MACHINE LEARNING, CHESS and MATHEMATICS

Zain K Aamer Sravani Bhamidipaty Nirgun Goel Bill Shepelak Xuan Duc Tran Ling Yu

University of Illinois at Chicago

MSCS Undergraduate Research Laboratory

UIC

About this project

This project consists of work that was done under the supervision of Andrew Shulman and Evangelos Kobotis at the MSCS Undergraduate Research Laboratory. One of the main ideas of the project was to begin from scratch, to understand how the game of chess can be parameterized and eventually to consider different research paths. The chosen research paths were followed by one or more of the participants at a time, individually or in collaboration.

The game of chess

Chess is a traditional board game that has been enjoyed by everyone from kings and queens to schoolchildren and hobbyists. It is frequently referred to as a game of strategy, skill, and patience. In the sixth century AD, the game of chess was invented in India and swiftly moved to Persia and eventually to the Islamic world. From there, it traveled to Europe in the ninth century, where the nobility took a liking to it. Chess is now played across the world by players of all ages and socioeconomic backgrounds. On a square board with 64 squares organized in an 8x8 grid, chess is played. A king, a queen, two rooks, two knights, two bishops, and eight pawns are given to each player at the start of the game. The goal of the game is to checkmate your opponent's king, which means trapping it so that it cannot escape capture.

Chess and computers

Looking back at the history of the game, chess computation is something relatively new, dating back about 70 years ago. The first chess program was created by Alan Turing and David Champernowne in 1948 and since then, we have developed chess programs that have become stronger as time has passed. In 1997, Deep Blue, a chess-playing supercomputer, defeated Grandmaster Garry Kasparov and becoming the first computer to beat a world-champion chess player in a classic match format. Chess program has never stopped developing and continues to improve, surpassing humans by compute billions of scenarios within a second. The highest-rated computer chess program, Stockfish, has more than 3500 ELO now.

Chess parametrization

In our project, we have created a chess board using a Python nested array. This involved generating eight arrays that corresponded to the eight columns of a chess board, with each array containing elements that represented individual squares on the board. The chess pieces were represented by integer values We have implemented a 6-digit Operation Code that allows users to access and move the chess pieces. The first two digits indicate the type of move (non-capture/capture, promotion, or castling), while the next two digits indicate the starting square of the piece that the player wants to move. Finally, the last two digits specify the destination square of the piece after the move is made. By utilizing this Operation Code, users are able to input all possible moves in a game of chess using only six digits.

Programming an interface

Once we achieved a basic parametrization of the game and we ensured that for each position we had all possible moves, we proceeded to program an interface on which the different patterns and methods could be implemented. To this end pygame was used in order to simulate the different game. The interface consolidated our understanding of how to play chess and how to test the algorithms that we considered.

Using neural networks

One of the most important aspects of our work was to understand how neural networks can be used in order to implement a chess engine. We considered the basics of neural networks and we posed the question of what is the simplest neural network that can be used in order to create a virtual non-trivial chess player. At this point, it should be noted that we have two examples of using neural networks in chess programming: the addition of neural networks to the open source popular Stockfish engine and the very advanced Alpha Zero program which is entirely based on neural networks. Our desire was to start from scratch. To this end we considered the implementation of a convolutional neural network that can be trained so as to play chess at a non-trivial level.

Evaluating a chess position

In a game of chess, certain pieces are considered stronger than others. By assigning each piece on the board with a value, we can know the strength of each pieces in chess. For example, pawns are assigned a value of one point, while knights and bishops are assigned with a value of three points. By calculating the values of all the pieces on the board and evaluating the game state accordingly, the chess program can utilize the minimax algorithm to look forward in the decision tree and make decisions up to 20 moves ahead. It is worth noting that the computer program will prioritize achieving checkmate over gaining material advantages, and may even sacrifice pieces if doing so will lead to a checkmate opportunity. In general, the process of evaluating a chess position is paramount in every area of chess programming. In fact efficient and effective evaluation is the key to program any playing or training chess engine.

Procedural study of chess openings

Chess openings are the initial moves of a chess game that set the stage for the rest of the game. They are crucial in determining the strategic direction of the game and can significantly impact the outcome. One of the paths that we took in our study is to look at large databases of games and to explore how we can analyze the different openings and come up with learning strategies. One way to do this is to come up with trees of possibilities and classify opening traps and opening blunders.

By importing the *chess* and *stockfish* libraries, the function plays chess moves and evaluates the resulting board positions using the Stockfish chess engine.

The code creates a chess board object using the *chess.Board()* function. It also creates an instance of the Stockfish engine using the *Stockfish()* class, passing the path to the Stockfish binary file as an argument. The depth and skill level of the Stockfish engine is set to 20 using the *set_depth()* and *set_skill_level()* methods, respectively. The current parameters of the Stockfish engine are printed using the *get_parameters()* method.

The function *play_pgn_moves(pgn_string)* takes a PGN (Portable Game Notation) string as input. The function splits the input PGN string to extract the moves and iterates over the moves in pairs using a for loop. Inside the loop, the code uses the *board* object to make the moves on the chess board using the *push_san()* method, which pushes the moves in Standard Algebraic Notation (SAN) format. After each move, the current board position is printed using *print(board)*.

