SCHOOL OF SCIENCE & ENGINEERING - SPRING 2025

Mini-Project no.2: Checkers

CSC-3309. Introduction to Artificial Intelligence

Realized by:

Khawla Ould Mansour < 101043 >

Marouane Essaid < 114534 >

Mehdi Mojab < 100620 >

Supervised by:

Dr. KETTANI Driss

Contents

1. Introduction:

This report documents the design, implementation, and performance evaluation of an AI-driven Checkers bot that plays against human competitors. This project leverages 3 search algorithms: **Minimax, Alpha-Beta Pruning, and Alpha-Beta Pruning with Node Ordering.** These algorithms empower the bot to generate optimal moves inside restricted time and search depth limits. The first player to start is the human player taking the white pieces, while the "bot" controls the black pieces. The game follows the basic rules, alongside mandatory captures, king promotions, and the final objective requires total enemy piece elimination or maneuver restriction.

The application contains four essential classes named **GameBoard**, **SearchToolBox**, PlayingTheGame and CheckersGUI. The game relies on four main classes which have essential functions including board management and move validation and search algorithm implementation and user interface delivery. The bot functions under a time limit ranging from 1 to 4 seconds per move together with a search depth of 3 to 5 plies to provide real-time competition and dynamic gameplay.

An overview of project elements follows through sections that detail each class with its associated contributions:

1. <u>GameBoard</u>

Through the **GameBoard** class, the Checkers board state gets handled by an 8x8 grid while pieces get placed accordingly at their initial positions. The class implements move validation together with execution as well as game-over monitoring to maintain Checkers rule compliance. The game operates through this class which ensures board accuracy during every stage of game play.

2. **SearchToolBox**

The **SearchToolBox** class runs the bot's decision algorithms which include Minimax together with Alpha-Beta Pruning and Alpha-Beta with Node Ordering. The tool evaluates different move options while pruning unneeded branches and maintains performance tracking of state expansion alongside pruned branch numbers. The class enables the bot to select optimal choices under time and depth restrictions.

3. **PlayingTheGame**

The PlayingTheGame class controls the game flow by enabling alternations between human player moves and bot moves. The class processes human moves before selecting bot moves using **SearchToolBox** while also reporting the number of expanded states and execution time. This class establishes fluid communication while delivering immediate performance data about the bot.

4. **CheckersGUI**

The **CheckersGUI** class implements a graphical interface by using the turtle library. The application uses the board drawing functionality to display pieces and updates the screen after every move. The GUI provides an optional enhancement of user experience which also earns the 20% bonus.

Taks Distribution:

- **Khawla Ould Mansour (101043)**: Handled the game setup, Minimax, and the analysis.
- **Mehdi Mojab (100620)**: Worked on Alpha-Beta pruning, its analysis, and the bonus task.
- **Marouane Essaid (114534)**: Focused on Alpha-Beta pruning with node ordering and the comparison.

2. Game Setup:

a-Game Rules

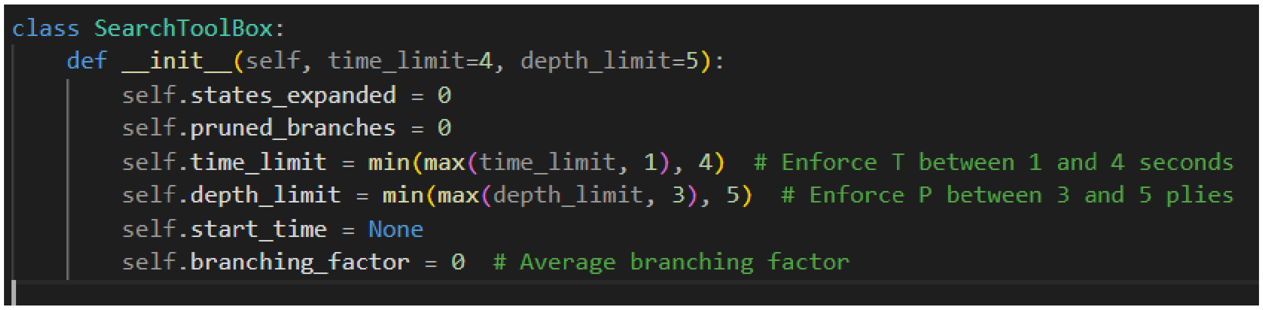
Checkers is a two-player strategy board game played on an 8x8 board. The main rules include:

1. The human player (White) always starts first.
2. Players take turns moving pieces diagonally forward (or backward for kings).
3. Capturing an opponent's piece is mandatory when possible.
4. A piece becomes a king only when it lands on the last row.
5. Kings can move and capture in all four diagonal directions.
6. Multiple jumps in one turn must be executed if available.
7. The game ends when a player has no pieces left or no valid moves.
8. Pieces can only move to empty dark squares.
9. A captured piece is removed immediately after a jump.
10. A player can only move their own pieces during their turn.

Our implementation follows standard Checkers rules and includes a textual interface for interaction.

b- Assignment Specifications: time limitation/number of plies:

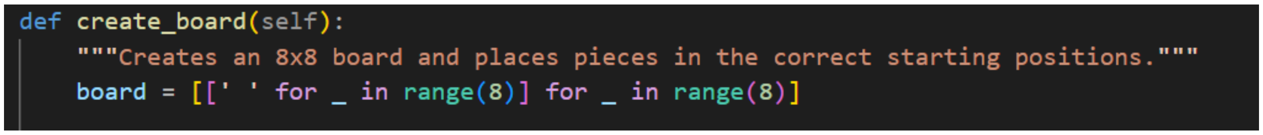
- T: 1 to 4
- P: 3 to 5



```python
class SearchToolBox:
    def __init__(self, time_limit=4, depth_limit=5):
        self.states_expanded = 0
        self.pruned_branches = 0
        self.time_limit = min(max(time_limit, 1), 4)  # Enforce T between 1 and 4 seconds
        self.depth_limit = min(max(depth_limit, 3), 5)  # Enforce P between 3 and 5 plies
        self.start_time = None
        self.branching_factor = 0  # Average branching factor
```

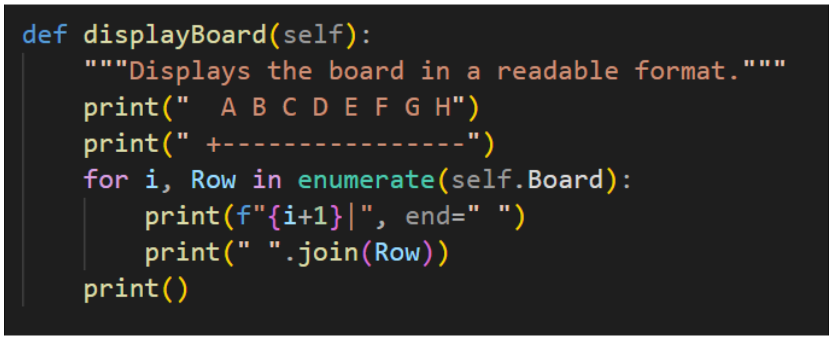
c- Representation of Game State:

- **Board**: A 2D 8x8 grid with alternating dark and light squares.



```python
def create_board(self):
    """Creates an 8x8 board and places pieces in the correct starting positions."""
    board = [[' ' for _ in range(8)] for _ in range(8)]
```

To display it:

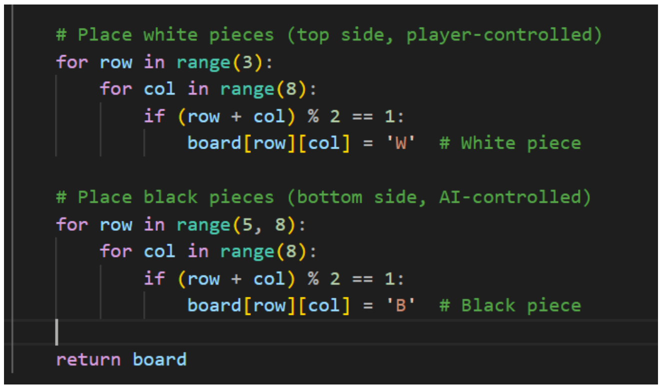


```python
def displayBoard(self):
    """Displays the board in a readable format."""
    print("   A B C D E F G H")
    print(" +----------------")
    for i, Row in enumerate(self.Board):
        print(f"{i+1}|", end=" ")
        print(" ".join(Row))
    print()
```

- **Pieces**:

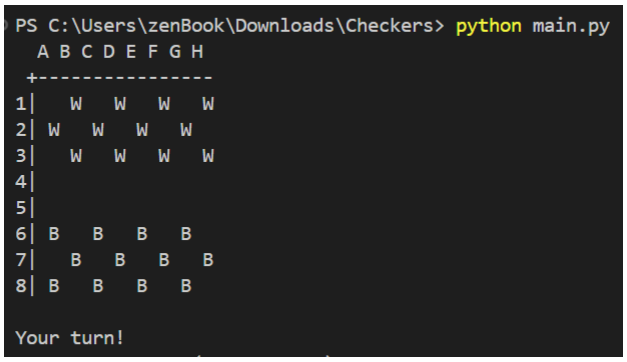
  - White pieces (W) are controlled by the human player and start on the top three rows.

- Black pieces (B) are controlled by the bot and start on the bottom three rows.



```python
    # Place white pieces (top side, player-controlled)
    for row in range(3):
        for col in range(8):
            if (row + col) % 2 == 1:
                board[row][col] = 'W'  # White piece

    # Place black pieces (bottom side, AI-controlled)
    for row in range(5, 8):
        for col in range(8):
            if (row + col) % 2 == 1:
                board[row][col] = 'B'  # Black piece

    return board
```

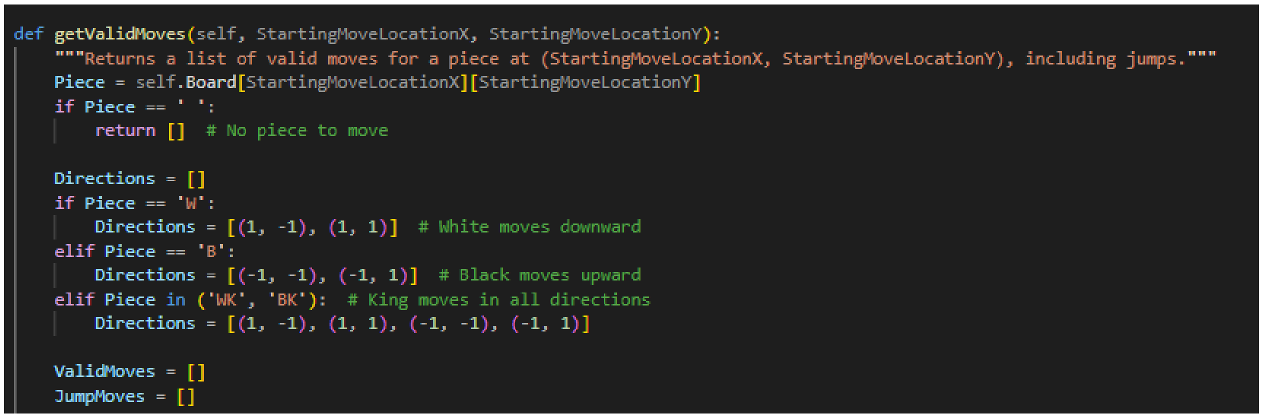
Output example:



```
PS C:\Users\zenBook\Downloads\Checkers> python main.py
   A B C D E F G H
  +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W   W   W
4|
5|
6| B   B   B   B
7|   B   B   B   B
8| B   B   B   B

Your turn!
```

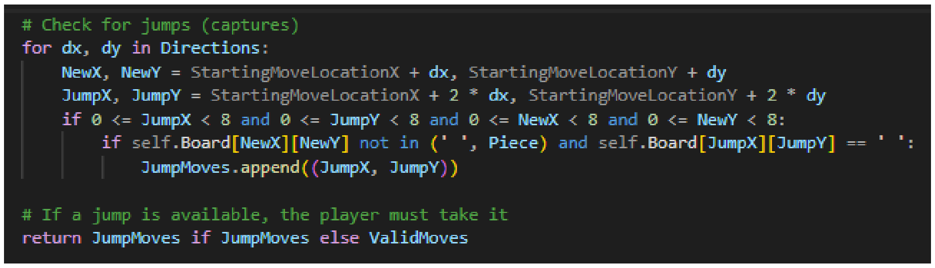
- **Movement**:

  - Pieces move diagonally forward.



