Project Title: *Chess Engine*

Team Members: 612303026 Aryan Jotshi

612303039 Yashwant Bhosale

Objectives:

1. This project aims to develop a fully functional Chess Game in the C Programming Language, with all the rules of the game and two game Modes.-

   - Dual Player (Human vs Human)

   - Single Player (Human vs Computer)

2. To develop a simple computer Chess Engine (with algorithms later discussed)
3. Chess Engine should give accurate results, will be tested against the leading Chess Engine in the world, namely **Stockfish**
4. To develop a simple and intuitive, yet appealing User Interface

Key Components:

1) **Game Board Representation:** 8x8 Chessboard using Bitboards
2) **Piece Representation:** representation of different pieces like King, Queen, Bishop, Knight, Rook and Pawn, with their respective parameters and legal moves
3) **Game Mechanics:** Legal moves evaluation based on piece type and rules, move execution and board update
4) **Basic Engine:** to evaluate potential moves and select an optimal one, using depth-search algorithms like Minimax Algorithm with Alpha-Beta Pruning, further optimized using Move Ordering
5) **Terminal based UI:** display board and pieces, input for player's moves (for eg. **a2a4**, it will be parsed as piece at a2 square will move to a4 square, doing capture, if necessary)

Data Structures And Their Relevance:

1) **Bitboards (represented by 64 bits unsigned integers):** Bitboards are used to represent the board inside a chess program in a piece centric manner. They are 64-bit unsigned integers with each bit representing a square on the board.
   To represent the board, we typically need one bitboard for each piece-type and color likely encapsulated inside a structure. A one-bit inside a bitboard implies the existence of a piece of this piece type on a certain square.

   Example:
   if a Bishop is at position a2, its bitboard representation would be "00000000100000000000000000000000000000000000000000000000" (9th bit is set), while if a Rook is at position h3, its bitboard representation would be "00000000000000000000010000000000000000000000000000000000"(24th bit is set)

2) **1-D and 2-D Arrays:** It would be relevant to store an array of bitboards or something similar (will be implemented using built-in functionality)

3) **Stack:** The flow of a chess game is typically represented by a sequence of moves. A stack will be used to maintain this sequence and implement features like undo a move, or resume game.

4) **Hash Tables:** Hash tables (Zobrist) are typically required to store and retrieve data faster than usual searching algorithms. Here, hash tables will be used to store things like previously evaluated positions to speed up the engine.

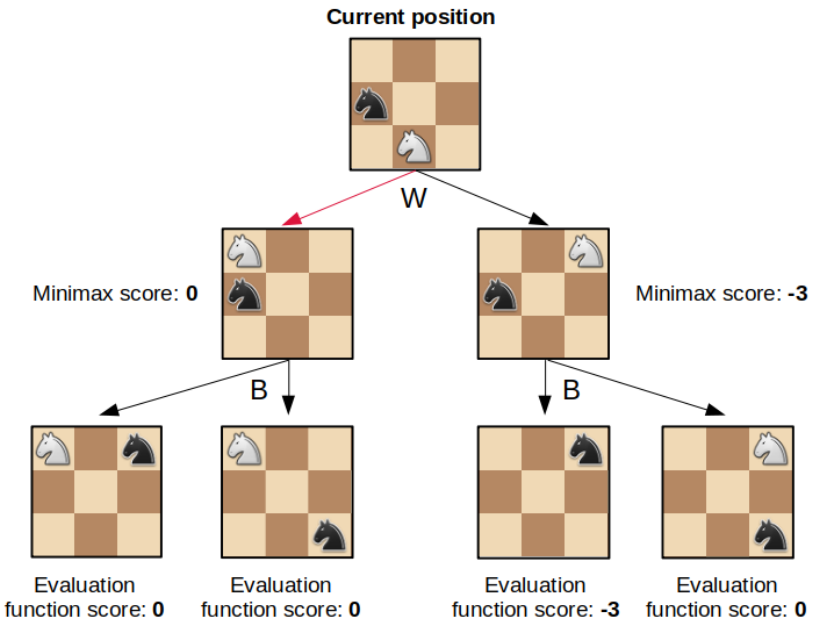
Algorithms And Their Efficiency:

1) **Minimax Algorithm**

The core of Chess playing is Minimax. This algorithm usually associates a Black Piece with a MAX value, and White Piece with MIN value, and always evaluates from the White's point of view.