The *stockfish* engine is then used to set the FEN (Forsyth-Edwards Notation) position on the board using the *set_fen_position()* method, and the evaluation score of the current position is obtained using the *get_evaluation()* method. The evaluation score represents the advantage of the current position for the side to move, with positive values indicating an advantage for White and negative values indicating an advantage for Black. The evaluation score is also printed using *print(evaluation)*. Finally, the function continues to the next move in the PGN string and repeats the process until all moves are made on the board.

Zobrist hashing algorithm in chess programming

In chess, an efficient hashing table is needed when traversing a decision tree without re-analyzing the board position each time. One of the most common hashing algorithms in chess is Zobrist Hashing. We implement Zobrist hashing by generating a random integer key using a 64-bit number, multiplying it with 12 (which stands for the number of individual pieces in chess), and then taking the random integer from the result. For each square on the chessboard that is not empty, we then perform XOR on every square that has a piece on the board, and return the total as the hash value of the given position on the board.

This approach is extremely effective when used with the Minimax Algorithm. When traversing the decision tree, the value of the position can be stored in the hash table and re-used later without having to re-analyze the same position, which speeds up the algorithm.

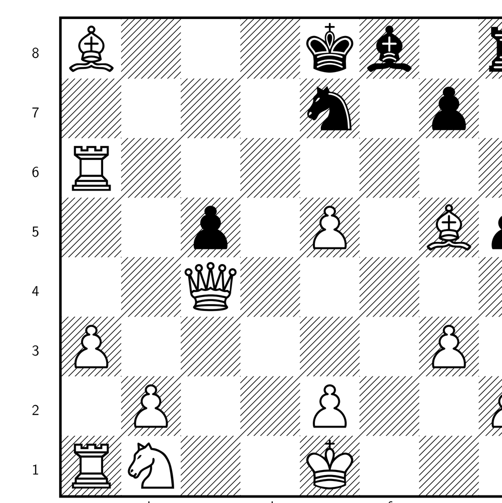
Algorithmic generation of chess problems

This branch involved algorithmically generating chess problems. Chess problems are board positions, accompanied by a prompt, such as "Mate in 3 as White." Here, we explored our main approach, one based on searching games.

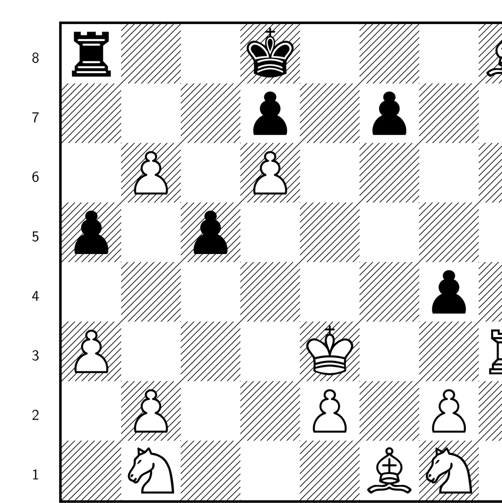
In order to support the number of board positions that have to be considered for both of these approaches, the chess problem generator uses a unique chess engine implementation in the Rust programming language. This system is generic over board size, allows combining of pieces trivially and allows easy addition of entirely new pieces, such as Berolina pawns. We had to focus on performance, using bitflags for marking a piece's movement abilities, Zobrist hashing for transposition tables and aggressive pruning when searching for a forced mate.

Our search-based algorithm generated chess games, by having the computer play against itself, but with White having a more intelligent AI. As a result, White always checkmated, giving a usable final position. It then looked $2n-1$ moves behind, for a mate-in-n problem, so for a mate-in-3 problem, it'd be 5 moves behind. It then checked if there was only one move resulting in a mate-in-n and that it was not a capture, check or promotion. If so, it was a valid puzzle. We called this method a search-based one as it searches real games for a valid problem. Originally, it had been intended to search historical games, but a high rate of drawing and resignations as soon as an eventual mate became obvious made that inefficient.

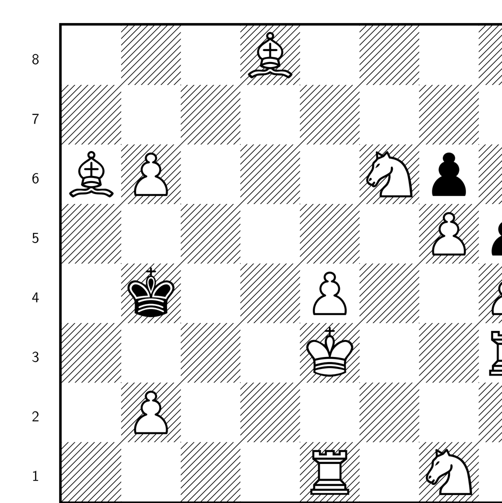
Here are three examples of such problems, where the goal is a mate in 3 by White:



Solution: Here, White should play **1 ♕e6**. No matter how Black responds, White will deliver mate in three moves.



Solution: Here, white should play **1 b7**. Black has to block the pawn's promotion, as a promotion to a queen would be checkmate. This allows **2 ♗f6+ ♔e8 3 ♖h8#**.



Solution: White should play **1 ♔d4**, which regardless of what Black plays, allows **2 ♖a3+ ♔b4 3 ♘d5#**. This problem shows the substantial imbalance in material we sometimes see.