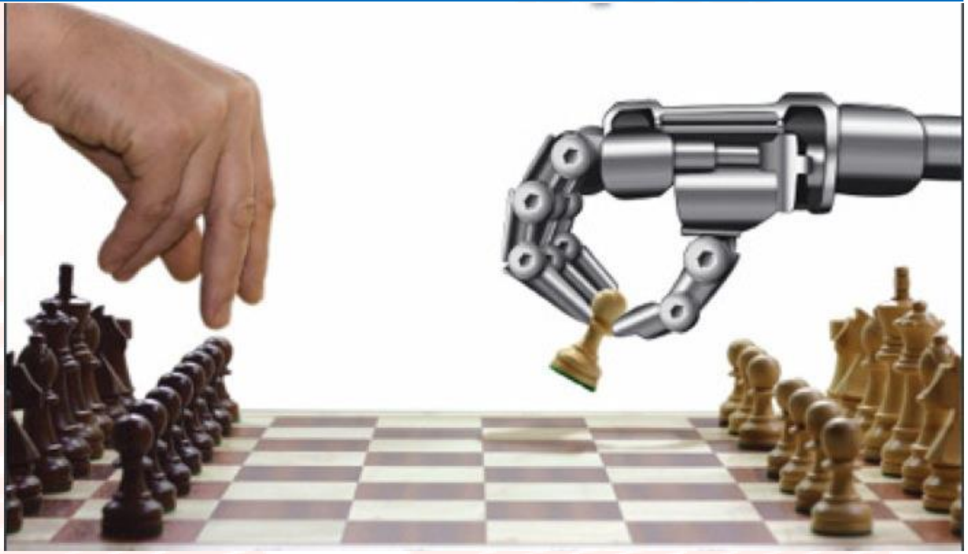
FACULTY OF INFORMATION TECHNOLOGY

Artificial Intelligence Fundamentals (NM TTNT)

Semester 1, 2021/2022

Chapter 5. Adversarial Search



Content

- What are games?
- Optimal decisions in games
  - Which strategy leads to success?
- $\alpha-\beta$ pruning
- Games of imperfect information
- Games that include an element of chance

What are and why study games?

- Games are a form of multi-agent environment
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
  - Competitive multi-agent environments give rise to adversarial problems a.k.a. games
- Game playing is a good problem for AI research

What are and why study games?

- Game playing is non-trivial
  - Players need "human-like" intelligence
  - Games can be very complex (e.g. chess, go)
  - Requires decision making within limited time
- Games usually are:
  - Well-defined and repeatable
  - Limited and accessible

Types of Games

	Deterministic	Chance
Perfect Information (fully observable)	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Inperfect Information (partially observable)	Stratego, Battleship	Brigde, Poker, Scrabble, Nuclear War

Relation of Games to Search

Search – no adversary

- Solution is (heuristic) method for finding goal
- Heuristics and CSP (Constraint Satisfaction Problems) techniques can find *optimal* solution
- Evaluation function: estimate of cost from start to goal through given node
- Examples: path planning, scheduling activities

Games– adversary

- Solution is strategy (specifies move for every possible opponent reply).
- Time limits force an approximate solution
- Evaluation function: evaluate "goodness" of game position
- Examples: chess, checkers, Othello, backgammon

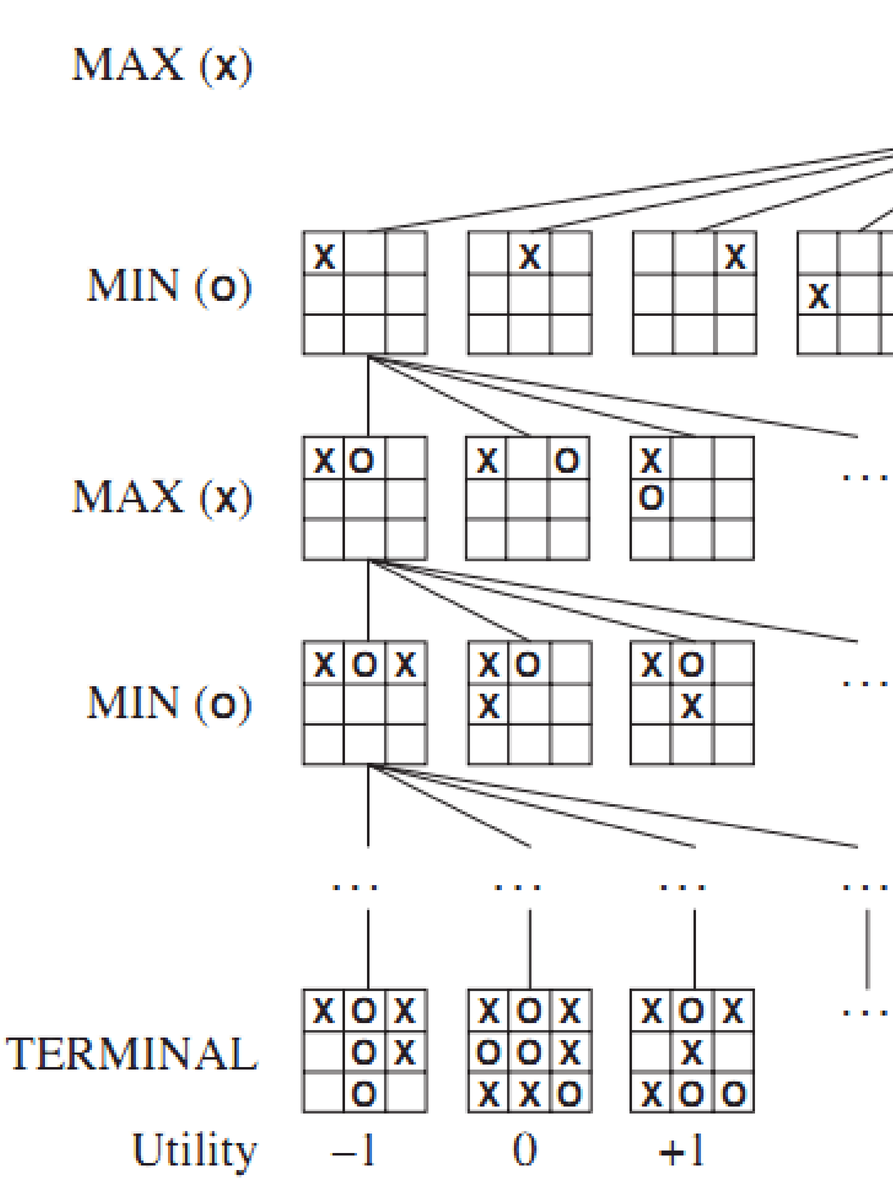
Game setup

- Two players: *MAX* and *MIN*
- *MAX* moves first and they take turns until the game is over.
  - Winner gets award,
  - Looser gets penalty.
- MAX uses search tree to determine next move.

Game Search

- Problem Formulation:
  - States: board configuration of chess
  - Successor function: legal moves a player can make.
  - Goal test: determines when the game is over.
  - initial state: start board configuration
  - Utility function: measures the outcome of the game and its desirability
- Search objective:
  - Find the sequence of player's decisions (moves) maximizing its utility
  - Consider the opponent's moves and their utility

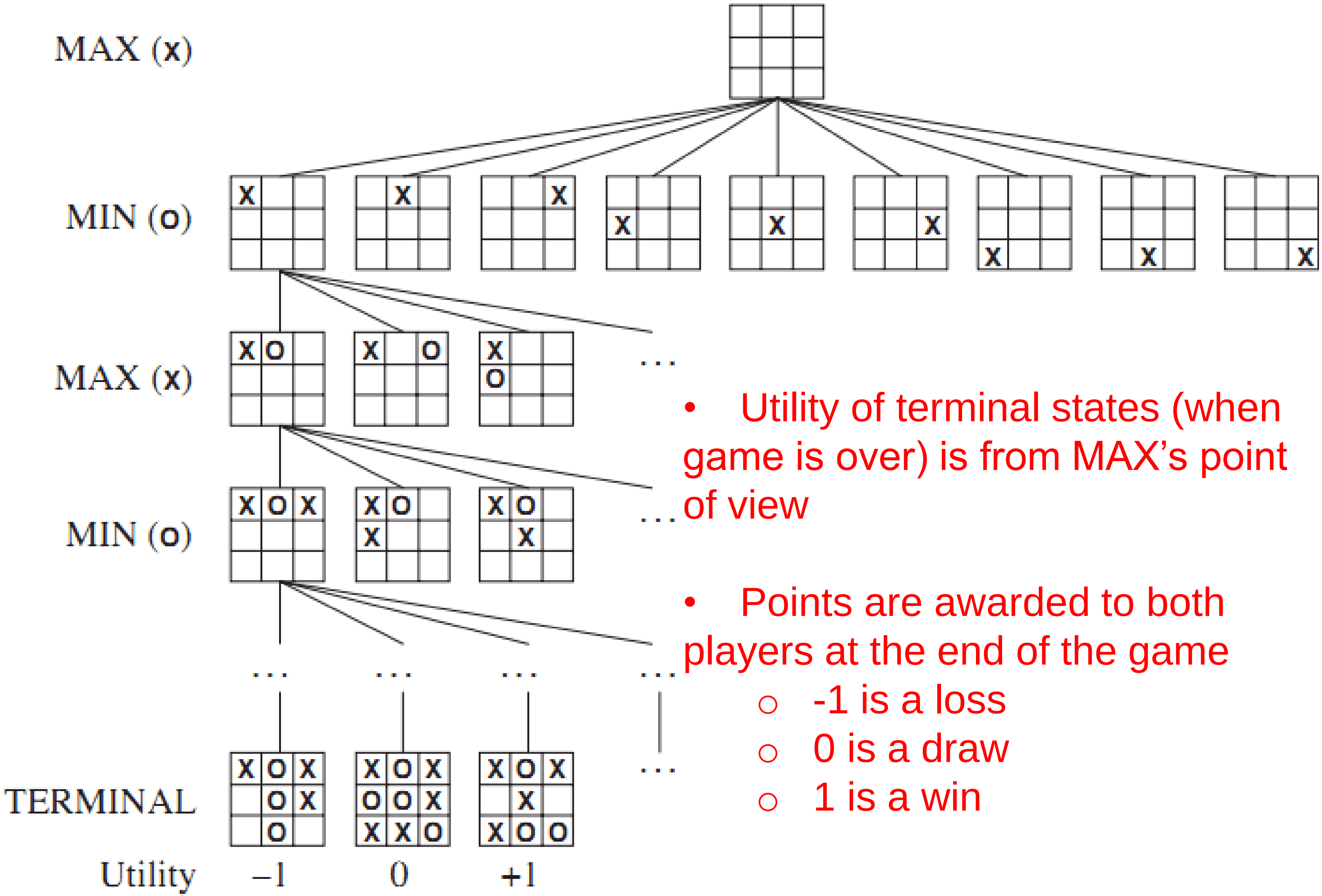
Game Tree



- MAX and MIN are the 2 players
- MAX goes first
- Players then take turns

- **MAX has 9 possible legal first moves**

Game Tree



- Utility of terminal states (when game is over) is from MAX's point of view
- Points are awarded to both players at the end of the game
  - -1 is a loss
  - 0 is a draw
  - 1 is a win

Game Playing as Search: Complexity

- Assume the opponent's moves *can* be predicted given the computer's moves.
- How **complex** would search be in this case?
  - Worst case: $O(b^d)$, **b**ranching factor, **d**epth
  - **Tic-Tac-Toe**: ~5 legal moves, max of 9 moves
    - $5^9 = 1,953,125$ states
  - **Chess**: ~35 legal moves, ~100 moves per game
    - $35^{100} \sim 10^{154}$ states (but "only" $\sim 10^{40}$ legal states)

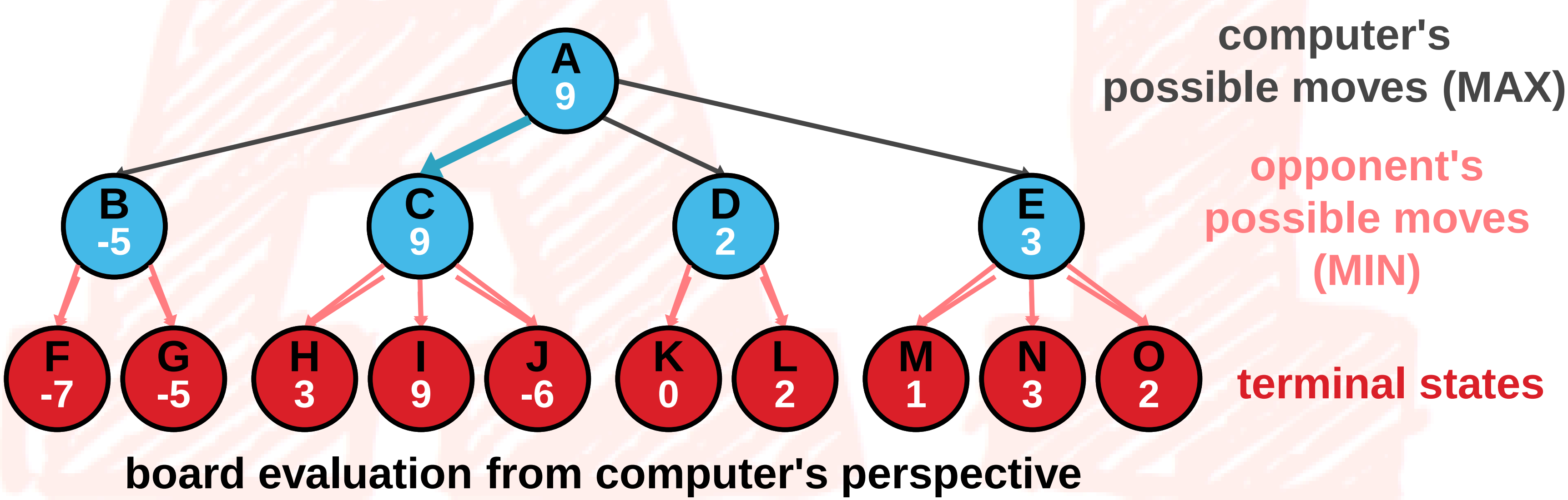
Common games produce enormous search trees!!

Greedy Search Game Playing

- A *utility* function maps each terminal state of the board to a numeric value corresponding to the value of that state to the computer.
  - positive for winning, large + means better for computer (MAX)
  - negative for losing, large – means better for opponent (MIN)
  - zero for a draw
  - typical values (lost to win):
    - –infinity to +infinity
    - –1.0 to +1.0

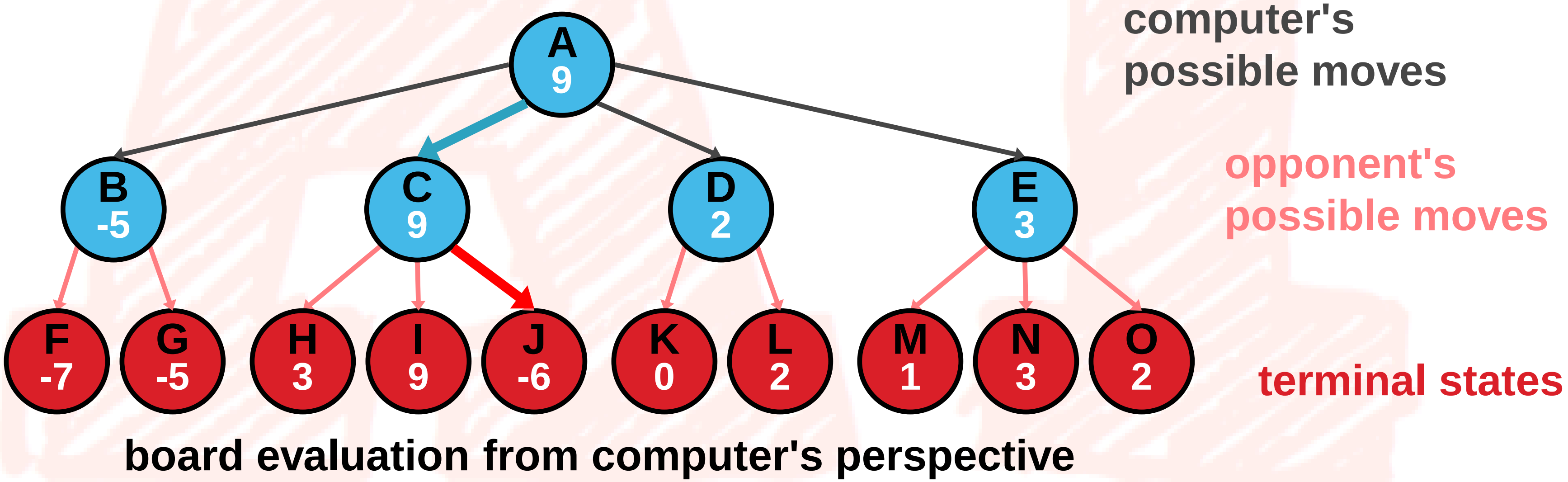
Greedy Search Game Playing

- Expand each branch to the terminal states
- Evaluate the utility of each terminal state
- Choose the move that results in the board configuration with the maximum value



Greedy Search Game Playing

- Assuming a reasonable search space, what's the problem with greedy search?
  - It ignores what the opponent might do!
  - e.g. MAX (computer) chooses C.
    MIN (opponent) chooses J and defeats computer.



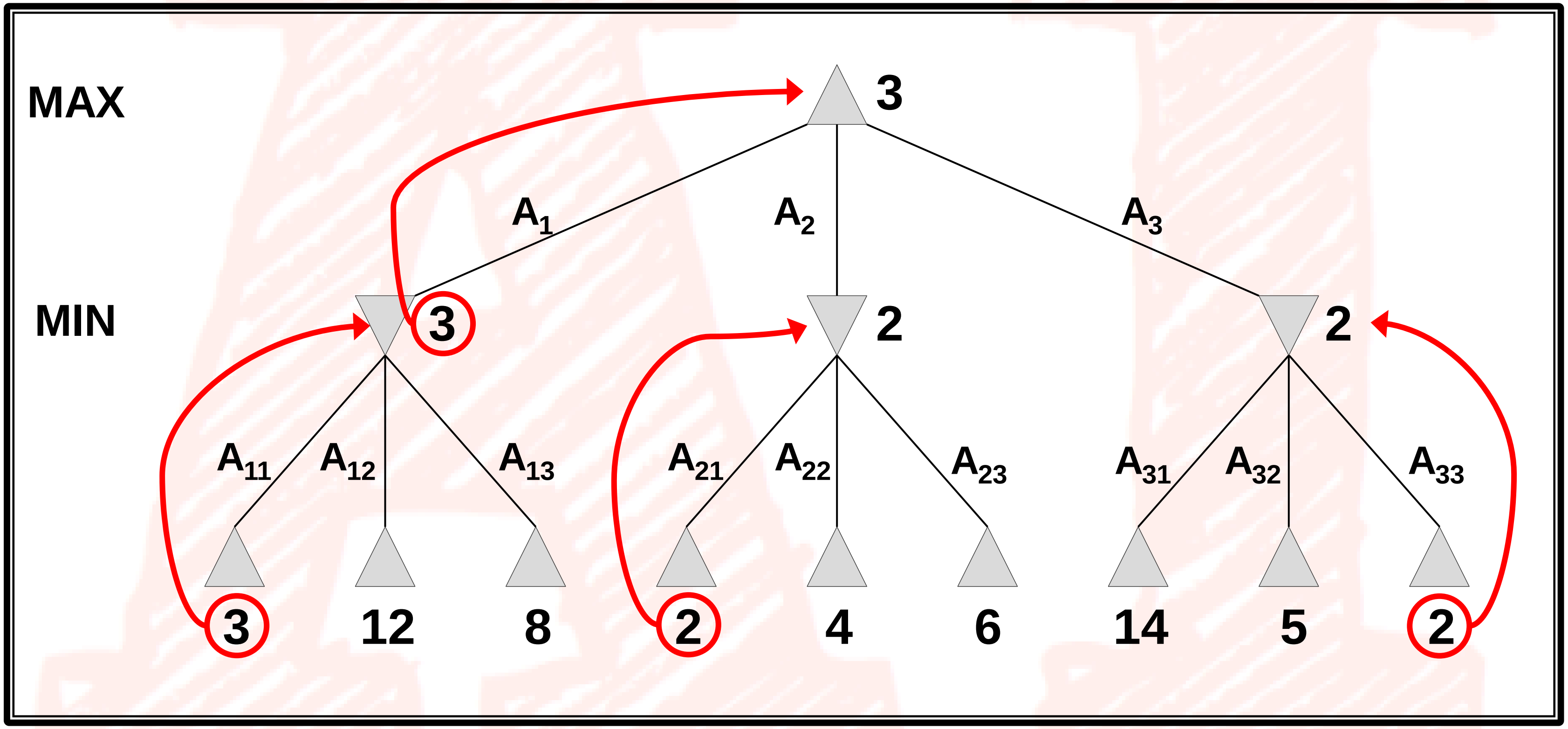
Minimax principle – Optimal strategies

- Chooses the best move considering both its move and the opponent's best move
- Assumption: Both players play optimally!!
  - **MAX** (computer) **maximizing** the utility under the assumption after it moves **MIN** (opponent) will choose the **minimizing** move.
- Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

MINIMAX-VALUE(n) =
UTILITY(n) If n is a terminal
$\max_{s \in \text{successors}(n)}$ MINIMAX-VALUE(s), If n is a max node
$\min_{s \in \text{successors}(n)}$ MINIMAX-VALUE(s), If n is a min node

Two-Players Game Tree

The minimax decision



Minimax maximizes the worst-case outcome for max.

What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally: maximizes worst-case outcome for MAX.
- But if MIN does not play optimally, MAX will do even better. [Can be proved.]

