COMP 3200
Artificial Intelligence

Lecture 10

Minimax Search Enhancements

Useful Links

- Chess Programming Wiki
  - https://www.chessprogramming.org/
  - https://www.chessprogramming.org/Recommended_Reading
- Coding Adventure: Chess
  - https://youtu.be/U4ogK0MIzqk
- Making a Better Chess Bot
  - https://youtu.be/_vqlIPDR2TU
- Computer (and human) Perfection at Checkers
  - https://youtu.be/VWqtNS9pmOI

Actions + Moves

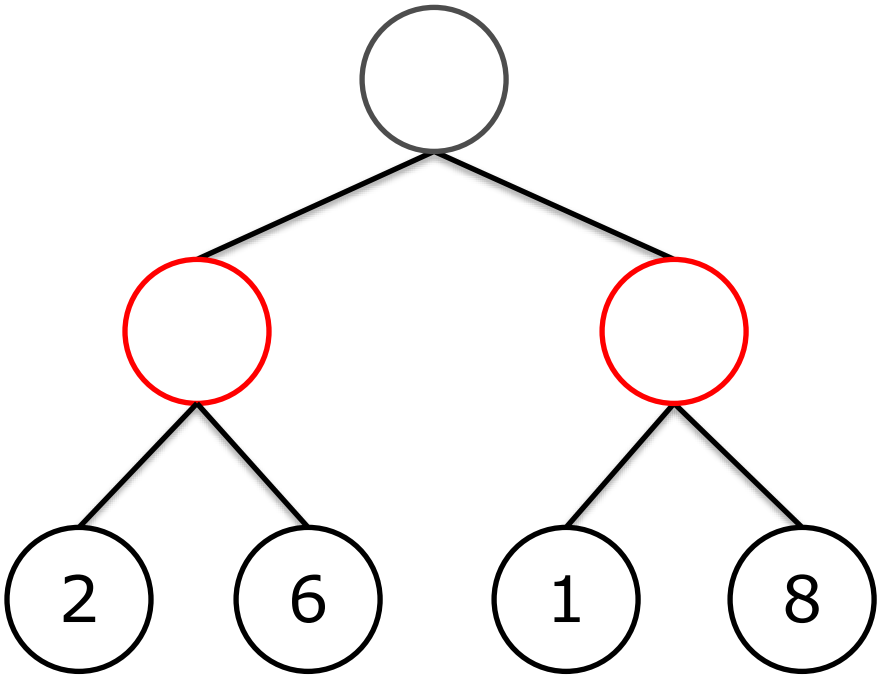
- Depending on the resource or game, you may see 'Action' or 'Move' used to describe the action(s) you take on a turn
- Usually used interchangeably
- Sometimes a 'Move' for a turn could consists of multiple individual 'Actions'
- This lecture: Action == Move