It utilizes the game tree and includes two player **MIN** and **MAX** and a **Result**, which stores the game state. Both players try to nullify the action of other. MAX tries to maximize the result whereas MIN tries to minimize the result. Both players play alternatively, under the assumption that both are playing optimally. Optimal play means both players are playing as per rule i.e., MIN is minimizing the result and MAX is maximizing the result.

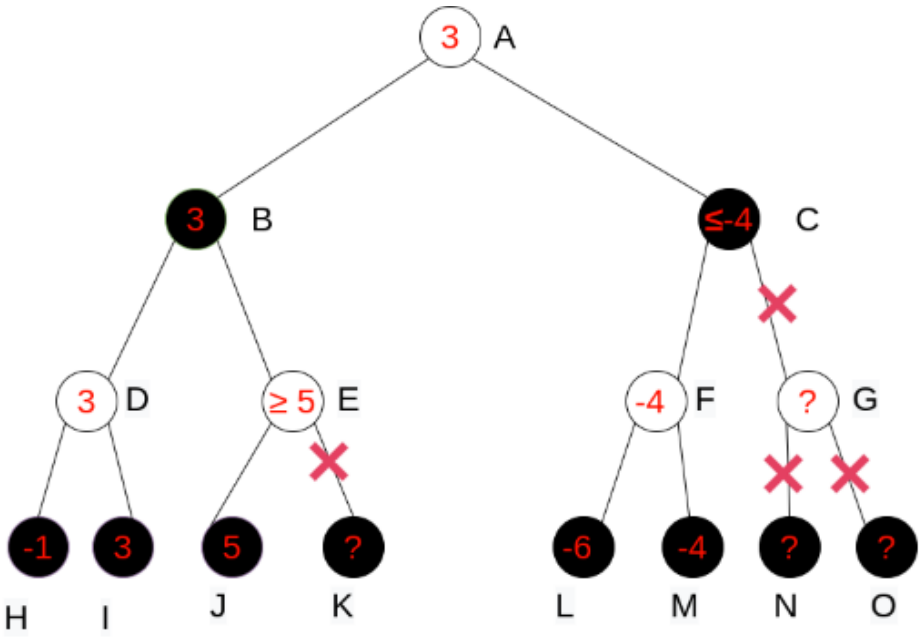
The Algorithm utilizes Depth First Search approach to find the result. Additionally, it also utilizes backtracking and recursion. **Algorithm will traverse till terminal node and then it will backtrack while comparing all child values**. It will select the minimum or maximum value, based on whose turn it is. It will then propagate the value back to their parent. It uses static evaluation function to determine the value at each leaf node.



2) Alpha-Beta Pruning

The Alpha-Beta Pruning method is a significant enhancement to the Minimax Search Algorithm that eliminates the need to search large portions of the game tree, by applying a branch-and-bound technique. Remarkably, it does this without any potential of overlooking a better move.

If one already has found a quite good move and search for alternatives, one refutation is enough to avoid it. No need to look for even stronger refutations. The algorithm maintains two values, **alpha** and **beta**. They



represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively.

3) **Move Ordering**
   If certain moves (like CAPTURES/PROMOTIONS/CHECK) are the legal moves from a certain position then it would be better to **evaluate them first** in the Minimax Tree, since during Alpha-Beta Pruning, these types of nodes will return **better evaluation** and hence other branches will be **pruned earlier**. This optimization decreases time spent on nodes enormously.

4) **Evaluation Function (for legal move generation)**
   An evaluation function is a function used by game-playing computer programs to estimate the value or goodness of a position (usually at a leaf or terminal node) in a game tree. Most of the time, the value is either a real number or a quantized integer, often in n^{th}'s of the value of a playing piece.

   The composition of evaluation functions is determined empirically by inserting a candidate function into an automaton and evaluating its subsequent performance (based on the state of the board, i.e, including the state of the pieces).

   Larger evaluations indicate a piece imbalance or positional advantage or that a winning a piece is usually imminent. Very large evaluations may indicate that checkmate is imminent.

5) **Zobrist Hashing**
   Zobrist hashing is a hash function construction used in computer programs to implement transposition tables, a special kind of hash table that is indexed by a board position and used to avoid analyzing the same position more than once.

   Zobrist hashing starts by randomly generating bitstrings for each possible element of a board game, i.e. for each combination of a piece and a position (in the game of chess, that's 12 pieces × 64 board

positions, or 18 × 64 if kings and rooks that may still castle, and pawns that may capture en passant, are treated separately for both colors).

