Introduction

Breakthrough Game

The goal of this assignment is to implement an agent to play a simple 2-player zero-sum game called **Breakthrough**

Implement agents to play the **Breakthrough** game, one using ***minimax search*** and one using ***alpha-beta search*** as well as two evaluation functions - one which is more Offensive. while the other which is more defensive. The evaluation functions are used to return a value for a position when the depth limit of the search is reached.

Minimax search is a decision-making algorithm that involves analysing the entire game tree to determine the best move to make, assuming that both players will play optimally. It is called minimax because it alternates between minimizing the score of the opponent and maximizing its own score. In a search tree of depth 3, the algorithm would evaluate all possible moves three levels deep and select the move with the highest score.

Alpha-beta search is an optimization of the minimax algorithm that reduces the number of nodes evaluated by pruning branches that are guaranteed to be unimportant. This algorithm maintains two values, alpha and beta, representing the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively. The algorithm prunes branches that are guaranteed to not lead to a better score than the current best.

Working of Search Algorithms

MiniMax Algorithm

The minimax algorithm is a decision-making algorithm used in game theory to find the best move to make in a game with two players, assuming that both players play optimally. The algorithm works by analysing the entire game tree and evaluating the scores of all possible moves.

The algorithm starts at the top of the game tree, representing the current state of the game, and recursively evaluates each node in the tree. Each node represents a possible move that can be made by the current player.

The algorithm alternates between two players: the maximizing player and the minimizing player. The maximizing player tries to maximize its score, while the minimizing player tries to minimize the score of the maximizing player.

At each node, the algorithm evaluates all possible moves that can be made by the current player and determines the score of each possible move. The score of a move is determined by evaluating the resulting game state after making the move. The score of a game state is a numerical value that represents how good the game state is for the maximizing player. For example, in a game of Tic Tac Toe, the score might be +1 for a winning game state, -1 for a losing game state, and 0 for a draw game state.

Once all possible moves have been evaluated at a node, the algorithm selects the move with the highest score if it is the turn of the maximizing player, or the move with the lowest score if it is the turn of the minimizing player. This process continues until the algorithm reaches a leaf node, representing the end of the game or a predetermined search depth.

At the leaf node, the algorithm returns the score of the game state. This score is propagated back up the tree to the parent node, where the algorithm selects the move with the highest score if it is the turn of the maximizing player, or the move with the lowest score if it is the turn of the minimizing player.

The minimax algorithm assumes that both players play optimally, so it evaluates all possible moves and selects the best move for the current player. However, in practice, it is often not feasible to evaluate all possible moves, so more efficient algorithms, such as alpha-beta pruning, are used to optimize the search process.

Alpha-Beta Algorithm

The alpha-beta algorithm is a variation of the minimax algorithm, used to determine the best move for a player in a game with two players. It works by exploring the game tree and evaluating the scores of all possible moves, but it reduces the number of nodes evaluated by pruning branches of the game tree that are unlikely to lead to a better decision.

The algorithm uses two values, alpha and beta, to keep track of the maximum score achievable by the maximizing player and the minimum score achievable by the minimizing player, respectively. It starts at the top of the game tree, representing the current state of the game, and evaluates each node in the tree recursively.

At each node, the algorithm evaluates all possible moves that can be made by the current player and determines the score of each possible move. The algorithm alternates between two players, maximizing and minimizing their scores, and updates the current alpha and beta values accordingly.

If the current node is a maximizing node, the algorithm checks if the score of the current node is greater than or equal to the current beta value. If it is, the algorithm prunes the branch and returns the current beta value, because the minimizing player will not choose this branch, since it leads to a worse outcome than a previously evaluated branch. Otherwise, the algorithm updates the alpha value to be the maximum of the current score and the current alpha value.

If the current node is a minimizing node, the algorithm checks if the score of the current node is less than or equal to the current alpha value. If it is, the algorithm prunes the branch and returns the current alpha value, because the maximizing player will not choose this branch, since it leads to a worse outcome than a previously evaluated branch. Otherwise, the algorithm updates the beta value to be the minimum of the current score and the current beta value.

Once all possible moves have been evaluated at a node, the algorithm selects the move with the highest score if it is the turn of the maximizing player, or the move with the lowest score if it is the turn of the minimizing player. This process continues until the algorithm reaches a leaf node, representing the end of the game or a predetermined search depth.

At the leaf node, the algorithm returns the score of the game state. This score is propagated back up the tree to the parent node, where the algorithm updates the alpha and beta values and selects the move with the highest score if it is the turn of the maximizing player, or the move with the lowest score if it is the turn of the minimizing player.

Overall, the alpha-beta algorithm is an efficient way to search the game tree and find the best move in a game with two players, by pruning branches of the game tree that are unlikely to lead to a better decision.

How code works

Initial Step

SELECT MATCHUP

(1) Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)

(2) Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

(3) Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

(4) Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)

(5) Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)

(6) Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)

(7) Random AI vs Random AI

or type -1 to exit

This is a menu that displays the available matchup options for the user to select from. The user can enter a number from 1 to 7 to select a matchup, and the program will execute the corresponding function for the selected matchup.

The matchup options involve different combinations of algorithms (Minimax, Alpha-beta) and heuristics (Offensive Heuristic 1 and 2, Defensive Heuristic 1 and 2) playing against each other, as well as a matchup between two random AIs. The purpose of these matchups is to demonstrate the different strategies and algorithms used in game AI.

The user can exit the program by typing "-1" instead of selecting a matchup.

main ()

This code block is the main game loop where the game runs until the user quits or a winner is declared. The loop constantly checks for any events in the game (such as the user quitting), clears the screen, and updates the display.

If there is no winner yet, it calls the appropriate "matchup" function based on the user's choice. Each matchup function is a specific algorithm for the game's AI to make its moves. If a winner is detected, the code updates the screen to display the winner message using a specific font and music.

min_Max(Depth,index,maxturn,moves,TDepth)

The function takes the following parameters:

- Depth: The current depth of the search tree.
- index: The current index in the moves list representing the current state of the game.
- maxturn: A Boolean indicating whether it's the maximizing player's turn or the minimizing player's turn.
- moves: A list of all possible moves in the game.
- TDepth: The maximum depth to search in the game tree.

