Contents

1 Analysis *(9 marks)* 3
1.1 Identification of the problem 3
1.2 Research carried out 4
1.3 Identification of the prospective user(s) 9
1.4 Detailed background to the problem 3
1.5 Numbered measurable, appropriate specific objectives of the project 10
1.6 Modelling diagrams 12
2 Design *(12 marks)* 13
2.1 System design overview 14
2.2 Algorithms 16
2.3 Data structures 16
2.4 File structure and organisation 39
2.5 Database design 4
2.6 SQL queries 4
2.7 User interface design (HCI) 39
2.8 Hardware selection/design 4
3 Technical Solution *(42 marks)* 41
3.1 Code listing 43
4 System testing *(8 marks)* 103
4.1 Test plan 104
4.2 Annotated screenshots of test results 6
5 Evaluation *(4 marks)* 107
5.1 Comparison of project performance against the objectives 107
5.2 Effectiveness of the solution 107
5.3 Analysis of user feedback 107
5.4 Possible improvements 107
6 Appendix 108
6.1 User feedback (authenticated by assessor) 8

1 Analysis *(9 marks)*

1.1 Background to/ Identification of the problem

Chess is an extremely popular board game played globally since ancient times. Two players take turns manoeuvring their pieces around the board in the aim of eventually trapping the opponent's King piece. There are many variations of the rules, including Chess960 and Crazyhouse, and it has been estimated that the number of possible chess games is at least 10^120 (Shannon's number), which is astronomically large.

Screenshot of lichess.com, a popular chess website

In the last few years, the explosion of the digital world due to the pandemic has led to a surge of interest in online chess, as it is much more convenient than having to physically setup a board. Online content creators have generated a huge amount of interest, for example GMHikaru, a 5-time US chess champion, has 1.44 million subscribers on YouTube as of 5 Dec 2022.

For the complete rules of chess, see `fide.com/FIDE/handbook/LawsOfChess.pdf` .

1.1.1 Existing solutions

Most existing solutions, for example lichess.com and chess.com, have a GUI in which the user drags pieces with a mouse to move them. They also have user accounts in which past games can be revisited. However, these GUIs are separate from the chess algorithms which evaluate moves. For example, lichess uses a popular powerful engine, Stockfish, to generate good moves from a position. Because of the difficulty in creating a good chess engine, most existing solutions (e.g. chessmultiplayer.com) do not offer viable chess playing algorithms and only offer an interface. In particular, the computer algorithm on chessmultiplayer.com plays close to random after the first few moves.

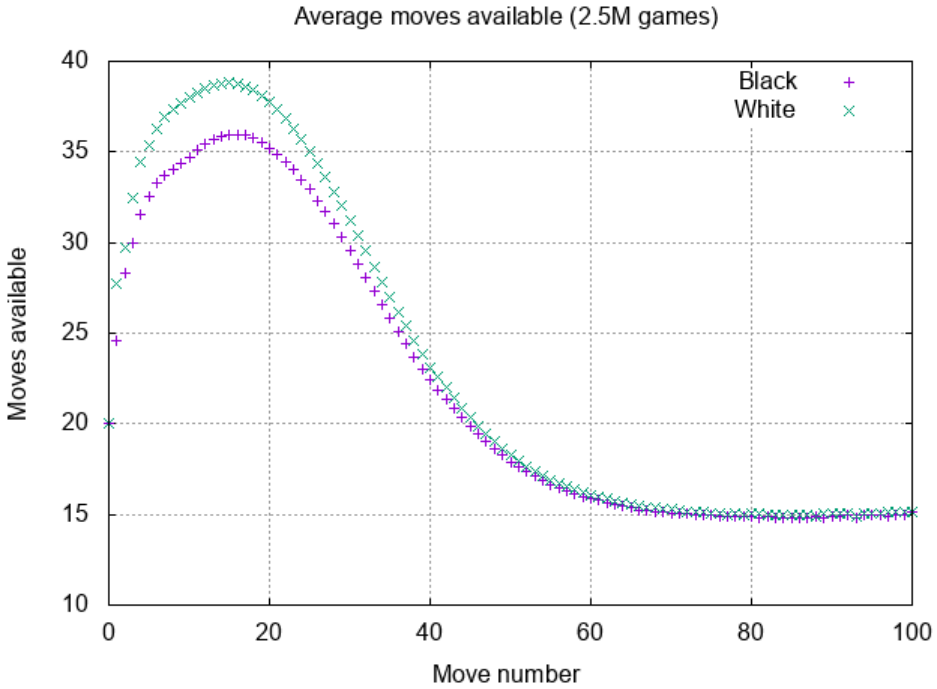
The most popular chess-playing algorithms are Minimax and Monte-Carlo Tree Search (MCTS). Of the available chess engines online, a significant proportion of them use Minimax (see section 1.2.2). These include Stockfish, Rybka, Crafty and Fritz. There are only a few that use MCTS (see section 1.2.3), including the recent Leela Chess Zero (Lc0) released in 2018 – before Lc0, most if not all engines used Minimax. The recent success of MCTS has been due to the huge developments in neural networks in the last 10 years, as MCTS and neural networks have been combined to create formidable algorithms. For example, in 2016 Google DeepMind used this strategy to create the AlphaGo program to play the game of Go, which is a perfect-information strategy board game like chess. AlphaGo beat the world champion Lee Sedol, being the first computer Go program to beat the top human player. Just as MCTS allowed computers to surpass humans at Go, Minimax did the same for chess, when in 1997 Deep Blue famously beat the reigning world champion Garry Kasparov.

In conclusion, there are currently an abundance of solutions which offer a graphical interface and user accounts. Most programs only offer a user-interface and not an engine, due to the difficulty of writing a good chess engine. Among the chess engines themselves, most use Minimax because of its well-established success, and only a small proportion use MCTS, although this proportion may increase with the introduction of neural networks.

1.2 Research carried out

1.2.1 Minimax

A common approach to creating a chess algorithm is to search breadth-first through possible moves in the game tree and play the move that results in the “best” position found so far, defined by an evaluation function that takes in a position and returns a score, where positive means better for white and vice versa.

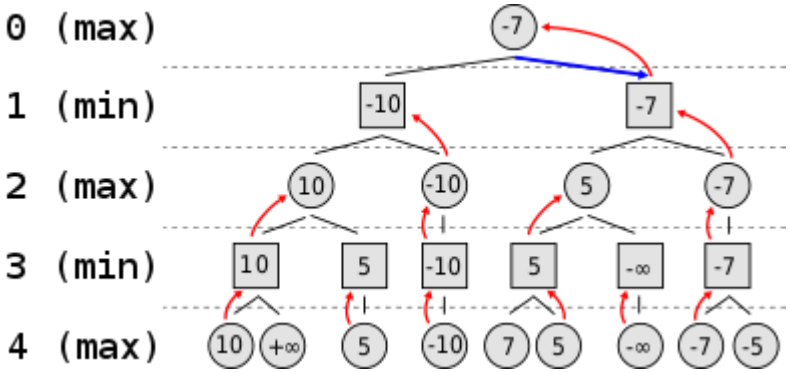


https://chess.stackexchange.com/questions/23135/what-is-the-average-number-of-legal-moves-per-turn

With this breadth-first strategy, it is infeasible to explore every possible move without the use of heuristics. In the graph above, for a lower bound, there are at least 15 legal moves available in a given position, and so searching to a depth of 6 moves will give at least 15^(6*2) ~= 10^14 possibilities, which is already too large for a program to keep track of.

And so, heuristic evaluation functions are needed. For example, a naïve function could sum the values of the player's pieces and subtract the sum of values of the opponent's pieces. However, this ignores information about the positions of the pieces, for example the material might be equal but one side controls more squares, which is desirable since it restricts the move options of the opponent. Thus, a good evaluation function will also need to consider the positions of the pieces.

The most common breadth-first strategy is Minimax (see section 2.3.3). Starting from a fixed depth in the game tree, the tree is iterated through backwards to ask the question "if this was the current position, how could I give my opponent the worst possible position, given what I know so far"? This involves trying to maximize/minimize the output of the evaluation function depending on whether it is white's or black's turn.



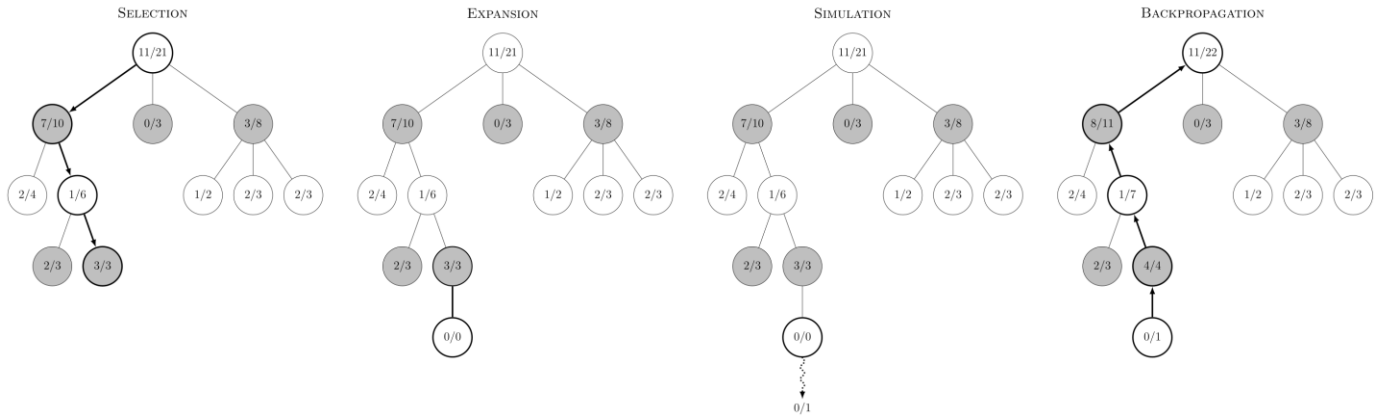
800px-Minimax.svg.png

In the example above, from an initial depth of 4, each node is considered in turn. For the leftmost pair of nodes, the evaluation function returns 10 and infinity. Thus if it was Black's turn to move at their parent node, Black would want to minimize the value and thus would play the move that results in the position worth 10 rather than infinity. So the value of the parent node is assigned as 10. This is done for every node at depth 4, then the same is repeated for every node at depth 3, etc. Eventually, the root node is reached, revealing the optimal move to play. In this example, the values of the root children have been assigned as –10 and –7 respectively. The root node is trying to maximize the score since they are white, thus white should play the move that results in the score of –7.

Note that a constraint of the Minimax algorithm is the need for a specific starting depth. For example, during a chess match where each player is only allowed one second per move, it is unclear what starting depth to choose – finishing the search with extra time left is suboptimal as there would be spare time that could have been spent searching moves, however not finishing the search at all means good moves could have been missed. Luckily, this problem can be alleviated with Iterative Deepening, where the depth is incremented until the time runs out, at which point the result of the last completed search is used. In fact, this can paradoxically even speed up the search, because the results of the last completed depth can be used to give an idea of good moves to explore at the start of the next search, which can then cut computation time significantly for the next depth, because alpha beta has a better opportunity to prune branches of the tree.

