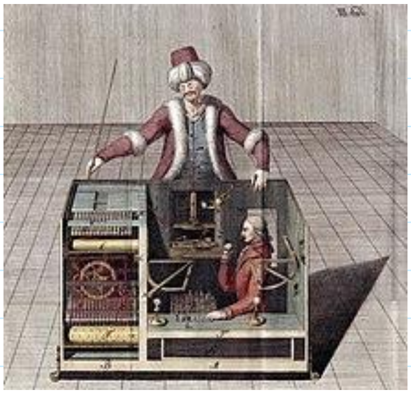
A brief primer on Computer Chess.

Thursday, April 8, 2021 6:26 PM

HISTORY

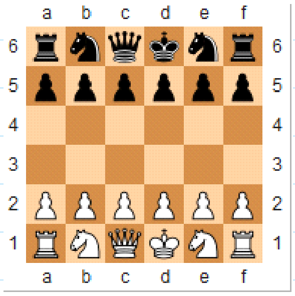
18c : The Turk



1950 : Shannon "Programming a computer for playing chess"

1951 : Turing *Turochamp*

1956 : Stain, Wells *Los Alamos Chess.*



1957 : McCarty : alpha-beta search algorithm.

1962 : *Kotok-McCarty* playing program at MIT.

..

1974 : First *World Computer Chess Championship.*

..

1980's :

- Chess Programs in PC's
- Chess Programs defeat human pros and masters
- Dedicated chess playing hardware *Mephisto*

1988 : CMU's *Deep Though* beats a human grandmaster (Bent Larsen), but loses to Gary Kasparov.

1996 : IBM's *Deep Blue* loses to Gary Kasparov

1997 : IBM's *Deep Blue* beats Gary Kasparov

2000's : Best chess programs reach super-human levels. Best chess programs run mostly on commodity hardware, development concentrate on algorithmic improvements.

2010's : *AlphaZero* and *Leela Chess Zero* incorporate machine learning to their engines.

BOARD REPRESENTATION

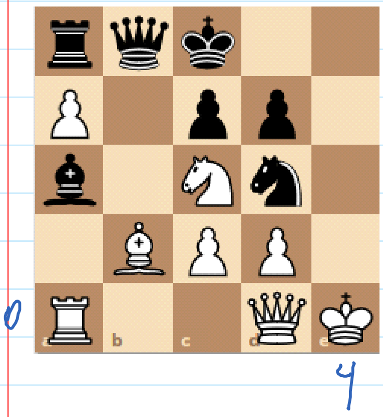
- Location of each piece on the board
- Whose turn is it?
- Can each player castle? kingside of queenside?
- Is *en passant* capture possible?

MOVE REPRESENTATION

- from square + to square
- Promotion

- Pseudo-Valid moves : Those allowed by each piece rules, ignoring possible checks.

PIECE LISTS

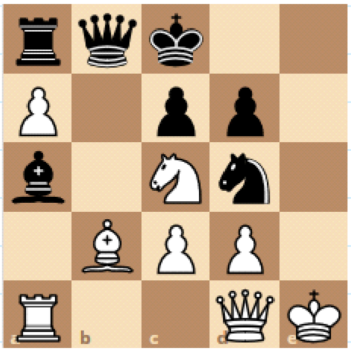


- Piece lists:

```
white_king = (4,0)
white_queen = [ (3,0) ]
...
white_pawns = [ (2,1), (3,1) ]
```

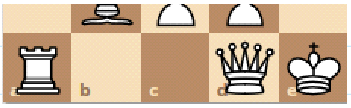
+ very memory efficient
+ easy to iterate over lists.
- spatial relation between pieces is hard

SQUARE CENTRIC REPRESENTATION



2D array:

-3	-2	-1		
		-6	-6	
-4		5	-5	
	4	6	6	
3			2	1



3			2	1

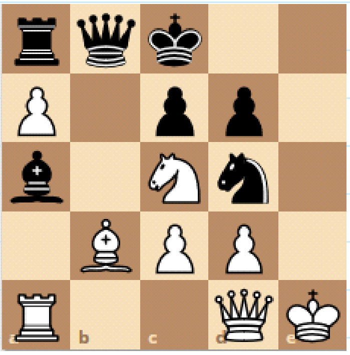
- + easy to understand
- - many empty spaces.
- - move generation requires iteration.