The function first checks if the current depth equals the maximum depth to search, in which case it returns the move at the current index.

If it's the maximizing player's turn, the function returns the maximum value of calling the min_Max function recursively on the left and right children of the current node (represented by index * 2 and index * 2 + 1, respectively). The Boolean maxturn is set to False when calling the function on the children to indicate that it's the minimizing player's turn next.

If it's the minimizing player's turn, the function returns the minimum value of calling the min_Max function recursively on the left and right children of the current node (represented by index * 2 and index * 2 + 1, respectively). The Boolean maxturn is set to True when calling the function on the children to indicate that it's the maximizing player's turn next.

Finally, The function recursively traverses the game tree, alternating between maximizing and minimizing player's turns, until it reaches the maximum depth to search, at which point it returns the best move found.

min_Max_Alpha_Beta(Depth,index,maxturn,moves,TDepth,alpha,beta)

The function takes the same parameters as the basic minimax algorithm, as well as two additional parameters:

- alpha: The value of the best (maximum) alternative found so far at any choice point along the path for the maximizer.
- beta: The value of the best (minimum) alternative found so far at any choice point along the path for the minimizer.

At each node of the game tree, the function checks whether the current node is a leaf node, in which case it returns the move at the current index. Otherwise, it checks whether it's the maximizing player's turn or the minimizing player's turn.

If it's the maximizing player's turn, the function first recursively calls itself on the right child of the current node (index * 2 + 1), passing in the current alpha and beta values. The best value returned from the recursive call is stored in the variable 'best'. The function then updates the value of alpha to be the maximum of the current alpha and the best value found, representing the best alternative found so far along the path of the maximizer. Finally, the function returns the value of alpha.

If it's the minimizing player's turn, the function does the same as above, but updates the value of beta instead of alpha, representing the best alternative found so far along the path of the minimizer.

At each level of the tree, if the value of alpha becomes greater than or equal to beta, the remaining nodes in the subtree can be pruned, as the minimizer will not choose this branch (since there exists another branch that results in a smaller value than the current beta). This is the main advantage of the alpha-beta pruning technique - it can avoid evaluating nodes that cannot affect the final decision.

Finally, the function recursively traverses the game tree, keeping track of the current best alternative values for both players, and pruning branches where possible. At the end, it returns the best move found.

min_Max_Alpha_Beta_mode(Depth,index,maxturn,moves,TDepth,alpha,beta,heuristic_mode,mat,player)

The function takes the same parameters as the previous implementation, with two additional parameters:

- heuristic_mode: A string representing the type of heuristic function to be used for evaluation.
- mat: The current state of the game board.

At each node of the game tree, the function first checks whether the current node is a leaf node. If it is a leaf node, it returns the move at the current index.

If it's the maximizing player's turn, the function first calculates the heuristic value of the current game state using the Heuristic(player) function. Then, it recursively calls itself on the right child of the current node (index * 2 + 1), passing in the current alpha and beta values, the heuristic mode, the game board state, and the player making the move. The best value returned from the recursive call is stored in the variable 'best'.

After that, the function calculates the heuristic value of the game state again, using the specified heuristic function corresponding to the 'heuristic_mode' parameter. If the new heuristic value is better than the previous one, the function returns the current value of alpha (the best alternative found so far along the path of the maximizer). Otherwise, the function returns the best value found in the recursive call.

If it's the minimizing player's turn, the function does the same as above, but updates the value of beta instead of alpha. The function first calculates the heuristic value of the current game state using the Heuristic(player) function. Then, it recursively calls itself on the right child of the current node (index * 2 + 1), passing in the current alpha and beta values, the heuristic mode, the game board state, and the player making the move. The best value returned from the recursive call is stored in the variable 'best'.

After that, the function calculates the heuristic value of the game state again, using the specified heuristic function corresponding to the 'heuristic_mode' parameter. If the new heuristic value is worse than the previous one, the function returns the current value of beta (the best alternative found so far along the path of the minimizer). Otherwise, the function returns the best value found in the recursive call.

Finally, the function recursively traverses the game tree, keeping track of the current best alternative values for both players, and pruning branches where possible based on the heuristics. At the end, it returns the best move found.

defensive_Heuristic_1(player)

The **defensive_Heuristic_1** function calculates a score based on the number of pieces that the player has on the board. The more pieces the player has, the higher the score. The function returns twice the number of pieces the player has, plus a random value. This heuristic can be used when the player wants to focus on protecting their own pieces and minimizing losses.

offensive_Heuristic_1(player)

The **offensive_Heuristic_1** function calculates a score based on the number of pieces the opponent has on the board. The fewer pieces the opponent has, the higher the score. The function returns twice the number of opponent pieces that are remaining on the board subtracted from 60 (total number of pieces on the board), plus a random value. This heuristic can be used when the player wants to focus on capturing as many opponent pieces as possible and reducing their opponent's ability to move.

defensive_Heuristic_2(player)

The defensive_Heuristic_2 function calculates a score based on the number of pieces that the player has on the board, similar to defensive_Heuristic_1. However, in this function, the score is simply the number of pieces the player has remaining on the board plus a random value. This heuristic can be used when the player wants to focus on protecting their own pieces and minimizing losses, but without giving as much emphasis to having a large number of pieces on the board.

offensive_Heuristic_2(player)

The offensive_Heuristic_2 function calculates a score based on the number of pieces the opponent has on the board, similar to offensive_Heuristic_1. However, in this function, the score is simply the number of opponent pieces remaining on the board plus a random value. This heuristic can be used when the player wants to focus on capturing as many opponent pieces as possible and reducing their opponent's ability to move, but without giving as much emphasis to having a small number of opponent pieces on the board.