1.2.2 Monte-Carlo Tree Search

There is another search algorithm that does not need an evaluation function. Instead, Monte-Carlo Tree Search (MCTS, see section 2.3.5) plays out move possibilities by choosing random moves until a loss/win/draw occurs, doing this many times to create an "average win rate from this position" statistic. This is used to guide which nodes of the game tree to explore more – for example, a move that has a high average win rate is a good move, and so the possible moves from that position should be explored more than one which has a low win rate. However, low win rates should still be explored as there could be hidden tactics, for example a move that worsens your position but forces checkmate on the next move. This depth-first, average approach is similar to that of the human brain, where only a few "intuitively feasible" move options are explored, but to a great depth. Thus, it is better than minimax for games with a high number of legal moves per position such as Go (where stones are placed at grid intersections on a 19x19 board, thus there may be up to ~300 legal moves in a position), as minimax would only be able to look a few moves into the future. However, MCTS is bad at detecting "shallow traps" and may miss good moves entirely.



[2880px-MCTS-steps.svg.png]

The figure above demonstrates the 4 stages of MCTS: selection, expansion, simulation, and backpropagation.

Selection: start at the root node and travel down the game tree until reaching a leaf, choosing which nodes to travel to based on their current win rate (moves with a higher win rate should be more likely to be travelled to).

Expansion: create some child nodes from the leaf (i.e. play some possible moves), and travel to one of them.

Simulation: play random moves until a win/loss/draw is reached.

Backpropagation: use the information to update the win rate of all the nodes travelled to from the root.

This overcomes the time control problem as it does not matter when the search stops, just the move with the highest win rate so far is chosen. However, with extremely low time per move, MCTS plays close to random, as the law of large numbers does not yet apply to stabilize the game tree.

The differences between Minimax and MCTS are summarized in a table below.

	Minimax	MCTS
Heuristic evaluation function	Needed to evaluate the nodes rather than playing every possibility to the end which would be infeasible	Not needed since games are played to the end, but by choosing random moves instead of considering every possibility, and it is played to the end many

		times to build a “win-rate” statistic
Time control	Iterative deepening is used which achieves the same result of MCTS, but the search increases in discrete jumps as the depth is incremented, so there may be some wasted time where the next depth is being explored but must be discarded as it is incomplete	Not affected – just stop the computation when the time limit is reached, all of the time is used efficiently
Computation time	Mostly used for creating/copying game states and evaluating positions	Mostly used for playing out the random moves: faster move generation means significantly faster searching
Low time per move	Performs better than MCTS	Performs very poorly, close to random

1.2.3 Fast Move Generation

For efficient storage of board positions and fast move generation, a popular technique is to use bitboards. For each piece type, a 64-bit integer is stored that represents the squares of the chessboard that contain that piece, where the bits map to the squares. For example, an arrangement of white pawns can be stored like so:

```
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 1 0 0 0 0 0
0 0 0 1 0 0 0 0
1 1 0 0 1 1 1 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0

0000000000000000111001110000100000010000000000000000000000000000
(= 508043628380160 in base 10)
```

Where 1 means a white pawn is present on that square. The bits are read off left to right then bottom to top, from least significant bit to most significant bit.

This allows for efficient move generation using bitwise operations, for example to generate the possible moves for the pawns above, the bitboard can simply be bitwise shifted left by 8, since this shifts the pawns up by one row. However there may be slight adjustments needed, such as for the “en passant” rule. The use of bitboards in this project is discussed in detail in section 2.3.1.

1.3 Identification of the prospective user(s)

Primary User interview

Interviewee: Robert Earle
Online meeting started: 05/03/22

Thank you for meeting with me today. This is about the chess program you're interested in. Have you played chess against a computer before?

Yeah, I've been playing against AIs for a couple years now. They can be fun when I don’t have time to play with my friends or close family.

Are there any problems you have with these AIs?

Most of the time, I feel like they don’t play like human beings and can never shake the feeling that they can see every possibility and I have no chance. I guess this is due to the brute-force nature of chess engines though.

I see. Yes, most chess engines use the Minimax algorithm to search all the move possibilities. Have you heard of Monte-Carlo Tree Search?

I have actually, didn’t Google use that to beat Lee Sedol at Go? By using a neural network, they could mimic the human brain to play more intuitively instead of searching every move.

Yep! They used it in Go because Minimax was too slow. Have you ever thought about using MCTS in a chess program?

Oh, I haven’t, that would be very interesting – maybe the AI would play more human moves? I would really like to see that. My intuition is that Minimax would be better at chess since all the top engines use it, but if there was an MCTS chess engine then I would enjoy playing against it as a substitute for my friends. I’d like to see how the two algorithms perform against each other, as well.

If I was to create a chess engine program with Minimax and MCTS options, would you highly value a GUI over a terminal interface?

Not really, I feel that for engines I enjoy the terminal interface more as it feels barebones and efficient – I'd like to able to run commands though. I guess I could always input the generated engine moves into an online GUI if I wanted.

I see. Thank you for your time today.

The primary user knows the rules of chess and wishes to be able to play chess against an AI, so that they can compare Monte-Carlo Tree Search and Minimax when applied to a perfect information game like chess.

As mentioned in section 1.1.1, currently not many chess engines use MCTS, and barely any use it without neural networks. This project aims to present a program to investigate this, and determine whether, for the same computational resources available per move, Minimax performs much better than MCTS. Furthermore, a mixture of the two algorithms is implemented.

A bespoke program will be presented to the primary user that plays chess using both algorithms, so that they can determine for themselves the upsides and downsides of each algorithm, since they mentioned in the interview that this was something they would want. The project objectives (see section 1.4) have been influenced by the primary user in the following way:

- They want a chess-playing program; objectives M1 to M5
- They want to be able to play against Minimax; objective M7
- They want to be able to play against MCTS; objective M6
- They want to be able to watch the two algorithms play each other; objective M8
- They want to be able to run commands in a terminal interface; objective M9
- They do not highly value a GUI; a GUI is not in the objectives.

1.4 Numbered measurable, appropriate specific objectives of the project

Minimum Viable Product

M1 A chess board containing the correct current pieces should be displayed to the user

M2 The user should be able to input their next move
M2.1 The user should be able to input their move in the form “start square – end square” by rank and file reference
M2.2 If promoting (see M3.1.5), the user should be able to choose which piece to promote to

M3 The program should be able generate legal moves correctly:
M3.1 Pawn moves
M3.1.1 (Pawn pushes) Pawns should be able to move one square forward, unless a piece exists on the target square

M3.1.2 (Double pawn pushes) On their first move, pawns should be able to move two squares forward, unless a piece exists on the target square
M3.1.3 (Pawn captures) Pawns should be able to capture by moving to their forward-left or forward-right neighbours, if that square contains an opponent piece
M3.1.4 (En passant) If a pawn attacks a square crossed by an opponent double pawn push on the last move, then that pawn may capture the opponent's pawn as if it had only moved one square forward
M3.1.5 (Promotion) If a pawn moves to the rank furthest from its starting rank, it should be replaced with a knight, bishop, rook of queen of the same colour
M3.2 (Knight moves) Knights should move two squares in any direction and one square in an orthogonal one
M3.3 (Bishop moves) Bishops should move to any square on a same diagonal as the starting square
M3.4 (Rook moves) Rooks should move to any square on the same file or rank
M3.5 (Queen moves) A queen should move to any square accessible as if a rook or bishop was on the same starting square
M3.6 King moves
M3.6.1 The king should be able to move to any of the 8 squares adjacent to the starting square, provided they are not attacked by an enemy piece
M3.6.2 (Castling) If the king has not moved, and a rook of the same colour has not moved or been captured, and the squares between them (including the king but not the rook) are not attacked by an opponent piece, then the king should be able to move two squares towards the rook and the rook should then be transferred to the square that the king has just crossed over
M3.6.3 (Check) If the king is under threat of being captured next move, then any move that does not prevent this capture should be classed as illegal
M3.7 Edge cases
M3.7.1 A piece should not be able to capture pieces of the same colour
M3.7.2 If moving a piece allows the king to be captured on the next move, it should not be allowed ("absolute pins")

M4 The program should determine whether the user's move is legal, and play it if so

M5 The game should be able to be won, lost or drawn
M5.1 (Checkmate - win/loss) If the side to move has no legal moves and their king is in check, then the game should end, with them losing
M5.2 (Stalemate - draw) If the side to move has no legal moves and is not in check, then the game should end as a draw

M6 The program should use the 4-stage MCTS algorithm to generate a candidate move for an AI
M7 The program should use Minimax to find the best move for an AI at a given depth
M8 The program should be able to have the two AIs play against each other
M9 The user should be able to run commands in a CLI to start new games or debug positions

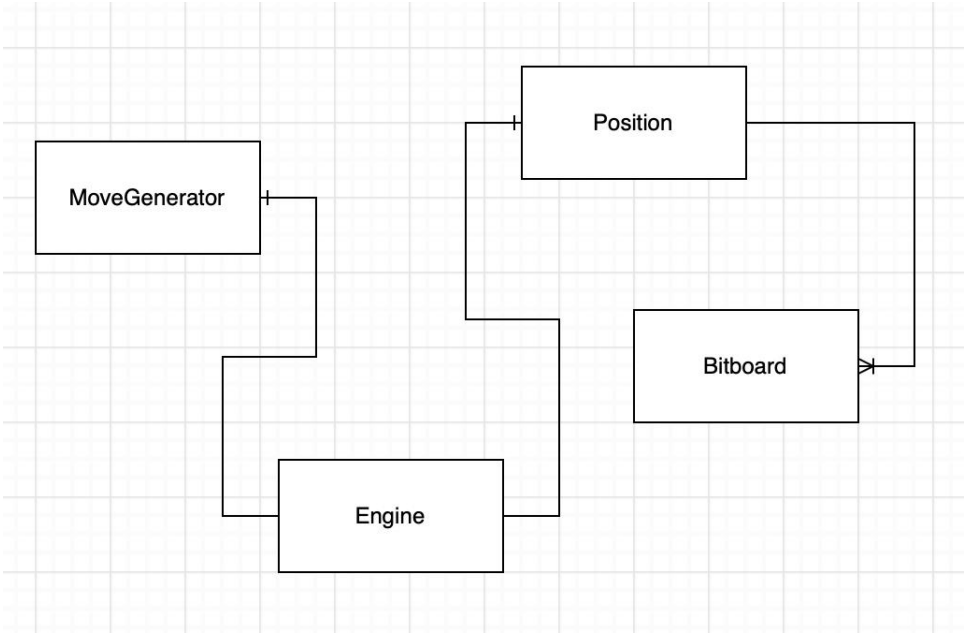
Extra objectives

E1 The user should be able to load different positions
E2 The AI should be able to win against a novice opponent
E3 The program should offer a combination of Minimax and MCTS to generate candidate moves
E4 A transposition table should be stored during Minimax search, for optimization
E4.1 Zobrist hashing should be implemented as the hash function for the table
E4.2 The minimax algorithm should use the transposition table to perform Iterative Deepening
E5 The user should be able to play in two-player mode

