Go Game with AI and GUI Using Python

Team Members :

1- Ahmed Gamal Mohammed	20220015
2- Abdelrahman Ali Abas	20220261
3- Abdelrahman Waleed Alm-Eldeen	20220271
4- Mohammed Ragab	20220399
5- Mohammed Hassannin	20220390
6- Mohamed Ahmed Abdalla	20220376

- **Project Idea:**

  Develop a functional Python-based desktop Go game with AI and a user-friendly GUI.

- **Main functionalities:**

  1) provide an interactive gaming experience
  2) customize the interactive board sizes (9x9 , 13x13 , 19x19)
  3) determine AI difficulty levels
  4) Real-time Score updates
  5) Save and Load game

- **Academic Context :**

  combines :

  1) Game Development
  2) AI Optimization
  3) Python Libraries

- **Similar Applications :**

  1) KGS Go Server : Online Platform for Go Players
  2) SmartGO : AI-based Go game for IOS and Android

- **Literature Review:**

1) “ The Elements of Go Strategy ” (by Bozulich and Davies) :
   Offers insights into strategic approaches in Go.

2) “ AlphaGo Zero : Mastering the game of Go ” (by Silver etal) :
   Highlights AI advancements in playing Go with deep reinforcement learning.

3) “ Minimax Algorithm with Alpha-Beta Pruning ” (AI-focused papers) :
   Discusses optimization techniques for decision-making algorithms.

4) “ Tkinter GUI Development ” (official Python documentation) :
   Explains best practices for building user-friendly interfaces in Python.

5) “ Heuristic Evaluation in AI ” (Practical guide to heuristics) :
   Guides the design of evaluation functions for AI decision making.

These resources guided the design of this game

- **Applied Algorithms:**

1) Minimax Algorithms :

A) evaluates all possible moves

B) chooses the move with highest score for the player

2) Alpha-Beta Pruning :

A) Optimizes Minimax by ignoring branches that don’t affect the outcome.

B) Improves computational efficiency.

C) reduces computation time

3) Heuristic Evaluation :

A) Scores board states based on territory, captured stones, and potential moves.

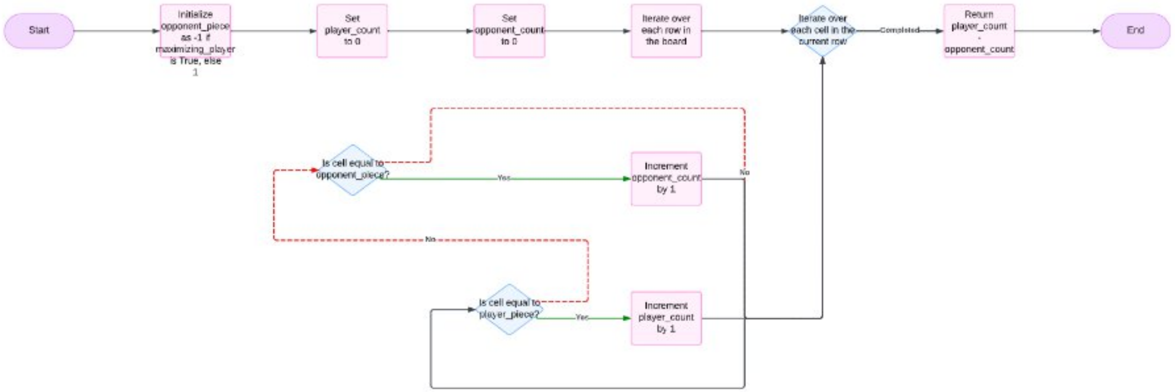
B) ensures The AI selects optimal moves without exhaustive search.

```python
def heuristic_func_1(self, board, maximizing_player):
    """
    First heuristic function: Counts the difference in the number of pieces
    between the current player (maximizing_player) and the opponent.
    maximizing_player = 1 for Black, -1 for White.
    """
    player_piece = maximizing_player
    opponent_piece = -maximizing_player

    player_count = 0
    opponent_count = 0

    for row in board:
        for cell in row:
            if cell == player_piece:
                player_count += 1
            elif cell == opponent_piece:
                opponent_count += 1

    return player_count - opponent_count
```



- **Heuristic Function 1** :
- This function calculates a score by comparing the number of pieces for the current player (maximizing_player) with the opponent.
- It determines how advantageous the board position is for the maximizing_player.

