COMP 4752
Computational Intelligence

Lecture 23
AI Programming Optimizations

Bit Operations

- In low level languages like C, we can work directly with bit information in memory
- Bit operations are extremely fast, can be used to perform many initially unintuitive but useful functions
- Recall: An integer is a 32-bit value
- We can manipulate integers with bit ops

Bit Operations

- Bitwise and &

```
          0101011101
&         1101010100
=         0101010100
```

- Bitwise or |

```
          0101011101
|         1101010100
=         1101011101
```

- Bitwise xor ^

```
          0101011101
^         1101010100
=         1000001001
```

Bit Operations

- Bit LShift <<

```
      0001011100
<<    3
=     1011100000
```

- Bit RShift >>

```
      0001011100
>>    3
=     0000001011
```

- Bit Negation ~

```
      0101011101
~=    1010100010
```

Bit Sets

- One of the most common uses for working with bits is the concept of a bit set
- Bit sets are sets of bits of length $N = 2^n$
- For example: 0101010110110010 N = 16
- Each element of the bit set typically represents a Boolean true or false, associated with the index in the set

Bit Set Example

- We have a set of integers
  - S = {1, 4, 6, 7, 8, 12, 14}
- Can represent as a bit set of length N=16
  - B = 0100101110001010
- We can store this in a 2-byte short in C
- We usually use a 4-byte int or 8-byte long
- ~No overhead going from short to int

Bit Set Operations

- RONE = 1 = 000…001
- LONE = 0x8000 = 100…000
- Set containing single 1 in bit n
  - RBIT(n): RONE << n
  - LBIT(n): LONE >> n
- Test to see if set contains 1 in bit n
  - TEST(S, n): S & BIT(n)

Bit Set Operations

- Bit set intersection / union
  - `S1 & S2` `S1 | S2`
- Add/remove set to/from set
  - `S1 |= S2` `S1 &= ~S2`
- Add/remove/toggle bit in a set
  - `ADDBIT(S, n):` `S |= BIT(n)`
  - `REMOVEBIT(S, n):` `S &= ~BIT(n)`
  - `TOGGLEBIT(S, n):` `S ^= BIT(n)`

Bit Set – Iterate Through Bits

```
S = 01010101010111
// iterate right to left
for (int b = RBIT(1); b != 0; b << 1)
     if (S & b)     // bit b is 1 in S
// iterate left to right
for (int b = LBIT(1); b != 0; b >> 1)
     if (S & b)     // bit b is 1 in S
```

Bit Set Uses – Bit Boards!

- Many board games use 8x8 grids = 64
- One byte per position to store piece
- Bit boards can be used to store the positions of sets of pieces on the board
- In C, 64-bit long int can store board
- This will allow us to do many convenient AI related tasks in parallel