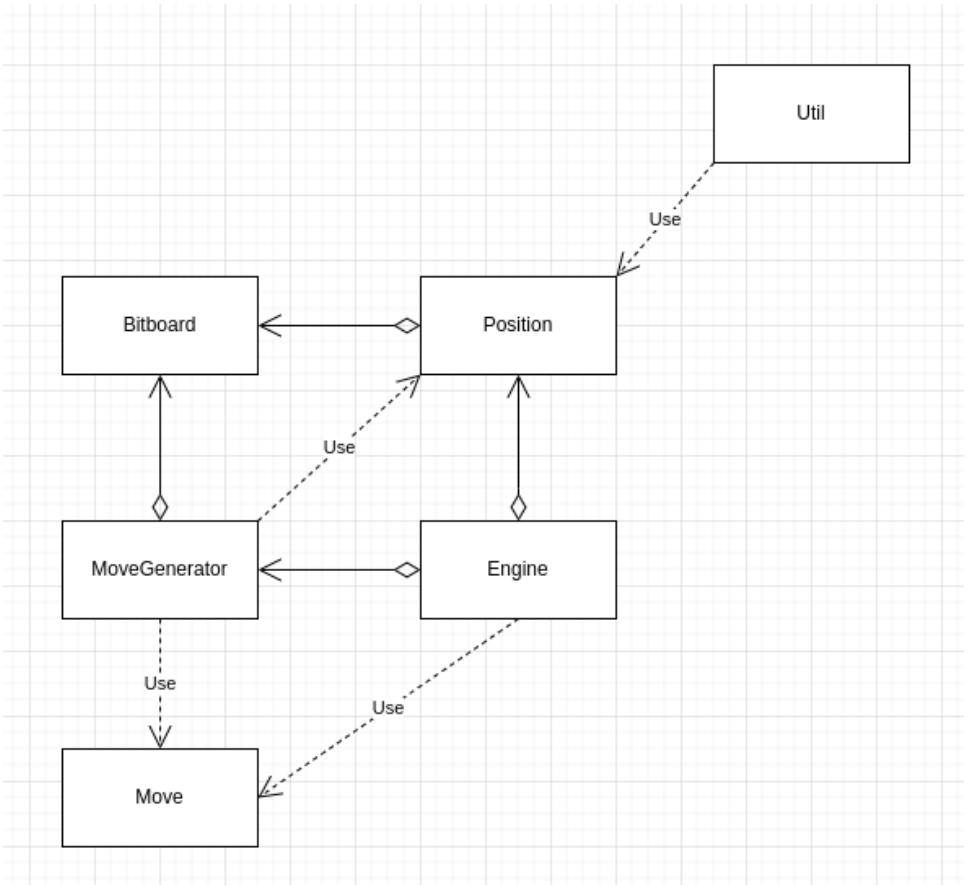
E6 The AI should play its move after searching for a given amount of time set by the user

1.5 Modelling diagrams

Entity Relationship Diagram



UML Diagram

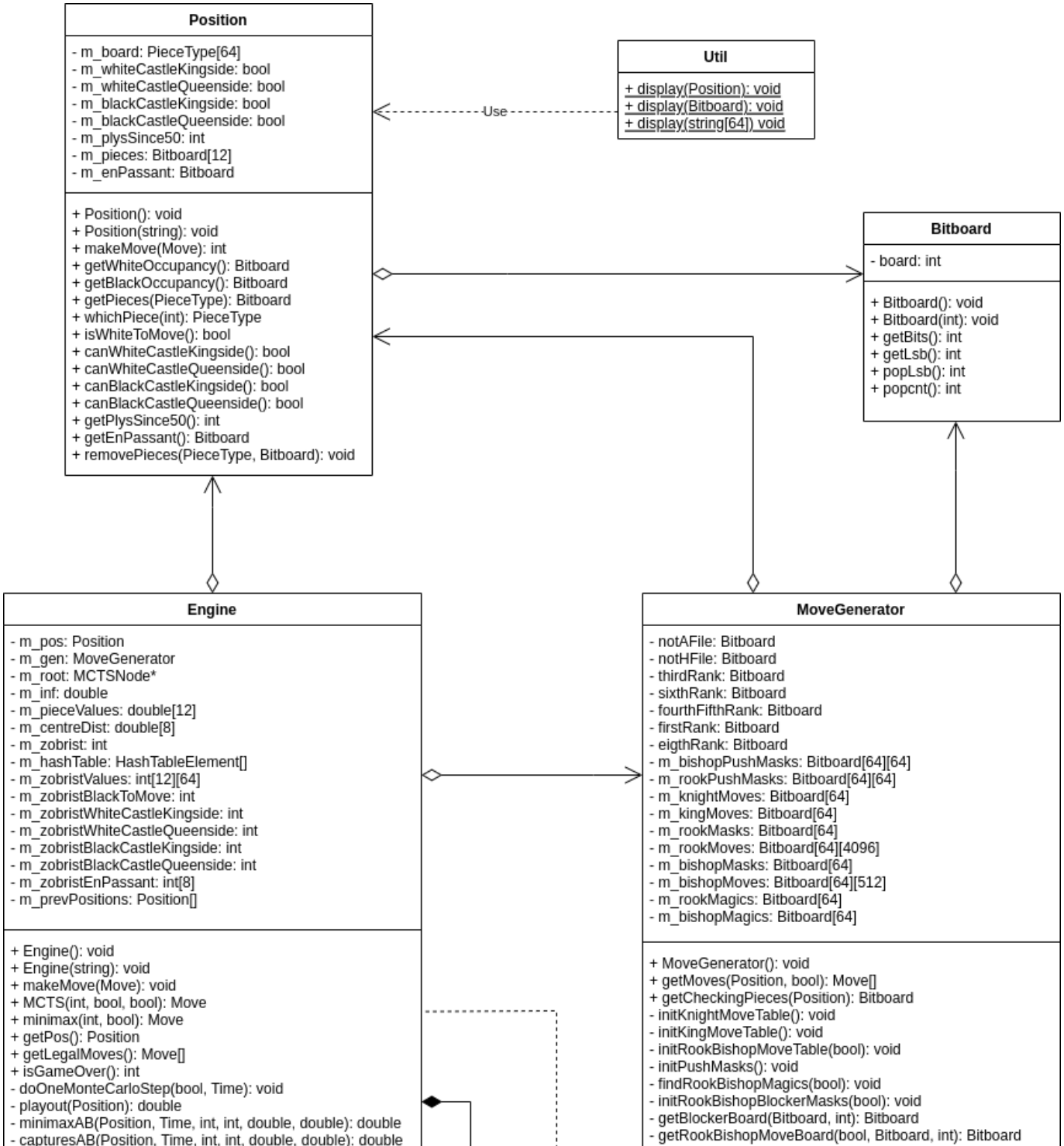


The `Util` class will consist of static help methods, for example to display a `Position` to the terminal.
A `Position` consists of multiple `Bitboard`s which describe the locations of all the pieces, as well as meta information about the position such as castling rights. The `MoveGenerator` class will contain methods that in general, take in a position and output the legal moves from that position. It references the `Move` class which represents a move by encoding the start square, end square, piece type and various helper flags e.g. whether the move was en passant.
The `Engine` class will run MCTS and minimax on a position.

2 Design (12 marks)

2.1 System design overview

UML Diagram


Position
- m_board: PieceType[64]
- m_whiteCastleKingside: bool
- m_whiteCastleQueenside: bool
- m_blackCastleKingside: bool
- m_blackCastleQueenside: bool
- m_plysSince50: int
- m_pieces: Bitboard[12]
- m_enPassant: Bitboard
+ Position(): void
+ Position(string): void
+ makeMove(Move): int
+ getWhiteOccupancy(): Bitboard
+ getBlackOccupancy(): Bitboard
+ getPieces(PieceType): Bitboard
+ whichPiece(int): PieceType
+ isWhiteToMove(): bool
+ canWhiteCastleKingside(): bool
+ canWhiteCastleQueenside(): bool
+ canBlackCastleKingside(): bool
+ canBlackCastleQueenside(): bool
+ getPlysSince50(): int
+ getEnPassant(): Bitboard
+ removePieces(PieceType, Bitboard): void
Util
+ display(Position): void
+ display(Bitboard): void
+ display(string[64]) void
Use
Bitboard
- board: int
+ Bitboard(): void
+ Bitboard(int): void
+ getBits(): int
+ getLsb(): int
+ popLsb(): int
+ popcnt(): int
Engine
- m_pos: Position
- m_gen: MoveGenerator
- m_root: MCTSNode*
- m_inf: double
- m_pieceValues: double[12]
- m_centreDist: double[8]
- m_zobrist: int
- m_hashTable: HashTableElement[]
- m_zobristValues: int[12][64]
- m_zobristBlackToMove: int
- m_zobristWhiteCastleKingside: int
- m_zobristWhiteCastleQueenside: int
- m_zobristBlackCastleKingside: int
- m_zobristBlackCastleQueenside: int
- m_zobristEnPassant: int[8]
- m_prevPositions: Position[]
+ Engine(): void
+ Engine(string): void
+ makeMove(Move): void
+ MCTS(int, bool, bool): Move
+ minimax(int, bool): Move
+ getPos(): Position
+ getLegalMoves(): Move[]
+ isGameOver(): int
- doOneMonteCarloStep(bool, Time): void
- playout(Position): double
- minimaxAB(Position, Time, int, int, double, double): double
- capturesAB(Position, Time, int, int, double, double): double
MoveGenerator
- notAFile: Bitboard
- notHFile: Bitboard
- thirdRank: Bitboard
- sixthRank: Bitboard
- fourthFifthRank: Bitboard
- firstRank: Bitboard
- eigthRank: Bitboard
- m_bishopPushMasks: Bitboard[64][64]
- m_rookPushMasks: Bitboard[64][64]
- m_knightMoves: Bitboard[64]
- m_kingMoves: Bitboard[64]
- m_rookMasks: Bitboard[64]
- m_rookMoves: Bitboard[64][4096]
- m_bishopMasks: Bitboard[64]
- m_bishopMoves: Bitboard[64][512]
- m_rookMagics: Bitboard[64]
- m_bishopMagics: Bitboard[64]
+ MoveGenerator(): void
+ getMoves(Position, bool): Move[]
+ getCheckingPieces(Position): Bitboard
- initKnightMoveTable(): void
- initKingMoveTable(): void
- initRookBishopMoveTable(bool): void
- initPushMasks(): void
- findRookBishopMagics(bool): void
- initRookBishopBlockerMasks(bool): void
- getBlockerBoard(Bitboard, int): Bitboard
- getRookBishopMoveBoard(bool, Bitboard, int): Bitboard

The above UML diagram shows the class relationship for the program.

The `Bitboard` class is a wrapper for a 64-bit integer that stores the bits of the bitboard. For example, the variable `notAFile` in the `MoveGenerator` class is a constant bitboard that stores all bits that are not on the A file (leftmost column of the board). As a hexadecimal integer, this is the bitwise NOT (~) of just those bits that are the A file, so `notAFile` is equal to `~0x0101010101010101` because the least significant bit is the bottom-left square of the board.
Utility methods are provided for getting/setting the bits, swapping two bits and popping (removing and returning) the least significant bit, which is useful when iterating through the bits of the bitboard that are equal to 1.

To store a position in an instance of the `Position` class, 12 bitboards are stored for each piece-colour pair (for example white pawns, black queens). There are 6 piece types (pawn, knight, bishop, rook, queek, king) and 2 colours (black, white), so 6*2 = 12 bitboards are needed. These are stored in the `m_pieces` array. For fast lookup of which piece is on a given square, the `m_board` array keeps track of this. Two helper bitboards also store the white and black occupancy, that is, all squares of the board occupied by a white piece. This is useful when calculating legal moves, because for example, a bitboard of possibly legal white moves can be bitwise AND-ed with NOT(whiteOccupancy) to remove all the moves that move a white piece onto a square containing another white piece, because you cannot capture your own piece.
The `makeMove` function takes in a start square and end square, and modifies the required bitboards.

