How Computers Play Chess

Jacky Xu

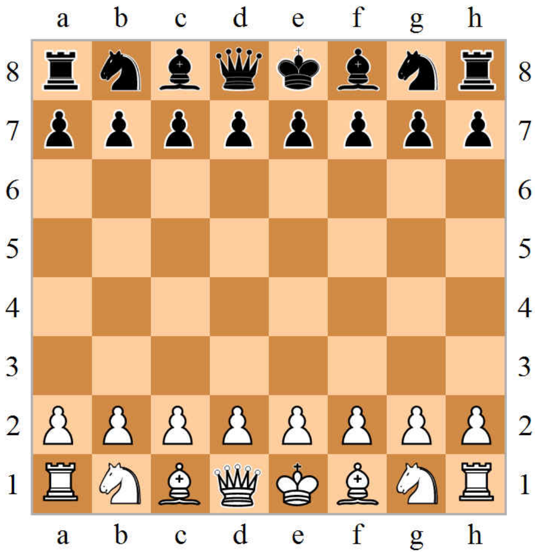
FDM Group

March 24, 2016

What is Chess?

- 2-player board game
- Originated in around 6 A.D. in India
- Turn-based, zero-sum game
- Each player has 16 pieces initially, objective is to capture the opponent's king
- Games can end in a win, loss, or a draw

The Chess Board



Source: chess-tips.blogspot.com

History of AI in Chess

- 1769 - "The Turk" by Wolfgang von Kempelen (hoax!)
- 1949 - Claude Shannon: "Programming a Computer for Playing Chess"
- 1951 - Turing created the first real algorithm for playing Chess: "TUROCHAMP"
- 1957 - First programs that can play a full game of chess were made
- 1967 - Mac Hack Six becomes the first chess program to win against a human in tournament play ($\sim$ 1500 ELO)
- 1981 - Cray Blitz wins the Mississippi State Championship with a perfect 5–0 score (first time a computer beats a master in tournament play)
- 1997 - Deep Blue defeats Chess champion Kasparov (3.5 - 2.5)

History of AI in Chess

Year	Computer Chess Rating (best fit)	Human Percentile
1950		0%
1955		0%
1960	1201	49%
1965	1400	61%
1970	1599	74%
1975	1797	87%
1980	1996	95%
1985	2194	98%
1990	2393	100%
1995	2592	100%
2000	2790	100%
2005	2988	100%
2010	3187	100%

Source: chess.com

Components of a Chess Engine

- Board Representation
- Move Generator
- Evaluation
- Search
- Opening Books and Tablebases

What is a Chess Position?

- Position of the pieces
- Which side to move
- En-passant square (if any)
- Castling rights
- A counter for draw by repetition & 50-move rule

Board Representation

- Piece Centric
  - Piece-Lists
  - Piece-Sets
  - Bitboards
- Square Centric
  - Mailbox
  - 8x8 Board
  - 10x12 Board
  - 0x88
  - Vector Attacks

8	70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d	7e	7f
7	60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f
6	50	51	52	53	54	55	56	57	58	59	5a	5b	5c	5d	5e	5f
5	40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f
4	30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f
3	20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
2	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
1	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	A	B	C	D	E	F	G	H								

Source: chessengine.co.uk

Bitboards

- Location of a piece (or pieces) is stored in a 64-bit integer (one bit for every square)
- Board is stored using 12 bitboards (one per color per piece-type)
- Ideal for x64 architecture (therefore fast!)
- Intuitive and easy to apply bitwise operations on multiple bitboards
- Used by almost all competitive chess engines today

Bitboards (Example)

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	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	0

Knight on d4 = 0x0000000008000000

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0
0	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0

Knight_Moves[d4] = 0x0000142200221400

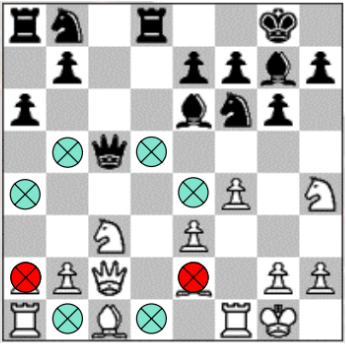
Move Generation

- What moves can I make given a particular position?
- Must ensure moves are legal
  - Destination square cannot be outside board boundaries
  - Destination square cannot be occupied by an own piece
  - Cannot move into check
  - Sliding pieces cannot jump over other pieces
  - Must account for special moves (e.g. promotions, castling and en passant)
- Speed is super important (bitboards are good at this!)