update_Valid_Pos_B()

updates the list of valid positions for black pieces by checking the three possible moves (diagonal left, forward, and diagonal right) for each black piece, and adding the position to the B_valid_temp list if it is a valid move.

update_Valid_Pos_Y

updates the list of valid positions for yellow pieces by checking the three possible moves (diagonal left, forward, and diagonal right) for each black piece, and adding the position to the Y_valid_temp list if it is a valid move.

is_validPos(Pos)

checks whether a given position Pos is valid or not, by verifying that both its row and column indexes are within the range [0, 7].

valid_B_Pos(Pos)

checks whether a given position Pos is a valid position for a black piece, by verifying that it is not already occupied by a black piece.

valid_Y_Pos(Pos)

checks whether a given position Pos is a valid position for a yellow piece, by verifying that it is not already occupied by a yellow piece.

update_black(PreviousPos,NewPos)

This function updates the position of a black piece on the game board. It takes in the previous position of the piece (PreviousPos) and the new position of the piece (NewPos).

First, it updates the mat matrix by setting the element at PreviousPos to be an empty space (' '), removing PreviousPos from B_Pos, and adding NewPos to B_Pos and setting the element at NewPos to be 'B'.

Next, it updates the B_valid list by removing PreviousPos from it and adding NewPos to it.

Finally, it updates the valid positions for all black pieces by calling the update_Valid_Pos_B function, which loops through all black pieces and updates their valid positions based on the new state of the game board.

update_yellow(PreviousPos,NewPos)

This function updates the position of a yellow piece on the game board. It takes in the previous position of the piece (PreviousPos) and the new position of the piece (NewPos).

First, it updates the mat matrix by setting the element at PreviousPos to be an empty space (' '), removing PreviousPos from Y_Pos, and adding NewPos to Y_Pos and setting the element at NewPos to be 'Y'.

Next, it updates the Y_valid list by removing PreviousPos from it and adding NewPos to it.

Finally, it updates the valid positions for all yellow pieces by calling the update_Valid_Pos_Y function, which loops through all yellow pieces and updates their valid positions based on the new state of the game board.

Black_Move()

This function defines the behavior of the Black player in the game. It starts by checking if there are any valid moves for the Black player to make. If there are, it proceeds to select one of the Black pieces to move using the pick_one_black() function. Then, it selects a move using the select_move_black() function and calculates the new position of the selected piece.

If the new position is invalid or if there is already a Yellow piece in the new position, the selection of the move is repeated until a valid move is selected. If the new position is valid and there is no Yellow piece there, the position of the selected Black piece is updated to the new position using the update_black() function.

Yellow_Move()

This function defines the behavior of the Yellow player in the game. It starts by checking if there are any valid moves for the Yellow player to make. If there are, it proceeds to select one of the Yellow pieces to move using the pick_one_yellow() function. Then, it selects a move using the select_move_yellow() function and calculates the new position of the selected piece.

If the new position is invalid or if there is already a Yellow piece in the new position, the selection of the move is repeated until a valid move is selected. If the new position is valid and there is no Yellow piece there, the position of the selected Yellow piece is updated to the new position using the update_yellow() function.

isBlackWinner()

It iterates over all black pieces and checks if any of them has reached the first or second row of the board (row 0 or 1), which would mean that the black player has won the game. If none of the pieces of a player has reached the winning row, then the function returns False, indicating that the player has not won yet.

isYellowWinner()

It iterates over all yellow pieces and checks if any of them has reached the seventh or sixth row of the board (row 7 or 6), which would mean that the yellow player has won the game. If none of the pieces of a player has reached the winning row, then the function returns False, indicating that the player has not won yet.

Random_Ai(turn)

This is a function named Random_Ai that takes a turn parameter (a Boolean) to determine whose turn it is to play. If turn is True, the function calls Black_Move() to make a move for the black player and sets turn to False. If turn is False, the function calls Yellow_Move() to make a move for the yellow player and sets turn to True.

The function returns the updated value of turn.

Black_Move_Minmax_()

This function is an implementation of the minimax algorithm for selecting the best move for the black player. The function first checks if there are any valid moves for the black player, and if so, it calls the `min_Max()` function to determine the best move.

The `min_Max()` function takes as input the current depth of the search, the current score, a Boolean indicating whether the current player is maximizing or minimizing, a list of valid positions for the current player, and the maximum depth to search.

The function then recursively generates all possible moves from the current positions, evaluates the resulting position using a simple evaluation function, and selects the best move based on whether the current player is maximizing or minimizing.

Once the `min_Max()` function has returned the best move, the `Black_Move_Minmax_()` function selects the move and updates the board if the move is valid. If the selected move is not valid, the function calls itself recursively until a valid move is found.

Black_Move_Minmax(mode)

This function Black_Move_Minmax() implements the minimax algorithm with alpha-beta pruning for the black player's move. The function takes one argument mode, which determines the heuristic function to be used during the search. The higher the mode, the more advanced the heuristic function.

First, it checks if there are valid moves for the black player. If there are valid moves, it finds the best move using the min_Max_Alpha_Beta_mode() function, which implements the minimax algorithm with alpha-beta pruning and returns the best move for the black player.

Next, it chooses a random move and checks if the new position is a valid one. If the new position is not valid, it continues to choose another move until it finds a valid move. If the new position is valid, it updates the board with the new position.

If the new position is not valid or there are no valid moves for the black player, the function recursively calls itself until a valid move is found.

Yellow_Move_Minmax_()

This function is an implementation of the minimax algorithm for selecting the best move for the yellow player. The function first checks if there are any valid moves for the yellow player, and if so, it calls the `min_Max()` function to determine the best move.

The `min_Max()` function takes as input the current depth of the search, the current score, a Boolean indicating whether the current player is maximizing or minimizing, a list of valid positions for the current player, and the maximum depth to search.

The function then recursively generates all possible moves from the current positions, evaluates the resulting position using a simple evaluation function, and selects the best move based on whether the current player is maximizing or minimizing.

