Sp-14 Red Chess Ai

By: Aaron Dailey ,Nicholas Kennel , Haige Zhu

Objective

- The project is an AI chess game where the user will play against a CPU chess player.

Rules

- Basic chess rules
- Chess pieces move and attack in their established way Ex: Pawn moves one square forward and attack diagonally in front of the pawn
- Chess games are won with checkmate
- Special rules : promotion, En Passant, and castling (Kingside and Queenside)
- different conditions that can cause Stalemate: King can't move, only Kings left, not enough pieces, etc.

Platform

- WinForms
- Originally planned to use Unity.
- Switched to using WinForms due to multiple reasons:
  - Game was already fully set up in WinForms
  - Converting to Unity would not provide any clear benefits
  - Easier to add to a complete project
  - Impressive UI

User interface

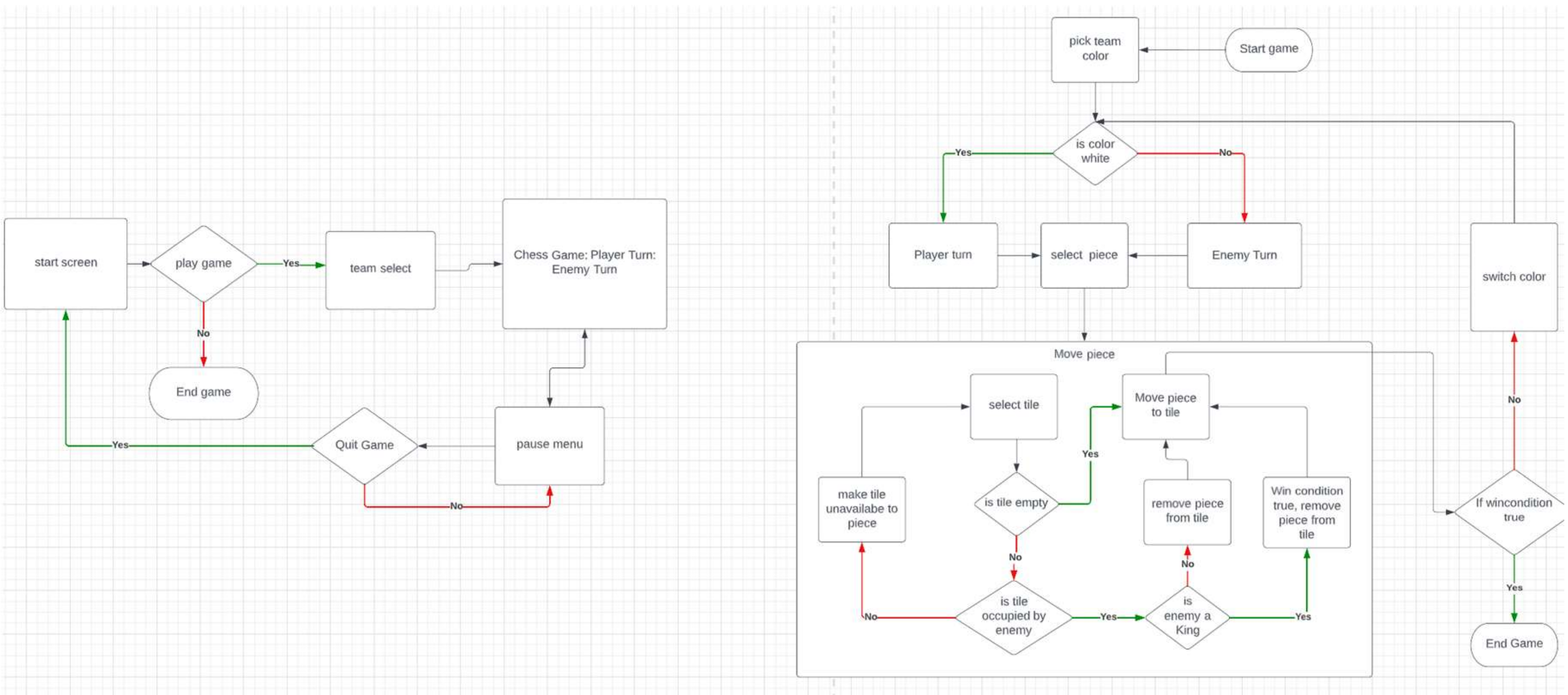


Diagram of program structure

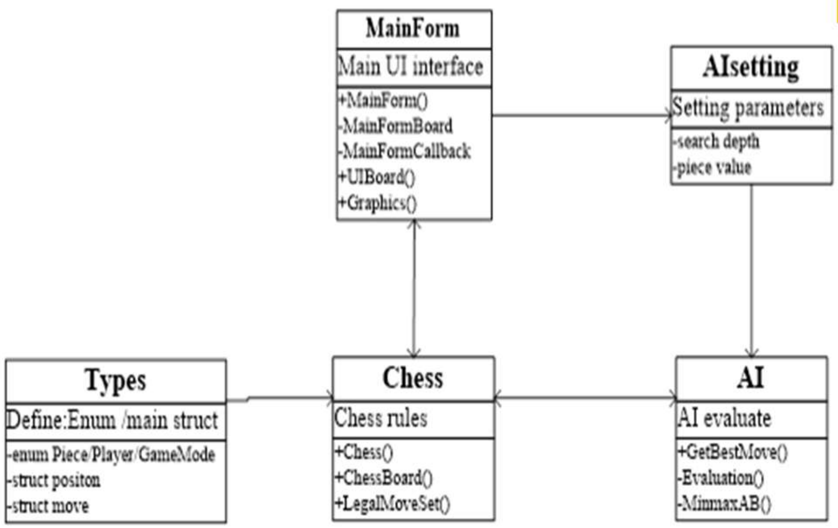
