**Patrick Walsh**
**UMass Amherst 2024**
**Computer Engineering**

# Creating and Developing a Chess Computer *Documentation*

**August 7 2022**

---

- Created repository ChessAlgo on GitHub

Much of this day was spent planning how I was going to implement the Chess Learning Algorithm. For the majority of the process I wanted to depend on my own problem solving skills and not rely on outside resources like StackOverflow or Youtube. After some debate, I decided to use Python and pygame to construct the environment. This project is my first time using pygame so I used the documentation to set up the environment.

I set up the environment and got the initial movement of the pawns and rooks down. I was handling the movement in setup.py, but I felt like that was severely cluttering the file. While the pawn movement, in my opinion, was best suited for setup.py (as pawn movement strongly depends on the color of the piece) I planned to implement the rest of the pieces

Another part I felt important to develop on this day was a fen string parser. Fen strings are an easy string that directs the computer to place each piece in a specific position. Putting this in on the first day was necessary as to be able to play around with different positions and make it so the file isn't setting up each piece itself, but instead reading from a preset string.

**August 9 2022**

---

- Implemented Queen, Bishop Movement, Fixed Rook + Pawn Movement

I moved all movement over to Piece.py to stop relying on setup.py so heavily. The Queen movement was made to take the rook and bishop functions and add their possible spots together in the circle list. While I wanted to do the same thing with the King, it was more difficult. While the Queen, Bishop, and Rook can all move across the entire board, the King can only move one space, and this proved to be difficult to implement into my current functions. I may implement

this in the future but my current goal is to get the Chess part working so I can begin work on the Chess Algorithm.

**August 10 2022**

---

- Added Knight movement, King movement and added King ability to castle

Knight and King movement wasn't difficult to implement as they only have eight total spots they can move. The way I thought it would be easiest to implement for the Knight and King was to have a list of all possible moves and check them all to see if they could move there. I also added the King's ability to castle, and the castling check will be handled by the king() function in Piece.py. I also realized movement of the pieces and maintaining all of Chess' intricate rules would be too much for setup.py to keep clean. Instead, I created a move.py which will handle all movement going forward.

**August 12 2022**

---

- Moved movement handling to move.py, added PGN creator, adding check mechanics

There wasn't too much movement handling to move to move.py as for much of it I was waiting to add the check mechanics. I can't add all the special movement permissions if that piece isn't supposed to be able to move because the King is in check. So I also began to add the check mechanics. While the current system I'm implementing may not be the most efficient in terms of memory, I believe it should be fast enough and do the job.

The Check system is handled by four lists and move.py. Two of the three are in Piece.py and are a part of the Piece() object. attackMoves is used to remember all spaces each piece is attacking and pieceobstr remembers which pieces stop pieces from attacking a space they could attack if it was a clear board. For example, if a Rook was on a1, the pawn on a2 is obstructing the line of the rook, so a2 would be added to pieceobstr. All squares white is attacking and all squares black is attacking will be held in their own separate lists. Every time a piece (we'll call p1) moves, move.py will check all pieces in the pieces list and see if any piece has p1 in their pieceobstr. If true, move.py will delete all the attack squares from that piece from its colored attackSquares list (which is why each piece must remember what squares they're attacking) and

refresh the squares to reflect the new spaces. While this system hasn't been implemented yet, the goal is to have it properly account for discovered checks and checkmates (if no piece can defend the king and all the king moves are in attackSquare list for opposite color).

The final addition was mostly minor but I felt it important to add now for en passant. The program, after termination, will now print the PGN for the game. Pieces on the same column or row have not been accounted for yet but will be later once necessary.

**August 15 2022**

---

- Created check system

The base system is down for checks and holding all attacking squares however the current system has some bugs that need to be fixed. The major one is it seems like the only working way to have all pieces know where each other piece is on the board is to calculate out all possible moves. I first thought I'd be able to itemize each piece and only check pieces that had the piece that was obstructing it, but that doesn't account for the piece it goes next to. There are too many ifs and the O(n) time is reasonable considering most of the systems are already built, and the case I'm working with now is essentially O(n) as O(n) is the worst case. There are some parts that run at O(n^2), like removing each piece's attackMoves from attackSquaresW/B. I will also be trying to think about a more efficient way to do this.

**August 23 2022**

---

- Finished most of check system, completely changed way moves are found

Created the basis for checks and checkmates. When the king is found to be in check, it checks to see what piece put it in check and from what coordinate. A string of conditional statements then determines what coordinates other pieces can move to to stop checkmate. The king moves according to env.attackingSquares<COLOR>, so its movements are independent of this conditional string. All pieces are also now given their circles in a self.circles list instead of returning circles to some value. This is because in the case of a check, a list called saveking, full of the coordinates to stop checkmate, is sent to each piece. If the initial system was kept, only the instant the King was put in check would the pieces move where they were supposed to. If clicked

on, they would move to all available spaces. Also, it's just more computationally efficient to not find all available spaces every time a piece is clicked on, and instead just have them already for every click. The black pawn and white pawn were also given their own two functions in piece.py which was long overdue.

**August 24 2022**

---

- Added pinned pieces, fixed various check issues

The main part added in this update was pinning of pieces. I had to ensure that the white and black king's movement functions were called first so each piece knew if they were pinned to the King, and from what angle. piece.findpinnedpieces() is the function that handles that. In move.py, I created a function called findsaveking() to place all of the conditional check code in to avoid the walls of text that are beginning to form. Auto-queening for pawns was also introduced. Some check issues were fixed, i.e. when in check, pieces could still capture other pieces, but no other moves were available.

**August 26 2022**

---

- Added draw by repetition, eval.py, determined initial structure of chess algorithm

This update added a draw by repetition and I finalized my initial thoughts on a chess algorithm. While I initially thought tensorflow would be a good match for this project, I now think I'm instead going to use a self-implemented regression algorithm to determine the evaluation of the position in place of tensorflow. With my own self-implemented regression algorithm, I can make it specifically to what I need for the model. Paired with the regression algorithm, I'll use alpha-beta pruning to move through the best moves for white and black to see what the end evaluation is, and how many pawns up either side is. As a side note I also thought the best way to keep track of the position was to take a fen string at each position as it only takes O(n) (n as the number of squares on the board) to complete.