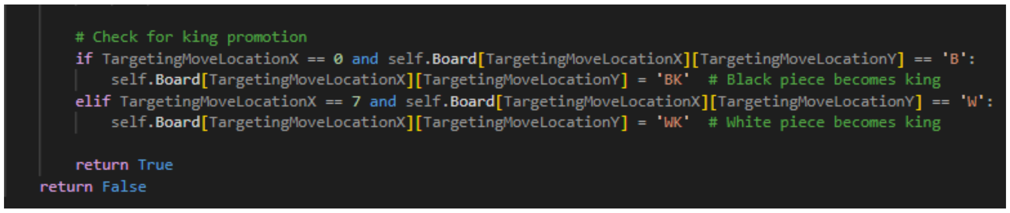
```python
def getValidMoves(self, StartingMoveLocationX, StartingMoveLocationY):
    """Returns a list of valid moves for a piece at (StartingMoveLocationX, StartingMoveLocationY), including jumps."""
    Piece = self.Board[StartingMoveLocationX][StartingMoveLocationY]
    if Piece == ' ':
        return []  # No piece to move

    Directions = []
    if Piece == 'W':
        Directions = [(1, -1), (1, 1)]  # White moves downward
    elif Piece == 'B':
        Directions = [(-1, -1), (-1, 1)]  # Black moves upward
    elif Piece in ('WK', 'BK'):  # King moves in all directions
        Directions = [(1, -1), (1, 1), (-1, -1), (-1, 1)]

    ValidMoves = []
    JumpMoves = []
```

  - Captures are made by jumping over an opponent’s piece.



```python
# Check for jumps (captures)
for dx, dy in Directions:
    NewX, NewY = StartingMoveLocationX + dx, StartingMoveLocationY + dy
    JumpX, JumpY = StartingMoveLocationX + 2 * dx, StartingMoveLocationY + 2 * dy
    if 0 <= JumpX < 8 and 0 <= JumpY < 8 and 0 <= NewX < 8 and 0 <= NewY < 8:
        if self.Board[NewX][NewY] not in (' ', Piece) and self.Board[JumpX][JumpY] == ' ':
            JumpMoves.append((JumpX, JumpY))

# If a jump is available, the player must take it
return JumpMoves if JumpMoves else ValidMoves
```

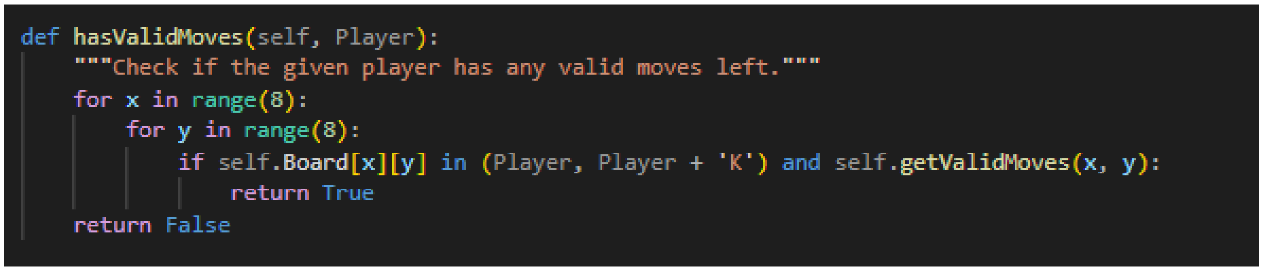
- Kings can move both forward and backward.



```python
        # Check for king promotion
        if TargetingMoveLocationX == 0 and self.Board[TargetingMoveLocationX][TargetingMoveLocationY] == 'B':
            self.Board[TargetingMoveLocationX][TargetingMoveLocationY] = 'BK'  # Black piece becomes king
        elif TargetingMoveLocationX == 7 and self.Board[TargetingMoveLocationX][TargetingMoveLocationY] == 'W':
            self.Board[TargetingMoveLocationX][TargetingMoveLocationY] = 'WK'  # White piece becomes king

        return True
    return False
```

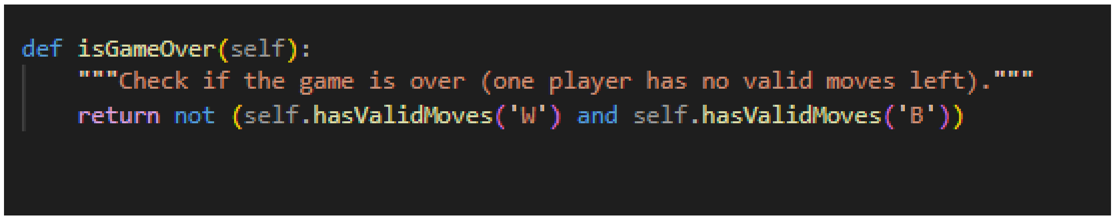
- **Objective/End of Game**: Capture all opponent pieces or block them from moving.

Check if given player has any valid moves yet



```python
def hasValidMoves(self, Player):
    """Check if the given player has any valid moves left."""
    for x in range(8):
        for y in range(8):
            if self.Board[x][y] in (Player, Player + 'K') and self.getValidMoves(x, y):
                return True
    return False
```

If one player has no valid moves left



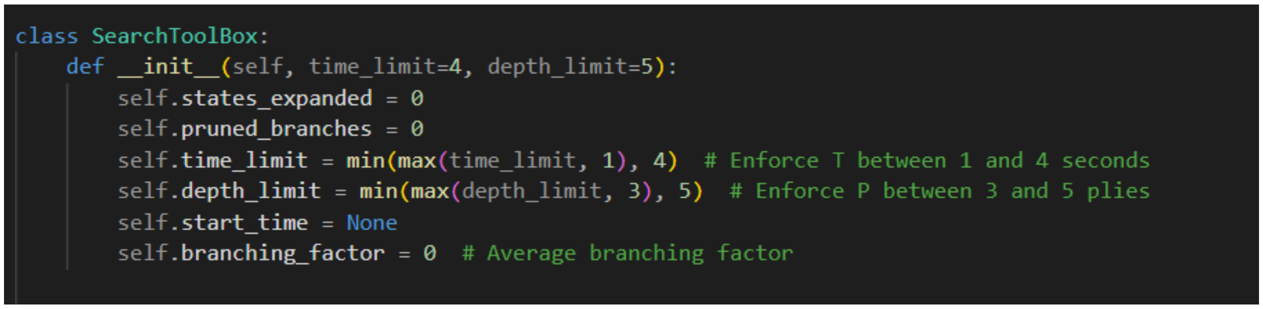
```python
def isGameOver(self):
    """Check if the game is over (one player has no valid moves left)."""
    return not (self.hasValidMoves('W') and self.hasValidMoves('B'))
```

2. Search Strategies Implementation:

2.1. SearchToolBox Overview:

The SearchToolBox class contains methods for implementing both Minimax and Alpha-Beta Pruning. It also tracks the number of states expanded, pruned branches, and limits on the time and depth of the search to ensure the algorithm runs efficiently.

- **Constructor (__init__)**:
  - Initializes the time and depth limits, ensuring they remain within the specified range (1 to 4 seconds for time and 3 to 5 plies for depth).
  - Tracks the number of expanded states and pruned branches.
  - Initializes the start time for the search.



```python
class SearchToolBox:
    def __init__(self, time_limit=4, depth_limit=5):
        self.states_expanded = 0
        self.pruned_branches = 0
        self.time_limit = min(max(time_limit, 1), 4)  # Enforce T between 1 and 4 seconds
        self.depth_limit = min(max(depth_limit, 3), 5)  # Enforce P between 3 and 5 plies
        self.start_time = None
        self.branching_factor = 0  # Average branching factor
```

2.2. Minimax Search Algorithm:

a. overview:

The Minimax algorithm is a classic decision-making technique employed in two-player, zero-sum games, such as Checkers, Chess, and Tic-Tac-Toe. It works by recursively evaluating all possible future game states up to a predefined depth (referred to as "plies"), with the objective of maximizing one player's score while minimizing the opponent's score. At each level of the game tree, the algorithm alternates between the two players, assuming that both will play optimally.

- **The time complexity** is O(b^d), where b represents the branching factor (the average number of possible moves per game state) and d is the depth of the search.
- **The space complexity** is O(b * d), as the algorithm must store the nodes of the game tree in memory during traversal.

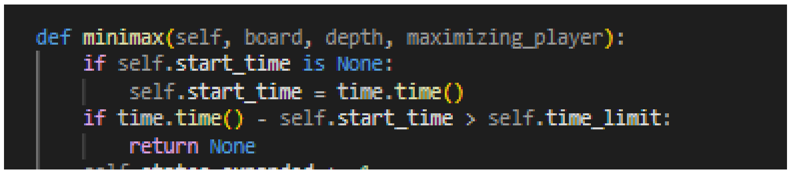
While effective in smaller games, the Minimax algorithm suffers from significant computational inefficiencies in games with large search spaces. the final decision, thereby reducing the number of nodes evaluated.

Below is an explanation of the key components and functions of the Minimax algorithm implemented in the SearchToolBox class:

b. Minimax Implementation:

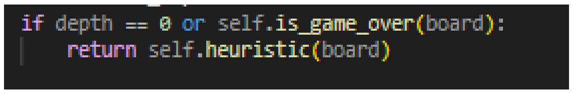
In the Minimax algorithm, the **maximizing player** is typically the AI, represented by maximizing_player = True. This player aims to maximize their score by selecting the best possible moves. The **minimizing player**, represented by maximizing_player = False, is usually the human player, who tries to minimize the AI's score and limit its advantage. The algorithm alternates between these two roles, with the AI attempting to maximize its benefit and the human aiming to reduce it.

1. Initialization of Time Limit Check:


```
def minimax(self, board, depth, maximizing_player):
    if self.start_time is None:
        self.start_time = time.time()
    if time.time() - self.start_time > self.time_limit:
        return None
```

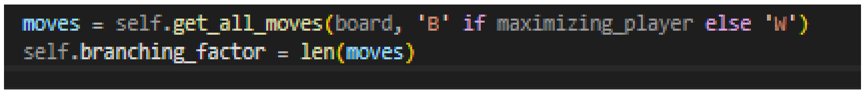
- self.start_time: This checks if the starting time of the evaluation has already been recorded. If not, it initializes the start time using time.time().
- Time Limit Check: The code then checks if the time spent on the search exceeds a predefined time limit (self.time_limit). If the time exceeds the limit, it terminates the search early and returns None to avoid excessive computation.

2. Base case - Depth or Game Over:


```
if depth == 0 or self.is_game_over(board):
    return self.heuristic(board)
```

- **Depth Check**: If the search depth reaches 0, the function will return the heuristic value for the current board. The depth is a measure of how many levels deep the algorithm will explore.
- **Game Over Check**: If the game is over (using the self.is_game_over(board) method), it will also return the heuristic value of the board. This is the base case of the recursion.