Once the `min_Max()` function has returned the best move, the `Yellow_Move_Minmax_()` function selects the move and updates the board if the move is valid. If the selected move is not valid, the function calls itself recursively until a valid move is found.

Yellow_Move_Minmax(mode)

This function Yellow_Move_Minmax() implements the minimax algorithm with alpha-beta pruning for the yellow player's move. The function takes one argument mode, which determines the heuristic function to be used during the search. The higher the mode, the more advanced the heuristic function.

First, it checks if there are valid moves for the yellow player. If there are valid moves, it finds the best move using the min_Max_Alpha_Beta_mode() function, which implements the minimax algorithm with alpha-beta pruning and returns the best move for the yellow player.

Next, it chooses a random move and checks if the new position is a valid one. If the new position is not valid, it continues to choose another move until it finds a valid move. If the new position is valid, it updates the board with the new position.

If the new position is not valid or there are no valid moves for the yellow player, the function recursively calls itself until a valid move is found.

min_Max_Alpha_Beta_Ai(turn)

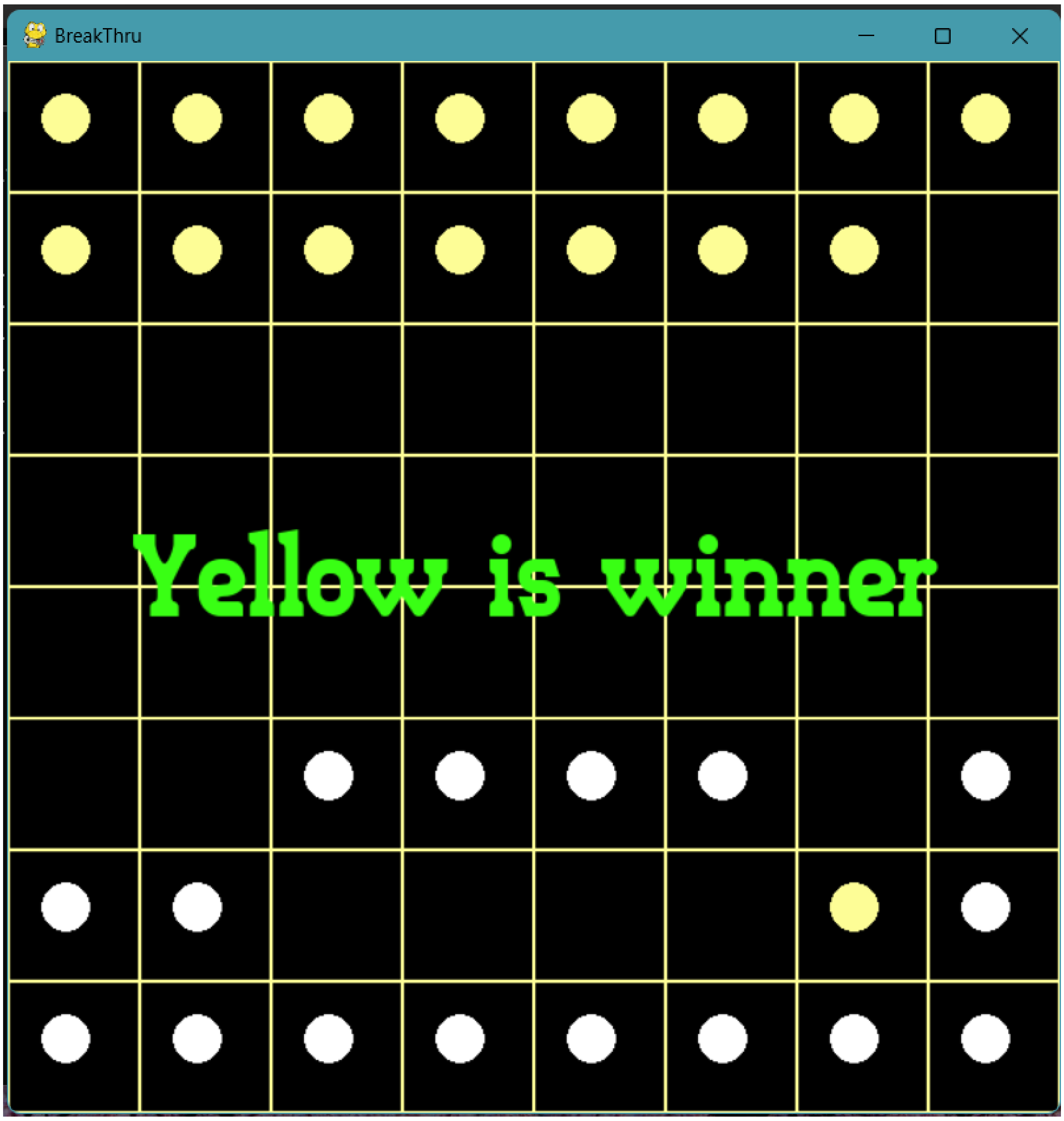
This function seems to be implementing a game-playing AI using the Minimax algorithm with alpha-beta pruning. It takes in a Boolean parameter turn which represents whose turn it is to play.

If turn is True, the function calls the Black_Move_Minmax() function to make a move for the black player using the Minimax algorithm with alpha-beta pruning. Then it sets turn to False, indicating that it's now the yellow player's turn.

