Universität Potsdam
Institut für Informatik und Computational Science
Studiengang Bachelor Informatik/CS

Chessam
A chess game with a GUI and different engines to play against.

Projektbericht

von

Samuel Rodriguez

Simulation und Modellierung
Betreuer: Dr. rer. nat. Ralf Tönjes
Winter Semester 2021-2022

Contents

1 Introduction and Goal of the Project

Chess is a game that has caused the fascination of thinkers for more multiple centuries, from hobbyists to mathematicians, kings, philosophers and, more recently, computer scientists, the famous game attracts attention from every direction. The study of computer chess is as old as computer science itself. Babbage, Turing, Shannon, and von Neumann devised hardware, algorithms and theory to analyse and play the game of chess. Chess subsequently became the grand challenge task for a generation of artificial intelligence researchers[1].

The goal of this project is to create a simulation of the game with a graphical user interface along with the possibility to play against different types of agents that use different heuristics for choosing the best move.

2 Implementation

All the code is publicly available at https://github.com/Sam1320/chessam. The language used is Python 3.9+ and the GUI library is TKinter, which is already included in most python installations. There are three mayor components that were implemented in this project. The core logic of the game, the graphical user interface and the different chess bots. The details of the GUI application will be for the most part ignored as they are specific to the library and language chosen and unlike the core game principles do not transfer to other implementations. The other two components will be described in detail in the following sections.

2.1 Core Logic

The first step before trying to program a chess engine or interface is to have a reliable simulation that complies with the rules of the game. The are several constraints that are relatively trivial to implement and will therefore not be described in this document for the sake of brevity. Some examples of these constraints are: initial board setup, piece placement, tracking the number of moves played, blocking or allowing moves depending on whose turn it is, keeping track of the pieces taken by each side, etc. Most of the difficulties when implementing the game revolved around checking the validity of a movement for a given piece at a given board position. Except for pawns, most of the moves follow a specific pattern for each piece and the procedure for checking valid moves is similar even for different pieces. However there are a few special move cases that will be explained separately.

The move pattern similarity between pieces allows for the creation of a base class 'Piece' with several common attributes and methods, sub-classes of this

[1]. M. Campbell, A. J. Hoane, and F. Hsu. Deep Blue. Artificial Intelligence, 134:57–83, 2002

base class are then created for each piece type. With this structure we take advantage of inheritance and increase code re-usability. In this setting each Piece is responsible for verifying the validity of a given move and also calculating all possible moves available to it.

The board is represented as 3 separated dictionaries:

- *pieces_coords* maps from objects of the class 'Piece' to the coordinates of the given piece on the board or to the value None if the piece has been taken.
- *coord_pieces* maps coordinates to either Pieces or the value None if the square is not occupied.
- *name_piece* maps piece names in the format *Color_Type_Column* to instances of the class Piece. This dictionary also has a key named *en_passant* that maps to an enemy pawn if it is available to be taken *"en passant"* or to the value none if there is no piece that can be taken *"en passant"* on this turn.

2.1.1 General Moves

Each piece type has a unique style in which they move:

- Pawns can move one square forward or one square diagonally in front of them when they capture an enemy piece. They are the only piece that captures differently than how they move and the only piece that cannot move backwards.
- Knights can move two steps vertically followed by one horizontal step or they can move two steps horizontally followed by a vertical step. Knights are the only piece that can "jump" other pieces, that is, as long as the landing square is valid, it doesnt matter if there are other pieces in its trajectory.
- Bishops can move diagonally in every direction.
- Rooks can move horizontally and vertically in every direction.
- Queens can move diagonally, horizontally and vertically in every direction.
- Kings can move like the Queen but restricted to only one square away from its current position.

Capturing pieces of the same color is not allowed. The base principle to determine if a piece can move to a given position is the following:

1. verify that the landing position is in the grid.

2. verify that the landing position would be reachable by the piece if the square was free. i.e. given the specific move patterns check if the position can be reached from the current location.