State Evaluation

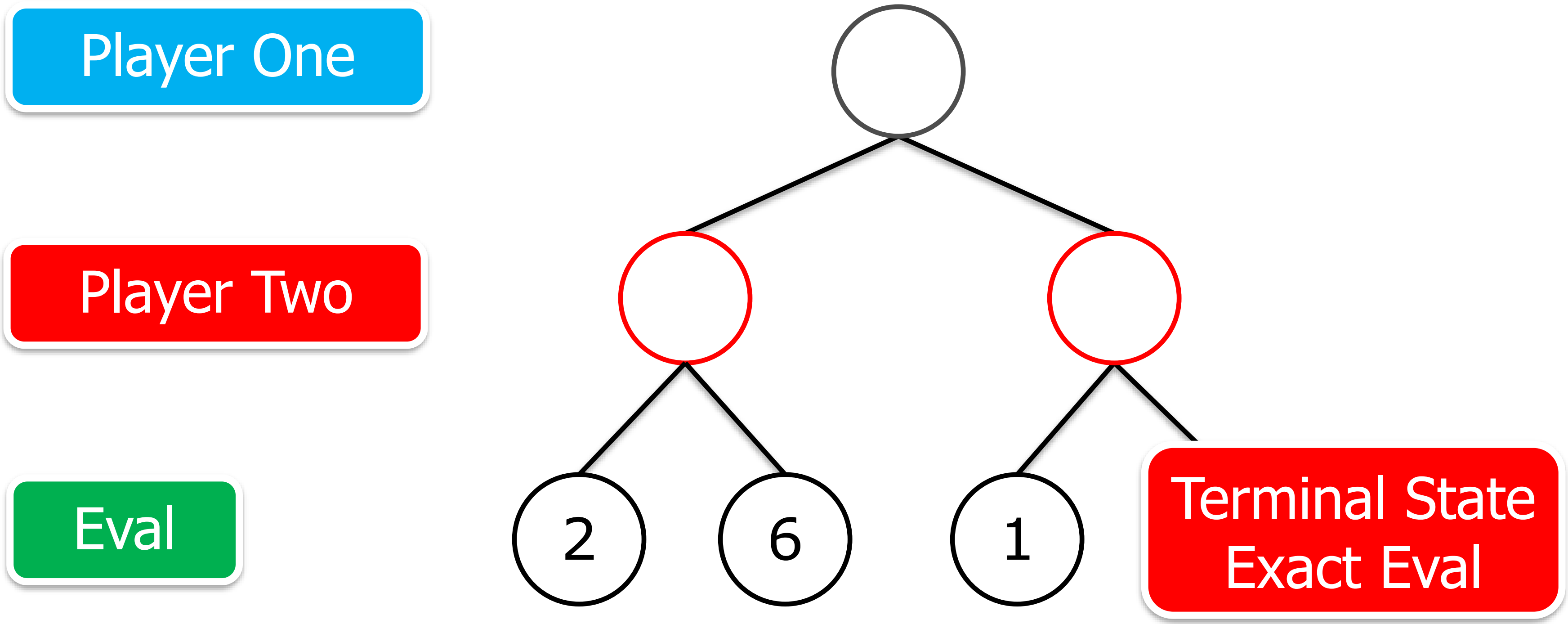
Player One

Player Two

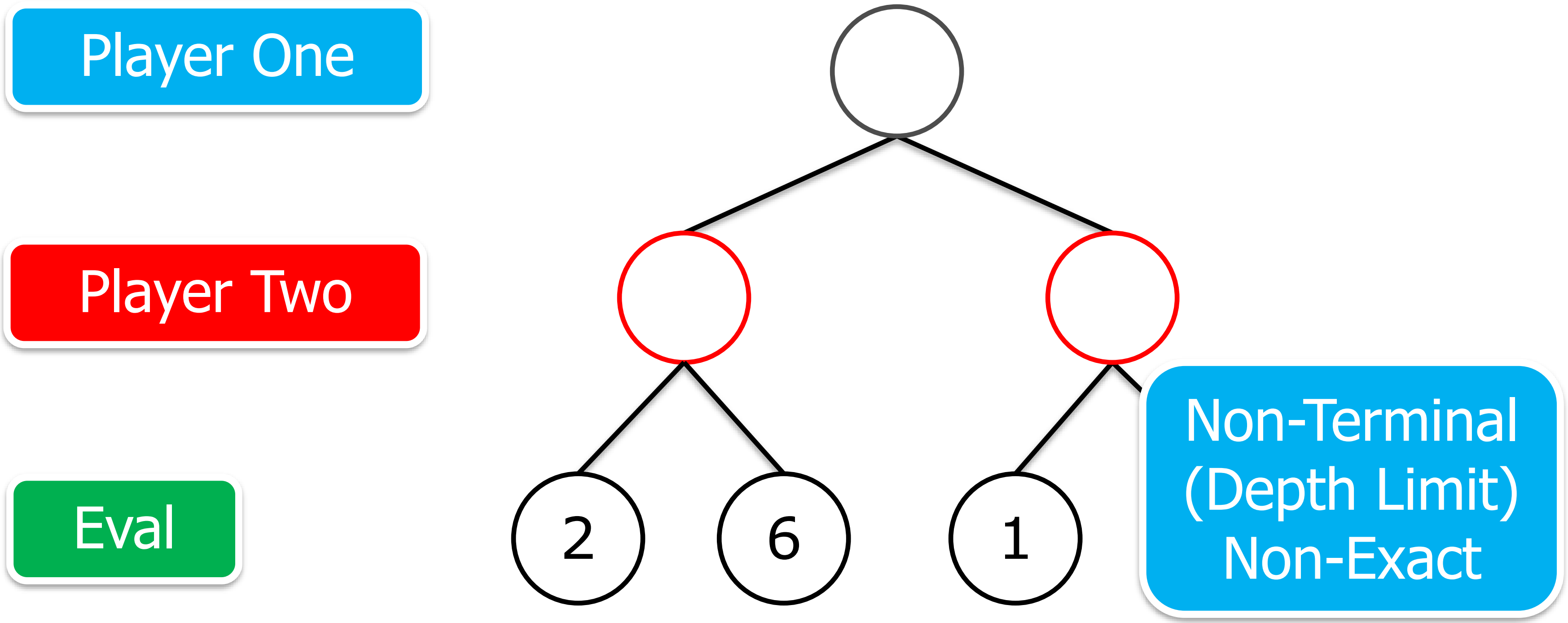
Eval

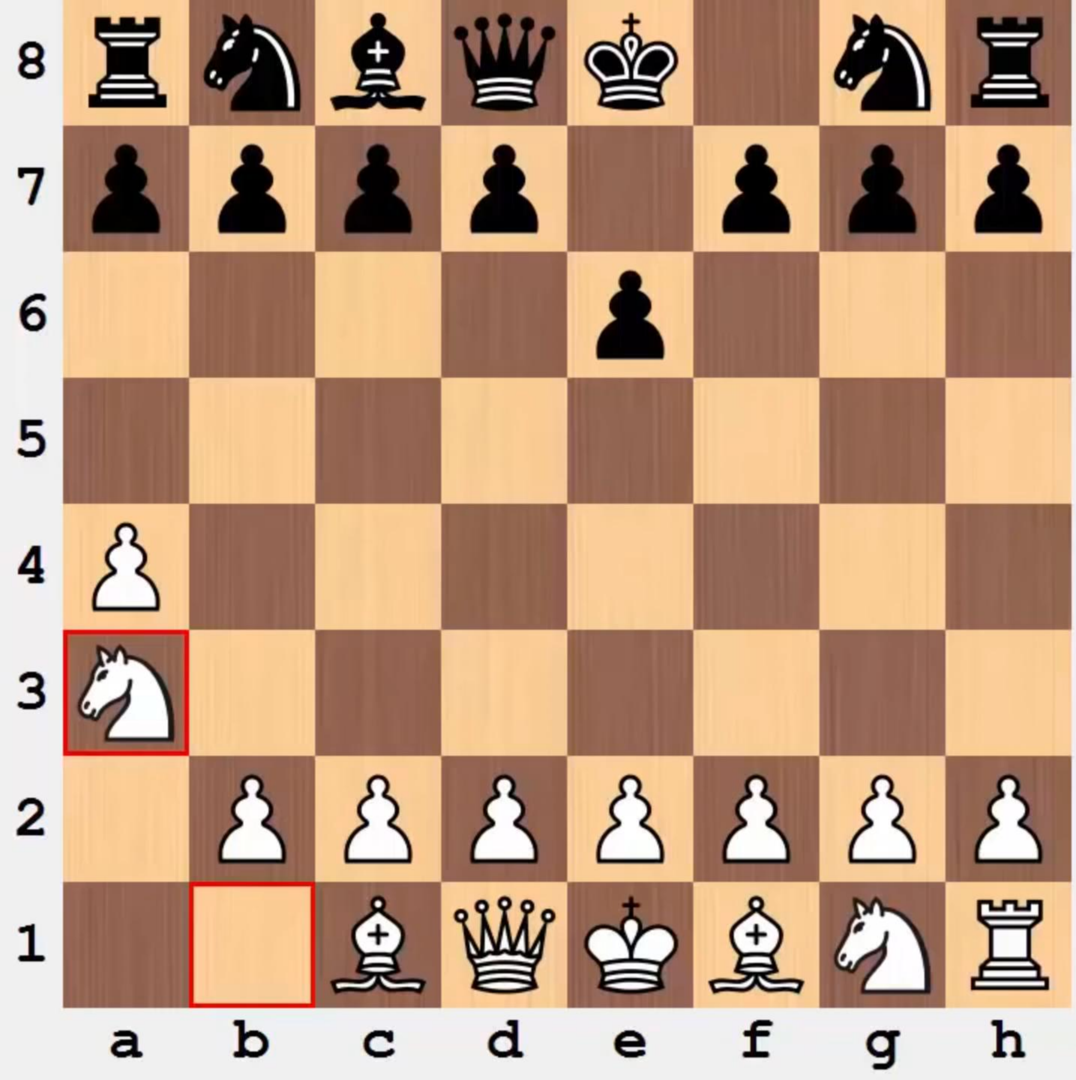


State Evaluation



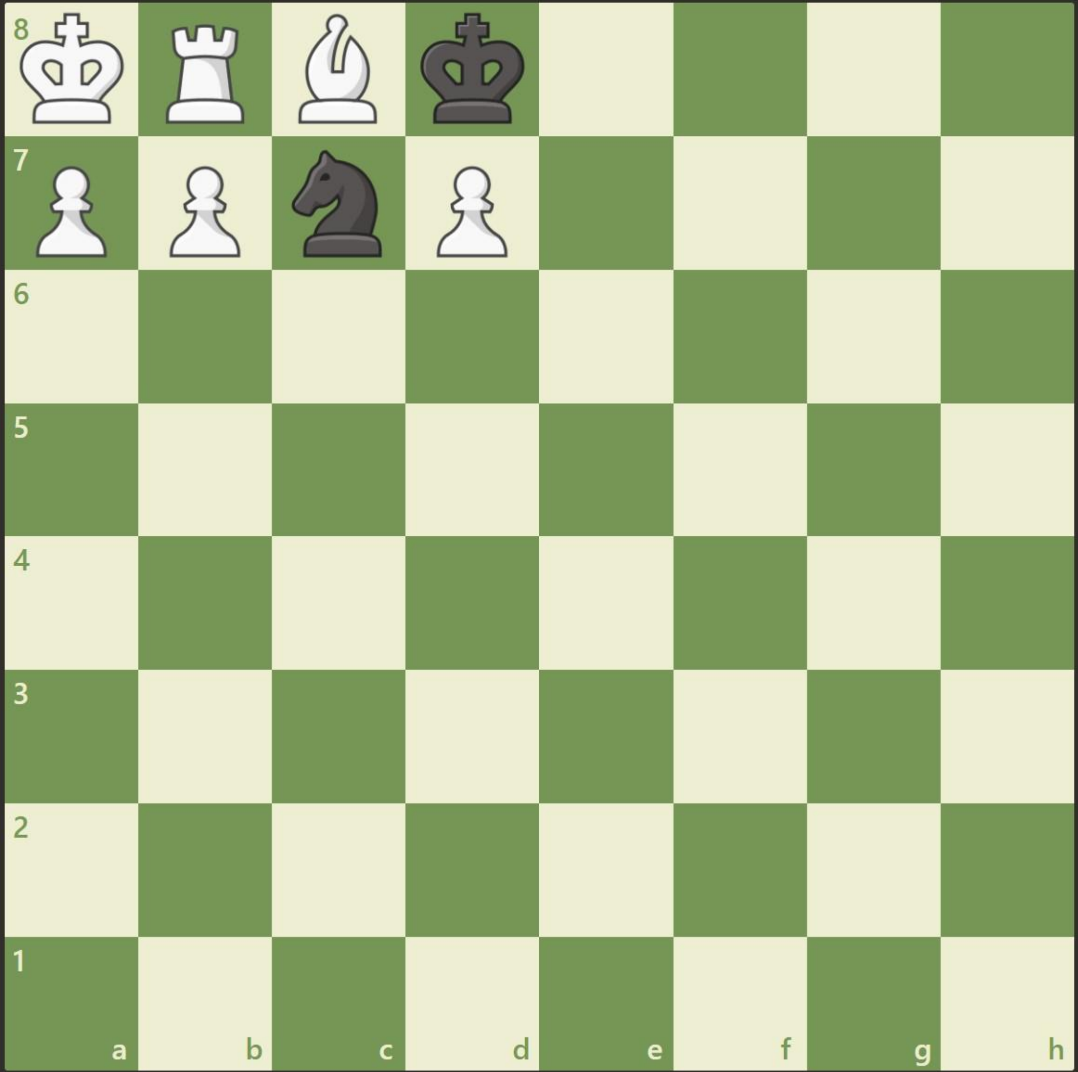
State Evaluation

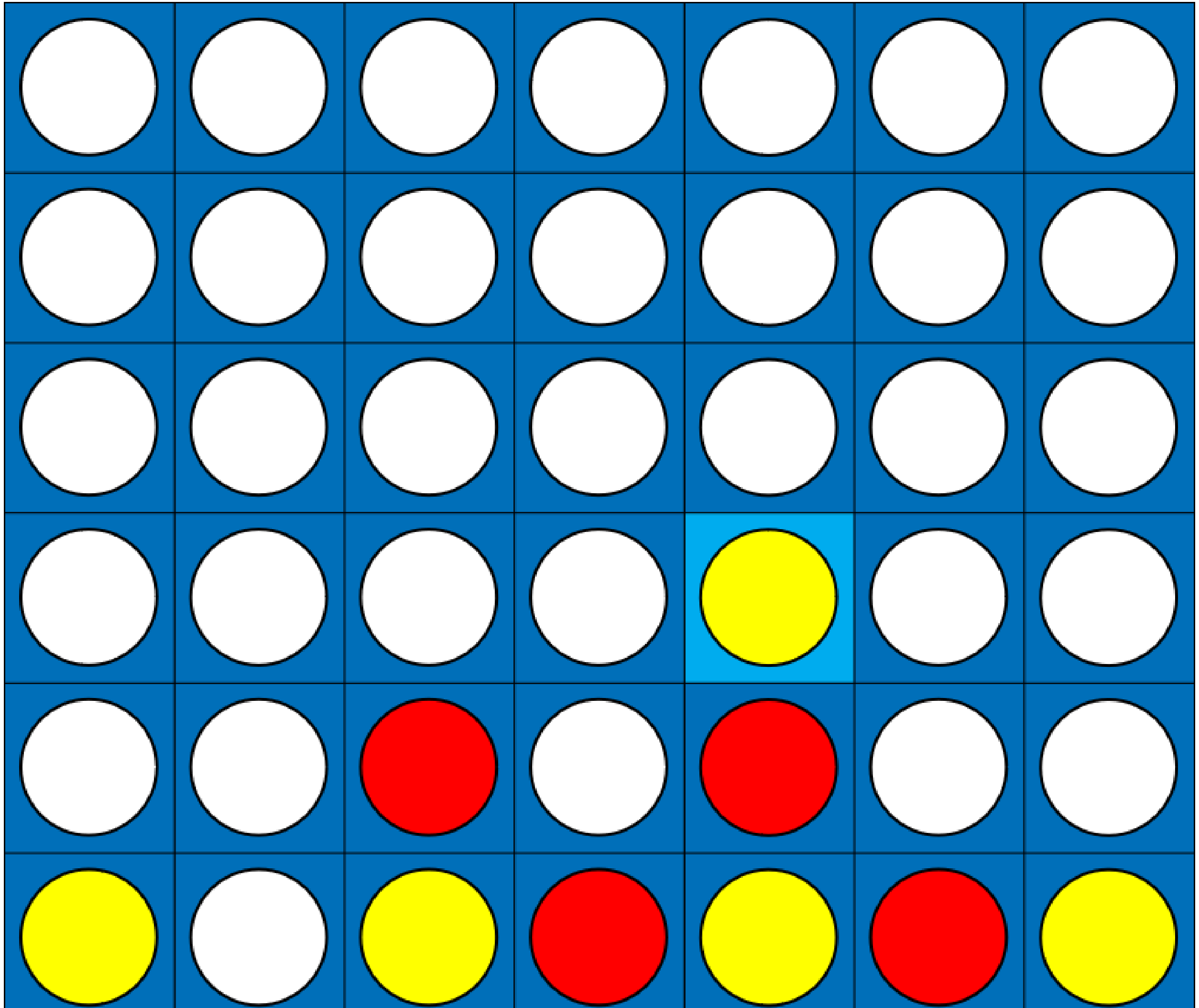




Basic Point System

- ♟ = 1 point
- ♞ = 3 points
- ♝ = 3 points
- ♜ = 5 points
- ♛ = 9 points
- ♚ = priceless


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



State Evaluation

- The heuristic evaluation (score) of a state can be a little confusing at first
- Maximizing player wants highest value
- Minimizing player wants lowest value
- Therefore, the score should always be calculated w.r.t. the maximizing player
    - Keep track of the maximizing player (this.maxPlayer)
- Typically: MaxGoodness = -MinGoodness

State Evaluation

- Two-player turn-based games
- 4 possible state evaluations
  - Player 1 has won the game `PLAYER_ONE`
  - Player 2 has won the game `PLAYER_TWO`