Minimax: Direct Algorithm

For each move by the MAX (computer):

- Perform depth-first search to a terminal state
- Evaluate each terminal state
- Propagate upwards the minimax values
  - if opponent's move minimum value of children backed up
  - if computer's move maximum value of children backed up
- choose move with the maximum of minimax values of children
- Note:
  - minimax values gradually propagate upwards as DFS proceeds: i.e., minimax values propagate up in “left-to-right” fashion
  - minimax values for sub-tree backed up “as we go”, so only O(bd) nodes need to be kept in memory at any time

Minimax Algorithm

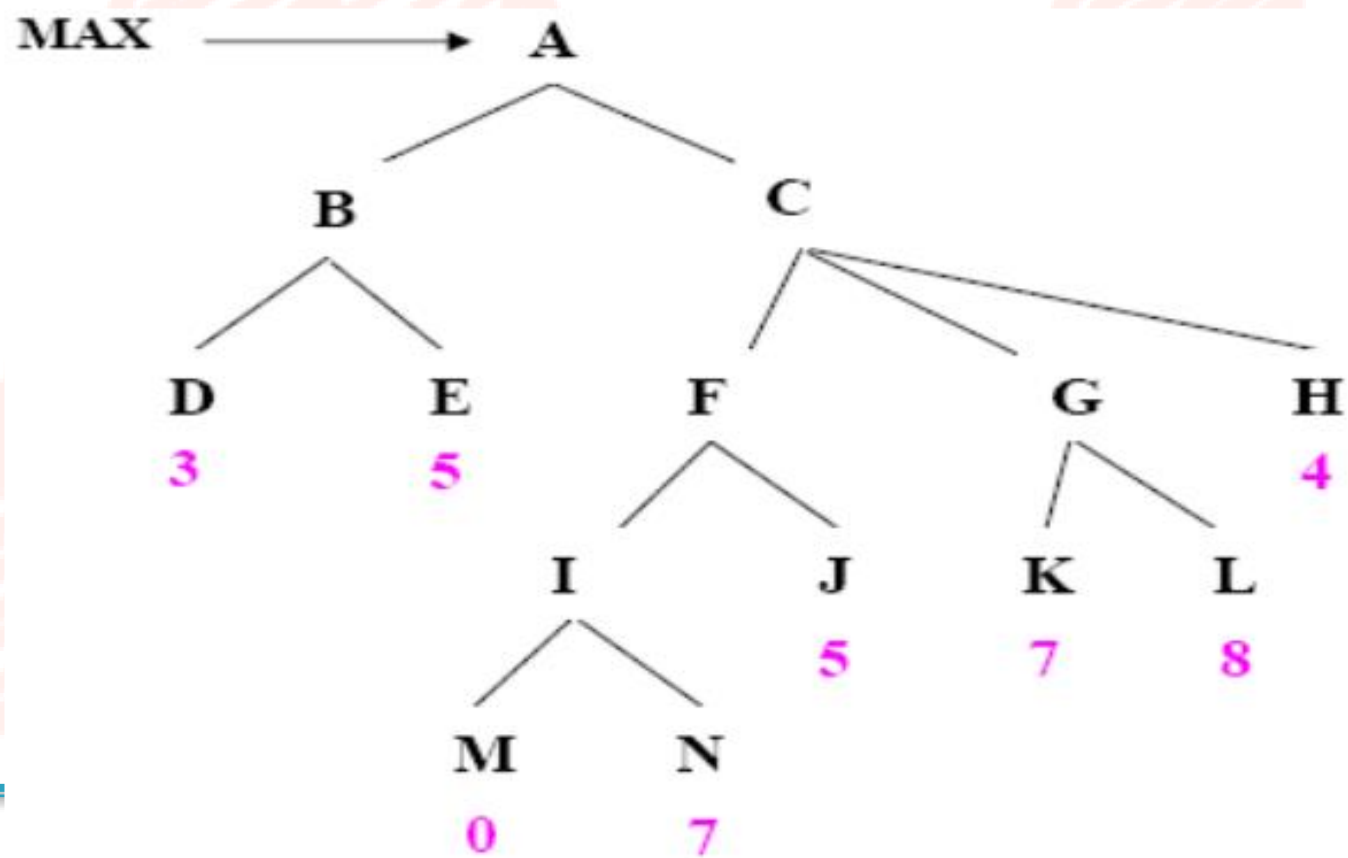
```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← -∞
   for each s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s))
   return v
```

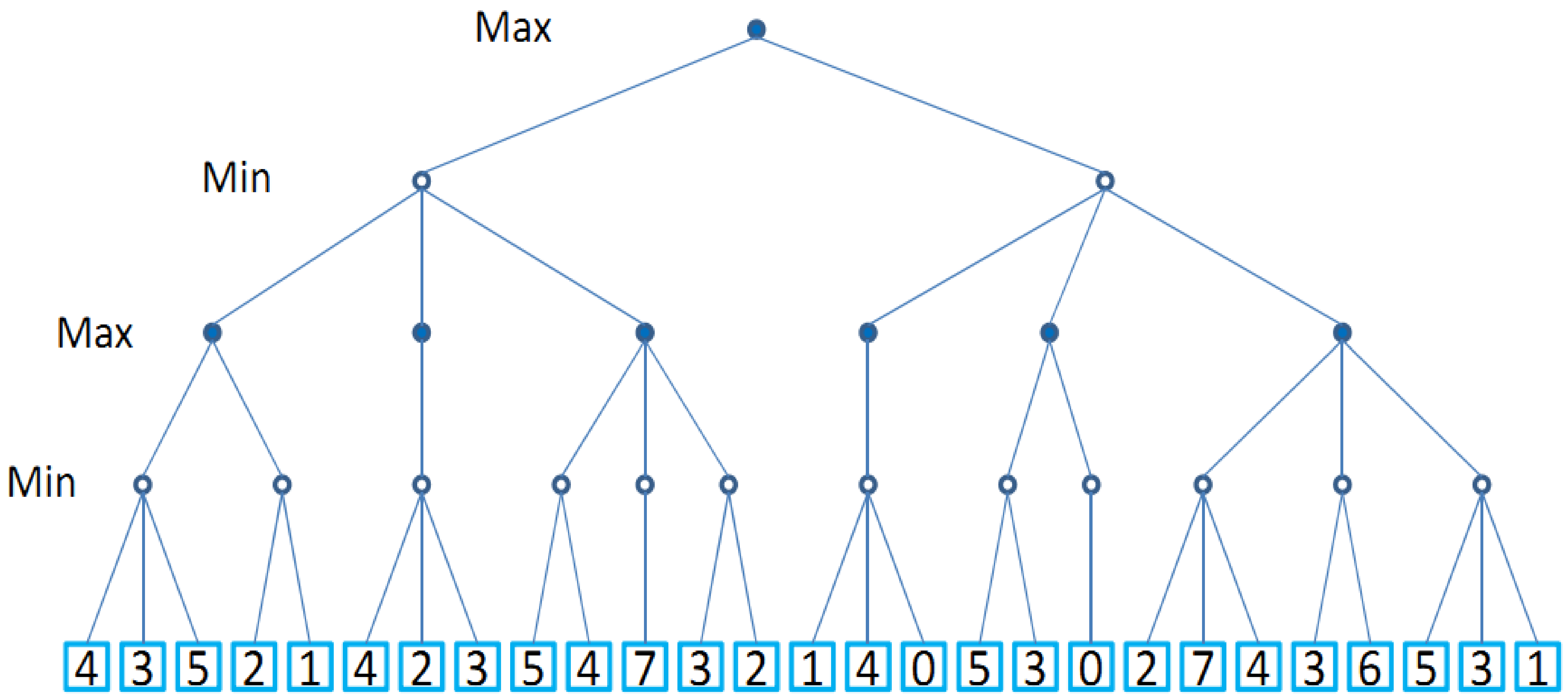
Exercise 1

- Perform the minimax algorithm on the figure below.



Exercise 2

- Perform the minimax algorithm on the figure below.



Properties of Minimax

Criterion	Minimax
Complete?	Yes (against an optimal opponent)
Time complexity	given branching factor b, $O(b^m)$
Space complexity	O(bm) (depth-first exploration)
Optimal?	Yes (if tree is finite)

- Time complexity is a major problem! Player typically only has a finite amount of time to make a move!!
- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
→ exact solution completely infeasible

Problem of minimax search

- Number of games states is exponential to the number of moves.
- Some of the branches of the game tree won't be taken if playing against an intelligent opponent
- Solution: can "prune" those branches from the tree ==> Alpha-beta pruning
- While doing DFS of game tree, keep track of:
  - Alpha = Highest value found so far at any choice point along the MAX path
    - Lower bound on node's utility
  - Beta = Lowest value found so far at any choice point along the MIN path
    - Higher bound on node's utility

Alpha–Beta pruning

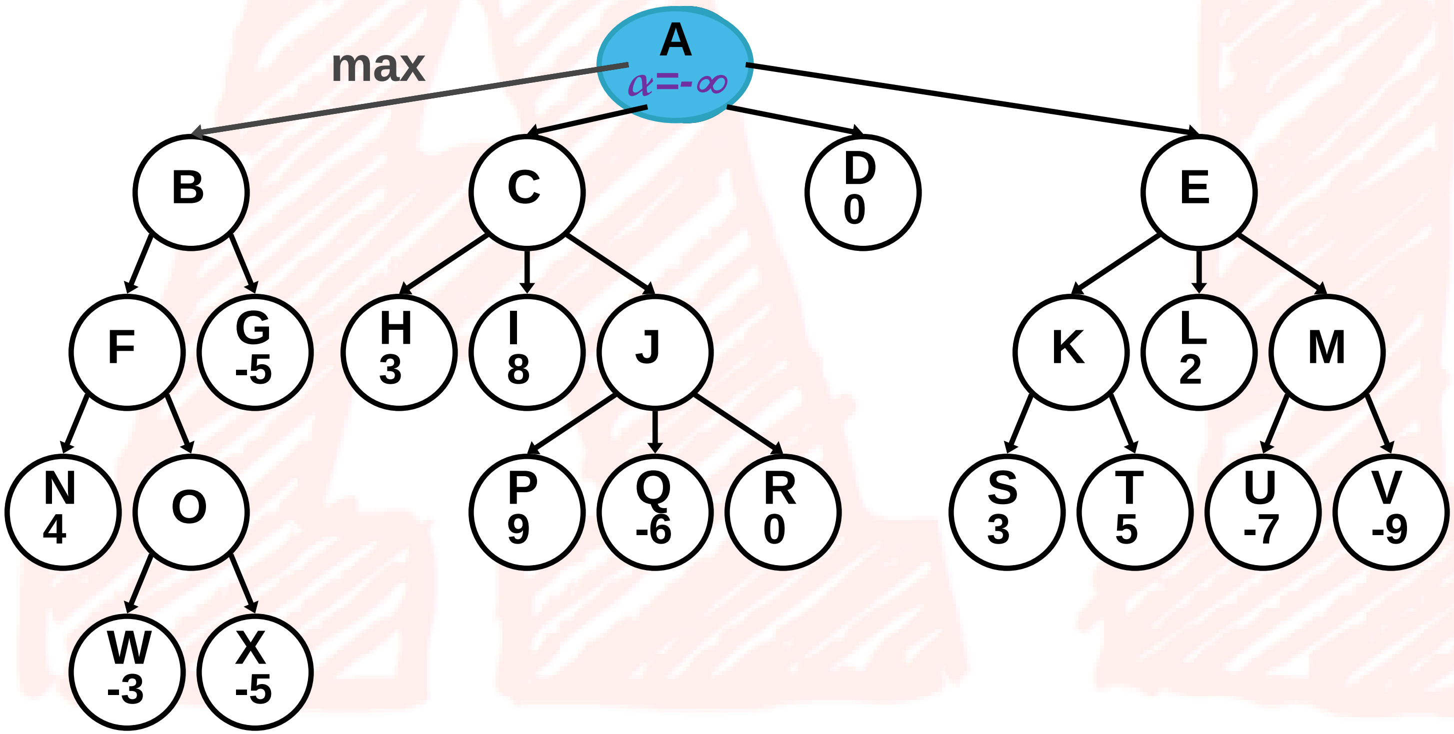
- Beta cutoff pruning
  occurs when maximizing (MAX's turn):
  - If **alpha ≥ parent's beta**, stop expanding
  - Why stop expanding children?
    - Opponent shouldn't allow the MAX to make this move

- Alpha cutoff pruning
  occurs when minimizing (MIN's turn):
  - If **beta ≤ parent's alpha**, stop expanding
  - Why stop expanding children?
    - MAX shouldn't take this route

Alpha-Beta Search Example

`minimax(A,0,4)` alpha initialized to -infinity

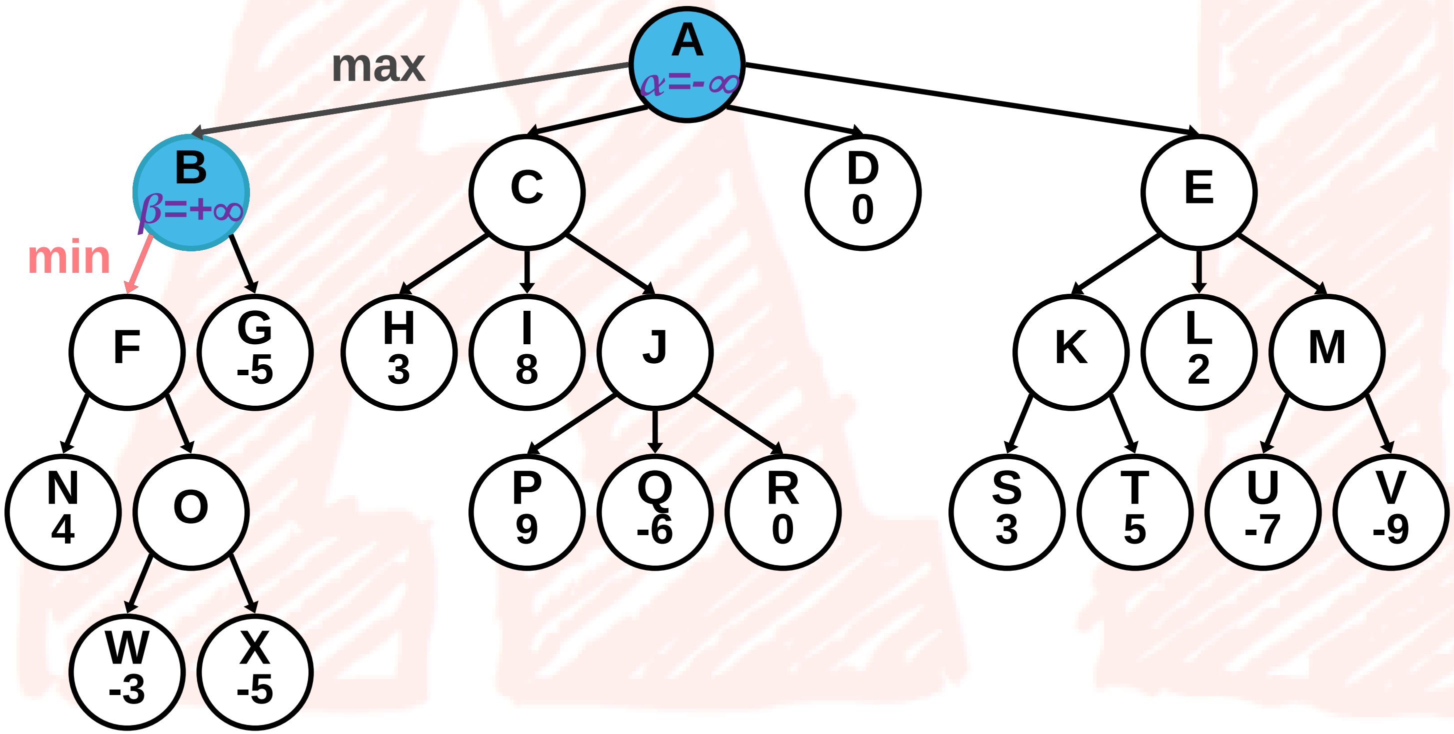
Expand A? Yes since there are successors, no cutoff test for root



Alpha–Beta Search Example

`minimax(B,1,4)` beta initialized to +infinity

Expand B? **Yes** since A's alpha >= B's beta is false, no alpha cutoff



Call Stack

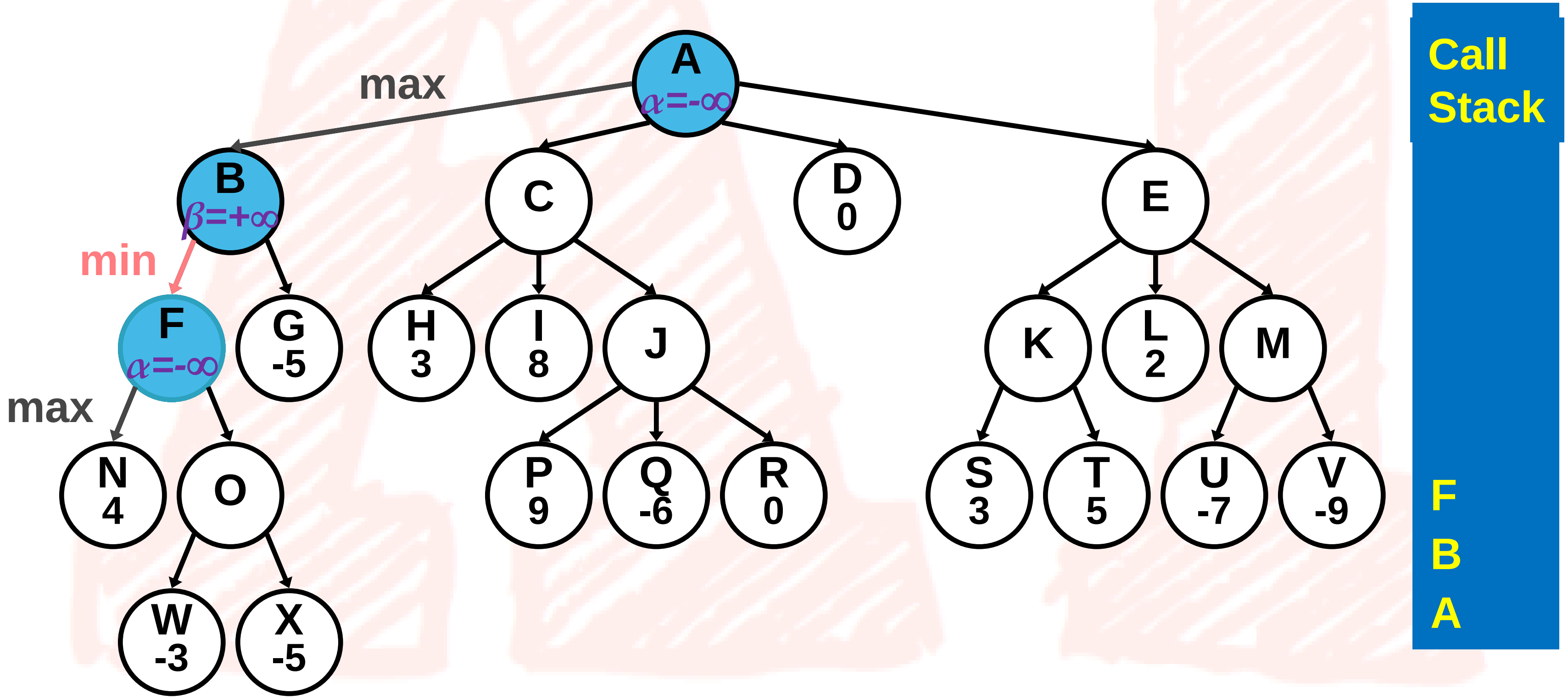
B

A

Alpha-Beta Search Example

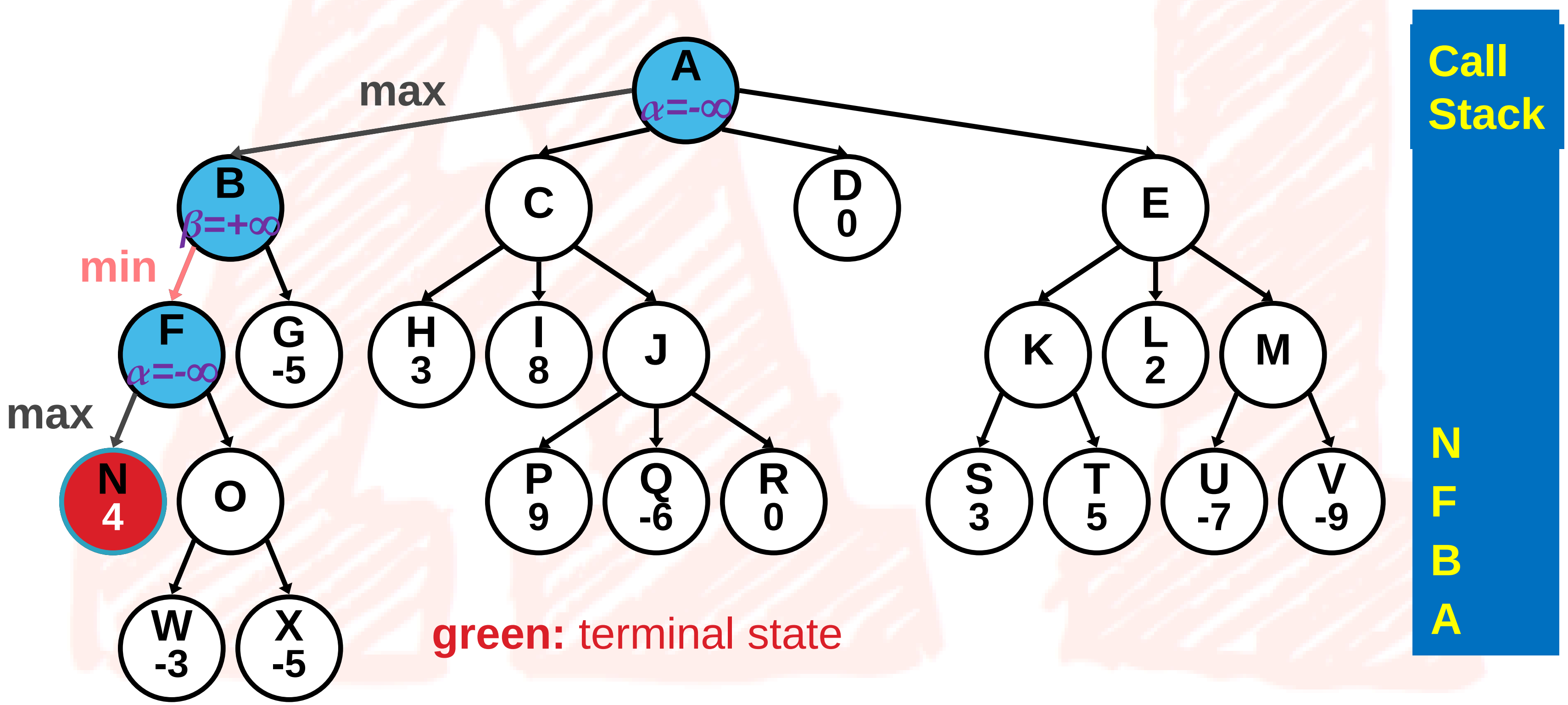
`minimax(F,2,4)` alpha initialized to -infinity

Expand F? **Yes** since F's alpha >= B's beta is false, no beta cutoff



Alpha-Beta Search Example

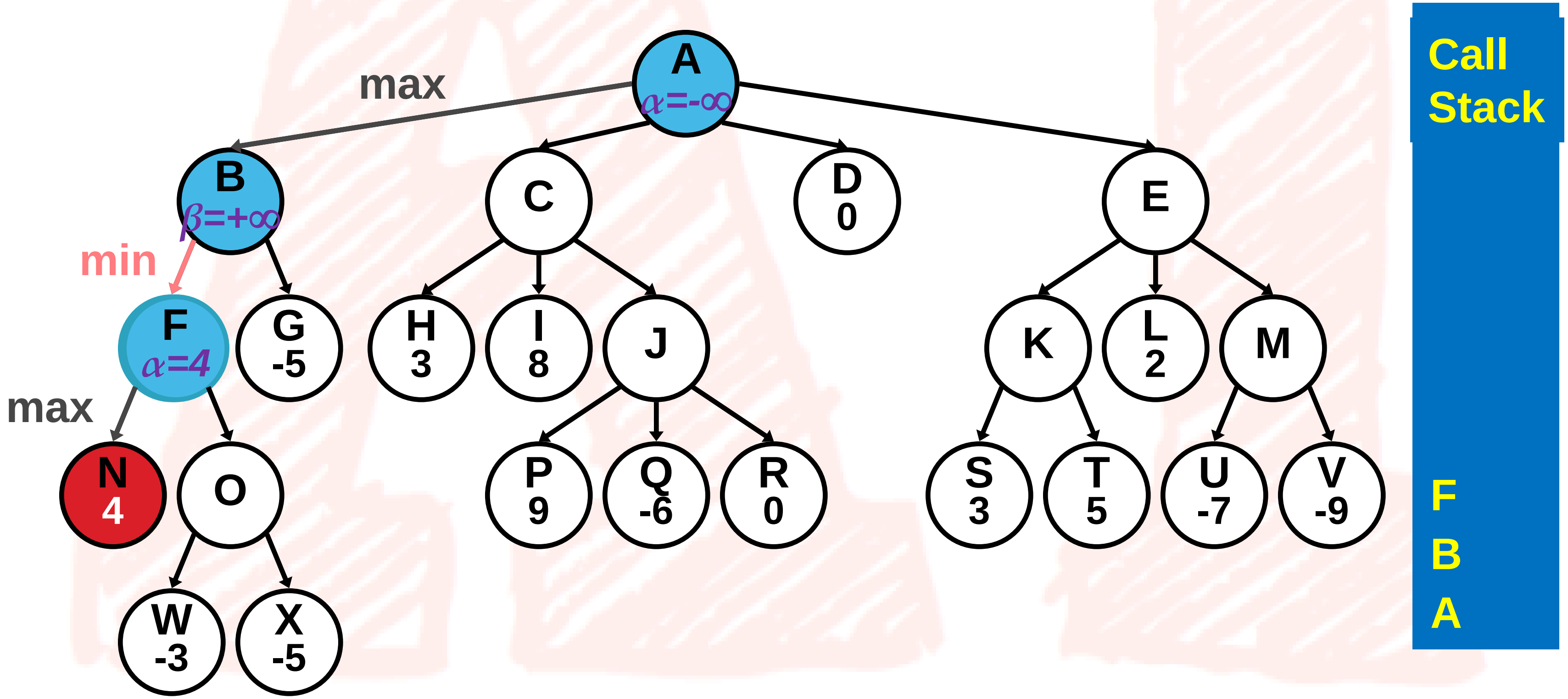
`minimax(N,3,4)` evaluate and return SBE value