3. verify if there is an obstacle in the path. (Not required for Knights, Kings and Pawns).

4. verify if the landing position is occupied by an enemy or by a friend.

5. verify if my king would be in check if the piece moves to the given position. i.e. the piece is currently protecting the king.

As an example to see how this principle is implemented, the following is the code for generating all possible Rook moves given a specific board setup (several lines are removed the sake of readability). The given implementation can be optimized in multiple ways, for time reasons most algorithms in this project are naive or not very optimized.

```
class Rook(Piece):
    ...
    def possible_moves(self, coords_pieces, pieces_coords, player,
         name_piece):
        x1, y1 = self.position
        moves = []
        for i in range(1, 8):
            # check moves to the right
            if self.valid_move(x1+i, y1, coords_pieces,
                pieces_coords, player, name_piece):
                moves.append((x1+i, y1))
            # check moves to the left
            ...
        return moves

    def valid_move(self, x2, y2, coords_pieces, pieces_coords,
        player, name_piece):
        if not self.legal_move(x2, y2, coords_pieces,
            pieces_coords, player):
            return False
        x1, y1 = self.position
        if (x1 == x2 or y1 == y2) and (x1 != x2 or y1 != y2):
            # upward movement
            if x1 == x2 and y1 > y2:
                for i in range(1, abs(y1 - y2)):
                    if coords_pieces[(x1, y1-i)]:
                        return False
            # downward movement
            elif x1 == x2 and y1 < y2:
            ...
            # rightward movement
            ...
            # leftward movement
            ...
            return True
        else:
            return False
```

Code Block 1: Rook Moves. For each direction the method possible_moves() checks the validity of each step on that direction. valid_move() is given a specific goal position x2, y2 and checks first the legality of the move with legal_move and then if there is any obstruction along the way, if the coordinates of the goal position are not consistent with the way the rook moves then the procedure returns False directly without looking for obstructions.

```
1  def legal_move(self, x, y, coords_pieces, pieces_coords, player):
2      pieces_coords_copy = deepcopy(pieces_coords)
3      coords_pieces_copy = deepcopy(coords_pieces)
4      if self.on_board(x, y) \
5              and self.my_turn(player) \
6              and not self.friend_here(x, y, coords_pieces_copy) \
7              and not self.my_king_checked(x, y, coords_pieces_copy,
                     pieces_coords_copy):
8          return True
9      return False
```

Code Block 2: General legal move check, as explained before several checks have to be done for certifying the legality of a move, most checks are trivial except for the one in *line:7* checking if the own king would be in check when moving the piece to the new position (x, y). For this verification the move to position (x, y) is first made and then the king has to be found to subsequently check for attacks in all directions, if an attack is detected the procedure returns False and the move is not permitted.

2.1.2 Castling

This special move allows the king to move two steps in the direction of either of the own rooks and at the same time move that rook one square to the side of the king that was opposite to its previous side with respect to the king. For example if the rook was to the left of the king, after castling the king will have moved two steps tho the left and the rook will be one square to the right of the king.

Conditions for castling:

- The king and the rook involved must not have moved before.
- The squares between the king and the rook have to be free.
- The squares between the king and the rook must not be under attack. (i.e. in the "line of sight" of one enemy piece)

To verify this move the king class has two special attributes, one that checks if the piece (King) has been moved and one that stores a reference to the two rooks of the same color, each of the rooks also have a flag variable to check if the piece has been moved. If either of the Pieces has been moved then castling is not allowed. Then we verify the that the squares in between are free and not under attack. They way we check if a square is under attack is by placing the king on that square and looking for checks. This way we avoid writing redundant code since the functionality for the verification of a check its already implemented in the king class.

2.1.3 Pawn first move

Pawns can move 2 squares forward in their first move. To encode this behaviour it is necessary to check whether the piece has been moved before or not, and since pawns can only move forward, it is enough to check if the pawn is in the starting row or not.