  - Game is a draw `PLAYER_DRAW`
  - Game not over `PLAYER_NONE`

Terminal Non-Terminal

State Evaluation

```
eval(state, player) {
    let winner = state.winner();
    if      (winner === player)             { return 10000; }
    else if (winner === (player + 1) % 2)   { return -10000; }
    else if (winner === PLAYER_DRAW)        { return 0; }
    else if (winner === PLAYER_NONE)        { return Heuristic(); }
}

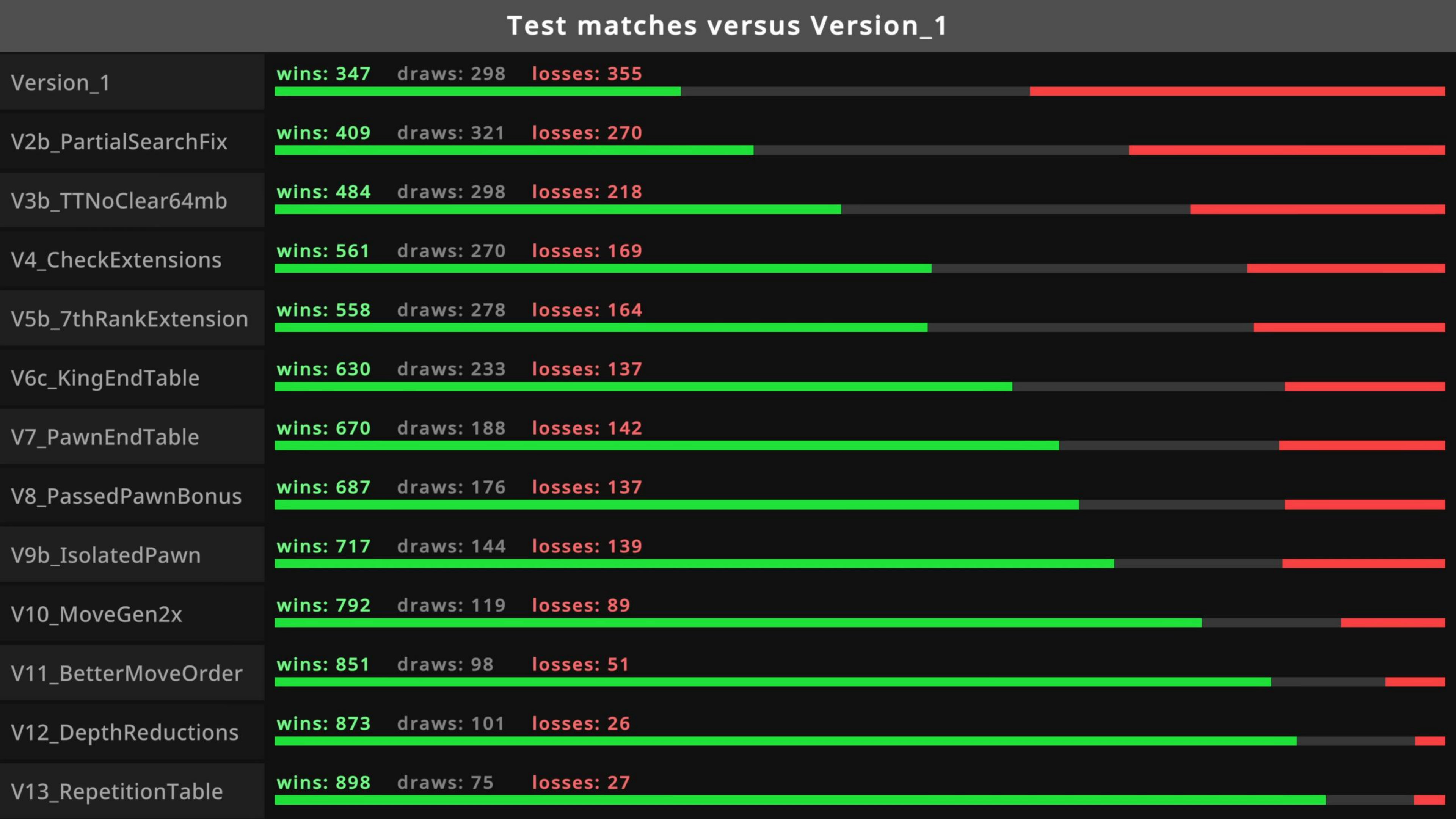
PLAYER_ONE = 0;  PLAYER_TWO = 1;
PLAYER_NONE = 2; PLAYER_DRAW = 3;
```

State Evaluation

```
eval(state, player) {
    let winner = state.winner();
    if      (winner === player)              { return 10000; }
    else if (winner === (player + 1) % 2)    { return -10000; }
    else if (winner === PLAYER_DRAW)         { return 0; }
    else if (winner === PLAYER_NONE)         { return Heuristic(); }
}
if (state.winner() != PLAYER_NONE || depth >= this.currentMaxDepth)
{
    return this.eval(state, this.maxPlayer);
}
```