Chess Bit Board

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

	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

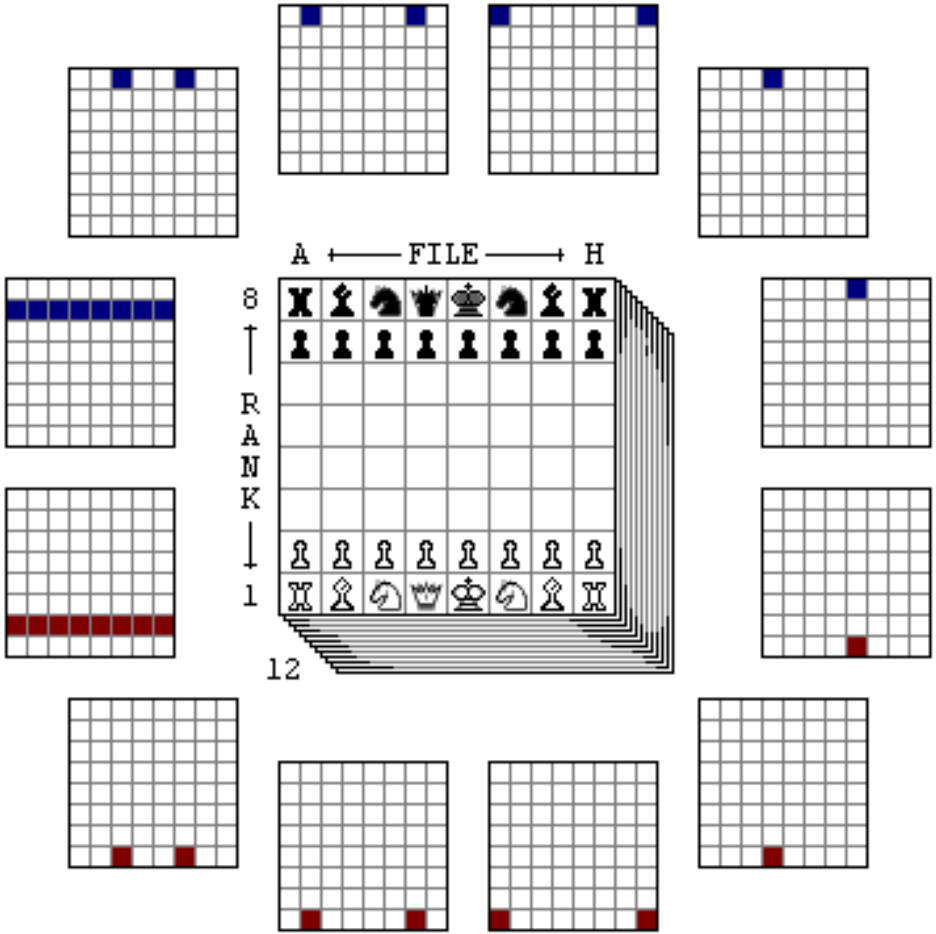
Bit Board Representation

white pawns

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

black rooks

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


A
FILE
H
8
RANK
1
12

Bit Board - Union

```
white pieces     |  black pieces     =  occupied squares

. . . . . . . .     1 . 1 1 1 1 1 1     1 . 1 1 1 1 1 1
. . . . . . . .     1 1 1 1 . 1 1 1     1 1 1 1 . 1 1 1
. . . . . . . .     . . 1 . . . . .     . . 1 . . . . .
. . . . . . . .     . . . . 1 . . .     . . . . 1 . . .
. . . . 1 . . .  |  . . . . . . . .  =  . . . . 1 . . .
. . . . . 1 . .     . . . . . . . .     . . . . . 1 . .
1 1 1 1 . 1 1 1     . . . . . . . .     1 1 1 1 . 1 1 1
1 1 1 1 1 1 . 1     . . . . . . . .     1 1 1 1 1 1 . 1
```

Bit Board - Intersection

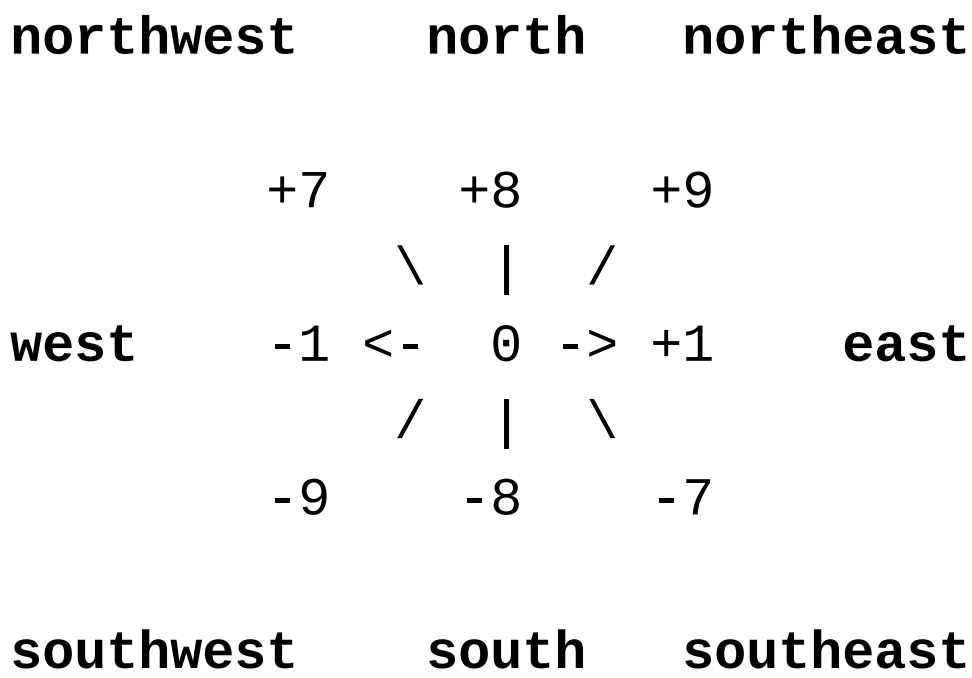
queen attacks		opponent pieces		attacked pieces
. . . .		1 . . 1 1 . . 1		. . . .
. . . 1 . . 1 .		1 . 1 1 1 1 1 .		. . . 1 . . 1 .
. 1 . 1 . 1 . .		. 1 1		. 1 . . . . . .
. . 1 1 1 . . .		. . . .		. . . .
1 1 1 * 1 1 1 .	&	. . . * . . 1 .	=	. . . * . . 1 .
. . 1 1 1 . . .		. . . .		. . . .
. . . 1 . 1 . .		. . . .		. . . .
. . . 1		. . . .		. . . .