Time Complexity O(m * n). = O (n^2)

- **Outer Loop**:

Iterates over each row in the board
→ O(m) (where m is the number of rows).

- **Inner Loop**:

Iterates over each cell in a row
→ O(n) (where n is the number of columns per row).

- **Total Complexity**:
- The function processes all cells in the board, resulting in **m * n** iterations. = O (n^2)

Space Complexity = O (1)

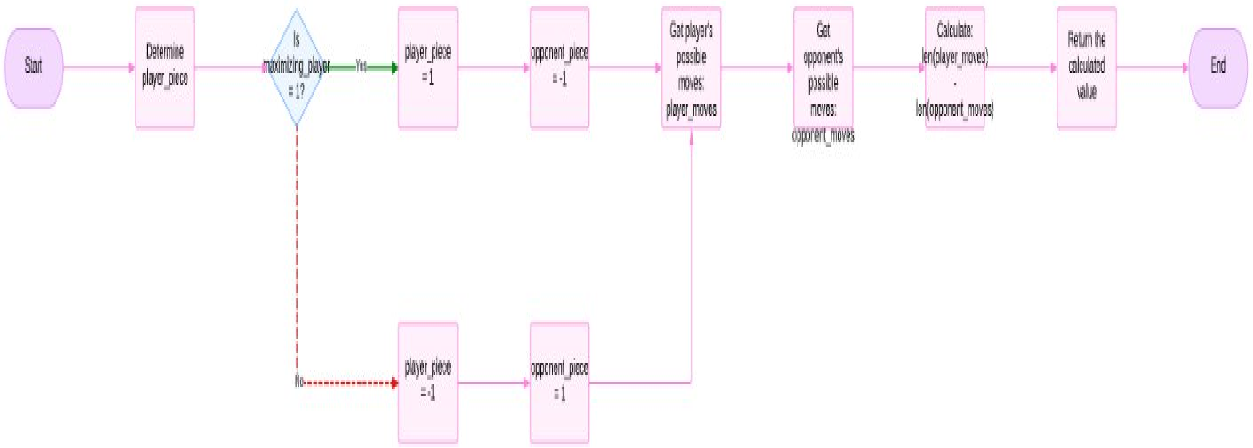
- The function uses a constant amount of memory for:
- Integer variables (player_piece, opponent_piece, player_count, opponent_count).
- It does not allocate extra space dependent on the size of the board.

```python
def heuristic_func_2(self, board, maximizing_player):
    """
    Second heuristic function: Count the number of possible moves for the player.
    (maximizing_player = 1 for the maximizing player, -1 for the minimizing player).
    """

    possible_moves = self.get_possible_moves()

    player_moves = [(x, y) for x, y in possible_moves if board[x, y] == maximizing_player]
    opponent_moves = [(x, y) for x, y in possible_moves if board[x, y] == -maximizing_player]

    return len(player_moves) - len(opponent_moves)
```



- **Heuristic Function 2** :

This function evaluates the board state by calculating the difference in the number of possible moves for the current player (maximizing_player) and their opponent

Time Complexity

- **self.get_possible_moves ()**:

  O(m * n) operations, where m and n are the dimensions.

- **Filtering**

  Checking the condition
  board[x, y] == maximizing_player or -maximizing_player
  is a constant time operation for each move.
  Let the number of possible moves be k (k ≤ m * n).

- **Total Complexity**:

  Assuming k ≈ m * n in the worst case:
  Time Complexity = O (m * n).

Space Complexity = O (k)

- O(k) , where k is the number of possible moves

- ## minimax :

evaluate the optimal move by
simulating all possible outcomes up to a specified depth.

Time Complexity

- **Base Case**

  O(n^2)..

- **Recursive Calls**.

  Each level evaluates O (**b^d**) board states in total.