Chess is a class that controls the logical operation of the game and forms the core of the software. It has two independent auxiliary classes: ChessBoard and LegalMoveSet. ChessBoard mainly records the current state of the chessboard in various forms. LegalMoveSets is a chess rule class used to determine whether a move is legal and whether the game is over. It can also form the action set of each chess piece under the current state.

MainForm is the form that runs the game and serves as the UI. It has two shadow forms, MainFormBoard and MainFormCallbacks, to ensure smoother interface operation.

AI is an AI that evaluates values and uses a minimax search with alpha-beta pruning to make the best move decisions.

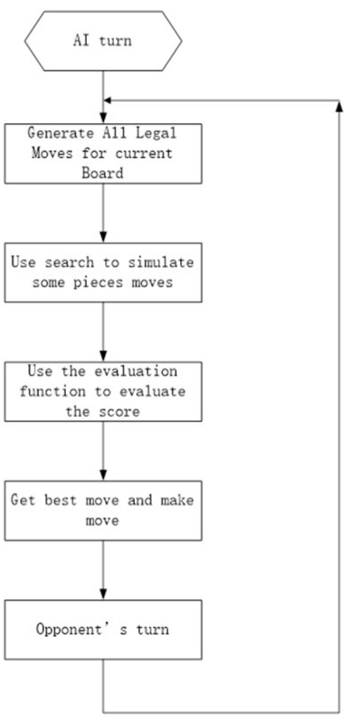
Types define various enumeration values and structures required by the system, such as chess piece enumeration, player enumeration, game mode enumeration, chess piece position structure, and chess piece movement model structure.

AIsetting allows users to set the game difficulty (search depth) and chess piece strength values through the FrmSetting form interface to affect the AI evaluation results.



