Lecture notes for *Principles of Artificial Intelligence* (COMS 4720/5720)
Yan-Bin Jia, Iowa State University

Alpha-Beta Cutoff

Outline

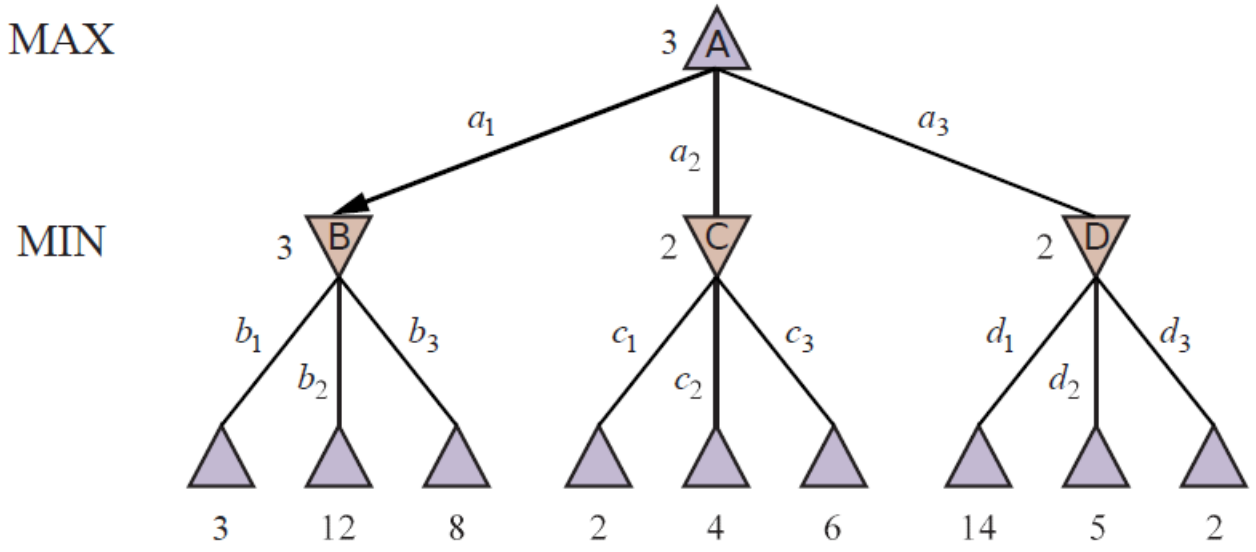
I. Alpha-beta pruning

II. Alpha-beta pruning algorithm

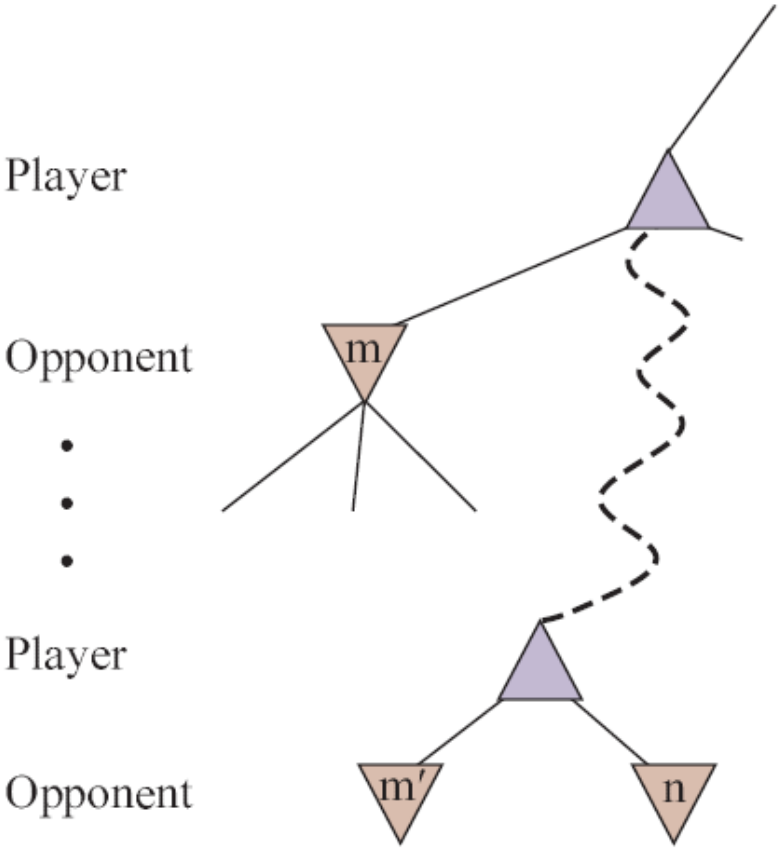
* Figures/images are from the textbook site (or by the instructor) . Otherwise, the source is specifically cited unless citation would make little sense due to the triviality of generating such an image.

I. Alpha-Beta Cutoff

- ♣ #states is exponential in the depth of the game tree.
- ♦ But we can often compute the correct minimax decision by pruning large parts of the tree that do not affect the outcome.



Node Pruning



The player will not move to node n if it has a better choice

- either at the same level (e.g., m')
- or at any node (e.g., m) higher up in the tree.

Prune n once we have found enough about it to reach the above conclusion.

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

α = the *highest*-value (i.e., the best choice) so far along a path for the player MAX. // MAX's best score so far.

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

α = the *highest*-value (i.e., the best choice) so far along a path for the player MAX. // MAX's best score so far.

eventual value $\geq \alpha$ "at least"

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

$\alpha =$ the *highest*-value (i.e., the best choice) so far along a path for the player MAX. // MAX's best score so far.

eventual value $\geq \alpha$ "at least"

$\beta =$ the *lowest*-value (i.e., the best choice) so far along a path for the player MIN. // MIN's best score so far.

eventual value $\leq \beta$ "at most"

Alpha and Beta Values

Alpha-beta pruning gets its name from two extra parameters α, β

α = the *highest*-value (i.e., the best choice) so far along a path for the player MAX. // MAX's best score so far.

eventual value $\geq \alpha$ "at least"

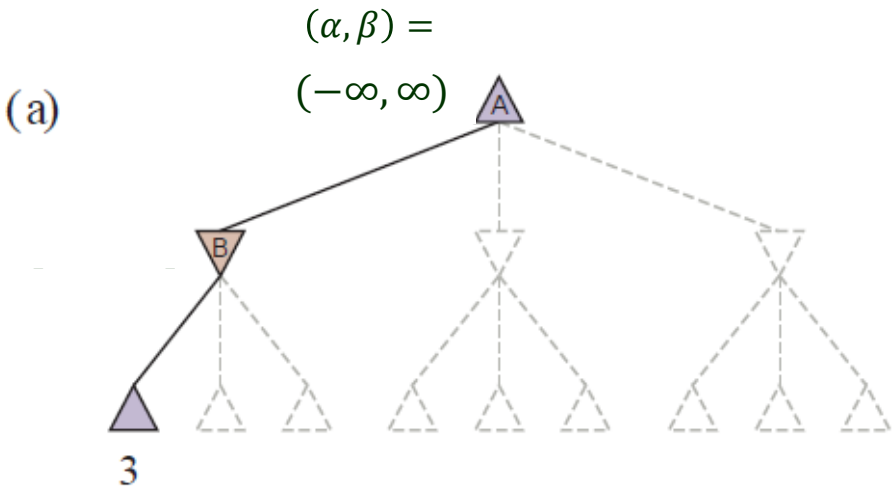
β = the *lowest*-value (i.e., the best choice) so far along a path for the player MIN. // MIN's best score so far.

eventual value $\leq \beta$ "at most"

- Update the values of α and β as the search goes along.
- Prune the remaining branches
  - at a MIN node with current value $\leq \alpha$ (because its parent, a MAX node, has from another child or is passed on the value α already).
  - at a MAX node with current value $\geq \beta$ (because its parent, a MIN node, has or is passed on the value β already).

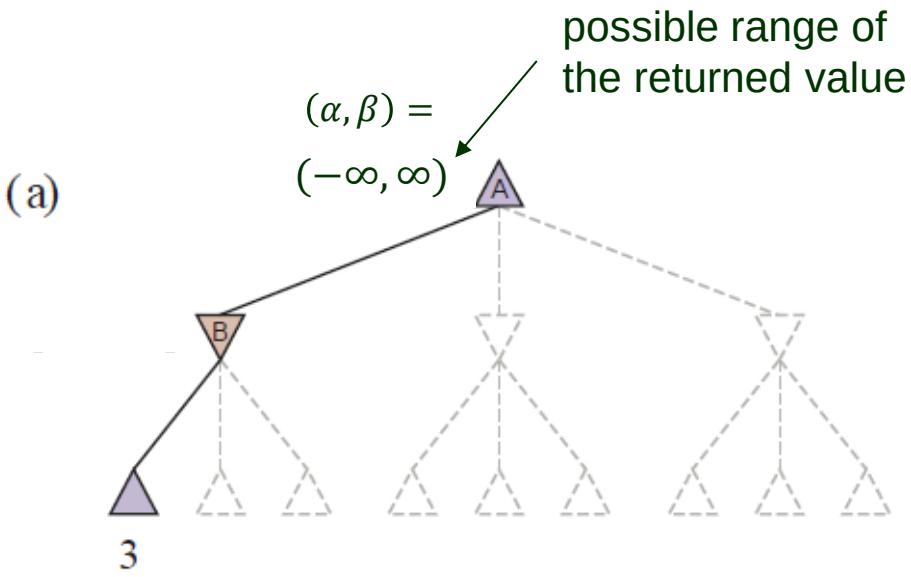
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



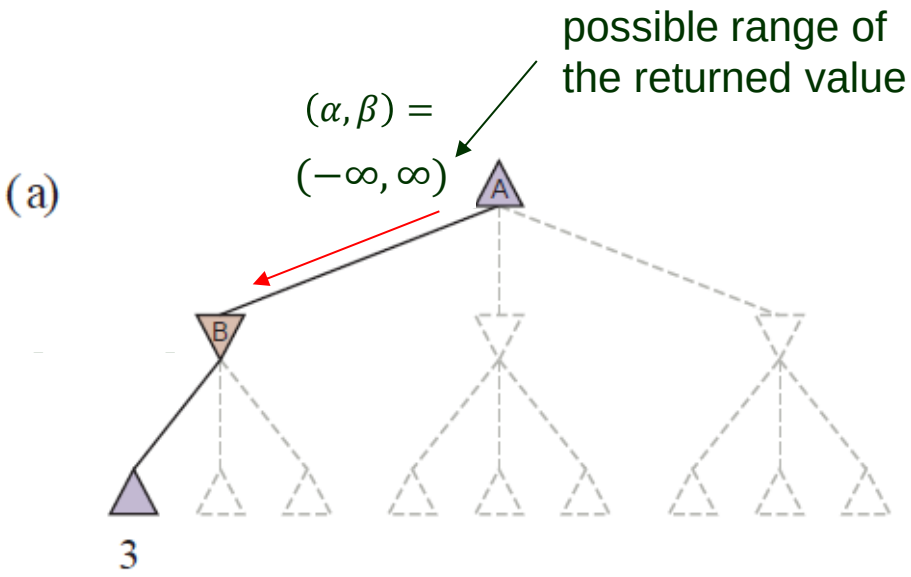
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



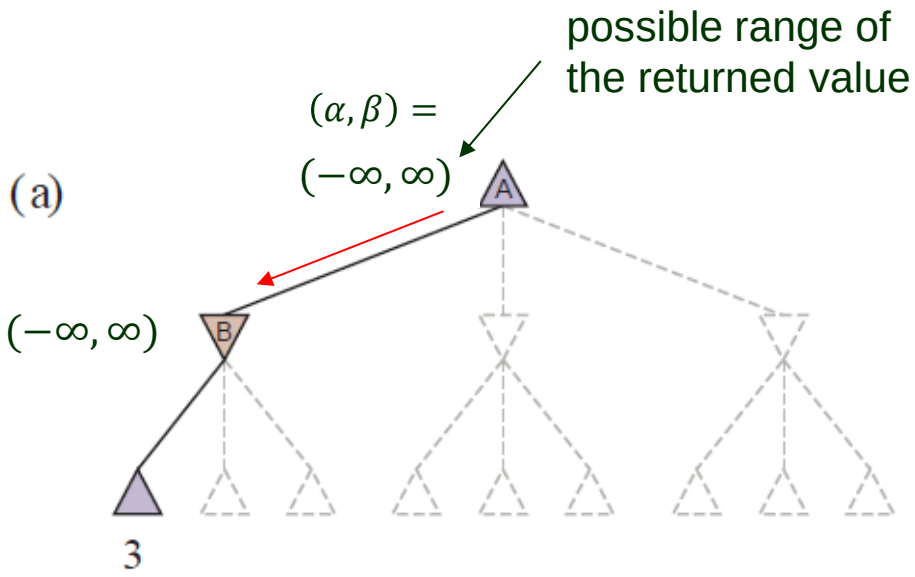
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



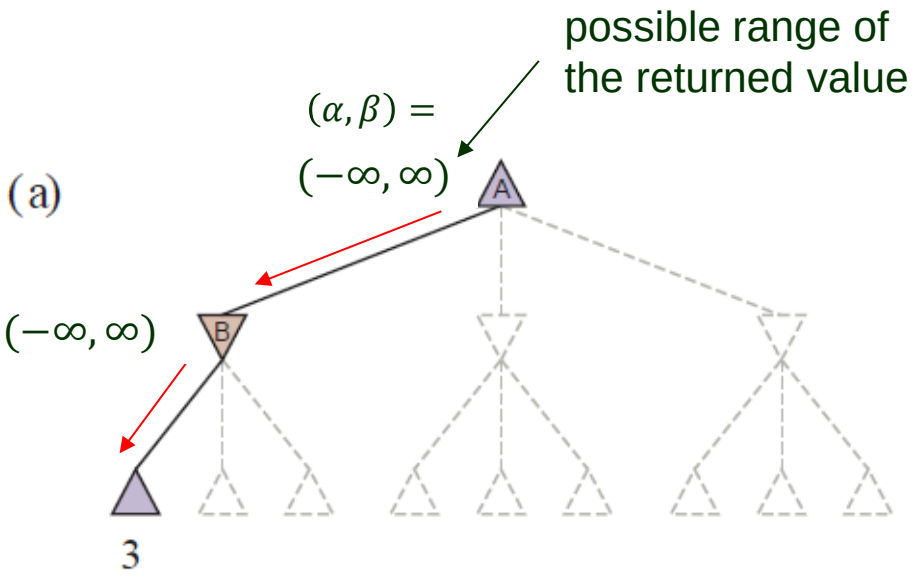
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



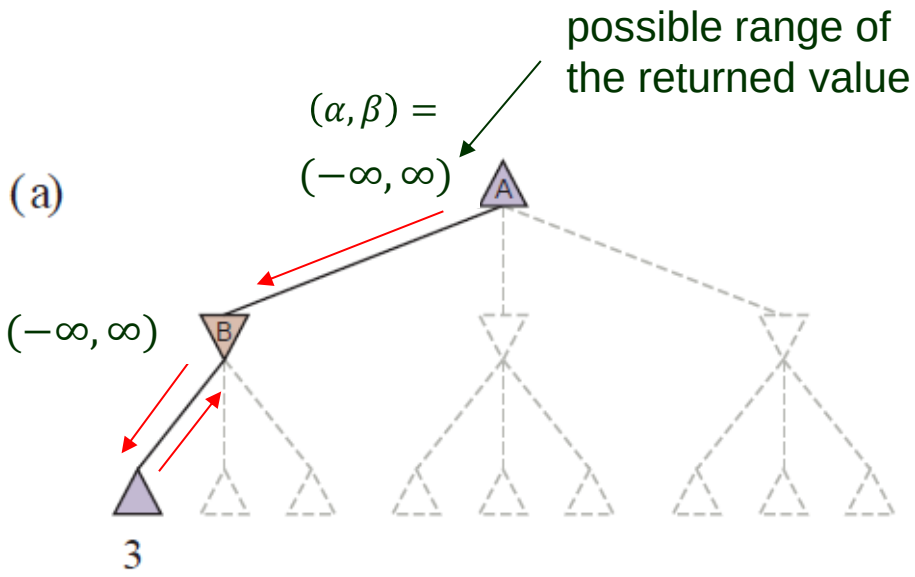
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



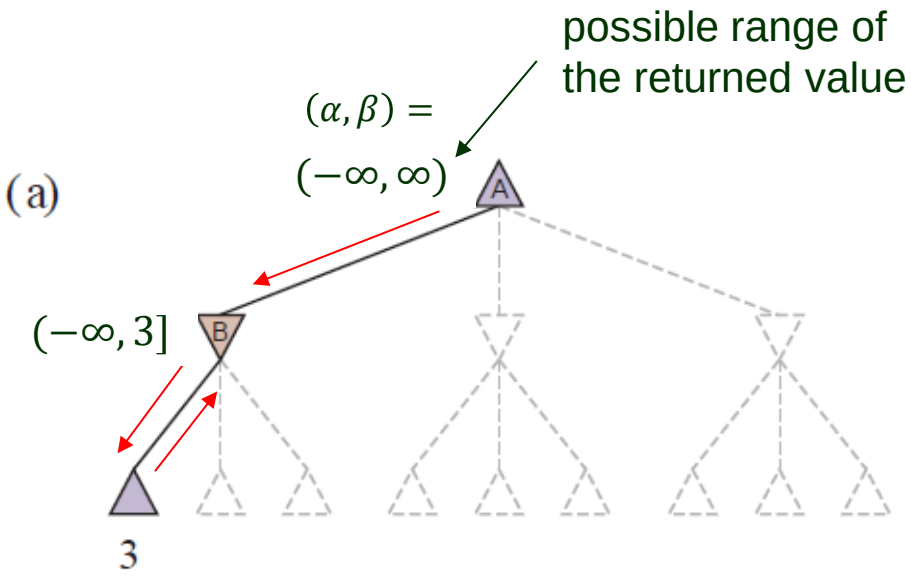
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