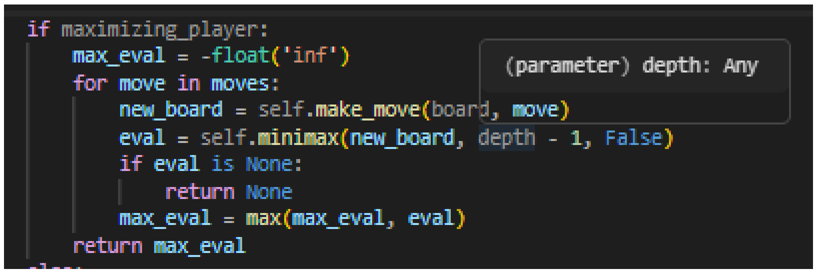
3. Get Possible Moves:


```
moves = self.get_all_moves(board, 'B' if maximizing_player else 'W')
self.branching_factor = len(moves)
```

- **self.get_all_moves**(board, 'B' if maximizing_player else 'W'): This generates all possible moves for the current player. If it's the maximizing player's turn, it considers 'B' (Black); otherwise, it considers 'W' (White).

- **Branching Factor**: The variable self.branching_factor is set to the number of possible moves, representing how many branches (or choices) are available at this stage of the search.

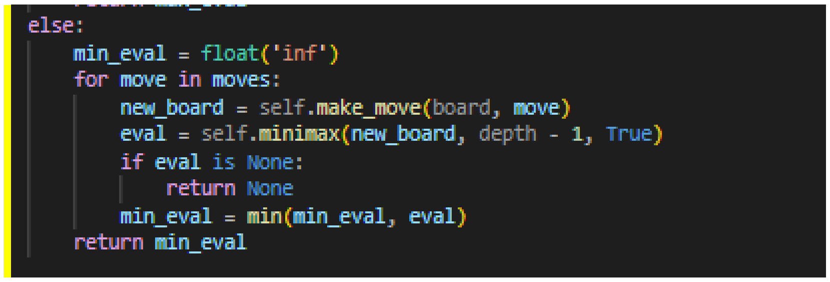
4. Maximizing Player (AI's Turn):



```
if maximizing_player:
    max_eval = -float('inf')
    for move in moves:
        new_board = self.make_move(board, move)
        eval = self.minimax(new_board, depth - 1, False)
        if eval is None:
            return None
        max_eval = max(max_eval, eval)
    return max_eval
```

- **Maximizing Player:** If it's the maximizing player's turn (AI), the algorithm aims to choose the move that gives the highest possible score. It initializes max_eval to negative infinity to ensure any evaluated move will be greater.
- **Recursive Call:** For each possible move, it generates a new board state (new_board), and recursively calls the minimax function with reduced depth (depth - 1) and switches the turn to the opponent (passing False to maximizing_player).
- **Check for Time Limit:** If eval is None, indicating the time limit was exceeded, the function returns None.
- **Update Best Move:** It updates max_eval to store the maximum evaluation of all moves.

5. Minimizing Player (Opponent's Turn):



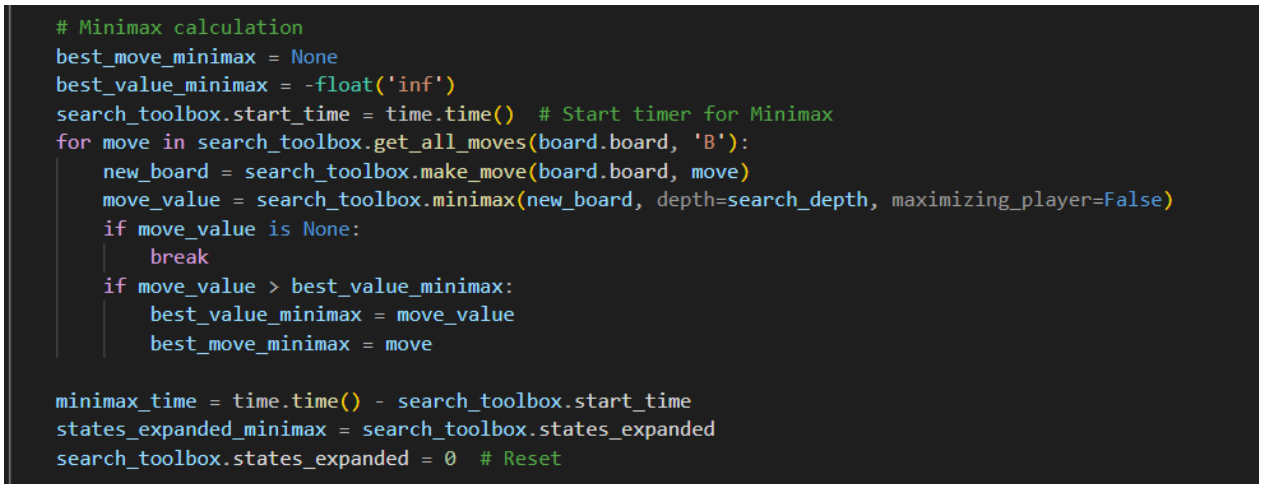
```
else:
    min_eval = float('inf')
    for move in moves:
        new_board = self.make_move(board, move)
        eval = self.minimax(new_board, depth - 1, True)
        if eval is None:
            return None
        min_eval = min(min_eval, eval)
    return min_eval
```

- **Minimizing Player**: If it's the minimizing player's turn (the opponent), the algorithm seeks to minimize the evaluation score. It initializes min_eval to positive infinity.
- **Recursive Call:** Similarly to the maximizing player's section, it makes a move, calls minimax recursively with reduced depth and the maximizing flag set to True (indicating the turn switches back to the AI), and evaluates the board.
- **Check for Time Limit:** Again, if eval is None, the function returns None.
- **Update Best Move:** It updates min_eval to store the minimum evaluation of all moves.

6. **Return the Best Evaluation:**

   For the maximizing player, it returns max_eval (the best possible value found), and for the minimizing player, it returns min_eval.

c. Minimax analysis:



```
# Minimax calculation
best_move_minimax = None
best_value_minimax = -float('inf')
search_toolbox.start_time = time.time()  # Start timer for Minimax
for move in search_toolbox.get_all_moves(board.board, 'B'):
    new_board = search_toolbox.make_move(board.board, move)
    move_value = search_toolbox.minimax(new_board, depth=search_depth, maximizing_player=False)
    if move_value is None:
        break
    if move_value > best_value_minimax:
        best_value_minimax = move_value
        best_move_minimax = move

minimax_time = time.time() - search_toolbox.start_time
states_expanded_minimax = search_toolbox.states_expanded
search_toolbox.states_expanded = 0  # Reset
```

The analysis for the Minimax calculation is done in PlayingTheGame.py involves evaluating the AI's decision-making process by measuring key components such as time, states expanded, and the chosen best move.

The **best move** is determined by simulating all possible moves for the AI (in this case, using the 'B' pieces), calculating the Minimax value for each move, and selecting the move with the highest value.

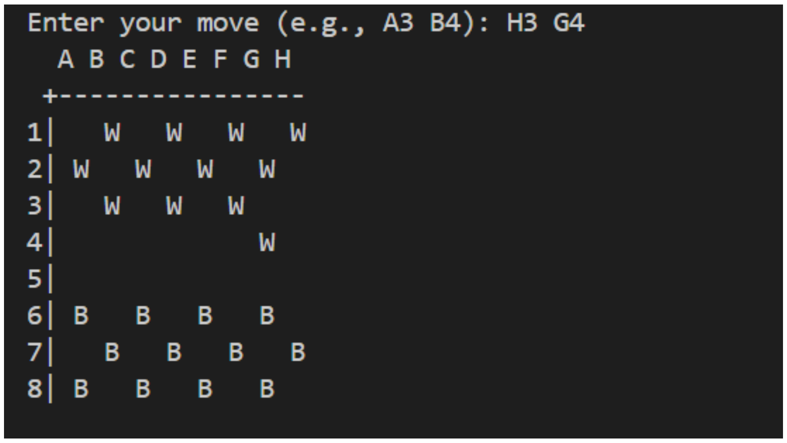
The **best value** represents the optimal score the AI can achieve from a given move.

The **time taken** for the Minimax algorithm to calculate the best move is recorded to measure performance.

While the **states expanded** tracks the number of game states evaluated during the process. This metric helps in understanding the complexity of the algorithm's search. After each evaluation, the **states_expanded** counter is reset to ensure that the count only reflects the current run.

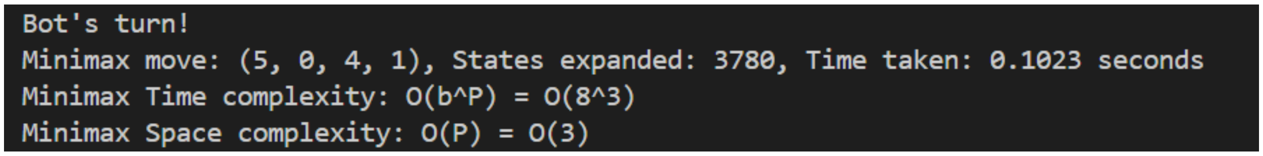
d. Minimax results:

The player's first move:

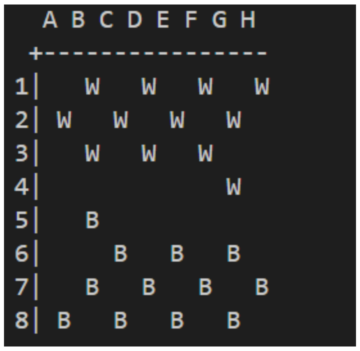


```
Enter your move (e.g., A3 B4): H3 G4
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W   W
4|             W
5|
6| B   B   B   B
7|   B   B   B   B
8| B   B   B   B
```

The bot's first move:

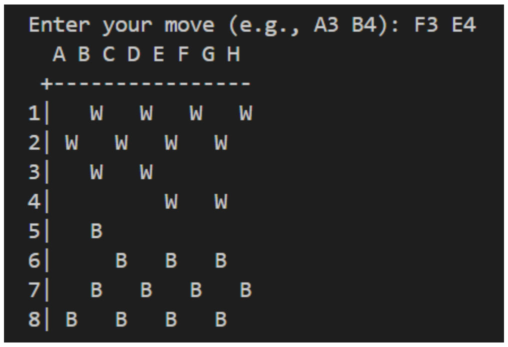


```
Bot's turn!
Minimax move: (5, 0, 4, 1), States expanded: 3780, Time taken: 0.1023 seconds
Minimax Time complexity: O(b^P) = O(8^3)
Minimax Space complexity: O(P) = O(3)
```



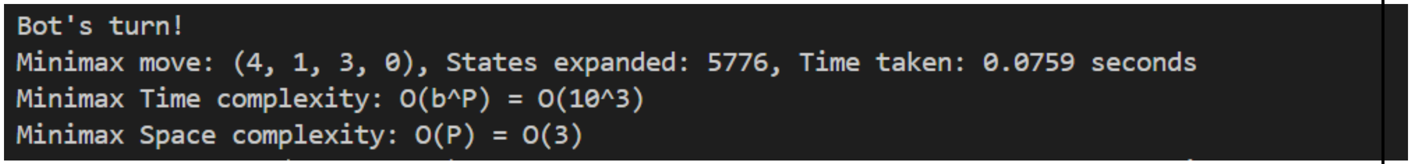
```
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W   W
4|             W
5|   B
6|     B   B   B
7|   B   B   B   B
8| B   B   B   B
```

The player’s second move:

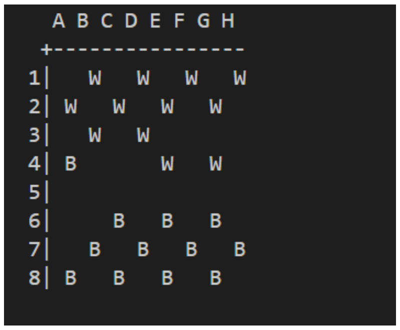


```
Enter your move (e.g., A3 B4): F3 E4
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W
4|         W   W
5|   B
6|     B   B   B
7|   B   B   B   B
8| B   B   B   B
```

The bot’s second move:

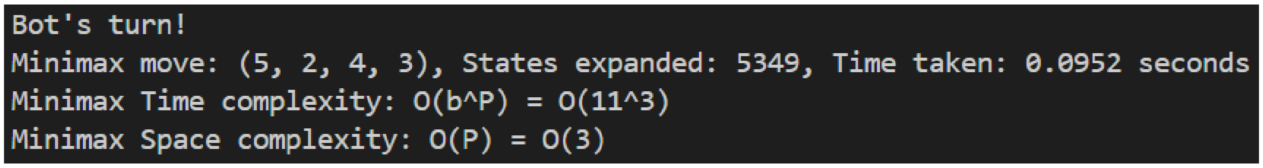


```
Bot's turn!
Minimax move: (4, 1, 3, 0), States expanded: 5776, Time taken: 0.0759 seconds
Minimax Time complexity: O(b^P) = O(10^3)
Minimax Space complexity: O(P) = O(3)
```

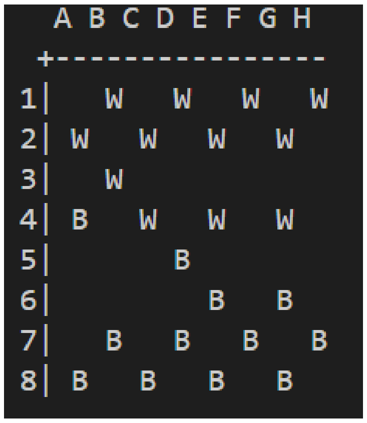


```
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W
4| B       W   W
5|
6|     B   B   B
7|   B   B   B   B
8| B   B   B   B
```

[………]bot's third move:

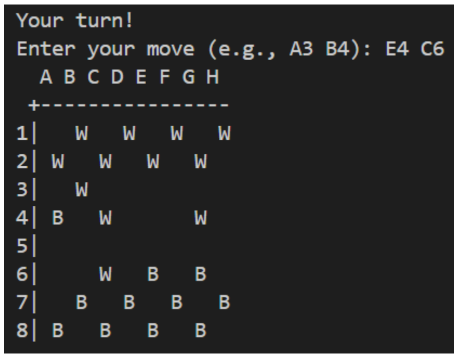


```
Bot's turn!
Minimax move: (5, 2, 4, 3), States expanded: 5349, Time taken: 0.0952 seconds
Minimax Time complexity: O(b^P) = O(11^3)
Minimax Space complexity: O(P) = O(3)
```



```
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W
4| B   W   W   W
5|       B
6|         B   B
7|   B   B   B   B
8| B   B   B   B
```

The player's second move (capturing the bot's checker):



```
Your turn!
Enter your move (e.g., A3 B4): E4 C6
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W
4| B   W       W
5|
6|     W   B   B
7|   B   B   B   B
8| B   B   B   B
```

[………]

A Full Game Demo will be displayed at the end of the report

2.3. Alpha-Beta Pruning Algorithm:

a. overview:

Alpha-Beta Pruning is an optimization technique applied to the Minimax algorithm to improve its efficiency in two-player, zero-sum games like Checkers, Chess, and Tic-Tac-Toe. The Alpha-Beta algorithm works by pruning branches of the game tree that will not influence the final decision, thereby reducing the number of nodes that need to be evaluated. This is achieved through two parameters: **alpha**, which tracks the maximum score achievable by the maximizing player, and **beta**, which tracks the minimum score achievable by the minimizing player. When the value of a node exceeds the current **beta** or falls below **alpha**, further exploration of that branch is stopped, thus optimizing the search.

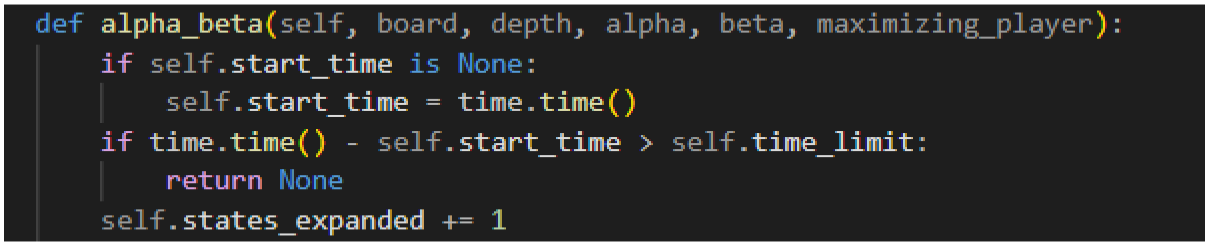
- **Time Complexity**: The worst-case time complexity is still O(b^d), where b is the branching factor and d is the depth of the tree, but Alpha-Beta Pruning can significantly reduce the number of nodes evaluated, improving performance in most cases. In practice, the algorithm can achieve a time complexity of O(b^(d/2)) under ideal conditions with perfect move ordering.
- **Space Complexity**: The space complexity remains O(b * d), as the algorithm still needs to store the game tree in memory during traversal.

By pruning unnecessary branches, Alpha-Beta Pruning makes the decision-making process more efficient and faster, especially in games with large search spaces. This makes it a powerful enhancement over the basic Minimax algorithm, especially when dealing with complex games like Checkers or Chess.

b. Alpha-Beta Pruning implementation:

The alpha-beta function is an implementation of the Alpha-Beta Pruning algorithm, which optimizes the Minimax search algorithm for two-player, zero-sum games by pruning branches of the game tree that cannot influence the final decision. Below is a detailed description of how the function works, step-by-step:

1. Initial setup:

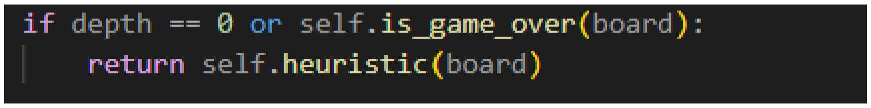


```
def alpha_beta(self, board, depth, alpha, beta, maximizing_player):
    if self.start_time is None:
        self.start_time = time.time()
    if time.time() - self.start_time > self.time_limit:
        return None
    self.states_expanded += 1
```

This section of the code checks the start time to track the execution time of the algorithm. If the algorithm has been running longer than the predefined time limit, the function returns None, indicating that the computation should be stopped due to a time constraint.

The states_expanded counter is incremented to track how many game states have been evaluated.

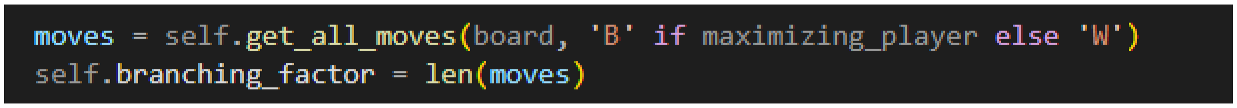
2. Base Case:



```
if depth == 0 or self.is_game_over(board):
    return self.heuristic(board)
```

The function checks if the search depth (depth) has reached zero, or if the game is over (using self.is_game_over(board)). If either condition is met, the function returns the heuristic evaluation of the current board state (self.heuristic(board)), which serves as the evaluation function to assess the quality of the board for the current player.

3. Generate Possible Moves:

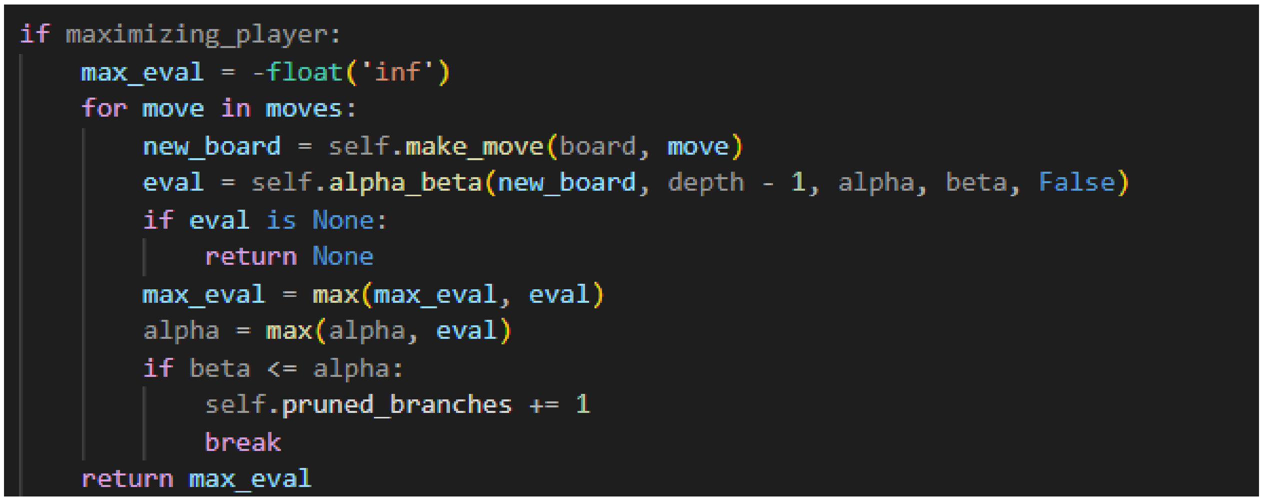


```
moves = self.get_all_moves(board, 'B' if maximizing_player else 'W')
self.branching_factor = len(moves)
```

The get_all_moves function is called to generate all valid moves for the current player. Depending on whether the current player is the maximizing player or the minimizing player, the function considers either 'B' (Black) or 'W' (White) as the player.

The branching factor is updated to reflect the number of valid moves available at the current state.

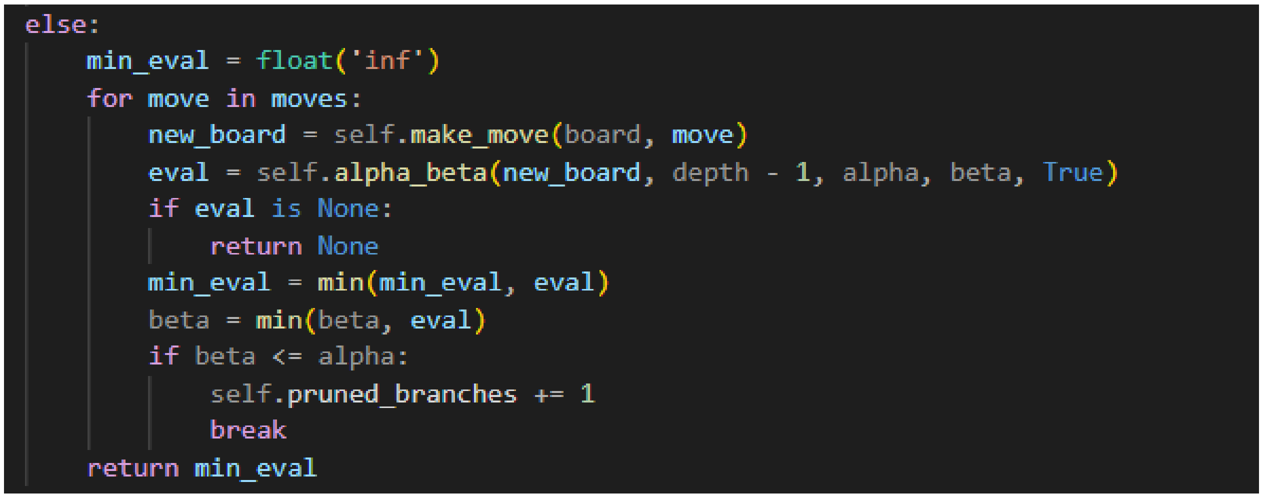
4. Maximizing Player's Move:



```
if maximizing_player:
    max_eval = -float('inf')
    for move in moves:
        new_board = self.make_move(board, move)
        eval = self.alpha_beta(new_board, depth - 1, alpha, beta, False)
        if eval is None:
            return None
        max_eval = max(max_eval, eval)
        alpha = max(alpha, eval)
        if beta <= alpha:
            self.pruned_branches += 1
            break
    return max_eval
```

- If the current player is the maximizing player, the algorithm sets max_eval to negative infinity, as it is searching for the maximum evaluation value.
- For each possible move, the board is updated with the move, and the alpha_beta function is recursively called to evaluate the new board state.
- If the returned evaluation (eval) is None (meaning time has expired or a cutoff condition was met), the function returns None.
- The maximum evaluation is updated with the best value found across all moves, and alpha is updated to ensure that the algorithm tracks the best possible score for the maximizing player.
- If beta is less than or equal to alpha, further moves are pruned (cut off from evaluation) because the opponent will not allow this branch to be explored (this is a crucial part of Alpha-Beta Pruning). The pruned_branches counter is incremented to track how many branches were pruned.