2.1.4 En passant

A pawn attacking a square crossed by an opponent's pawn which has advanced two squares in one move from its original square may capture this opponent's pawn as though the latter had been moved only one square. Each time a pawn moves two squares on the first move the piece is stored in the *name_piece* dictionary to be able to be retrieved by the enemy on the next turn. Each time a pawn is assesing the validity of a move it checks if there is a piece available to be taken *"en passant"* and then checks if this piece is to the right or to the left of the current piece, if this is the case, then the move *"en passant"* is available to be made.

2.1.5 Pawn Promotion

When a pawn reaches the last row on the board it can be "promoted" into any other piece with the exception of the king. The promotion is mandatory i.e. it cannot be the case that the pawn reaches the last row and stays a pawn. The promotion is managed by the GUI class which displays four buttons when promotion is available, one button for each possible new piece type. Once selection is made the new piece is created and placed on the board.

2.2 Agents

Several bots were implemented to be played against. Each agent consist of a function that takes in a game state, which includes the current board position and turn, and outputs a piece and a board coordinate to move the given piece.

2.2.1 Random Move

This move consist of randomly selecting a piece and move from all the possible legal options. First an empty dictionary is created that will map pieces to lists of moves. Then all possible moves for each piece on the board of the current color are found and the piece and respective moves are added to the dictionary. We then randomly select a piece (i.e. a key) from the dictionary, retrieve the corresponding move list and also randomly sample a move from this list. The result is a random piece moving to a random square from all the possible valid piece and move combinations.

2.2.2 Random Attack

Similar to random move this function will return a random attacking move from all attacking moves available, if not attack is possible then a random move is selected. The same dictionary as in the random move generation, which contains all available moves, is created. Then from these available moves another dictionary created by filtering the lists of moves to only contain attacking moves, hence the new dictionary also maps from pieces to lists of moves but this time all moves are attacks. Again a random piece and move are sampled.

2.2.3 N-Step Lookahead

This function is the first one that does some more interesting computation. It consists of estimating the best move by analyzing subsequent possible moves n steps in the future. Two main components are necessary: The first is a way to assign a numerical value to a board position to be able to compare hierarchically between alternative options. The second is a way to organize the sequence of possible moves and go through each one.

Functions that give us a numerical estimate of the quality of a game state are called heuristics. Luckily, chess pieces have a relative value assigned to them that is a measure of their usefulness or strength. These values are the following: pawns = 1, knights and bishops = 3, rooks = 5, queen = 9. Since loosing the king is equivalent to loosing the game, the value of the king is not considered. One very simple but useful heuristic is to sum the values of all pieces of the color of the player and subtract from that the sum of all values of the pieces of the opposite color. Hence the bigger the value of the heuristic the more favorable the position for the player regardless of the color.

```
def board_eval(node, player):
   pieces_coords = node["pieces_coords"]
   score = 0
   for piece, coords in pieces_coords.items():
       if piece and coords:
           if piece.player == player:
               score += piece.value
           else:
               score -= piece.value
   return score
```

Code Block 3: Board evaluation heuristic

The core logic of most AIs for adversarial or zero-sum games, where the advantage of one player signifies a loss for the opponent, is based on an algorithm called minimax. This algorithm provides a way to search the space of possible moves n steps in the future and choose the best assuming perfect play from both sides according to the used heuristic. It is based on the principle that the player is trying to choose the move that maximizes his score while the opponent will choose the move that minimizes the score of the player. Based on this a tree of possible moves of depth n is built recursively where at each level the parent node returns the maximum or minimum value from all children nodes, at each recursion the depth n is decreased by 1, and when n = 0 or when there are no more moves available a static evaluation of the game is returned.