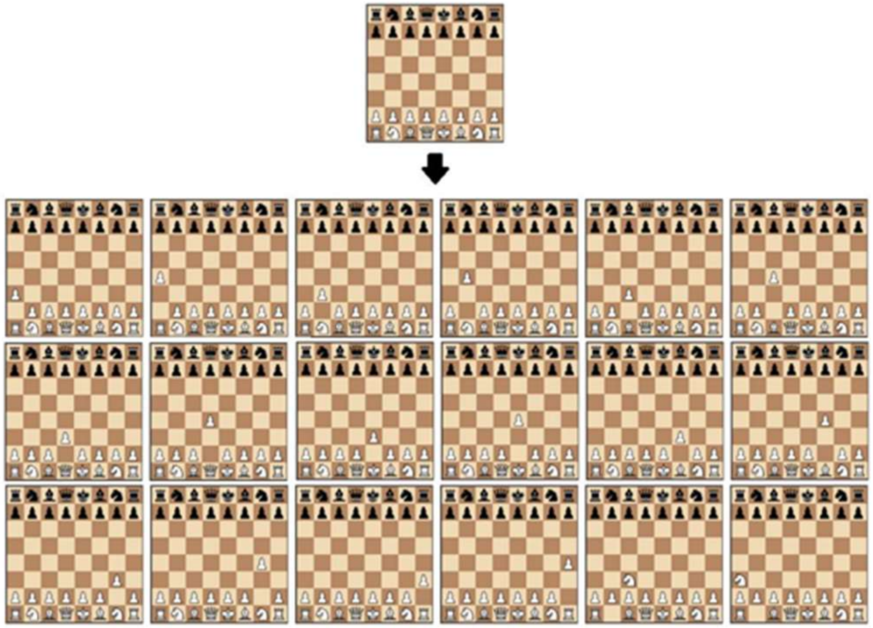
AI Workflow diagram

- By utilizing the LegalMoveSet.getPlayerMoves function to obtain a dictionary of all legal moves for our side, the program conducts a search based on evaluation to ultimately determine the optimal move for our side. The search and evaluation functions work together to enhance the AI's "thinking" ability from two aspects: :
- Evaluation function: directly impacts the AI's choices by evaluating the quality of the game state.
- Search method: accelerates the search speed and reduces the time required for deep searches, providing the potential to deepen the search depth and improve the AI's long-term and comprehensive "thinking" abilities.



STEP1: Iterate to get all possible walking methods.

- In this step, the LegalMoveSet.getPlayerMoves library will be used to iterate through moves and calculate all possible legal moves for a specific chess position.

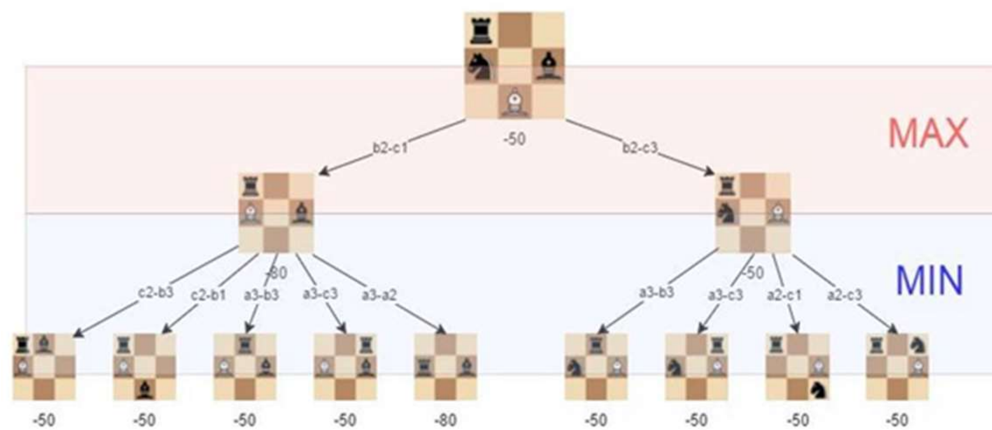


STEP2: Simulated moves and position evaluation

- This section aims to determine which side has the advantage in a given chess position. The simplest method is to calculate the relative piece value of each chess piece on the board using a table. Through the evaluation function, we can obtain the move that maximizes the calculated evaluation value. However, in this case, all possible moves need to be calculated, which requires high computational power and is time-consuming

STEP 3 Search tree using the minimax method

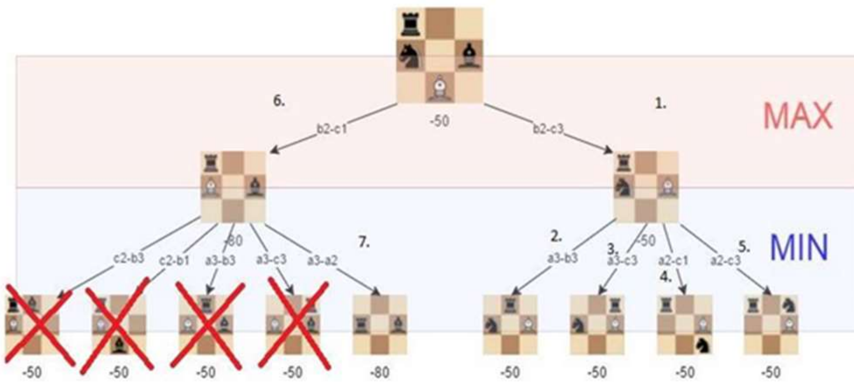
The algorithm constructs a search tree to choose the best move, using the minimax algorithm, which is widely used in decision theory, game theory, and statistics. Its basic principle is to minimize the loss under the worst possible scenario (i.e., the maximum possible loss). This algorithm searches all possible paths of the recursive tree based on a given depth and gives an evaluation at the "leaf" node. After that, the algorithm returns the minimum and maximum values from the child nodes to the parent node depending on whether it is the turn of the white or black player.



As shown in the figure, the best move for white (our side) is b2-c3, as we can minimize the loss to -50.