A static display method in the `Util` class is used to display a bitboard (or a board of ASCII text symbols) to the terminal.

The `Move` class is a wrapper for six variables, consisting of a start square, end square, piece type, and three booleans for castling, promotion and en passant.

The `MoveGenerator` class is responsible for calculating possible moves from a given position. The move generation is slightly different for each type of piece, and each implementation is covered in more detail in section 2.3. In general, static lookup tables are generated at program start, then used for fast constant-time move generation, instead of calculating the possible moves on the fly. This is a worthwhile optimization because as mentioned in section 1.1, fast move generation can significantly speed up Monte-Carlo Tree Search.
There are private methods for calculating moves for each type of piece, then the public-facing `genMoves()` method calls each of these and compiles them into a list of possible moves.

The `Engine` class stores the current game position and is responsible for performing MCTS and minimax. It thus stores the variables it needs to do this, such as `m_zobrist` which stores the zobrist hash (see section 2.3.3) of the current position. The public facing AI functions are `MCTS()` and `minimax()`.

2.2 Data structures

Hash Tables

- The rook and bishop move tables (`m_rookMoves` and `m_bishopMoves` in the `MoveGenerator` class), which store possible moves from a given position and arrangement of pieces. Hash function: the arrangement of pieces is hashed using magic bitboards (see section 2.3.1).

- The transposition table (`m_hashTable` in the `Engine` class), which stores evaluated positions so that the minimax algorithm can look them up later, to avoid repeating work. Hash function: the current position is hashed with zobrist hashing (see section 2.3.3).

Trees
- The Monte-Carlo Tree Search game tree (`m_root` in the `Engine` class): implemented as a doubly linked list where each node has multiple child nodes, stored as a list of pointers. A pointer to the root is stored.

Lists
- `m_prevPositions` in the `Engine` class, which stores all the previous moves played.
- `getLegalMoves()` in the `Engine` class (which calls `gen_moves` in the `MoveGenerator` class) returns a list of `Move` objects that are legal moves.
- `children` in the `MCTSNode` class is a list of pointers to child nodes.

Arrays
- `m_board` in the `Position` class, which stores the current piece on each of the 64 squares.
- `m_pieces` in the `Position` class, which stores a bitboard of positions for each piece type.
- `m_pieceValues` in the `Engine` class, which stores constants for the value of each piece type. The values used are 1 for pawns, 3 for knights and bishops, 5 for rooks, 9 for queens.
- `m_centreDist` in the `Engine` class, which stores the distance to the edge for each file/rank, for convenience so that it does not have to be constantly recalculated during position evaluation (which is where it is used).
- `m_zobristValues` and `m_zobristEnPassant` in the `Engine` class, which store many pseudorandom numbers to be used in zobrist hashing (see section 2.3.3).
- `m_rookPushMasks` and `m_bishopPushMasks` in the `MoveGenerator` class, which are 2D arrays storing bitboards of the squares between two given squares (I.e. reachable by rook/bishop move from both squares), if any. The 2D-index is the two squares (they are symmetric, because the squares between A and B are the same as the squares between B and D),
- `m_knightMoves`, `m_kingMoves`, `m_rookMasks` and `m_bishopMasks` in the `MoveGenerator` class, which store possible corresponding piece moves from a given starting square, assuming the rest of the board is empty.

2.3 Algorithms

2.3.1 Pseudo-Legal Move Generation

Lookup tables are used to optimize the speed of move generation. For example, the possible knight moves from a given square are always the same, so we can simply store these possibilities in a lookup table of size 64 (an entry for each square) and look these up with a constant time. Rook and bishop moves are more complicated as their possible moves depend on the arrangement of other pieces on the board, so to create a lookup table for these pieces, magic numbers are used.

Knight moves
At program initialization, a knight move lookup table is created inside the `MoveGenerator` class. Then it is populated with the possible moves from each square.

```
// generate knight move lookup table
void Movegen::initKnightMoveTable() {
  // knight moves
  int dx[] = {2, 2, -2, -2, 1, 1, -1, -1};
  int dy[] = {1, -1, 1, -1, 2, -2, 2, -2};
  // for each square
  for(int i=0; i<64; ++i) {
    m_knightMoves[i] = 0;
    for(int j=0; j<8; ++j) {
      // add (dx,dy) to position then check if valid
      int x = (int)(i&7) + dx[j];
      int y = (int)(i>>3) + dy[j];
      if(0<=x && 0<=y && x<64 && y<64) m_knightMoves[i] |= 1ull<<(8*y+x); // set the bit at (x,y) to 1
    }
  }
}
```

Then, during the running of the engine, to look up the possible knight moves from a given square, the `m_knightMoves` table is looked up.

Rook moves
A rook, unlike a knight, can move any number of squares (horizontally or vertically), but cannot move past another piece, and so the possible moves for a given square also depends on the arrangement of other pieces on the board.
It may then seem infeasible to create a lookup table, since there are 63 other squares and so <= 2^63 possible arrangements of other pieces, which is too large to store. However, the only pieces that affect the possible moves of a rook are the squares on the same row or column. Also, pieces on the edge of the board do not restrict the movement of the rook, because they can still be taken. So this is at most 12 possible affecting squares that a “blocking piece” could be on (achieved when the rook is on a1). And thus we need to store at most 2^12 = 4096 items per square in the lookup table, for each possible arrangement of blocking pieces (“blockers”). This is well within memory constraints.

First, to define some helpful terms:
A **blocker mask** is all the possible positions of a piece that could restrict the movement (a blocker) of a rook on a given square. There is one blocker mask per square.

A **blocker board** is one specific arrangement of blockers. It is a subset of the blocker mask, and equal to the occupancy bitboard, bitwise AND-ed with the blocker mask to extract the relevant bits. There are many blocker boards per blocker mask.

A **move board** is the resulting possible moves from a blocker board, including possible moves that capture your own piece, but the move board can later be bitwise ANDed with NOT(our occupancy) to fix this. There is one move board per blocker board.

```
/* Example, Rook on e4:
 *
 *    The blocker mask        A blocker board         The move board
```

```
 *     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 1 0 0 0          0 0 0 0 1 0 0 0          0 0 0 0 0 0 0 0
 *     0 0 0 0 1 0 0 0          0 0 0 0 0 0 0 0          0 0 0 0 0 0 0 0
 *     0 0 0 0 1 0 0 0          0 0 0 0 1 0 0 0          0 0 0 0 1 0 0 0
 *     0 1 1 1 0 1 1 0          0 1 1 0 0 0 0 0          0 0 1 1 0 1 1 1
 *     0 0 0 0 1 0 0 0          0 0 0 0 0 0 0 0          0 0 0 0 1 0 0 0
 *     0 0 0 0 1 0 0 0          0 0 0 0 1 0 0 0          0 0 0 0 1 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
 */
```

The blocker boards are the keys of the move table, and the move boards are the data. To convert blocker boards into a convenient index form from 0 to 4095, magic numbers are used: for each square, we find a “magic” number x by trial and error such that the operation of multiplying a blocker board by x and taking the first 12 bits is a valid hashing operation, i.e. two different move boards are never referred to by the same index. Then the correct move boards are placed at the needed indices in the table.

To find magic numbers, we first generate all the blocker masks and store them in `m_rookMasks`:

```
// generate blocker masks, for rooks or bishops
void Movegen::initRookBlockerMasks() {
  // for each square
  for(int i=0; i<64; ++i) {
    m_rookMasks[i] = 0;

    int rookXChanges[] = {1, -1, 0, 0};
    int rookYChanges[] = {0, 0, 1, -1};
    for(int k=0; k<4; ++k) {
      // move in direction specified by k, until arriving at the board edge
      int x = i&7;
      int y = i>>3;
      int dx = rookXChanges[k];
      int dy = rookYChanges[k];
      while(true) {
        x+=dx;
        y+=dy;
        if(dy==0 && (x<1||x>6) || dx==0 && (y<1||y>6) ) ) break; //still possible to be on edge of board with valid moves
        // set the bit at position (x,y)
        int index = y*8 + x;
        m_rookMasks[i] |= (1ull<<index);
      }
    }
  }
}
```

We then need a way to calculate move boards:

```
// given an arrangement of blockers, calculate the set of (pseudo-)legal moves
Bitboard Movegen::getRookMoveBoard(Bitboard blockerBoard, int square) {
  Bitboard moveBoard = blockerBoard;
```

```
  int rookXChanges[] = {1, -1, 0, 0};
  int rookYChanges[] = {0, 0, 1, -1};
  for(int k=0; k<4; ++k) {
    // move in direction specified by k, until the first blocker is met
    int x = square&7;
    int y = square>>3;
    int dx = rookXChanges[k];
    int dy = rookYChanges[k];
    bool foundBlocker = false;
    while(true) {
      x+=dx;
      y+=dy;
      if(x < 0 || x > 7 || y < 0 || y > 7) break;
      int index = y*8 + x;
      if(foundBlocker) moveBoard &= ~(1ull<<index); // wipe the bit
      else moveBoard |= (1ull<<index); // set the bit
      if((blockerBoard >> index)&1) foundBlocker = true;
    }
  }
  return moveBoard;
}
```

Finally we can iterate through each square and find magic numbers by trial and error. For a given square, if two blocker boards ever generate the same magic index but they have different move boards, then we need to try a different magic number. If the magic number works, then calculate the move board using the algorithm above, and put it at the correct index in the table.

```
// given a mask of possible blocker positions, returns the index-th configuration of blockers (sorting lexicographically)
uint64_t Movegen::getBlockerBoard(uint64_t mask, int index) {
  // turn index into binary then distribute its bits among where the mask indicates the bits should go
  uint64_t blockerBoard = mask;
  int bitIndex = 0;
  for(int sq = 0; sq<64; ++sq) {
    if((mask >> sq) & 1) {
      // set the sq-th bit of the board to (bitIndex-th bit of j)
      uint64_t val = (index>>bitIndex)&1; // val = bitIndex-th bit of j
      blockerBoard ^= (1ull << sq); // clear the sq-th bit
      blockerBoard |= (val<<sq); // or the sq-th bit with val, so that sq-th bit = val
      bitIndex++;
    }
  }
  return blockerBoard;
}

void Movegen::findRookBishopMagics(bool isRook) {
  // number of bits needed to store all possible blocker configurations, which equals the max number of potential blockers
  int bits = (isRook ? 12 : 9); // a rook has at most 12 potential blockers, bishop has at most 9
  int size = 1<<bits; // at most 2^bits possible blocker configurations: i.e. 4096 for rook, 512 for bishop

  // generate all possible blocker configurations
  for(int i=0; i<64; ++i) {
    // try magic numbers until one works i.e. hashes with no collisions
    while(true) {
      // generate a random 64-bit trial magic number with a low density of 1s
```

```
      // note: assumes std::rand is 32-bit
      uint64_t random1 = ((long long)rand() << 32) | rand();
      uint64_t random2 = ((long long)rand() << 32) | rand();
      uint64_t random3 = ((long long)rand() << 32) | rand();
      uint64_t trialNum = random1 & random2 & random3;

      bool fail = false;
      uint64_t moveTable[size];
      bool used[size];
      for(int j=0; j<size; ++j) {
        used[j] = false;
      }
      for(int j=0; j<size; ++j) {
        uint64_t blockerBoard = getBlockerBoard(isRook ? m_rookMasks[i] : m_bishopMasks[i], j);
        uint64_t index = (trialNum * blockerBoard) >> (64 - bits); // use first 12 (if rook, 9 if bishop) bits as magic index
        uint64_t moveBoard = getRookBishopMoveBoard(isRook, blockerBoard, i);
        // if two blocker boards generate the same magic index but different move boards, then fail
        if(used[index] && moveBoard != moveTable[index]) {
          fail = true;
          break;
        } else {
          used[index] = true;
          moveTable[index] = moveBoard;
        }
      }
      if(!fail) {
        std::cout << trialNum << "\n";
        break;
      }
    }

  }
}
```