Now any board configuration can be broken up into independent piece/position components, which are mapped to the random bitstrings generated earlier. The final Zobrist hash is computed by combining those bitstrings using bitwise XOR.

It is also required in checking of some rules like 3-Repetition Draw or the Fifty Move Rule.

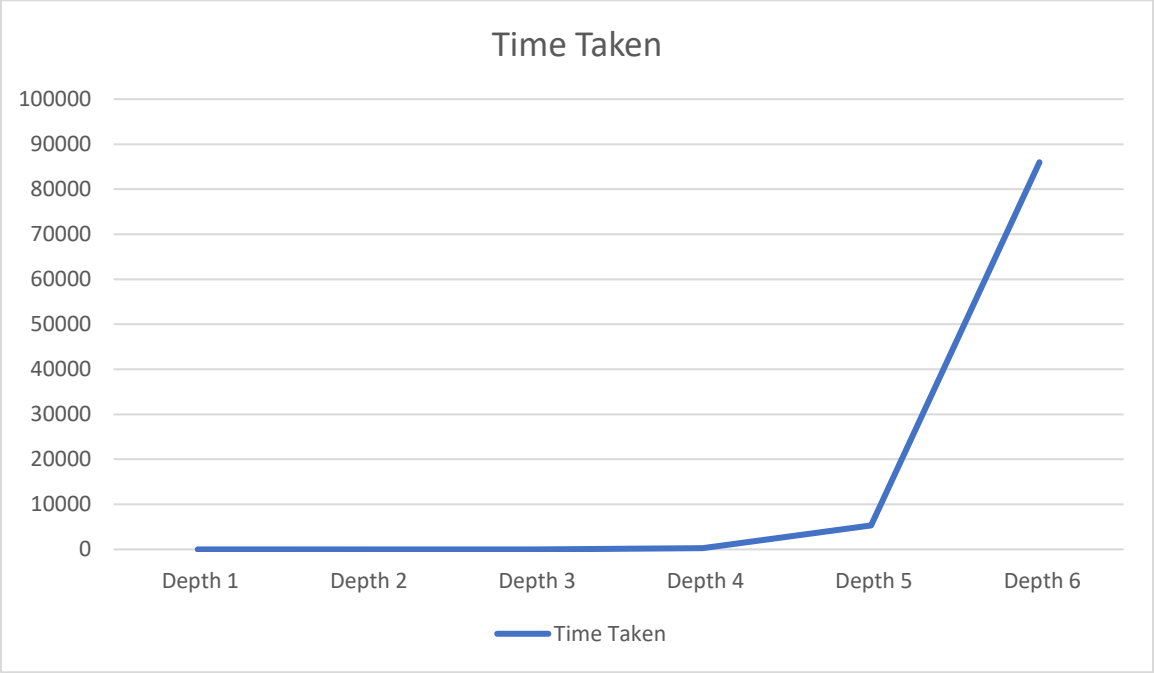
6) **Transposition Tables**