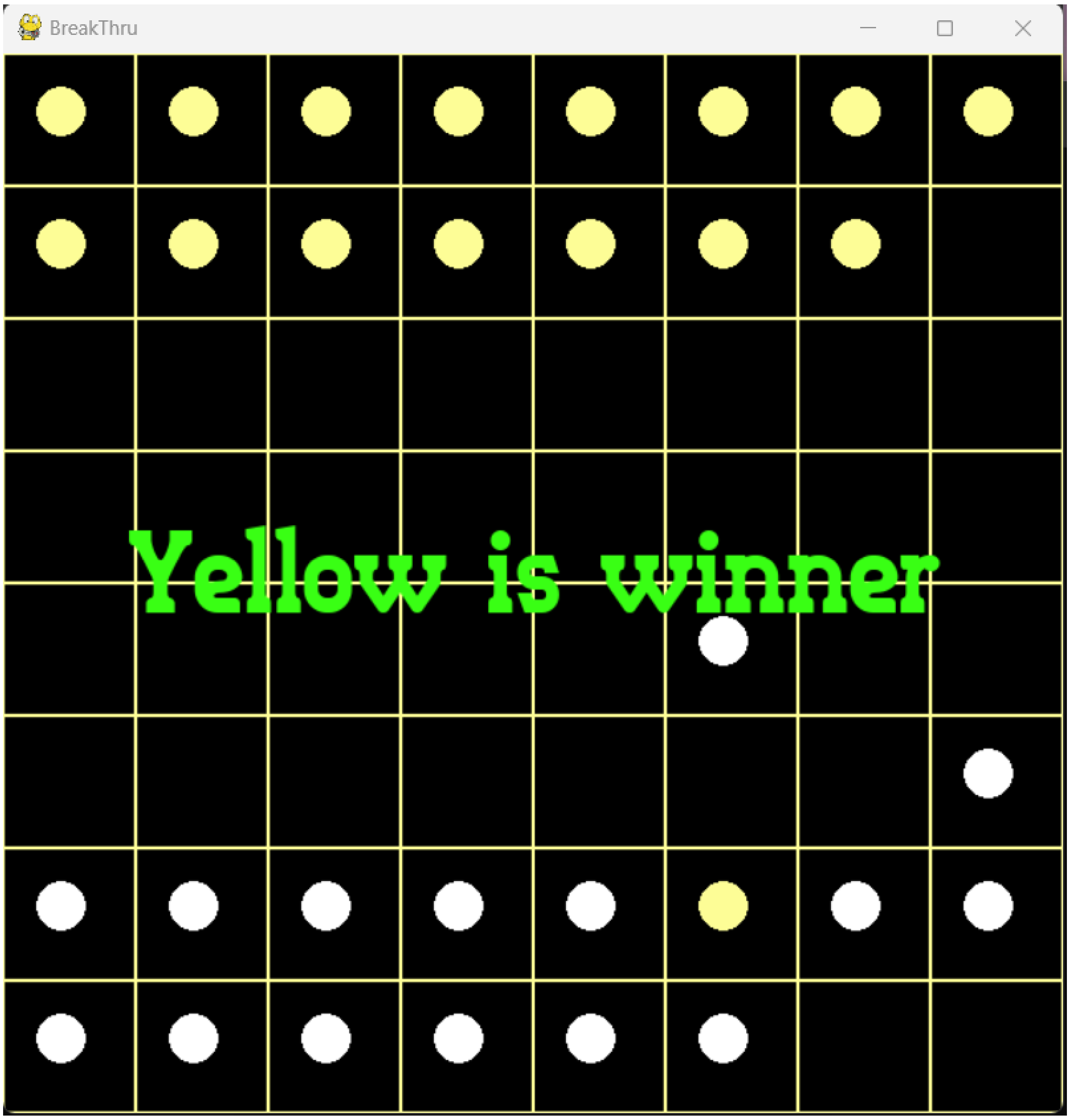
If turn is False, the function calls the Yellow_Move_alphaBeta() function to make a move for the yellow player using the alpha-beta pruning algorithm. Then it sets turn to True, indicating that it's now the black player's turn. Finally, the function returns the updated value of turn.

Results

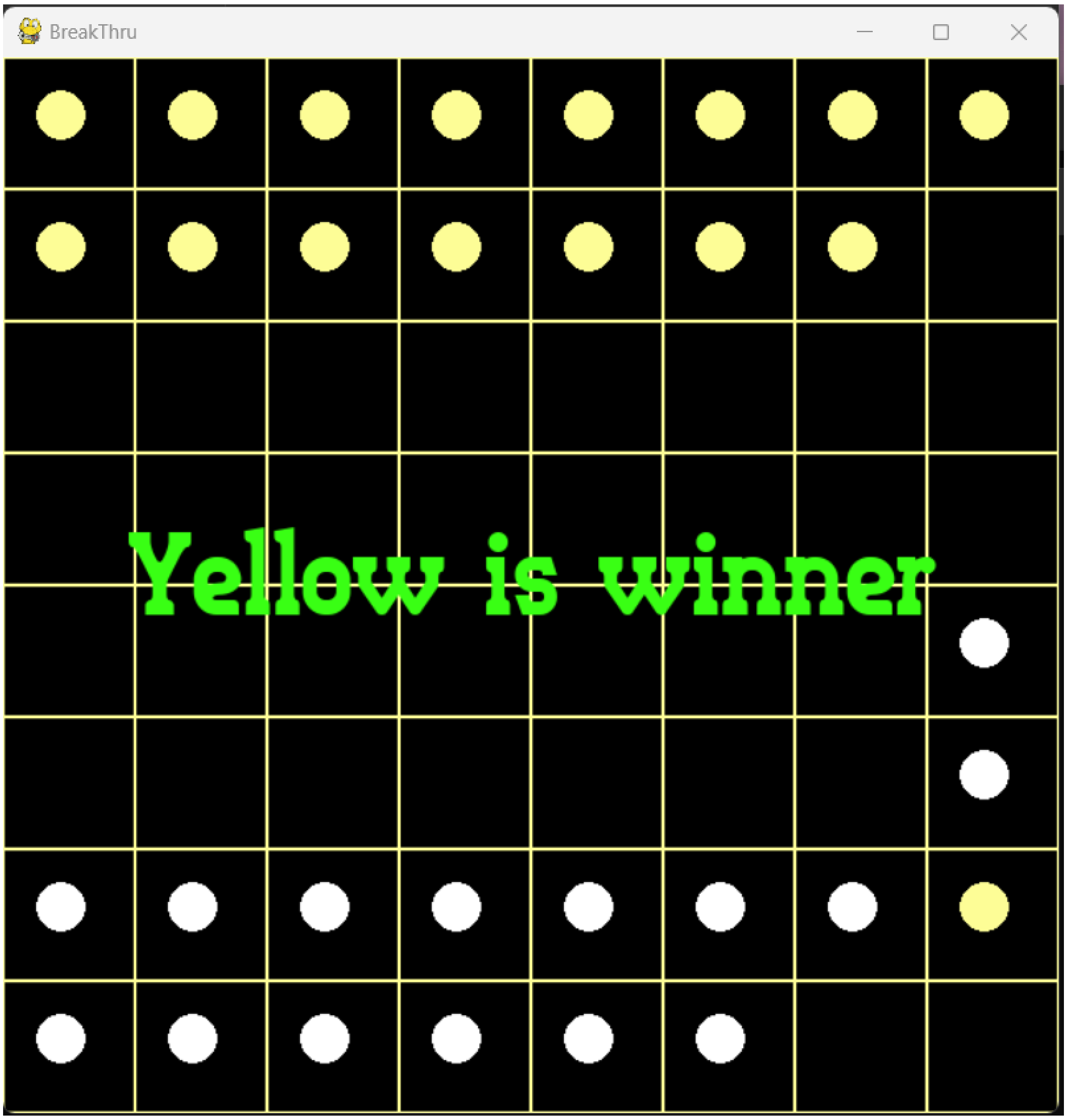
Minimax (Offensive Heuristic 1) vs Alpha-beta (Offensive Heuristic 1)



