NTHU I2P3 MiniChess AI

state value function

Assign a material value to each piece. The state value is the sum of the value of pieces on the board

```
/** state.hpp */
const int material_value[7] = {0, 2, 6, 7, 8, 20, 1000};
// empty, pawn, rook, knight, bishop, queen, king
```

```
/** state.cpp */

int State::evaluate(){
  // [TODO] design your own evaluation function
  if (game_state == WIN) return INF;

  int value = 0;
  for(int i = 0; i < BOARD_H; i++) {
    for(int j = 0; j < BOARD_W; j++) {
      value += material_value[(int) board.board[player][i][j]];
      value -= material_value[(int) board.board[1 - player][i][j]];
    }
  }
  return value;
}
```

```
int State::evaluate(){
  // [TODO] design your own evaluation function
  if (game_state == WIN) return INF;

  int value = 0;
  if (player == 0) {
    for(int i = 0; i < BOARD_H; i++) {
      for(int j = 0; j < BOARD_W; j++) {
        value += material_value[(int) board.board[player][i][j]] - white_piece_square_table[(int) board.board[player][i][j]][i][j];
        value -= material_value[(int) board.board[1 - player][i][j]] - white_piece_square_table[(int) board.board[1 - player][i][j]]
      }
    }
  }
  else {
    for(int i = 0; i < BOARD_H; i++) {
      for(int j = 0; j < BOARD_W; j++) {
        value += material_value[(int) board.board[player][i][j]] - white_piece_square_table[(int) board.board[1 - player][i][j]][BOA
        value -= material_value[(int) board.board[1 - player][i][j]] - white_piece_square_table[(int) board.board[player][i][j]][i][
      }
    }
  }
  // int value = 0;
  // for(int i = 0; i < BOARD_H; i++) {
  //   for(int j = 0; j < BOARD_W; j++) {
  //     value += material_value[(int) board.board[player][i][j]];
  //     value -= material_value[(int) board.board[1 - player][i][j]];
  //   }
  // }
  return value;
}
```

```
// empty, pawn, rook, knight, bishop, queen, king
const int white_piece_square_table[7][BOARD_H][BOARD_W] = {
  {
    // empty
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0},
  }, {
    // pawn
    {50, 50, 50, 50, 50},
    {30, 30, 20, 10, 10},
    {5, 5, 10, 10, 25},
    {5, -5, 0, 0, 0},
    {5, 10, 10, -20, -20},
    {0, 0, 0, 0, 0},
  }, {
    // rook
    {0, 0, 0, 0, 0},
    {5, 100, 10, 10, 5},
    {-5, 0, 0, 0, -5},
    {-5, 0, 0, 0, -5},
    {-5, -5, 0, 0, -5},
    {-5, -5, 5, 0, -5},
  },{
    // knight
    {-50, -40, -30, -40, -50},
    {-40, -20, 0, -20, -40},
    {-30, 15, 20, 15, -30},
    {-30, 15, 20, 15, -30},
    {-40, -20, 5, -20, -40},
    {-50, -40, -30, -40, -50},
  }, {
    // bishop
    {-20, -10, -10, -10, -20},
    {-10, 5, 5, 0, -10},
    {-10, 5, 10, 5, -10},
    {-10, 5, 5, 5, -10},
    {-10, 0, 5, 0, -10},
    {-20, -10, -10, -10, -20},
  }, {
    // queen
    {-20, 0, 0, 0, -20},
    {-10, 0, 0, 0, -10},
    {-10, 5, 5, 5, -10},
    {-10, 5, 5, 5, -10},
    {-10, 0, 0, 0, -10},
    {-20, -10, -10, -10, -20},
  }, {
    // king
    {-30, -40, -50, -40, -30},
    {-30, -40, -50, -40, -30},
    {-30, -40, -50, -40, -30},
    {-30, -30, -30, -30, -30},
    {20, 0, 0, 0, 20},
    {20, 30, 0, 30, 20},
  }
};
```

Minimax tree search

1. Basically, our agent plays as the "max" and the opponent is "min". The max player's goal is to maximize the state value of its next state. So, we'd like to explore

all next states of the current state, and decide which is the max state. However, only looking at 1 layer might not be enough, since the opponent is smart and probably won't let you benefit. Thus, we should look forward to more steps and simulate how opponent thinks.

2. I chose to start with depth 4, and increment depth by 2 while our ai is not killed yet.

```
void write_valid_spot(std::ofstream& fout) {
  // Keep updating the output until getting killed.
  // output result from depth: 3, 5, 7, ... (if can be calculated in time)
  // Reason: avoid not outputting anything after exceeding the time limit.
  int depth = 4;
  while(true) {
    auto move = Submission::get_move(root, depth);
    fout << move.first.first << " " << move.first.second << " "\
         << move.second.first << " " << move.second.second << std::endl;

    // Remember to flush the output to ensure the last action is written to file.
    fout.flush();
    depth += 2;
  }
}
```

3. In the get_move_helper function, we loop through all next states and return the move s.t. its next state has the max. alphabeta() value

```
Move Submission::get_move_helper(State *state, int depth) {
  int max_value = -INF;
  int alpha = -INF;
  int beta = INF; // NOTE: beta in the root node is always INF
  std::vector<Move> potential_moves;

  for (Move action : state->legal_actions) {
    int potential_value = alphabeta(state->next_state(action), depth - 1,
                                    alpha, beta, false);

    if (potential_value > max_value) {
      max_value = potential_value;
      potential_moves.clear();
      potential_moves.push_back(action);
    }
    else if (potential_value == max_value) {
      potential_moves.push_back(action);
    }

    alpha = std::max(alpha, max_value);
    if (alpha > beta) break;
  }
  return potential_moves[rand() % potential_moves.size()];
}
```

Alpha-Beta pruning

1. alpha: maximum value the player has in the current search process (its parent node)
2. beta: minimum value the player has in the current search process
3. basic idea: if alpha is greater than beta, then we can stop searching in the current branch, since the parent node will not choose this value. (if parent is player, then the player'll just choose alpha rather than beta, so there's no meaning for us to keep searching for smaller beta)
4. alpha-beta is guaranteed to give the same result as minimax, but is much more efficient.

```
int Submission::alphabeta(State *state, int depth, int alpha, int beta, bool is_max_player) {
  if (state->game_state == WIN || depth == 0) {
    if (is_max_player) return state->evaluate();
    else return -(state->evaluate()); // NOTE: evaluate() returns opponent's value
  }

  if (is_max_player) { // find max value
    int max_value = -INF;
    for (Move action : state->legal_actions) {
      max_value = std::max(max_value, alphabeta(state->next_state(action), depth - 1,
                                                alpha, beta, false));
      alpha = std::max(alpha, max_value);
      if (alpha > beta) break; // beta cutoff
    }
    return max_value;
  }
  else { // find min value
    int min_value = INF;
    for (Move action : state->legal_actions) {
      min_value = std::min(min_value, alphabeta(state->next_state(action), depth - 1,
                                                alpha, beta, true));
      beta = std::min(beta, min_value);
      if (alpha > beta) break; // alpha cutoff
    }
    return min_value;
  }
}
```