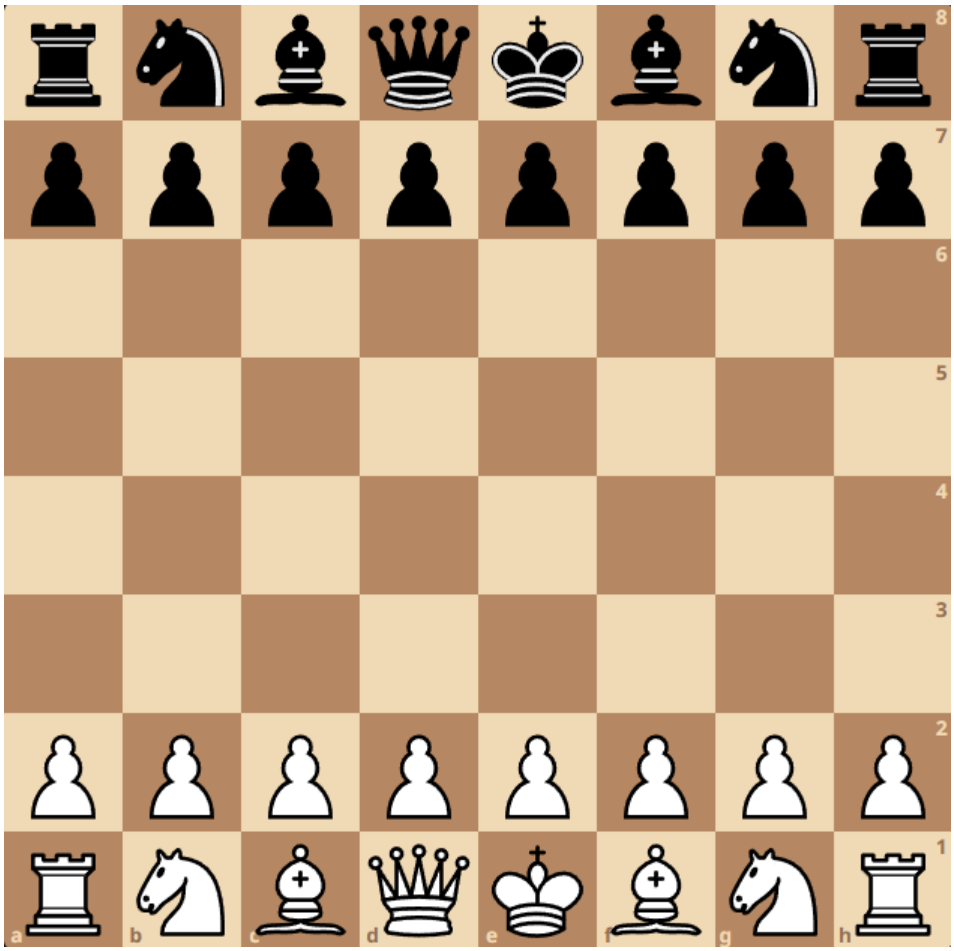
A transposition table is a cache of previously seen positions, and associated evaluations, in a game tree generated by a computer game playing program.

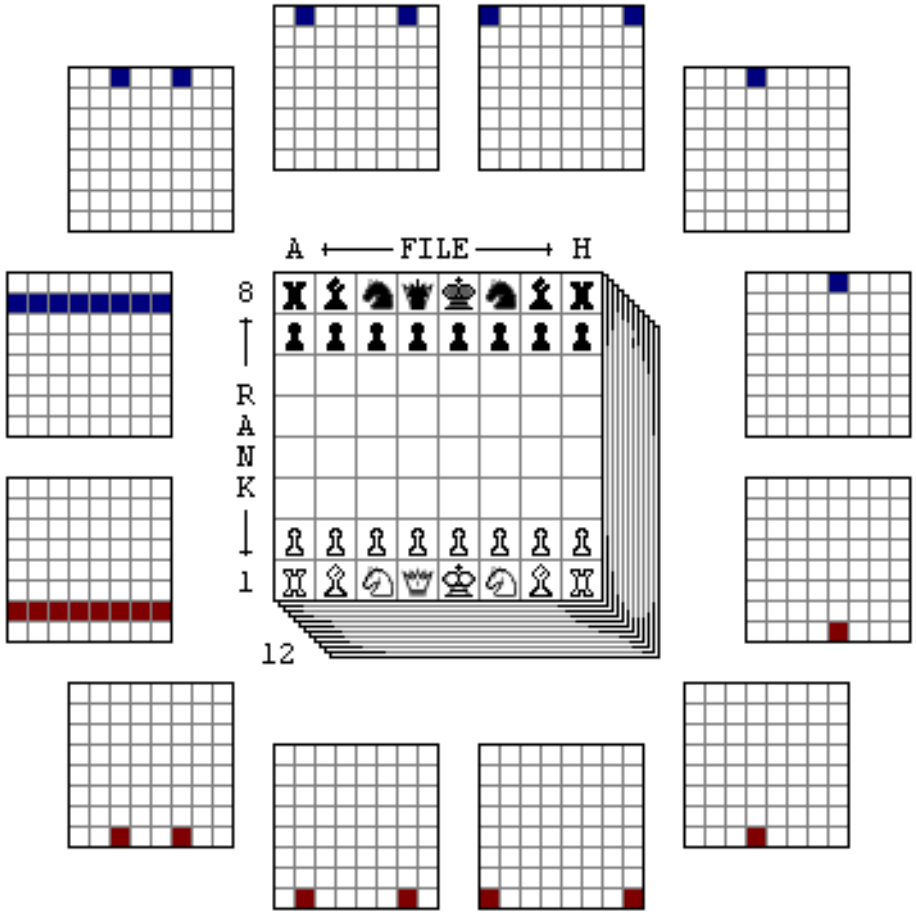
If a position recurs via a different sequence of moves, the value of the position is retrieved from the table, avoiding re-searching the game tree below that position.

