Chess Engine Programming

Rhys Rustad-Elliott

Slides: rhysre.net/chess_engine_programming.pdf


IBM
the rematch
GARRY
KASPAROV
DEEP
BLUE


SPAROV VS
DEEP BLU
the rematch

Goals

1) Understand how chess engines work at a fundamental level
2) Gain enough insight to build your own engine

Shallow Blue

- My own chess engine
- Named after IBM's Deep Blue chess computer
- Written in C++
- Plays a fairly reasonable game of chess
  - I can't beat it
- Will be using examples in Shallow Blue throughout this presentation
- Source all on GitHub for those interested
  - github.com/GunshipPenguin/shallow-blue

Outline

- Board representation
- Move generation
- Evaluation
- Search
  - Alpha-beta pruning

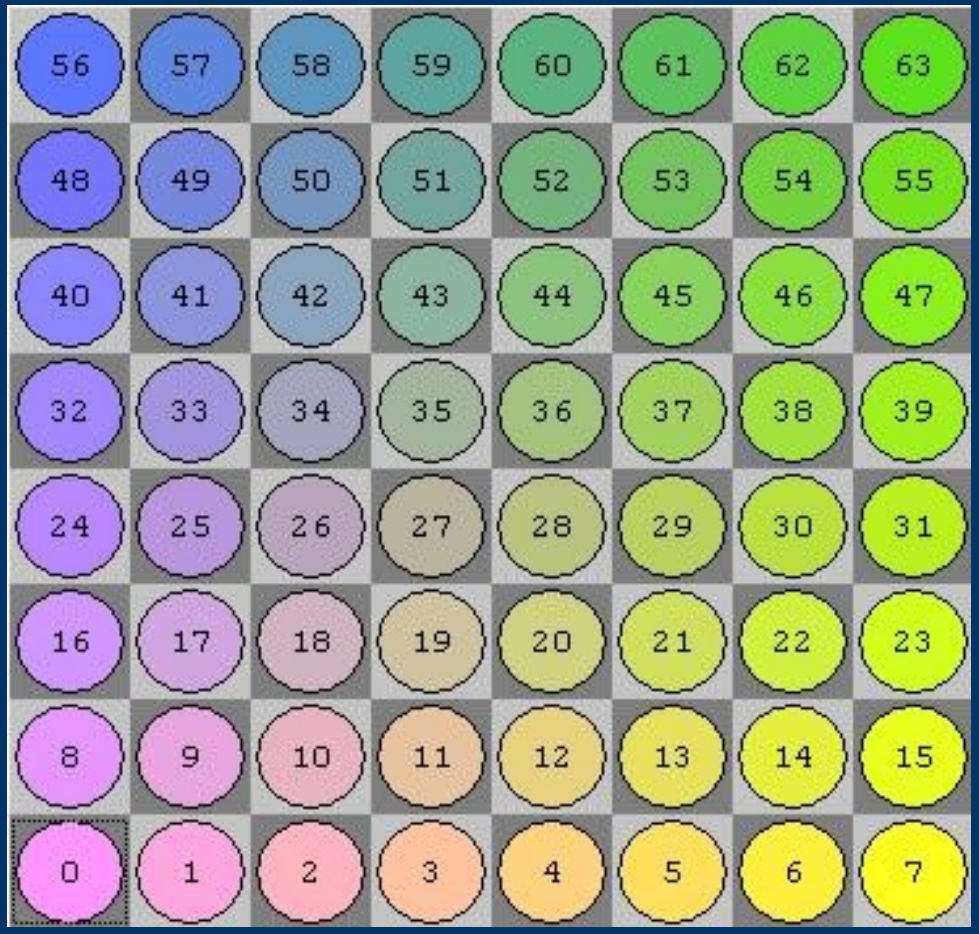
Bit Manipulation Crash Course

- Base 2
- Bitwise AND
- Bitwise OR
- Bitwise XOR

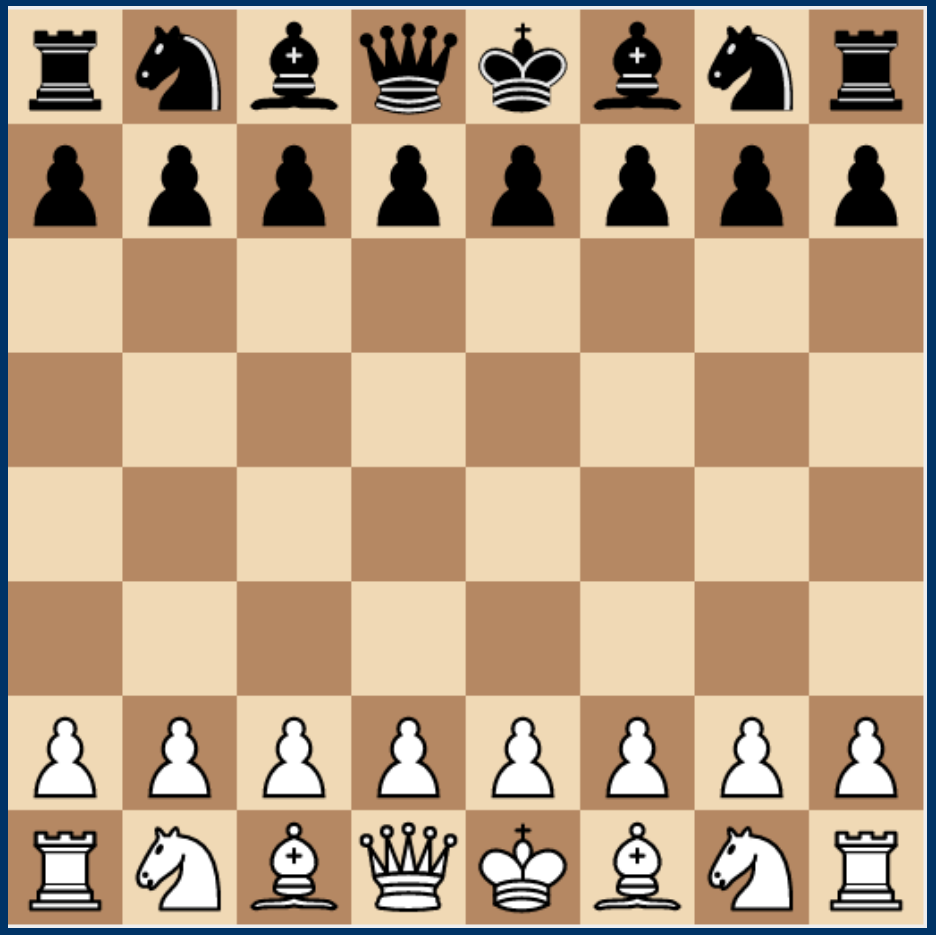
Board Representation

- Naive approaches
  - Easy to implement
  - Incredibly slow for move generation
- Bitboards allow for fast move generation
- Just a 64 bit value, bit 0 is a1, bit 63 is h8
- Bitboards exploit a nice coincidence
- 12 are required to fully represent a board, more are often used
  - Eg. Occupied squares, White pieces, Black pieces

Bit Indexes


56 57 58 59 60 61 62 63
48 49 50 51 52 53 54 55
40 41 42 43 44 45 46 47
32 33 34 35 36 37 38 39
24 25 26 27 28 29 30 31
16 17 18 19 20 21 22 23
8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7