Enhancements

- Two main types of enhancement
- Implementation efficiency
  - Speeds up implementation
  - Doesn’t affect nodes searched per depth
  - More nodes searched usually means stronger AI
- Search enhancements
  - Heuristic evaluation improvement
  - Techniques for guiding the tree search
  - Can result in fewer nodes searched per depth


Test matches versus Version_1
Version_1
wins: 347 draws: 298 losses: 355
V2b_PartialSearchFix
wins: 409 draws: 321 losses: 270
V3b_TTNoClear64mb
wins: 484 draws: 298 losses: 218
V4_CheckExtensions
wins: 561 draws: 270 losses: 169
V5b_7thRankExtension
wins: 558 draws: 278 losses: 164
V6c_KingEndTable
wins: 630 draws: 233 losses: 137
V7_PawnEndTable
wins: 670 draws: 188 losses: 142
V8_PassedPawnBonus
wins: 687 draws: 176 losses: 137
V9b_IsolatedPawn
wins: 717 draws: 144 losses: 139
V10_MoveGen2x
wins: 792 draws: 119 losses: 89
V11_BetterMoveOrder
wins: 851 draws: 98 losses: 51
V12_DepthReductions
wins: 873 draws: 101 losses: 26
V13_RepetitionTable
wins: 898 draws: 75 losses: 27

Tie-Breaking Scores

Tie-Breaking

- Sometimes your program will see far into the future and see it will win or lose no matter what
- In such cases, it may make stupid moves until the win or loss because it sees no difference in the value of states until then
- "I'm going to lose anyway so who cares what I do until I lose" However your opponent may not have figured out that it can actually win
- How can we ensure we always take best moves

Tie-Breaking

- Differentiate between wins and losses
  - Prefer to win as soon as possible
  - Prefer to lose as late as possible
- Tie-break the win/loss value by incorporating the turn number of the game (maximize score)
  - WinValue – TurnNumber = Win Faster
    (1000 – 40) > (1000 – 50)
  - LossValue + TurnNumber = Lose Slower
    (-1000 + 50) > (-1000 + 40)
- Or, Eval = [Value, TurnNumber]

State Depth Parity Effect

State Depth Parity

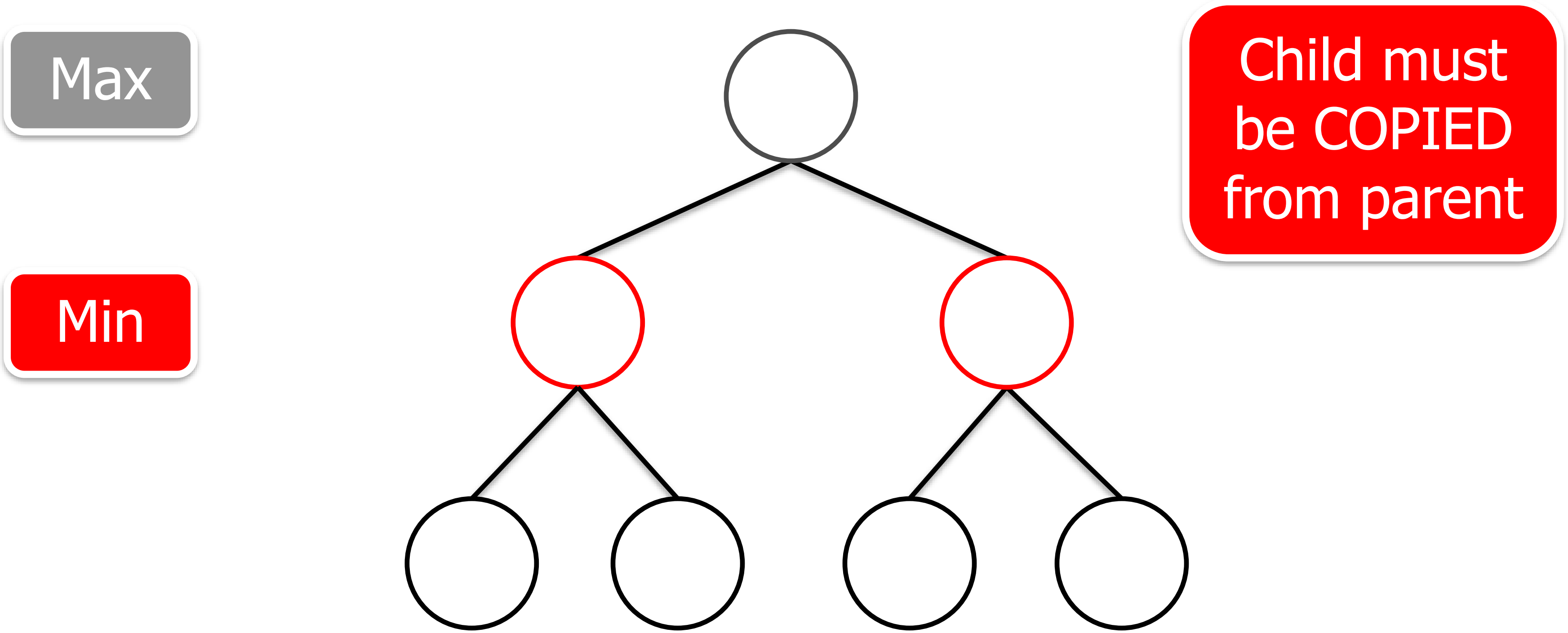
- Search tree depth parity
  - Even (0,2,4) – Maximizing Player Moving
  - Odd (1,3,5) – Minimizing Player Moving
- Evaluations on even/odd depths:
  - Even = Optimistic (maybe I just captured)
  - Odd = Pessimistic
- Can have interesting effects on evaluation

Avoiding State Copies

Avoid-Copy Optimization

- To build the search tree we issue actions to the current state to create a child state
- Child state is passed to next minimax depth
- Two issues:
  - When looping through actions, state is changed
  - Pass-by-value objects mean current state will be modified when child states are modified if it's not copied
- Solution: copy state when we create child

Avoid-Copy Optimization



Avoid-Copy Optimization

```
Function MiniMax(s, d, max)
    if (terminal(s) or d > maxD)
        return eval(s)
    if (max)          // maximizing player
        v = -infinity
        for (c in children(s))
            v = max(v, MiniMax(c, d+1, false))
        return v
    else              // minimizing player
        v = +infinity
        for (c in children(s))
            v = min(v, MiniMax(c, d+1, true))
        return v
```

Avoid-Copy Optimization

```
for a : state.getLegalActions()
   state.doAction(a)
   v = max(v, MiniMax(state, d+1, false))
```

This will NOT work

Avoid-Copy Optimization

This will work

```
for a : state.getLegalActions()
  child = state.copy()
  child.doAction(a)
  v = max(v, MiniMax(child, d+1, false))
```

Undo Action Optimization

- Copying can be very slow, depending on the size of the environment state
- Do we really need to copy?
- Some environments do not need to copy a state, if it is possible to UNDO a move
- How does this change our code?