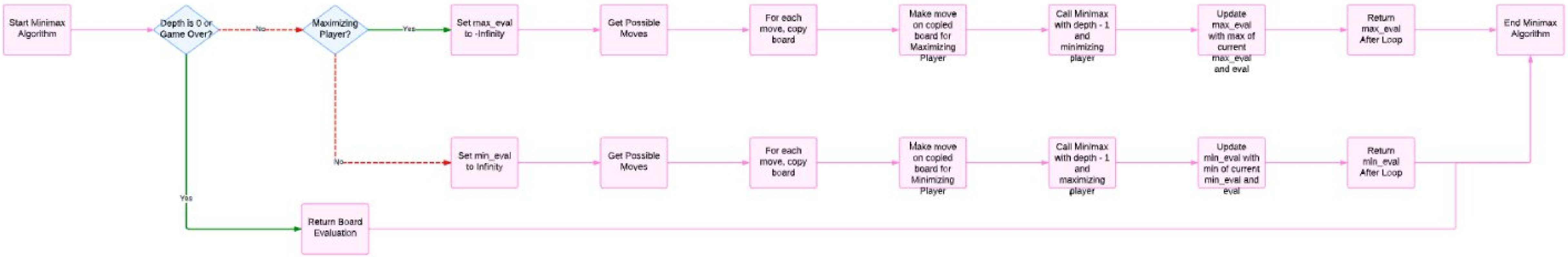
- **Total Complexity**:

  Time Complexity = O (b^d × n^2).

```python
def minimax(self, board, depth, maximizing_player):
    """Basic Minimax algorithm without pruning."""
    if depth == 0 or self.game.is_game_over(board):
        return self.evaluate_board(board)
```

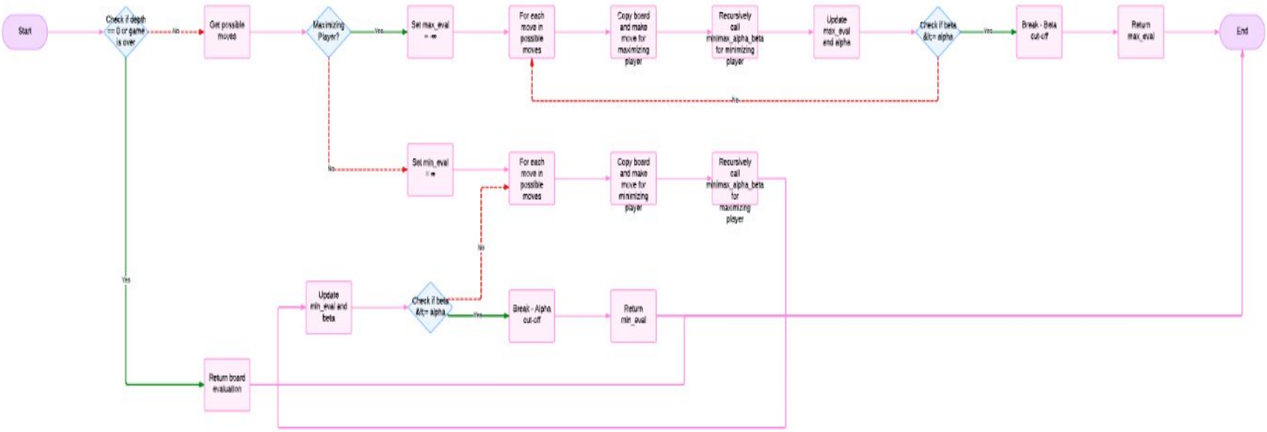
Space Complexity

- Recursive Call Stack : O (n)
- Board Copies : O (b × n^2).
- Total Space Complexity : O(d + b × n^2).



```python
    def minimax_alpha_beta(self, board, depth, alpha, beta, maximizing_player):
        """Minimax algorithm with Alpha-Beta Pruning."""
        if depth == 0 or self.game.is_game_over(board):
            return self.evaluate_board(board)

        possible_moves = self.game.get_possible_moves(board)
        if maximizing_player:
            max_eval = -float('inf')
            for move in possible_moves:
                board_copy = board.copy()
                self.game.make_move(board_copy, move, 1)  # 1 for maximizing player
                eval = self.minimax_alpha_beta(board_copy, depth - 1, alpha, beta, False)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break  # Beta cut-off
            return max_eval
```



- **minimax_alpha_beta :**

Alpha (best value for the maximizing player so far).

Beta (best value for the minimizing player so far).