Alpha-Beta Search Example

back to alpha = 4, since 4 >= -infinity (maximizing)
minimax(F,2,4)

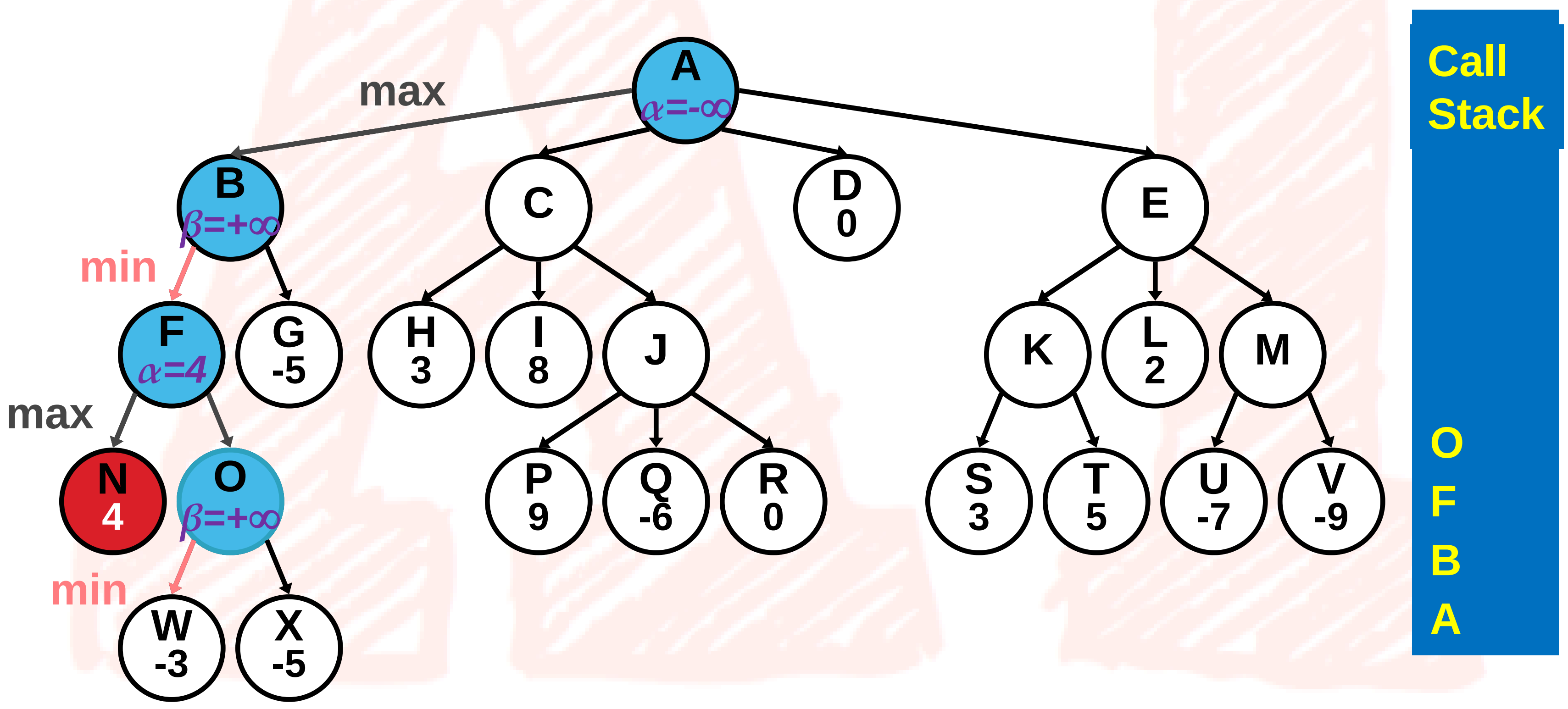
Keep expanding F? Yes since F's alpha >= B's beta is false, no beta cutoff



Alpha-Beta Search Example

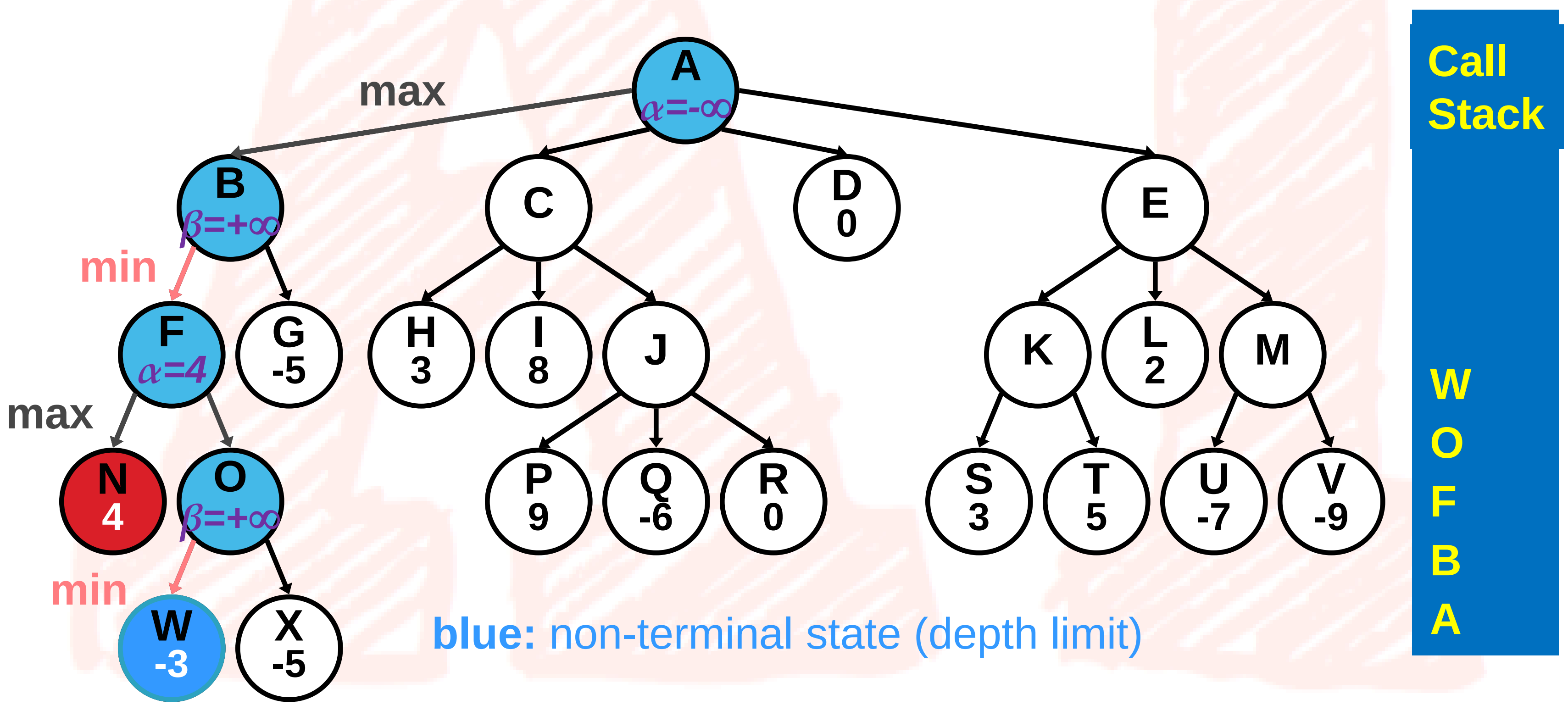
`minimax(O,3,4)` beta initialized to +infinity

Expand O? **Yes** since F's alpha >= O's beta is false, no alpha cutoff



Alpha-Beta Search Example

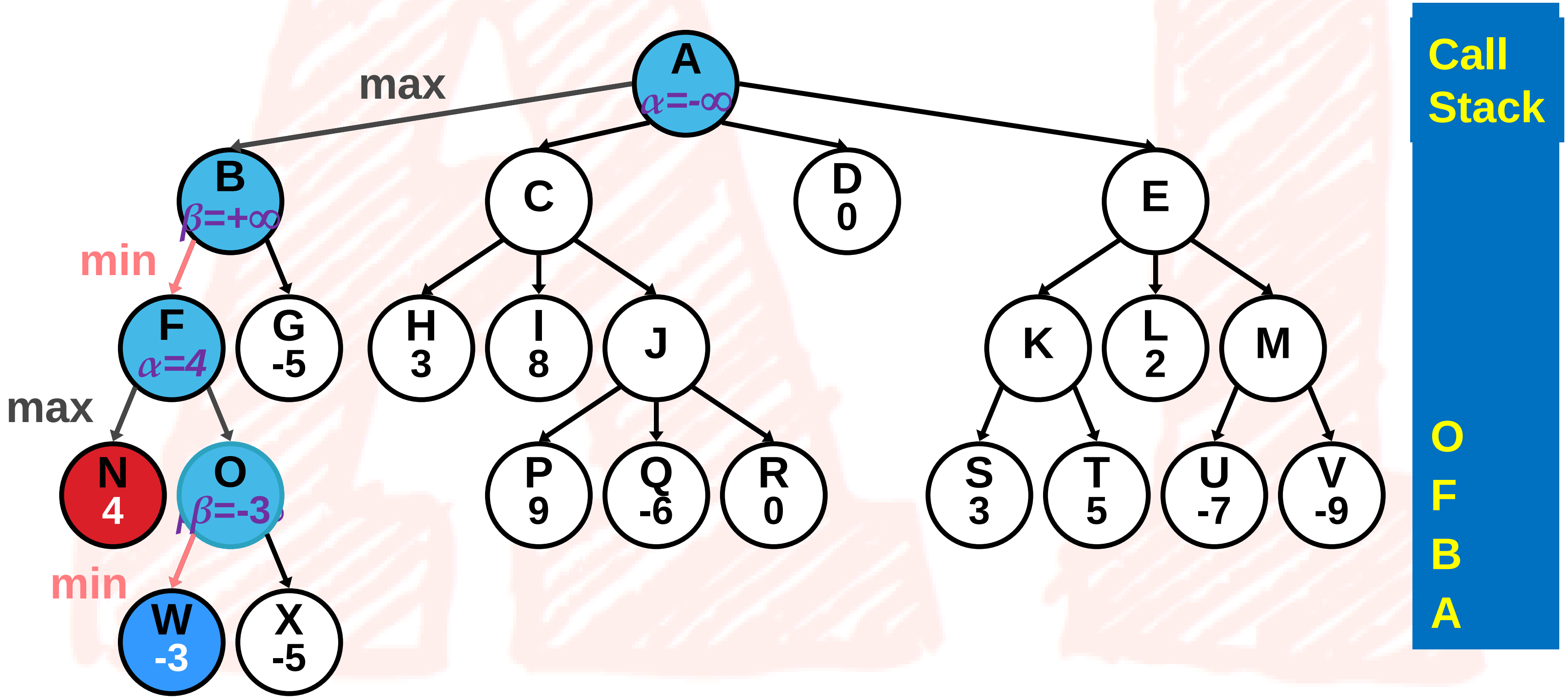
`minimax(W,4,4)` evaluate and return SBE value



Alpha-Beta Search Example

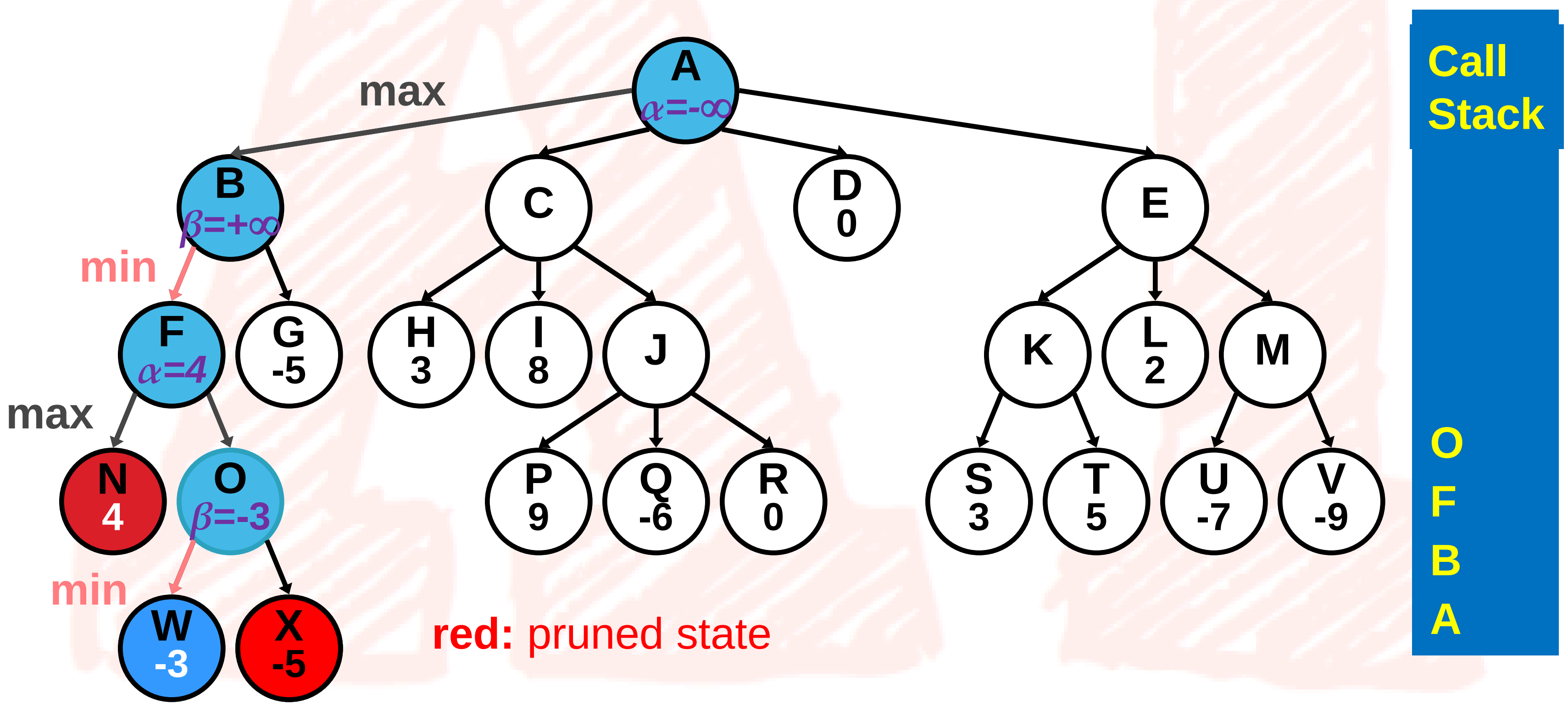
back to `minimax(O,3,4)` beta = -3, since -3 <= +infinity (minimizing)

Keep expanding O? No since F's alpha >= O's beta is true: **alpha cutoff**



Alpha-Beta Search Example

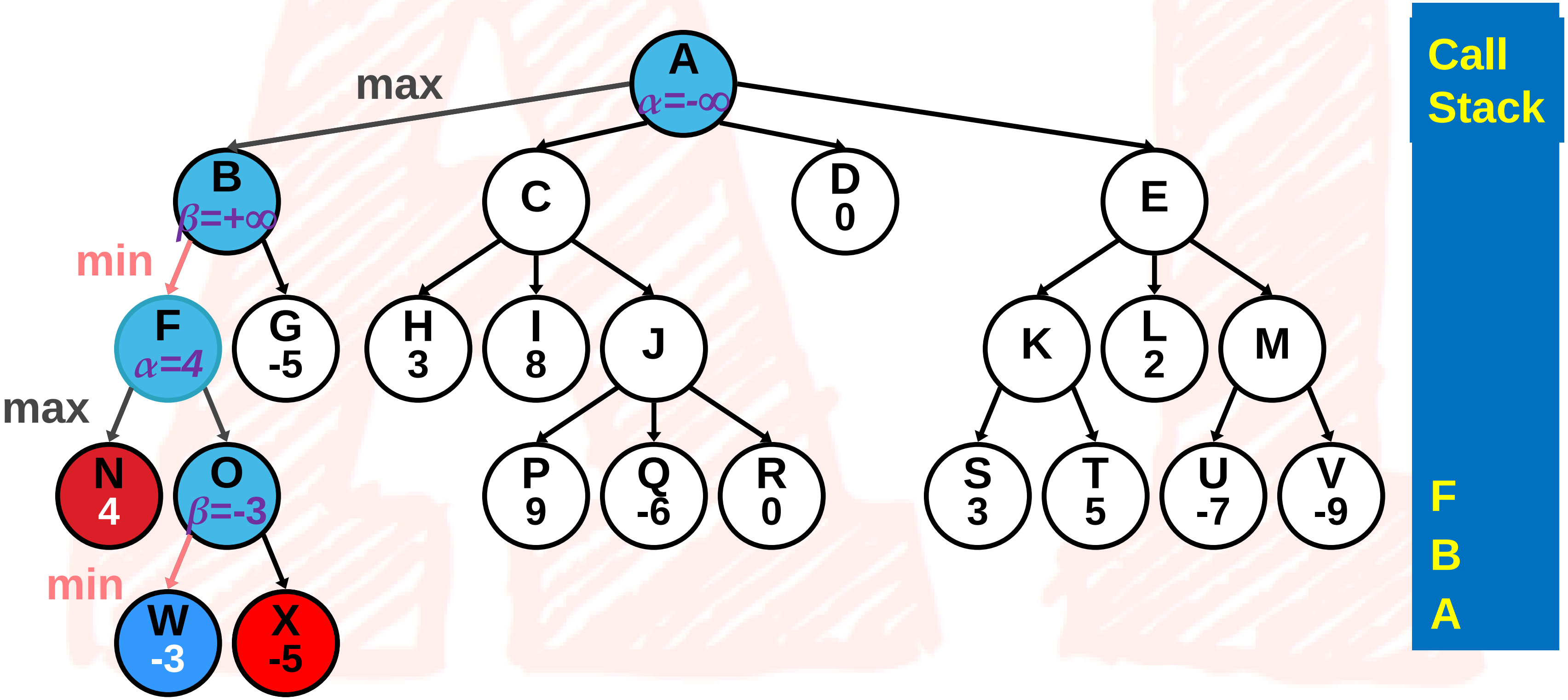
- Why?
  - Smart opponent will choose W or worse, thus O's upper bound is –3.
    Computer already has better move at N.



Alpha–Beta Search Example

back to `minimax(F,2,4)` alpha doesn't change, since -3 < 4 (maximizing)

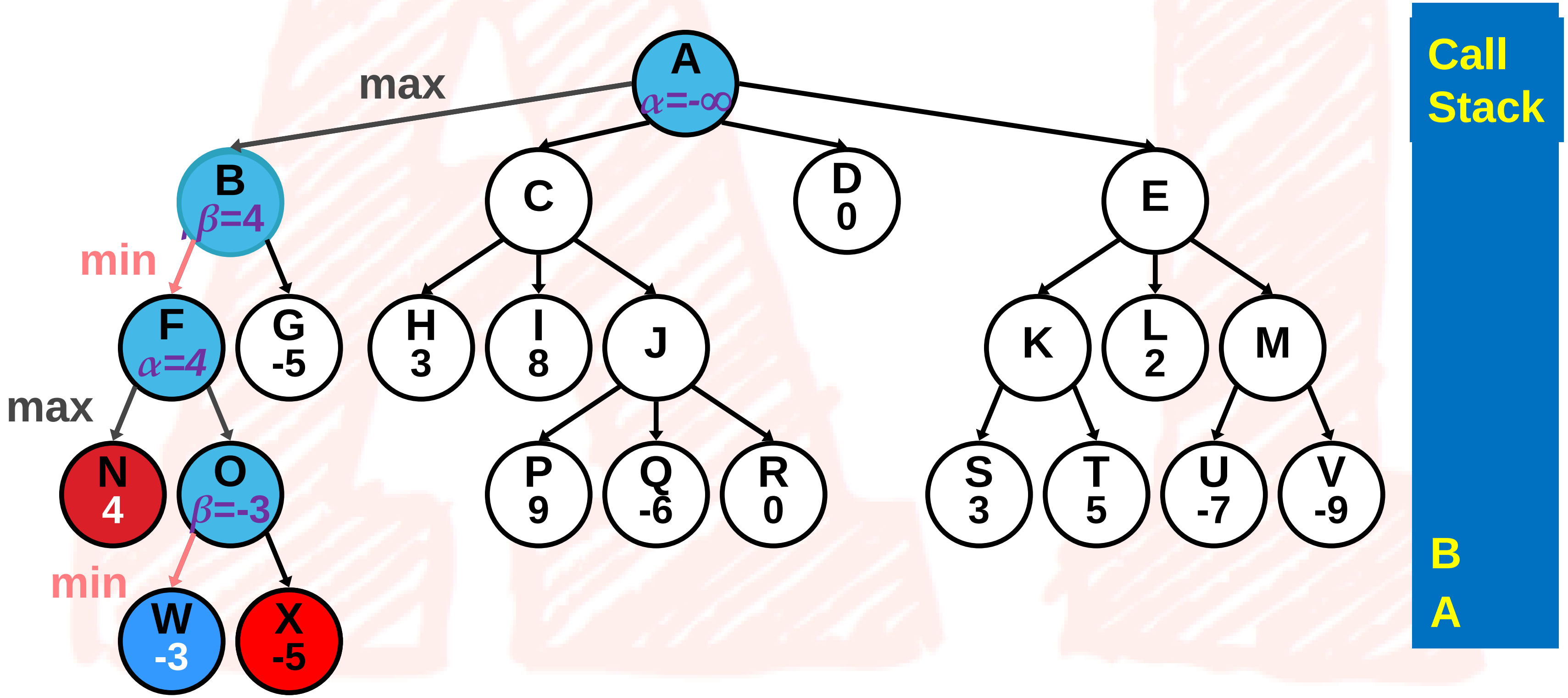
Keep expanding F? No since no more successors for F