These magic numbers only need to be generated once. From then on, at program startup the blocker masks and move boards are calculated and placed in the move table by the following function:

```
void Movegen::initRookBishopMoveTable(bool isRook) {
  // number of bits needed to store all possible blocker configurations, which equals the max number of potential blockers
  int bits = (isRook ? 12 : 9); // a rook has at most 12 potential blockers, bishop has at most 9
  int size = 1<<bits; // at most 2^bits possible blocker configurations: i.e. 4096 for rook, 512 for bishop

  // for each square
  for(int i=0; i<64; ++i) {
    uint64_t magicNum = isRook ? m_rookMagics[i] : m_bishopMagics[i];
    // for every blocker configuration
    for(int j=0; j<size; ++j) {
      uint64_t blockerBoard = getBlockerBoard(isRook ? m_rookMasks[i] : m_bishopMasks[i], j);

      uint64_t index = magicNum * blockerBoard >> (64 - bits); // magic index
      uint64_t moveBoard = getRookBishopMoveBoard(isRook, blockerBoard, i);
      // write to the table
      if(isRook) m_rookMoves[i][index] = moveBoard;
```

```
      else m_bishopMoves[i][index] = moveBoard;
    }
  }
}
```

Then, when needed the rook moves can be looked up like so:

```
Bitboard Movegen::rookMoves(int square, bool isWhite) {
  Bitboard blockerBoard = (whiteOccupancy | blackOccupancy) & m_rookMasks[square];
  uint64_t index = m_rookMagics[square] * blockerBoard >> (64 - 12);
  // bitboard of possible moves, then AND with NOT(occupancy) to get rid of moves which capture your own piece
  return m_rookMoves[square][index] & ~(isWhite ? whiteOccupancy : blackOccupancy);
}
```

Bishop moves
We can use the same “magic number” approach as for rook moves, but for diagonal moves instead of orthogonal ones. The size of the blocker mask is then at most 9 bits, needing 64 * 2^9 items.

Queen moves
A queen moves like a bishop OR rook, so it can be treated as a bishop and rook on the same square. So we treat the queen like a rook and generate the possible moves, then like a bishop.

King moves
Since the possible moves from a square are the same (one square in every direction), an identical approach to knight moves can be used.

Pawn moves
Pawns move one square forward at a time, and so to generate pawn moves we can bitwise shift the bitboard of white pawns by 8 (or black pawns by –8). For pawn captures, we shift by 7 and 9 (or –9,-7) to get the north-east and north-west neighbours of the pawns, and wraparound issues are fixed by not considering pawns on the A, then H, file. Pawns can also move two squares forward on their first move, which is dealt with by a bitboard that filters the pawns on the third or sixth rank.

```
uint64_t Movegen::pawnMoves(uint64_t pawns, bool isWhite) {
  // bitwise shift by +-8 to get places the pawns could advance to
  uint64_t moves = pawns << (isWhite ? 8 : -8);
  // remove moves which are blocked
  moves &= ~(whiteOccupancy | blackOccupancy);
  // factor in double pawn moves: if can move to third or sixth rank, then it must be the pawn's first move
  moves |= (moves & thirdSixthRank) << (isWhite ? 8 : -8);

  // bitwise shift left by 7 (or -9 if black) to get the captures to the left
  uint64_t leftCaptures = pawns << (isWhite ? 7 : -9);
  leftCaptures &= notHFile; // a left capture can't be on the H file, fixes wraparound issues

  uint64_t rightCaptures = pawns << (isWhite ? 9 : -7);
  rightCaptures &= notAFile;

  // capture is only valid if it captures an enemy piece
  uint64_t captures = (leftCaptures | rightCaptures) & (isWhite ? blackOccupancy : whiteOccupancy);
```

```
  return moves | captures;
}
```

Creating a move list

All the above piece-specific move generation algorithms return a bitboard of possible positions to move to. After filtering out the moves that are not actually legal (for example ones that capture your own piece), this then needs to be converted into a list of moves consisting of start squares, end squares and piece types. To do this, for each bitboard of moves returned by the above functions, we repeatedly pop the least significant bit until the bitboard becomes 0, and create a `Move` object out of each popped bit. The `popLsb` method of the `Bitboard` class is used, which uses bitwise operations to clear the last bit, then binary search to return the index of the bit that was cleared.

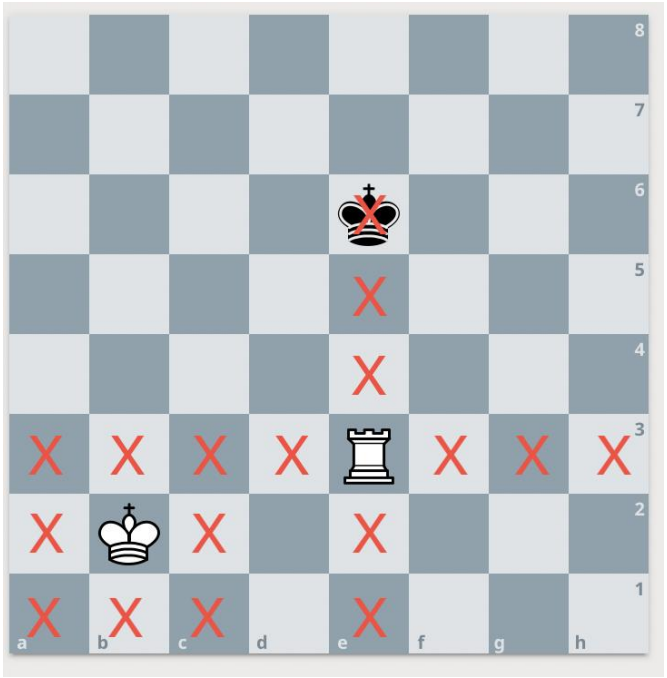
```
int Bitboard::popLsb() {
  Bitboard bb = m_board & -m_board; // for all x, x & -x is x with the last 1 set to 0
  M_board = m_board ^ poppedPiece; // clear the bit
  // binary search to convert the popped piece bitboard, bb (which only has one set bit) to an index of the set bit
  int index = 0;
  if(bb>>32) { bb>>=32; index+=32; }
  if(bb>>16) { bb>>=16; index+=16; }
  if(bb>>8) { bb>>=8; index+=8; }
  if(bb>>4) { bb>>=4; index+=4; }
  if(bb>>2) { bb>>=2; index+=2; }
  if(bb>>1) { bb>>=1; index++; }
  return index;
}
```

2.3.2 Legal move caveats

The bitboard approach in section 2.3.1 allows for almost constant-time pseudo-legal move generation. However, some of these pseudo-legal moves will not actually be legal, for example if we are in check then our possible moves are severely restricted. One way to filter out the extraneous moves is to attempt to make them, and check if the resulting position leaves our king directly free to be captured by the opponent next move. If so, then the move is illegal. However, it is more efficient to not generate the extraneous moves in the first place.

King moves

It is illegal to move the king to a square attacked by an opponent's piece. Thus, when generating valid king moves we calculate a bitboard of all the squares attacked by enemy pieces, referred to as danger squares. Then the moves returned by the king lookup table (as explained in section 2.3.1) are bitwise AND-ed with NOT(danger squares) to ensure we never move the king into a danger square. However, when calculating the danger squares, we must pretend our king is not on the board, because of the following situation with black to move:



If we do not ignore the black king, then the danger squares as highlighted in red will be calculated as shown. Importantly, e7 is not marked as a danger square because the white rook cannot move there, even though king to e7 is an illegal move since the black king would still be under attack from the rook! To fix this issue we remove the king from the board when calculating the danger squares.

```
Bitboard MoveGenerator::getDangerSquares(Position& position) {
  Bitboard dangerSquares = 0;

  bool isWhite = position.isWhiteToMove();
  Bitboard occ = position.getWhiteOccupancy() | position.getBlackOccupancy();
  occ &= ~position.getPieces(isWhite ? wk : bk); // ignore our king

  // enemy pawn attacks
  dangerSquares |= pawnAttacks(position.getPieces(isWhite ? bp : wp), !isWhite);

  // for each other piece type
  for(int t=wn; t<=wk; ++t) {
    PieceType type = (PieceType) (t + isWhite*6); // enemy piece type
    Bitboard i = position.getPieces(type);
    while(i.getBits()) {
      int index = i.popLsb();
      switch(t) {
        case wn:
          dangerSquares |= m_knightMoves[index];
```

```
          break;
        case wb:
          dangerSquares |= bishopMoves(index, occ);
          break;
        case wr:
          dangerSquares |= rookMoves(index, occ);
          break;
        case wq:
          dangerSquares |= queenMoves(index, occ);
          break;
        case wk:
          dangerSquares |= m_kingMoves[index];
          break;
      }
    }
  }

  return dangerSquares;
}
```

Evading check

Being in check restricts the possible moves from a position. To calculate whether a given position is check, for each of the enemy's piece types we pretend there is that piece on the square of our king, and see if it attacks any actual enemy pieces. If so, then that enemy piece must be attacking our king, so we are in check.

```
Bitboard MoveGenerator::getCheckingPieces(Position& position) {
  Bitboard checkers = 0;
  bool isWhite = position.isWhiteToMove();
  Bitboard occ = position.getWhiteOccupancy() | position.getBlackOccupancy();
  int kingSquare = position.getPieces(position.isWhiteToMove() ? wk : bk).getLsb();

  // for each piece type, pretend there is that piece type on the king square, then see if that piece cancapture an actual enemy piece of that type
  // ignore kings because king can't check the other king
  checkers |= pawnAttacks(1ull<<kingSquare, isWhite) & position.getPieces(isWhite ? bp : wp);
  checkers |= m_knightMoves[kingSquare] & position.getPieces(isWhite ? bn : wn);
  checkers |= bishopMoves(kingSquare, occ) & position.getPieces(isWhite ? bb : wb);
  checkers |= rookMoves(kingSquare, occ) & position.getPieces(isWhite ? br : wr);
  checkers |= queenMoves(kingSquare, occ) & position.getPieces(isWhite ? bq : wq) ;

  return checkers;
}
```

The above function returns a bitboard of pieces that are checking the king. If this bitboard is equal to 0, then we are not in check; if it has one bit set then we are in check from a single piece, otherwise we are in check from multiple pieces ("double check"). If we are in double check, then the only legal moves are king moves, with no exceptions. If in single check however we can also capture the checking piece or block the check.

This is done by calculating a **capture mask** and a **push mask**. The capture mask is a bitboard of all valid squares a piece can move to that captures a checking piece, and similarly the push mask represents valid squares to block a check (from a bishop, rook or queen). If not in check, then these are both set to all squares, because we can move anywhere.