If **beta <= alpha**, stop further exploration (cut-off).

Time Complexity:

- **Best Case**: **O(b^(d/2) × n)** (effective pruning).
- **Worst Case**: **O(b^d × n)** (no pruning).

Space Complexity: **O(d + b × n)**.

- Depends on recursion depth **d** and memory for board copies

Why Alpha-Beta Pruning is Faster

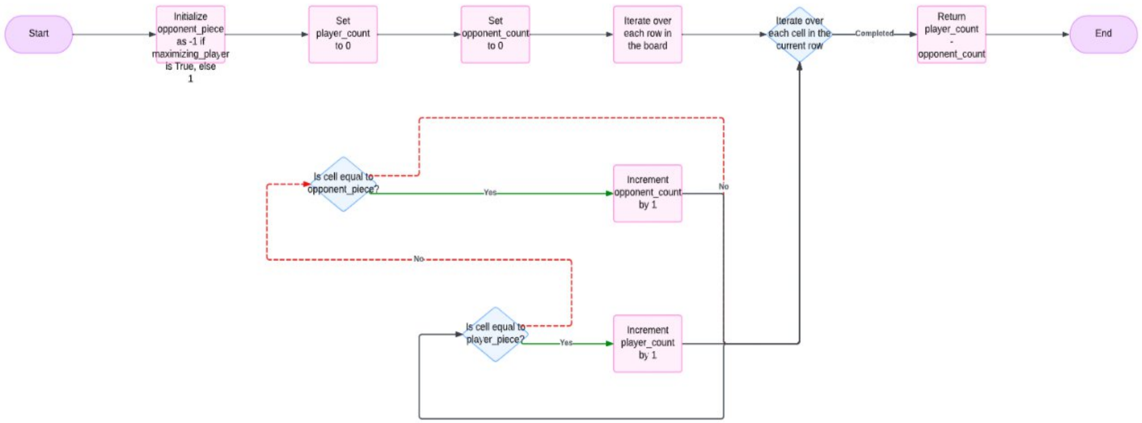
Prunes Redundant Moves: Cuts off branches that cannot affect the final outcome.

Efficient Search: Reduces the effective branching factor, making it faster than standard minimax.

Same Result: Produces the same outcome as minimax but with fewer evaluations.

```python
def minimax_with_heuristic_1_alpha_beta(self, board, depth, alpha, beta, maximizing_player):
    """Minimax with Heuristic 1 and Alpha-Beta Pruning."""
    if depth == 0 or self.is_game_over():
        return self.heuristic_func_1(board, maximizing_player)

    possible_moves = self.get_possible_moves()
    if maximizing_player:
        max_eval = -float('inf')
        for move in possible_moves:
            board_copy = board.copy()
            self.make_move(board_copy, move)
            eval = self.minimax_with_heuristic_1_alpha_beta(board_copy, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval
```



- **minimax_with_Heuristic 1 _alpha_beta :**

  evaluates board states efficiently by using pruning
  to skip unnecessary computations and employs Heuristic 1
  for intermediate evaluations
  when the maximum depth is reached..

Time Complexity:

- **Best Case effictive pruning**: **O(b^(d/2) × n)**.
- **Worst Case** no pruning: **O(b^d × n)**.

Space Complexity: **O(d + b × n)**.

```python
def minimax_with_heuristic_2_alpha_beta(self, board, depth, alpha, beta, maximizing_player):
    """Minimax with Heuristic 2 and Alpha-Beta Pruning."""
    if depth == 0 or self.is_game_over():
        return self.heuristic_func_2(board, maximizing_player)

    possible_moves = self.get_possible_moves()
    if maximizing_player:
        max_eval = -float('inf')
        for move in possible_moves:
            board_copy = board.copy()
            self.make_move(board_copy, move)
            eval = self.minimax_with_heuristic_2_alpha_beta(board_copy, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for move in possible_moves:
            board_copy = board.copy()
            self.make_move(board_copy, move)
            eval = self.minimax_with_heuristic_2_alpha_beta(board_copy, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

```

- **minimax_with_Heuristic 2 _alpha_beta :**

  evaluate the board state instead of heuristic_func_1. Heuristic 2 counts the number of possible moves for the player and opponent, making the search more dynamic. Alpha-Beta Pruning is used to cut off branches that do not need to be explored, improving efficiency.