5. Minimizing Player's Move:



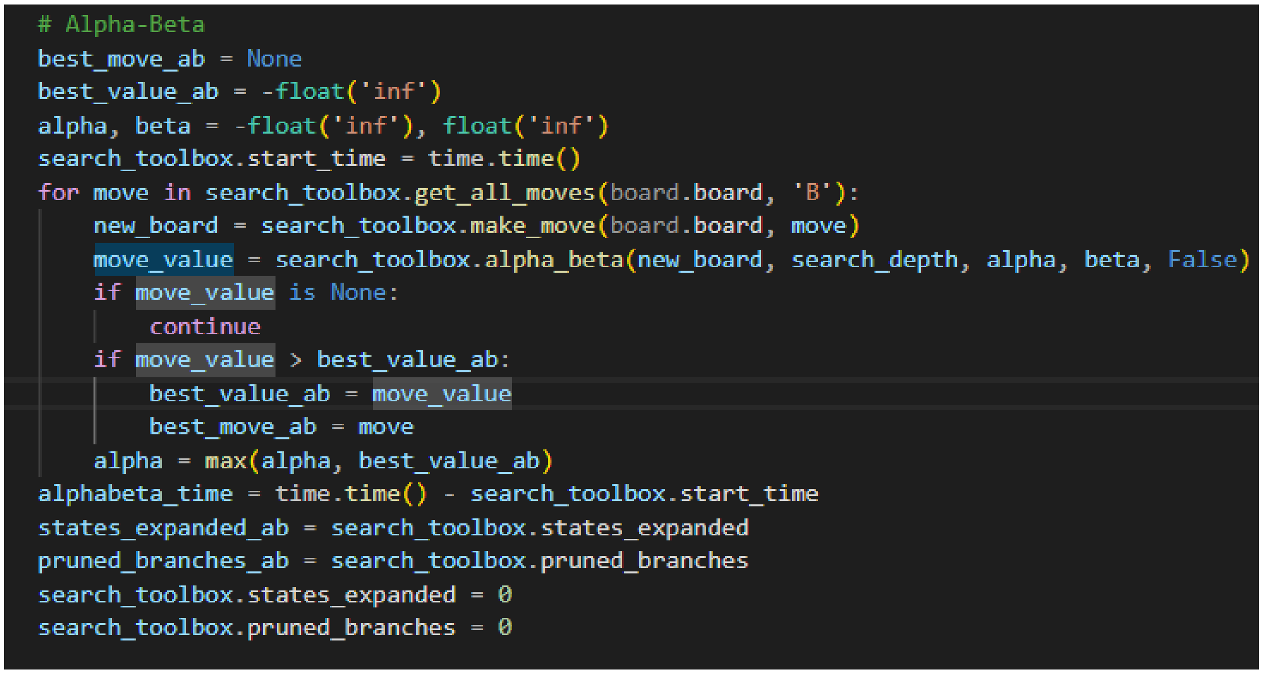
```
else:
    min_eval = float('inf')
    for move in moves:
        new_board = self.make_move(board, move)
        eval = self.alpha_beta(new_board, depth - 1, alpha, beta, True)
        if eval is None:
            return None
        min_eval = min(min_eval, eval)
        beta = min(beta, eval)
        if beta <= alpha:
            self.pruned_branches += 1
            break
    return min_eval
```

- If the current player is the **minimizing player**, the algorithm sets **min_eval** to positive infinity, as it is searching for the minimum evaluation value.
- Similarly to the maximizing player, for each possible move, the board is updated, and the alpha_beta function is recursively called to evaluate the new board state.
- If the evaluation is None, the function returns None.
- The **minimum evaluation** is updated with the best value found across all moves, and **beta** is updated to track the worst possible score for the minimizing player.
- If **beta** is less than or equal to **alpha**, the branch is pruned, and the **pruned_branches** counter is incremented.

6. Final Return:

After evaluating all moves for both players, the function returns the best evaluation found (max_eval for the maximizing player or min_eval for the minimizing player).

c. Alpha-Beta pruning Analysis:



```python
# Alpha-Beta
best_move_ab = None
best_value_ab = -float('inf')
alpha, beta = -float('inf'), float('inf')
search_toolbox.start_time = time.time()
for move in search_toolbox.get_all_moves(board.board, 'B'):
    new_board = search_toolbox.make_move(board.board, move)
    move_value = search_toolbox.alpha_beta(new_board, search_depth, alpha, beta, False)
    if move_value is None:
        continue
    if move_value > best_value_ab:
        best_value_ab = move_value
        best_move_ab = move
    alpha = max(alpha, best_value_ab)
alphabeta_time = time.time() - search_toolbox.start_time
states_expanded_ab = search_toolbox.states_expanded
pruned_branches_ab = search_toolbox.pruned_branches
search_toolbox.states_expanded = 0
search_toolbox.pruned_branches = 0
```

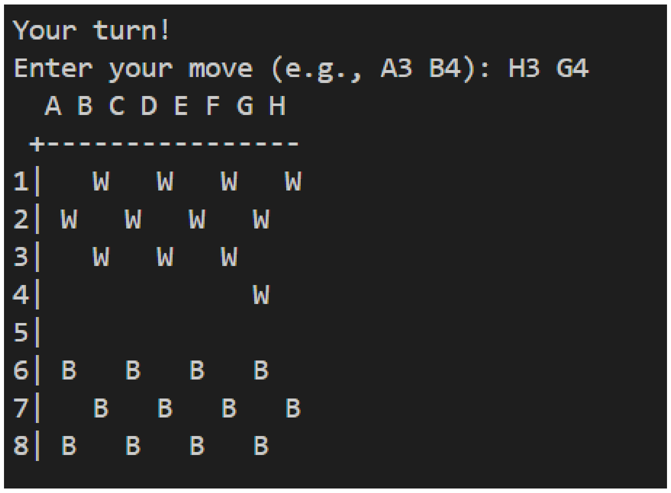
The analysis for the Alpha-Beta pruning calculation is done in PlayingTheGame.py, where the AI's decision-making process is evaluated by tracking key components such as time, states expanded, pruned branches, and the selected best move.

The best move is determined by simulating all possible moves for the AI (using the 'B' pieces), calculating the value for each move through the Alpha-Beta pruning algorithm, and selecting the move with the highest value. This value represents the optimal score the AI can achieve from a given move, while also considering the pruning of less promising branches of the game tree.

The time taken for the Alpha-Beta pruning to evaluate the best move is recorded, helping to measure the performance and efficiency of the algorithm. The states_expanded metric tracks the number of game states evaluated during the process, providing insight into the complexity of the search. Meanwhile, the pruned_branches metric counts the number of branches that were pruned, showing how much computation was saved by ignoring irrelevant branches. After each evaluation, the states_expanded and pruned_branches counters are reset to ensure that the metrics only reflect the current run.

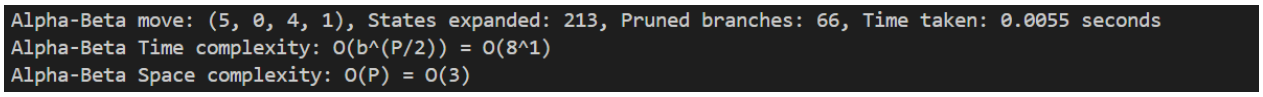
d. Alpha-Beta pruning Results:

The player’s first move:

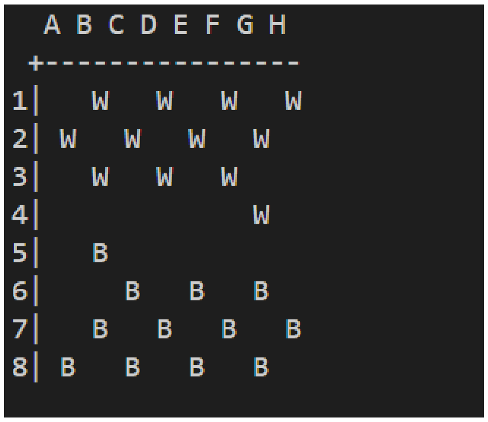


```
Your turn!
Enter your move (e.g., A3 B4): H3 G4
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W   W
4|             W
5|
6| B   B   B   B
7|   B   B   B   B
8| B   B   B   B
```

The bot's first move:

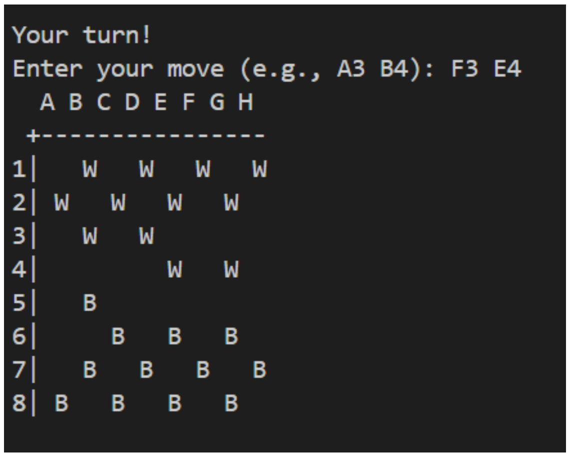


```
Alpha-Beta move: (5, 0, 4, 1), States expanded: 213, Pruned branches: 66, Time taken: 0.0055 seconds
Alpha-Beta Time complexity: O(b^(P/2)) = O(8^1)
Alpha-Beta Space complexity: O(P) = O(3)
```



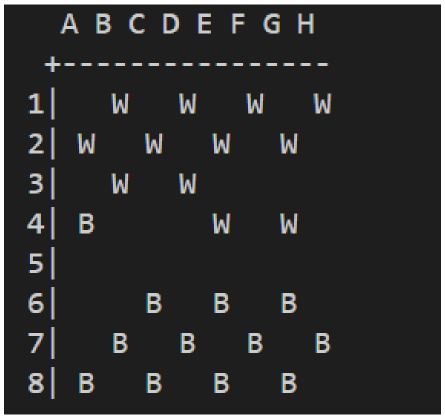
```
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W   W
4|             W
5|   B
6|     B   B   B
7|   B   B   B   B
8| B   B   B   B
```

The player's second move:

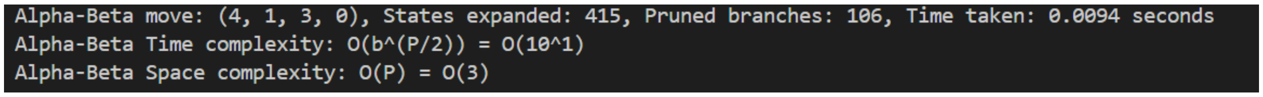


```
Your turn!
Enter your move (e.g., A3 B4): F3 E4
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W
4|         W   W
5|   B
6|     B   B   B
7|   B   B   B   B
8| B   B   B   B
```

The bots's second move:



```
  A B C D E F G H
 +----------------
1|   W   W   W   W
2| W   W   W   W
3|   W   W
4| B       W   W
5|
6|     B   B   B
7|   B   B   B   B
8| B   B   B   B
```



```
Alpha-Beta move: (4, 1, 3, 0), States expanded: 415, Pruned branches: 106, Time taken: 0.0094 seconds
Alpha-Beta Time complexity: O(b^(P/2)) = O(10^1)
Alpha-Beta Space complexity: O(P) = O(3)
```

[………]

A Full Game Demo will be displayed at the end of the report

2.4. Alpha-Beta Pruning with ordering of nodes Algorithm:

a. overview:

The alpha_beta_ordered function is an optimized version of the Alpha-Beta Pruning algorithm that enhances efficiency by incorporating move ordering. By sorting potential moves based on their heuristic values before evaluation, the function prioritizes stronger moves for the maximizing player and weaker moves for the minimizing player. This ordering increases the likelihood of pruning irrelevant branches earlier, significantly reducing the number of game states explored. As a result, the algorithm maintains optimal decision-making while improving performance, making it a more efficient approach for AI-driven decision processes in competitive games like Checkers.