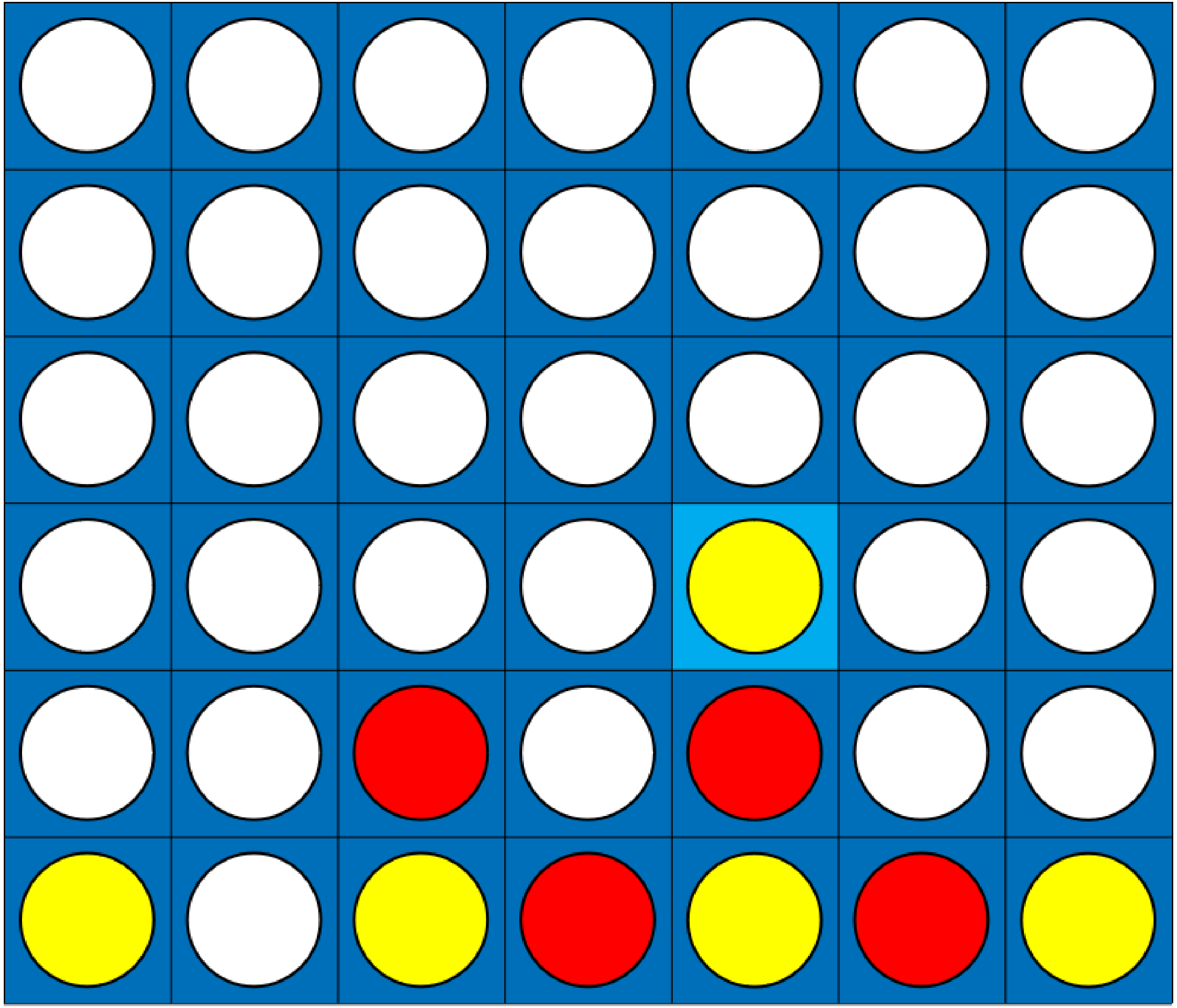
Undo Action Optimization

This will work best

```
for a : state.getLegalActions()
   state.doAction(a)
   v = max(v, MiniMax(state, d+1, false))
   state.undoAction(a)
```

Undo Action Optimization

- Not always possible to undo an action
  - What if pieces disappear (chess, checkers)
  - Random environments
- It will be possible in Assignment 3
  - Connect 4
  - Pieces always stay on board, none captured
  - Can just remove top piece from column



Move Ordering

Move Ordering

- With minimax, all actions are considered to generate all possible children of a given state
- With Alpha-Beta pruning, not all actions checked
- If better actions are checked first, the AB window is narrower and more cuts are made
- Ordering actions with a heuristic evaluation can potentially lead to more cuts
- No guarantee, must be evaluated experimentally

Search Extensions

Search Extensions

- Minimax looks equally deep for all actions
- Might be in our interest to look deeper for actions with interesting outcomes
- Examples
  - Chess – Look +1 depth if action leads to check
  - Connect 4 – Look deeper if 3 in a row?
- Not guaranteed – environment dependent

Bit Operation

Based Optimizations

Bit Operations

- In low level languages like C, we can work directly with bit data 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 ~

```
      010101101
~=    101010010
```

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
- Note: Size limit is the main bitset issue

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 = 0101010101 0111
// 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 – Storing Config/Options

- You may have seen this before:
  - `int options = OPTION_1 | OPTION_2`
  - `display = FULL_SCREEN | V_SYNC | NO_TITLE`
- Store integer option variables in bit set
  - `O1 = 0001, O2 = 0010, O3 = 0100`
  - `Options = O1 | O3 = 0101`
- Fast efficient way to store a SET of options

XOR Operator

- ```
    100101   a       100101 a      100101 a
  ^ 010101   b     ^ 000000 0    ^ 100101 a
    110000   c       100101 a      000000 0
  ```
- `a ^ b = c`
- `a ^ a = 0`
- `a ^ 0 = a`
- `a ^ a ^ b = b`
- `a ^ b = c, c ^ a = b, c ^ b = a`

- XOR OPERATOR IS REVERSIBLE (very important)
  ```

XOR Data Recovery

Disk 1	Disk 2	Disk 3	Disk 4	
1	0	1	0	
0	1	1	1	
1	1	1	0	
1	0	1	1	
1	1	0	0	
0	1	0	1	
1	0	1	0	

XOR Data Recovery

Disk 1	Disk 2	Disk 3	Disk 4	XOR Disk
1	0	1	0	
0	1	1	1	
1	1	1	0	
1	0	1	1	
1	1	0	0	
0	1	0	1	
1	0	1	0	

XOR Data Recovery

Disk 1	Disk 2	Disk 3	Disk 4	XOR Disk
1	0	1	0	0
0	1	1	1	1
1	1	1	0	1
1	0	1	1	1
1	1	0	0	0
0	1	0	1	0
1	0	1	0	0

XOR Data Recovery

Disk 1	DEAD	Disk 3	Disk 4	XOR Disk
1		1	0	0
0		1	1	1
1		1	0	1
1		1	1	1
1		0	0	0
0		0	1	0
1		1	0	0

XOR Data Recovery

Disk 1	XOR	Disk 3	Disk 4	XOR Disk
1		1	0	0
0		1	1	1
1		1	0	1
1		1	1	1
1		0	0	0
0		0	1	0
1		1	0	0

XOR Data Recovery

Disk 1	XOR	Disk 3	Disk 4	XOR Disk
1	0	1	0	0
0	1	1	1	1
1	1	1	0	1
1	0	1	1	1
1	1	0	0	0
0	1	0	1	0
1	0	1	0	0

XOR Data Recovery

Disk 1	Disk 2	Disk 3	Disk 4	
1	0	1	0	
0	1	1	1	
1	1	1	0	
1	0	1	1	
1	1	0	0	
0	1	0	1	
1	0	1	0	

Bit Boards

(do not recommend for A3 in js)

Bit Set Uses – Bit Boards!

- Many board games use 8x8 grids = 64
- One bit 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 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

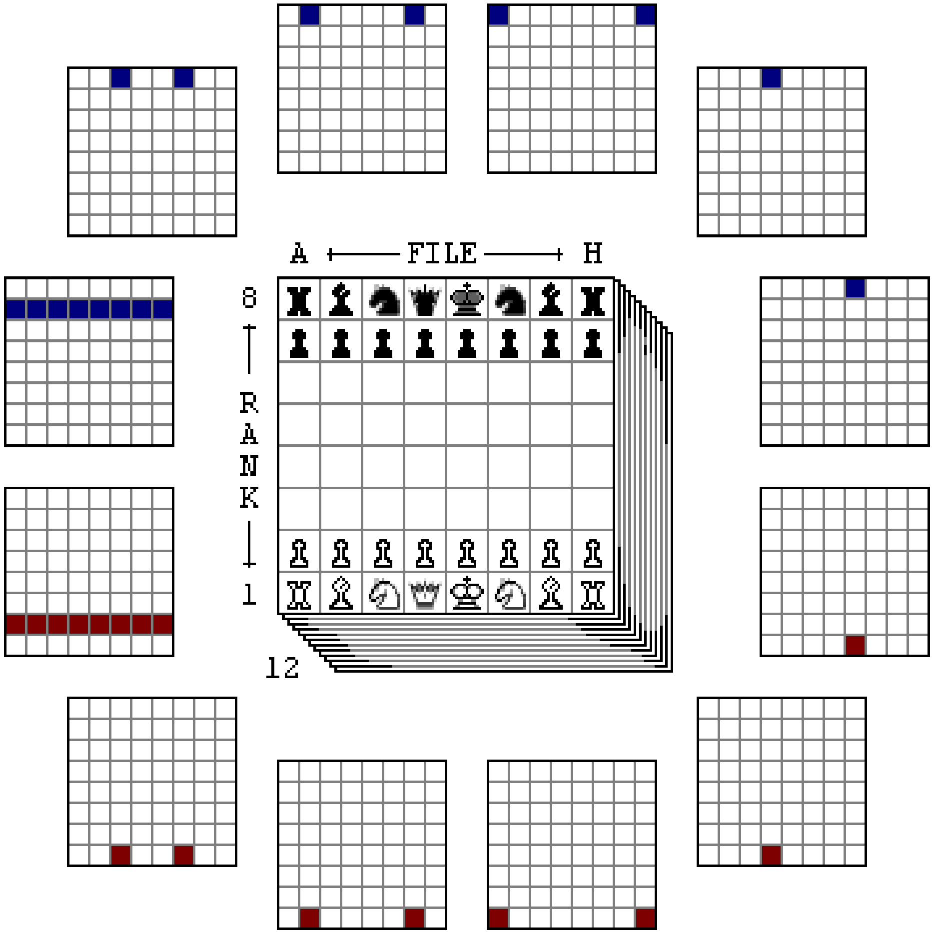
Bit Board Representation

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