Alpha-Beta Search Example

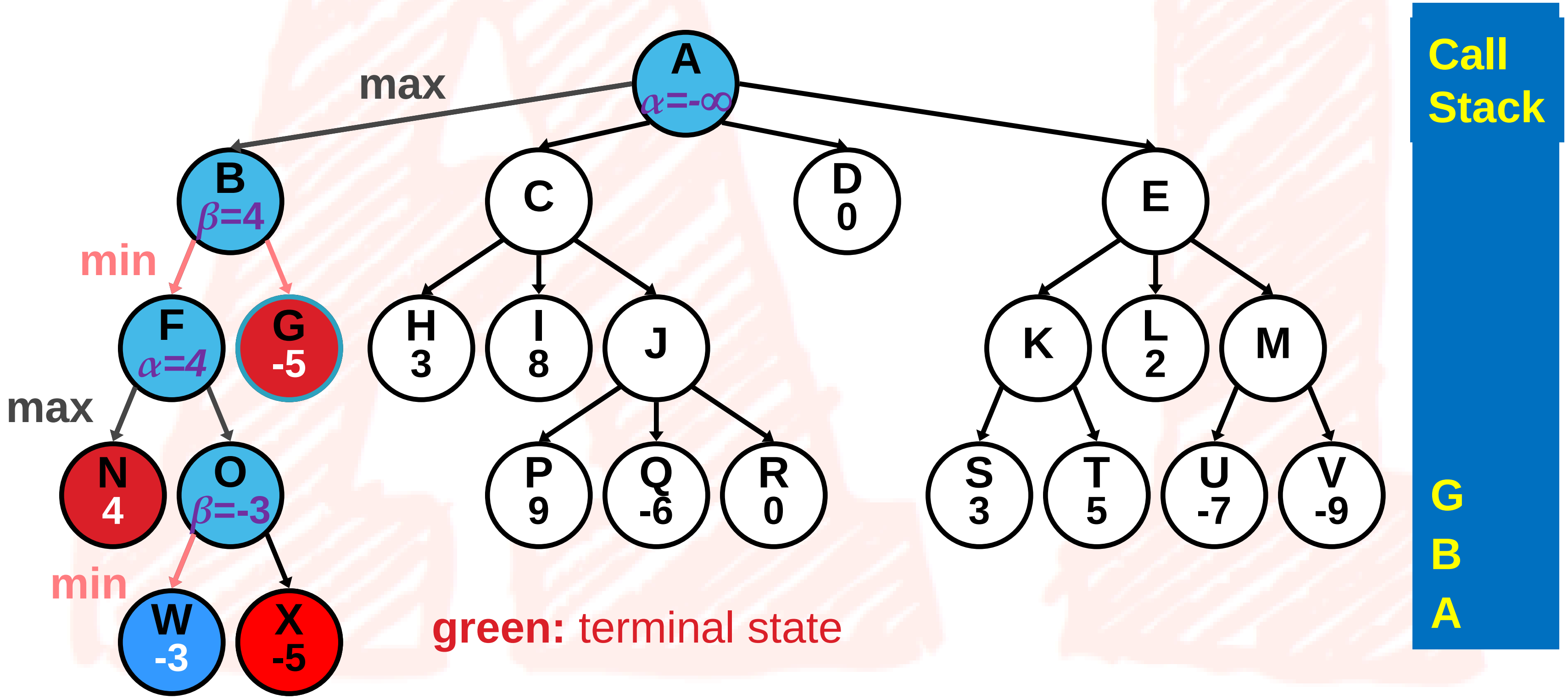
back to beta = 4, since 4 <= +infinity (minimizing)
`minimax(B,1,4)`

Keep expanding B? Yes since A's alpha >= B's beta is false, no alpha cutoff



Alpha-Beta Search Example

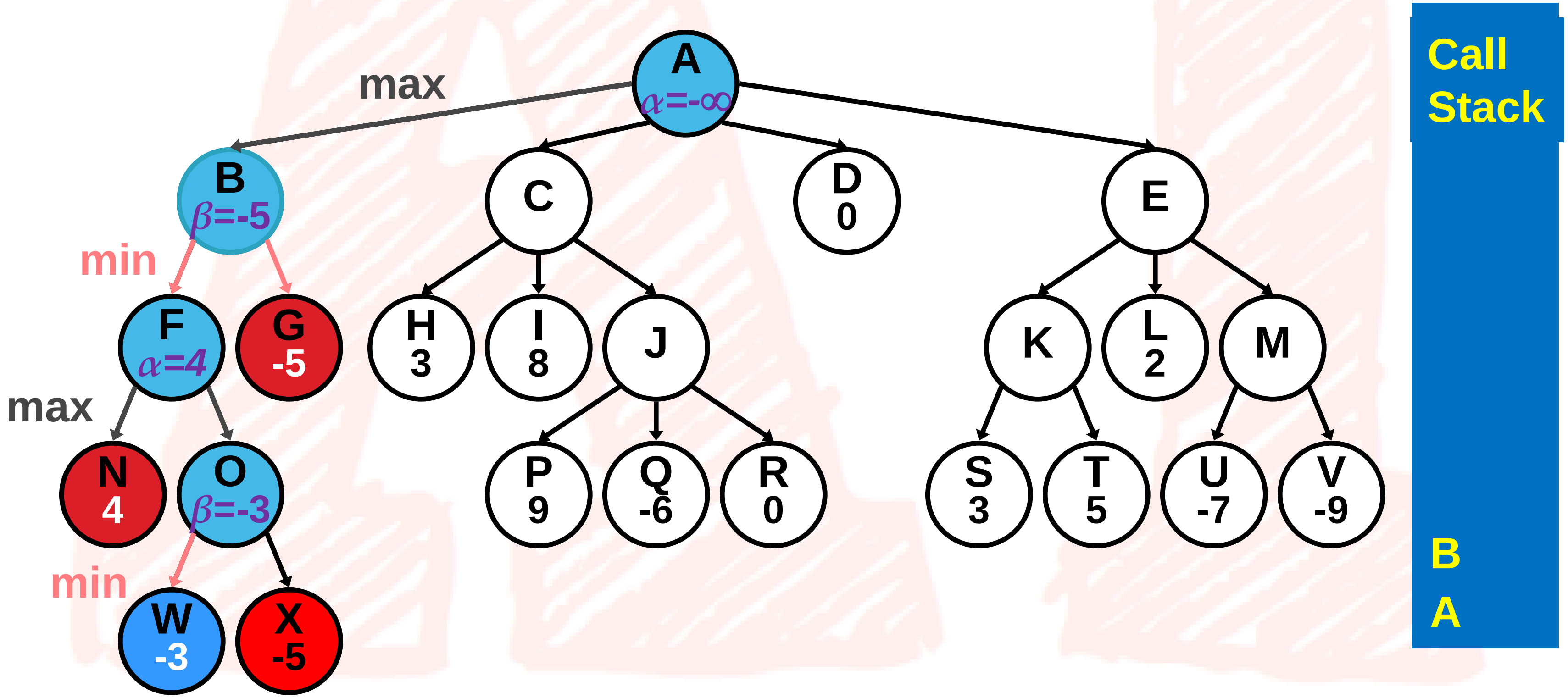
`minimax(G,2,4)` evaluate and return SBE value



Alpha-Beta Search Example

back to beta = -5, since -5 <= 4 (minimizing)
`minimax(B,1,4)`

Keep expanding B? No since no more successors for B

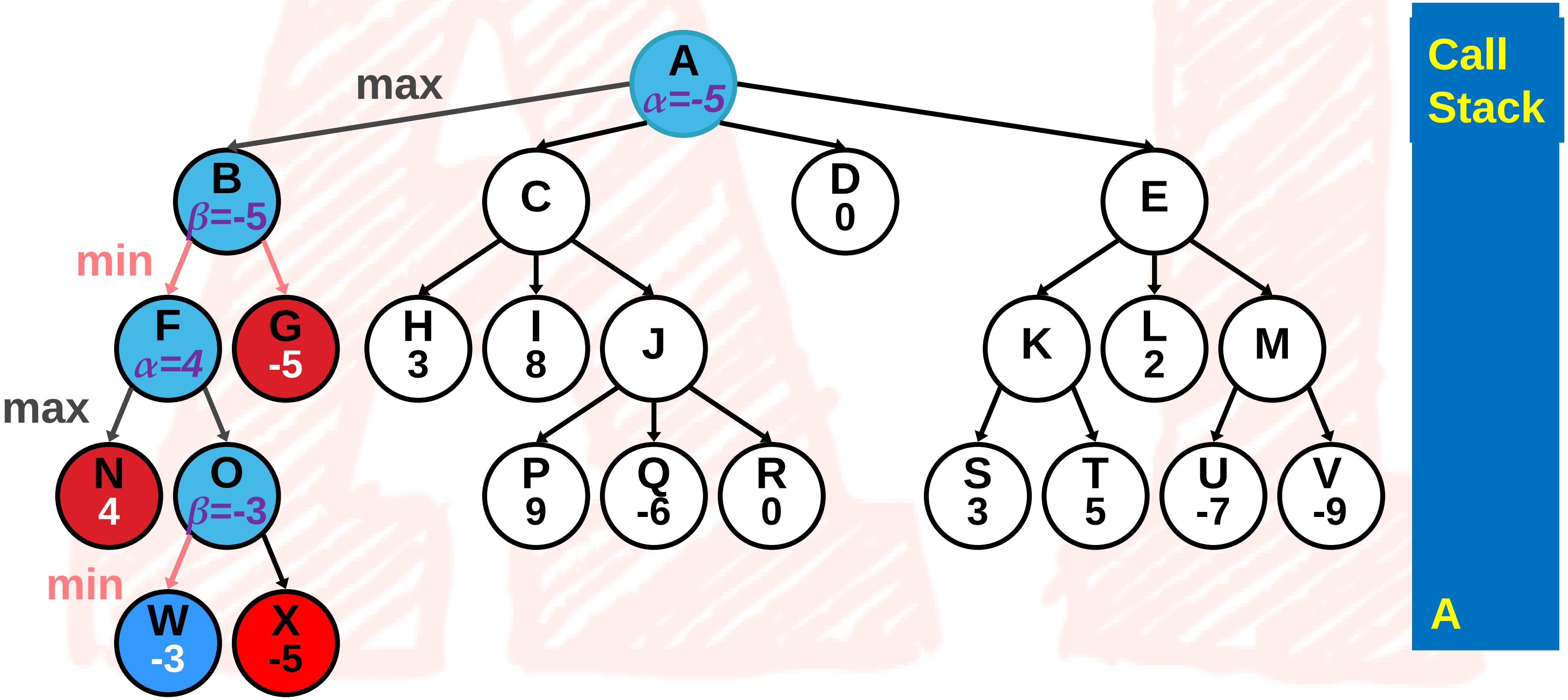


Alpha-Beta Search Example

back to alpha = -5, since -5 >= -infinity (maximizing)

`minimax(A,0,4)`

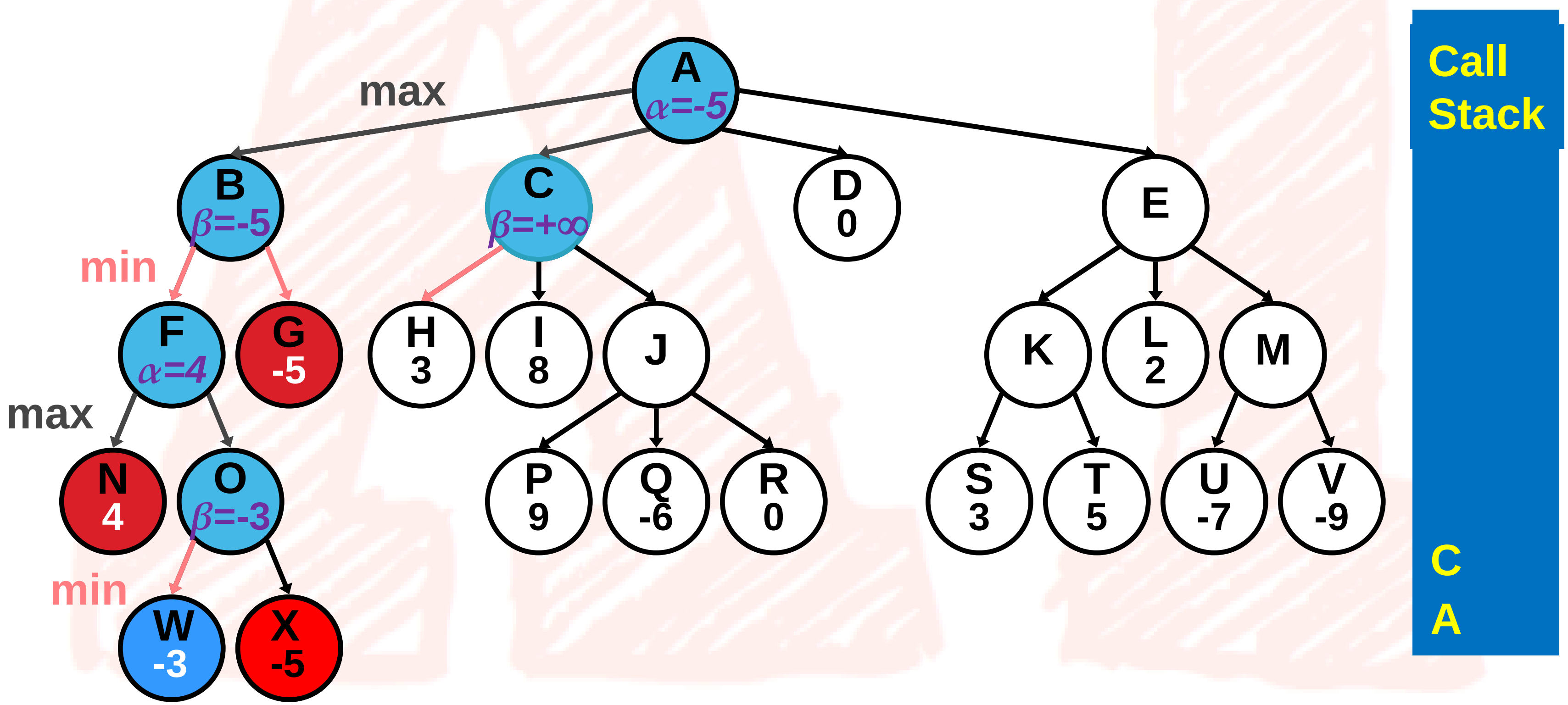
Keep expanding A? Yes since there are more successors, no cutoff test



Alpha-Beta Search Example

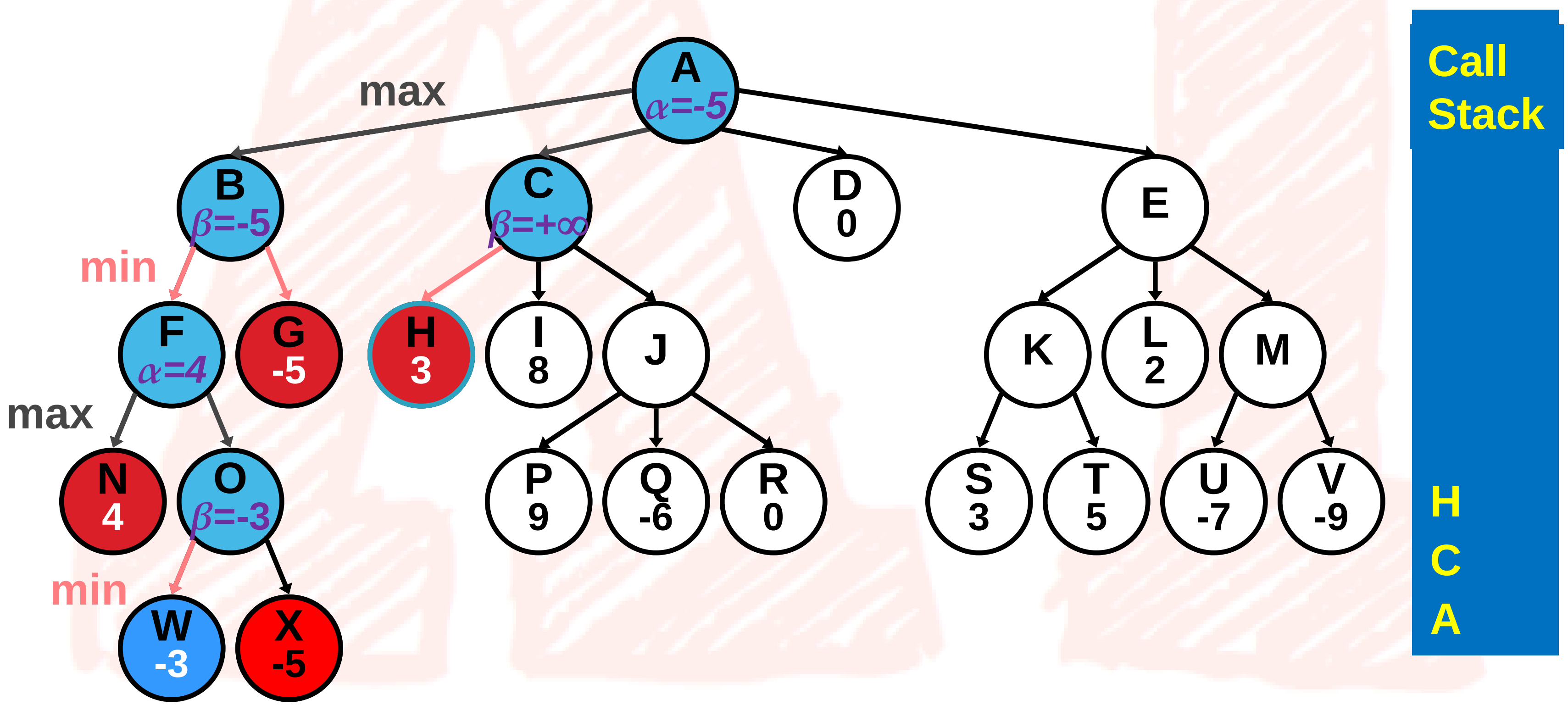
`minimax(C,1,4)` beta initialized to +infinity

Expand C? **Yes** since A's alpha >= C's beta is false, no alpha cutoff



Alpha-Beta Search Example

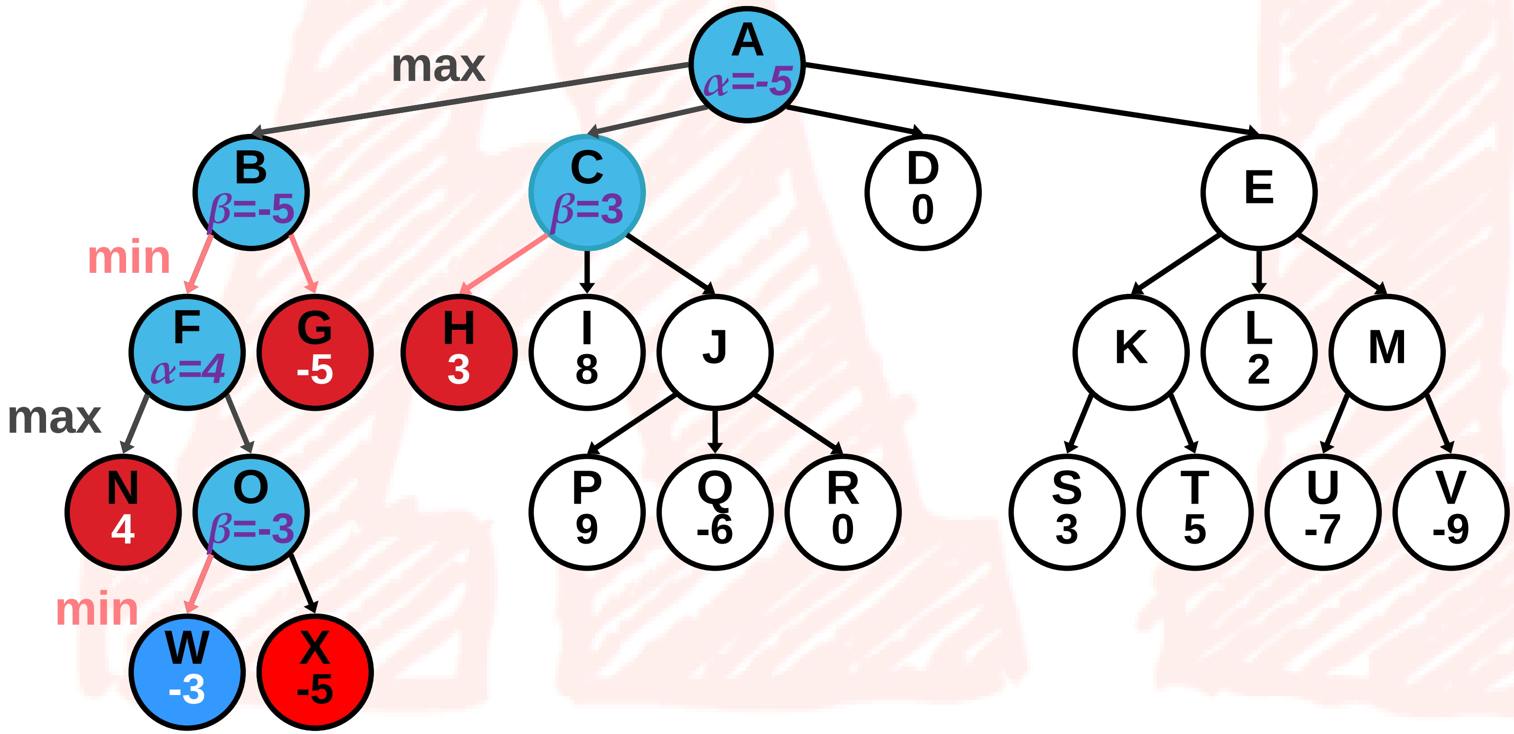
`minimax(H,2,4)` evaluate and return SBE value