The complexity depends on how well pruning is applied:

- **Worst Case (No Pruning, Equivalent to Minimax):**

O(bd)O(b^d)O(bd), where b is the branching factor (number of possible moves per turn), and d is the search depth. This happens when pruning is ineffective, making it as slow as Minimax.

- **Best Case (Perfect Pruning with Optimal Move Ordering):**

  O(bd/2)O(b^{d/2})O(bd/2), which means the number of evaluated states is reduced exponentially. The search depth effectively cuts in half, leading to significant performance gains.

- **Average Case:**

Falls between O(bd/2)O(b^{d/2})O(bd/2) and O(bd)O(b^d)O(bd), depending on the effectiveness of move ordering and pruning opportunities.

Space Complexity:

- The algorithm uses recursive depth-first search (DFS), leading to a recursive call stack of depth d.
- Storing legal moves at each step takes O(b) extra space.

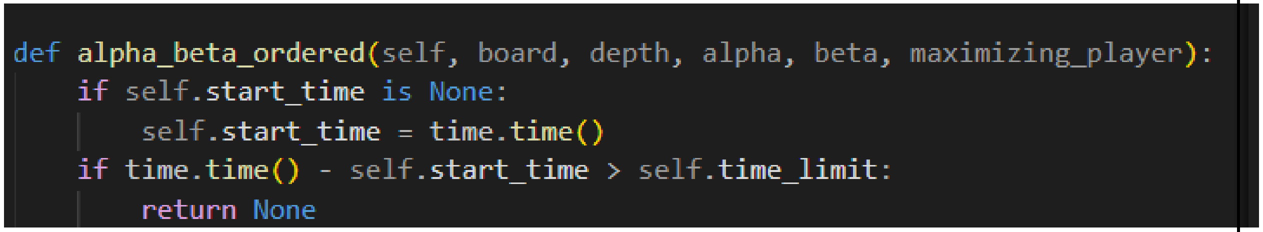
Thus, the overall space complexity is:

- O(d) in recursive implementations (due to call stack depth).
- O(1) additional memory in iterative implementations, since only a few variables (alpha, beta, best move) need to be stored at each level.

By efficiently pruning unnecessary branches, Alpha-Beta Pruning significantly improves Minimax's performance while maintaining optimal decision-making.

b. Alpha-Beta Pruning with ordering of nodes implementation:

1. Handling Execution Time Constraints:

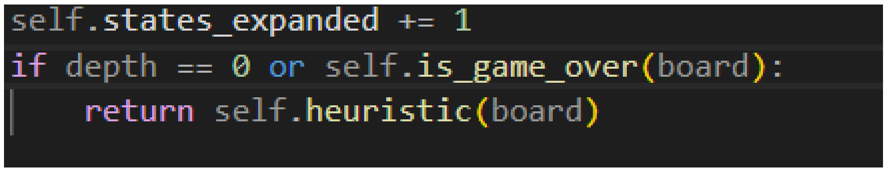


```
def alpha_beta_ordered(self, board, depth, alpha, beta, maximizing_player):
    if self.start_time is None:
        self.start_time = time.time()
    if time.time() - self.start_time > self.time_limit:
        return None
```

- **Purpose:** Ensures the algorithm does not exceed a predefined time limit.

- **How it works:** The function records the start time and continuously checks if the elapsed time has exceeded self.time_limit. If exceeded, it returns None to stop execution.

2. Base Case: Stopping Condition



```
self.states_expanded += 1
if depth == 0 or self.is_game_over(board):
    return self.heuristic(board)
```

Purpose: Defines when to stop recursion.

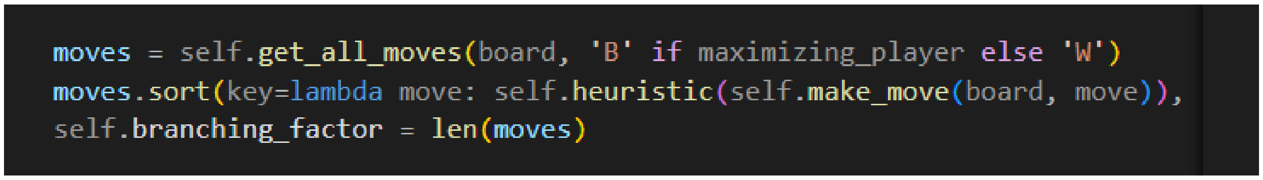
Stopping Conditions:

If depth == 0, meaning we have reached the maximum search depth.

If is_game_over(board) returns True, indicating the game has ended.

Return Value: The heuristic evaluation of the board, which represents how favorable the board is for the AI.

3. Generating and Sorting Moves



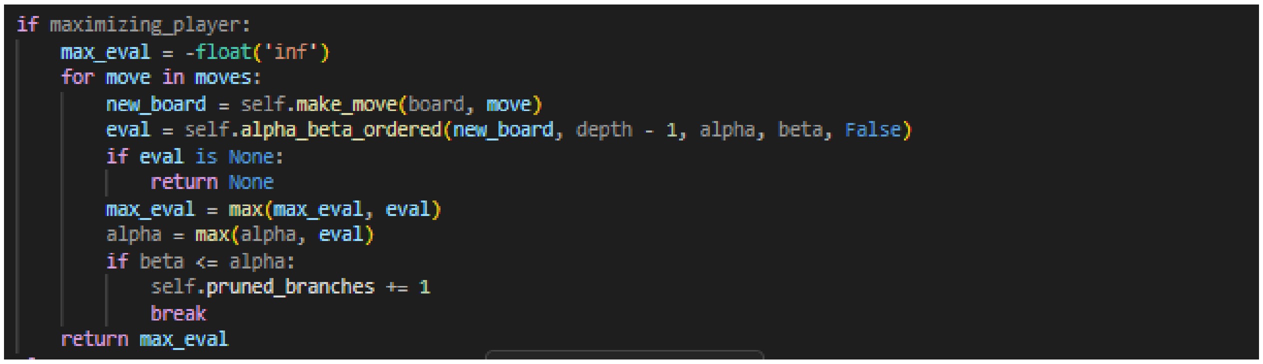
```
moves = self.get_all_moves(board, 'B' if maximizing_player else 'W')
moves.sort(key=lambda move: self.heuristic(self.make_move(board, move)),
self.branching_factor = len(moves)
```

- **Purpose:** Retrieves all possible moves and sorts them using a heuristic function.
- **How Sorting Helps:**

- If the AI is **maximizing**, it sorts moves in **descending order** (best moves first).
- If the AI is **minimizing**, it sorts moves in **ascending order** (worst moves first).
- This increases the chances of early pruning, improving efficiency.

4. Recursive Search with Alpha-Beta Pruning

Maximizing Player's Turn (`'B' player)



```
if maximizing_player:
    max_eval = -float('inf')
    for move in moves:
        new_board = self.make_move(board, move)
        eval = self.alpha_beta_ordered(new_board, depth - 1, alpha, beta, False)
        if eval is None:
            return None
        max_eval = max(max_eval, eval)
        alpha = max(alpha, eval)
        if beta <= alpha:
            self.pruned_branches += 1
            break
    return max_eval
```

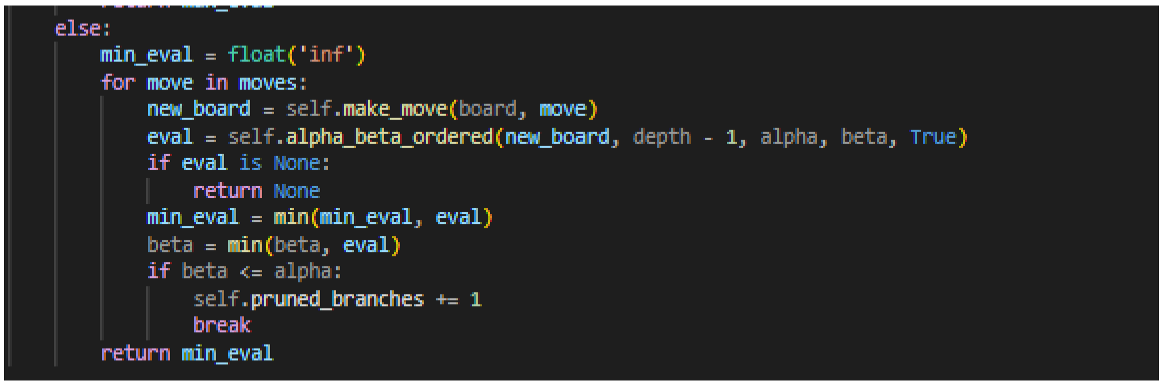
Purpose: The AI selects the best move by maximizing the evaluation score.

Key Steps:

1. Loops through each possible move.
2. Simulates the move using make_move(board, move).
3. Calls alpha_beta_ordered recursively for the **opponent** (minimizing player).
4. Updates max_eval to keep track of the best score found.
5. Updates alpha = max(alpha, eval), ensuring **future branches are pruned** if beta <= alpha.

6. **If pruning occurs (beta <= alpha)**, the loop **breaks** early.

Minimizing Player's Turn (`'W' player)



```
else:
    min_eval = float('inf')
    for move in moves:
        new_board = self.make_move(board, move)
        eval = self.alpha_beta_ordered(new_board, depth - 1, alpha, beta, True)
        if eval is None:
            return None
        min_eval = min(min_eval, eval)
        beta = min(beta, eval)
        if beta <= alpha:
            self.pruned_branches += 1
            break
    return min_eval
```

- **Purpose:** The AI selects the best move by minimizing the evaluation score.

- **Key Steps:**

    1. Loops through each move.

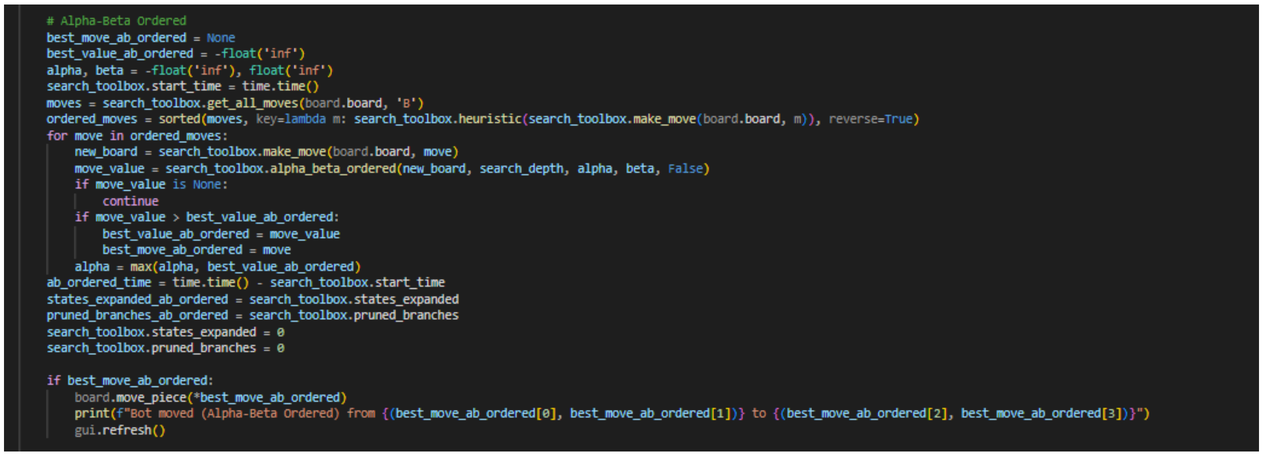
    2. Simulates the move and evaluates it recursively.

