CS2109S Mini Project Report

Name: Foo Dun Liang
Student ID: A0236443H

Description

I have created two extra classes which are the State class and the AlphaBetaMinimx class.
The State class is created to represent every state where different actions are taken to reach each state.
The **State** class includes variables:

- *State.turn(boolean)* represents current turn, if is black turn return True, else False.
- *State.from_(1D list with length of 2)* represents the previous position of the pawns
- *State.to_(1D list with length of 2)* represents the current position of the pawns
- *State.board(2D list)* represents the board configuration of every state.

The **AlphaBetaMinimax** class is the class that contains all the searching algorithms (i.e iterative deepening search, alpha-beta minimax) and the helper functions that are required.

- `Alpha_beta_iterative_deepening(State state)` returns the best move which is a list containing *State.from_* and *State.to_* after repeat searching and recursively executing alpha-beta pruning in every iteration until it reaches the MAX_SEARCH_DEPTH (a constant value defined by users).

- Within the `Alpha_beta_iterative_deepening(State state)` function, there are other nested functions created inside it, which are `alpha_beta_pruning(State state, int alpha, int beta, int depth)`, `max_value(State state, int alpha, int beta, int depth)` and `min_value(State state, int alpha, int beta, int depth)`. Alpha_beta_pruning function will return the max value or min value depending on the players' turn, if the player moves black pawns it will return the max value, else return the min value. The depth in the function indicates which level it is searching on at the moment, if the depth equals 0 means that it has reached the lowest node(State) of the game tree and it will return the value of the leaf nodes by running the evaluation function. This value will then be compared with other leaf nodes' values within the iterative deepening search function, it will cut off some branches in the game tree which need not be explored if there already exists a state with a higher value when following the searching order.

- `get_next_states(State state)` returns a list that includes all the possible next states that can be achieved by all the valid one-step movements. For every pawn, I will try three directions (diagonally left, in front, and diagonally right) and check the validity of each movement. Those valid moves will be applied to a duplicate board and used all the values required to create a new object of the State class. This new object will then be added to the possible next states list that the function is going to return.

Evaluation function

Within the evaluation function, I have SIX FACTOR VALUES that are taken into consideration, which are

- *Num_of_pawns:*
    - The number of pawns left on the board after executing the possible move. Since there is a possible situation where the player's pawns are all captured by the opponent. Therefore, I will take the number of pawns left on the board of each state into account to ensure this kind of situation is very less likely to happen.
- *Attack_val:*
    - The total number of opponents' pawns that can be captured within one-step moves. By attacking the opponent's pawns (capturing the opponent's pawn) in the next step can lead us into a more favorable situation because losing all pieces is also considered a loss inside the game.
- *Protection_val:*
    - The number of pawns that are able to capture the opponent's pawns after the opponent's turn if the opponent chooses to capture our pawns. When there exists a pawn where its front diagonal left or front diagonal right is the opponent's pawn and back diagonal left or back diagonal right is the friendly's pawn, it can be counted into the protection value.
- *Danger_val:*
    - A pawn is considered dangerous when it is near the winning row. In this case, I will define pawns that are two steps distance away from the winning line as dangerous pawns. It is because in the best-case scenario if there is no defender protecting the last row, a piece can reach the winning state in two steps.
- *High_danger_val*:
    - The number of pawns that are on the positions which are only one row away from reaching the winning row, which is basically those able to reach the winning row in one step distance unless the opponent has pieces that can capture my "high danger" pawns.
- *last_row_defense*:
    - The pawns in the last row play a pivotal role in defending the opponent and preventing them from winning the game.

Apart from that, I have also defined SIX CONSTANTS which represent the weightage of the values according to their priorities.

CONSTANT TABLE

1. *LAST_ROW_DEFEND_CONSTANT* = 1
2. *PROTECTION_CONSTANT* = 2
3. *DANGER_CONSTANT* = 3
4. *SIZE_CONSTANT* = 4
5. *ATTACK_CONSTANT* = 5
6. *HIGH_DANGER_CONSTANT* = 6

Documentation

State class

1. *is_black() -> boolean:*
   - Check the player's turn. If it is black turn return True, else return False.
2. *is_white() -> boolean:*
   - Check the player's turn. If it is white turn return True, else return False.
3. *get_src_and_dst() -> 2D list:*
   - Get the previous piece's position as the first element inside the list, the second element will be the destination of the same piece moved to.
4. *get_black_position() -> 2D list:*
   - Get all the black pawns' positions in current board configuration. Returning a 2D list which length of the list will be equal to the number of black pawns left.
5. *get_white_position() -> 2D list:*
   - Get all the white pawns' positions in current board configuration. Returning a 2D list which length of the list will be equal to the number of white pawns left.

AlphaBetaMinimax class

1. *alpha_beta_iterative_deeepeening (State state) -> 2D list:*
   - Return the list that contains the best move (the move that has the highest/ lowest evaluation value)
2. *max_value(State state, int alpha, int beta, int depth): -> int*
   - Return the maximum value of current state.
3. *min_value(State state, int alpha, int beta, int depth): -> int*
   - Return the minimum value of current state.
4. *alpha_beta_pruning(State state, int alpha, int beta, int depth):*
   - Check if any branches can be cut off without exploring them.
5. *evaluation_func(State state) -> int:*
   - Return the evaluation value.
6. *get_protection_and_attack_val(boolean is_black_turn, int[][] black_pawn_list, int[][] white_pawn_list) -> (int, int):*
   - Return the number of pawns that are able to capture the opponent's piece and the number of pawns that are able to defend against the opponent's pawns.
7. *get_danger_val(boolean is_black_turn, int[][] black_pawn_list, int[][] white_pawn_list) -> int:*
   - Return the number of pawns that are two rows away from the winning state.
8. *get_danger_val(boolean is_black_turn, int[][] black_pawn_list, int[][] white_pawn_list) -> int:*
   - Return the number of pawns that are one row away from the winning state.
9. *get_last_row_defense(boolean is_black_turn, int[][] black_pawn_list, int[][] white_pawn_list) -> int:*
   - Return the number of pawns that are at the first row(black => row 0; white => row 5)