Alpha-Beta Search Example

back to beta = 3, since 3 <= +infinity (minimizing)
`minimax(C,1,4)`

Keep expanding C? **Yes** since A's alpha >= C's beta is false, no alpha cutoff



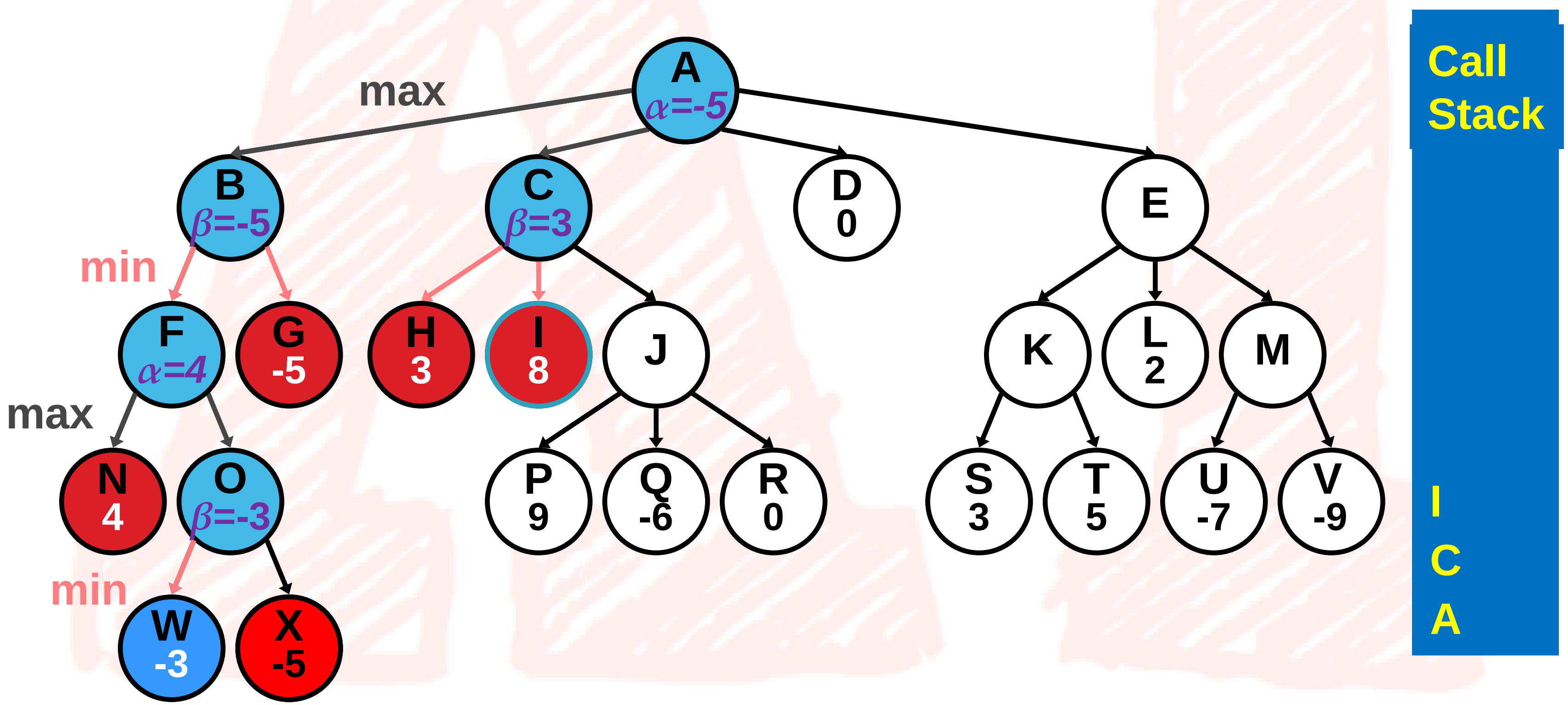
Call Stack

C

A

Alpha-Beta Search Example

`minimax(I,2,4)` evaluate and return SBE value

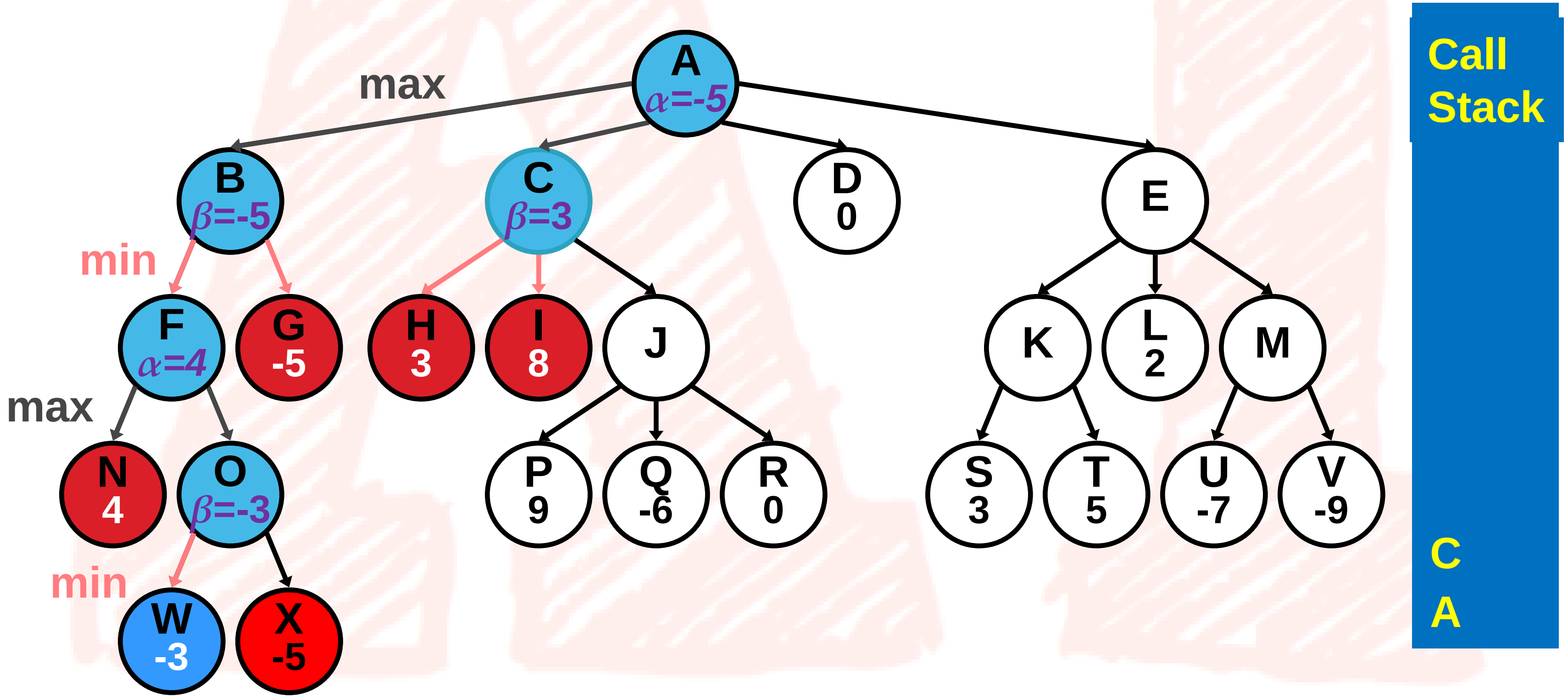


Alpha-Beta Search Example

back to `minimax(C,1,4)`

beta doesn't change, since 8 > 3 (minimizing)

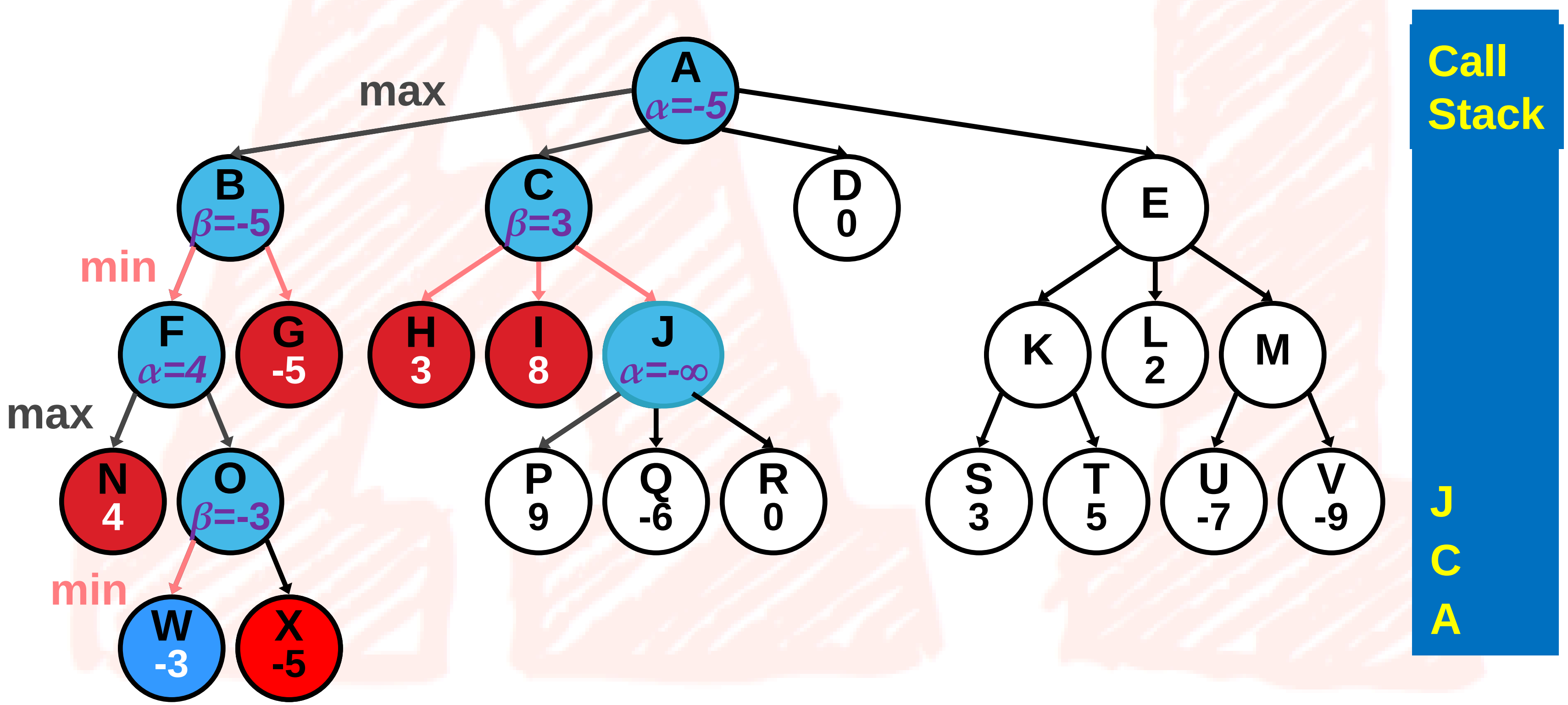
Keep expanding C? Yes since A's alpha >= C's beta is false, no alpha cutoff



Alpha-Beta Search Example

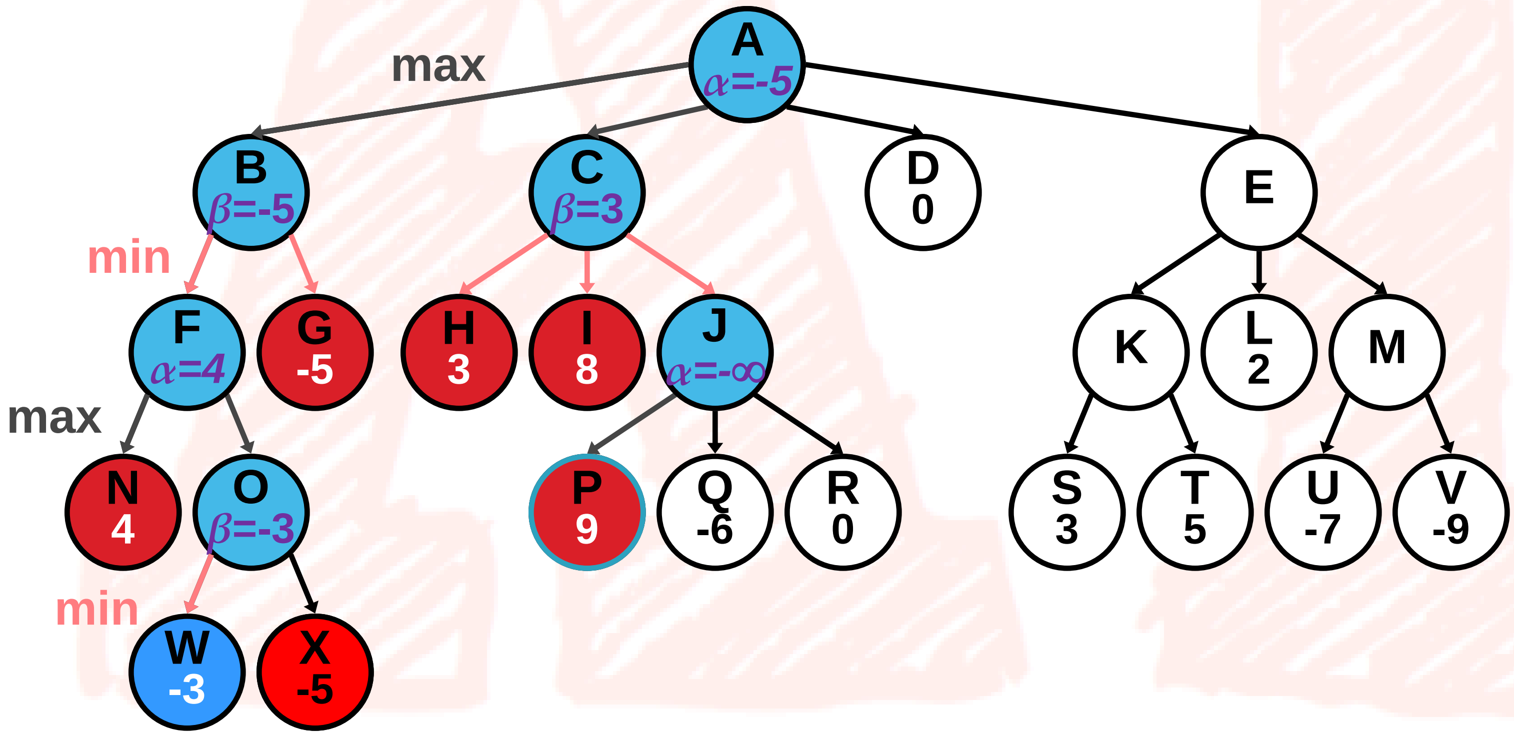
`minimax(J,2,4)` alpha initialized to -infinity

Expand J? **Yes** since J's alpha >= C's beta is false, no beta cutoff



Alpha-Beta Search Example

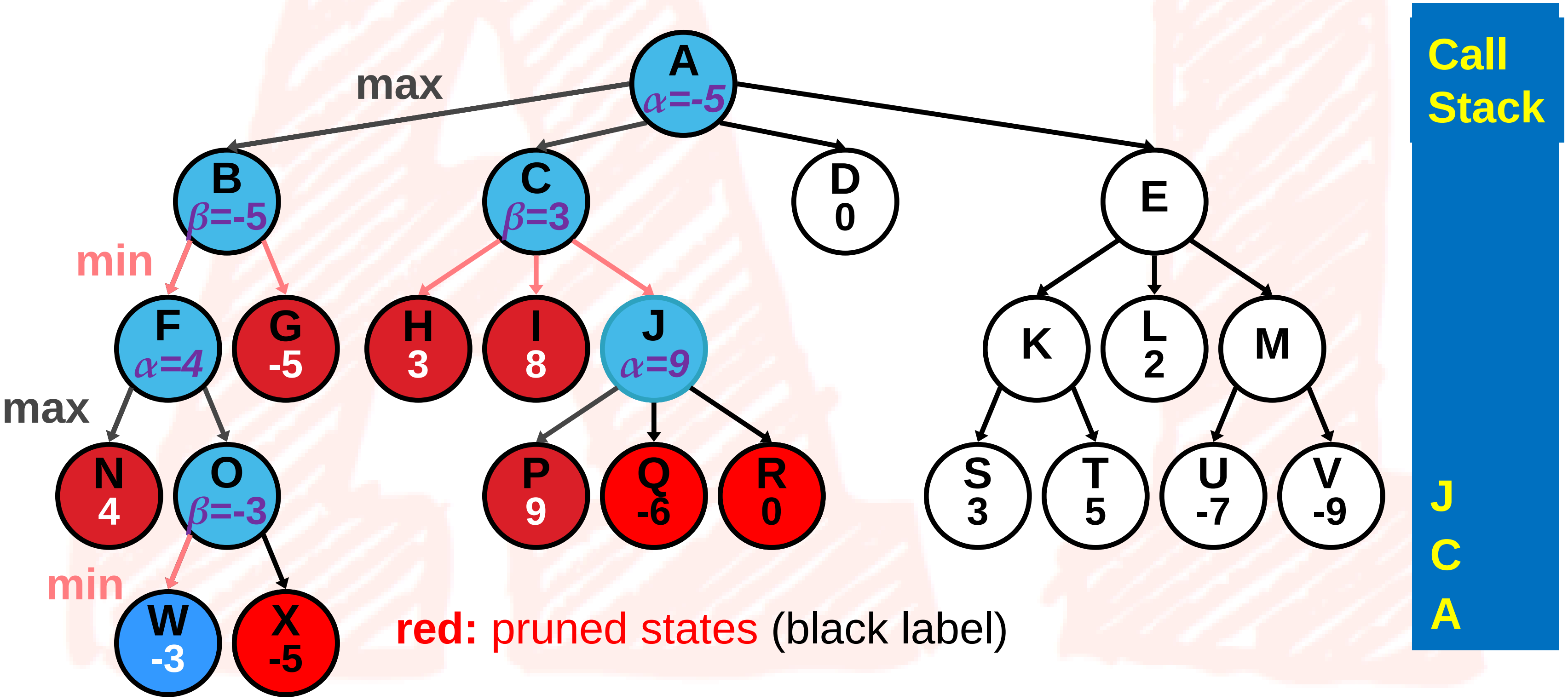
`minimax(P,3,4)` evaluate and return SBE value



Alpha-Beta Search Example

back to `minimax(J,2,4)` alpha = 9, since 9 >= -infinity (maximizing)

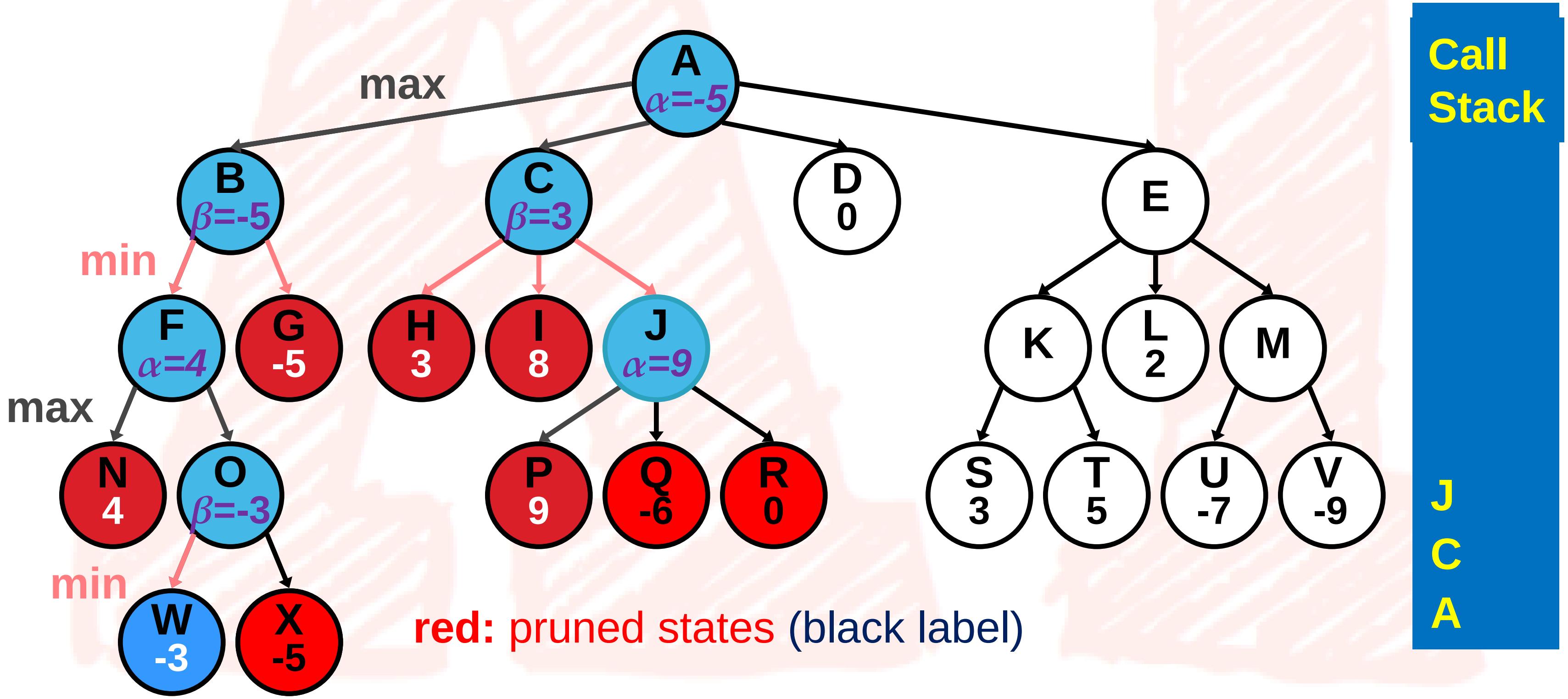
Keep expanding J? No since J's alpha >= C's beta is true: **beta cutoff**



red: pruned states (black label)

Alpha-Beta Search Example

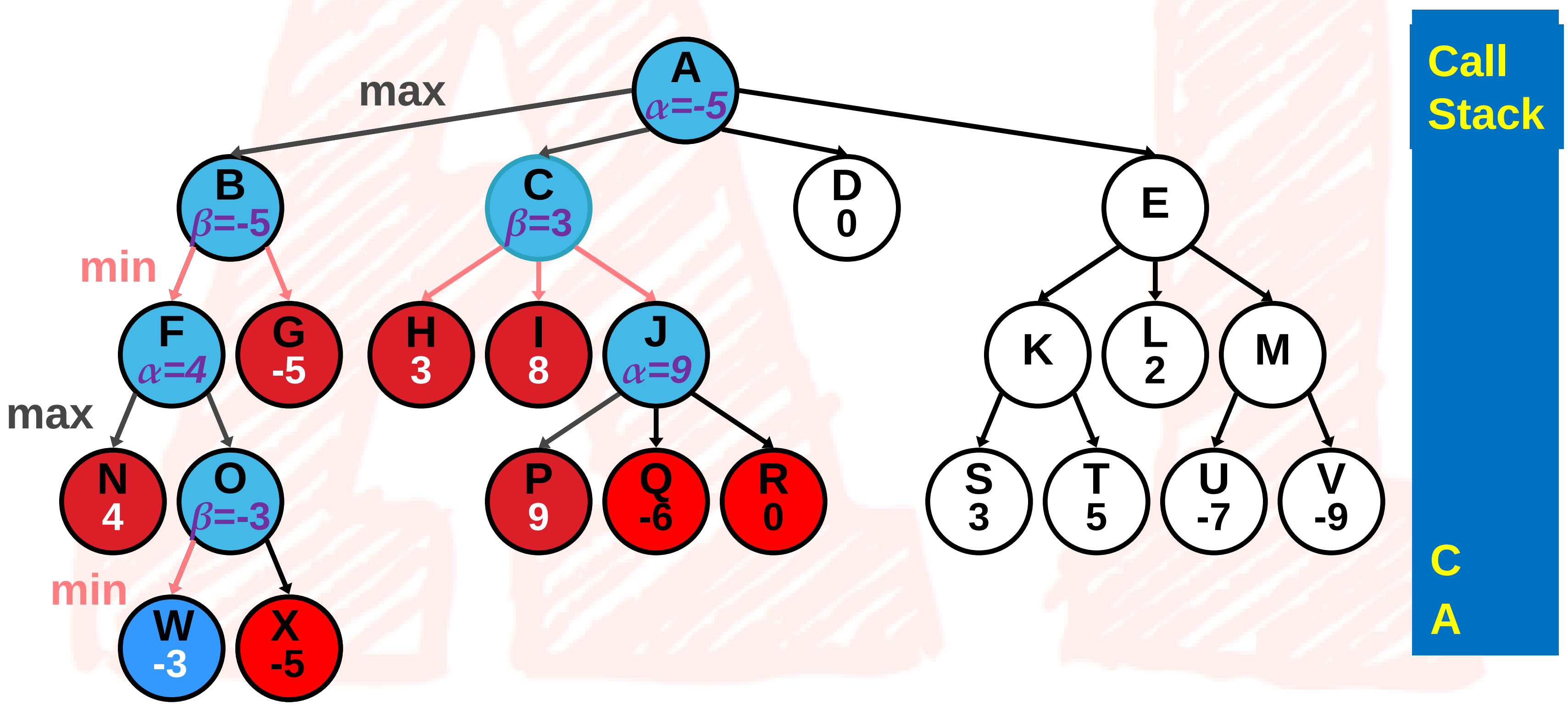
- Why?
  - Computer will choose P or better, thus J's lower bound is 9.
    Smart opponent won't let computer take move to J (since opponent already has better move at H).