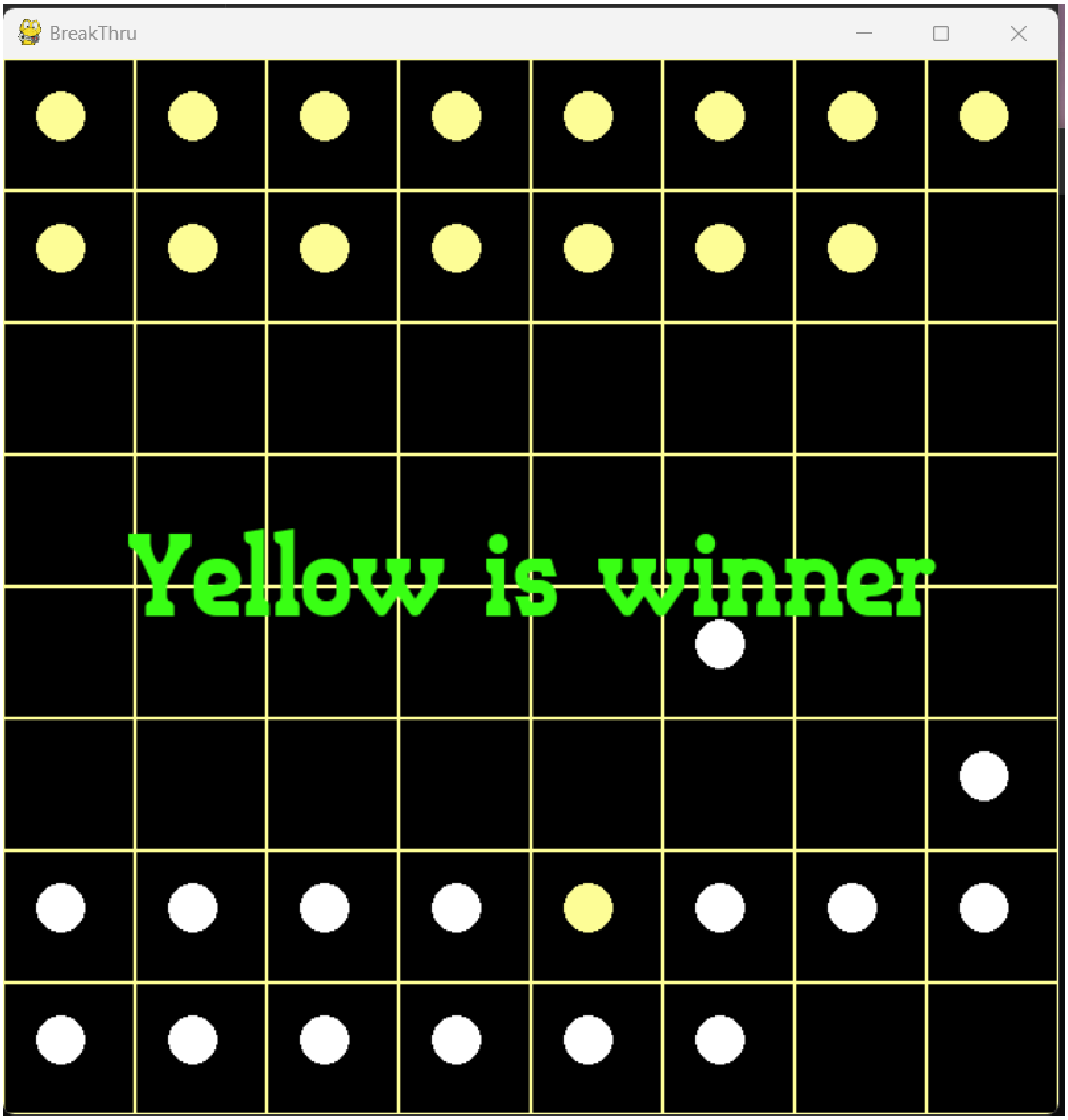
Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)