    3. Updates min_eval to keep track of the **lowest** score found.

    4. Updates beta = min(beta, eval) to allow pruning.

    5. **If pruning occurs (beta <= alpha)**, the loop **breaks** early.

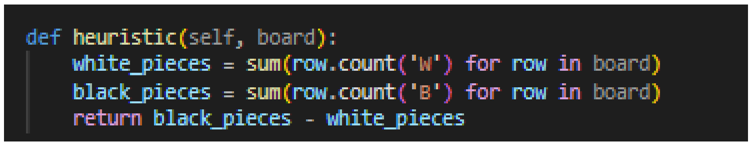
c. Alpha-Beta_ordered pruning Analysis:



```python
# Alpha-Beta Ordered
best_move_ab_ordered = None
best_value_ab_ordered = -float('inf')
alpha, beta = -float('inf'), float('inf')
search_toolbox.start_time = time.time()
moves = search_toolbox.get_all_moves(board.board, 'B')
ordered_moves = sorted(moves, key=lambda m: search_toolbox.heuristic(search_toolbox.make_move(board.board, m)), reverse=True)
for move in ordered_moves:
    new_board = search_toolbox.make_move(board.board, move)
    move_value = search_toolbox.alpha_beta_ordered(new_board, search_depth, alpha, beta, False)
    if move_value is None:
        continue
    if move_value > best_value_ab_ordered:
        best_value_ab_ordered = move_value
        best_move_ab_ordered = move
    alpha = max(alpha, best_value_ab_ordered)
ab_ordered_time = time.time() - search_toolbox.start_time
states_expanded_ab_ordered = search_toolbox.states_expanded
pruned_branches_ab_ordered = search_toolbox.pruned_branches
search_toolbox.states_expanded = 0
search_toolbox.pruned_branches = 0

if best_move_ab_ordered:
    board.move_piece(*best_move_ab_ordered)
    print(f"Bot moved (Alpha-Beta Ordered) from {(best_move_ab_ordered[0], best_move_ab_ordered[1])} to {(best_move_ab_ordered[2], best_move_ab_ordered[3])}")
    gui.refresh()
```

The Alpha-Beta Ordered algorithm starts by initializing variables to track the best move, its evaluation value, and the Alpha-Beta bounds (alpha and beta). The process is timed, and all possible moves for the bot (player 'B') are retrieved. These moves are ordered based on the heuristic values of the resulting boards, prioritizing moves that provide better outcomes. The algorithm then iterates through the ordered moves, applying the Alpha-Beta pruning method recursively to evaluate the best move. During this process, the algorithm continuously updates the best move and its evaluation value. When a pruning condition is met (beta ≤ alpha), branches are pruned, improving efficiency by reducing unnecessary calculations. After the search concludes, the best move is executed on the board, and the time taken for the evaluation is calculated, helping assess the algorithm's speed. The number of states expanded is tracked to measure the complexity of the search, showing how many possible board configurations were evaluated. Additionally, the number of pruned branches is recorded, highlighting how much the pruning process reduced the search space. Finally, these metrics are reset for the next search cycle, and the board is updated, with the GUI reflecting the bot's chosen move. These metrics: time, states **expanded, and pruned branches, provide insights into the algorithm's efficiency and effectiveness** in selecting the best move.

2.5. The heuristic function:



```python
def heuristic(self, board):
    white_pieces = sum(row.count('W') for row in board)
    black_pieces = sum(row.count('B') for row in board)
    return black_pieces - white_pieces
```

The heuristic function evaluates the board state by counting the number of pieces for each player and returning a numerical value based on the difference between the two.

1. **Counting White Pieces**:

   white_pieces = sum(row.count('W') for row in board)

   This line calculates the total number of white pieces ('W') on the board. It loops through each row in the board and counts how many occurrences of 'W' there are, summing them across all rows.

2. **Counting Black Pieces**:

   black_pieces = sum(row.count('B') for row in board)

   Similarly, this line calculates the total number of black pieces ('B') on the board. It works in the same way as the previous line, counting the 'B' pieces in each row and summing the total.

3. **Returning the Difference**:

   return black_pieces - white_pieces

   Finally, the function returns the difference between the number of black pieces and the number of white pieces. A positive result means there are more black pieces on the board, while a negative result indicates more white pieces.

Purpose:

This heuristic function is a simple evaluation method that gives a value indicating which player has a greater presence on the board. It can be used in search algorithms like Minimax or Alpha-Beta Pruning to guide the AI towards moves that maximize its advantage (black pieces in this case) and minimize its opponent's advantage.

3. Comparing Results:

3.1. Expected Results:

Algorithm	States Expanded	Pruned Branches	Efficiency
Minimax (No Pruning)	High	None	Slowest
Alpha-Beta (Unordered)	Moderate	Some	Faster
Alpha-Beta (Ordered)	Lowest	Most	Fastest

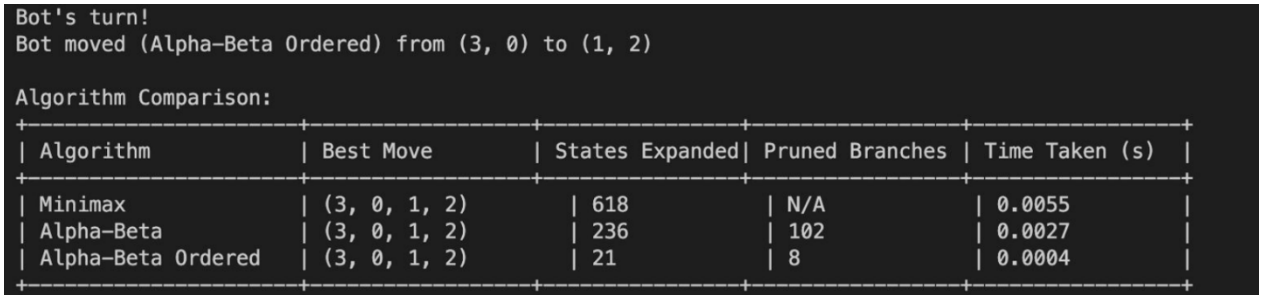
3.2. Justification:

The **Minimax (No Pruning)** algorithm explores the entire search tree without any optimization techniques like pruning. As a result, the number of states expanded is very high since it evaluates all possible moves and their subsequent consequences without cutting off any branches. This exhaustive exploration of all possibilities leads to a significant computational cost and slow performance, making it the slowest among the three algorithms. Since no pruning occurs, all branches are explored, which means no branches are eliminated from the search process, contributing to the algorithm's inefficiency.

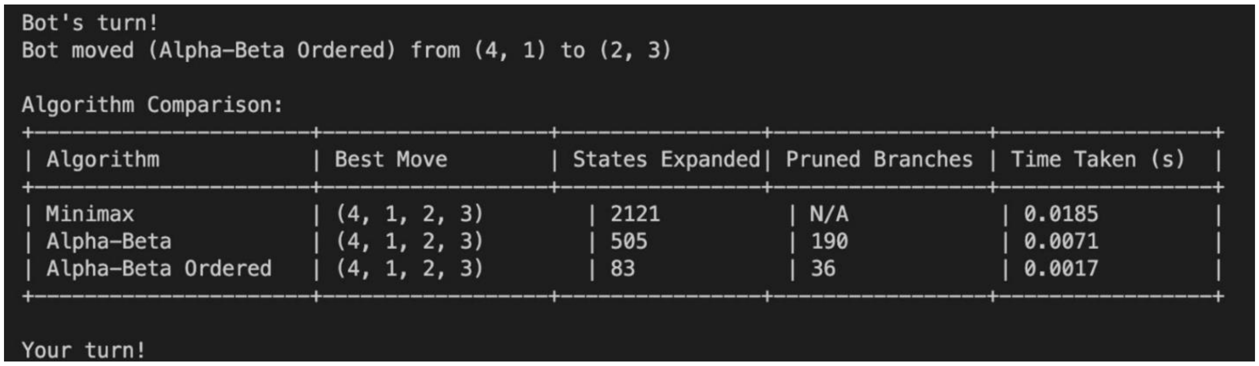
In comparison, the **Alpha-Beta (Unordered)** algorithm introduces pruning, which significantly reduces the search space by eliminating branches that will not affect the final decision. This pruning allows the algorithm to explore fewer states than Minimax, resulting in a moderate number of states expanded. However, since the moves are not ordered, the pruning is less effective, and some branches are still explored unnecessarily. While the pruning helps improve efficiency, the lack of move ordering means that the algorithm still has to explore a considerable portion of the tree, making it faster than Minimax but not as efficient as it could be.

The **Alpha-Beta (Ordered)** algorithm further enhances efficiency by sorting moves before evaluating them. By prioritizing the most promising moves, the algorithm maximizes pruning, significantly reducing the number of states expanded. With optimal move ordering, it prunes the majority of irrelevant branches early in the process, resulting in the lowest number of expanded states. This leads to the fastest performance among the three algorithms, as it can eliminate many branches before evaluating them in detail. With the most pruned branches, this algorithm exhibits the highest efficiency, making it the fastest solution for decision-making in this scenario.

3.3. Our Results:



```
Bot's turn!
Bot moved (Alpha-Beta Ordered) from (3, 0) to (1, 2)

Algorithm Comparison:
+----------------------+-------------------+----------------+------------------+------------------+
| Algorithm            | Best Move         | States Expanded| Pruned Branches  | Time Taken (s)   |
+----------------------+-------------------+----------------+------------------+------------------+
| Minimax              | (3, 0, 1, 2)       | 618            | N/A              | 0.0055           |
| Alpha-Beta           | (3, 0, 1, 2)       | 236            | 102              | 0.0027           |
| Alpha-Beta Ordered   | (3, 0, 1, 2)       | 21             | 8                | 0.0004           |
+----------------------+-------------------+----------------+------------------+------------------+
```