Bitboard Example: Start Position



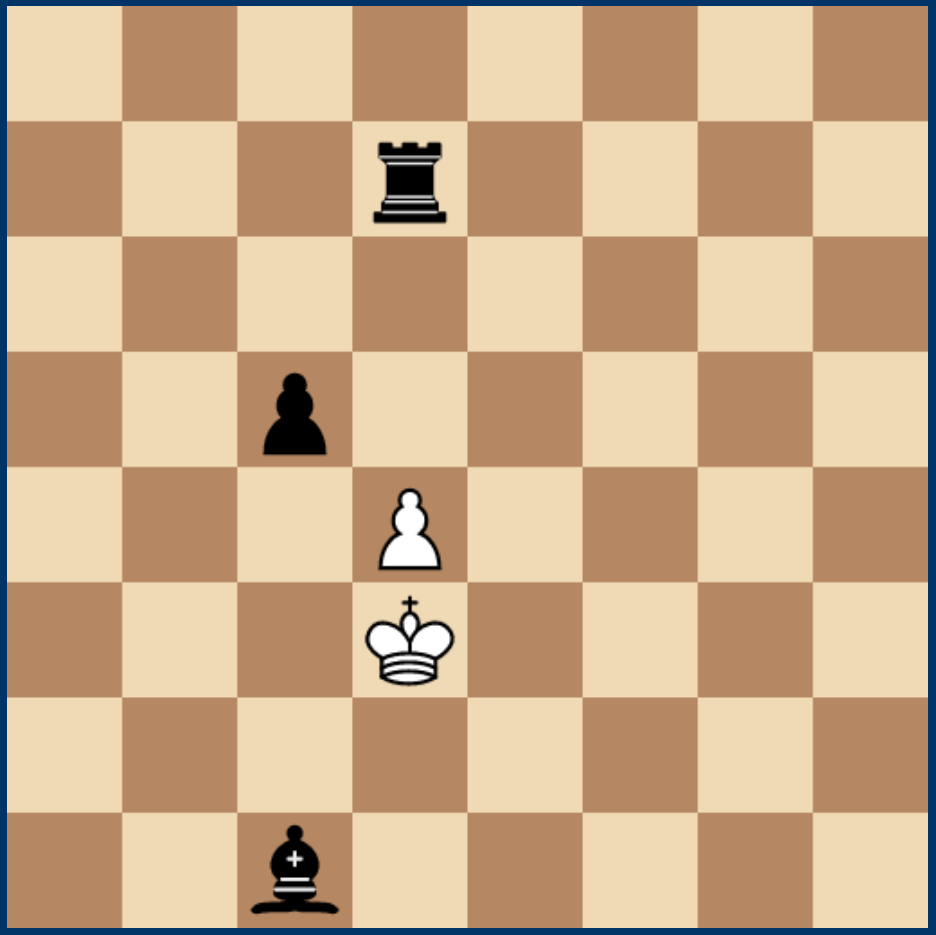
Move Generation

- For a given position, we need to generate all possible legal moves
- Must work perfectly for your engine to actually play a game
- Two main approaches exist

Move Generation Approaches

- Two approaches
  - Pseudo-legal move generation
  - Legal move generation
- Pseudo-legal – Generate all moves that follow the normal rules of movement, disregarding whether or not they move us into check. (Moves that move us into check are discarded after move generation is finished)
- Legal – Generate all legal moves

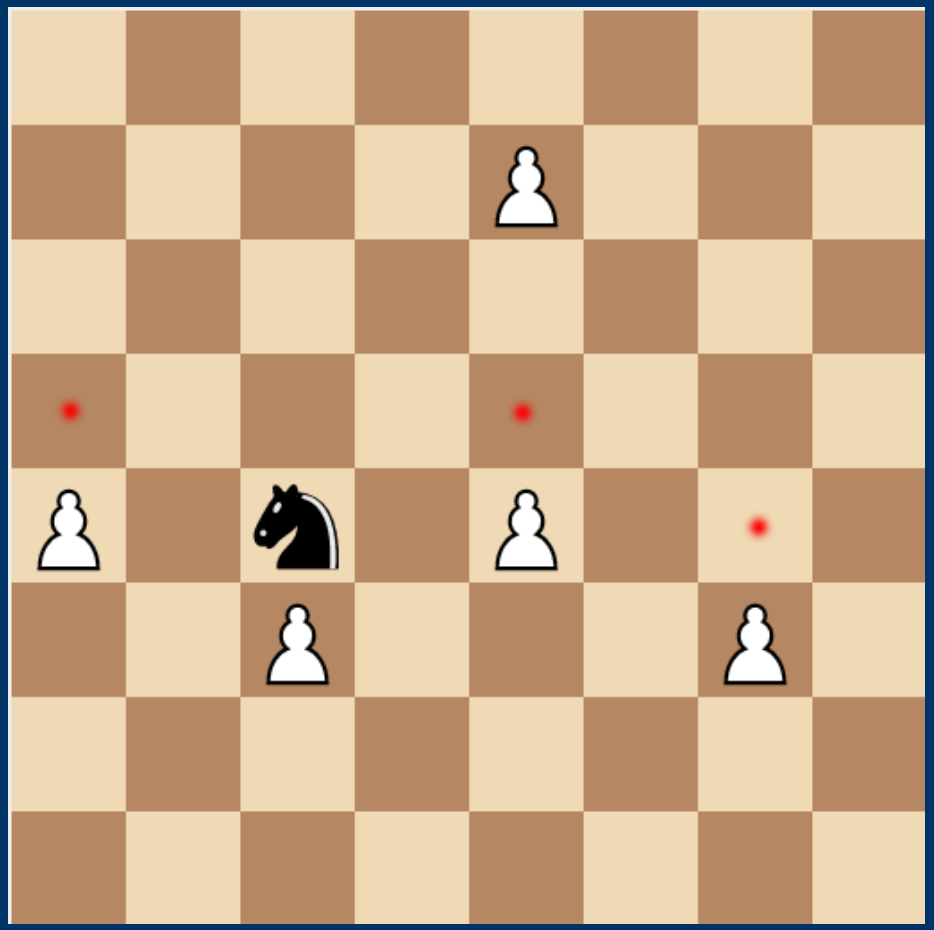
Legal vs. Pseudo-Legal Moves



Move Generation (cont.)