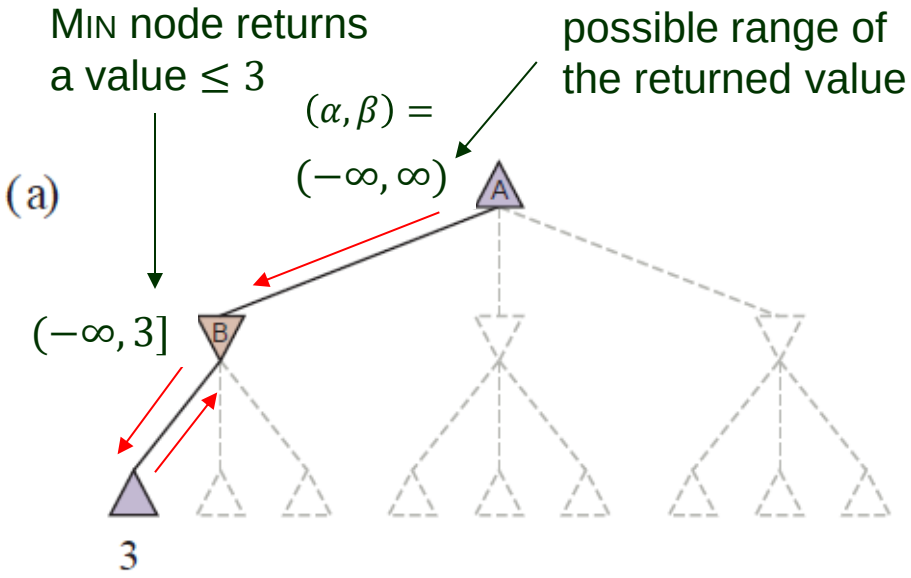
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



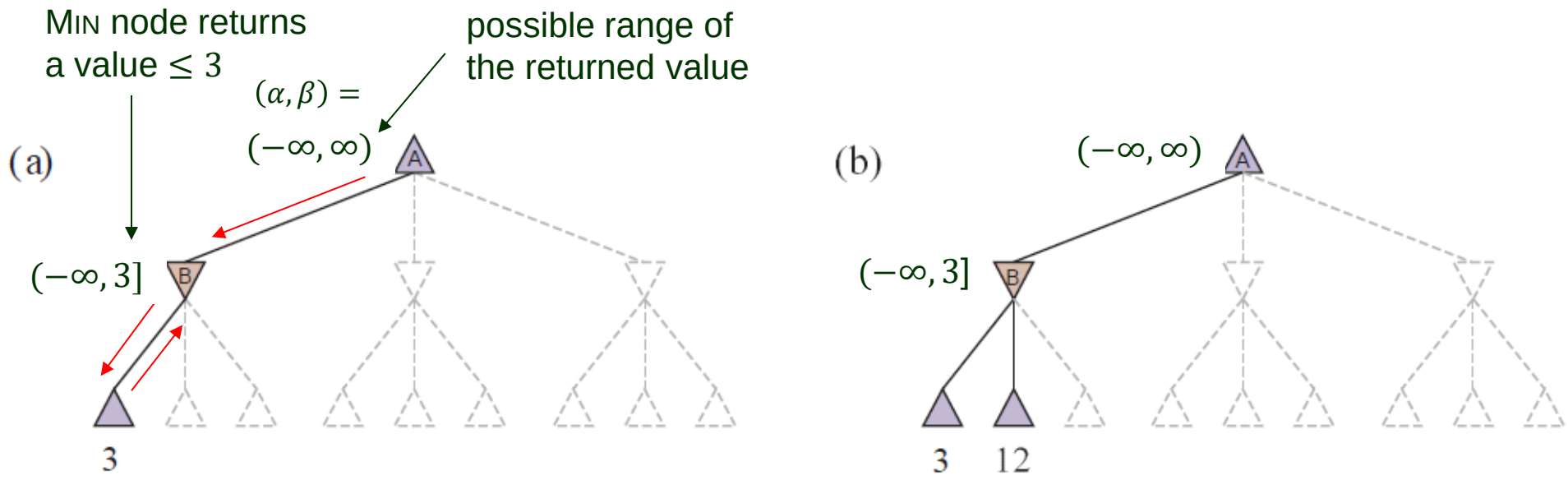
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



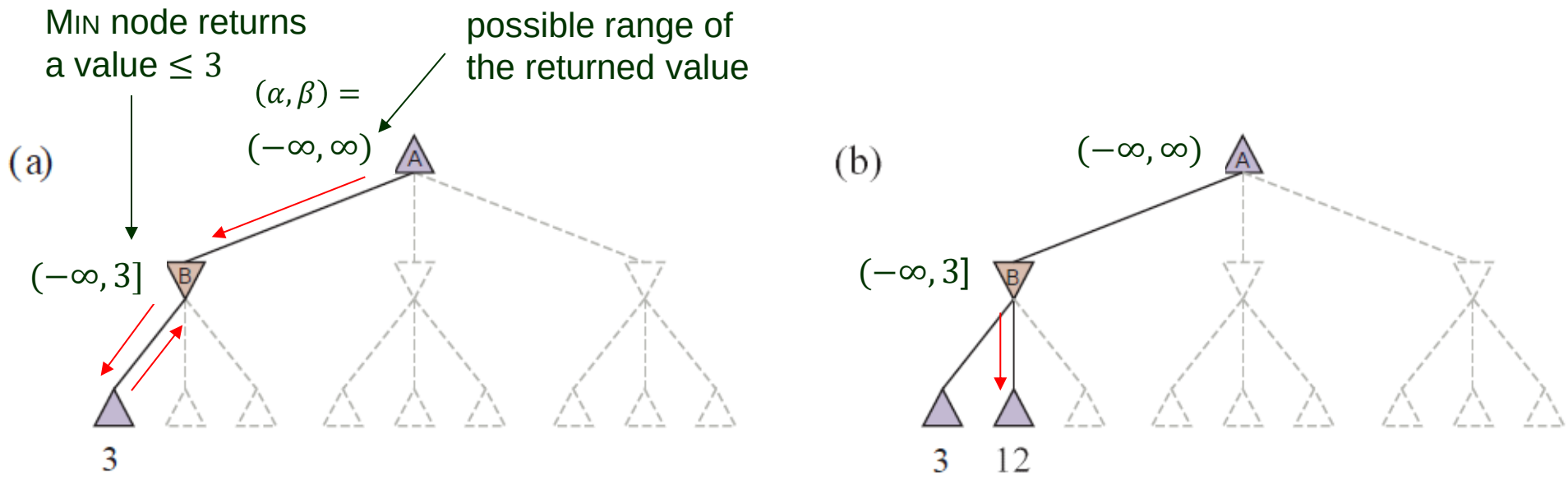
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



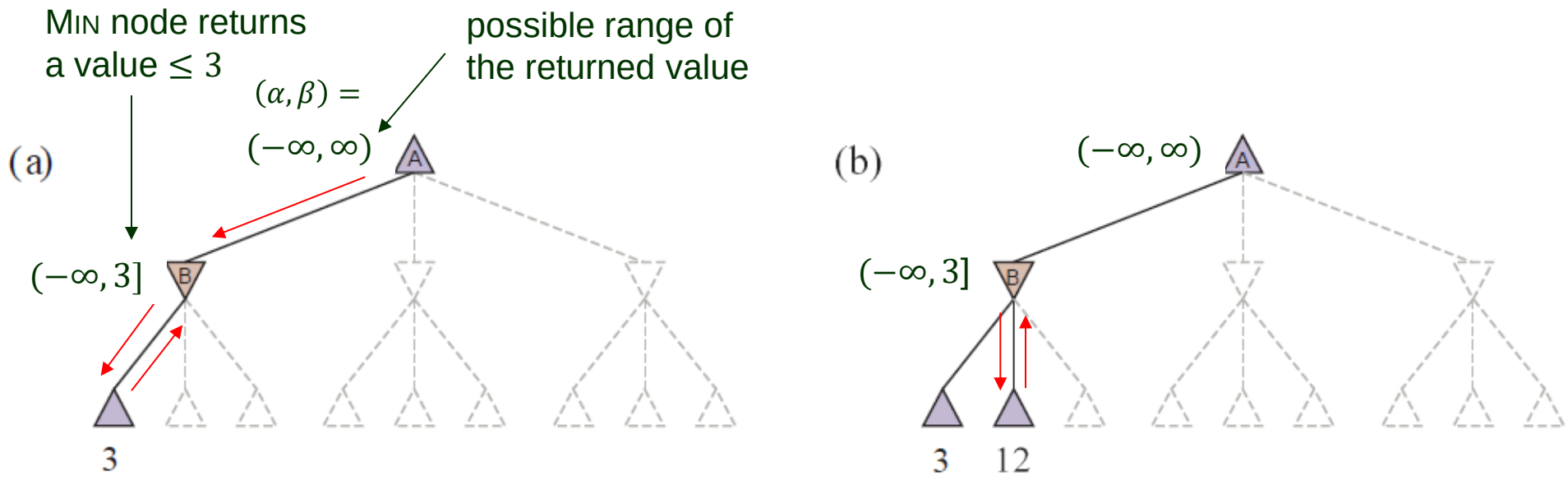
Re-examining the Game Tree

Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.

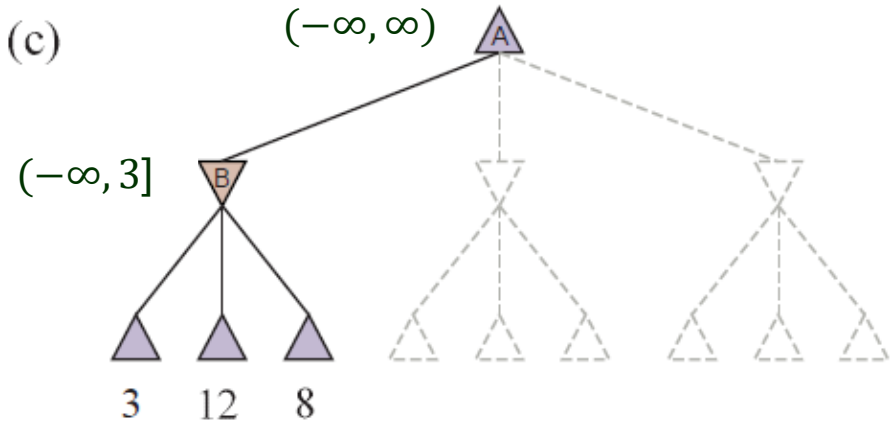


Re-examining the Game Tree

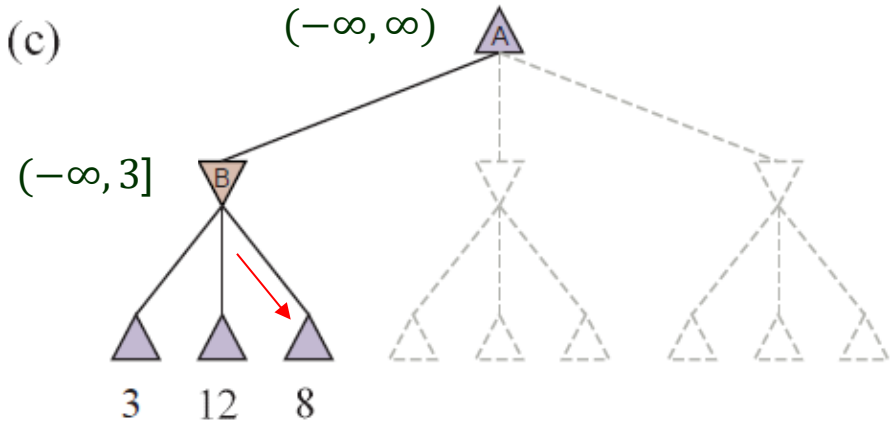
Fig. 5.5 in the 4th edition of the textbook *incorrectly executes* the algorithm ALPHA-BETA-SEARCH in Fig. 5.7.