Alpha-Beta Search Example

back to beta doesn't change, since 9 > 3 (minimizing)
minimax(C,1,4)

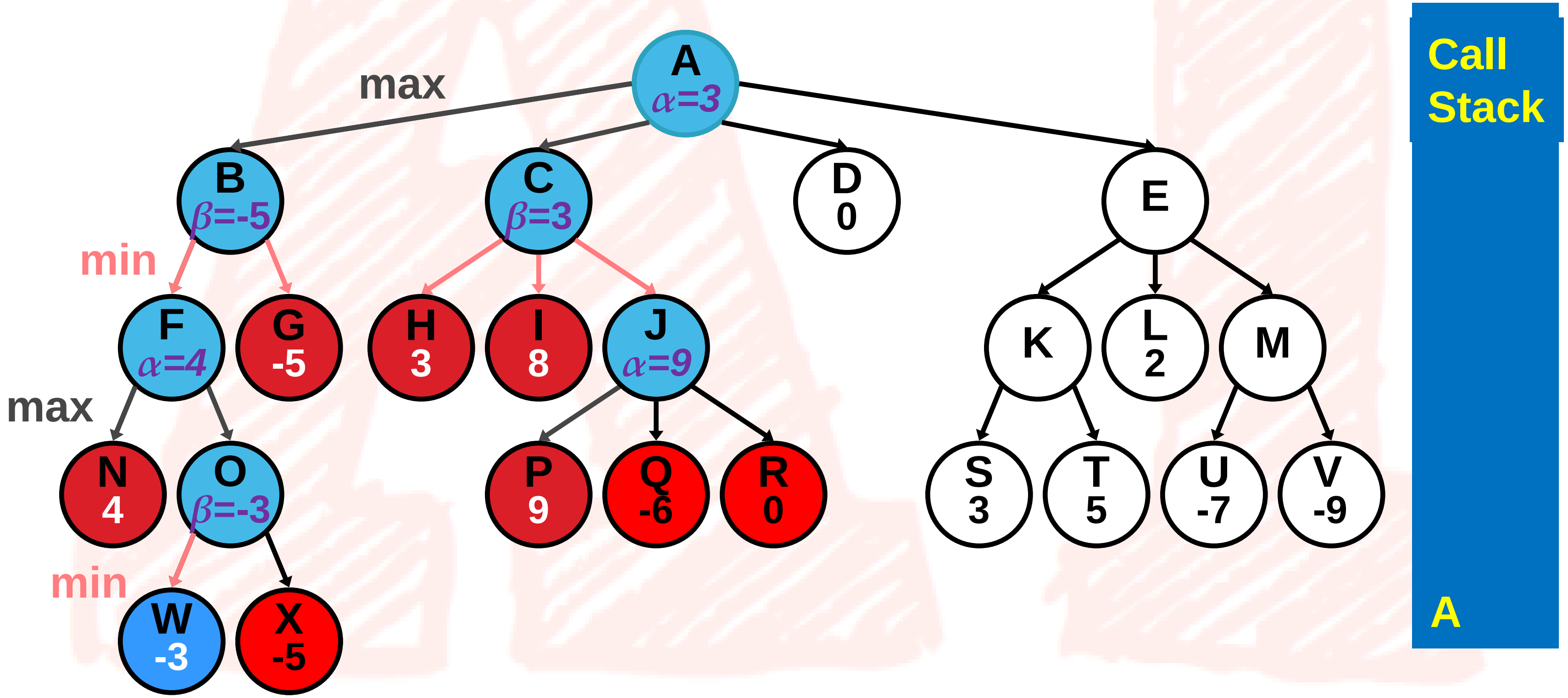
Keep expanding C? No since no more successors for C



Alpha-Beta Search Example

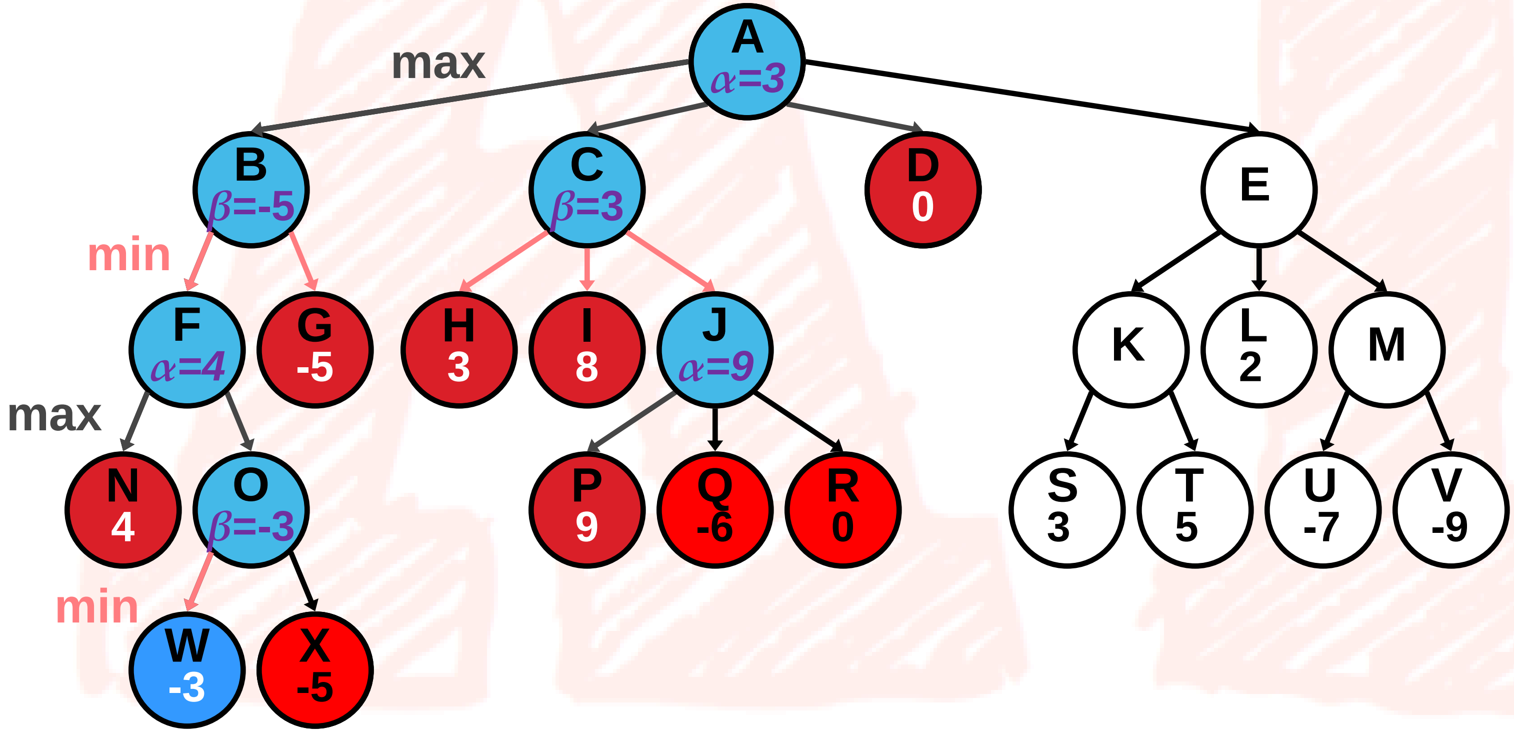
back to alpha = 3, since 3 >= -5 (maximizing)
`minimax(A,0,4)`

Keep expanding A? **Yes** since there are more successors, no cutoff test



Alpha-Beta Search Example

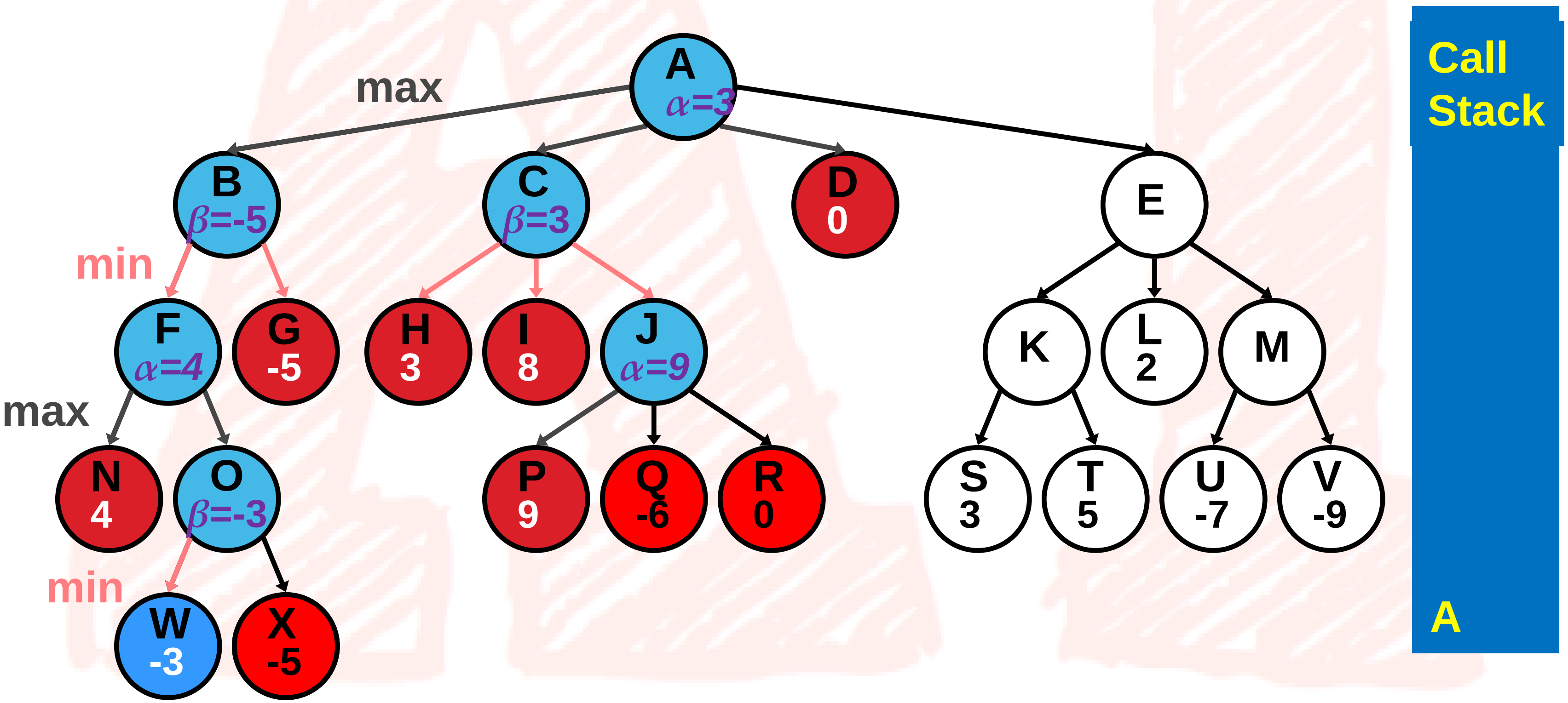
`minimax(D,1,4)` evaluate and return SBE value



Alpha-Beta Search Example

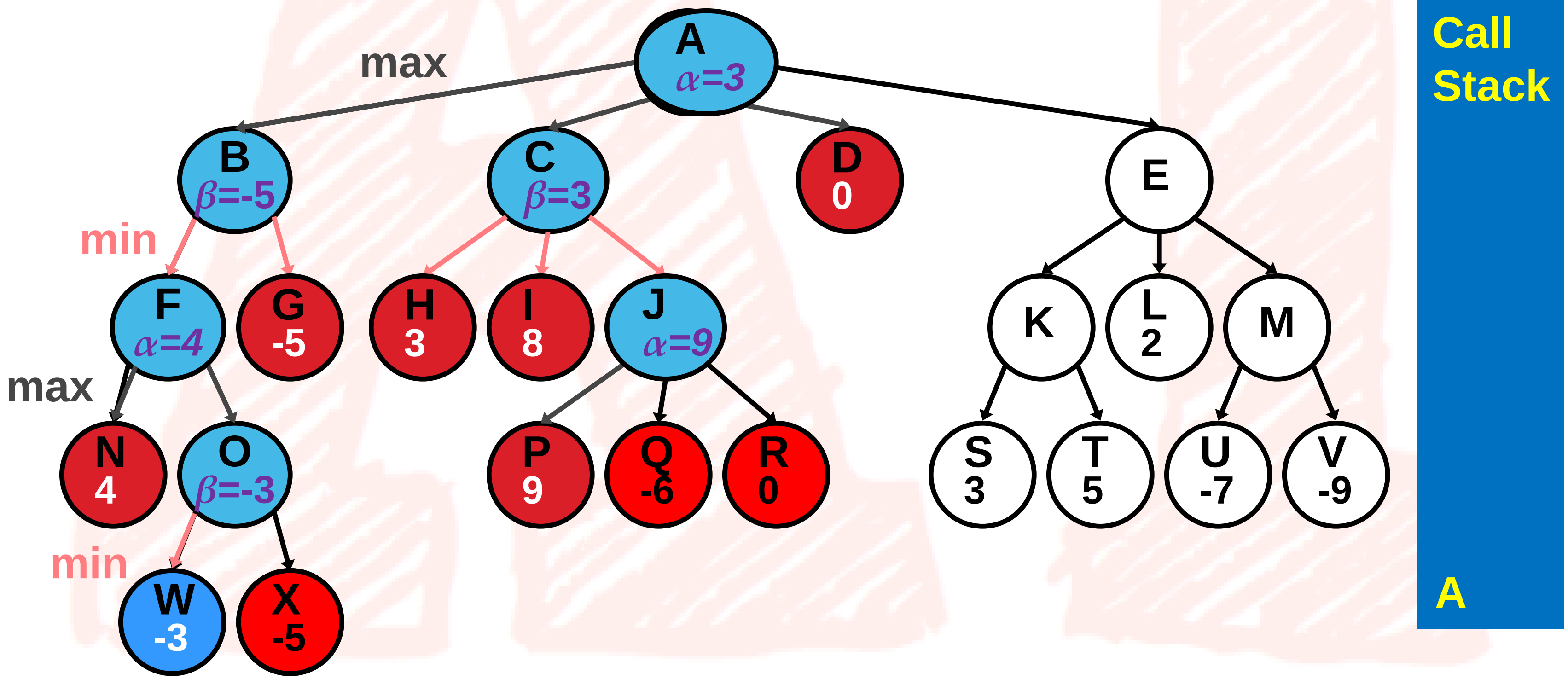
back to `minimax(A,0,4)` alpha doesn't change, since 0 < 3 (maximizing)

Keep expanding A? Yes since there are more successors, no cutoff test



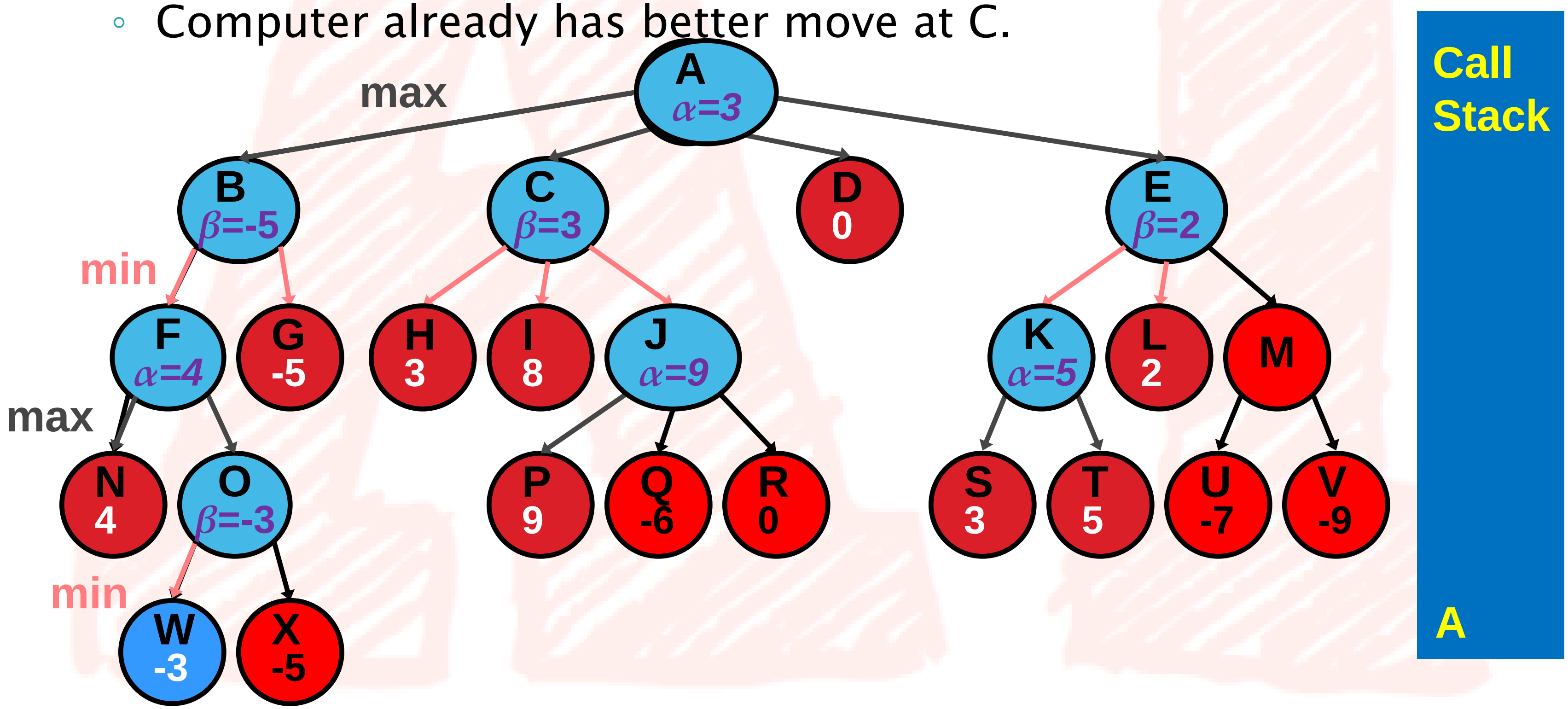
Alpha-Beta Search Example

- How does the algorithm finish searching the tree?



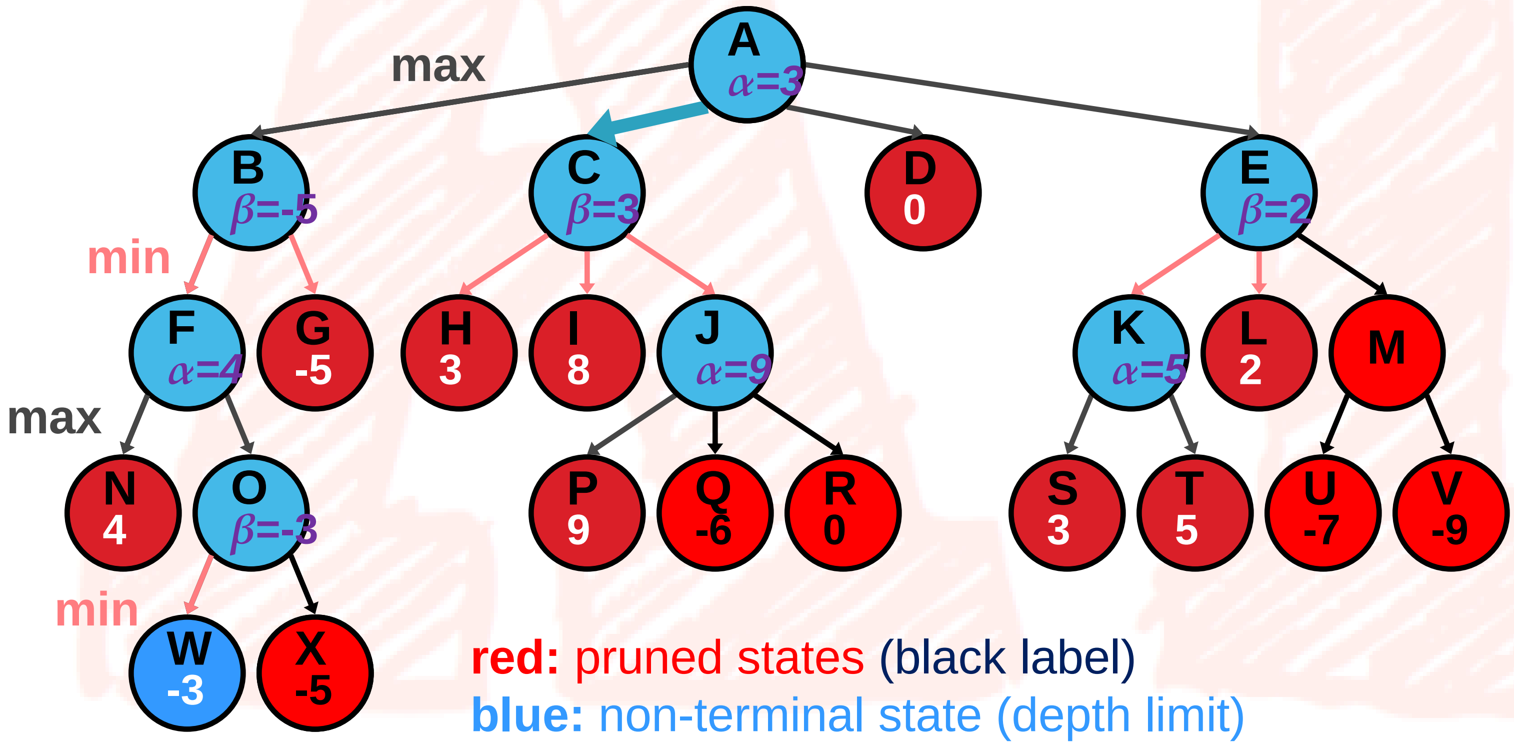
Alpha-Beta Search Example

- Stop Expanding E since A's alpha >= E's beta is true: alpha cutoff
- Why?
  - Smart opponent will choose L or worse, thus E's upper bound is 2.
  - Computer already has better move at C.