- Shallow Blue uses pseudo-legal move generation
- Pieces can be put into one of 2 categories
  - Non-sliding: King, Knight, Pawn
  - Sliding: Queen, Rook, Bishop
- Sliding pieces can move an indefinite number of squares along a rank/file/diagonal until hitting an edge
- Non sliding pieces can move to a predetermined number of squares depending on their starting position

Example: White Pawn Single Moves



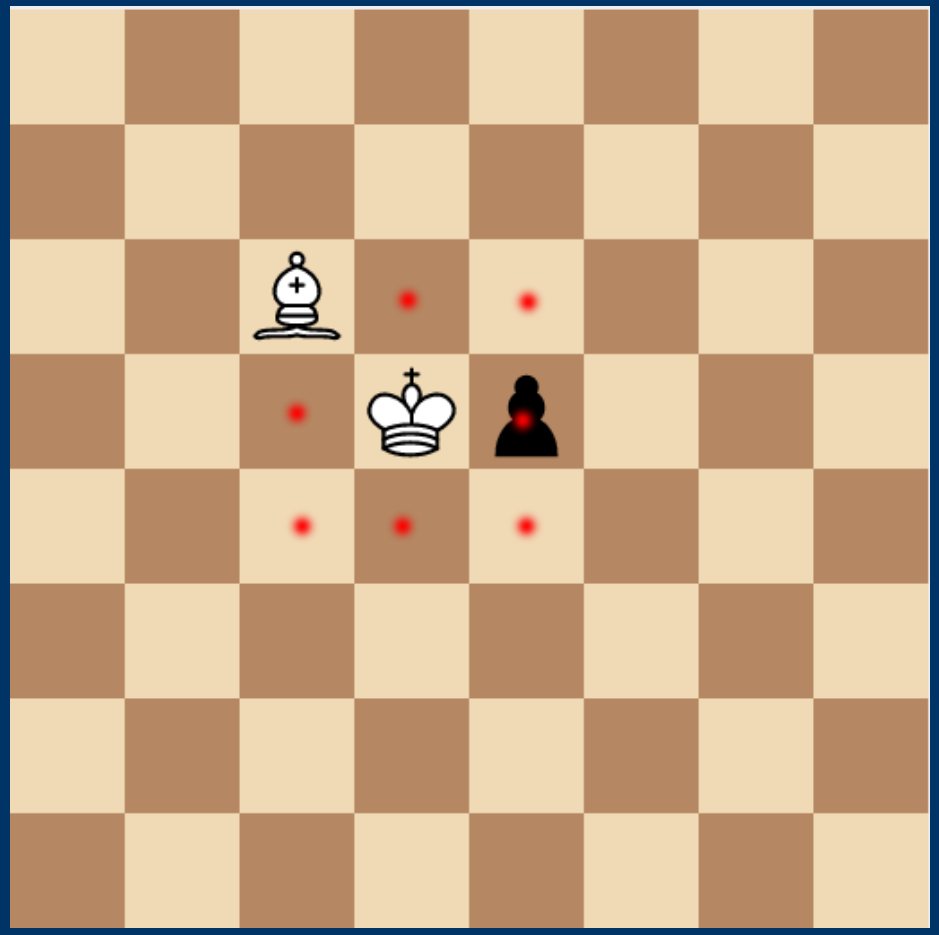
P = Pawn Bitboard

C = Occupancy Bitboard

Move Bitboard =

```
((P << 8) & (~C)) &
~RANK_8
```

Example: King Moves



K = King Bitboard

F = Friendly Pieces Bitboard

Move Bitboard = ?

Other non sliding pieces

- Technique is the same for Knights
- Non sliding move generation is generally fairly straightforward