Move Generation

To generate knight moves:

```
// 'Knights' is a bitboard of all
// white knights
Knights = B->WhiteKnights;
while (Knights) {
  // Get the first available knight
  from = FirstPiece(Knights);
  // Mask out illegal moves
  to = KnightMoves[from] & ~(B->WhitePieces);
  // Add potential moves to the global movelist
  AddMovesToList(from,to);
  // Remove this knight from the list
  RemoveFirst(Knights);
}
```



Source: cs.bham.ac.uk

Evaluation

- Given a position, who has the advantage and by how much?
- An evaluation function looks at the characteristics of a Chess position and returns a score
- Typically uses a weighted sum model - assign weights to each characteristic and sum the terms up

$$
\begin{aligned}
\mathit{score} = \; & \mathit{weight}_{\mathit{pawn}} \cdot (\#\mathit{White\ Pawns} - \#\mathit{Black\ Pawns}) \; + \\
& \mathit{weight}_{\mathit{knight}} \cdot (\#\mathit{White\ Knights} - \#\mathit{Black\ Knights}) \; + \\
& \mathit{weight}_{\mathit{bishop}} \cdot (\#\mathit{White\ Bishops} - \#\mathit{Black\ Bishops}) \; + \\
& \mathit{weight}_{\mathit{rook}} \cdot (\#\mathit{White\ Rooks} - \#\mathit{Black\ Rooks}) \; + \\
& \mathit{weight}_{\mathit{queen}} \cdot (\#\mathit{White\ Queens} - \#\mathit{Black\ Queens})
\end{aligned}
$$

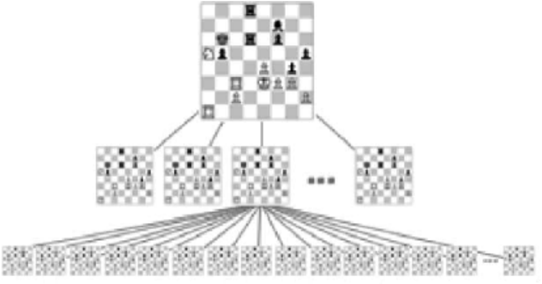
Evaluation

Other factors to consider:

- King safety (How many friendly pieces are near my king?)
- Pawn structure (connected pawns are good, double and isolated pawns are bad)
- Individual position bonuses for each piece (A knight on the rim is dim)
- Mobility (How many moves do I have available?)
- Passed pawns
- Stage of the game (opening, middle, or endgame?)
- Many, many, others ...

Search

- Static evaluation is not enough
- Must take into account opponent's best move, and our best move in reply, and ...
- Keep a game tree, where nodes are positions, and branches are moves



Source: stanford.edu

Search: Minimax Algorithm

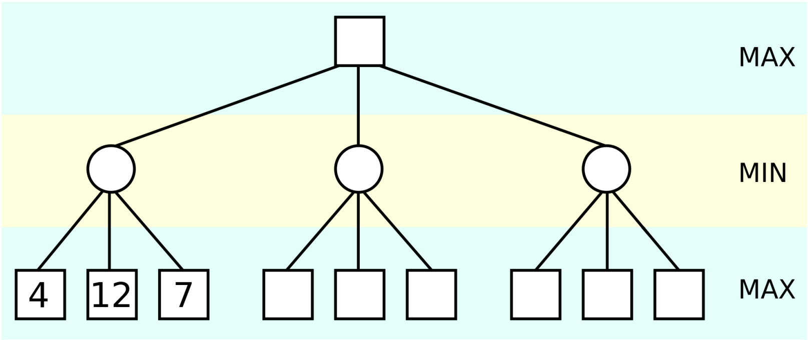
Basic idea:

- MAX-player wants to maximize his score, will always choose the move with the highest score
- MIN-player wants to minimize her opponent's score, will always choose the move with the lowest score

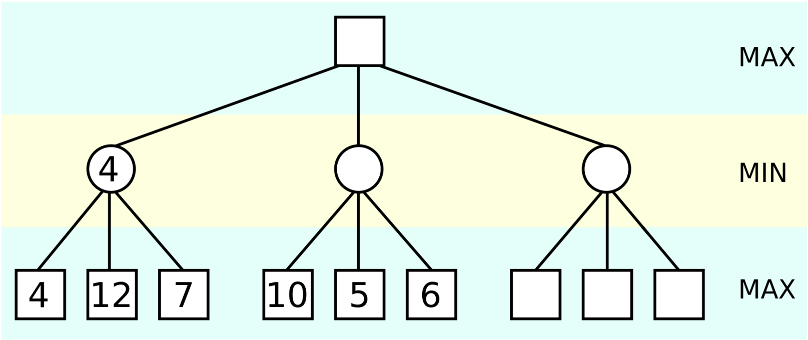
In the game tree:

- Eval. of internal nodes = maximum (if MAX-player's turn) or minimum (if MIN-player's turn) evaluation of children nodes
- Eval. of leaf nodes = calculated from the evaluation function
- Due to time and space constraints, we stop searching after reaching a certain depth

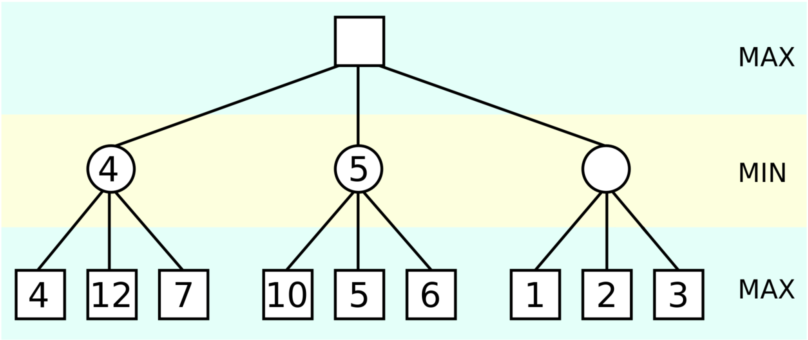
Search: Minimax Algorithm (Example)



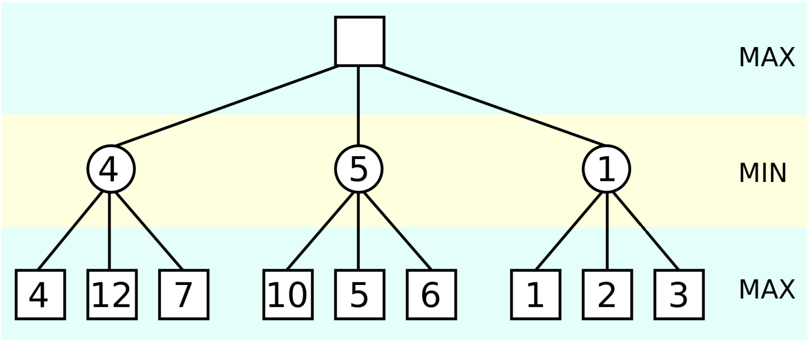
Search: Minimax Algorithm (Example)



