Checkers Game Engine: Data Structures and Algorithms

This document provides an overview of the key data structures and algorithms implemented in the Checkers game engine. Understanding the below components is essential to grasp how the game state is managed and how game rules are enforced.

1. Data Structures

The game primarily relies on two custom classes and standard Python collections to represent its state and actions.

1.1 GameState Class

The GameState class is the central data structure that encapsulates the entire state of the game at any given moment.

- **Attributes:**
  - **board**: A 2D list (list of lists) representing the NxN game board. Each element in the inner lists stores a string constant ("_", "WP", "WK", "BP", "BK") indicating the piece occupying that square, or EMPTY if it's vacant. This is a direct representation of the physical board.
  - **current_player**: An integer (0 for WHITE, 1 for BLACK) indicating whose turn it is.
  - **white_pieces_count, black_pieces_count:** Integers tracking the number of active pieces for each player. Used for determining win/loss conditions.
  - **last_move_was_capture:** A boolean flag that is True if the immediately preceding move was a capture. Crucial for enforcing multi-jump rules.
  - **captured_piece_position:** A tuple (row, col) storing the new position of the piece that just made a capture. This is used to identify the piece that must continue a multi-jump sequence.
  - **possible_multi_jump_moves:** A list of Move objects. If last_move_was_capture is True and the captured_piece_position can make further jumps, this list will contain only those forced continuation moves.

1.2 Move Class

The Move class represents a single action a player can take on the board.

- **Attributes:**
  - start_row, start_col: Integers indicating the starting coordinates of the piece.
  - end_row, end_col: Integers indicating the ending coordinates of the piece.

    - is_capture: A boolean flag, True if this move involves capturing an opponent's piece.
    - captured_piece_row, captured_piece_col: Integers storing the coordinates of the piece that was captured (only relevant if is_capture is True).

1.3 Python set for Move Generation

Within the get_valid_moves function, Python's built-in set data structure is used:

- all_potential_moves: A set to temporarily store all possible moves (both captures and non-captures) found during the board scan. Using a set automatically handles duplicate Move objects, ensuring each unique valid move is listed only once.
- capture_moves: A set to specifically store only the capture moves found. This is critical for implementing the mandatory capture rule.

2. Algorithms

The game engine implements several core algorithms to manage gameplay, enforce rules, and handle player interaction.

2.1 Board Initialization (GameState._initialize_board)

- **Algorithm:** Iterates through the NxN board grid. Based on the row and column indices, it places BLACK_PAWN pieces in the top three rows (0-2) and WHITE_PAWN pieces in the bottom three rows (N-3 to N-1), ensuring they are placed only on the "dark" squares (where row + col is odd). All other squares are initialized to EMPTY.

2.3 Finding Captures for a Single Piece (find_captures_for_piece)

- **Algorithm:** Given a piece's position (r, c) and the player it belongs to, this function explicitly checks all four diagonal directions for potential captures.
    - For each direction, it calculates the coordinates of the adjacent_piece and the landing_square.
    - It verifies that both these squares are within board boundaries.
    - It checks if the adjacent_piece is an opponent's piece and the landing_square is EMPTY.
    - It applies pawn-specific rules: white pawns can only capture "up" (decreasing row index), and black pawns only "down" (increasing row index). Kings can capture in any diagonal direction.
    - If all conditions are met, a Move object representing the capture is created and added to a list.

2.4 Generating Valid Moves (get_valid_moves)

- **Algorithm:** This is a comprehensive function that determines all valid moves for the current_player, adhering to Checkers' complex rules.
    1. **Multi-Jump Check:** First, it checks game_state.last_move_was_capture and game_state.possible_multi_jump_moves. If a multi-jump sequence is in progress, it immediately returns *only* those pre-calculated forced moves.
    2. **Board Scan for Captures:** If no multi-jump is forced, it iterates through every square on the board. For each piece belonging to the current_player, it calls find_captures_for_piece to identify all potential captures for that specific piece. All found captures are added to a capture_moves set.
    3. **Board Scan for Non-Captures:** If a piece *cannot* make a capture, it then checks for simple (non-capturing) moves for that piece. These are added to an all_potential_moves set.
    4. **Mandatory Capture Enforcement:** After scanning the entire board:
        - If the capture_moves set is not empty (meaning at least one capture is available anywhere on the board), the function returns *only* the moves from the capture_moves set.
        - If capture_moves is empty (no captures are available), the function returns all moves from the all_potential_moves set (which at this point only contains non-capture moves).
    5. **Existence Check Optimization:** The function includes a check_only_existence parameter. If True, it returns True as soon as it finds any valid move (a forced multi-jump, a capture, or a non-capture if no captures are available), optimizing checks where only existence matters (like is_game_over).

2.5 Applying a Move (apply_move)

- **Algorithm:** Takes a GameState and a Move object, and returns a *new* GameState reflecting the board after the move.
    1. Creates a deep_copy of the input GameState.
    2. Moves the piece from start_row, start_col to end_row, end_col on the new board.
    3. If move.is_capture is True, it removes the captured_piece from the board and decrements the opponent's piece count. It also sets new_state.last_move_was_capture to True and updates new_state.captured_piece_position.
    4. Checks for **King Promotion**: If a pawn reaches the opponent's back rank, its type is changed to a King.
    5. **Multi-Jump Continuation Check:** If the move was a capture, it immediately calls find_captures_for_piece for the *capturing piece from its new position*.
        - If further jumps are found, new_state.possible_multi_jump_moves is populated, and new_state.current_player remains the same (forcing the multi-jump).