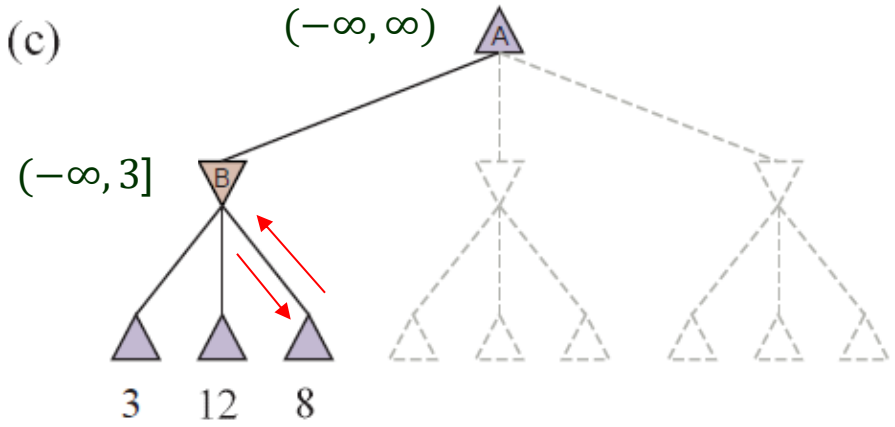
Cont'd



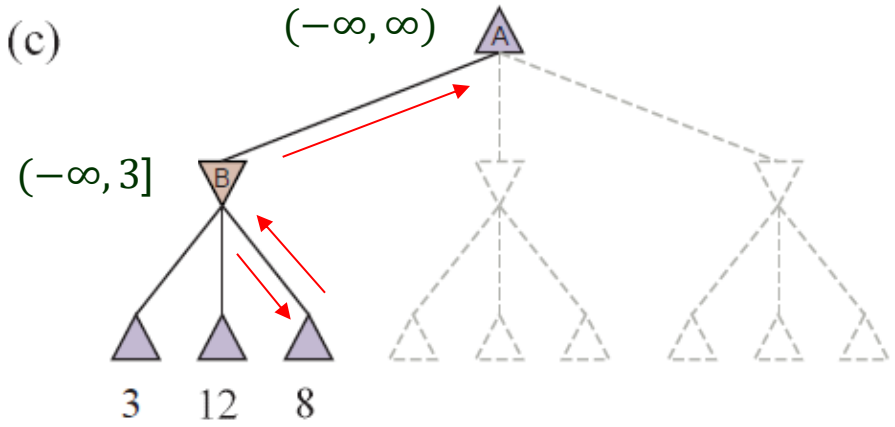
Cont'd



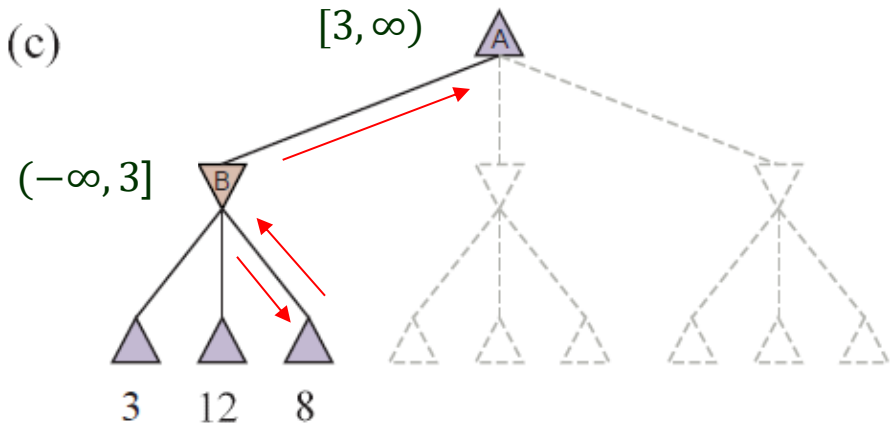
Cont'd



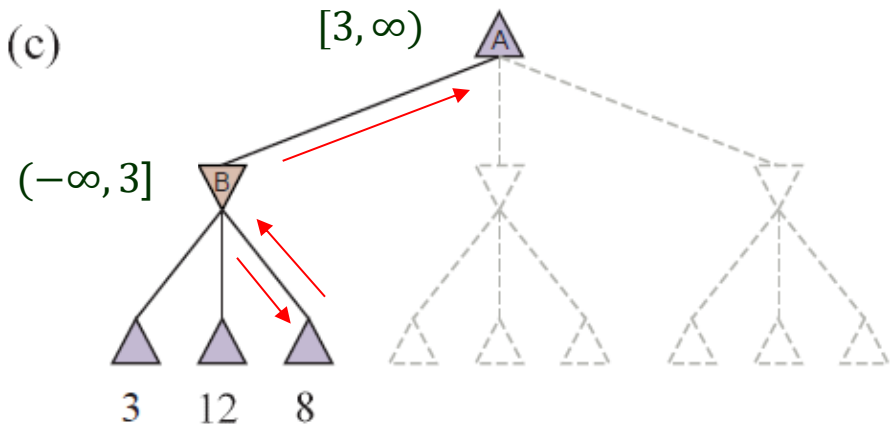
Cont'd

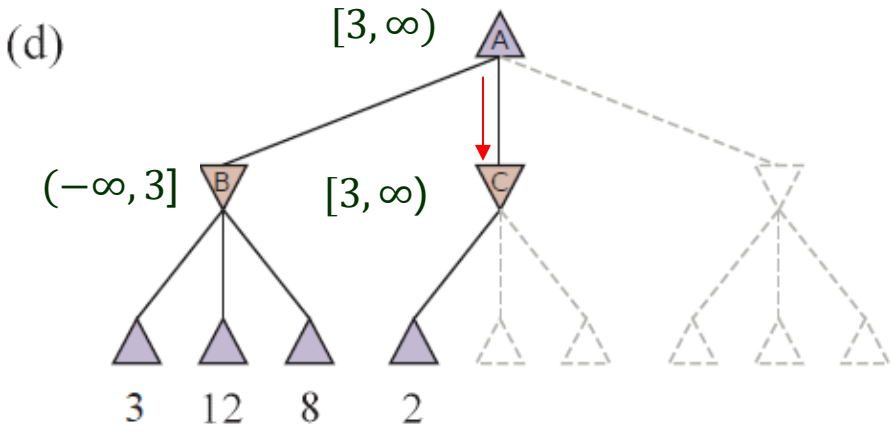


Cont'd

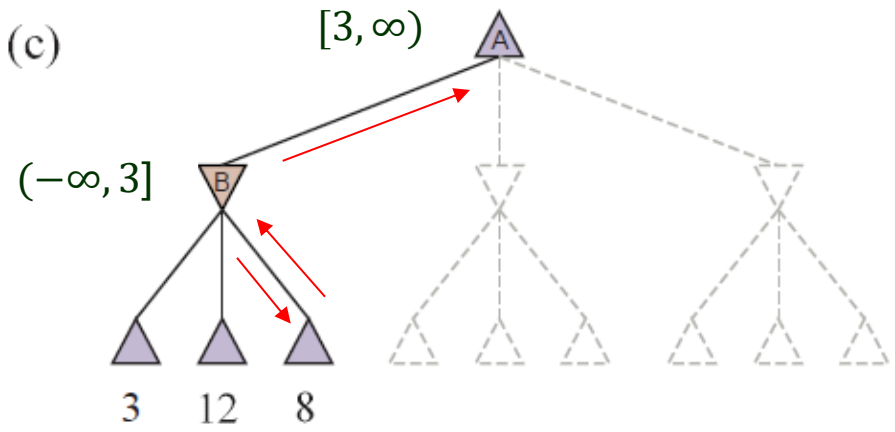


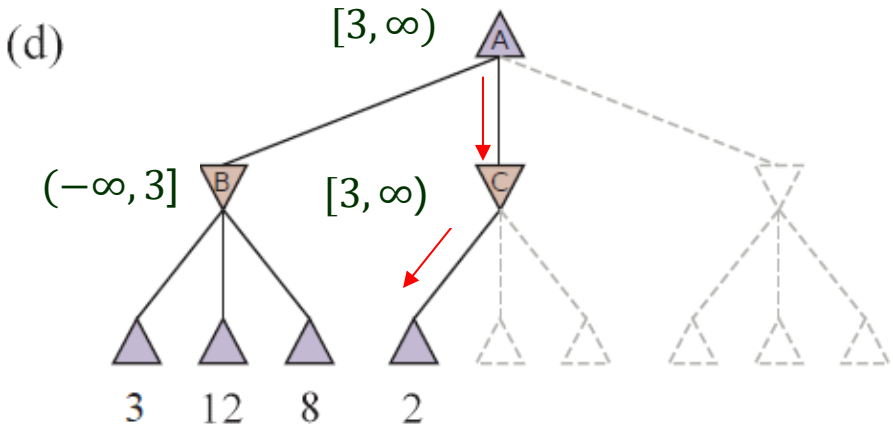
Cont'd



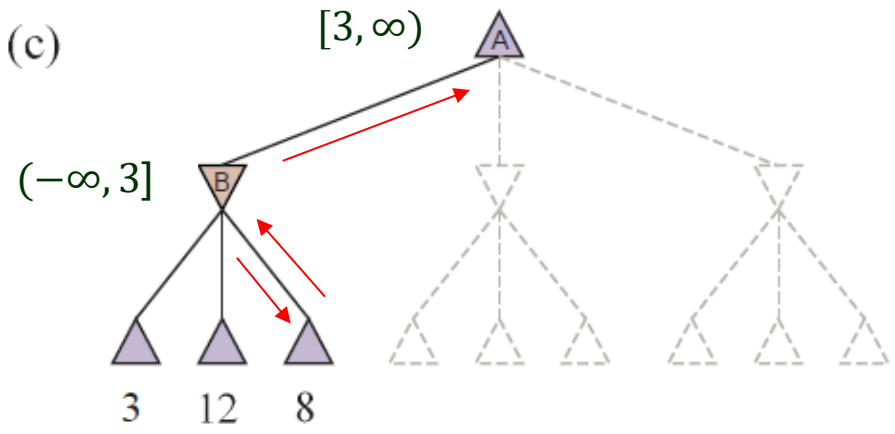


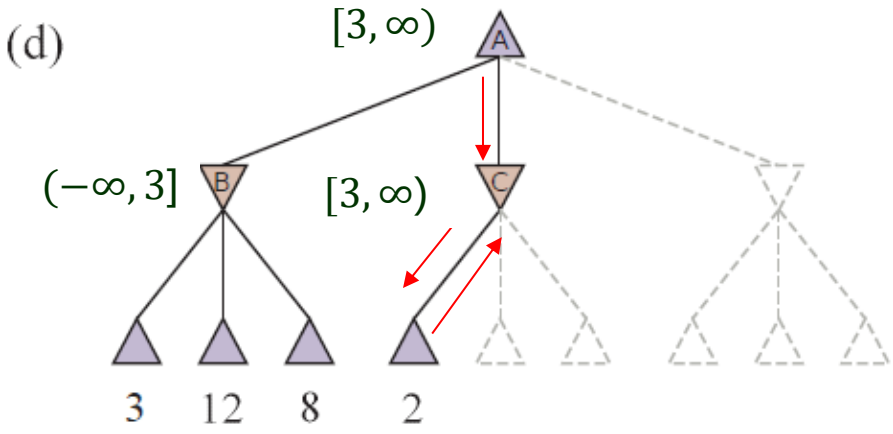
Cont'd



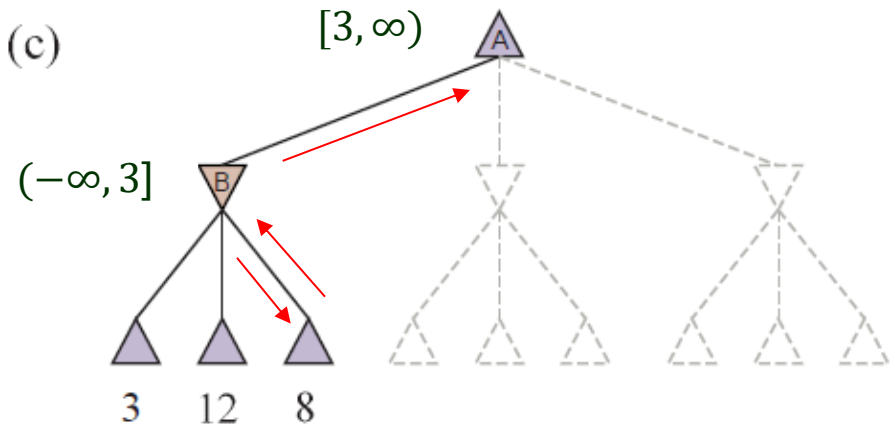


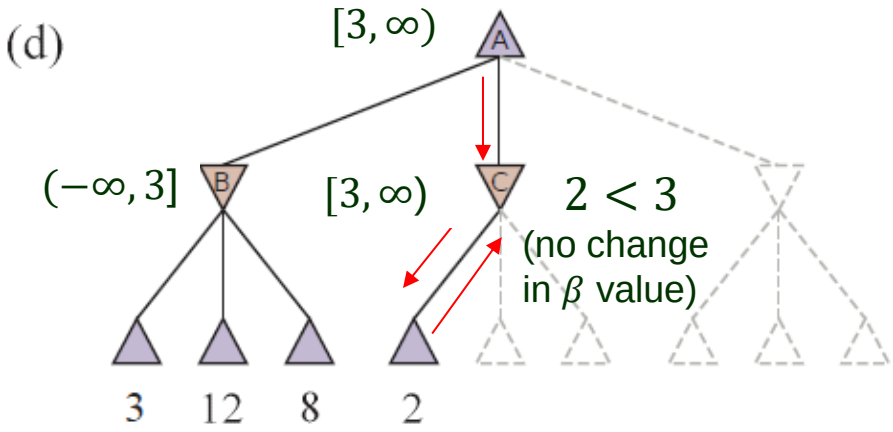
Cont'd



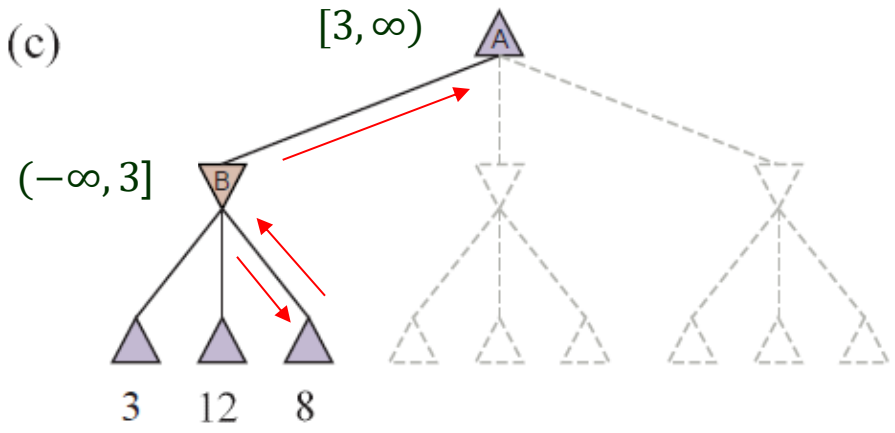


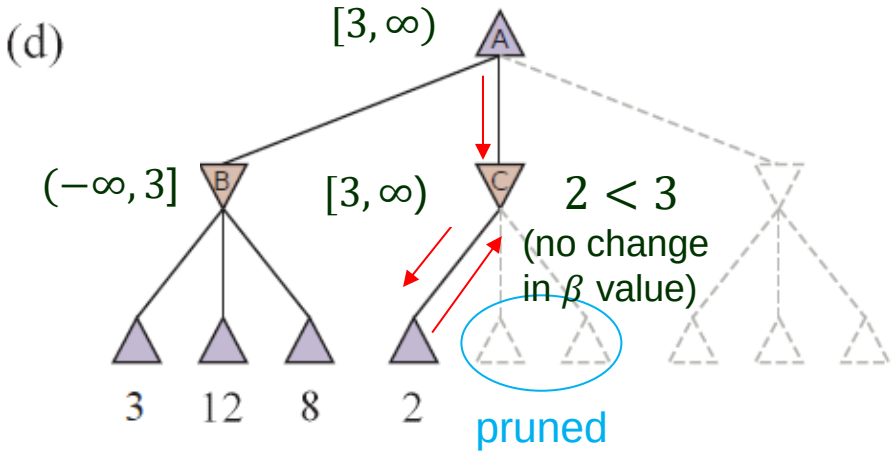
Cont'd


(c)
$[3, \infty)$
A
$(-\infty, 3]$
B
3
12
8

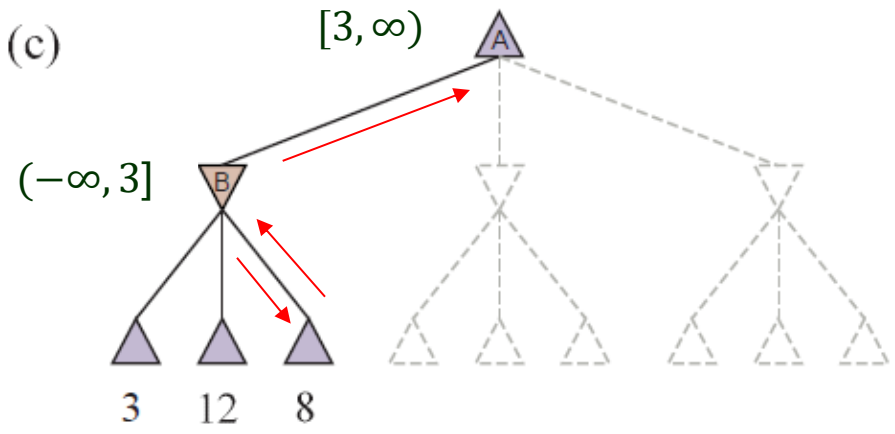

(d)
$[3, \infty)$
A
$(-\infty, 3]$
B
$[3, \infty)$
C
$2 < 3$
(no change in β value)
3
12
8
2

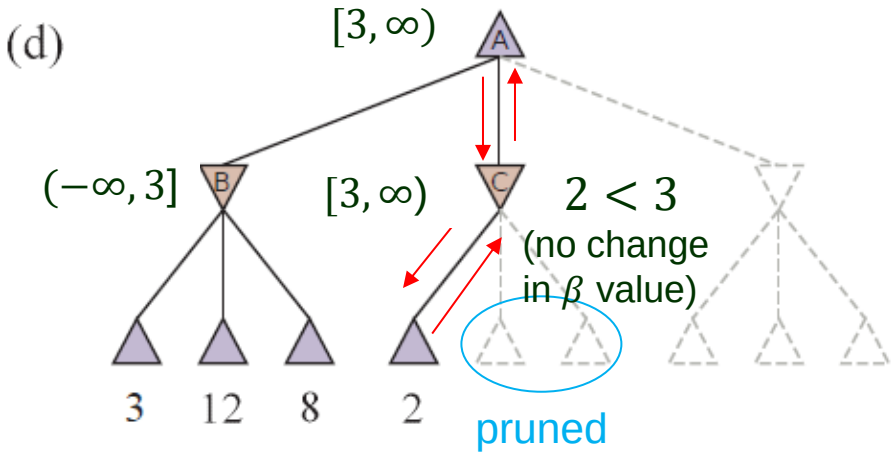
Cont'd


(c)
$[3, \infty)$
A
$(-\infty, 3]$
B
3
12
8

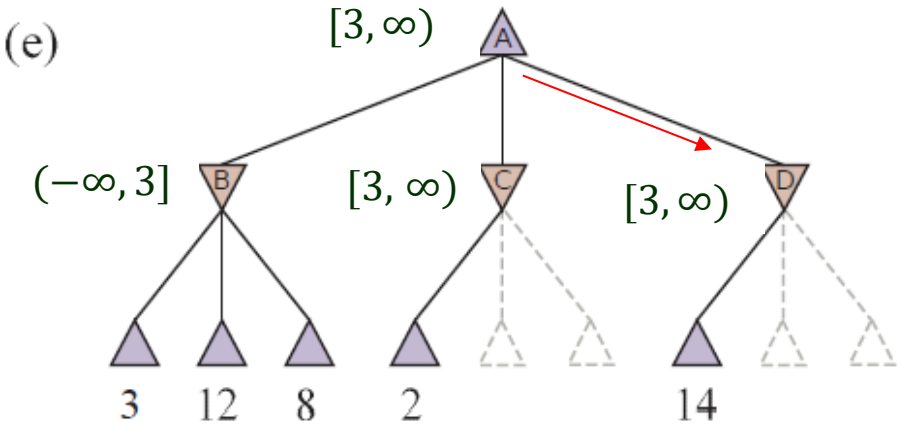

(d)
$[3, \infty)$
A
$(-\infty, 3]$
B
$[3, \infty)$
C
$2 < 3$
(no change in β value)
3
12
8
2
pruned

Cont'd

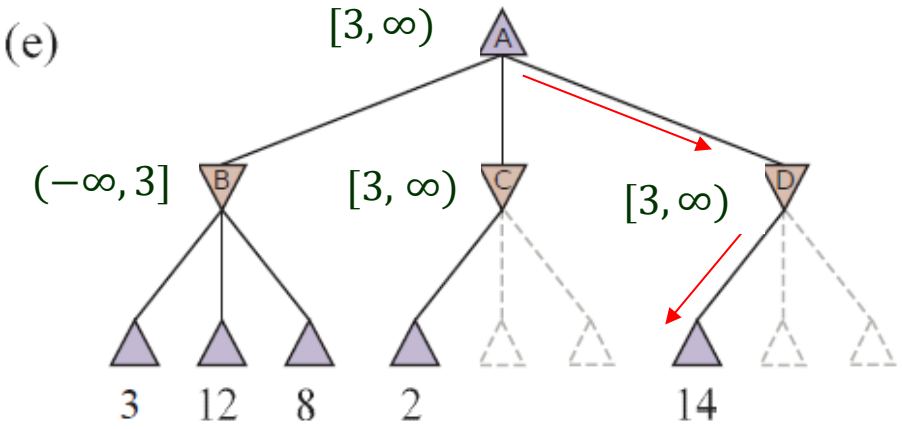

(c)
$[3, \infty)$
A
$(-\infty, 3]$
B
3
12
8


(d)
$[3, \infty)$
A
$(-\infty, 3]$
B
$[3, \infty)$
C
$2 < 3$
(no change in β value)
3
12
8
2
pruned