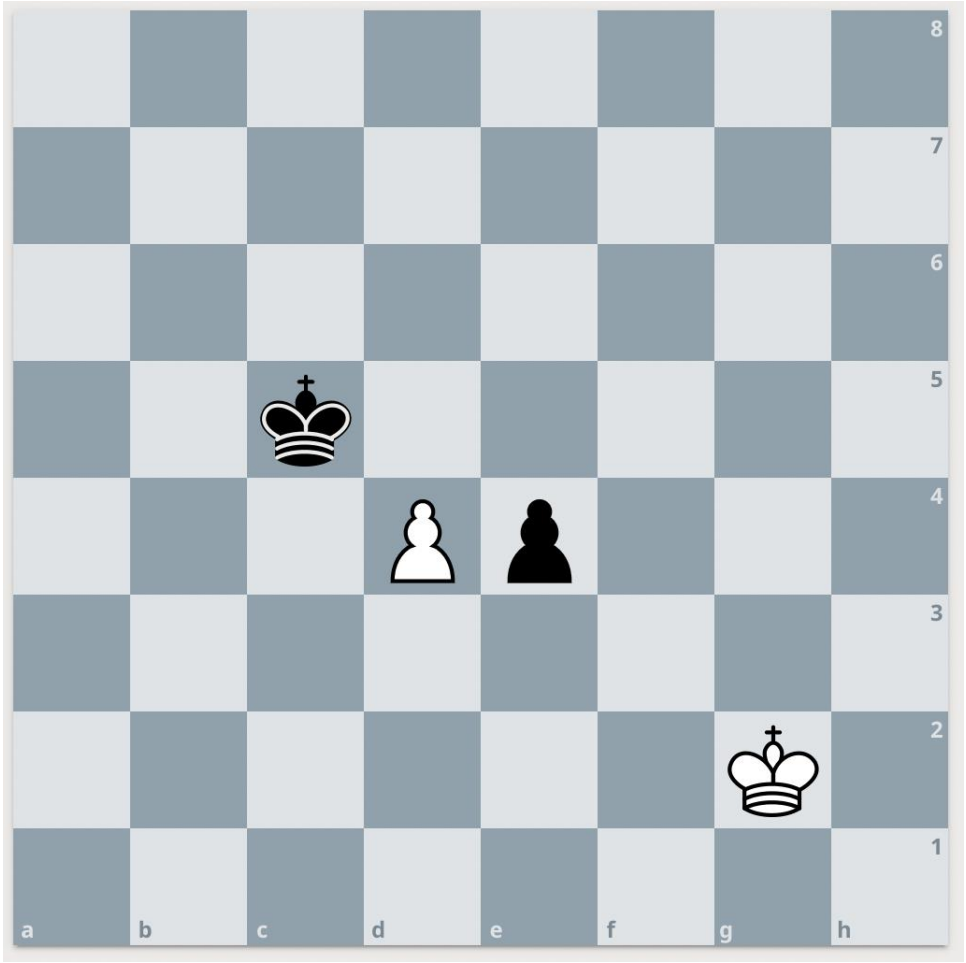
```
Bitboard checks = getCheckingPieces(position);
Bitboard pushMask = 0xffffffffffffffff; // valid squares to move that block a check - if not in check, then this is all squares
Bitboard captureMask = 0xffffffffffffffff; // valid squares to move that capture a checking piece - if not in check, then this is all squares
// if in single check
if(checks.popcnt() == 1) {
  captureMask = checks;
  pushMask = m_rookPushMasks[kingSquare][checks.getLsb()] | m_bishopPushMasks[kingSquare][checks.getLsb()];
}

// generate king moves here (omitted)

// if in double check, then can only move king
if(checks.popcnt() > 1) return moveList;

// generate other piece moves here (omitted)
```

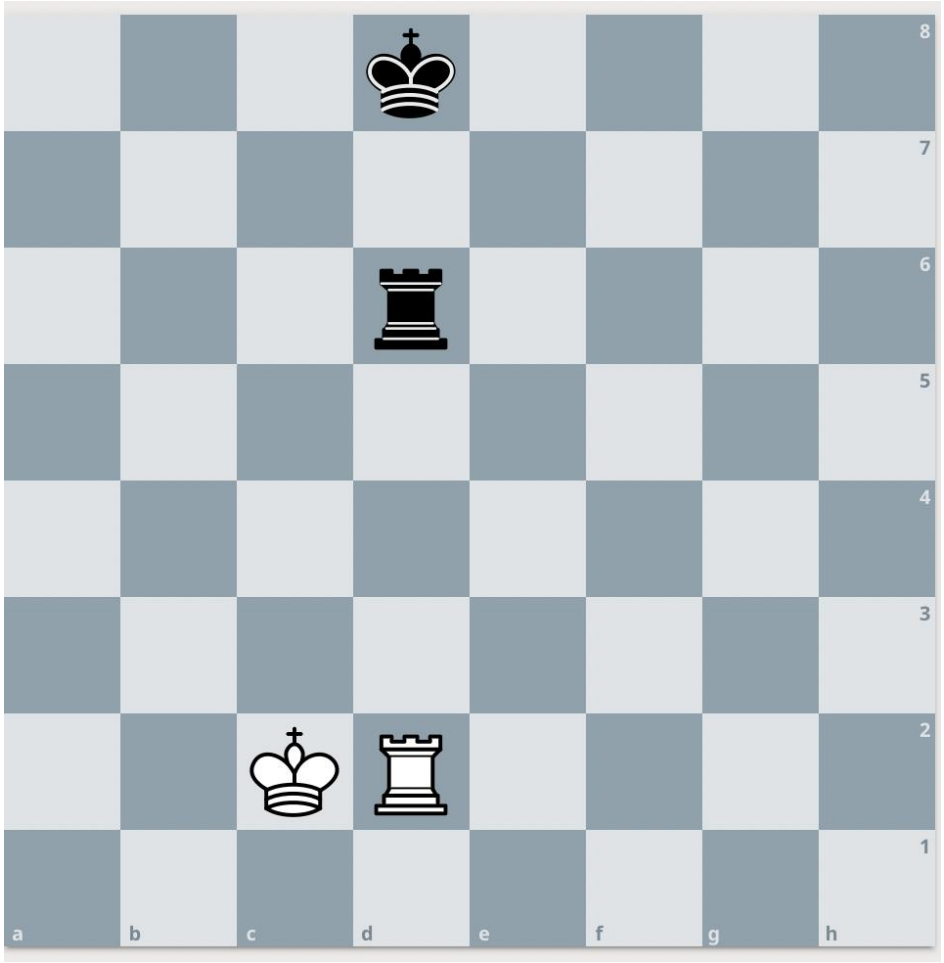
The reason for using two separate masks (capture and push) rather than just one mask of squares that deal with check (i.e. storing their bitwise OR), is because of the edge case of en passant, which is unique because the piece being captured is not on the square being moved to. Consider the following example.



White has just played pawn d2 to d4, and it is black to move. To deal with the checking pawn, black can either move the king, or play pawn e4 to d3, capturing the white pawn by en passant. Thus, the capture mask should include d4 since the pawn on that square can be captured to stop the check. However, if we attempted to store both the capture and push masks as a single bitboard, then “pawn e4 to d4” would be calculated as a valid move since d4 would be included in this mask. But that is an illegal pawn move, so we need to separate pushes and captures.

Pinned pieces

An “absolute pin” in chess is when one of your pieces cannot move because doing so would expose an attack on your king.



As an example, here with black to move the rook on d6 is pinned to the d-file (can only move vertically) because moving the rook horizontally would allow the white rook to capture the king.

The legal moves for pinned pieces are calculated as follows. Note that only sliding pieces (bishops, rooks, queens) can pin other pieces. So for each of the opponents sliding pieces, we find the squares that are between that piece and our king. This is done via a lookup table that is generated at startup:

```
// calculate+store sliding piece moves between two squares
void MoveGenerator::initPushMasks() {
  // for each square
  for(int i=0; i<64; ++i) {
    // each of the 8 directions (first 4 rook, last 4 bishop)
    int dxs[] = {0, 0, 1, -1, 1, 1, -1, -1};
    int dys[] = {1, -1, 0, 0, 1, -1, -1, 1};
```

```
    for(int k=0; k<8; ++k) {
      // move in direction specified by k, until arriving at the board edge
      int x = i&7;
      int y = i>>3;
      int dx = dxs[k];
      int dy = dys[k];
      Bitboard bb = 1ull<<i;
      while(true) {
        x+=dx;
        y+=dy;
        if( x<0 || x>7 || y<0 || y>7) break;
        int index = y*8 + x;
        bb |= 1ull<<index;
        if(k<4) // if rook
          m_rookPushMasks[i][index] = bb & ~(1ull<<i) & ~(1ull<<index); // remove start and end squares
        else
          m_bishopPushMasks[i][index] = bb & ~(1ull<<i) & ~(1ull<<index); // remove start and end squares
      }
    }
  }
}
```

Then we AND the result with the occupancy to find the bitboard of any pieces that are in between. If there is only one piece found and it is ours, then that piece is pinned and its moves are restricted to ones that stay on the pin ray.

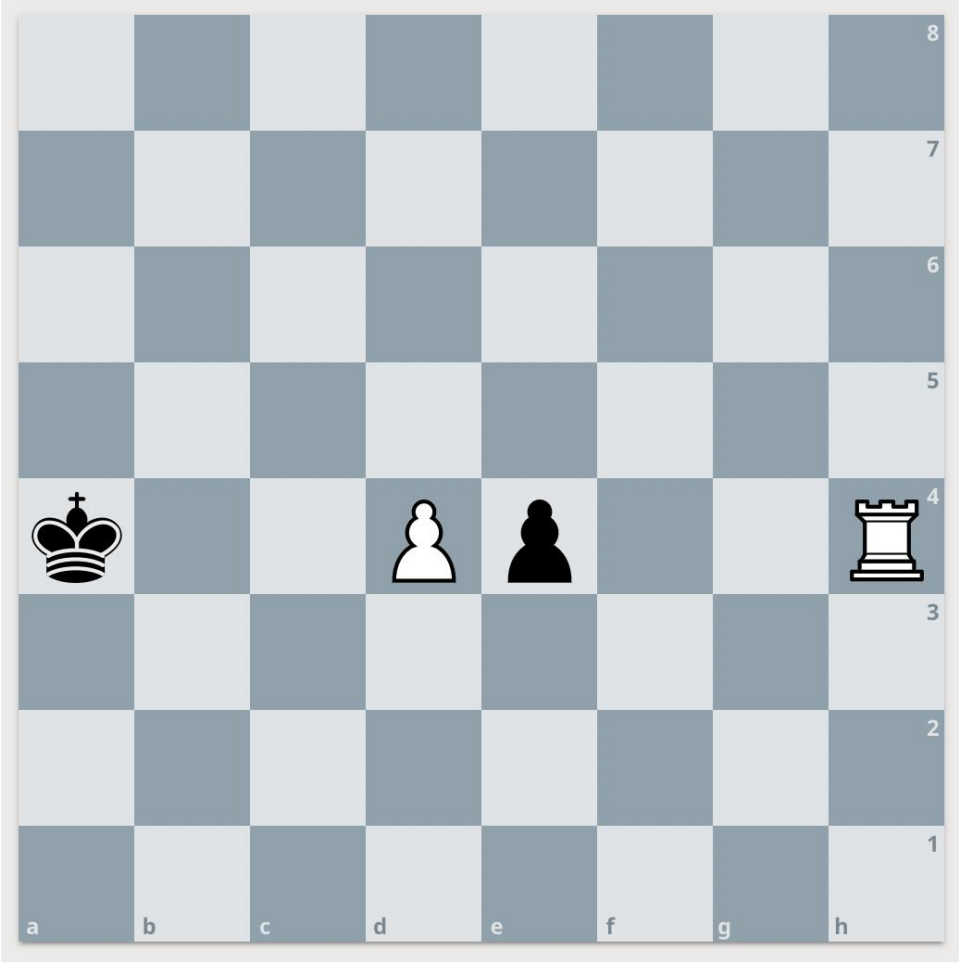
```
  Bitboard pinnedPieces = 0;
  // for each of the sliding pieces
  for(int t=wb; t<=wq; ++t) {
    Bitboard enemyPieces = position.getPieces((PieceType)(t + isWhite*6));
    while(enemyPieces.getBits()) {
      int slidingPiece = enemyPieces.popLsb();
      Bitboard squaresBetween = 0;
      if(t!=wb) squaresBetween |= m_rookPushMasks[kingSquare][slidingPiece];
      if(t!=wr) squaresBetween |= m_bishopPushMasks[kingSquare][slidingPiece];
      Bitboard piecesBetween = squaresBetween & (own|enemy);
      if(piecesBetween.popcnt() == 1 && (piecesBetween&enemy) == 0) { // if only one piece in between, and that piece isn't an enemy piece
        pinnedPieces |= piecesBetween;

        // different piece types have different move options e.g. knights can never move when pinned
        PieceType type = position.whichPiece(piecesBetween.getLsb());
        if(type==wp || type==bp) {
          // if the pinned piece is a pawn
          // generate pinned pawn moves (omitted)
        } else if(t!=wr && (type==wb || type==bb)) {// bishop can't move if pinned by rook
          // if the pinned piece is a bishop
          // generate pinned bishop moves (omitted)
        } else if ...etc
        }
      }
    }
```