Alpha-Beta Search Example

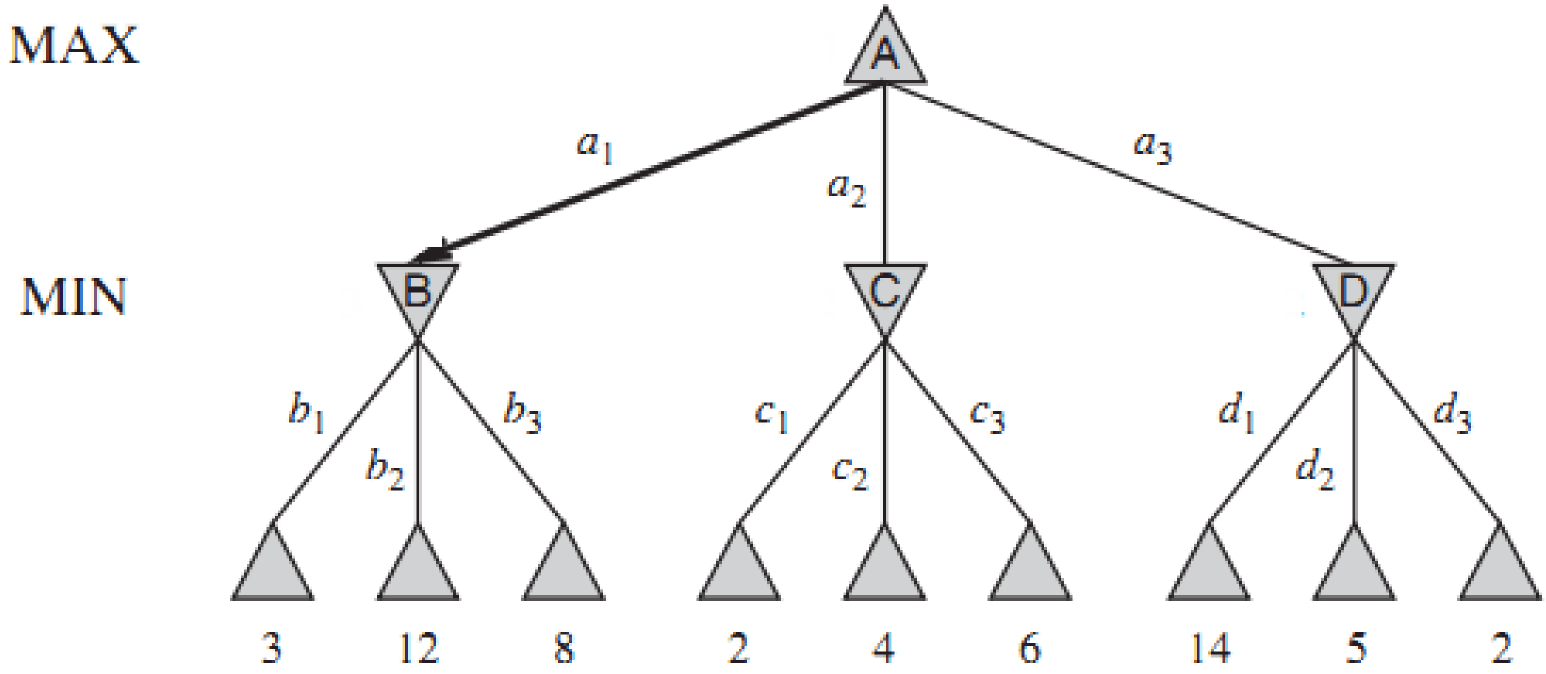
- Result: Computer chooses move to C.



red: pruned states (black label)
blue: non-terminal state (depth limit)

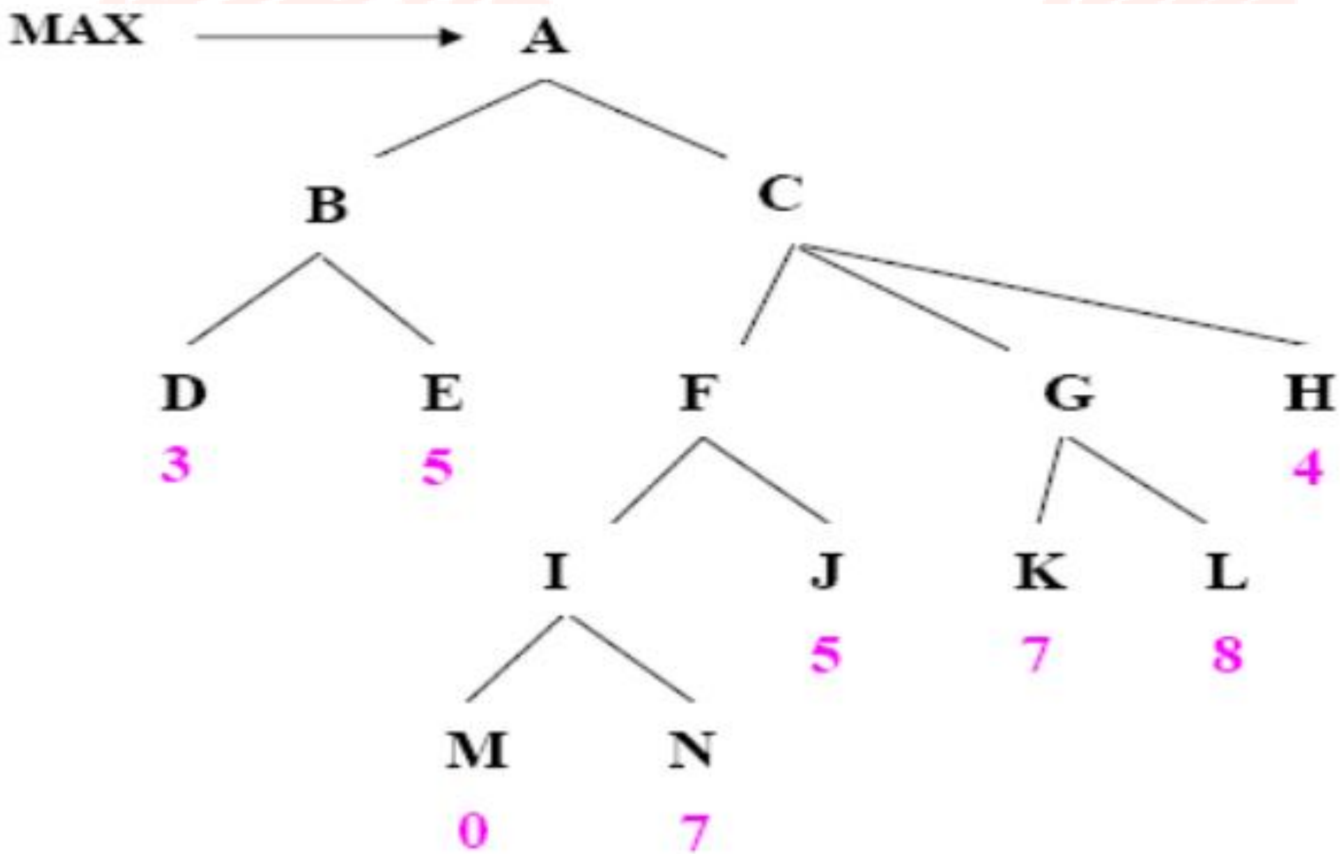
Exercise 3

- Perform the minimax algorithm on the figure below with Alpha-Beta-pruning



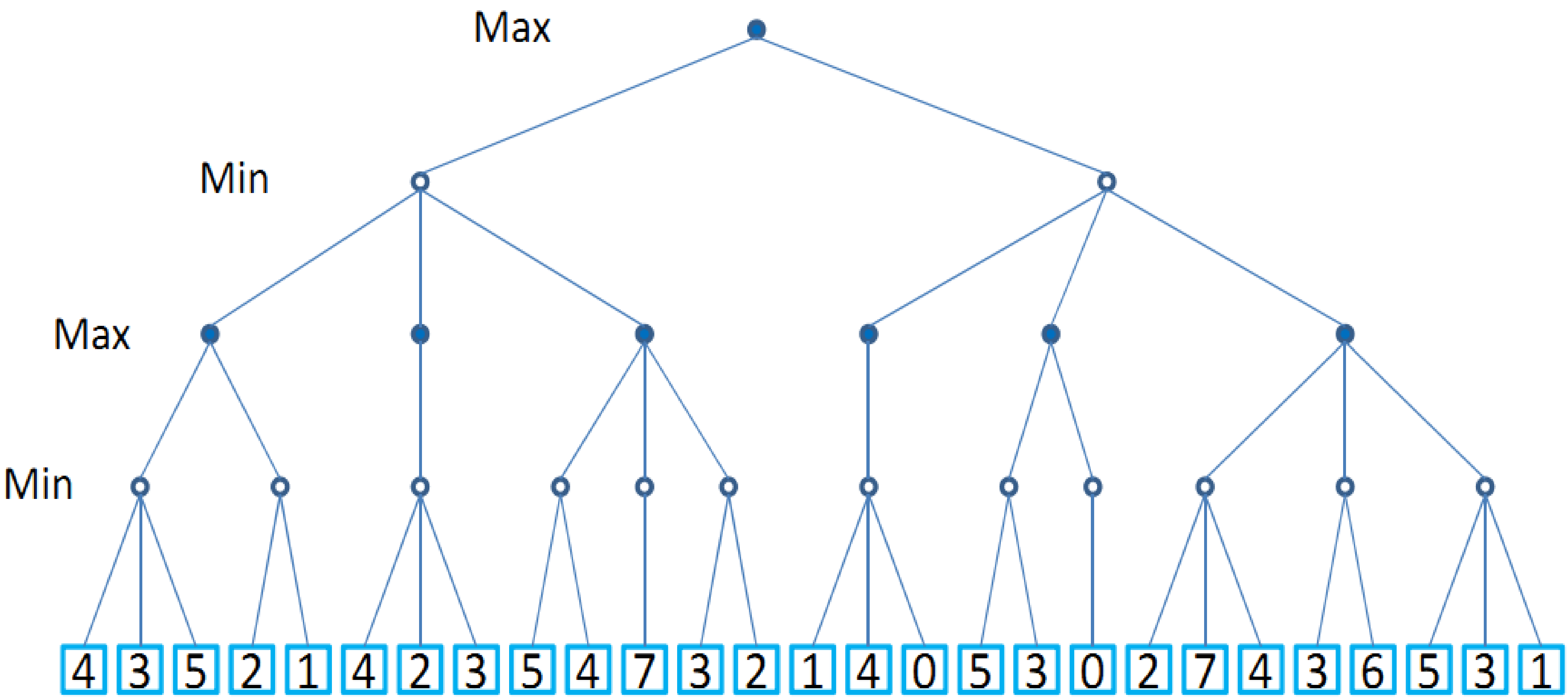
Exercise 4

- Perform the minimax algorithm on the figure below with Alpha-Beta-pruning



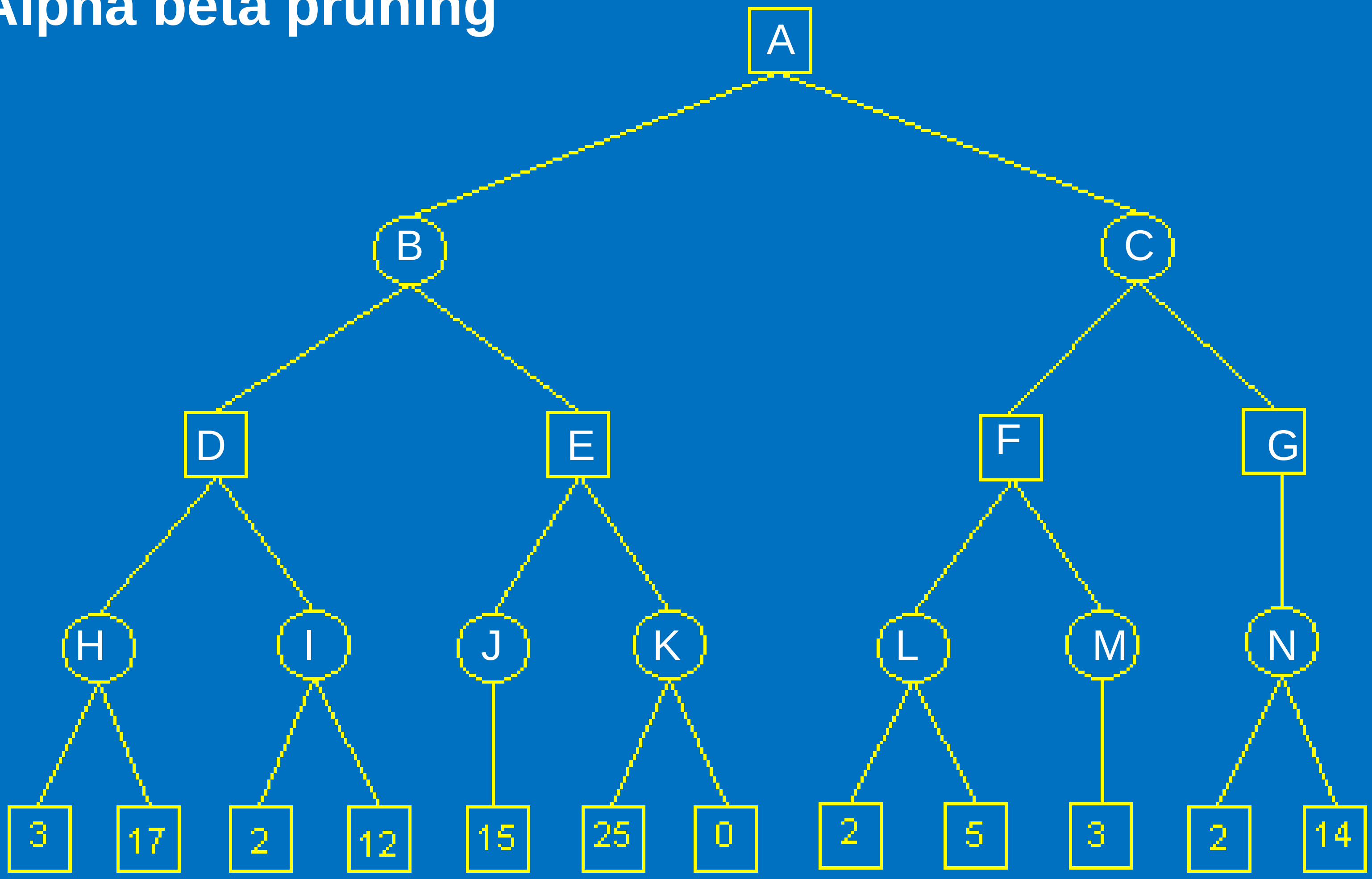
Exercise 5

- Perform the minimax algorithm on the figure below with Alpha-Beta-pruning



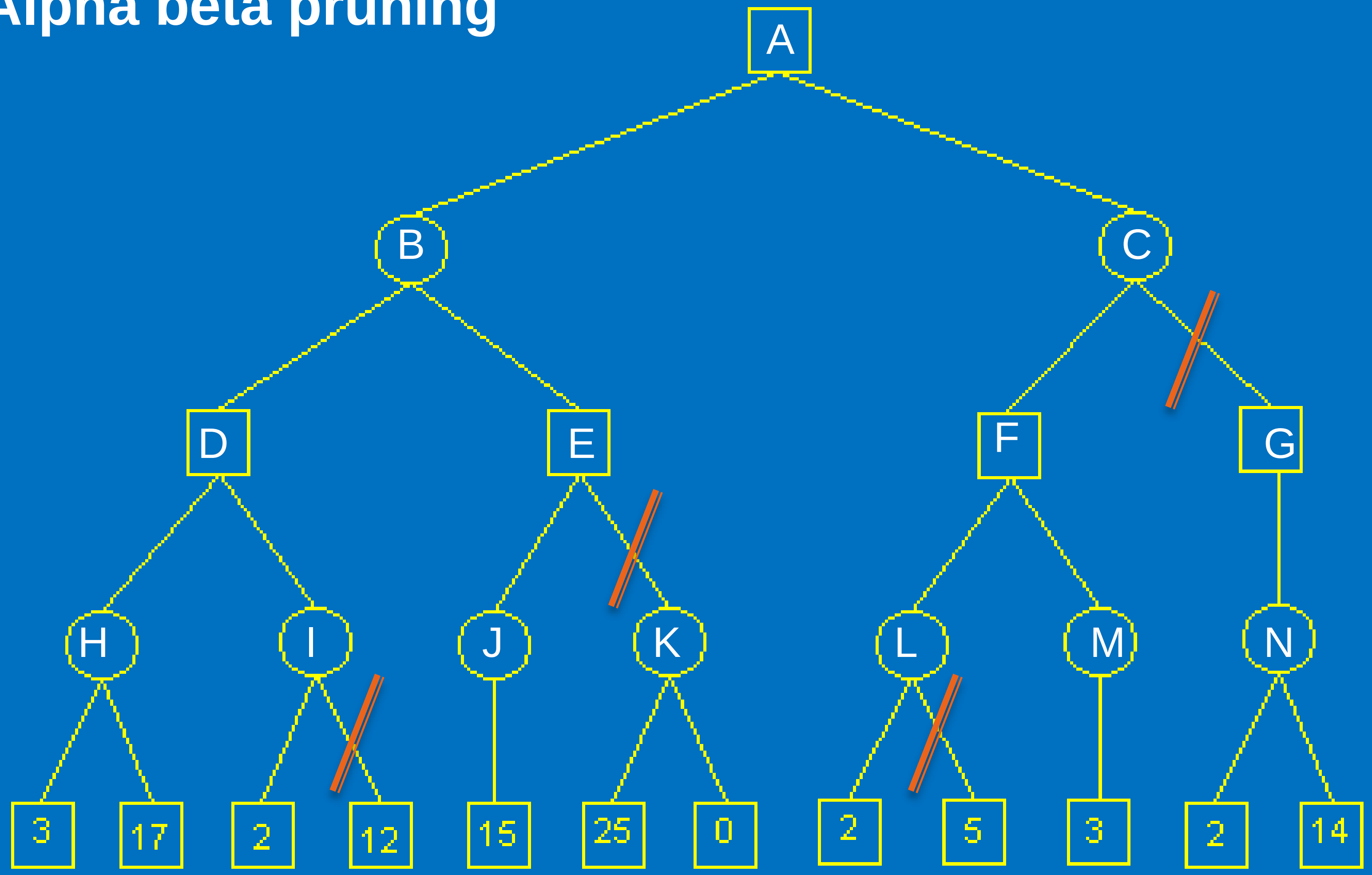
Exercise 6

- **Alpha beta pruning**



Exercise 6: Result

- **Alpha beta pruning**



Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  inputs: *state*, current state in game
  $\boxed{v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty)}$
  return the *action* in SUCCESSORS(*state*) with value *v*

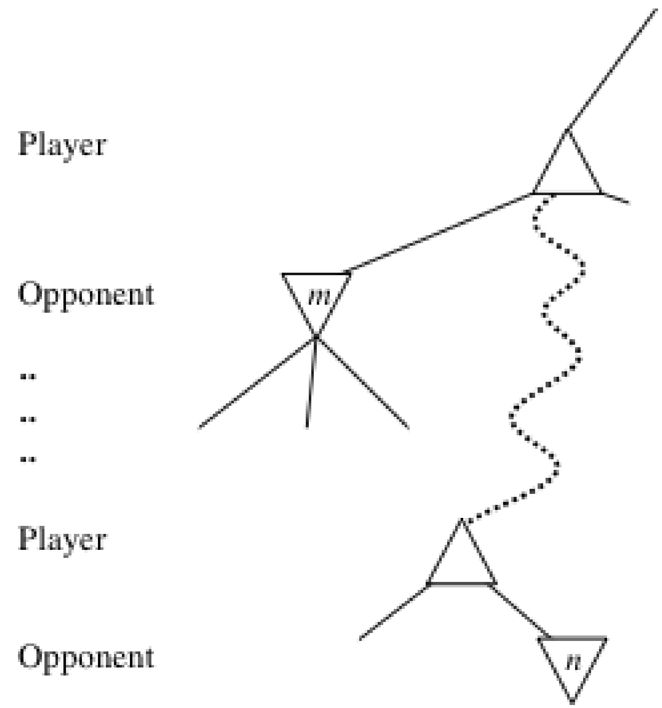
function $\boxed{\text{MAX-VALUE}(state, \alpha, \beta)}$ **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *s* **in** SUCCESSORS(*state*) **do**
    $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$
    if $v \geq \beta$ **then return** *v*
    $\alpha \leftarrow \text{MAX}(\alpha, v)$
  return *v*

Alpha-Beta Algorithm

```
function MIN-VALUE(state, α , β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← + ∞
    for each s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α , β))
        if v ≤ α then return v
        β ← MIN(β ,v)
    return v
```

General alpha-beta pruning

- Consider a node *n* somewhere in the tree
- If player has a better choice at
  - Parent node of n
  - Or any choice point further up
- *n* will never be reached in actual play.
- Hence when enough is known about *n*, it can be pruned.



Final Comments: Alpha-Beta Pruning

- Pruning does not affect final results
- Entire subtrees can be pruned.
- Good move *ordering* improves effectiveness of pruning
- With "perfect ordering" time complexity is $O(b^{m/2})$
  - Branching factor of sqrt(b) !!
  - Alpha-beta pruning can look twice as far as Minimax in the same amount of time
- Repeated states are again possible.
  - Store them in memory = transposition table

Imperfect Real-Time Decisions

- Minimax require too much leaf-node evaluations.
- May be impractical within a reasonable amount of time.
- SHANNON (1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)

Cutting off search

- Change:

```
if TERMINAL-TEST(state) then
   return UTILITY(state)
```

into

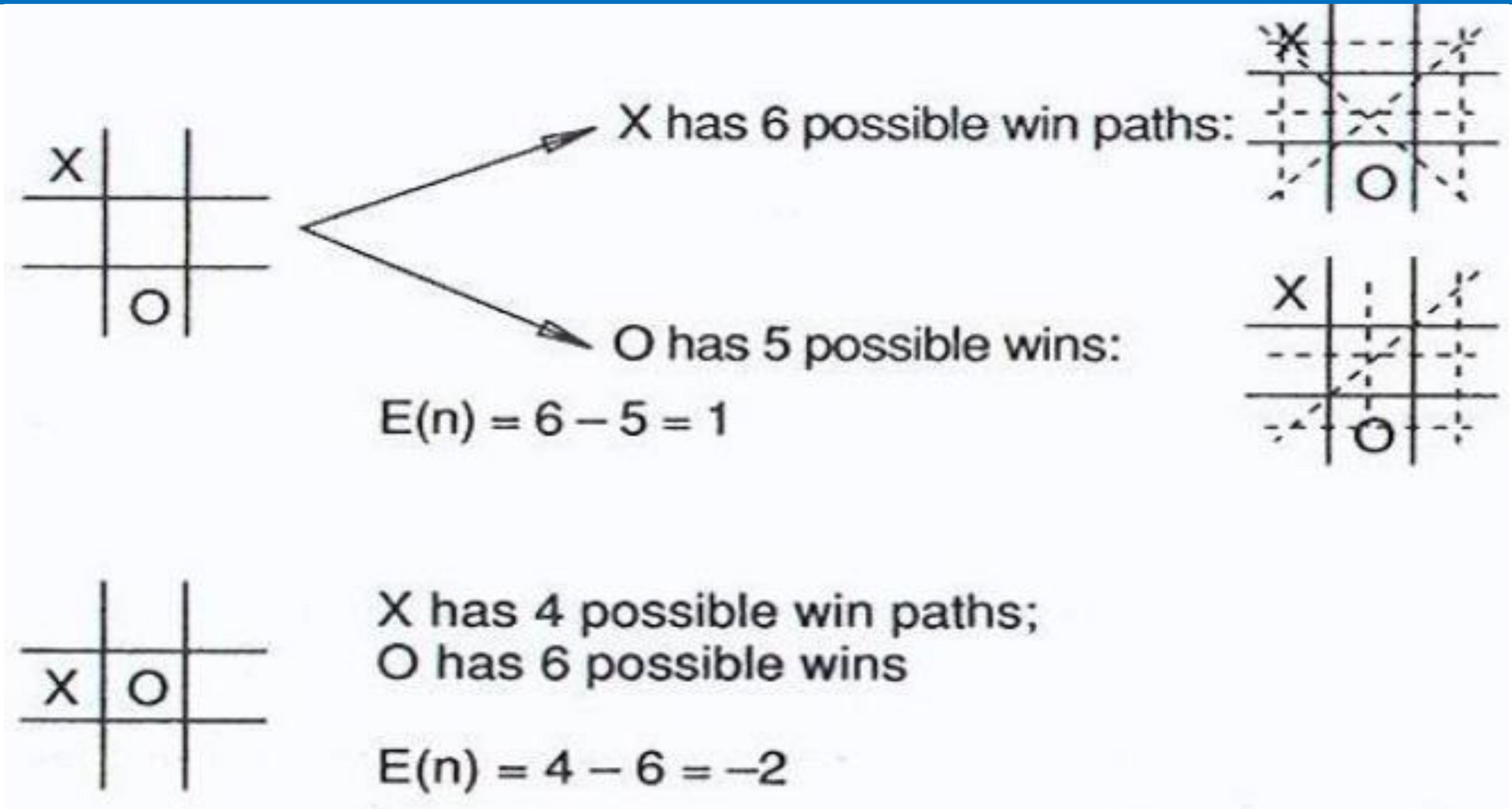
```
if CUTOFF-TEST(state, depth) then
   return EVAL(state)
```

- Introduces a fixed-depth limit depth
  - Is selected so that the amount of time will not exceed what the rules of the game allow.
- When cuttoff occurs, the evaluation is performed.