One optimization of this algorithm is called *alpha beta pruning* which is takes advantage of the fact that some parts of the tree do not need to be searched because they are not going to be 'considered' by the previous level. For example take a minimizing node which has searched three children and the smallest value returned by the children so far is 10, then if the fourth children node which is maximizing the score, finds a children with a value of 11, then no further children need to be explored because although a value greater than 11 could be found, this value would be ignored since it is worst than the best possible option of the minimizing node which was 10. This pruning results in a considerable smaller number of positions to be evaluated.

```
def minimax(node, depth, maximizing_player, alpha, beta):
    player = node["player"]
    is_terminal = is_terminal_node(node)
    if depth == 0 or is_terminal:
        return board_eval(node, player)
    valid_moves = available_moves(node)
    if maximizing_player:
        value = -np.inf
        for piece, moves in valid_moves.items():
            for move in moves:
                child = do_move(node, (piece, move))
                value = max(value, minimax(child, depth-1, False,
                    alpha, beta))
                alpha = max(alpha, value)
                if beta <= alpha:
                    return value
        return value
    else:
        value = np.inf
        for piece, moves in valid_moves.items():
            for move in moves:
                child = do_move(node, (piece, move))
                value = min(value, minimax(child, depth-1, True,
                    alpha, beta))
                beta = min(value, beta)
                if beta <= alpha:
                    return value
        return value
```
Code Block 4: Minimax algorithm with alpha beta prunning

2.2.4 Stockfish

Because the implementation of the game is mostly based on a naive approach and hence very poorly optimized for speed, the time it takes for the n-step-lookahead bot to see more than 3 steps in the future is prohibitively long. So, in order to offer a challenging playing experience, an interface to the stockfish engine was created. The Stockfish project started with the open source Glaurung engine, authored by Tord Romstad. Meanwhile, Stockfish quickly rose to become the strongest open source chess engine, with frequent updates every few months. Today, it remains one of the strongest engines in the world.[2]

The engine uses FEN [3] notation to convert a board the position to an interal representation and then asses the position and return the best move also in FEN notation. To interact with stockfish two functions were created, one to convert the board state to FEN notation and one to convert the move returned by stock fish to the corresponding representation used by the system. After the player moves the board is passed to stockfish which returns a move which is then translated and played. It is worth noticing how fast stockfish makes a decision considering the strength of the engine.

3 Results

This project resulted in a perfectly usable chess game for human vs human play, with all the rules of the game properly encoded and a visualization of the possible moves when we click on any given piece as seen on figure 1. The game shows when a king is checked or checkmated, what the current board evaluation is and also whose turn it is. When a pawn reaches the last rank four buttons are displayed, each with a piece drawn on it, to offer the 4 possible promotion options. The human vs bot option also functions properly with the disadvantage of the waiting time while the bot makes a move, the more steps in the future the bot calculates, the longer the wait. Table 1 summarizes the average times required for the different bots to make a move, here one can see the drastic improvement in efficiency when applying alpha beta prunning to the mini-max algorithm. There is a trade-off between bot thinking time and bot quality of play. For beginners, the 2-step-lookahead bot might be a good adversary for simple practice and relative quick games, the 3-step-lookahead bot can represent a challenge for complete beginners with a time per move short enough to still be playable. For more experienced players the interface to Stockfish allows for a very competitive level of play and quick bot reaction time.

[2]https://stockfishchess.org/about/
[3]https://en.wikipedia.org/wiki/Forsyth_Edwards_Notation