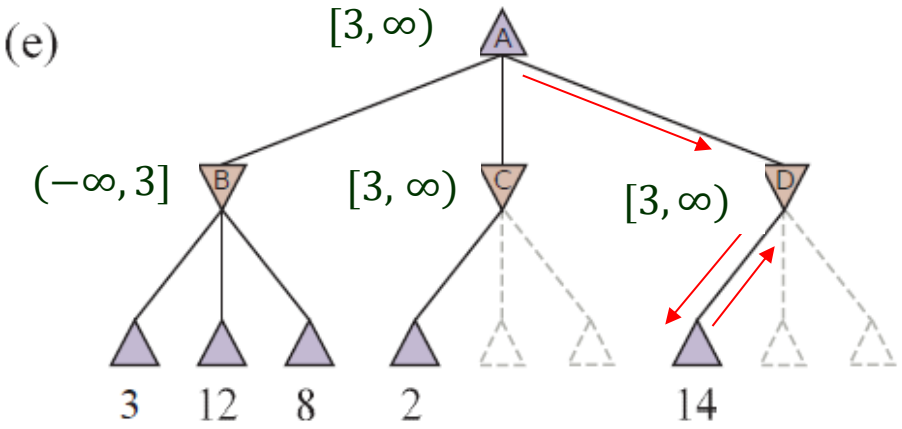
Cont'd



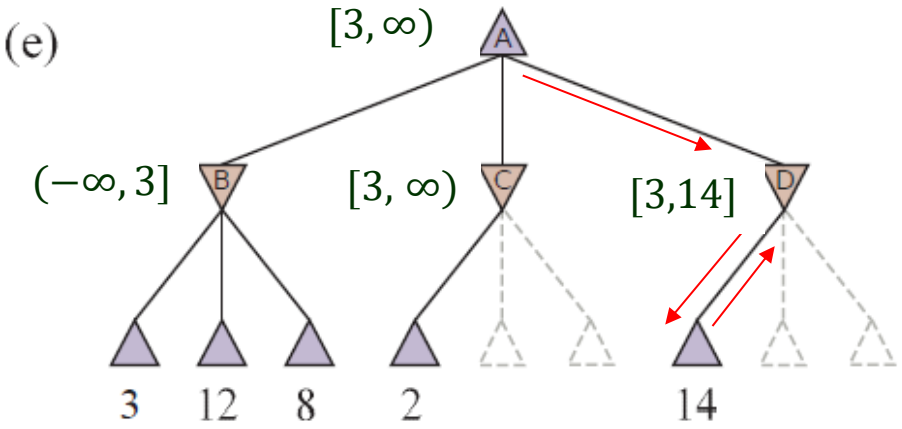
Cont'd



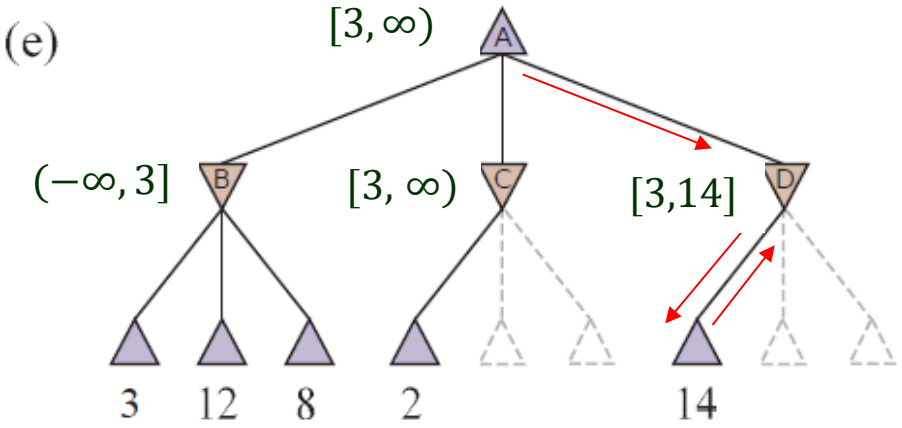
Cont'd

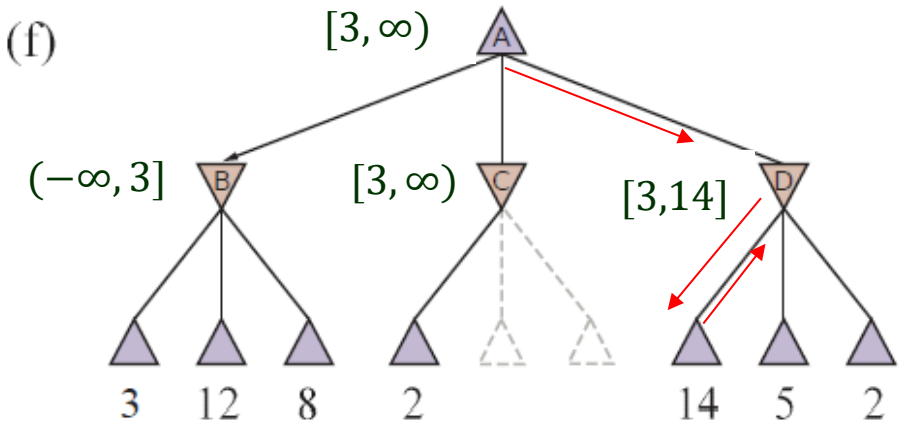


Cont'd

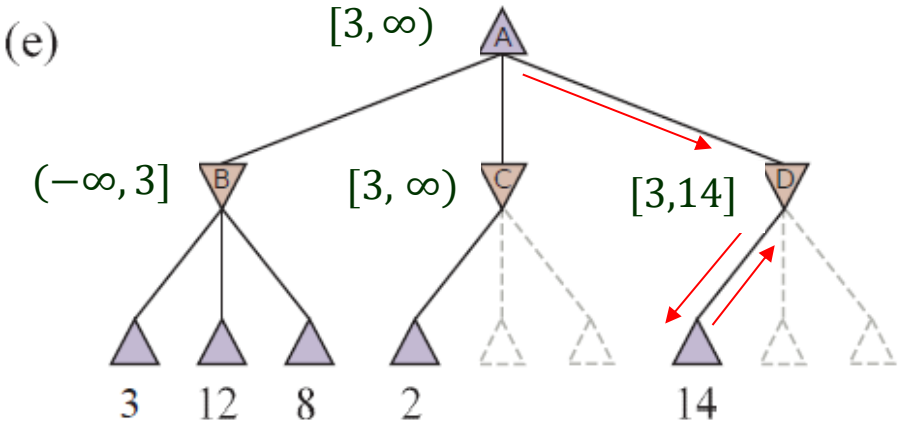


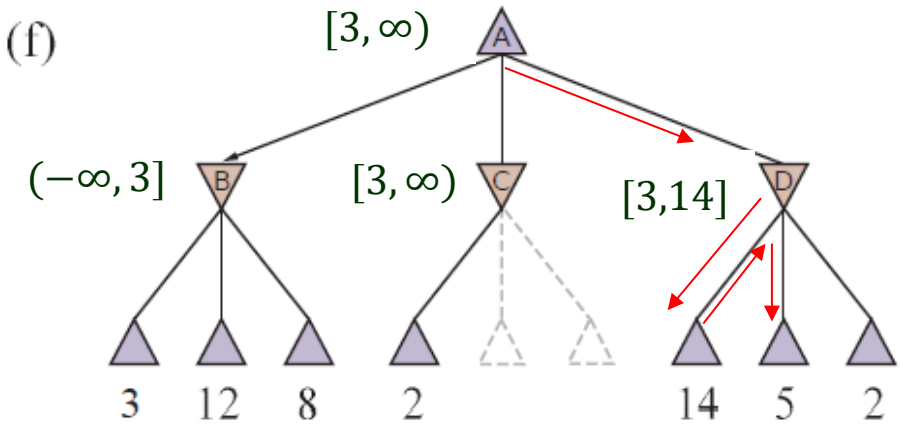
Cont'd



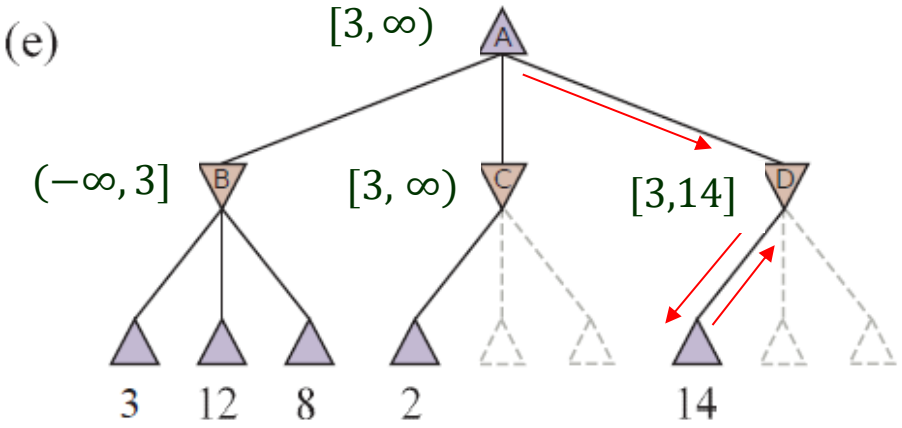


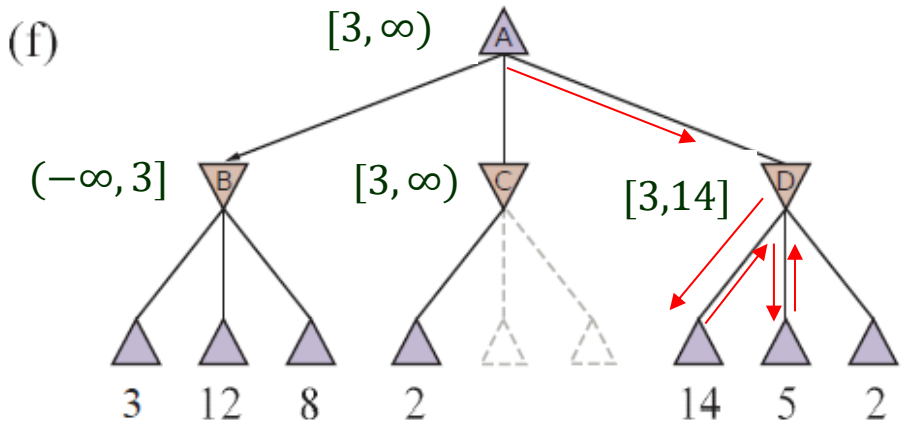
Cont'd


(e)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,14]
D
3
12
8
2
14

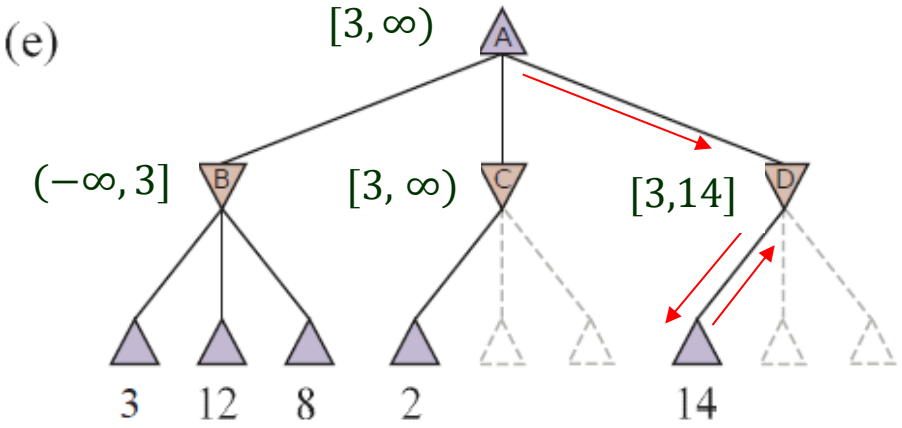

(f)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,14]
D
3
12
8
2
14
5
2

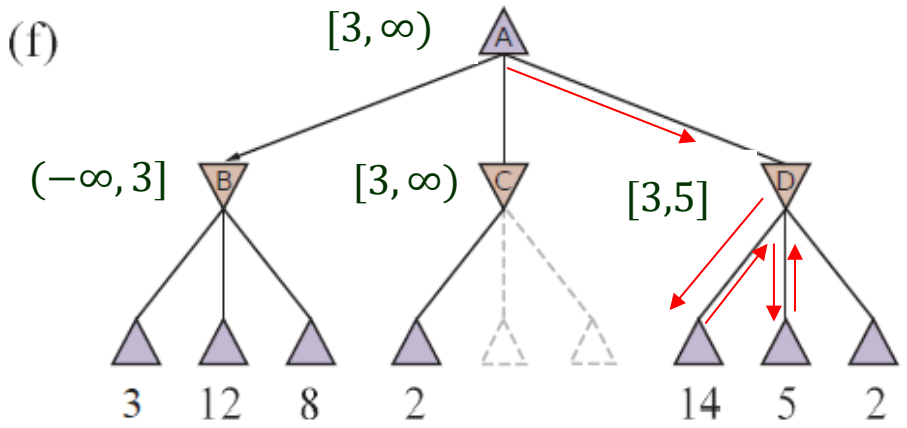
Cont'd


(e)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,14]
D
3
12
8
2
14

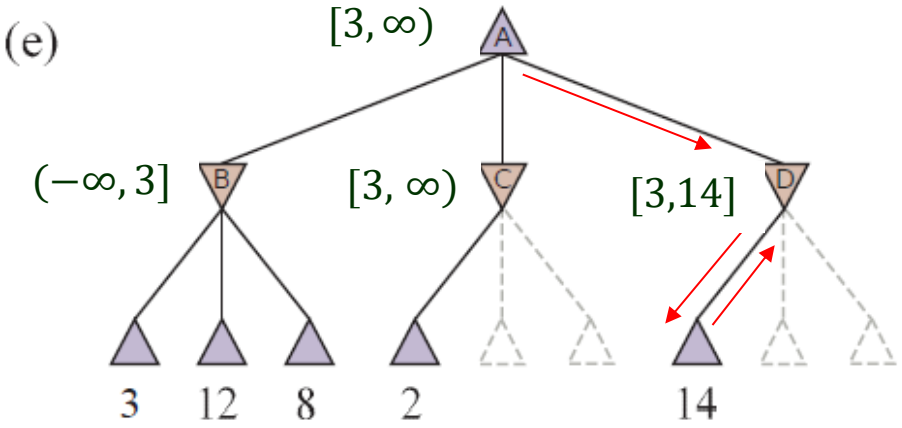

(f)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,14]
D
3
12
8
2
14
5
2

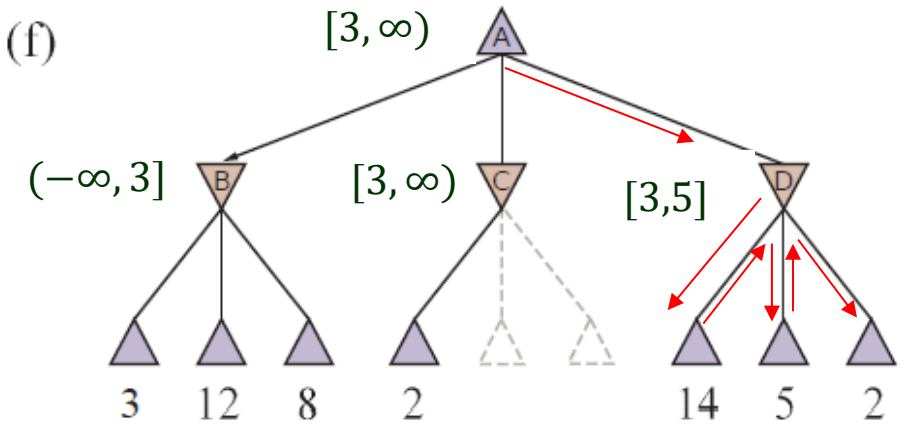
Cont'd


(e)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,14]
D
3
12
8
2
14

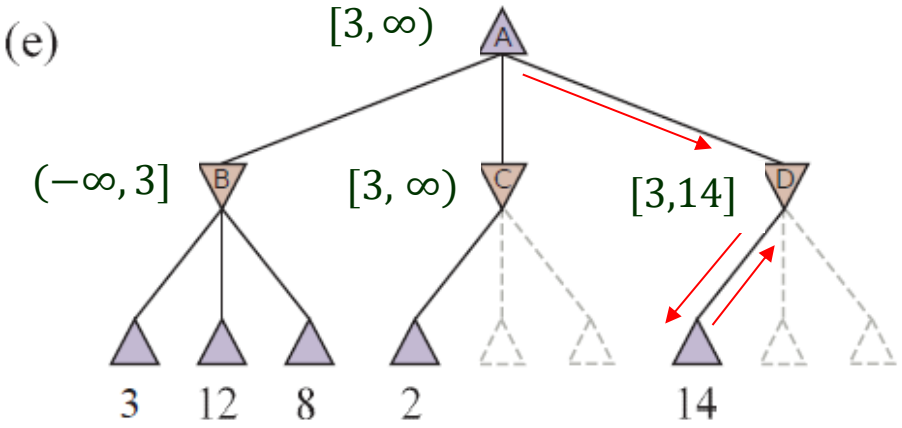

(f)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,5]
D
3
12
8
2
14
5
2

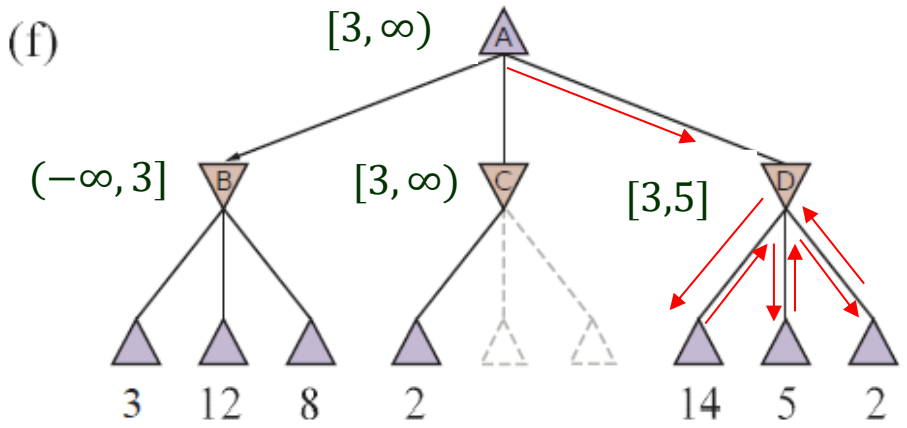
Cont'd


(e)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,14]
D
3
12
8
2
14

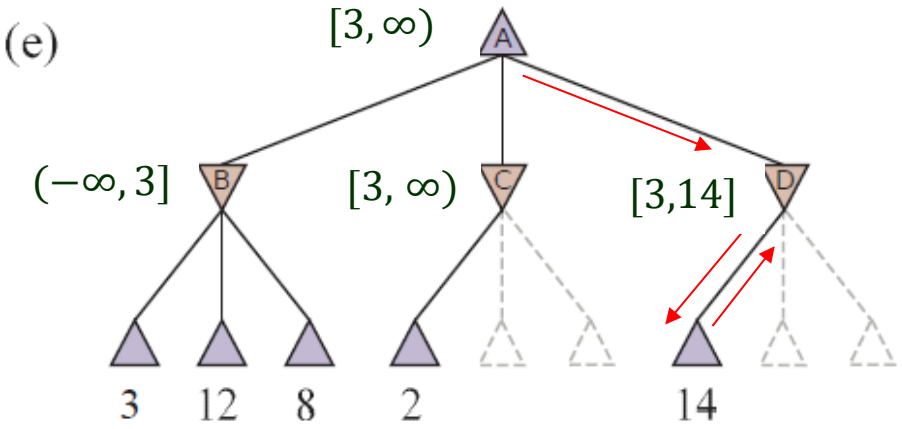

(f)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,5]
D
3
12
8
2
14
5
2

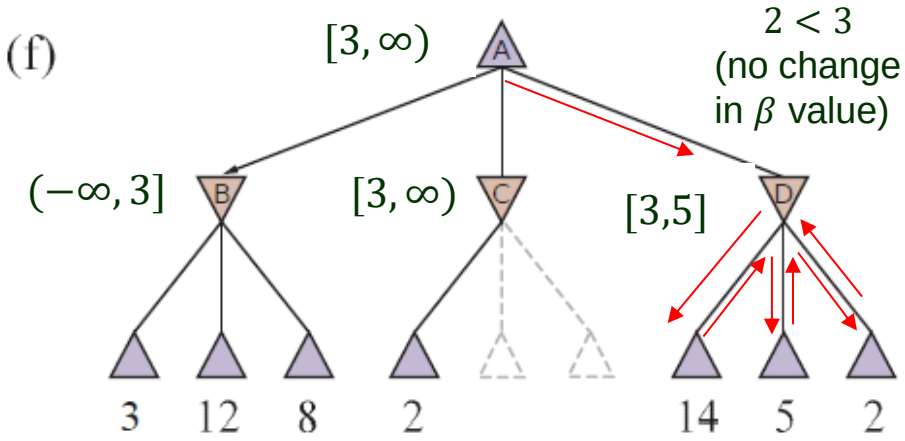
Cont'd


(e)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,14]
D
3
12
8
2
14

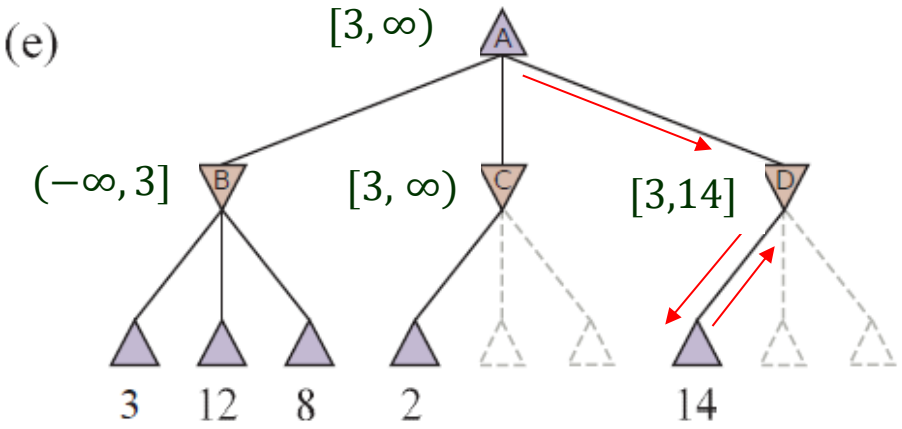

(f)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
[3,5]
D
3
12
8
2
14
5
2

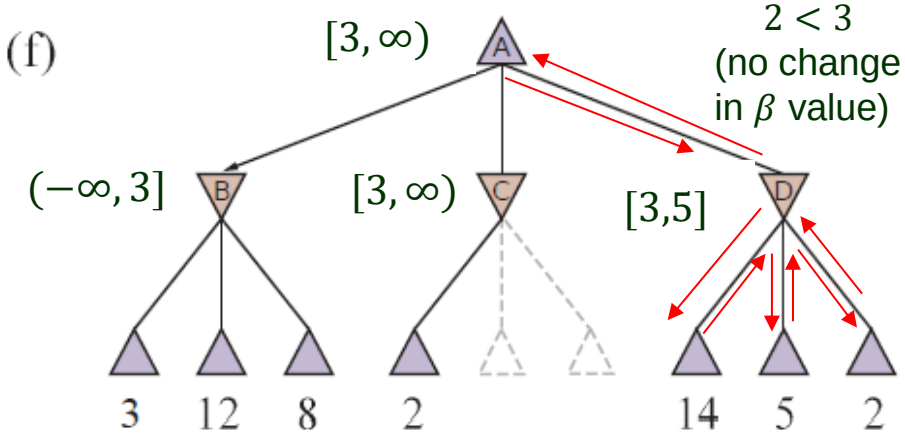
Cont'd


(e)
$[3,\infty)$
A
$(-\infty,3]$
B
$[3,\infty)$
C
$[3,14]$
D
3
12
8
2
14

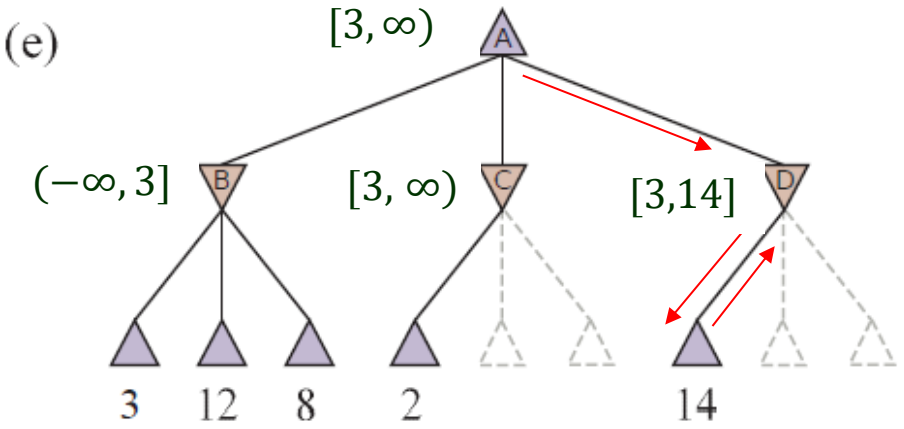

(f)
$[3,\infty)$
A
$2 < 3$
(no change
in β value)
$(-\infty,3]$
B
$[3,\infty)$
C
$[3,5]$
D
3
12
8
2
14
5
2

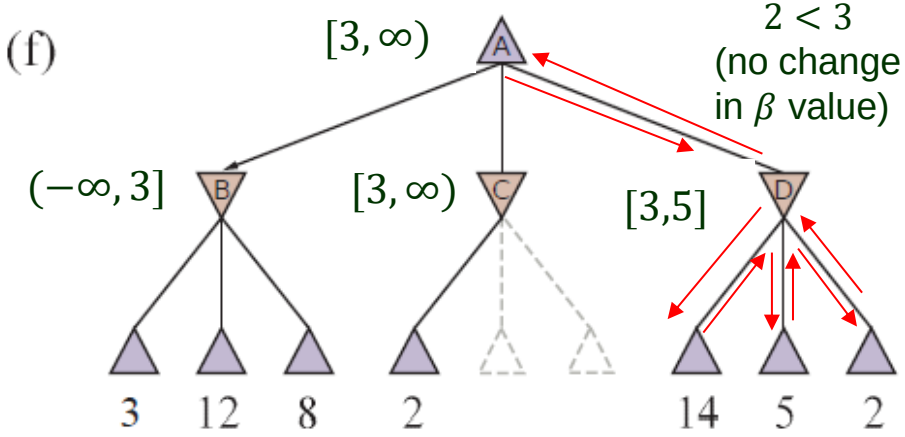
Cont'd


(e)
$[3, \infty)$
A
$(-\infty, 3]$
B
$[3, \infty)$
C
$[3,14]$
D
3
12
8
2
14


(f)
$[3, \infty)$
A
$2 < 3$
(no change
in β value)
$(-\infty, 3]$
B
$[3, \infty)$
C
$[3,5]$
D
3
12
8
2
14
5
2

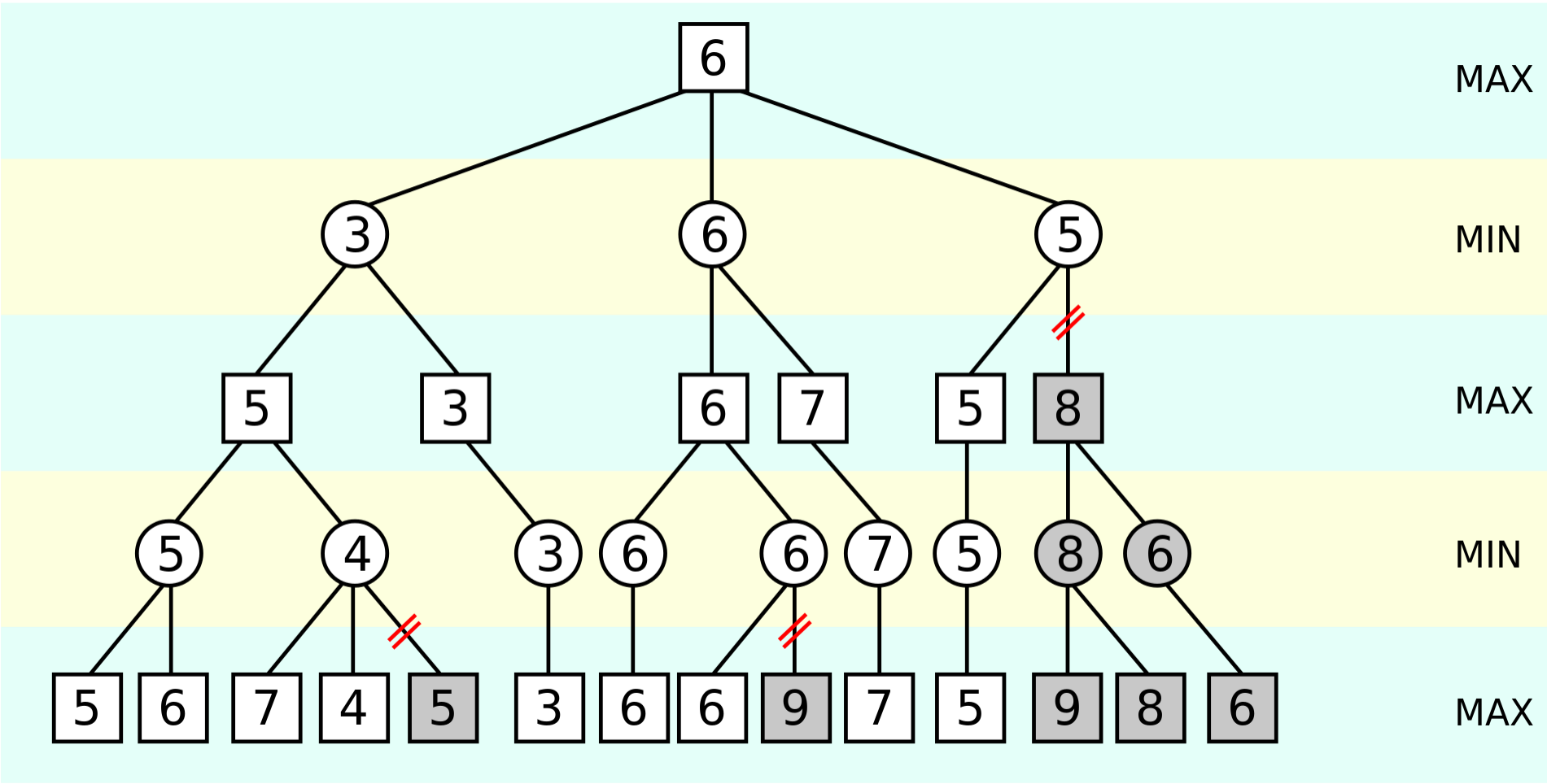
Cont'd



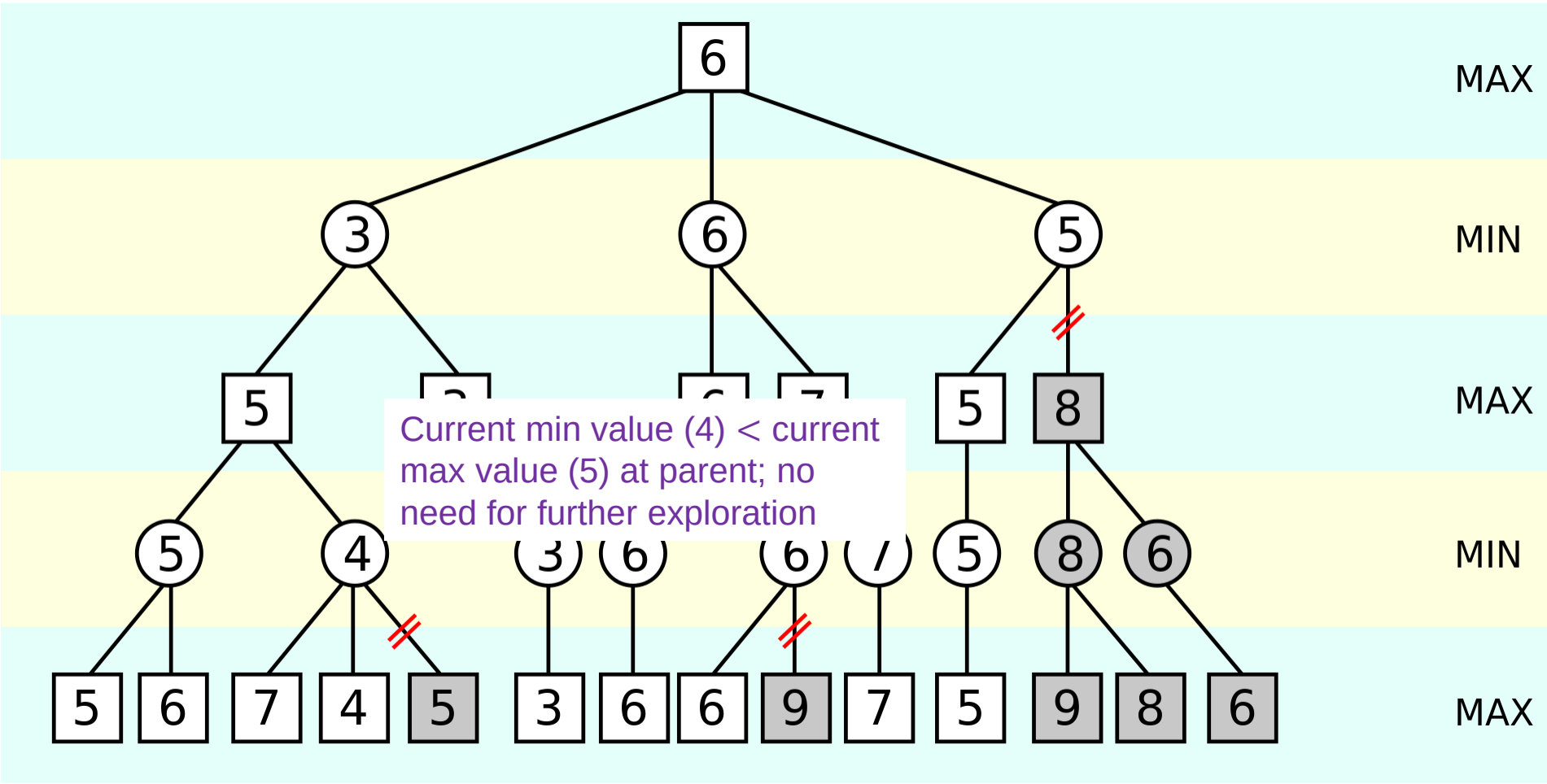


MINIMAX(root) $= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2))$
$= \max(3, \min(2,x,y), 2)$
$= \max(3,z,2)$ where $z = \min(2,x,y) \leq 2$
$= 3.$

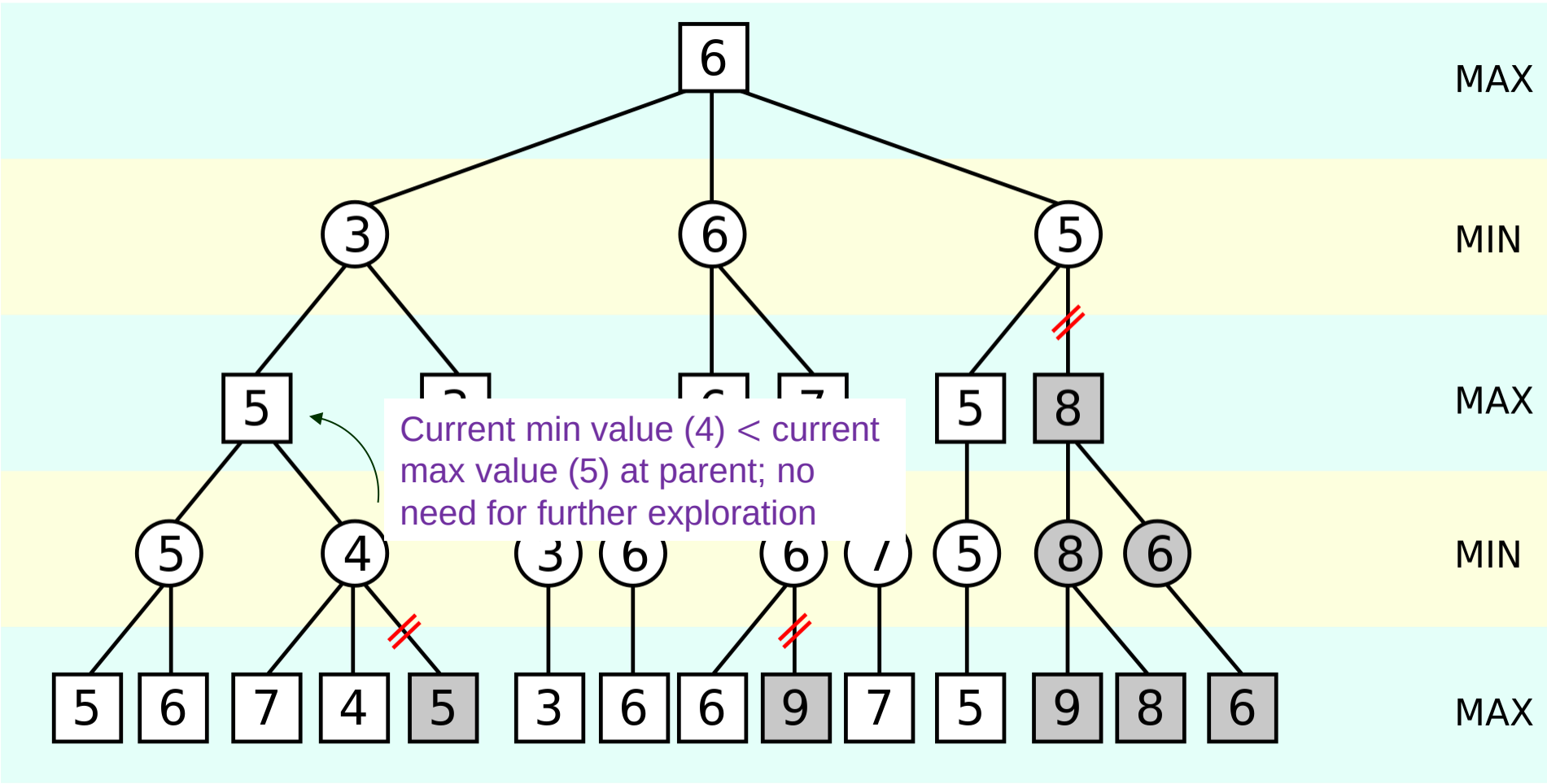
A Larger Example (Wikipedia)



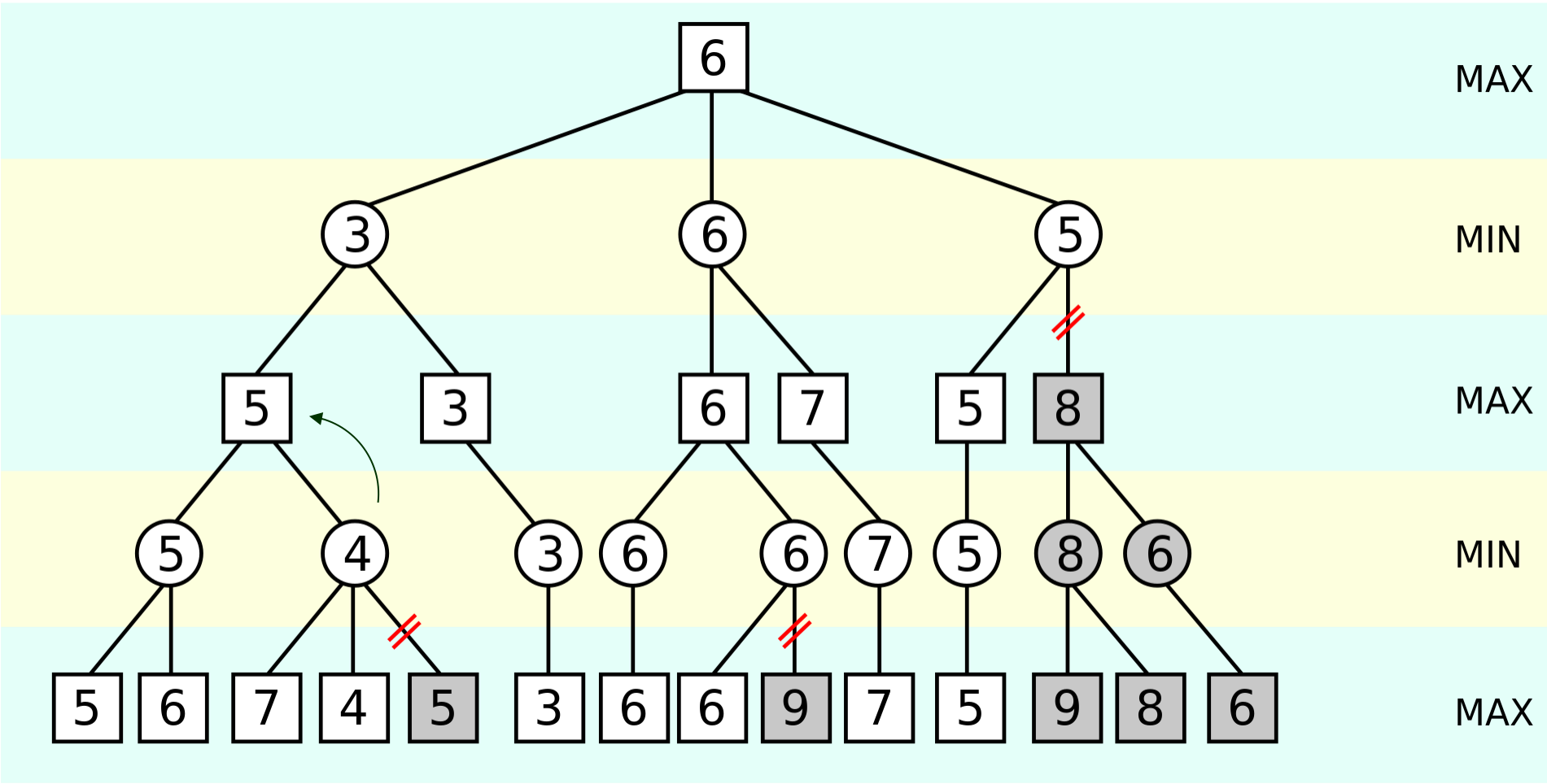
A Larger Example (Wikipedia)



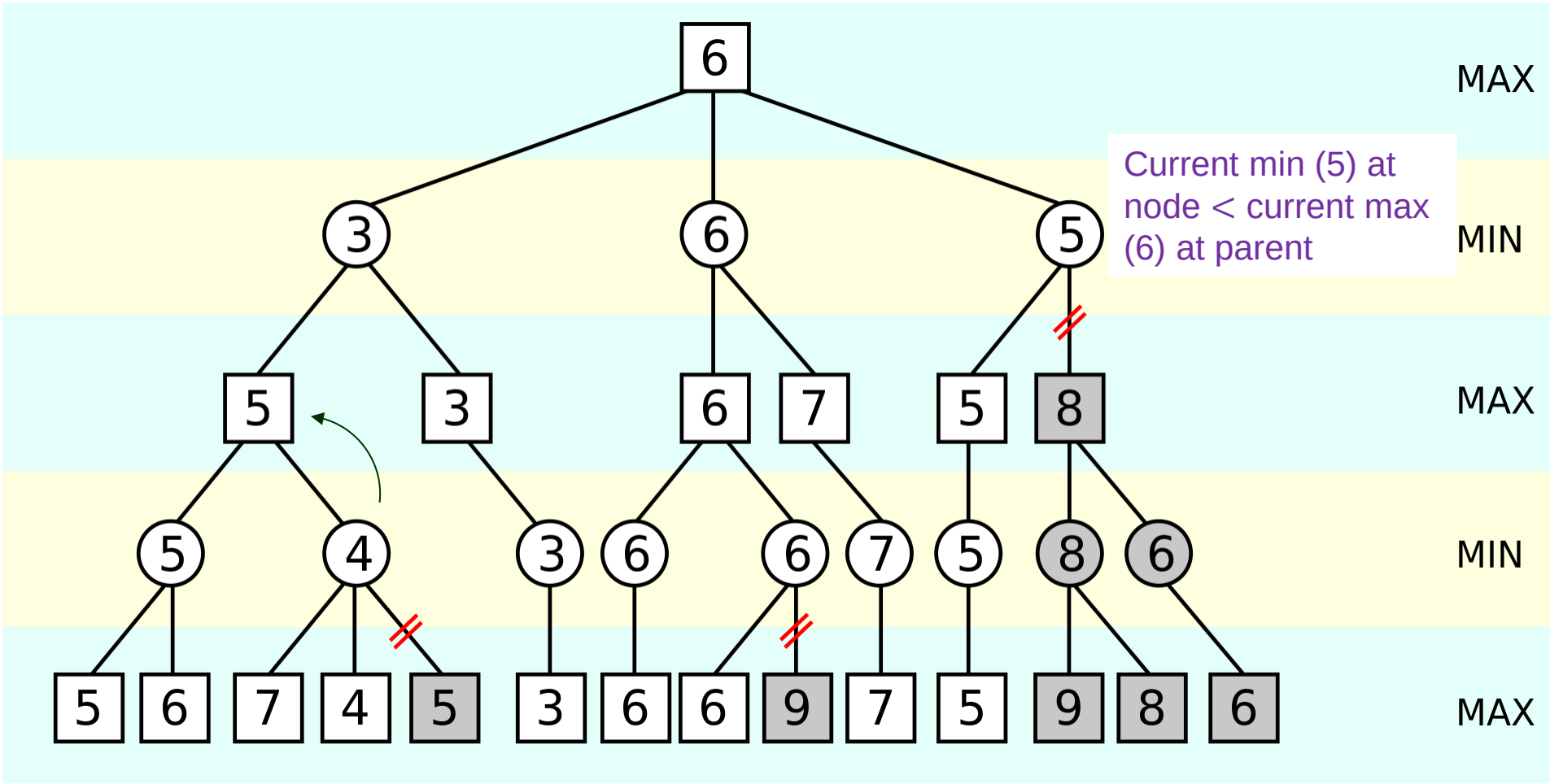
A Larger Example (Wikipedia)



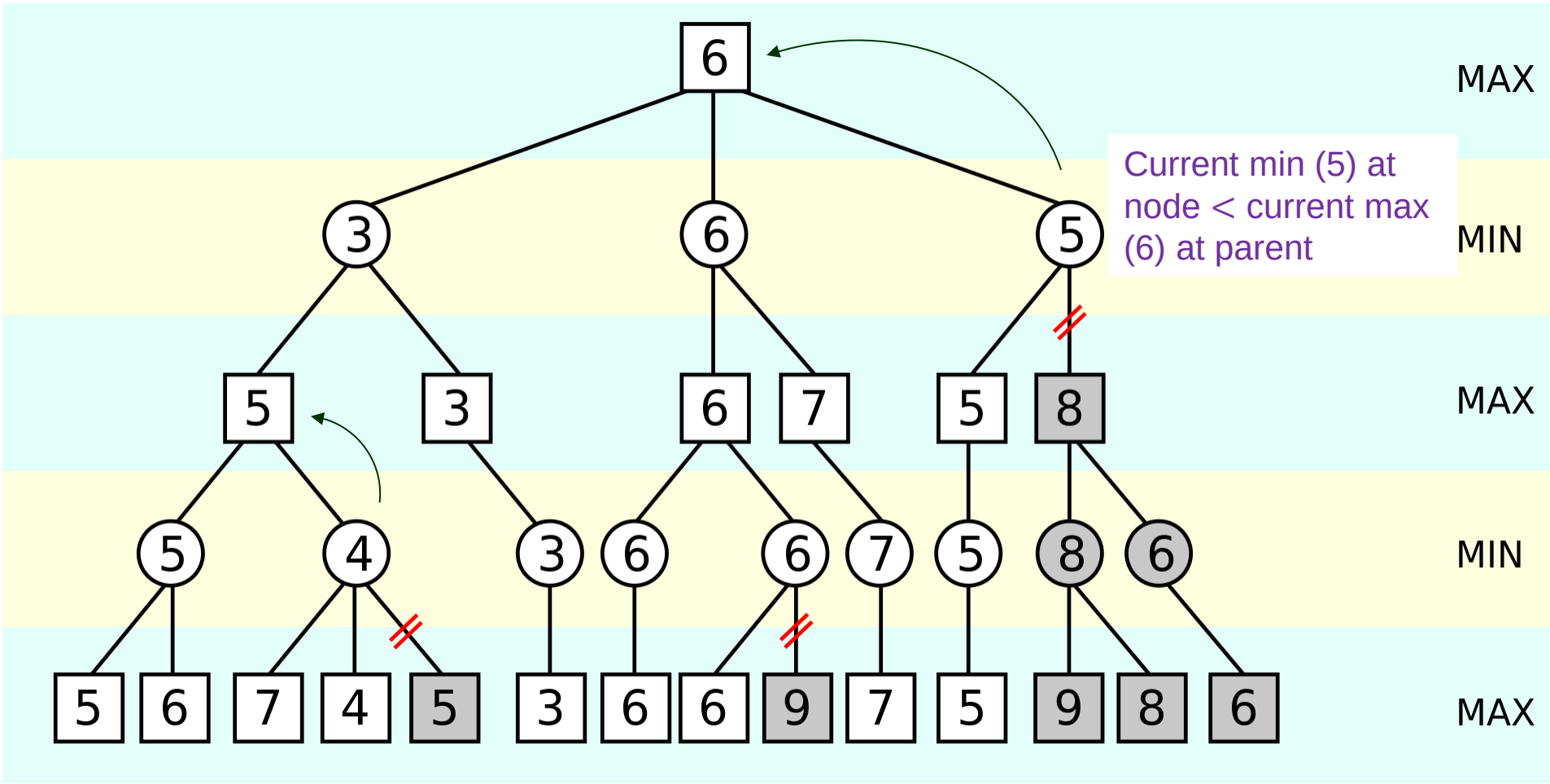
A Larger Example (Wikipedia)