}

Edge-case: En-passant pins

There is one more caveat to deal with. Consider the following situation:



White has just played the move pawn d2 to d4. If the rook on h4 did not exist, then black should be able to play pawn e4 to d3, capturing the white pawn by en passant. However, in this case it would reveal an attack of the white rook on the black king, and thus the en passant move is illegal. This is not detected by the pinned piece algorithm above, because there is more than one piece in between the rook and the king. To fix this, since en passant rarely occurs, we can just check for this situation if the move is en passant, and the engine performance will be negligibly impacted.

Once we have calculated the legal king moves and the legal moves for the pinned pieces as described above, then we have dealt with the move-generation edge cases, so we continue with the other pieces as normal (using the algorithms described in section 2.3.1). In this way, legal moves are generated quickly.

2.3.3 Minimax

To find the best move from a given position, one approach used by the program is the Minimax algorithm introduced in section 1.2.2. In this program it is implemented using three optimizations; **alpha-beta pruning**, **zobrist hashing** and **iterative deepening**. Furthermore, once depth 0 is reached in the search, instead of evaluating straight away, all captures are searched so that the evaluation returned is less volatile. For example, if we did not search all captures then the program may miss that a white queen can be captured next move, which would significantly change the outcome of the game. The procedure for searching only captures is identical to the minimax algorithm, except only capture moves are checked from the given position, instead of all moves.

Evaluation

As mentioned in section 1.2.2, Minimax needs a heuristic evaluation function. The program considers the material count, and favours endgames in which the enemy king is near the edge of the board and the kings are close togther, since this helps with delivering checkmate.

```
// positive if current player is winning, negative otherwise
double Engine::eval(Position& p) {
  double evaluation = 0;
  bool isWhite = p.isWhiteToMove();
  // material count
  int whitePawns = p.getPieces(wp).popcnt() * m_pieceValues[wp];
  int whiteOther = p.getPieces(wn).popcnt() * m_pieceValues[wn]
                 + p.getPieces(wb).popcnt() * m_pieceValues[wb]
                 + p.getPieces(wr).popcnt() * m_pieceValues[wr]
                 + p.getPieces(wq).popcnt() * m_pieceValues[wq];
  int blackPawns = p.getPieces(bp).popcnt() * m_pieceValues[bp];
  int blackOther = p.getPieces(bn).popcnt() * m_pieceValues[bn]
                 + p.getPieces(bb).popcnt() * m_pieceValues[bb]
                 + p.getPieces(br).popcnt() * m_pieceValues[br]
                 + p.getPieces(bq).popcnt() * m_pieceValues[bq];
  evaluation = blackPawns + blackOther - whitePawns - whiteOther;
  if(isWhite) evaluation *= -1;

  // count non-pawn enemy pieces to determine whether endgame
  double endgameWeight = 1 - (double)(isWhite ? blackOther : whiteOther)/(double)14.5; // initially there are 29 points worth of non-pawn material

  // in endgames, reward positions where the enemy king is at the board edge (flipped for openings)
  int enemyKing = p.getPieces(isWhite ? bk : wk).getLsb();
  int enemyRank = enemyKing>>3;
  int enemyFile = enemyKing&7;
  double enemyDistFromCentre = m_centreDist[enemyRank] + m_centreDist[enemyFile];
```

```
  evaluation += 0.1 * enemyDistFromCentre * endgameWeight;

  // in endgames, punish positions where our king is far from the centre (flipped for openings)
  int ourKing = p.getPieces(isWhite ? wk : bk).getLsb();
  int ourRank = ourKing>>3;
  int ourFile = ourKing&7;
  double ourDistFromCentre = m_centreDist[ourRank] + m_centreDist[ourFile];
  evaluation -= 0.1 * ourDistFromCentre * endgameWeight;

  // in endgames, reward positions where our king is close to enemy king
  double distBetween = abs(ourRank - enemyRank) + abs(ourFile - enemyFile);
  evaluation += 0.05 * (14-distBetween) * endgameWeight;

  return evaluation;
}
```

Alpha-beta pruning

The idea behind alpha-beta pruning is that while performing the tree search, we can check if the current branch of the tree is redundant because the best possible move this branch could give is worse than one in another branch that we have already explored. To do this, we keep track of two variables during the search.
Alpha is the minimum score that the maximizing (current) player can always achieve.
Beta is the maximum score that the minimizing (enemy) player can always achieve.
Thus, for a position reachable from a current node, alpha becomes –beta, and beta becomes –alpha.

Zobrist hashing and transposition tables

When searching a given node, the same future position may be reachable by many different sequences of moves, and thus that future position will be evaluated many times, which is unnecessary. To give a significant performance boost (especially in the endgame), we can store searched positions in a hash table (“transposition table”) to be able to quickly lookup whether we have evaluated a position before, so that if we have, we can simply use that evaluation instead of having to recurse deeper into the tree.
The hash function used is “zobrist hashing”, which works as follows.

At program start, 781 pseudorandom numbers are generated:
- 12*64 for each piece type and square
- one for whether it is black to move
- 4 for the castling rights (white kingside, white queenside etc.)
- 8 for each en passant file

```
// pseudorandom numbers
uint64_t m_zobristValues[12][64]; // each piece at each square
uint64_t m_zobristBlackToMove;
uint64_t m_zobristWhiteCastleKingside;
```

```
uint64_t m_zobristWhiteCastleQueenside;
uint64_t m_zobristBlackCastleKingside;
uint64_t m_zobristBlackCastleQueenside;
uint64_t m_zobristEnPassant[8]; // each of the 8 files
```

Then the zobrist hash of a position is the XOR of all the relevant numbers for that position.
This is efficient because when making a move, we do not have to recaculate the whole hash because of the fact that XOR is self-inverse. Instead we can update the hash with only a few operations. For example, to update the zobrist hash when a white knight moves, we can XOR the hash with the relevant pseudorandom number for the knight on the start square, and XOR with the relevant one for a white knight on the end square. The full pseudocode is as follows:

```
void Engine::updateZobrist(Move move, int castlingRemovedFlags) {
  m_zobrist ^= m_zobristValues[move.piece][move.start]; // toggle start square
  m_zobrist ^= m_zobristValues[move.promotion ? move.promotion : move.piece][move.end]; // toggle end square
  m_zobrist ^= m_zobristBlackToMove; // toggle side to move

  // if castling, update rook
  if(move.castle) {
    switch(move.castle) {
      case 1:
        m_zobrist ^= m_zobristValues[wr][7];
        m_zobrist ^= m_zobristValues[wr][5];
        break;
      case 2:
        m_zobrist ^= m_zobristValues[wr][0];
        m_zobrist ^= m_zobristValues[wr][3];
        break;
      case 3:
        m_zobrist ^= m_zobristValues[br][63];
        m_zobrist ^= m_zobristValues[br][61];
        break;
      case 4:
        m_zobrist ^= m_zobristValues[br][56];
        m_zobrist ^= m_zobristValues[br][59];
        break;
    }
  }
  // if double pawn push, update en passant
  if(move.piece==wp && move.end-move.start==16) {
      m_zobrist ^= m_zobristEnPassant[move.start&7];
  } else if(move.piece==bp && move.end-move.start==-16) {
      m_zobrist ^= m_zobristEnPassant[move.start&7];
  }

  // update castling rights
  if(castlingRemovedFlags & 1) m_zobrist ^= m_zobristWhiteCastleKingside;
  if(castlingRemovedFlags & 2) m_zobrist ^= m_zobristWhiteCastleQueenside;
  if(castlingRemovedFlags & 4) m_zobrist ^= m_zobristBlackCastleKingside;
  if(castlingRemovedFlags & 8) m_zobrist ^= m_zobristBlackCastleQueenside;
```

Using the above two optimizations, the following pseudocode performs minimax with alpha-beta. Note that when generating moves, they are ordered by a heuristic function, which makes alpha beta pruning more aggressive. The use of a transposition table also makes Iterative Deepening possible, because the results of previous searches have been stored.

```
// alpha beta minimax
double Engine::minimaxAB(Position& p, std::chrono::time_point<std::chrono::steady_clock> startTime_ms, int timeLimit_ms, int depth, double alpha, double
beta) {

  // try and lookup the position to see if already evaluated
  auto el = m_hashTable[m_zobrist % getHashTableSize()];
  if(el.key == m_zobrist && el.depth >= depth)
    return el.eval;

  std::vector<Move> legalMoves = m_gen.genMoves(p, false);
  if(legalMoves.size() == 0) {
    return m_gen.getCheckingPieces(p).getBits()==0 ? 0 : -m_inf;
  }
  order(p, legalMoves); // order the candidate moves by the evaluations in the transposition table

  if(depth == 0) {
    return capturesAB(p, startTime_ms, timeLimit_ms, alpha, beta);
  }

  Move bestMove = Move(-1, -1, empty, false, false, false); // dummy move
  bool foundBetter = false;
  for(Move move : legalMoves) {
    Position newPos = p;
    int castling = newPos.makeMove(move);
    updateZobrist(move, castling);
    double evaluation = -minimaxAB(newPos, startTime_ms, timeLimit_ms, depth-1, -beta, -alpha);
    updateZobrist(move, castling);
    if(evaluation >= beta) {
      return beta;
    }
    if(evaluation > alpha) {
      alpha = evaluation;
      bestMove = move;
      foundBetter = true;
    }
    if(getTimeElapsed(startTime_ms) >= timeLimit_ms) return 0;
  }

  if(foundBetter) writeHash(depth, alpha, bestMove);
  return alpha;
}

double Engine::capturesAB(Position& p, std::chrono::time_point<std::chrono::steady_clock> startTime_ms, int timeLimit_ms, double alpha, double beta) {
  // captures aren't forced, so check the eval before making a capture
```

```
  // otherwise, if only bad captures are available then this will evaluate the position as bad, even if other good moves exist
  double evaluation = eval(p);
  if(evaluation >= beta) return beta;
  if(evaluation > alpha) alpha = evaluation;

  std::vector<Move> captureMoves = m_gen.genMoves(p, true);
  order(p, captureMoves);

  for(Move move : captureMoves) {
    Position newPos = p;
    newPos.makeMove(move);
    double evaluation = -capturesAB(newPos, startTime_ms, timeLimit_ms, -beta, -alpha);
    if(evaluation >= beta) return beta;
    if(evaluation > alpha) alpha = evaluation;
    if(getTimeElapsed(startTime_ms) >= timeLimit_ms) return 0;
  }

  return alpha;

}
```