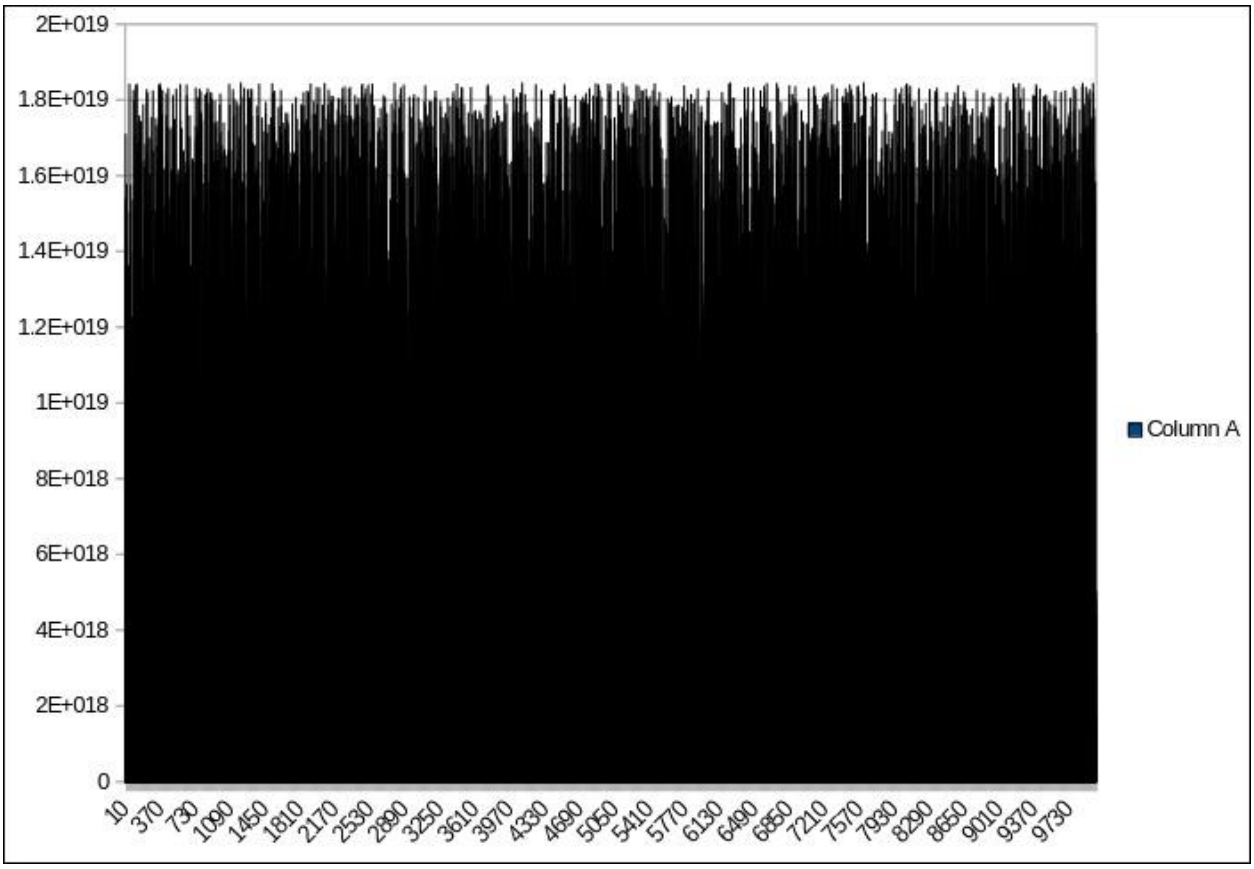
Transposition tables are typically implemented as hash tables encoding the current board position as the hash index. The number of possible positions that may occur in a game tree is an exponential function of depth of search, and can be thousands to millions or even much greater. **Transposition tables may therefore consume most of available system memory and are usually most of the memory footprint of game playing programs.**

Depth	Nodes (Engine)	Nodes (Stockfish)	Time Taken (ms)
1	20	20	0.1
2	400	400	0.92
3	8902	8902	12.48
4	197281	197281	284.3
5	4865609	4865609	5314.16
6	119060324	119060324	85990.06






A ⟵ FILE ⟶ H
8
RANK
1
12



TOC: