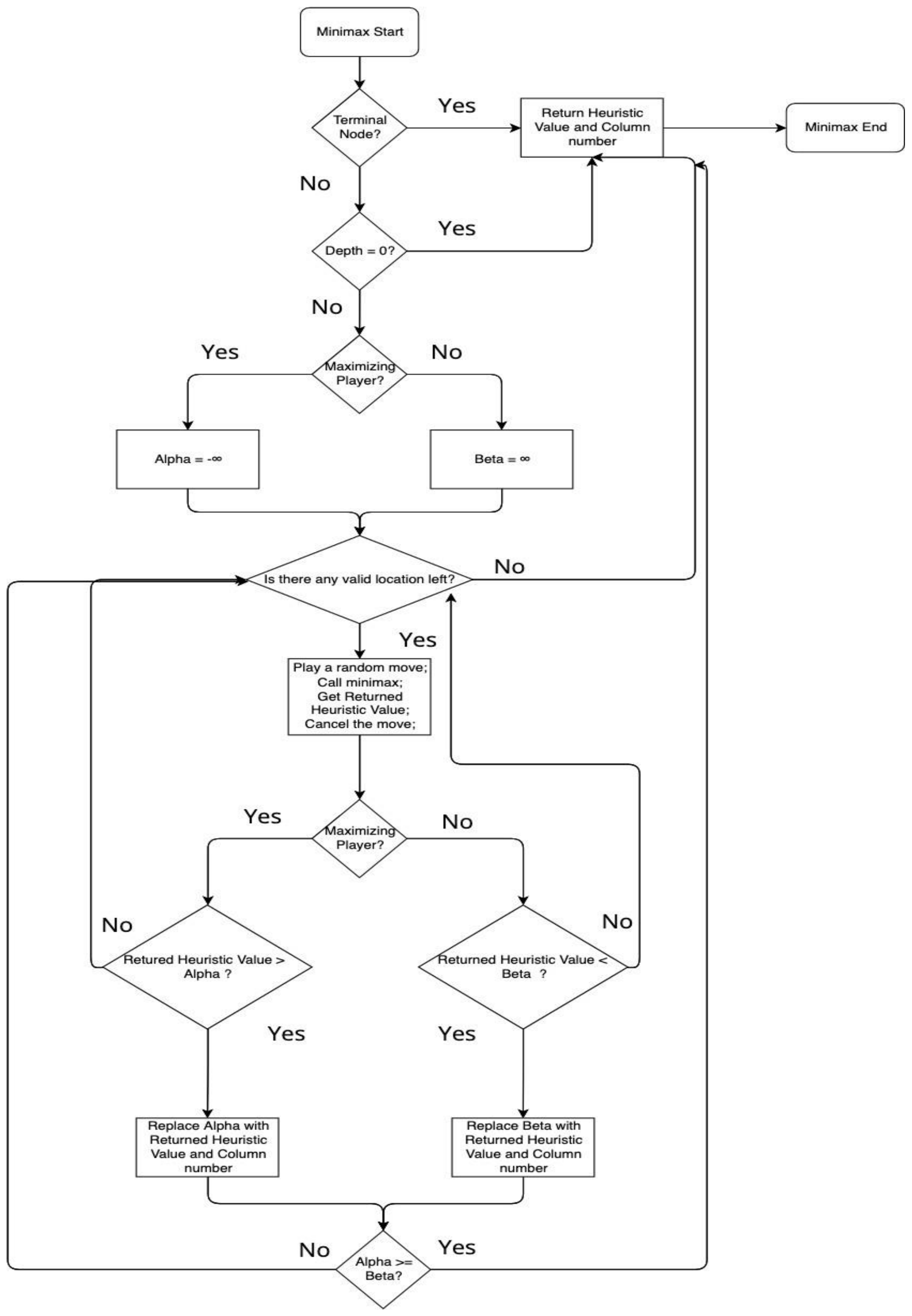


Minimax Start
Terminal Node?
Yes
No
Return Heuristic Value and Column number
Minimax End
Depth = 0?
Yes
No
Maximizing Player?
Yes
No
Heuristic Value = -∞
Heuristic Value = ∞
Is there any valid location left?
No
Yes
Play a random move; Call minimax; Get Returned Heuristic Value; Cancel the move;
Maximizing Player?
Yes
No
Retured Heuristic Value > Heuristic Value?
Retured Heuristic Value < Heuristic Value?
No
No
Yes
Yes
Replace Heuristic Value with Returned Heuristic Value
Root node?
No
Yes
Store the column number and the score

Psuedocode:

```
function minimax(node, depth, maximizingPlayer)
 if depth = 0 or node is a terminal node
   return the heuristic value of node
 if maximizingPlayer  bestValue := −∞
   for each child of node
   v := minimax(child, depth − 1, FALSE)  bestValue := max(bestValue, v)
  return bestValue
 else      (* minimizing player *)
  bestValue := +∞
   for each child of node
   v := minimax(child, depth − 1, TRUE)  bestValue := min(bestValue, v)
 return bestValue
```

Implementing Alpha-Beta Algorithm:



Minimax Start
Terminal Node?
Yes
Return Heuristic Value and Column number
Minimax End
No
Depth = 0?
Yes
No
Maximizing Player?
Yes
No
Alpha = -∞
Beta = ∞
Is there any valid location left?
No
Yes
Play a random move; Call minimax; Get Returned Heuristic Value; Cancel the move;
Maximizing Player?
Yes
No
No
Retured Heuristic Value > Alpha ?
Returned Heuristic Value < Beta ?
No
Yes
Yes
Replace Alpha with Returned Heuristic Value and Column number
Replace Beta with Returned Heuristic Value and Column number
No
Alpha >= Beta?
Yes

Psuedocode:

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
        If depth == 0 or game over in position  return static evaluation of position
        if maximizingPlayer
                maxEval = -infinity
         for each child of position
           eval = minimax(child, depth - 1, alpha, beta false)
           maxEval = max(maxEval, eval)
           alpha = max(alpha, eval)
          if beta <= alpha
```

____

Heuristic Functions:

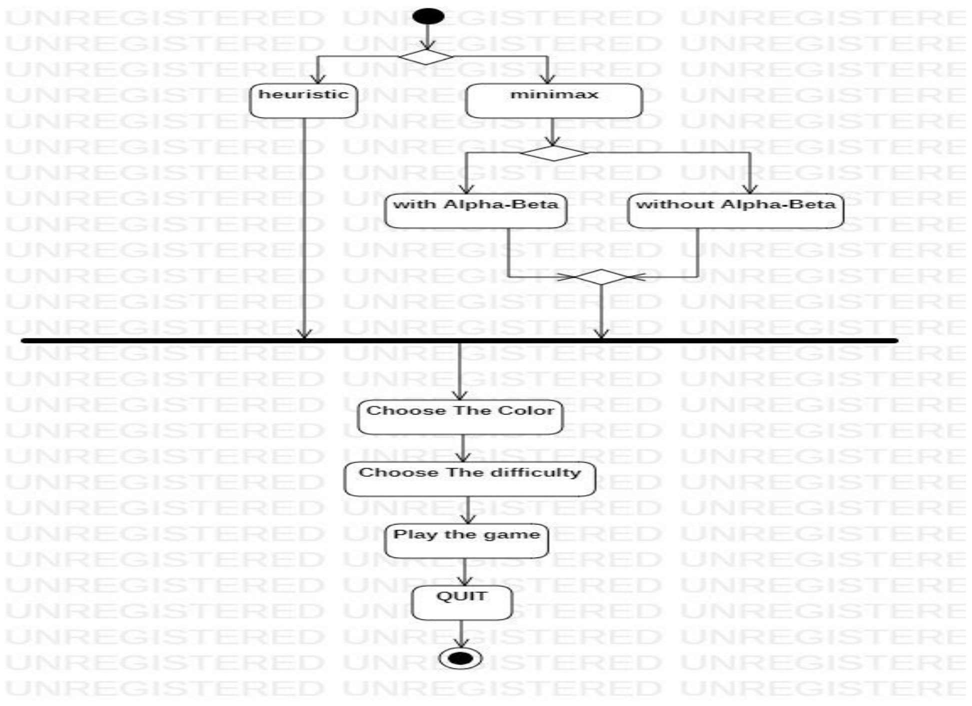
score_position that evaluates the score of a given game board position for a specific piece in a Connect Four-like game. It looks like the function utilizes a helper function evaluate_window to assess the score of a particular window of pieces in the board. Here's a breakdown of what the function does: Center Score: It checks the central column of the board and awards points for the number of occurrences of the specified piece in that column. Horizontal Score: It iterates through each row and considers consecutive windows of pieces of length WINDOW_LENGTH in the horizontal direction. It uses the evaluate_window function to assess the score of each window and adds it to the total score. Vertical Score: Similar to the horizontal score, it iterates through each column and considers consecutive windows of pieces of length WINDOW_LENGTH in the vertical direction. It again uses the evaluate_window function to assess the score of each window and adds it to the total score. Diagonal Score: It considers both diagonals (from top-left to bottom-right and from top-right to bottom-left) and looks for consecutive windows of pieces of length WINDOW_LENGTH. It uses the evaluate_window function to assess the score of each diagonal window and adds it to the total score. The overall score is the sum of scores obtained from the center, horizontal, vertical, and diagonal evaluations.

Pseudocode of Heuristic Functions:

```
Function ew(window ,piece):
  score, opp_piece = 0, human_piece
  if piece == HUMAN_PIECE
   else AI_PIECE

   if number  of windows (piece)== 4:
  score += 100
   elif number  of windows (piece) == 3 and number  of windows (piece,     empty) == 1:
 score += 10
   elif number  of windows (piece )== 2 and number  of windows (piece, empty) == 2:
  score += 4

   if number  of windows (piece )== 4:
    score -= 100
   elif number  of windows (piece )== 3 and number  of windows (piece ) == 1:
   score -= 20

 return score

Function sp(board, piece):
   score = 0
   center_array = [int(i) for i in list(board[cols//2])]
   score += center_array.count(piece) * 3

   for row in range(rows):
       row_array = [int(i) for i in list(board[row])]
       for col in range(cols-3):
           window = row_array[col:col+window_length]
           score += ew(window, piece)

    for col in range(cols):
       col_array = [int(i) for i in list(board[:,col])]
       for row in range(rows-3):
           window = col_array[row:row+ window_length]
           score += ew(window, piece)

    for row in range(rows-3):
       for col in range(cols-3):
           window = [board[row+i][col+i] for i in range(window_length)]
           score += ew(window, piece)

   for row in range(rows-3):
       for col in range(cols-3):
           window = [board[row+3-i][col+i] for i in range(window_length)]
           score += ew(window, piece)

   return score
```

```
Function pbm(board, piece):
    valid_locations = gvl(board)
    best_score, best_col = -10000, random.choice(valid_locations)

    for col in valid_locations:
        row = gnor(board, col)
        temp_board = cb(board)
        dp(temp_board, row, col, piece)
        score = sp(temp_board, piece)

          if score > best_score:
            best_score, best_col = score, col

    return best_col


Fspws(board, piece):
    symmetrical_board = np.rot90(board, k=2)
    score_original, score_symmetrical = sp(board, piece), sp(symmetrical_board, piece)
    return max(score_original, score_symmetrical)
```

Activity Diagram



heuristic
minimax
with Alpha-Beta
without Alpha-Beta
Choose The Color
Choose The difficulty
Play the game
QUIT