Sizes of Games

Minimax(pos)

b = branching factor = avg moves possible

d = depth = avg length

pos ≈ b^d

If pos is terminal (game over), return value determined by rules ↪ check who won (+1 : P1, 0 : draw, −1 : P2)

Else if pos is P1's turn then return $\max_{pos \to pos'} MM(pos')$

Else return $\min_{pos \to pos'} MM(pos')$

	very rough est of # pos
Tic-Tac-Toe	$\leq 3^9 \approx 20000$
Mancala	$\leq \binom{49}{36} \approx 262$ billion
2-player Yahtzee	$\leq 4 \cdot 10^{18}$
Checkers	$\leq 10^{20}$
Chess	$\leq 10^{43}$
Go	$\leq 10^{172}$

What to do with games of high complexity?

heuristics - estimate of position value

Ex: checkers — % of remaining pieces that are black (scaled to (−1, 1))

0 8
0 4
heuristic = 1/3

chess — assign value to each type of piece (pawn = 1, knight, bishop = 3, rook = 5, queen = 9)

% of value of remaining pieces belonging to white (scaled)

Minimax(pos, h, depth) → depth bound; higher bounds yield results closer to unbounded MM

If pos is terminal, return value(pos)

If depth == 0, return h(pos)

Else if pos is P1's turn then return $\max_{pos \to pos'} MM(pos', h, depth-1)$

Else if pos is P1's turn then return $\max_{pos \to pos'} MM(pos', h, depth-1)$

Else return $\min_{pos \to pos'} MM(pos', h, depth-1)$

Negamax(pos, h, depth, sign)

If pos is terminal, return value(pos) · sign

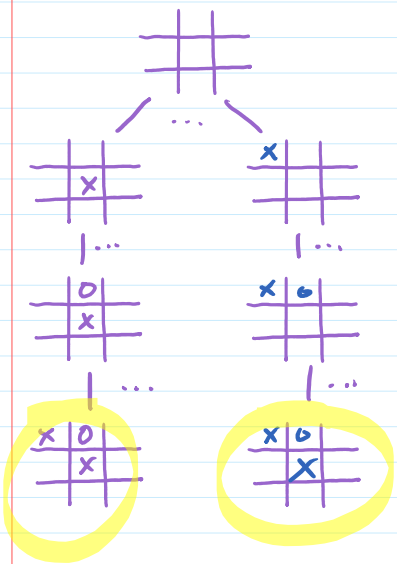
If depth == 0, return h(pos) · sign

Else return $\max_{pos \to pos'} -MM(pos', h, depth-1, -sign)$

Iterative Deepening - to get value from MM in set time w/ depth high

depth ← 2
while not out of time
do MM(pos, h, depth)
depth ← depth + 1
return value, best move returned by last call to MM that finished

Transposition Table (memo)



multiple paths to same pos

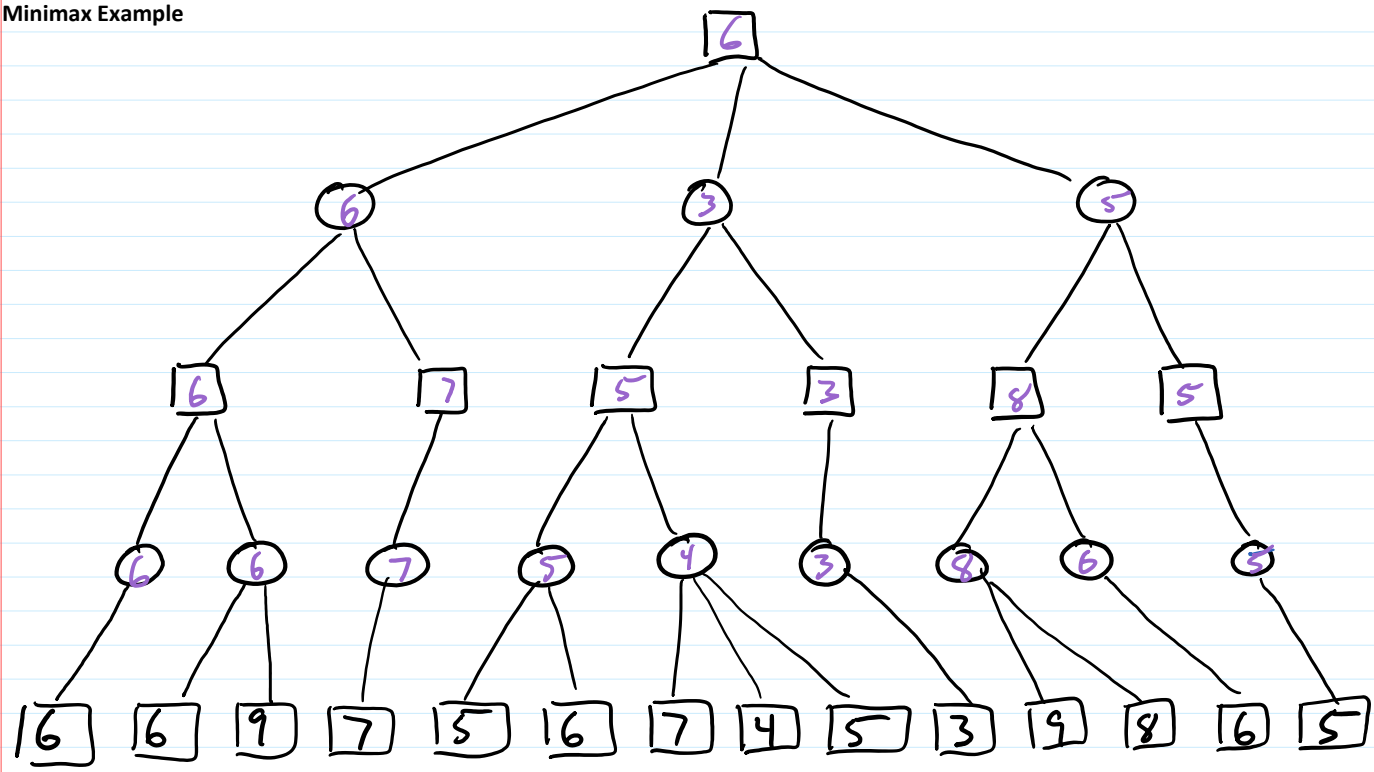
to un ounde

If pos is terminal, return value(pos)

If depth == 0, return h(pos)
if pos in TT return TT[pos]
Else if pos is P1's turn then ~~return~~ TT(pos) = $\max_{pos \to pos'} MM(pos', h, depth-1, TT)$
return TT(pos)
Else ~~return~~ TT(pos) = $\min_{pos \to pos'} MM(pos', h, depth-1, TT)$
return TT[pos]

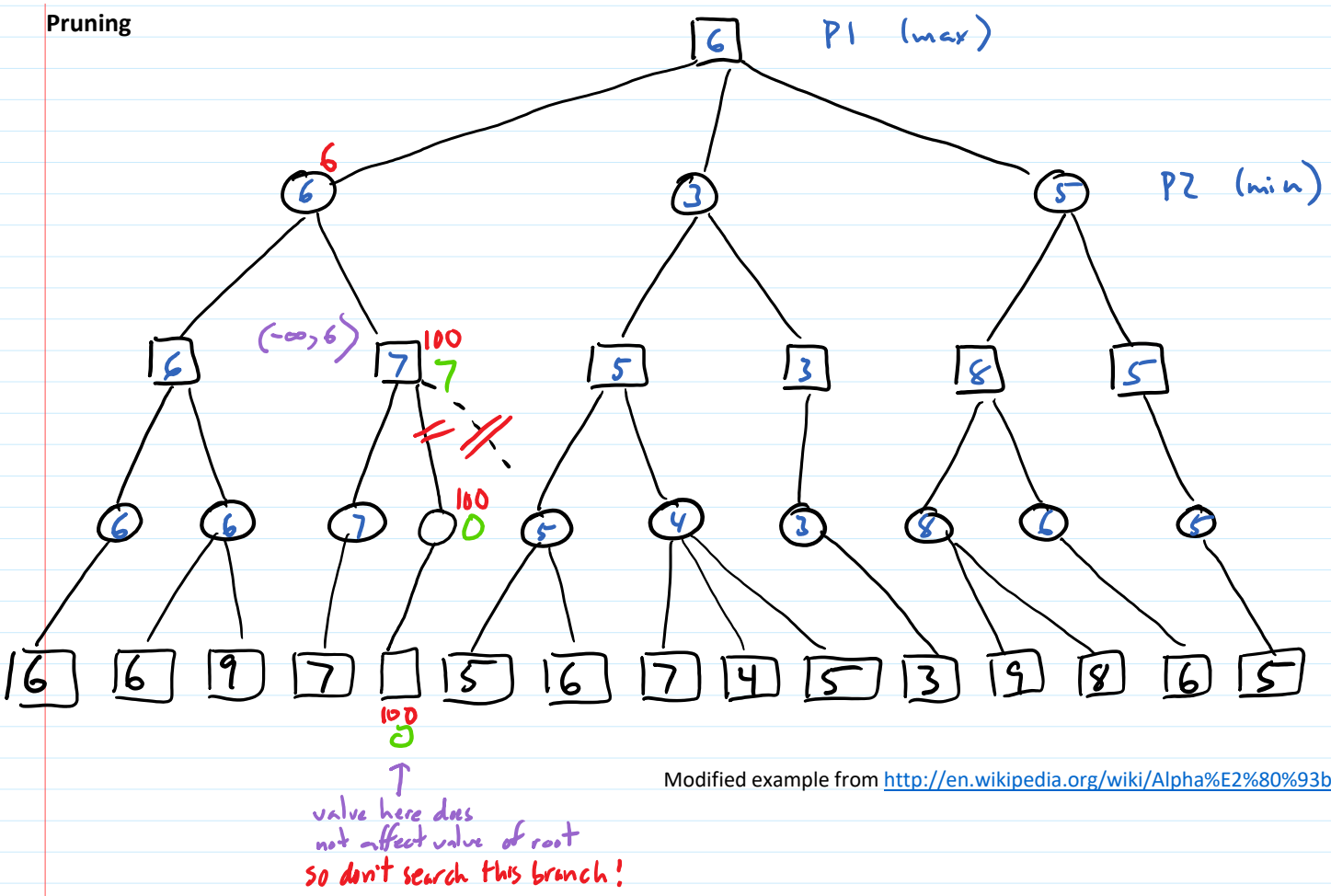
(memory intensive)
↳ replacement policy?

Minimax Example



Modified example from http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Pruning



Modified example from http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Alpha-Beta Pruning

bounds – if MM value in (α, β), want exact value
if $\leq \alpha$ or $\geq \beta$, don't need exact value

Alpha-Beta $(p, \alpha, \beta, depth, h)$ returns

value (p)	if p terminal
$h(p)$	if $depth = 0$
$MM(p)$	if $\alpha \leq MM(p) \leq \beta$
lower bound on $MM(p)$ some a s.t. $\beta \leq a \leq MM(p)$	if $MM(p) \geq \beta$
upper bound $MM(p)$ some b s.t. $MM(p) \leq b \leq \alpha$	if $MM(p) \leq \alpha$

postconditions

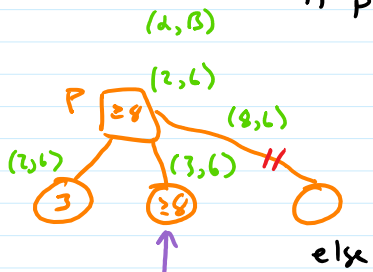
value from MM using same depth, h

if $depth = 0$ then return heuristic (p)

if p is terminal then return value (p)

if p is a max position
$a \leftarrow -\infty$
for each position p' reachable in one move from p while $\alpha < \beta$
$a \leftarrow \max(a, AB(p', \alpha, \beta, depth-1, h))$
$\alpha \leftarrow \max(\alpha, a)$
return a fail-soft (fail-hard returns α after pruning)