Iterative Deepening

Finally, the public minimax function performs searches at successive depths. This is an optimization because above, the candidate moves were ordered based on the results of the previous depth, which gives alpha-beta pruning a high chance of pruning branches, since the first move searched will probably be one of the best so alpha is high.

```
Move Engine::minimax(int timeLimit_ms, bool verbose) {
  auto begin = std::chrono::steady_clock::now();
  std::vector<Move> legalMoves = m_gen.genMoves(m_pos, false);
  if(legalMoves.size()==0) return Move(-1, -1, empty, false, false, false); // dummy move

  Move lastBestMove = Move(-1, -1, empty, false, false, false);
  double lastBestEval = -m_inf;

  // iterative deepening
  int curDepth = 0;
  while(true) {
    bool timeLimitReached = false;

    Move bestMove = Move(-1, -1, empty, false, false, false);
    double bestEval = -m_inf;

    for(Move m : legalMoves) {
      Position p = m_pos;
      int res = p.makeMove(m);
      updateZobrist(m, res);
      double eval = -minimaxAB(p, begin, timeLimit_ms, curDepth, -m_inf, -bestEval);
      updateZobrist(m, res);
```

```
      if(eval > m_inf/2) {
        // stop as soon as mate reached, at lowest depth possible
        if(verbose) std::cout << "Minimax found checkmate\n";
        return m;
      }
      if(getTimeElapsed(begin) >= timeLimit_ms) {
        timeLimitReached = true;
        break;
      }
      if(eval > bestEval) {
        bestEval = eval;
        bestMove = m;
      }
    }

    if(timeLimitReached) break;

    // search was completed at this depth, safe to update
    lastBestMove = bestMove;
    lastBestEval = bestEval;

    curDepth++;

  }

  return lastBestMove;
}
```

Note that the search is stopped as soon as checkmate is found (at the lowest depth possible), which prevents a bug of if the engine looks deep enough to find that most moves can force checkmate, so it plays any of them, and repeats this for the next move, and so on. But then the engine will never actually deliver checkmate, as it may always lurk 5 moves in the future.

2.3.4 Monte-Carlo Tree Search

To generate and test candidate good moves, one approach used is the “Monte-Carlo Tree Search” (MCTS) algorithm. In one step of this algorithm, a game tree of states is expanded and a random game is played to influence the algorithm’s view of which moves are good. A high-level overview of MCTS was given in section 1.2.3, and in this section each of the 4 steps (selection, expansion, simulation, backpropagation) are described in detail.

Data storage

The MCTS game tree is stored as a linked list of nodes, where and each node represents a position, and its children are possible positions reachable from this node by a move, and the root of the tree is the current position.

```
struct MCTSNode {

  MCTSNode(Position& pos, Move move);
```

```
  // data
  Position pos;
  Move move;
  double score; // sum over all playouts of (0 for loss, 0.5 for draw, 1 for win)
  double playouts; // number of playouts; so (score/playouts) is win percentage

  // links to other nodes
  std::vector< std::shared_ptr<MCTSNode> > children;
  std::shared_ptr<MCTSNode> parent;

};
```

Initially, the game tree is a single node consisting of the starting chess position, but after running the MCTS algorithm many times the game tree will have expanded. After a move is made, the tree is reset back to a single node so that the algorithm can start again for the next move, because we do not want to keep the data about win rates from the last MCTS result as those results likely miss some candidate good moves and we want another chance to be able to detect them.

Along with a position, each node also stores the current number of wins from that position along with the number of games simulated from that position, so that win rates can be calculated.

Stage 1 - Selection

In the first stage of MCTS, we start at the root of the game tree and select children to travel to until we reach a leaf. The choice of child node is biased by the win rates of the children so far, because we should want to explore moves with a high win rate. However, we also want to explore moves that have not yet been visited, to decrease the chance of being caught in shallow traps. The child selected is the one with the highest value of the following statistic:

$$\frac{w_i}{n_i} + c\sqrt{\ln\left(\frac{n_{i-1}}{n_i}\right)}$$

Where for a node i, w is the number of wins and n is the number of playouts. The term on the left side of the plus is high for nodes with a high win rate, and the term on the right is high for nodes with a low number of simulations compared to the parent node. Thus, summing these terms takes both factors into account. c is a positive constant that can be tweaked to control the ratio of these terms (the “exploitation/exploration ratio” as for example, setting c to be very small e.g. c=0.01 would place high emphasis on win rates, whereas c=10 would place high emphasis on low simulations. The program sets c = sqrt2.

```
  std::shared_ptr<MCTSNode> curNode = m_root;
  while(curNode->children.size()>0) {
    // choose the child node with the largest value of w_i/n_i + sqrt(c*ln(n_{i-1})/n_i)
    // where c is an adjustable constant to control the exploitation / exploration ratio
    // note: 0.000001 is added to n_i since dividing by 0 is undefined
    std::shared_ptr<MCTSNode> nextNode;
    double maxVal = -1;
    for(auto child : curNode->children) {
      double val = child->score / (child->playouts+0.000001) + sqrt(2*log2(curNode->playouts) / (child->playouts+0.000001) );
      if(val>maxVal) {
        maxVal = val;
```

```
        nextNode = child;
      }
    }
    curNode = nextNode;
  }
```

Stage 2 – Expansion

Once we have reach a leaf of the game tree through stage 1, the next step is to create new child nodes, by creating one for every legal move from the current node. Then we pick a random node to travel to so that we are still at a leaf.

```
  // create new nodes
  std::vector<Move> legalMoves = m_gen.genMoves(curNode->pos);
  if(legalMoves.size()>0) {
    for(Move move : legalMoves) {
      Position nextPos = curNode->pos;
      nextPos.makeMove(move);
      std::shared_ptr<MCTSNode> newNode = std::shared_ptr<MCTSNode>(new MCTSNode(nextPos, move));
      newNode->parent = curNode;
      curNode->children.push_back(newNode);
    }
    // pick random child
    curNode = curNode->children[std::rand() % curNode->children.size()];
  }
```

Stage 3 – Simulation

The program has two different options for the simulation stage – it either plays out random moves until the game ends (pure MCTS), or it performs a shallow minimax alpha-beta search (MCTS-AB). Performing an alpha-beta search gives a better opportunity for shallow traps to be detected and reduces the chance of missing a good move. The alpha-beta algorithm was discussed in section 2.3.3, so here we will show the pseudocode for the playout option.

```
double Engine::playout(Position& p) {
  while(true) {
    std::vector<Move> legalMoves = m_gen.genMoves(p, false);

    // terminal conditions
    if(legalMoves.size()==0)
      return m_gen.getCheckingPieces(p).getBits()==0 ? 0.5 : 1; // if no legal moves, then stalemate if not being checked, else loss
    if(p.getPlysSince50()>50) // if the 50 move rule has been exceeded, it is probably a draw, so evaluate the playout as a draw to save time
      return 0.5;
    if(p.getWhiteOccupancy().popcnt()==1) { // if white only has a king left
      bool isWhite = p.isWhiteToMove();
      if(p.getPieces(br).popcnt()) return isWhite ? 1 : 0; // rook endgame
      if(p.getPieces(bq).popcnt()) return isWhite ? 1 : 0; // queen endgame
      if(p.getPieces(bb).popcnt() > 1) return isWhite ? 1 : 0; // two bishops endgame
      if(p.getPieces(bb).popcnt() && p.getPieces(bn).popcnt()) return isWhite ? 1 : 0; // bishop+knight endgame
      if(p.getPieces(bb).popcnt() && p.getPieces(bn).popcnt()) return isWhite ? 1 : 0; // two knights endgame
      Bitboard occ = p.getBlackOccupancy();
      if(occ.popcnt()==1) return 0.5; // only kings left
      else if(occ.popcnt()==2 && (p.getPieces(bn)|p.getPieces(bb)).popcnt() == 1) return 0.5; // black only has one bishop, or knight
    }
```

```
    // same as above, for black
    if(p.getBlackOccupancy().popcnt()==1) {
      bool isBlack = !p.isWhiteToMove();
      if(p.getPieces(wr).popcnt()) return isBlack ? 1 : 0; // rook endgame
      if(p.getPieces(wq).popcnt()) return isBlack ? 1 : 0; // queen endgame
      if(p.getPieces(wb).popcnt() && p.getPieces(wn).popcnt()) return isBlack ? 1 : 0; // bishop+knight endgame
      Bitboard occ = p.getWhiteOccupancy();
      // note: only kings case has been handled above
      if(occ.popcnt()==2 && (p.getPieces(wn)|p.getPieces(wb)).popcnt() == 1) return 0.5;
    }

    // play a random legal move
    p.makeMove(legalMoves[rand() % legalMoves.size()]);
  }
}
```

Stage 4 – Backpropagation

In this stage, the results of the simulation are backpropagated throughout the game tree.

```
   // travel back up the tree, updating the information
  while(true) {
    curNode->score += result;
    curNode->playouts++;
    // flip the result because the player flips between black and white
    result = 1-result;
    curNode = curNode->parent;
    if(curNode==nullptr) break;
  }
```

2.4 File structure and organisation

There are 12 files for the program:

Bitboard.h & Bitboard.cpp (bitboard operations)
Position.h & Position.cpp (stores information that encodes a position)
Move.h (encodes move information)
MoveGenerator.h & MoveGenerator.cpp (legal move generation)
Engine.h & Engine.cpp (implements Minimax and Monte-Carlo Tree Search)
Util.h & Util.cpp (helper terminal display functions)
main.cpp (runs the CLI and instantiates the engine)

2.5 User interface design

Commands available in the CLI:

```
Format:
command <argument:type(default_value)> <...> | description
```

```
------------------------------------------------------------

help | get help about the CLI

perft <depth:int(3)> | calculate the number of games at a certain depth

position | set/reset the current position

d | display the current position

mcts <time:int(3000)> | run mcts for a set number of milliseconds

mctsab <time:int(3000)> | run mcts-ab for a set number of milliseconds

minimax <time:int(3000)> | run minimax for a set number of milliseconds

game <debug:bool(false)> | start a game

quit | quit the program
```

2.6 User guide

To run the program:

- Download the project source on the Github page.
- Make sure g++ is installed and on the PATH system environment variable.
- In the terminal, in the Chess folder run `g++ -std=c++11 *.cpp -o chess.out` to compile the program (unix, may be different for windows)
- Run the generated executable file with the command `./chess.out` (unix, may be different for windows)

Alternatively, the project source can be imported as a Code::Blocks project (or other popular C++ IDE)