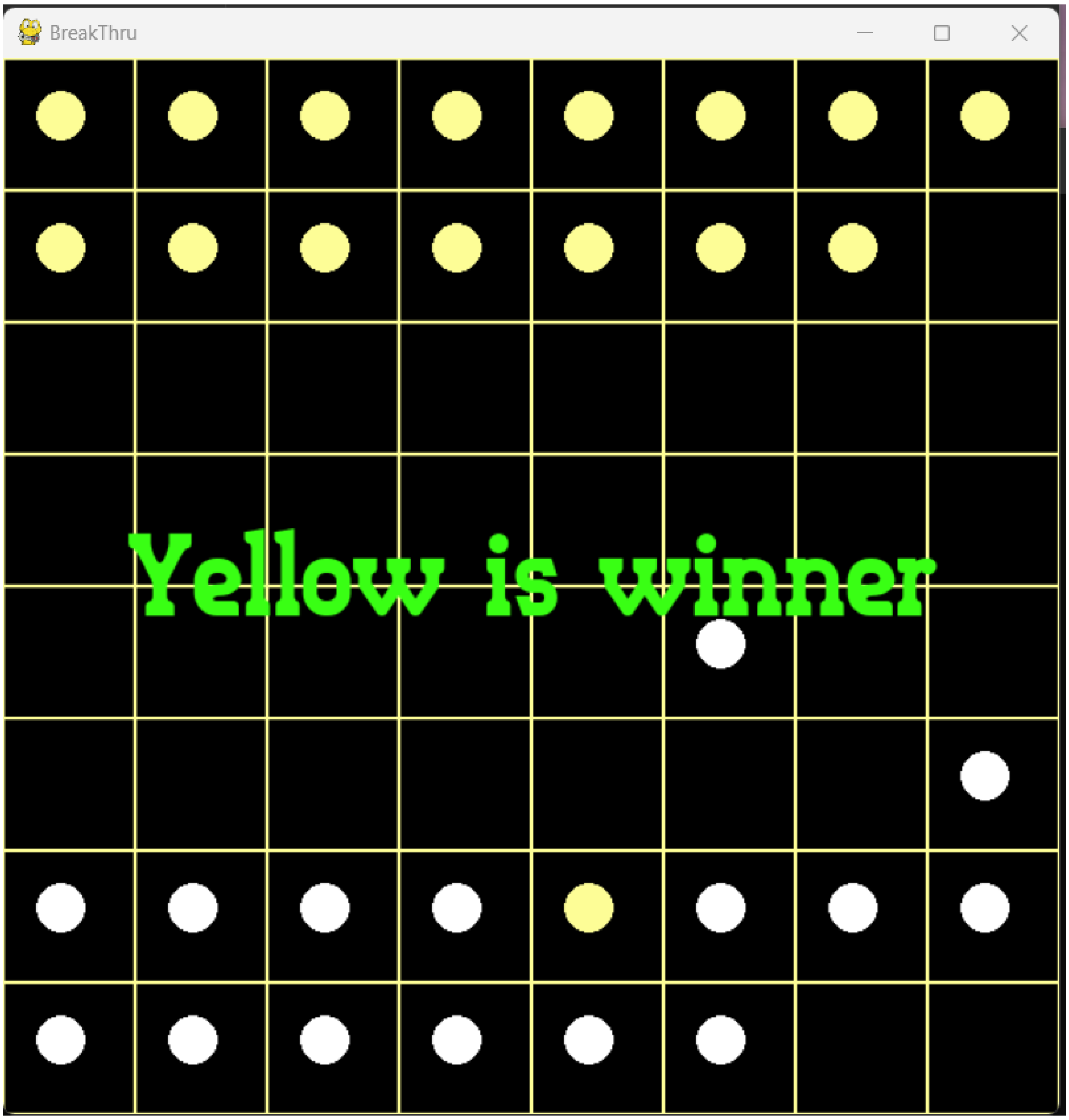
Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)



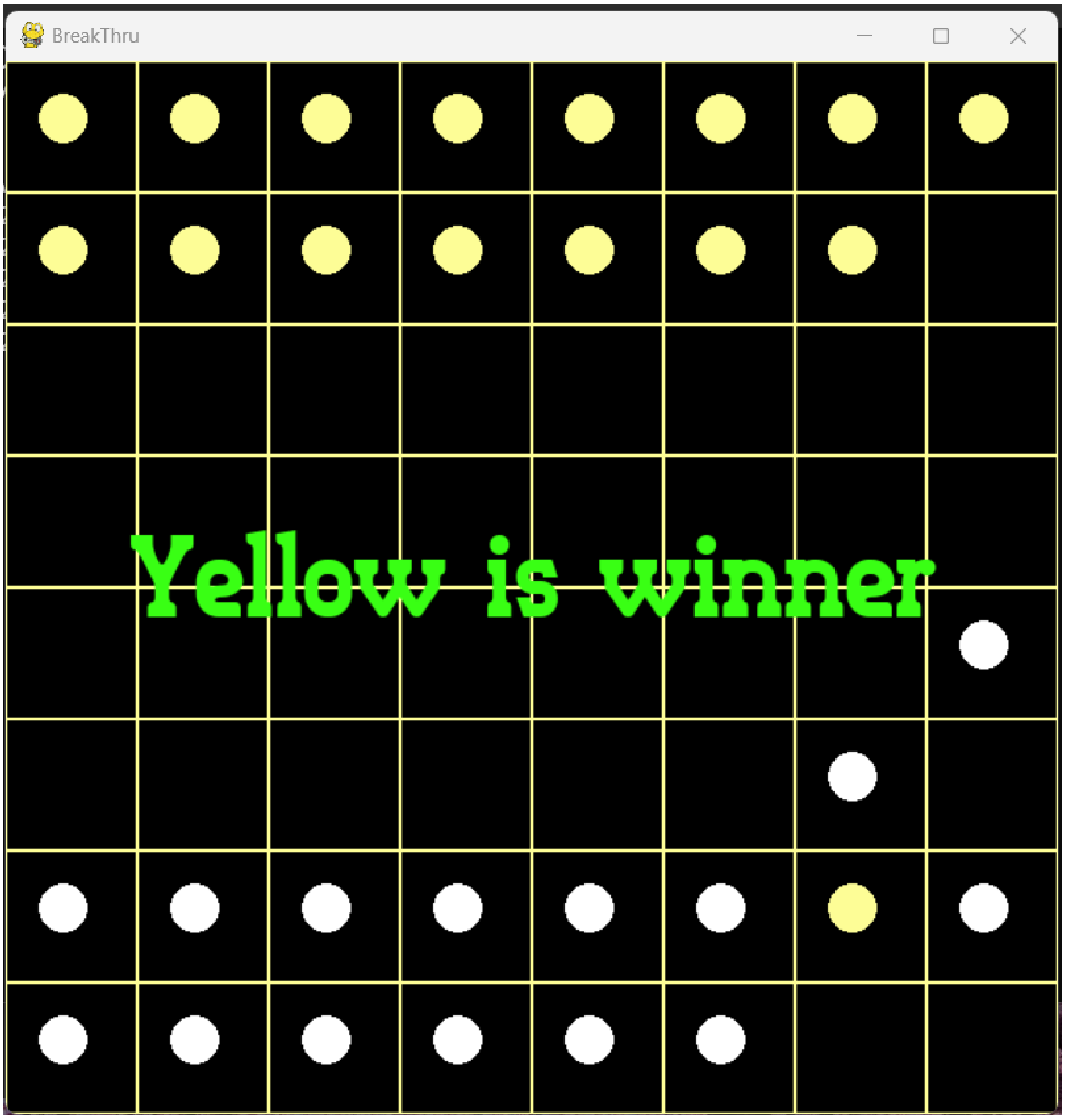
Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Offensive Heuristic 1)