```
northwest        north      northeast

          +7       +8       +9
               \    |    /
west      -1  <-    0   ->  +1      east
               /    |    \
          -9       -8       -7

southwest        south      southeast
```

Bit Board – Population Count

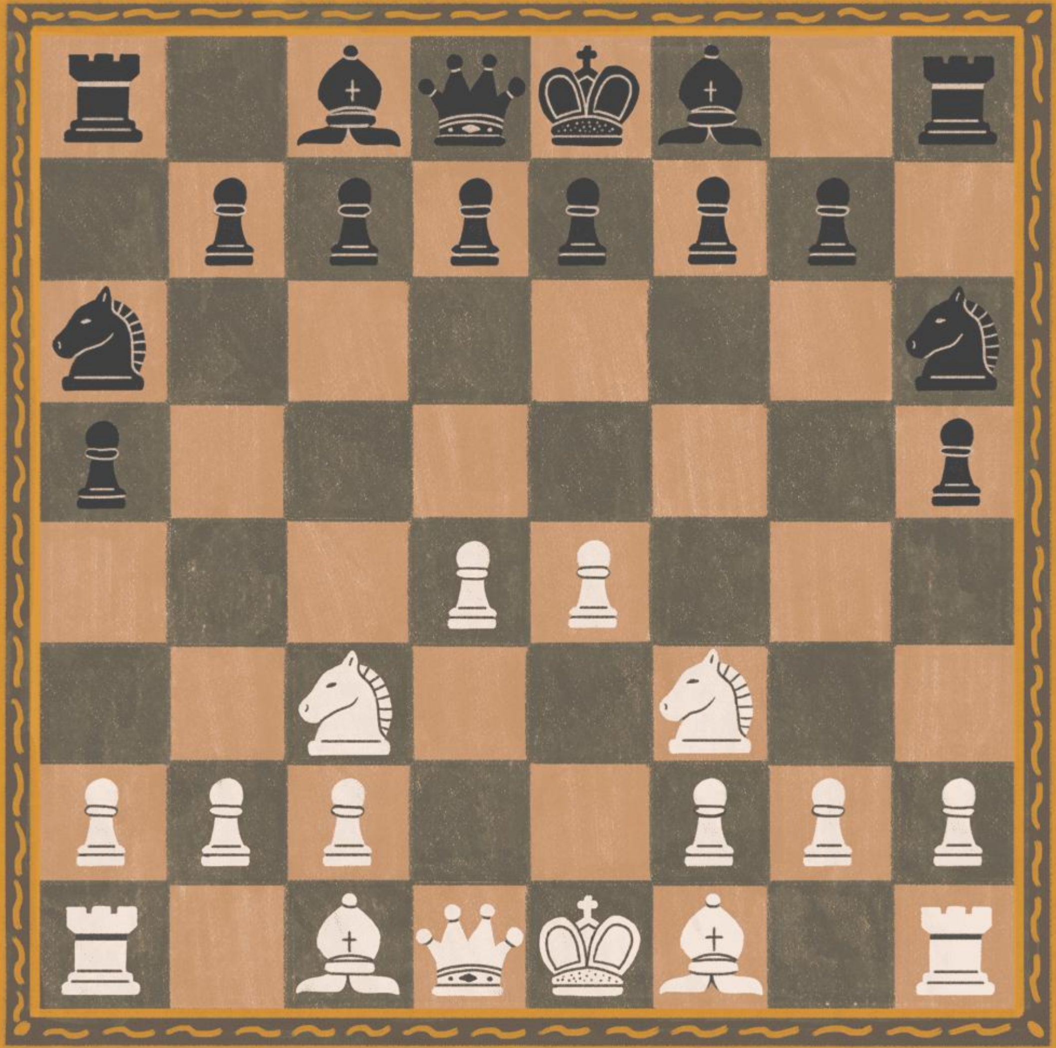
- How many bits are 1 in a set?
- MANY ways of doing this
- MANY optimized versions
- POPCOUNT hardware instructions

- 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 array cells

Transposition Tables

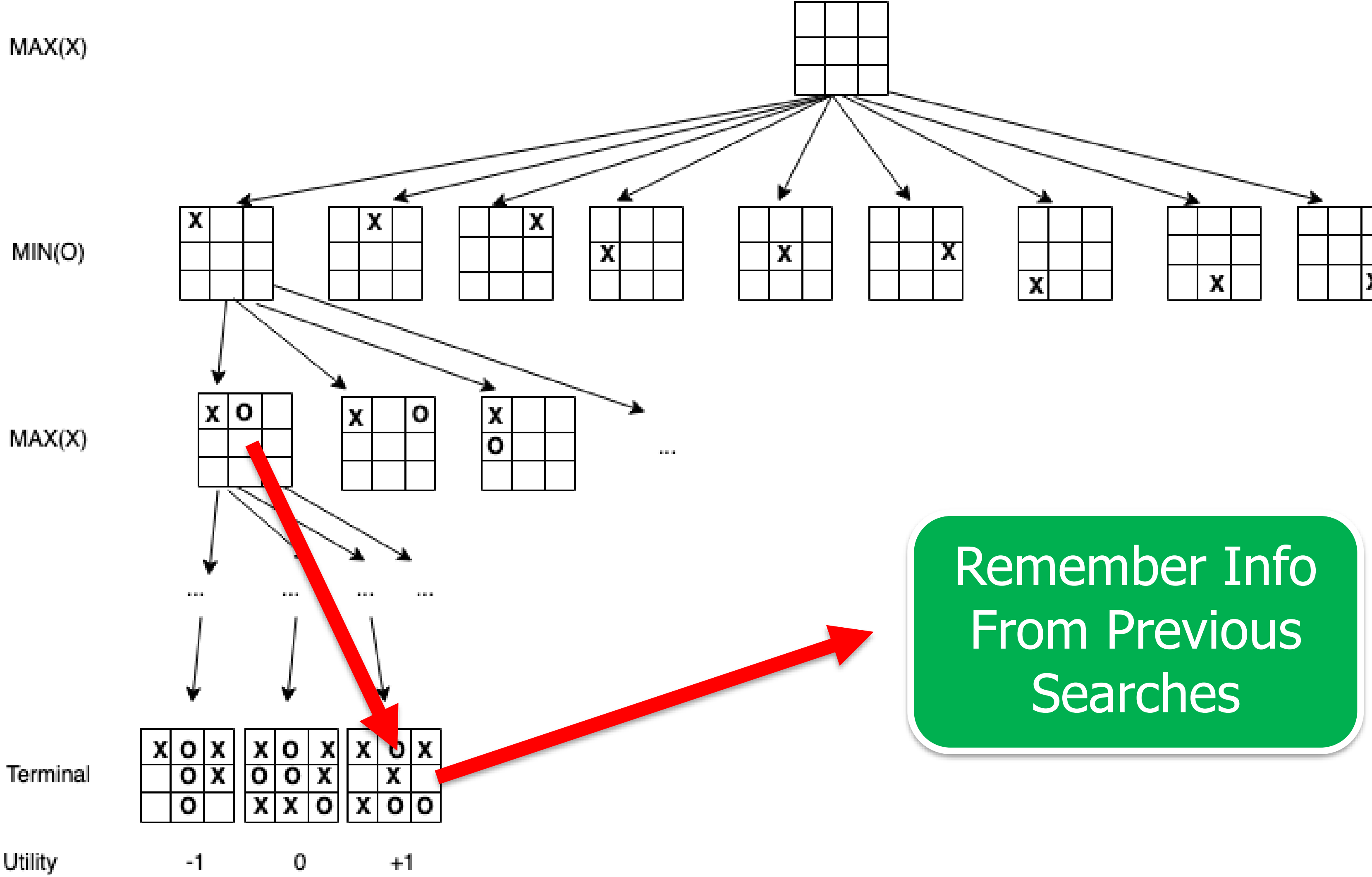


Transpositions

- In a given search tree, we may come across the same state many times
- Transposition causes:
  - Many ways to get to the same game state
  - IDAB will search through same paths while it resets to go deeper through the tree
- Duplicate states are called transpositions

Transposition Tables

- It may be advantageous to store information about states as we encounter them in the search tree
  - Heuristic evaluation (value)
  - Best move / action calculated
  - Search depth
  - Flag (real value, upper bound, lower bound)
  - Identifier, other info


MAX(X)
MIN(O)
MAX(X)
Terminal
Utility
-1
0
+1
Remember Info From Previous Searches

Transposition Tables

- Main idea of table is to store information about a state that we have seen before to use when we encounter it again
- Do we know the exact value of a state from a previous search? Then stop the search and return the value
- Similar with previous alpha/beta windows

	zonder transpositietabel		met transpositietabel		
diepte	# sec.	# nodes	# sec.	# nodes	% besparing
1	0	4	0	4	0,0
2	0	15	0	15	0,0
3	0	48	0	48	0,0
4	0	120	0	111	7,5
5	0	308	0	230	25,3
6	1	701	0	399	43,1
7	2	1760	0	649	63,1
8	3	3917	1	974	75,1
9	6	9557	1	1476	84,6
10	18	26424	2	2616	90,0
11	36	56886	3	3465	93,9
12	74	116670	3	4425	96,2
13	163	282463	4	5640	98,0
14			5	7054	-
15			6	8904	-
16			8	10979	-
17			10	14176	-
18			15	21445	-
19			19	28779	-
20			26	38765	-
21			31	48371	-
22			37	58422	-
23			42	66230	-
24			48	76113	-
25			56	89638	-
26			89	141190	-
27			146	237252	-

Transposition Table

- Theoretical optimization
- Potentially saves exponential computation
- Largest potential increase in playing strength comes from adding a TT
- Can be tough to implement, but the increase in performance is worth it

Transposition Table

- Complete details of TT implementation are out of scope for this course… BUT

- In order to store data, we need to associate a game state with an index
- If we could someone hash the game state to a table index we could store this data

Zobrist Hashing

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

Random Numbers

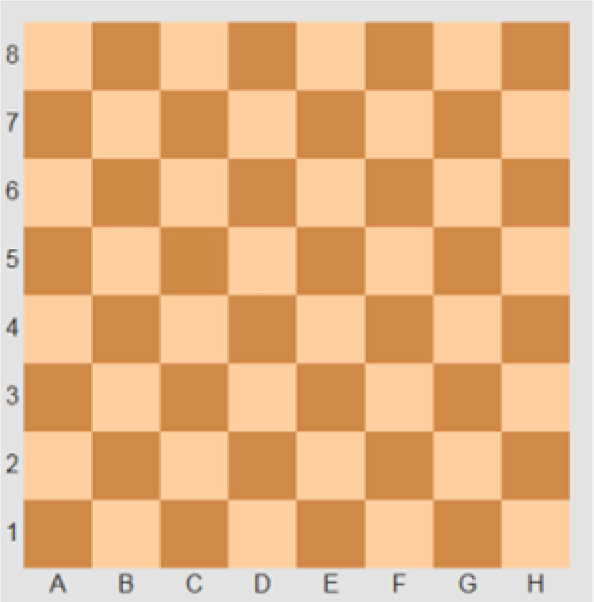
- Why do we use random numbers?
- If we have a large random number associated with every possible piece being at every possible location, it is unlikely to cause a hash collision with another
- XOR on those numbers creates a new number associated with the new state

Random Numbers