Pseudo: Generating moves for a bishop at location a.

```
rays = [ ne, se, sw, nw  ]
FOR EACH r in rays DO
    i ← 1
    done ← false
    WHILE not done DO
        b ← i steps in r direction from a.
        IF b is outside the board THEN
            done ← true
        ELSE
            IF b is occupied THEN
                done ← true
                IF b is occupied by an opponent THEN
                    move is a to b //capture move
            move is a to b // quiet move
        i++
```

THE MAILBOX

One-dimensional array with indexes:



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

content:

#	#	#	#	#	#	#
#	#	#	#	#	#	#
#	-3	-2	-1			#
#	6		-6			#
#	-6		5	-5		#

"	"	"	"	"	"	"
#	-3	-2	-1			#
#	6		-6			#
#	-6		5	-5		#
#		4	6	6		#
#	3			2	1	#
#	#	#	#	#	#	#
#	#	#	#	#	#	#

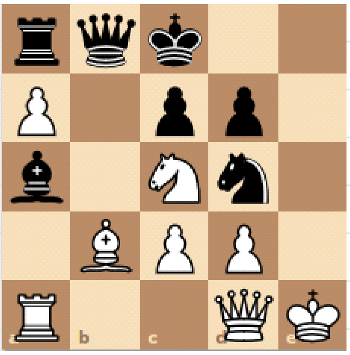
+ easier to check for out-of-bounds moves

+ easy to compute move cells:

n+6	n+7	n+8
n-1	n	n+1
n-8	n-7	n-6

BITBOARDS

- Exploit the fact that there is only 64 squares, and that is the number of bits in an integer.
- Need 12 integers for a board. One per piece type per side.
- Use bitwise operators to compute moves or properties of a board



```
white_king = 0000100000000000000000000  = 4096
00000
00000
00000
00000
00001
```

Bin

Dec.

```
white_pawns =       black_king =        black_pawns =
00000               00100               00000
00000               00000               00110
00000               00000               00000
00110               00000               00000
00000               00000               00000
```

```
white_pieces =        black_pieces =
00000                 11100
10000                 00110
00100                 10010
01110                 00000
10011                 00000
```

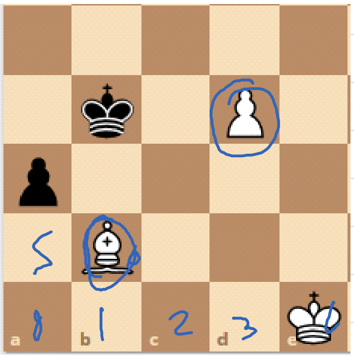
AND

11100
10110
10110
10011

+ very compact representation, can be easily hashed / serialized

+ very compact representation, can be easily hashed / serialized
+ uses very fast bitwise operation
- is fast only of hardware supports operations (count leading zeros and count trailing zeroes)
- sliding pieces (rook, bishop, queen) require complex operations or the creation of large tables.

EXAMPLE: Moves of a bishop.



```
my_bishop =
.....
.....
.....
.1...
.....
```

```
ne_ray[6] =
....1
...1.
..1..
.0...
.....
```

```
all_pieces =
.....
.1.1.
1....
.1...
....1
```

```
blockers =
all_pieces AND ne_ray[6]
.....
...1.
.....
.0...
.....
```

```
leading_zeroes( blockers ) = 6
ne_ray[24 - 6]
....1
...0.
.....
.0...
.....
```

```
my_bishop_ne =
(ne_ray[6] XOR ne_ray[18]) AND black_pieces
....0
...0.
..1..
.0...
.....
```

→ only more NE for our bishop.

"MAGIC"BITBOARDS

- Create a (hash)table , from bitboards of all possible blockers to bitboards of available moves

```
my_bishop =
........
........
........
...1....
........
........
```

```
bishop_moves[19] =
1.....1.
.1...1..
..1.1...
...0....
..1.1...
.1...1..
```

```
possible locations for blockers =
........
.1...1..
..1.1...
...0....
..1.1...
........
```

```
........          .1...1..          ........
```

Only 4*4*2*2 = 64 possible blocker patterns → Avoid more bitboards

REFERENCES:

- History:
  - https://en.wikipedia.org/wiki/Computer_chess#Chess_engines
- The Chess Programming Wiki:
  - https://www.chessprogramming.org