STEP4 Alpha-beta pruning

Alpha-beta pruning is an optimization technique for the Minimax algorithm, which involves cutting off certain branches during the search process. This can save computational resources and increase the depth of the algorithm. The pruning algorithm abandons a branch if it discovers that the branch will lead to a worse outcome. This method does not affect the Minimax algorithm but can speed up the algorithm. Of course, if the optimal path can be found at the beginning, the alpha-beta algorithm will be more effective.



STEP5 Improving the evaluation function

To optimize the function, we added a factor that evaluates the position of the chess pieces. For example, a knight in the middle of the board is better than one in the corner (because it has more moves and is more active).

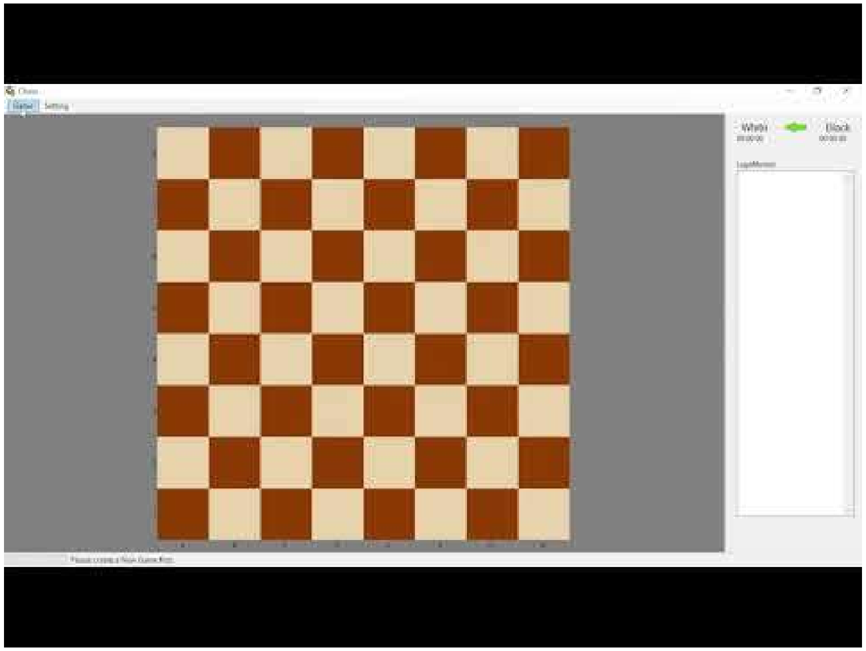
AI algorithm pseudocode

```
//"This function parameter is passed by value."
function MINIMAX(state, cur_position, alpha, beta){
    //"The search tree has reached the game-ending state."
    if (TERMINATE_TEST(state)){
        //The game has ended, return the final state
        return UTILITY(state);
    }

    //Our turn, maximize our own benefit.
    if (MAXIMIZING_PLAYER){
        //Search for the child node with the maximum value to achieve maximum benefit.
        max_eval = -INFINITY;
        for each child of cur_position{
            eval = MINIMAX(state, child, alpha, beta);
            max_eval = MINIMAX(max_eval, eval);

            //MAX turn, alpha records the maximum value of child nodes.
            alpha = MINIMAX(alpha, eval);
            if(beta <= alpha){
                break;
            }
        }
        return max_eval;
    }

    //Opponent's turn, minimize the opponent's benefit.
    if (MINIMIZING_PLAYER){
        //Search for the child node with the minimum value to achieve minimizing the opponent
        min_eval = INFINITY;
        for each child of cur_position{
            eval = MINIMAX(state, child, alpha, beta);
            min_eval = min(min_eval, eval);

            //MIN turn, beta records the minimum value of child nodes.
            beta = MINIMAX(beta, eval);
            if(beta <= alpha){
                break;
            }
        }
        return min_eval;
    }
}
```

AI Difficulty levels

The difficulty levels are categorized as Beginner, Easy, Medium, Hard, and Very Hard, and they correspond to the depth of analysis performed by the AI. Beginner, Easy, Medium, Hard, and Very Hard correspond to analysis depths of 1 to 5 levels, respectively. When the analysis depth is set to 3, the AI's response time is about 5-6 seconds. With an analysis depth of 4, the AI's response time is around 40 seconds, and for an analysis depth of 5, the AI's response time is about 5-10 minutes (or even longer, depending on the state of the board).

Demo



What's next

- New menus: Start and Pause
- Possible changes to Existing Ui
- Optimize the analysis time required for each level of analysis in the AI

Questions