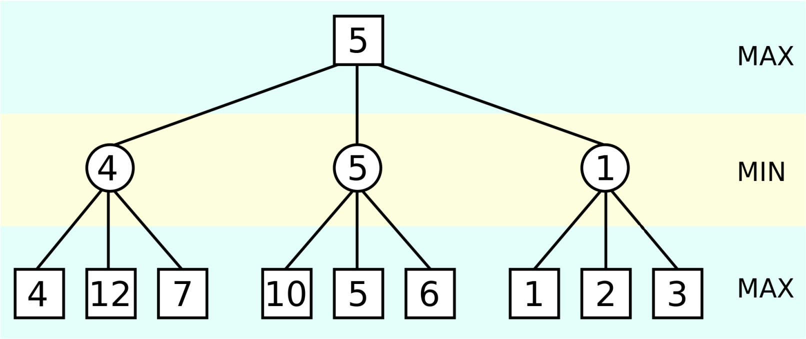
Search: Minimax Algorithm (Example)



Search: Minimax Algorithm (Example)



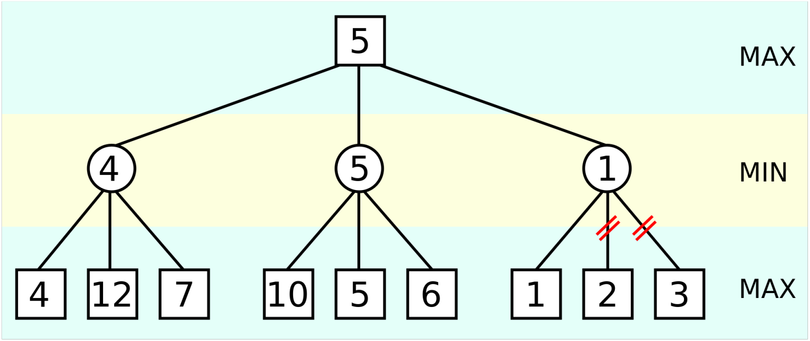
Search: Minimax Algorithm (Example)



Search: Alpha-Beta Pruning

- Problem with Minimax: too expensive
- Prune the game tree by not considering irrelevant branches
- Alpha = Value of the best possible move you can make, that you have computed so far
- Beta = Value of the best possible move your opponent can make, that you have computed so far
- If at any time Alpha $>=$ Beta, then your opponent can force a worse position than your best move so far, so there's no need to evaluate this move
- Initially, start the search with Alpha $= -\infty$ and Beta $= \infty$

Search: Alpha-Beta Pruning (Example)



Source: wikipedia.org

Search: Enchancements to Alpha-Beta Pruning

- Move-ordering heuristics (e.g. killer moves) - search the best moves first to get an early cutoff for other moves
- Quiescence search - don't stop searching the tree until a "quiet" position is reached
- Iterative deepening - gradually increase the depth of the search tree
- And many more (PVS, NegaScout, etc ...)

Search: Transposition Tables

- What if we found a position which we've already searched before? (a transposition)
- Rather than evaluate the position again, look up the evaluation from a table
- Evaluations are indexed by a hash value
- Hash value is generated (almost uniquely) by the position itself
- E.g. Zobrist hashing

Opening Books

- Chess openings are too complex
- Often strategies prove to be good/bad after 20-30 moves
- Choose the opening moves from a database of known Chess openings

Endgame Tablebases

- Endgames are surprisingly complex
- Tablebases allow perfect play of the endgame
- Unfeasible for large number of pieces
  - 4-piece tablebases $\sim$ 30 Mb
  - 5-piece tablebases $\sim$ 7 Gb
  - 6-piece tablebases $\sim$ 1.2 Tb
  - 7-piece tablebases $\sim$ 140k Tb!

Summary: Components of a Chess Engine

- Board representation - How is a chess position stored?
- Move generation - How do we generate all legal moves for a position quickly
- Evaluation - How to assess a given position based on various factors
- Search - How to find the best possible move in a tree of game positions
- Opening books and Tablebases - Tools to help computers play the opening and endgame

Future of Chess AI

- We will probably never solve Chess ($\sim 10^{120}$ possible positions in a typical game)
- Stockfish (open-source) and Komodo (closed-source, commercial) continue to improve
- Diminishing returns for additional search
- Can we make more "human-like" evaluation methods work (e.g. pattern recognition, general planning)?

Questions?

Chess Programming Wiki:
https://chessprogramming.wikispaces.com