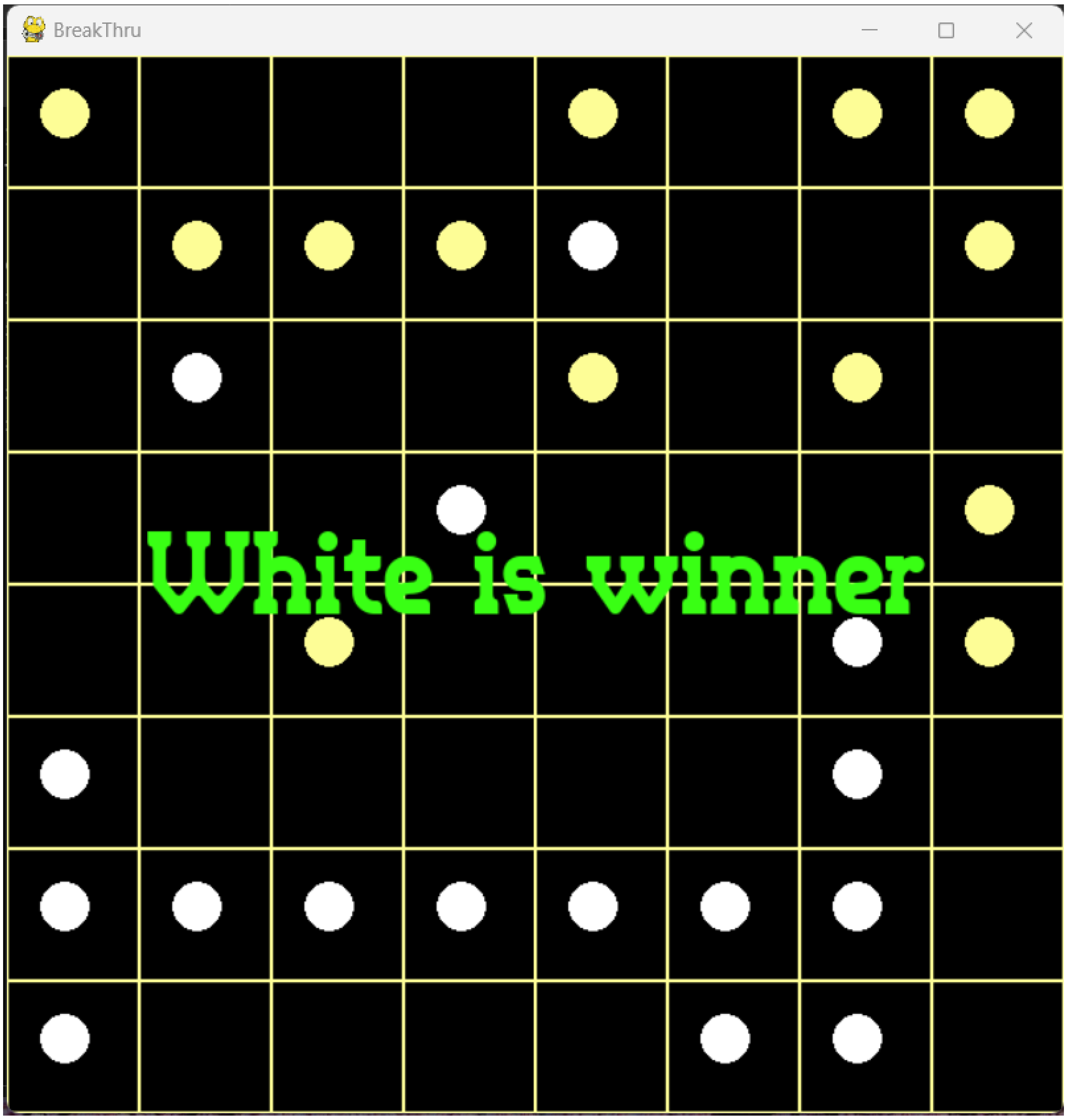
Alpha-beta (Defensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 1)



Alpha-beta (Offensive Heuristic 2) vs Alpha-beta (Defensive Heuristic 2)



Random AI vs Random AI



References

Minimax → https://en.wikipedia.org/wiki/Minimax

Alpha_beta → https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Pygame → https://www.pygame.org/docs/

Heuristic → https://en.wikipedia.org/wiki/Heuristic#Artificial_intelligence

Breakthrough → https://en.wikipedia.org/wiki/Breakthrough_(board_game)