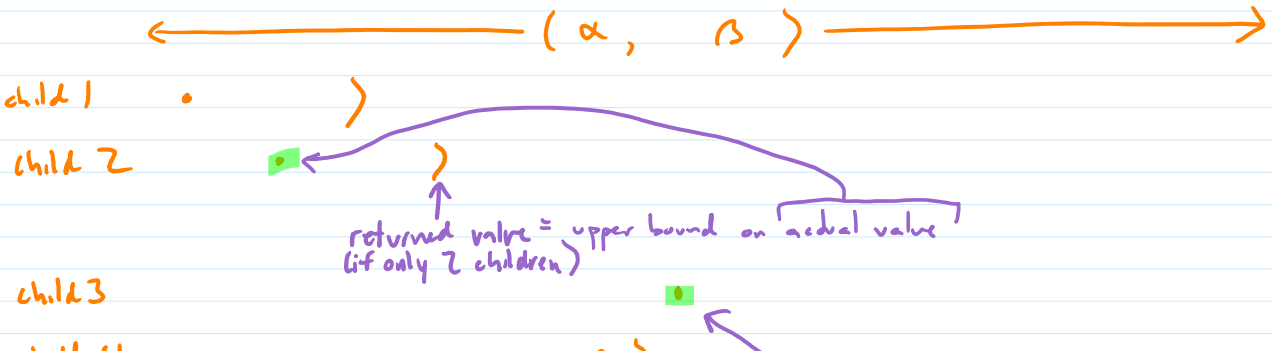
$(\alpha, \beta) = (2, 6)$
1st p' 3, 2nd p' 8



$b \leftarrow \infty$
for each position p' reachable in one move from p while $\alpha < \beta$
$b \leftarrow \min(b, AB(p', \alpha, \beta, depth-1, h))$
$\beta \leftarrow \min(b, \beta)$
return b

start with call $AB(curr, -\infty, \infty, depth, h)$ on current pos curr
postconditions guarantee returned value is $= MM(curr, depth, h)$

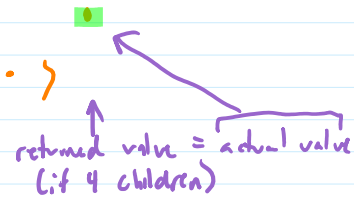
if could guess value, could start w/narrower window
$AB(curr, 2, 10, depth, h)$ returns 0
$AB(curr, 0, 10, depth, h)$ and widen window if guess was wrong
⋮



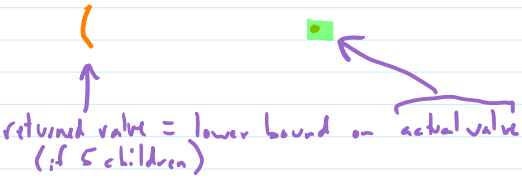
(if only 2 children)