Time Complexity:

- **Best Case**: **O(b^(d/2) × n)**.
- **Worst Case**: **O(b^d × n)**.

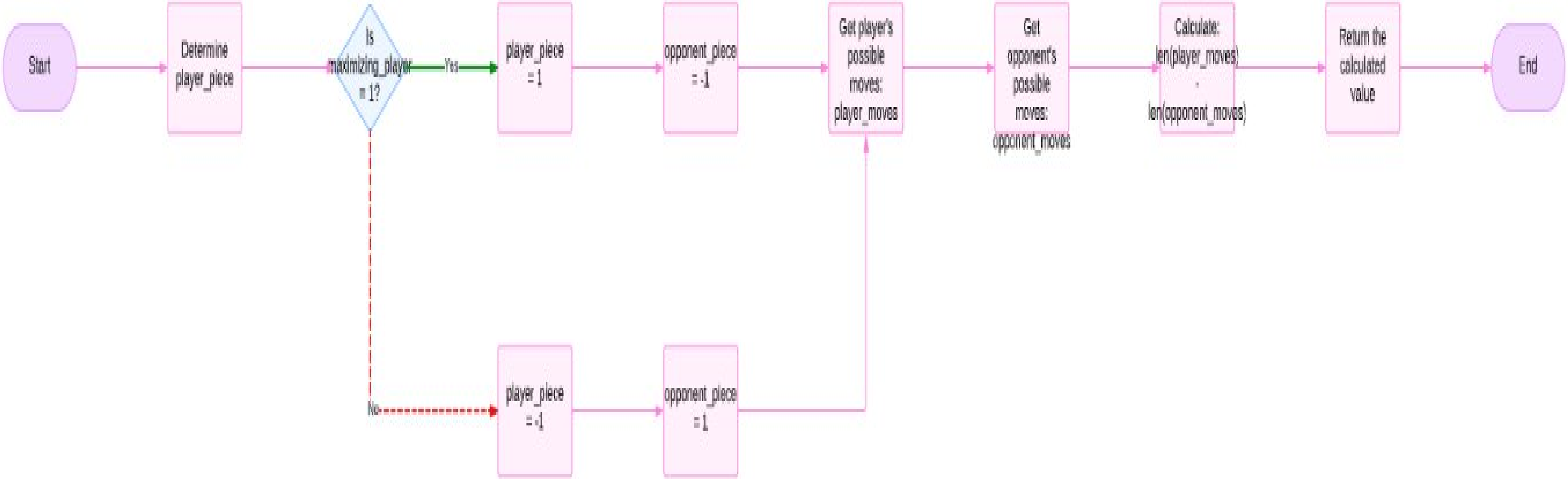
Space Complexity: **O(d + b × n)**.

Pruning Saves Time: Alpha-Beta pruning significantly reduces the number of states explored compared to standard minimax.

Heuristic Reduces Depth: Using `heuristic_func_2` reduces the need to explore every possible outcome, as it gives a useful intermediate evaluation based on the number of possible moves.

Same Result: The algorithm will still find the optimal solution, but it does so more efficiently than regular minimax.

- **minimax_with_Heuristic 2 _alpha_beta :**



- **Board Representation :**

  2D Array implemented using NumPy for Efficient StateTracking.

- **State Tracking** :

  1) Current Board State
  2) Captured Stones (dynamic score updates)

- **State Storage :** Uses JSON Serialization.

- **User Guide** :

  1) install Python and required libraries
  2) run main.py to start the application
  3) use menu options to :
     A) Start a new game
     B) change board size
     c) save or load game states
  4) choose AI difficulty Level and interact with the board by Clicking intersections

- **Developer Guide (Code Structure)**:

  1) game_logic.py : (handles rules , scoring and move validation) .
  2) ai.py : (implements Minimax and Alpha-Beta Pruning).
  3) gui.py : (manages the tkinter interface)
  4) utils.py : includes helper functions for state management).

- **Future Work** :

1) implement monte carlo tree search (MCTS) for improved AI
2) add online multiplayer functionality
3) enhance gui design using advanced frameworks like PyQt or Kivy