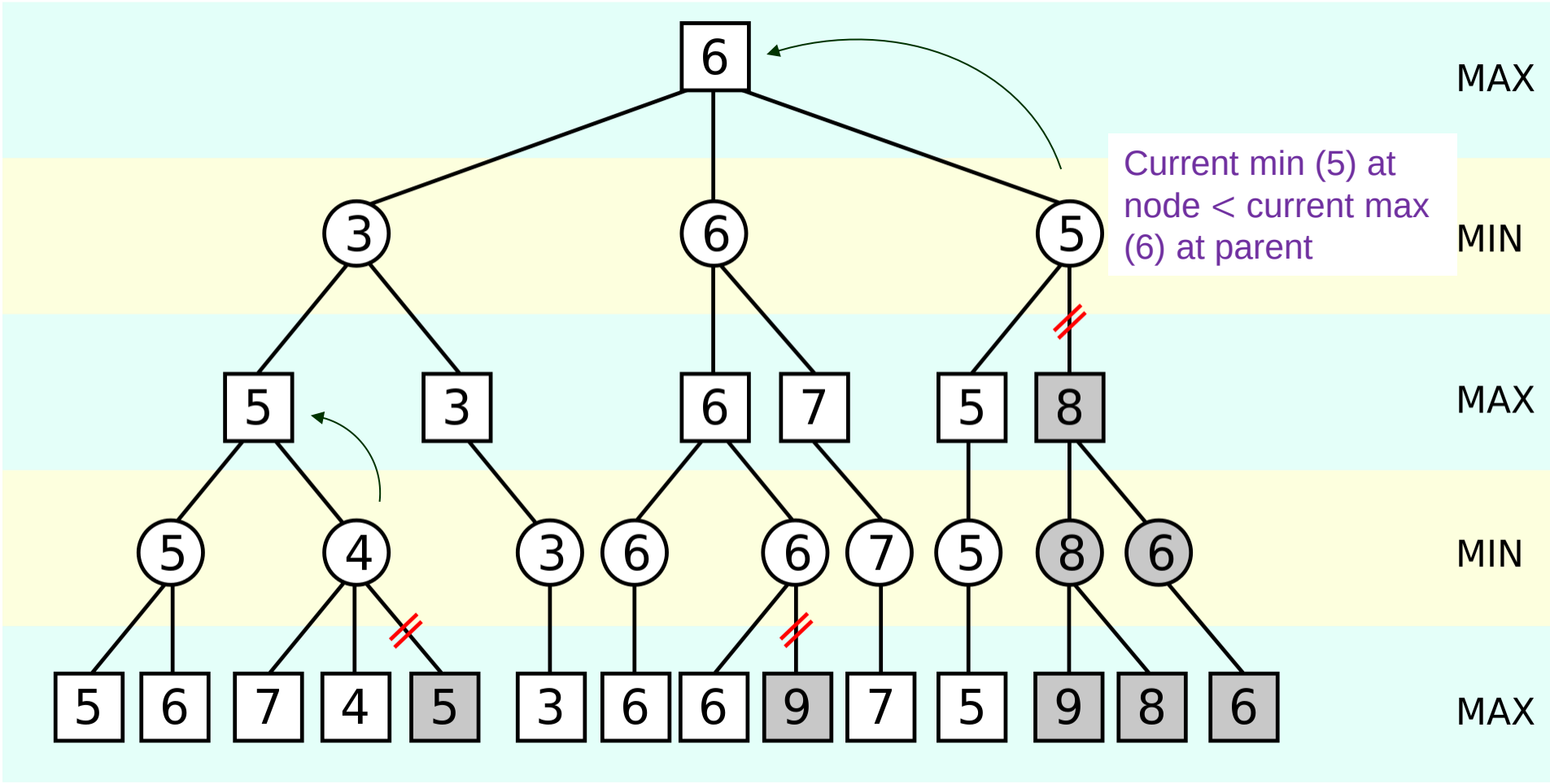
A Larger Example (Wikipedia)



A Larger Example (Wikipedia)



A Larger Example (Wikipedia)



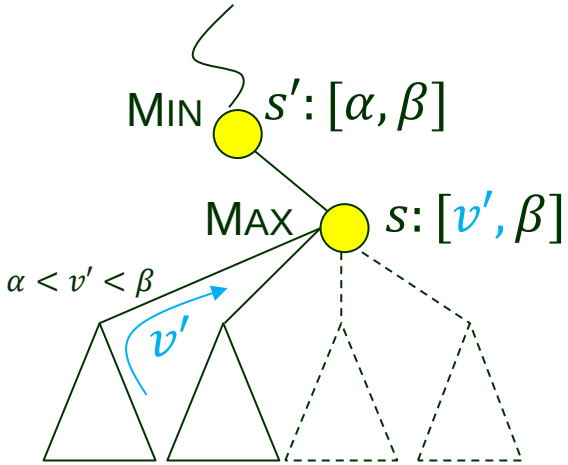
Exercise (suggested): Trace the execution of ALPHA-BETA-SEARCH (next page) and dynamically update the $[\alpha, \beta]$ value of every expanded node.

II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

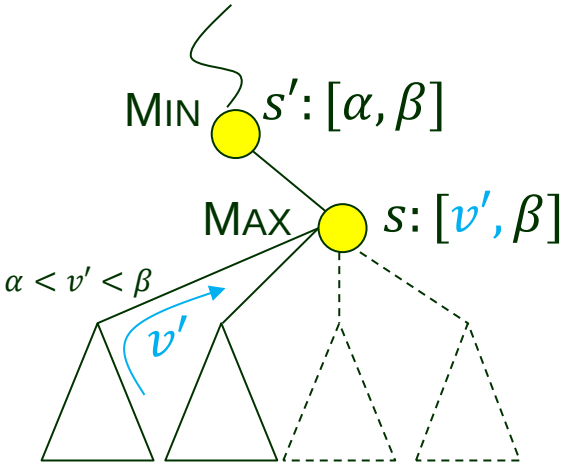
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$MAX(α, v)
  return v

function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$MIN(β, v)
  return v



II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

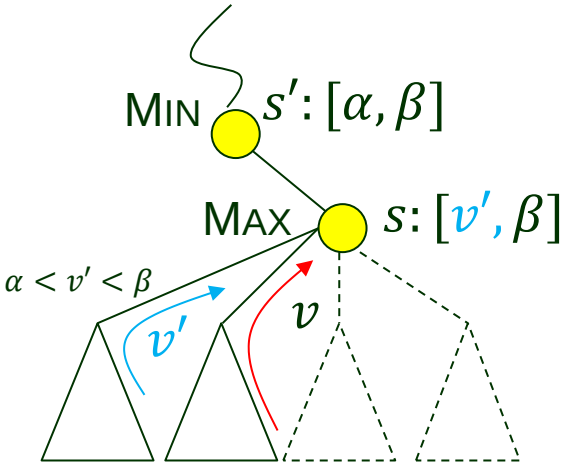


function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v

function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$MIN(β, v)
  return v

II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

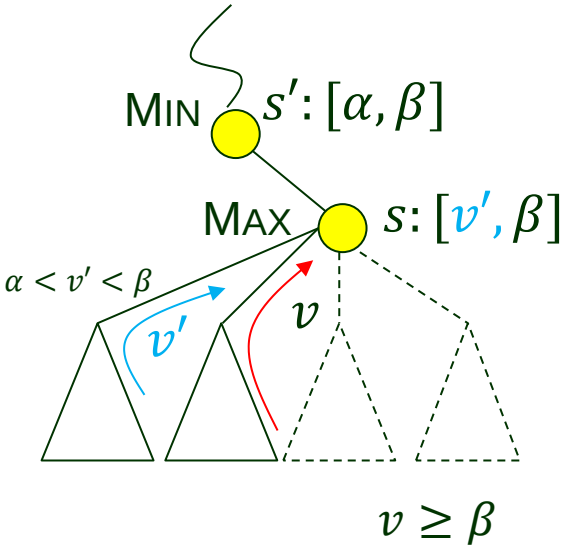


function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v

function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$MIN(β, v)
  return v

II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v



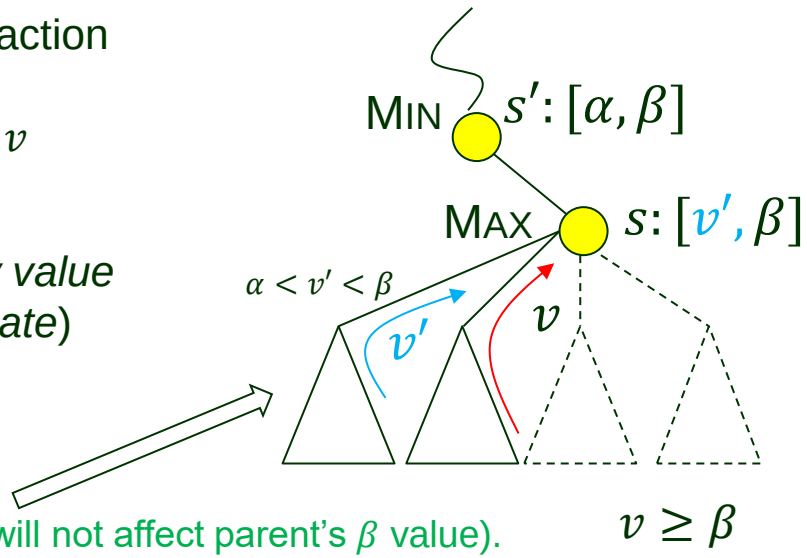
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v

function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$MIN(β, v)
  return v

II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning (returned v will not affect parent's β value).
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v

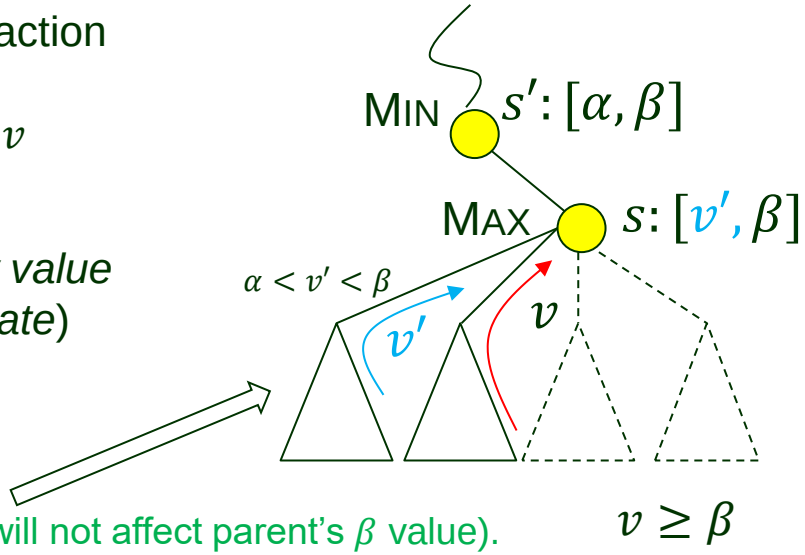


function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$MIN(β, v)
  return v

II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning (returned v will not affect parent's β value).
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v // maximum value of all children if none are pruned

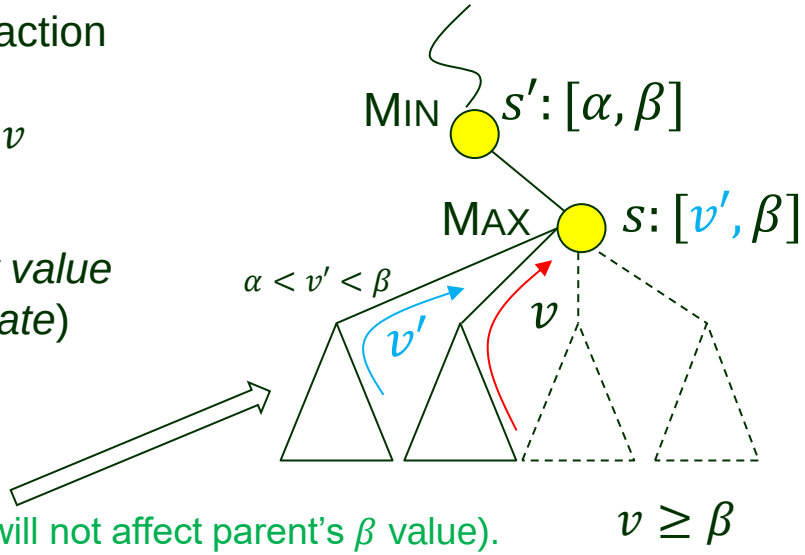


function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$MIN(β, v)
  return v

II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning (returned v will not affect parent's β value).
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v // maximum value of all children if none are pruned

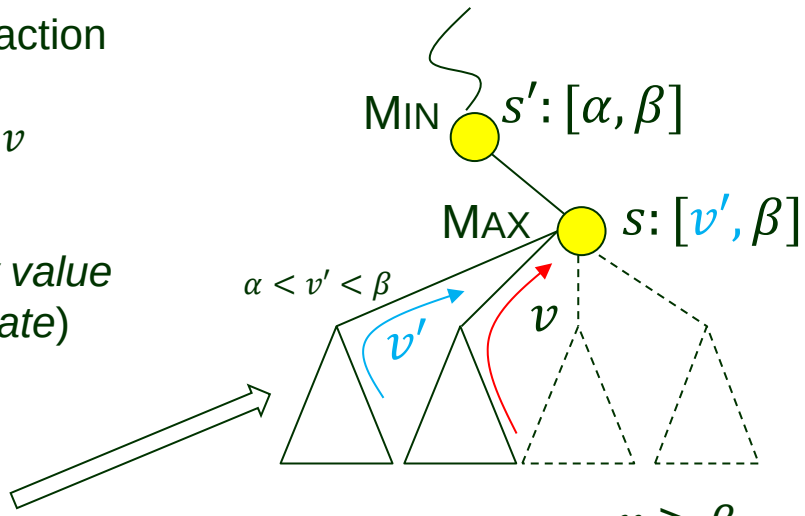


function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$MIN(β, v)
  return v

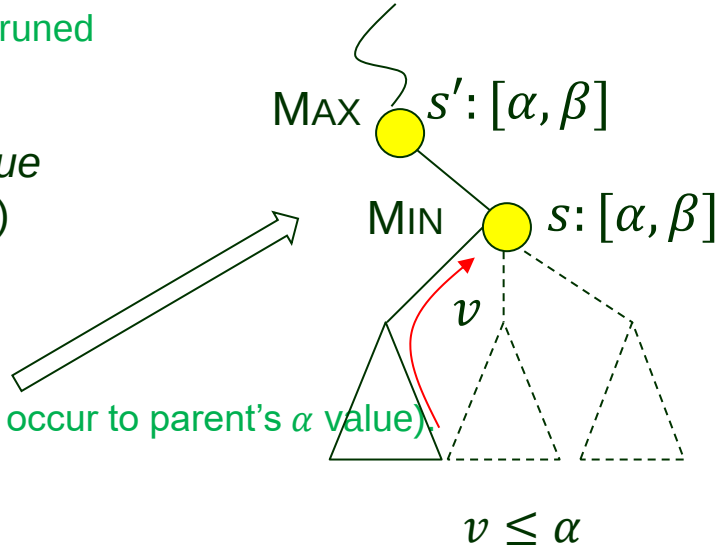
II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning (returned v will not affect parent's β value).
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v // maximum value of all children if none are pruned



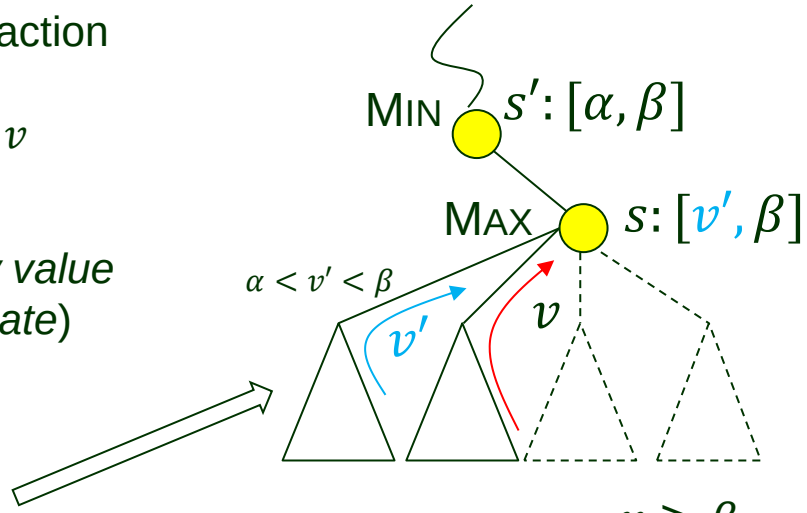
function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v // pruning (no change will occur to parent's α value)
    $\beta \leftarrow$MIN(β, v)
  return v



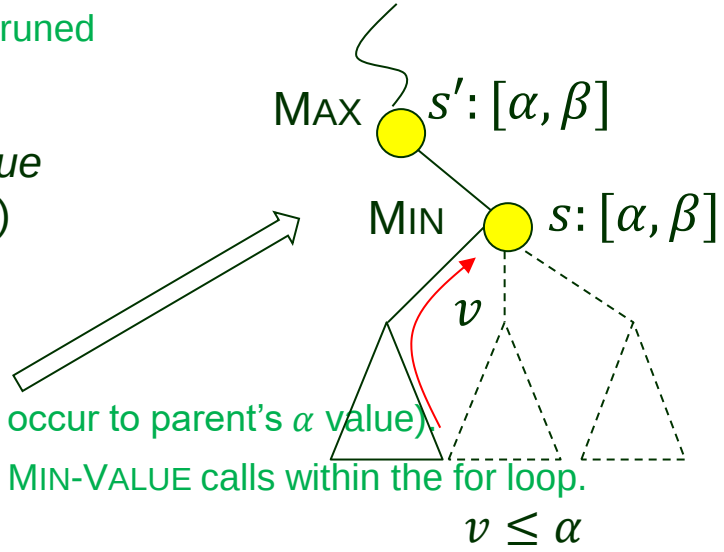
II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning (returned v will not affect parent's β value).
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v // maximum value of all children if none are pruned