Sliding Piece Generation

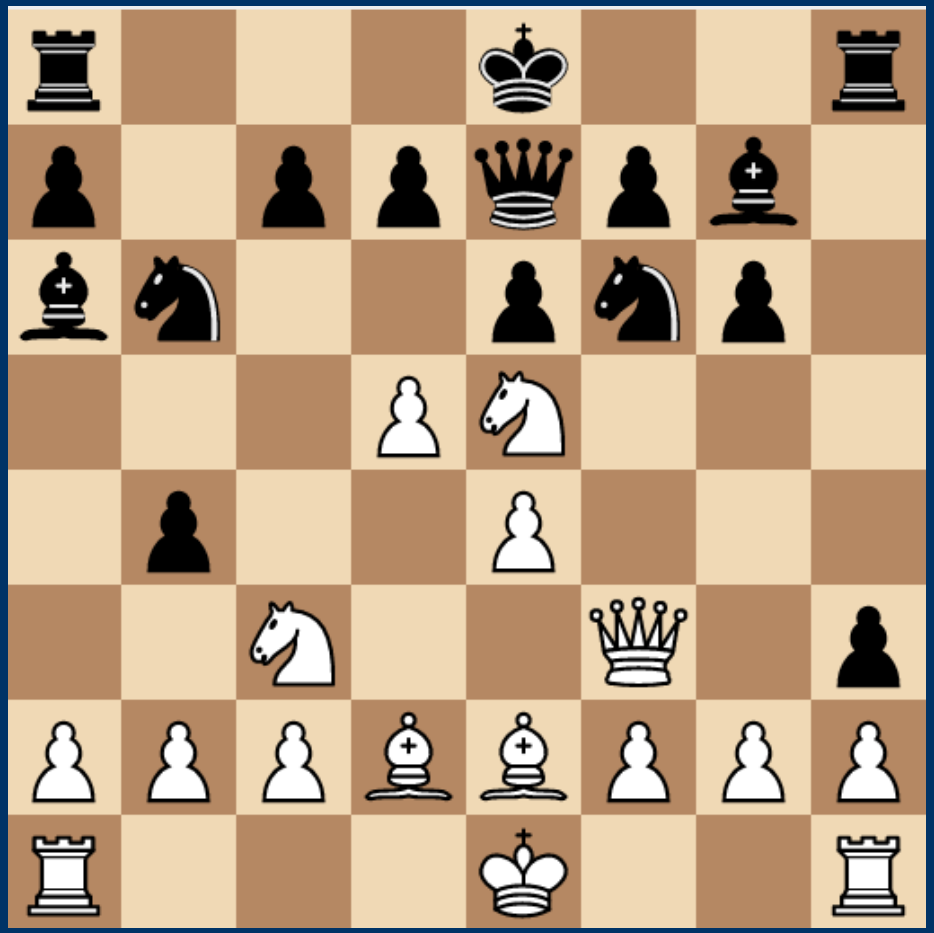
- Many ways to implement this
- Shallow Blue uses the “Classical Approach”
  - Not the fastest, but fairly straightforward

Example: Classical Approach

Perft Testing

- Way of throughly testing move generation
- Generate all moves at a certain depth in the future and compare against precomputed values

Why Test Move Generation?



- White to move
- 5 plys (halfmoves) in the future, 193,690,690 possible positions
- Your engine generates 193,690,693
- You have no tests
- Have fun debugging :)

Evaluation

- How the engine determines how good a position is
- Scores are given in centipawns (1/100th of a pawn)
  - This is the de facto standard
- Scores need to be given relative to the other player (both scores will always sum to 0)
- Evaluation functions can take many things into account, each one weighted differently
- I'll only be covering what Shallow Blue uses

Material Value

- Value of pieces
- Assignments in Shallow Blue
  - Pawn = 100 cp (by definition)
  - Knight = 320 cp
  - Bishop = 330 cp
  - Rook = 500 cp
  - Queen = 900 cp

Piece Square Tables

- The same piece can be more valuable on different squares
  - Eg. A Queen in the center is more valuable than one in the corner
- Piece Square Tables allow this to be taken into account during evaluation

Mobility

- Number of legal moves available to you
- Having more moves at your disposal is usually an advantage
- Usually assigned a small weight in relation to other factors
  - Shallow Blue assigns each move 1 cp

Other Evaluation Features

- King Safety
- Pawn Structure

Search

- The beating heart of the chess engine
- How it actually determines what moves are best
- Minimax Search

Minimax Search

- How a Chess Engine “looks ahead”
- Searches through a tree of min nodes and max nodes
  - Max player = us (trying to maximize score)
  - Min player = our opponent (trying to minimize score)
- All possible move combinations x plys in the future represented in a tree
- Notice the efficiency

Alpha Beta Pruning

- A way of optimizing minimax search
- Performs the same recursive search, but maintains 2 values
  - Alpha: Best score for the max player from current position (lower bound on score)
  - Beta: Best score for the min player from current position (upper bound on score)
- In the best case, can reduce b^d nodes to $\sqrt{b^d}$

What I Haven’t Covered

- Move ordering
- Transposition tables / Zobrist hashing
- Iterative deepening
- Chess communication protocols (UCI/Xboard)
- Quiescence search
- Principal Variation Search

Thanks!

- Resources
  - Chess Programming Wiki
    - chessprogramming.wikispaces.com
  - Shallow Blue
    - github.com/GunshipPenguin/shallow-blue
    - rhysre.net/shallowblue_docs
- How to yell at me
  - me@rhysre.net
  - Happy to answer any questions :)