```
  010101110101010111010101011101
^ 100101101010101010101010101010
^ 010101010101010100101100101010
^ 100101001101010101010100010010
^ 101101010010101010101010101010

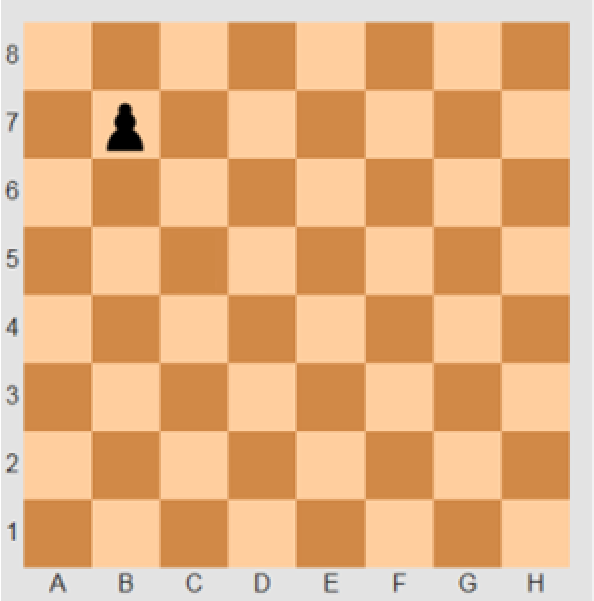
= likely to be unique for diff states
```

Zobrist Hashing



- Zobrist Hash = 0

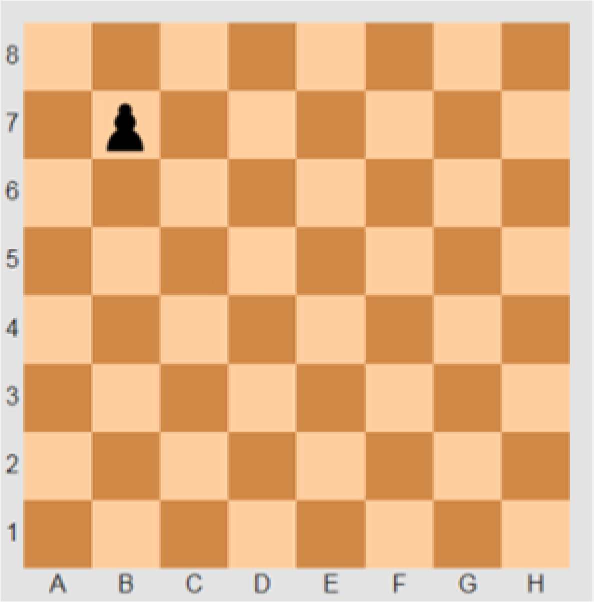
Zobrist Hashing



- `Z[Black][Pawn][B7]   = R4`
- `Zobrist Hash = 0 ^ R4`

Zobrist Hashing

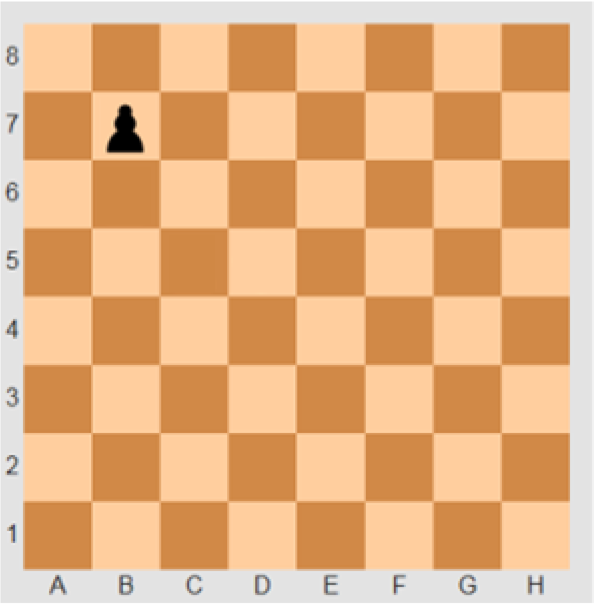
- Z[Black][Pawn][B7] = R4



- Zobrist Hash = R4

Zobrist Hashing

- Z[Black][Pawn][B7]   = R4



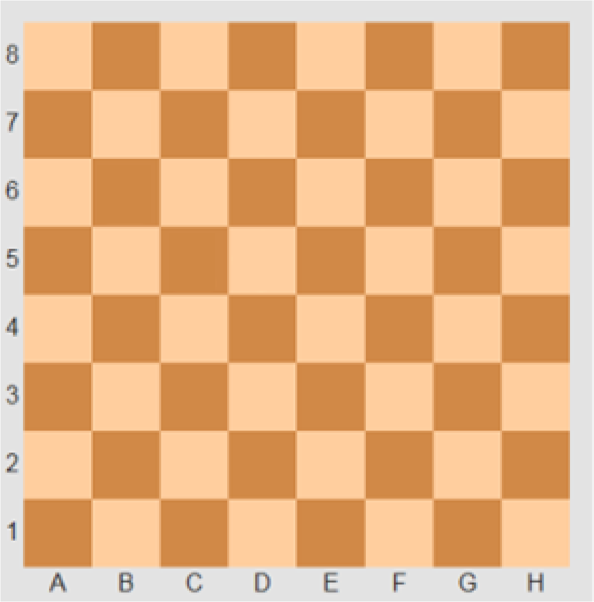
- Zobrist Hash = R4

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

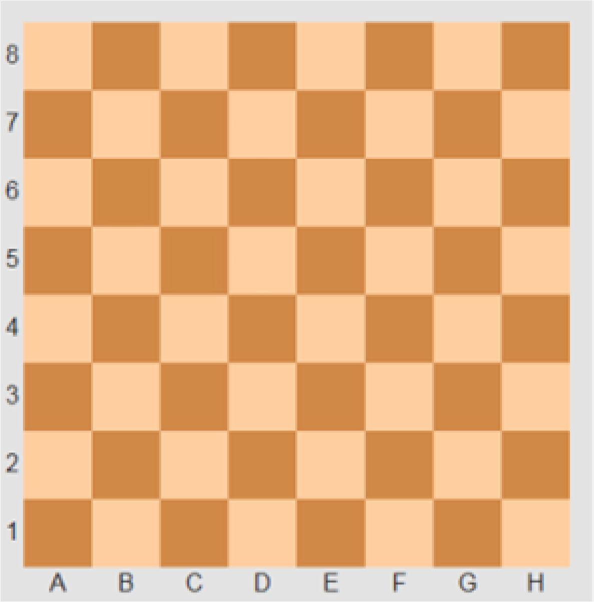
Zobrist Hashing

- Z[Black][Pawn][B7] = R4



- Zobrist Hash = R4 ^ R4

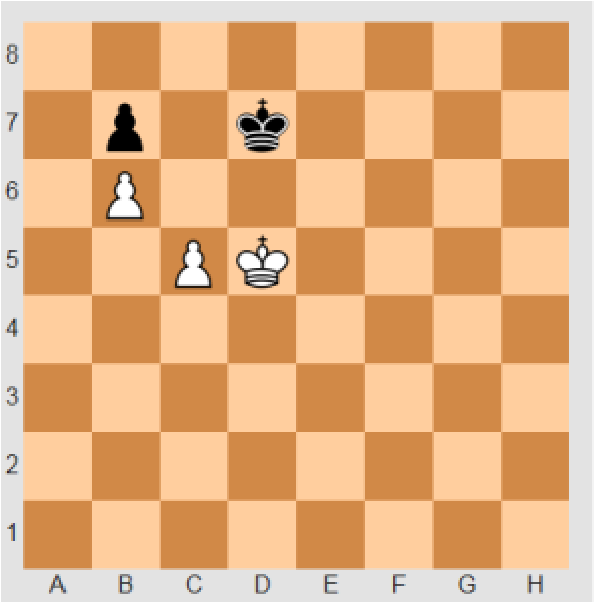
Zobrist Hashing



- Zobrist Hash = 0

Zobrist Hashing

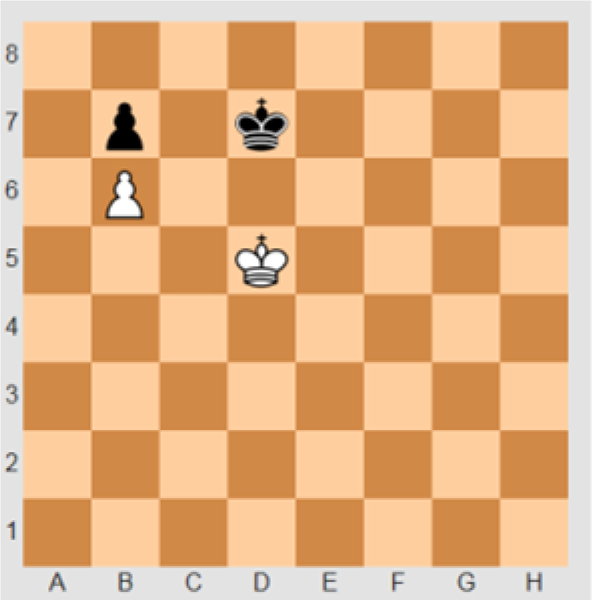
- Z[White][Pawn][B6] = R1
- Z[White][Pawn][C5] = R2
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5