        - If no further jumps, new_state.possible_multi_jump_moves is cleared, and the new_state.current_player is toggled to the opponent.
    6. If the move was not a capture, new_state.possible_multi_jump_moves is cleared, and the new_state.current_player is simply toggled to the opponent.

2.6 Validating Player Input (is_valid_move)

- **Algorithm:** Parses a human-readable move string (e.g., "F0->E1") into a Move object and checks its validity.
    1. Parses the input string to extract start and end row/column coordinates. It uses helper functions (char_to_row) to convert letter-based row inputs.
    2. Validates that the parsed coordinates are within the board boundaries.
    3. Creates a Move object from the parsed input.
    4. Calls get_valid_moves (with check_only_existence=False to get the full list) for the current_player.
    5. Iterates through the list of truly valid moves and compares the parsed_move using the Move class's __eq__ method.
    6. Returns the validated Move object if it's found in the valid list, otherwise returns None.

2.7 Game Over Check (is_game_over)

- **Algorithm:** Determines if the game has reached a terminal state.
    1. Checks if either player's pieces_count has reached zero (meaning the other player wins).
    2. Calls get_valid_moves with check_only_existence=True for the current_player. If this returns False (meaning the current_player has no valid moves), the game is over, and the current_player loses.

2.8 Random Computer Move Selection (get_random_computer_move)

- **Algorithm:** Implements the computer's basic intelligence.
    1. Calls get_valid_moves (with check_only_existence=False to get the full list) for the computer's player.
    2. If the list of valid moves is not empty, it uses random.choice() to select one move at random from the list.
    3. Returns the randomly chosen Move object, or None if no valid moves are available.

The time complexity is $O(N^2)$ for each move as we iterating over the board multiple times to find valid moves and printing the board. It is okay for a board of size 8*8 but if the board size increases then it’s not optimal.

3. Algorithms for Future Enhancements

To expand the capabilities and intelligence of the Checkers game, several more advanced algorithms could be integrated:

3.1 Minimax Algorithm

- **Purpose:** To enable the AI to make optimal decisions by exploring possible game states to a certain depth. It's a recursive algorithm that evaluates all possible moves for the current player, assuming the opponent will also play optimally.
- **How it works:** It builds a "game tree" where nodes represent game states and edges represent moves. It then assigns scores to the leaf nodes (terminal states or states at the maximum search depth) using an evaluation function. These scores are propagated up the tree, with the maximizing player choosing the move that leads to the highest score, and the minimizing player choosing the move that leads to the lowest score.

3.2 Alpha-Beta Pruning

- **Purpose:** An optimization technique for the Minimax algorithm. It significantly reduces the number of nodes that need to be evaluated in the game tree without affecting the final decision.
- **How it works:** It maintains two values, alpha (the best score the maximizing player can guarantee so far) and beta (the best score the minimizing player can guarantee so far). If, at any point, a branch's value falls outside the current alpha-beta window, that branch can be "pruned" (skipped) because it cannot possibly lead to a better outcome for the current player than what's already known.

3.3 Evaluation Function

- **Purpose:** To assign a numerical score to a non-terminal game state (a leaf node in the Minimax search tree that isn't the end of the game). This score estimates how favorable the position is for the AI.
- **How it works:** It considers various factors such as:
    - **Material Advantage:** The number and type (pawn vs. king) of pieces each player has.
    - **Positional Advantage:** Control of the center, pieces closer to promotion, pieces on the back row (for safety), and connected pawn structures.
    - **Mobility:** The number of valid moves a player has.
    - **Threats and Defenses:** Pieces under attack or defended.

  The function returns a weighted sum of these factors.

3.4 Draw Detection Algorithms

- **Purpose:** To correctly identify draw conditions beyond just a lack of valid moves.
- **How it works:**
    - **Threefold Repetition:** Requires storing a history of board hashes and checking if any position has occurred three times.
    - **50-Move Rule (or 40-move rule in Checkers):** Requires tracking the number of moves since the last capture or pawn promotion. If this count exceeds a certain threshold, the game is a draw.

This overview provides a roadmap for enhancing the Checkers AI and game logic with more sophisticated algorithms.

4. Future Optimizations

While the current 2D list for the `board` is intuitive and sufficient for an 8x8 Checkers game, more advanced data structures can offer significant performance improvements, especially for larger board sizes or highly optimized AI engines.

4.1 Bitboards

- **Description:** Instead of a 2D array, the board state is represented by one or more large integers (e.g., 64-bit integers for an 8x8 board). Each bit in the integer corresponds to a specific square on the board.
- **Advantages:**
    - **Extreme Speed for Operations:** Bitwise operations (AND, OR, XOR, shifts) are incredibly fast at the CPU level.
    - **Memory Efficiency:** Very compact storage.
    - **Parallelism:** Bitwise operations can conceptually operate on multiple squares simultaneously.
- Time Complexity reduces to O(N) where N is number of pieces on the board.

4.2 Sparse Board Representation (e.g., Hash Map/Dictionary)

- **Description:** Instead of storing every square in a 2D array, only the squares that are *occupied* by a piece are stored. This can be implemented using a hash map (dictionary in Python) where keys are square coordinates (e.g., `(row, col)` tuples) and values are the piece types.
- **Advantages for Complexity:**

    - **Memory Efficiency for Sparse Boards:** Can save significant memory in endgames where most of the board is empty and only a few pieces remain.
- Time complexity reduces to O(N) where N is the number of pieces on the board.