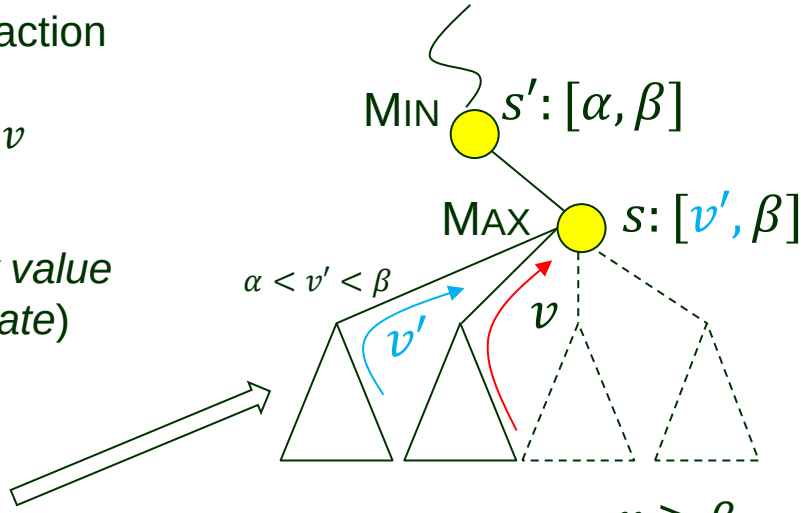
function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v // pruning (no change will occur to parent's α value)
    $\beta \leftarrow$MIN(β, v) // $v > \alpha$: new β passed on to the rest MIN-VALUE calls within the for loop.
  return v



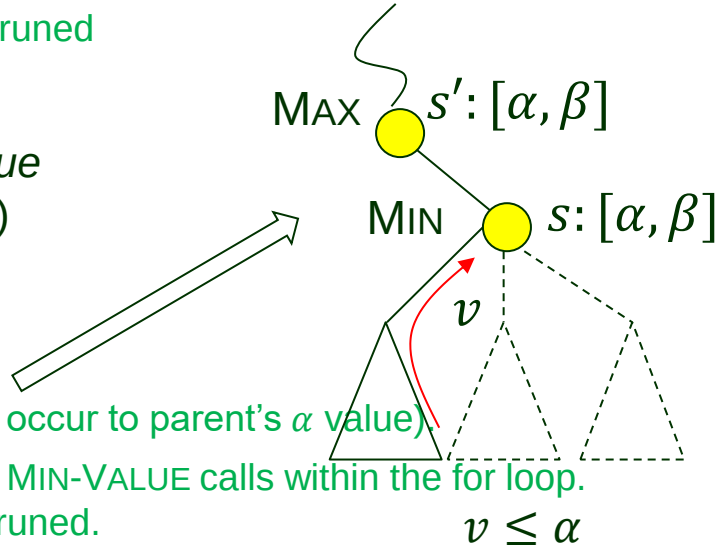
II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning (returned v will not affect parent's β value).
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v // maximum value of all children if none are pruned



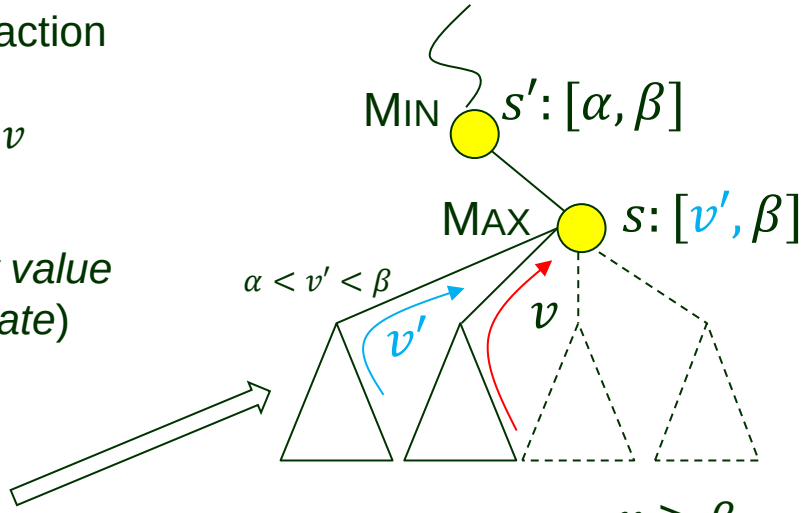
function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v // pruning (no change will occur to parent's α value)
    $\beta \leftarrow$MIN(β, v) // $v > \alpha$: new β passed on to the rest MIN-VALUE calls within the for loop.
  return v // minimum value of all children if none are pruned.



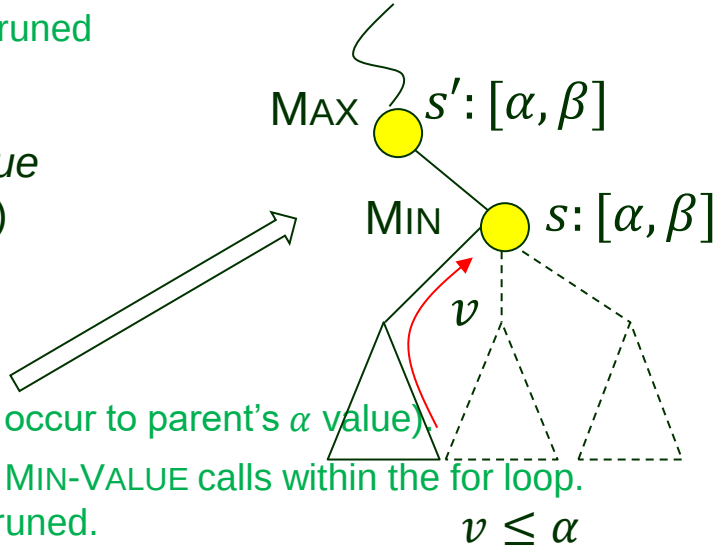
II. Alpha-Beta Search Algorithm (3rd Edition of Textbook)

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$MAX-VALUE(*state*, $-\infty, +\infty$)
  return the action in ACTIONS(*state*) with value v

// no change of β value within MAX-VALUE()
function MAX-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MAX(v, MIN-VALUE(RESULT(s, a),α, β))
    if $v \geq \beta$ **then return** v // pruning (returned v will not affect parent's β value).
    $\alpha \leftarrow$MAX(α, v) // $v < \beta$: new α passed on to the remaining MIN-VALUE calls within the for loop.
  return v // maximum value of all children if none are pruned



// no change of α value within MIN-VALUE()
function MIN-VALUE (*state*,α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a in ACTIONS(*state*) **do**
    $v \leftarrow$MIN(v, MAX-VALUE(RESULT(s, a),α, β))
    if $v \leq \alpha$ **then return** v // pruning (no change will occur to parent's α value)
    $\beta \leftarrow$MIN(β, v) // $v > \alpha$: new β passed on to the rest MIN-VALUE calls within the for loop.
  return v // minimum value of all children if none are pruned.



Short Summary on the Algorithm

Both MAX and MIN nodes

- Receive α and β values from the parent.
- Pass the best value of the executed actions back to the parent.

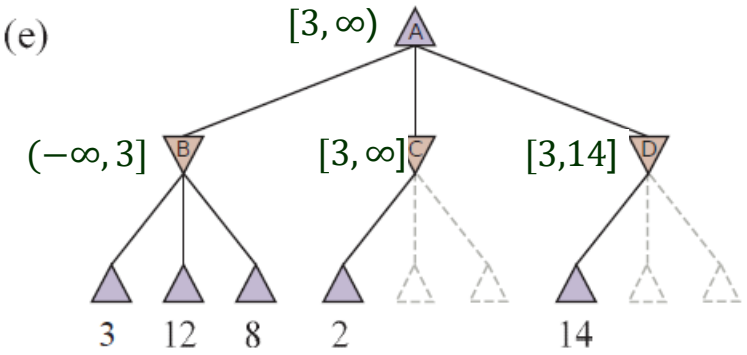
MAX node

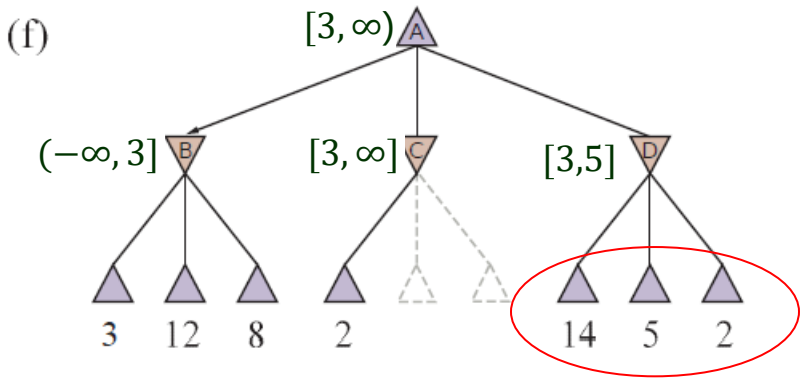
- Updates the α value if one of its actions yields a better value.
- Does not update the β value.
- Use the β value to skip actions that do not affect the outcome.

MIN node

- Updates the β value if one of its actions yields a better value.
- Does not update the α value.
- Use the α value to skip actions that do not affect the outcome.

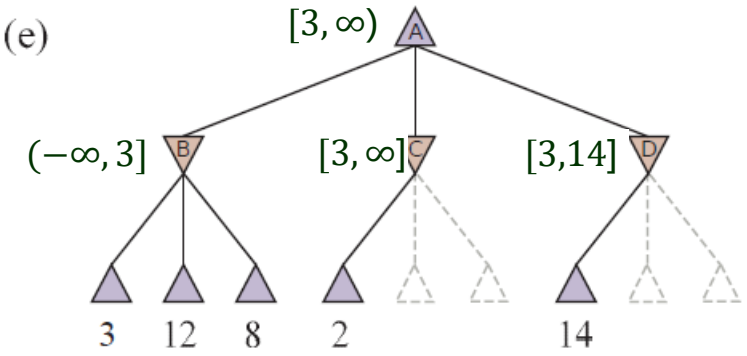
Move Ordering

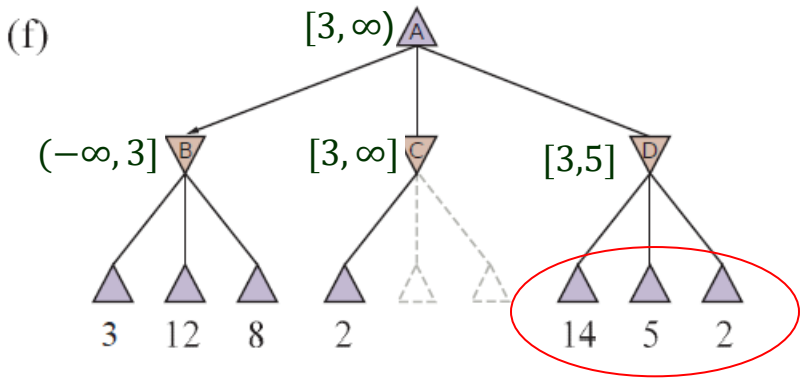




Successors 14 and 5 would've been pruned had 2 been generated first.

Move Ordering

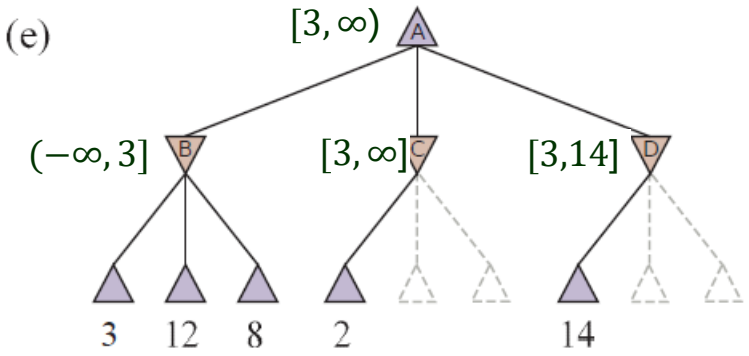


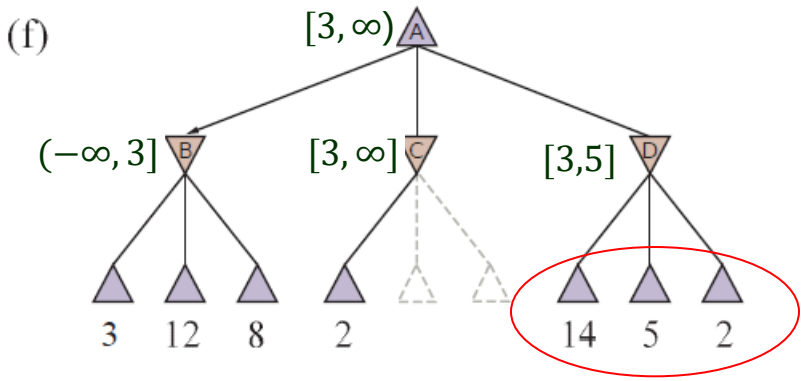


Successors 14 and 5 would've been pruned had 2 been generated first.

- Effectiveness of pruning is highly dependent on the order in which successors are generated.

Move Ordering



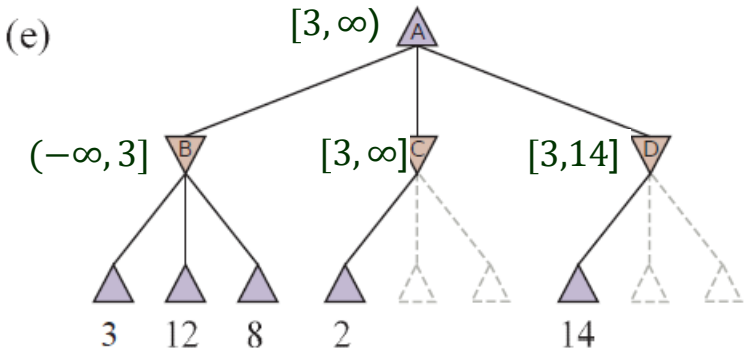


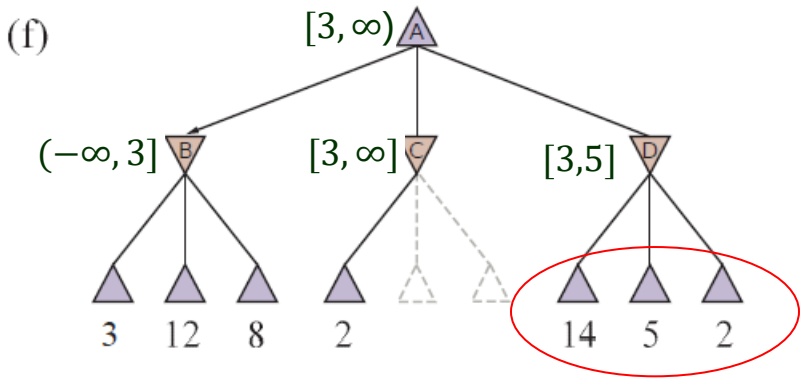
Successors 14 and 5 would've been pruned had 2 been generated first.

- Effectiveness of pruning is highly dependent on the order in which successors are generated.

- "Perfect ordering" has effective branching factor $\sqrt{b}$, which limits examination to only $O(b^{m/2})$ nodes compared to $O(b^m)$ for minimax.

Move Ordering





Successors 14 and 5 would've been pruned had 2 been generated first.

- Effectiveness of pruning is highly dependent on the order in which successors are generated.
- "Perfect ordering" has effective branching factor $\sqrt{b}$, which limits examination to only $O(b^{m/2})$ nodes compared to $O(b^m)$ for minimax.
- $O(b^{3m/4})$ nodes for random move ordering.

Two Strategies

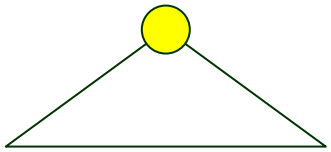
- ♦ For chess, a simple ordering function (sequentially considering captures, threats, forward moves, backward moves) could get close to the best case.
- ♠ Even with alpha-beta pruning and clever move ordering, minimax won't work well enough for games like chess and Go due to their vast state spaces.

Two Strategies

- ♦ For chess, a simple ordering function (sequentially considering captures, threats, forward moves, backward moves) could get close to the best case.
- ♠ Even with alpha-beta pruning and clever move ordering, minimax won't work well enough for games like chess and Go due to their vast state spaces.

Claude Shannon (1950) suggested two strategies:

- Type A (heuristic alpha-beta tree search) – chess
  - ♣ Considers all possible moves to a certain depth.
  - ♣ Use a heuristic function to estimate utilities of states at that depth.



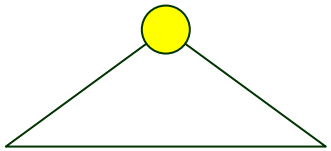
Explores wide & shallow portion of the search tree.

Two Strategies

- For chess, a simple ordering function (sequentially considering captures, threats, forward moves, backward moves) could get close to the best case.
- Even with alpha-beta pruning and clever move ordering, minimax won't work well enough for games like chess and Go due to their vast state spaces.

Claude Shannon (1950) suggested two strategies:

- Type A (heuristic alpha-beta tree search) – chess
  - Considers all possible moves to a certain depth.
  - Use a heuristic function to estimate utilities of states at that depth.
- Type B (Monte Carlo tree search) – Go
  - Ignore moves that look bad.
  - Follow promising lines "as far as possible".



Explores wide & shallow portion of the search tree.

Explores deep but narrow portion of the search tree.