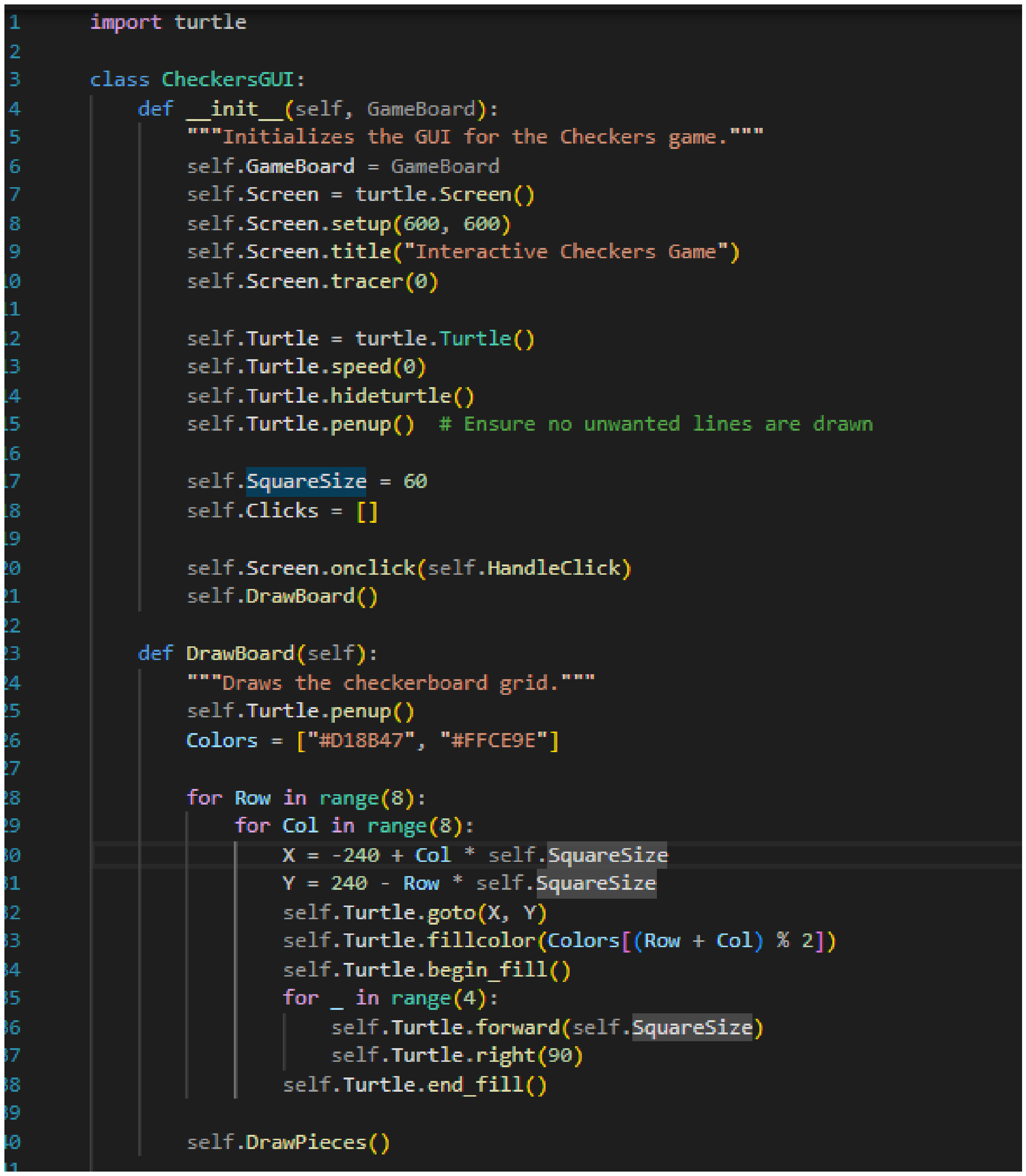


```
Bot's turn!
Bot moved (Alpha-Beta Ordered) from (4, 1) to (2, 3)

Algorithm Comparison:
+----------------------+-------------------+----------------+------------------+------------------+
| Algorithm            | Best Move         | States Expanded| Pruned Branches  | Time Taken (s)   |
+----------------------+-------------------+----------------+------------------+------------------+
| Minimax              | (4, 1, 2, 3)       | 2121           | N/A              | 0.0185           |
| Alpha-Beta           | (4, 1, 2, 3)       | 505            | 190              | 0.0071           |
| Alpha-Beta Ordered   | (4, 1, 2, 3)       | 83             | 36               | 0.0017           |
+----------------------+-------------------+----------------+------------------+------------------+

Your turn!
```

The comparison of the three algorithms: Minimax, Alpha-Beta, and Alpha-Beta Ordered, demonstrates significant differences in efficiency and performance. The **Minimax** algorithm, though it selected the same best move as the others, in one of the moves, it expanded a high number of states (4764) and did not utilize pruning, leading to a longer computation time of 0.0422 seconds. In contrast, **Alpha-Beta** reduced the number of states expanded to 496, utilizing pruning to eliminate unnecessary branches, resulting in a much faster time of 0.0066 seconds. Finally, the **Alpha-Beta Ordered** algorithm, which also used pruning but included move ordering, expanded slightly fewer states (47) and a time of 0.0022 seconds. While the best move was consistent across all algorithms, Alpha-Beta and Alpha-Beta Ordered achieved notable improvements in efficiency and speed.

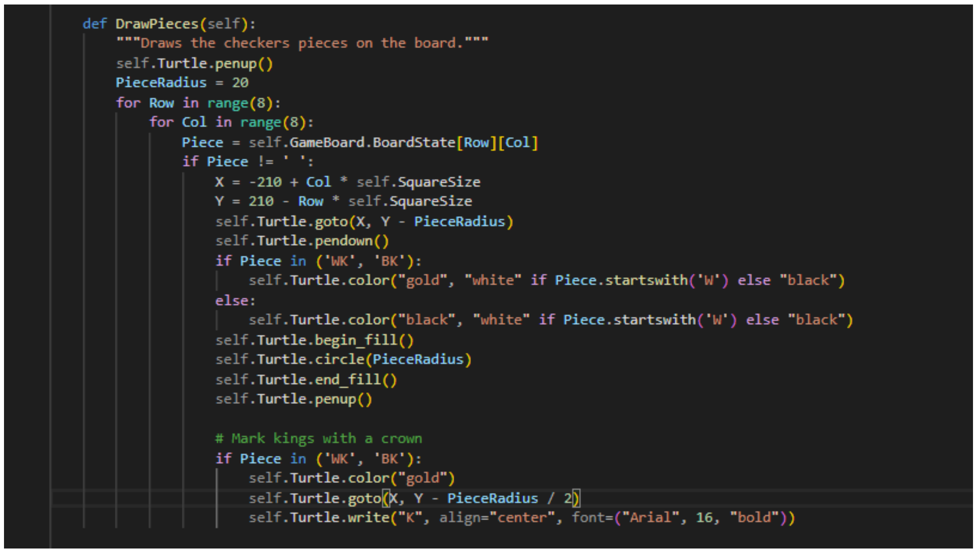
4. Bonus:

Bonus demo link: https://youtu.be/FCTBNwTOaRE



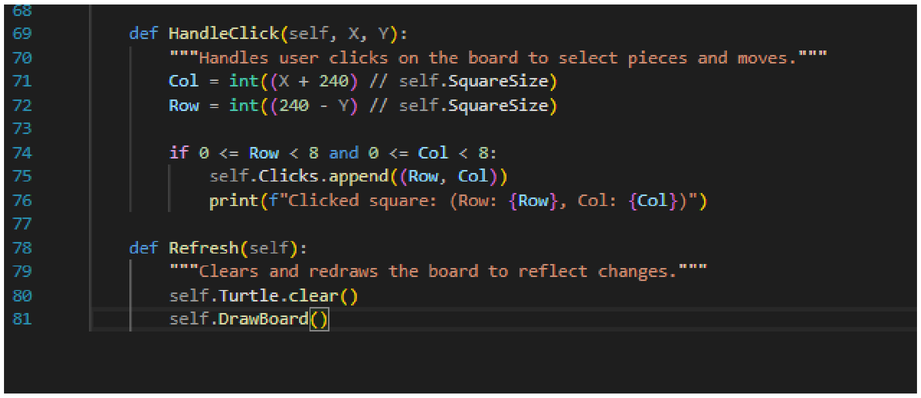
```python
import turtle

class CheckersGUI:
    def __init__(self, GameBoard):
        """Initializes the GUI for the Checkers game."""
        self.GameBoard = GameBoard
        self.Screen = turtle.Screen()
        self.Screen.setup(600, 600)
        self.Screen.title("Interactive Checkers Game")
        self.Screen.tracer(0)

        self.Turtle = turtle.Turtle()
        self.Turtle.speed(0)
        self.Turtle.hideturtle()
        self.Turtle.penup()  # Ensure no unwanted lines are drawn

        self.SquareSize = 60
        self.Clicks = []

        self.Screen.onclick(self.HandleClick)
        self.DrawBoard()

    def DrawBoard(self):
        """Draws the checkerboard grid."""
        self.Turtle.penup()
        Colors = ["#D18B47", "#FFCE9E"]

        for Row in range(8):
            for Col in range(8):
                X = -240 + Col * self.SquareSize
                Y = 240 - Row * self.SquareSize
                self.Turtle.goto(X, Y)
                self.Turtle.fillcolor(Colors[(Row + Col) % 2])
                self.Turtle.begin_fill()
                for _ in range(4):
                    self.Turtle.forward(self.SquareSize)
                    self.Turtle.right(90)
                self.Turtle.end_fill()

        self.DrawPieces()
```

The CheckersGUI class is responsible for creating an interactive graphical interface for the Checkers game using the Turtle graphics library. It initializes the game window with a 600x600 pixel screen and sets up the checkerboard grid with alternating colors to resemble a traditional Checkers board. The board is drawn using the DrawBoard method, which iterates through an 8x8 grid and fills each square with the appropriate color. Additionally, the GUI includes functionality to handle user interactions through mouse clicks, which are registered using the Screen.onclick method. The class ensures smooth rendering by disabling automatic updates with tracer(0), and a dedicated turtle object is used to draw the board and game pieces efficiently. This class provides the foundation for visualizing and interacting with the Checkers game.



```python
def DrawPieces(self):
    """Draws the checkers pieces on the board."""
    self.Turtle.penup()
    PieceRadius = 20
    for Row in range(8):
        for Col in range(8):
            Piece = self.GameBoard.BoardState[Row][Col]
            if Piece != ' ':
                X = -210 + Col * self.SquareSize
                Y = 210 - Row * self.SquareSize
                self.Turtle.goto(X, Y - PieceRadius)
                self.Turtle.pendown()
                if Piece in ('WK', 'BK'):
                    self.Turtle.color("gold", "white" if Piece.startswith('W') else "black")
                else:
                    self.Turtle.color("black", "white" if Piece.startswith('W') else "black")
                self.Turtle.begin_fill()
                self.Turtle.circle(PieceRadius)
                self.Turtle.end_fill()
                self.Turtle.penup()

                # Mark kings with a crown
                if Piece in ('WK', 'BK'):
                    self.Turtle.color("gold")
                    self.Turtle.goto(X, Y - PieceRadius / 2)
                    self.Turtle.write("K", align="center", font=("Arial", 16, "bold"))
```

The DrawPieces method in the CheckersGUI class is responsible for rendering the checkers pieces on the board based on the current game state. It iterates through the BoardState matrix from the GameBoard class, checking each position for a piece. If a piece is found, it calculates the correct position on the screen and draws a circular representation of the piece using Turtle graphics. White pieces are filled with white, and black pieces are filled with black, while kings (denoted by 'WK' or 'BK') have an additional gold outline and a "K" label to indicate their status. This method ensures that the visual representation of the board accurately reflects the current game state, updating piece positions dynamically.



```python
    def HandleClick(self, X, Y):
        """Handles user clicks on the board to select pieces and moves."""
        Col = int((X + 240) // self.SquareSize)
        Row = int((240 - Y) // self.SquareSize)

        if 0 <= Row < 8 and 0 <= Col < 8:
            self.Clicks.append((Row, Col))
            print(f"Clicked square: (Row: {Row}, Col: {Col})")

    def Refresh(self):
        """Clears and redraws the board to reflect changes."""
        self.Turtle.clear()
        self.DrawBoard()
```

The HandleClick method processes user clicks on the board, determining which square was selected based on the X and Y coordinates of the click. It converts these coordinates into row and column indices within the 8x8 board and stores the selected square in the Clicks list while printing the selection for debugging purposes. The Refresh method ensures the board remains visually updated by clearing the Turtle graphics and redrawing the board. This allows the interface to reflect any game state changes, such as piece movements, by refreshing the board dynamically.

5. Conclusion:

In this project, we successfully implemented an AI-powered Checkers bot using Minimax and Alpha-Beta pruning techniques, with further enhancements such as Alpha-Beta pruning with node ordering. The project provided a deep understanding of search algorithms and pruning techniques, as well as their application in a real-time, strategic game environment.

Each team member contributed to different parts of the project, from the game setup and Minimax algorithm to the more advanced Alpha-Beta pruning techniques, including node ordering and its impact on performance. The comparison between the different algorithms demonstrated the significance of optimization techniques like node ordering, which led to more efficient pruning and faster decision-making.

Through this project, we gained hands-on experience in AI decision-making, algorithm analysis, and coding practices, all of which have contributed to our understanding of computational problem-solving. The project not only demonstrated the practical application of AI algorithms but also allowed us to develop a structured and collaborative approach to problem-solving in a team setting.