child 3

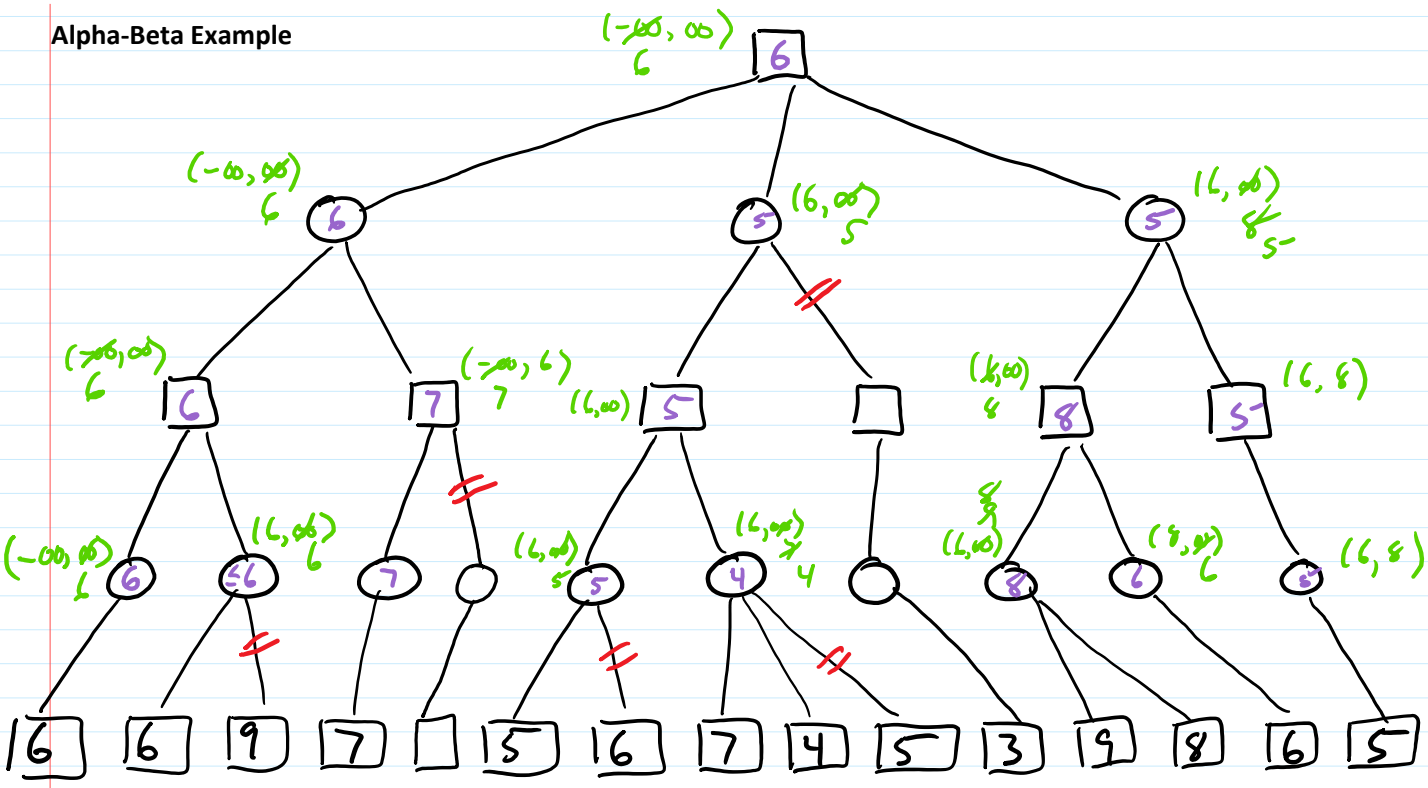
child 4



child 5

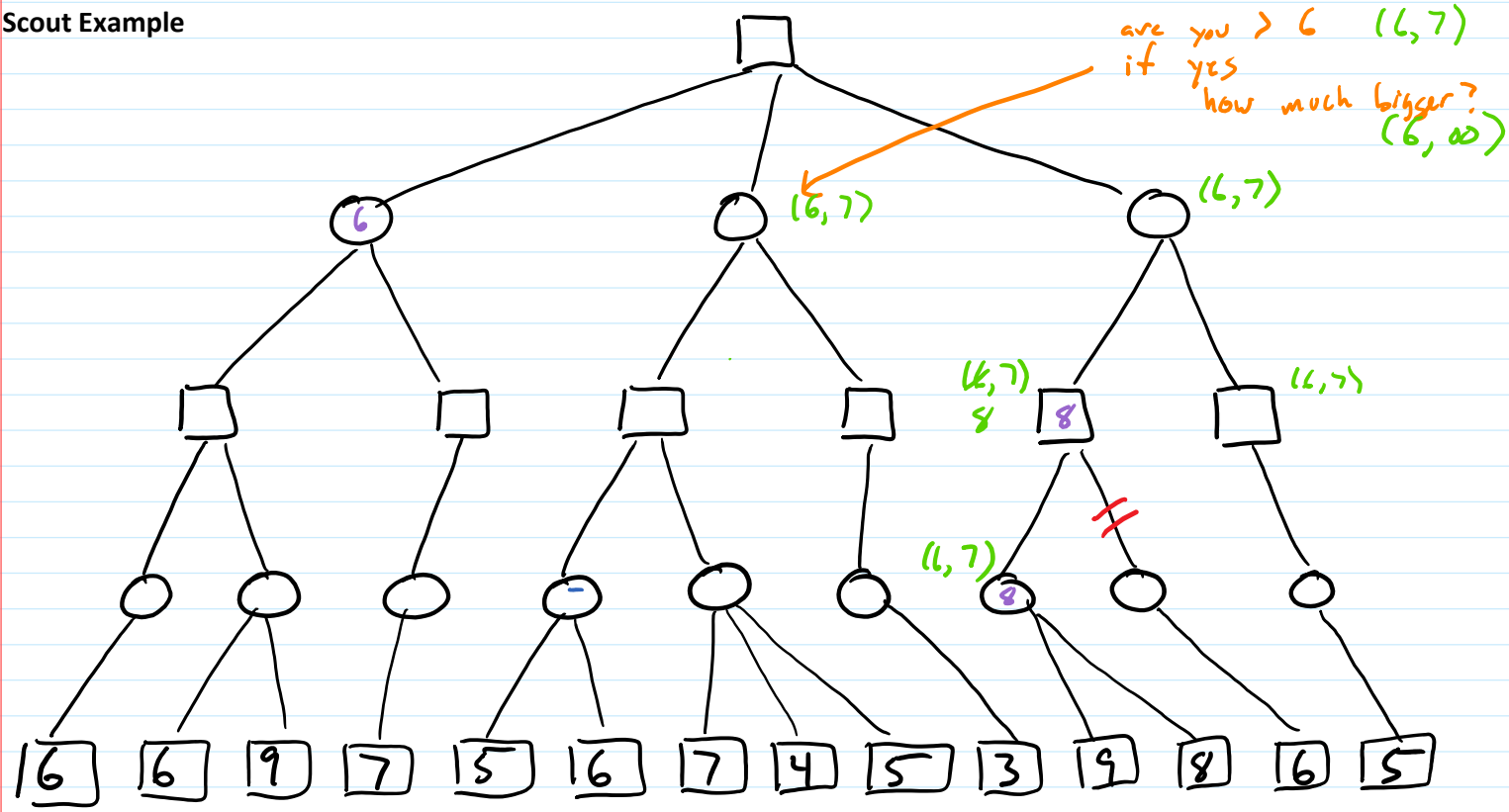


Alpha-Beta Example



Modified example from http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Scout Example



```
Scout (p, α, β, depth, h)
    if p is terminal then return value(p)
    if depth = 0 then return heuristic(p)
    if p is a max position
        best ← -∞
        for each reachable position p' ∧ and while α < β      (in order of ↓ goodness)
            if p' is first pos
                score ← Scout(p', α, β, depth-1, h)
            else                                              (null window (assuming integers))
                score ← AB(p', α, α+1, depth-1, h)            (already passing null window, so no reason for Scout)
                if α < score < β
                    score ← Scout(p', score, β, depth-1, h)   (already know value(pos) ≥ score)
            best ← max(best, score)
            α ← max(best, α)
        return best                                           (or fail-hard version)
    else
        ⋮ min position; symmetric
        ⋮
```