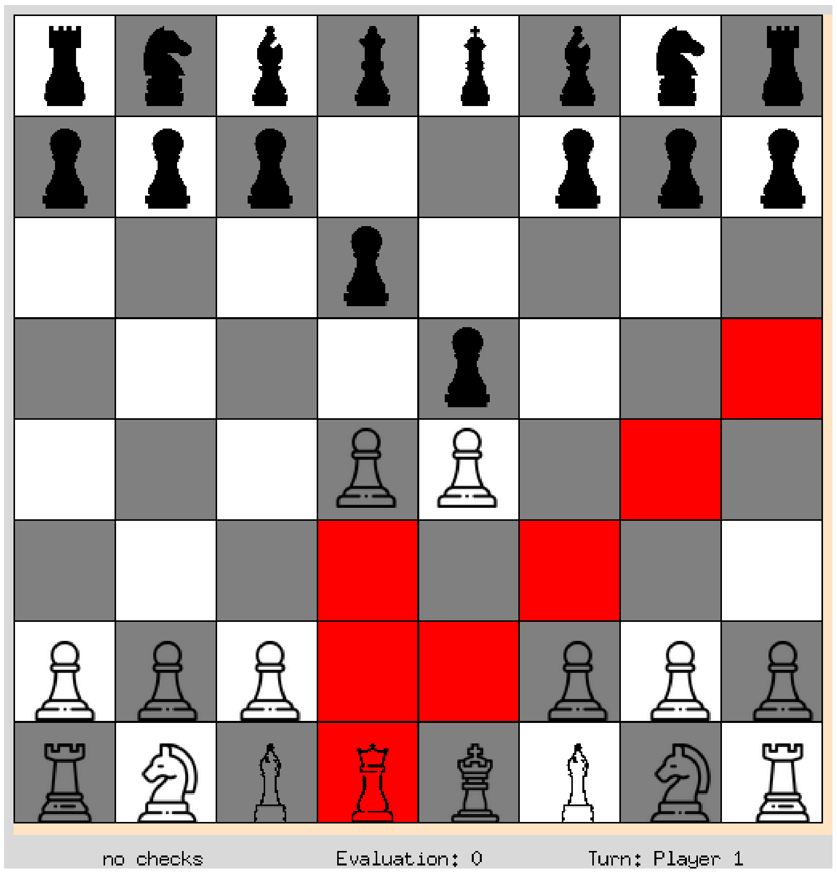

Figure 1: Board position with possible squares where the queen can move highlighted.

	random	1-step	2-step	3-step	stockfish
without prunning	0.3s	0.34s	12.52s	854.23s	0.51s
with prunning	— " —	0.31s	8.59s	119.25s	— " —

Table 1: Execution time of different chess bots when all the pieces are still on the board. The first column shows the average time the random bot takes to make a move, this time is in the O(1) time complexity class since it doesn't evaluate any future moves and is hence constant in expectation. The other columns show the average time it takes the n-step look-ahead bot to make a move with a full board for n = 1, n = 2 and n = 3.

3.1 Next Steps

The are several improvements that can be done to the project, the biggest one is to use bitboards as data-structure for representing the state of the board. Bitboards are used by most modern chess engines, they represent the game as 12 8x8 binary arrays, one for each piece type of each color, this allows to exploit the efficiency of binary operations and assess square occupations and allowed moves in parallel instead of sequentially. This efficient representation would allow for looking further steps ahead and hence better playing bots. Another improvement is to refine the heuristic for the board evaluation, accounting for more details in the position for example giving more value to pawns that are close to promotion or less value to trapped bishops. Regarding the minimax algorithm there are also several possible further optimizations, one of them would be sort the possible moves by the expected value from bigger to smaller, that way we would find faster sections of the tree that need to be pruned.

4 Conclusion

The development of a chess game completely from scratch represents a big challenge, there is a huge amount of cases and exceptions to consider as well as a lot of different possible paths to be taken when trying to optimize the implementation. It is worth noticing that when working on this project I avoided looking at any solution on the internet or books and focused on implementing everything myself, this had several advantages and disadvantages. The main disadvantage of a mostly naive implementation is the poor efficiency when a bot evaluates thousands of positions due greatly to the data representation and sequential move validation. Another disadvantage of not relying on already existing solutions was the long time it took to overcome some obstacles, for example to solve the en passant move, several different approaches where taken, which took some days of work. However, in my opinion, the advantages greatly outweigh the drawbacks. When one solves a problem without recurring to templates or external help, there is the need to truly understand what the details of the issue are, to device different plans and through trial and error find the best solutions and gradually move forward on the project. This has the positive side effect of developing deep familiarity with the field and most importantly learning directly from experience why some approaches are better than other, instead of directly looking for the best solution in the literature without truly understanding why it is the best.