Bit Board - Negation

~occupied squares = empty squares

```
  1 . 1 1 1 1 1 1        . 1 . . . . . .
  1 1 1 1 . 1 1 1        . . . . 1 . . .
  . . 1 . . . . .        1 1 . 1 1 1 1 1
  . . . . 1 . . .        1 1 1 1 . 1 1 1
~ . . . . 1 . . .  =     1 1 1 1 . 1 1 1
  . . . . . . 1 .        1 1 1 1 1 . 1 1
  1 1 1 1 . 1 1 1        . . . . 1 . . .
  1 1 1 1 1 1 . 1        . . . . . . 1 .
```

Bit Boards – Direction Shift



Bit Board – Population Count

- How many bits are 1 in a set?
- MANY ways of doing this
- MANY optimized versions
- POPCOUNT hardware instructions

- https://chessprogramming.wikispaces.com/Population+Count

Bit Board Optimizations

- Many people have worked on chess specific optimizations for bit boards
- Functions for operations such as:
  - Is any piece attacking a given square
  - Is a given side attacking a given square
- These optimized functions are much faster than iterating over all cells

Hashing Game Boards

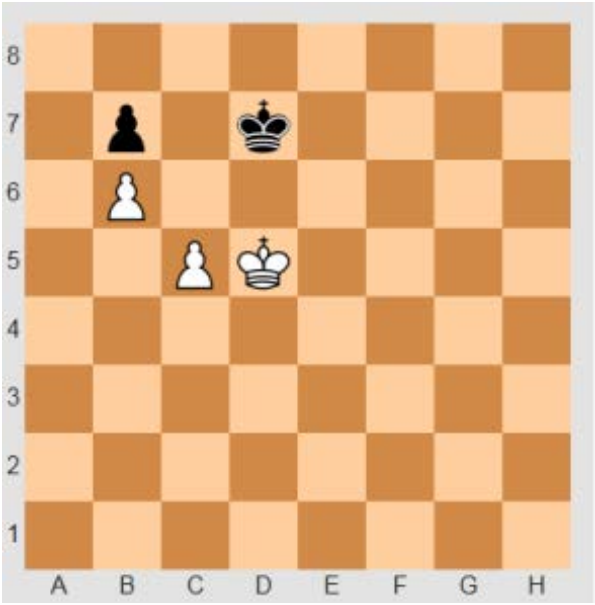
- Often we want to store information about a particular board configuration
- Table[boardPosition] = ImportantInfo
- We need a hash function which can turn a given board state into an integer

- One method: Zobrist Hashing

Zobrist Hashing

- Zobrist relies makes use of two tools
  - Tables of random numbers
  - Bitwise XOR operator
- Create tables of random numbers representing pieces on given squares
  - `Z[Player][Piece][Square] = Random`
- Zobrist Hash Function
  - `Hash = XOR Z[Player][Piece][Square]` for all pieces

Zobrist Hashing



- Z[White][Pawn][B6] = R1
- Z[White][Pawn][C5] = R2
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5

- Zobrist Hash = R1 ^ R2 ^ R3 ^ R4 ^ R5

Zobrist Hashing

- Z[White][Pawn][B6] = R1
- **Z[White][Pawn][C5] = R2**
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5

- Zobrist Hash = R1 ^ R2 ^ R3 ^ R4 ^ R5

Zobrist Hashing

- Recall that XOR is reversible
- A = 1001, B = 1101
- A ^ B = 0100
- A ^ B ^ A = 1101 = B
- A ^ A = 0000
- We can remove a piece by applying XOR

Zobrist Hashing

- Z[White][Pawn][B6] = R1
- **Z[White][Pawn][C5] = R2**
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5

- Zobrist Hash = R1 ^ R2 ^ R3 ^ R4 ^ R5

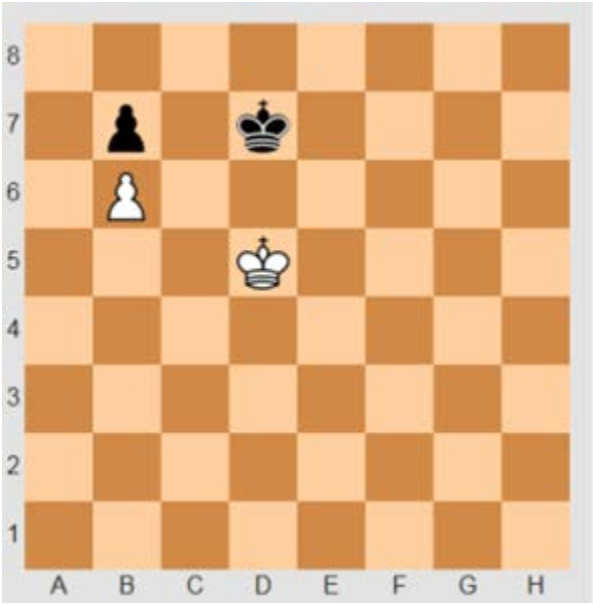
Zobrist Hashing

- Z[White][Pawn][B6] = R1
- **Z[White][Pawn][C5] = R2**
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5

- Zobrist Hash = R1 ^ R2 ^ R3 ^ R4 ^ R5 ^ R2

Zobrist Hashing

- Z[White][Pawn][B6] = R1
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5



- Zobrist Hash = R1 ^ R3 ^ R4 ^ R5

Zobrist Hashing

- Z[White][Pawn][B6] = R1
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5
- Z[White][Pawn][C7] = R6

- Zobrist Hash = R1 ^ R3 ^ R4 ^ R5 ^ R6

Zobrist Hashing

- Zobrist Hashing is incremental
- When a piece is added, we simply XOR the existing hash by the Z-value
- When a piece is removed, we XOR the existing hash by the Z-value
- Moving a piece = remove then add at a different location = 2 XOR ops (3 if capt)