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

Zobrist Hashing

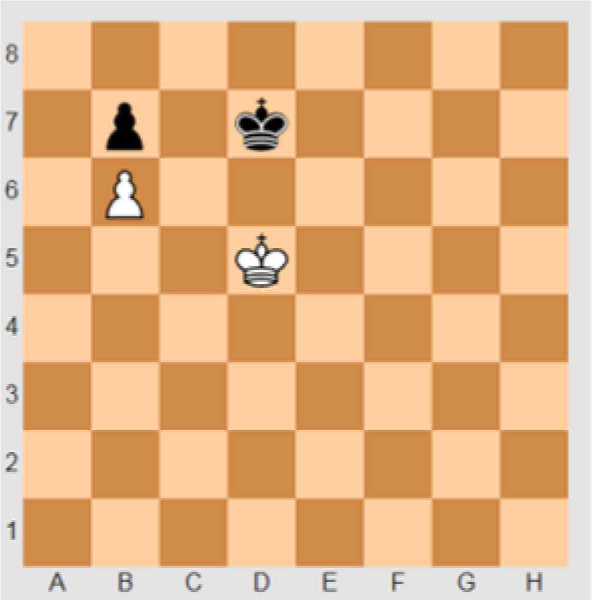
- Z[White][Pawn][B6]   = R1
- **Z[White][Pawn][C5]   = R2**
- Z[White][King][D5]   = R3
- Z[Black][Pawn][B7]   = R4
- Z[Black][King][D7]   = R5



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

Zobrist Hashing

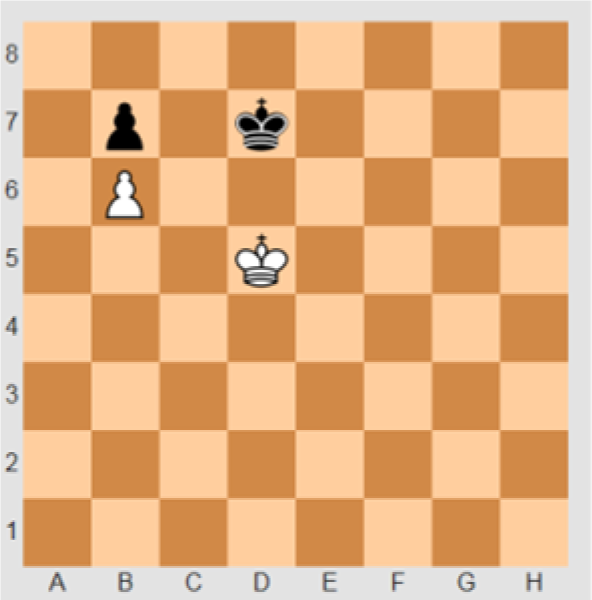
- `Z[White][Pawn][B6]  = R1`
- **`Z[White][Pawn][C5]  = R2`**
- `Z[White][King][D5]  = R3`
- `Z[Black][Pawn][B7]  = R4`
- `Z[Black][King][D7]  = R5`



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

Zobrist Hashing

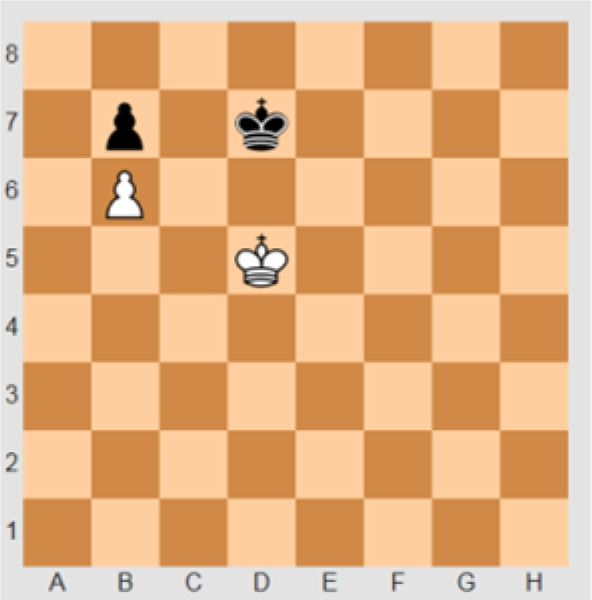
- Z[White][Pawn][B6] = R1
- **Z[White][Pawn][C5] = R2**
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5



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

Zobrist Hashing

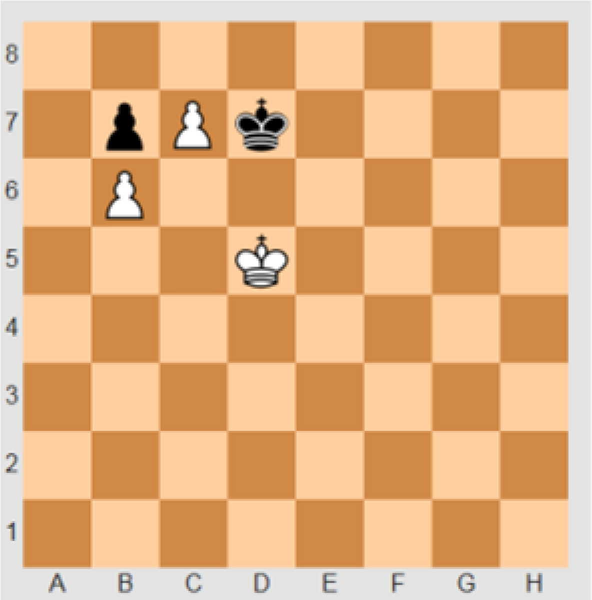
- Z[White][Pawn][B6] = R1
- Z[White][King][D5] = R3
- Z[Black][Pawn][B7] = R4
- Z[Black][King][D7] = R5



- 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)