Heuristic EVAL

- EVAL function retuns an estimate of the expected utility of the game from a given position.
- Performance of game playing depends on quality of EVAL.
- Requirements:
  - EVAL must agree with terminal-nodes in the same way as UTILITY.
  - Computation may not take too long.
  - For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.
- Only useful for quiescent (no wild swings in value in near future) states

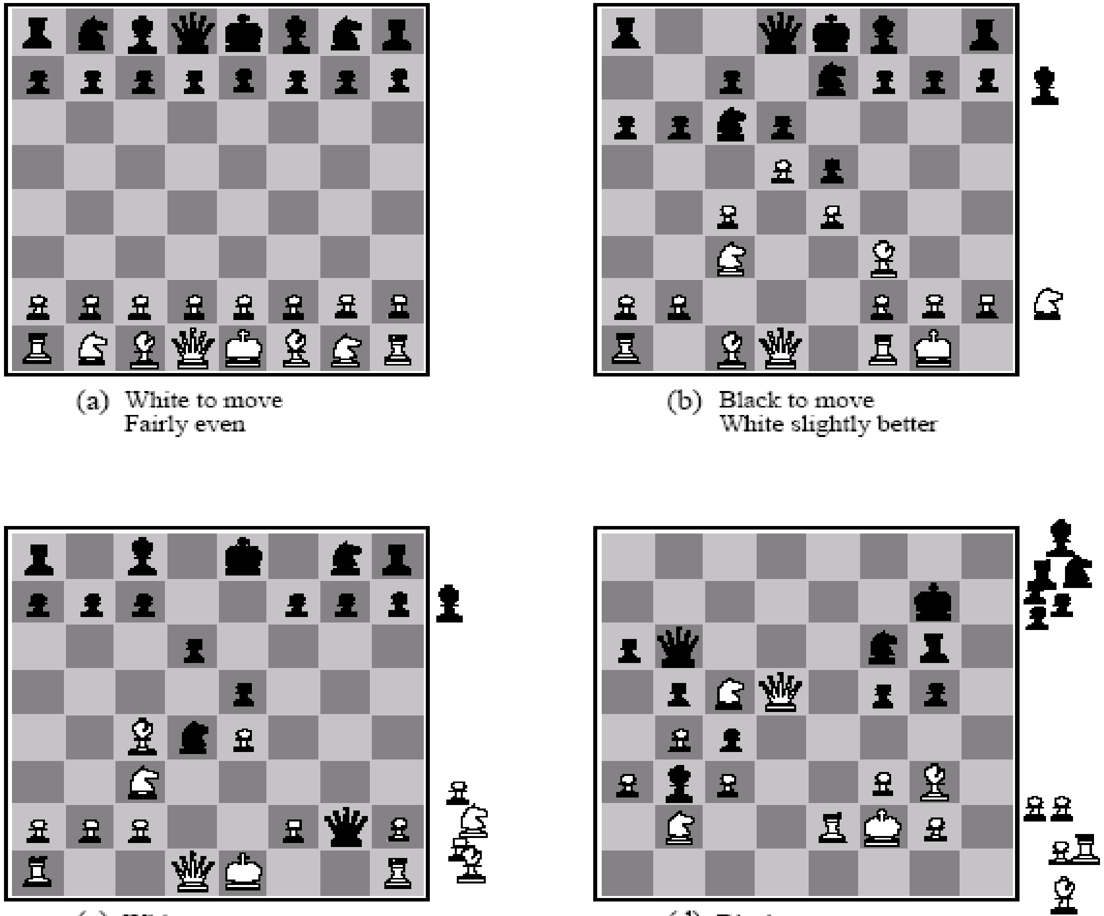
Heuristic EVAL example



- Heuristic: E(n) = M(n) – O(n)
  - M(n): total win paths of X,
  - O(n): total win paths of O

Heuristic EVAL example

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$



(a) White to move
Fairly even

(b) Black to move
White slightly better

(c) White to move
Black winning

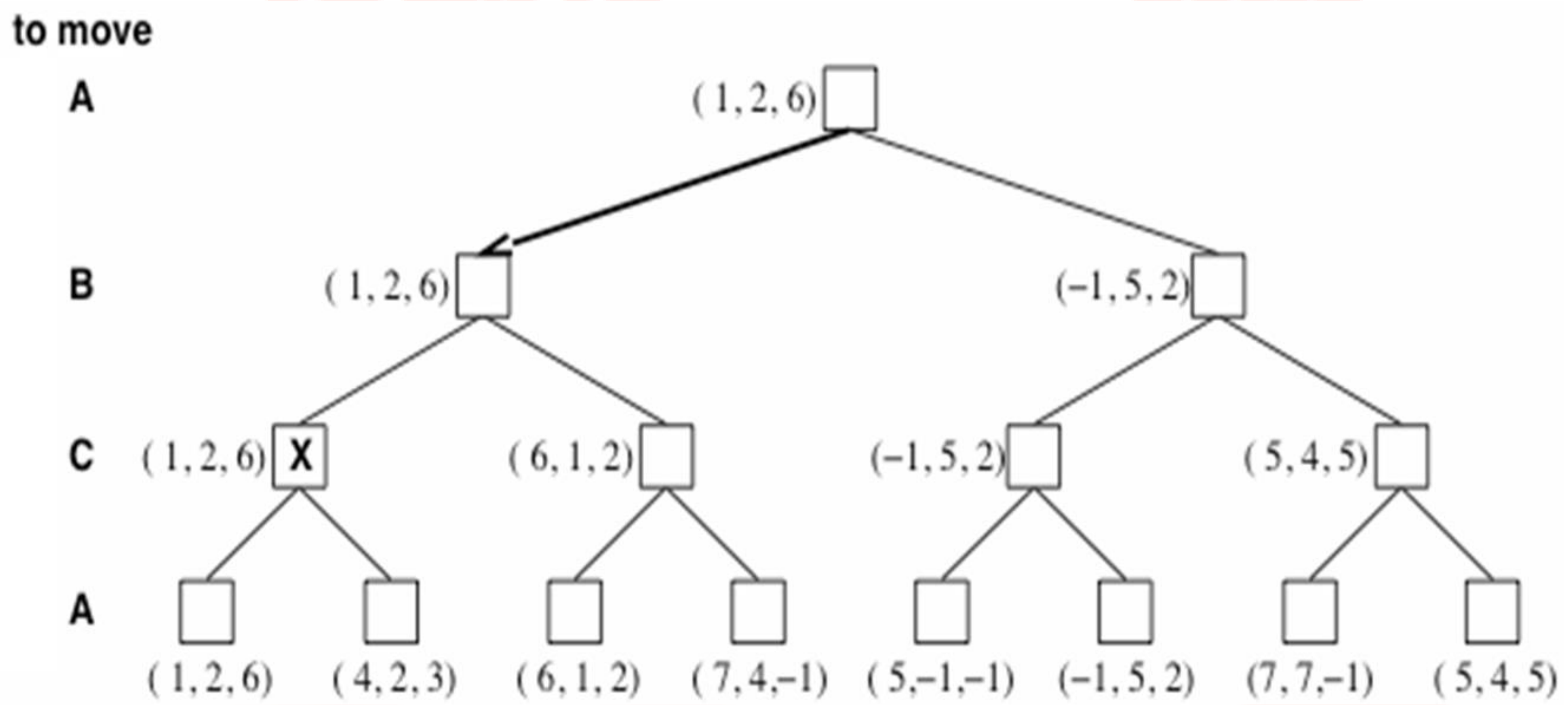
(d) Black to move
White about to lose

- w_i: a weight,
- f_i: a feature of the position

- Pawn: 1,
- Knight: 3,
- Rook: 5,
- Queen: 9

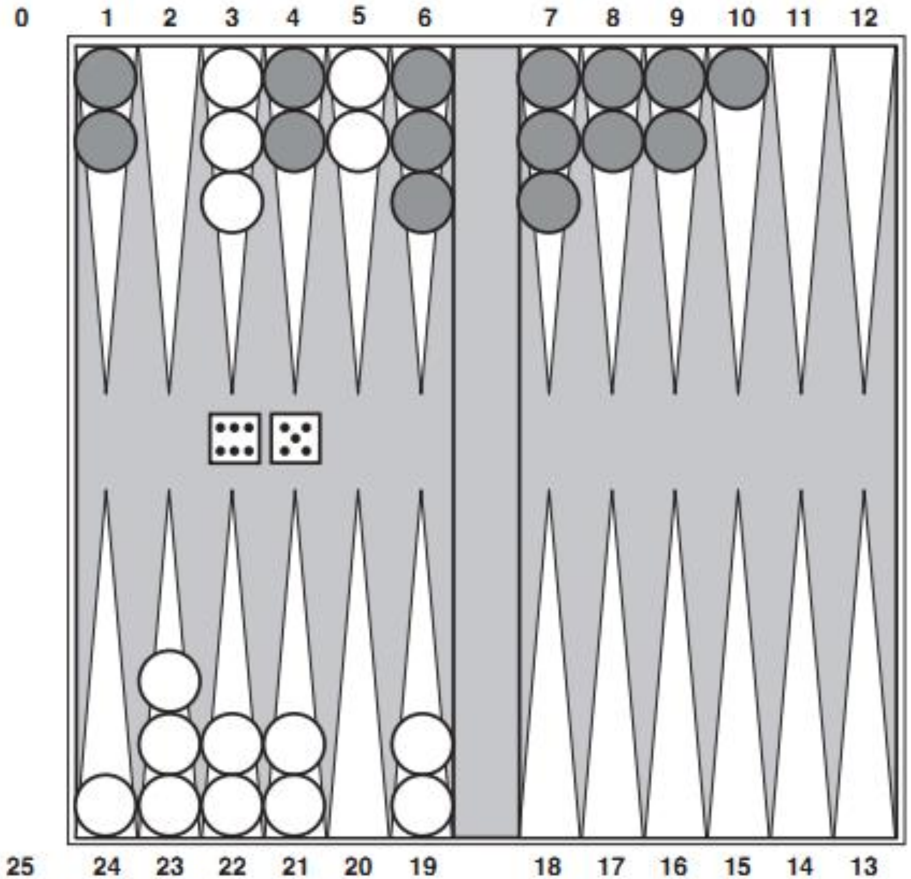
Multiplayer games

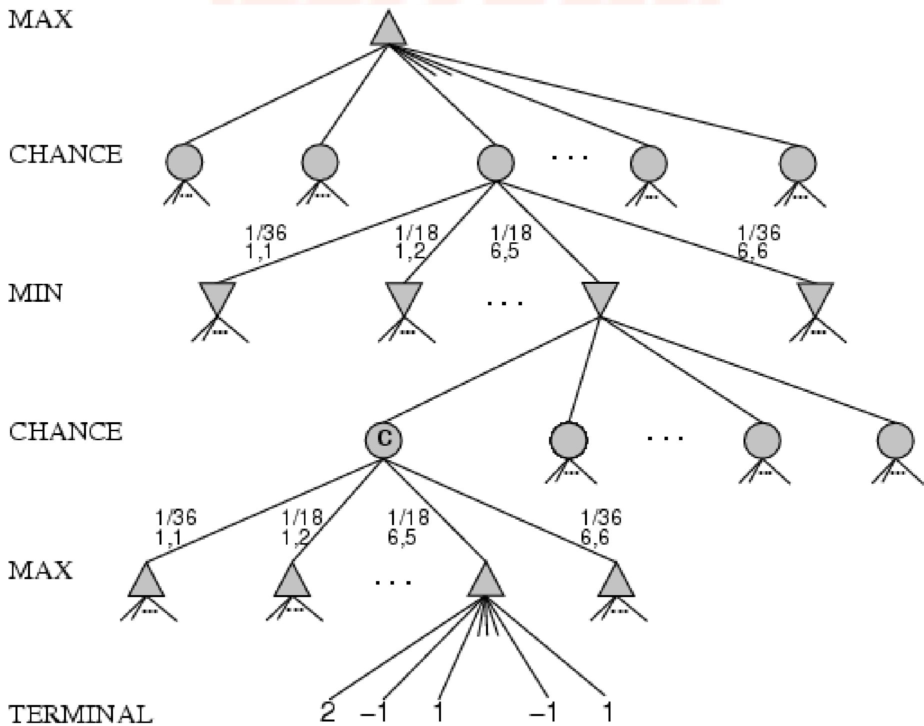
- Games allow more than two players
- Single minimax values become vectors



Games that include chance

Backgammon

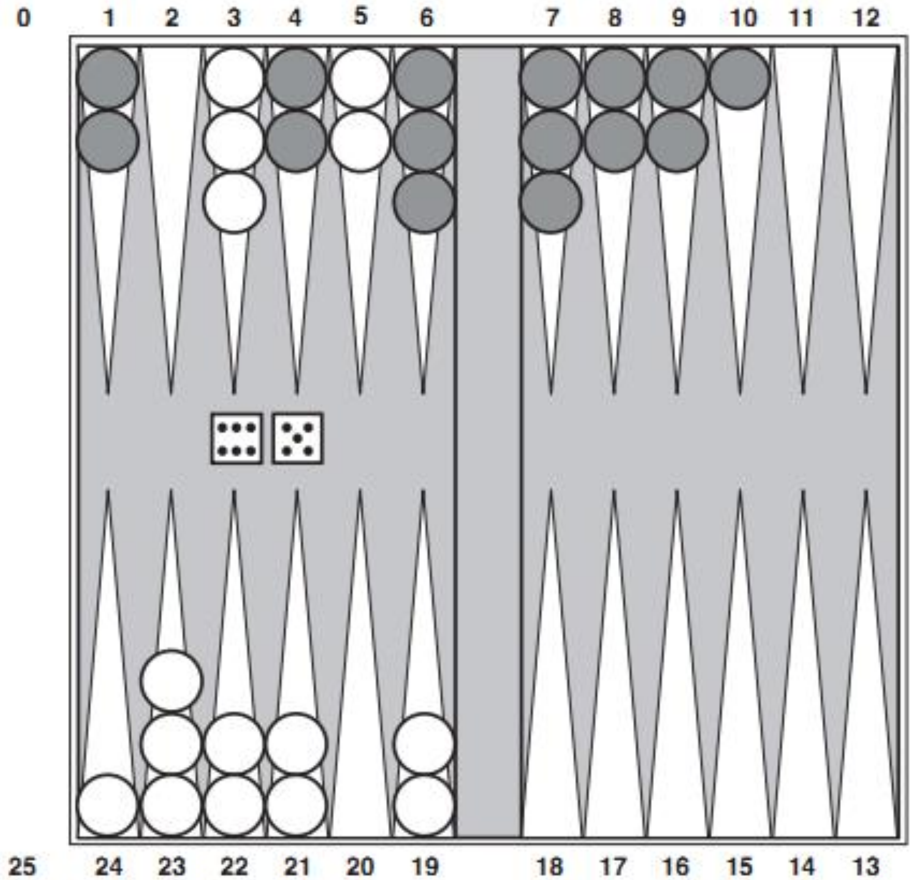


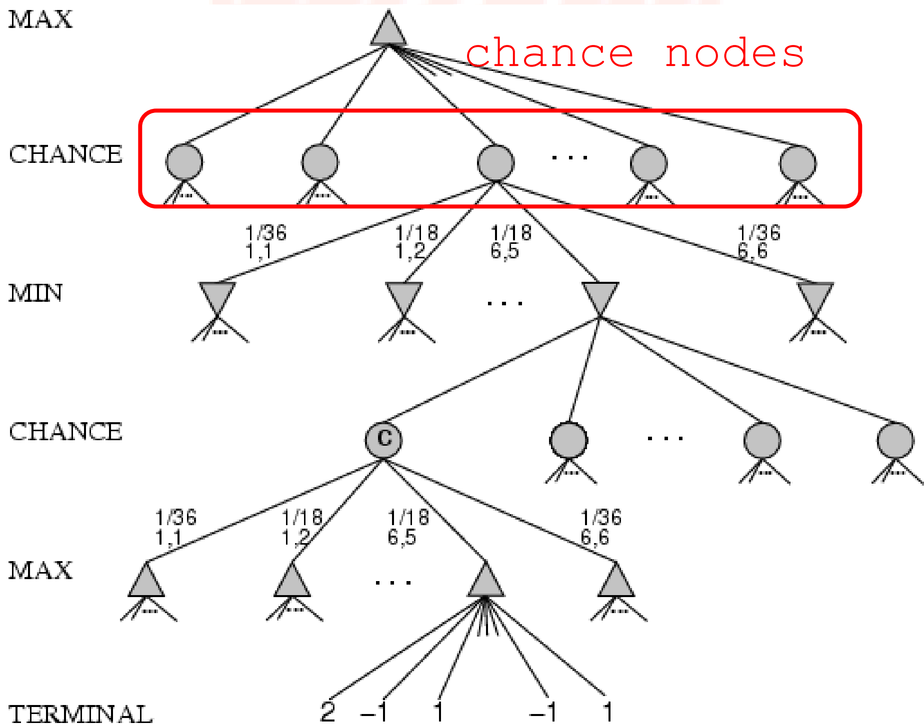


- Possible moves (5–10, 5–11), (5–11, 19–24), (5–10, 10–16) and (5–11, 11–16)

Games that include chance

Backgammon

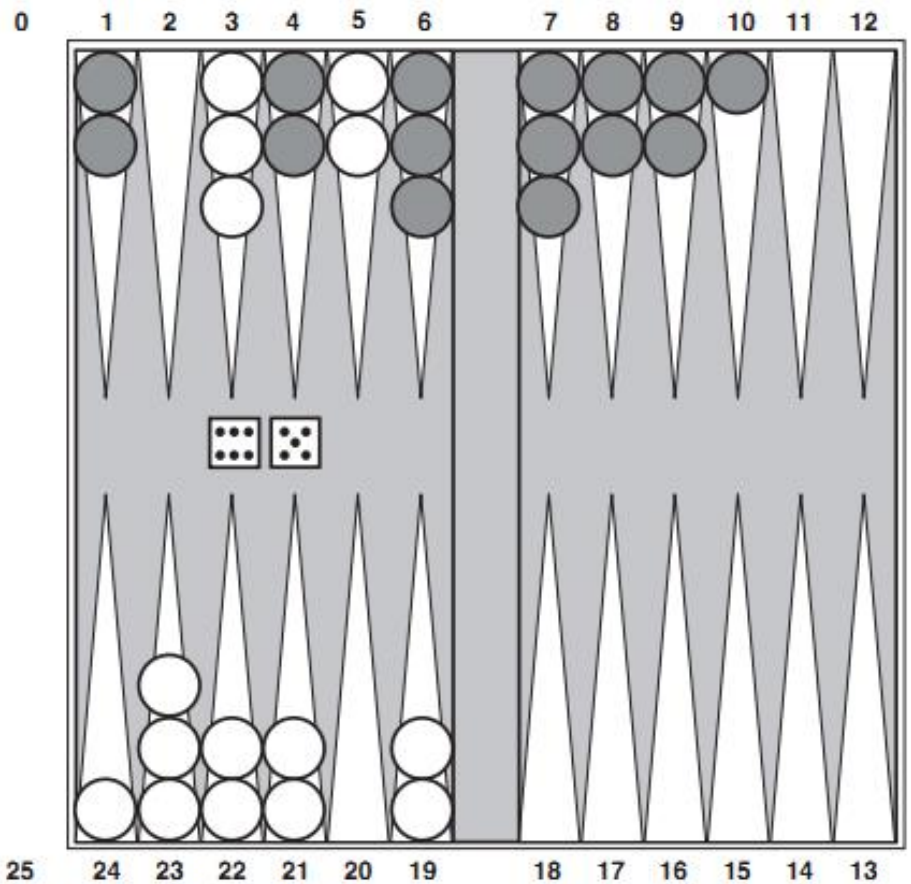


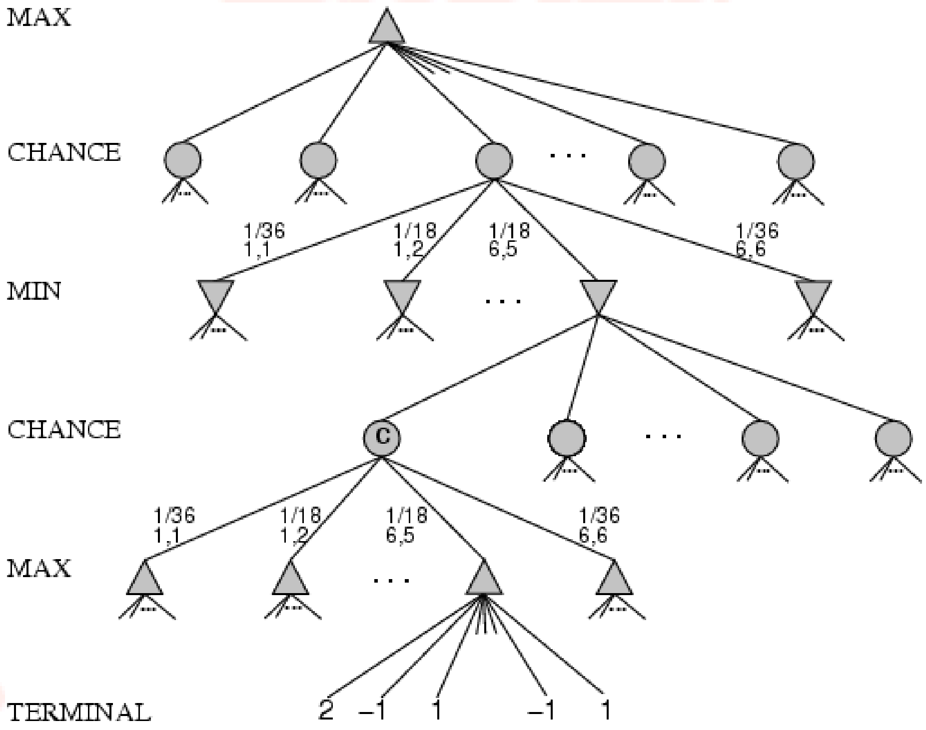


- Possible moves (5–10, 5–11), (5–11, 19–24), (5–10, 10–16) and (5–11, 11–16)
- [1,1], [6,6] chance 1/36, all other chance 1/18

Games that include chance

Backgammon





- [1,1], [6,6] chance 1/36, all other chance 1/18
- Can not calculate definite minimax value, only expected value

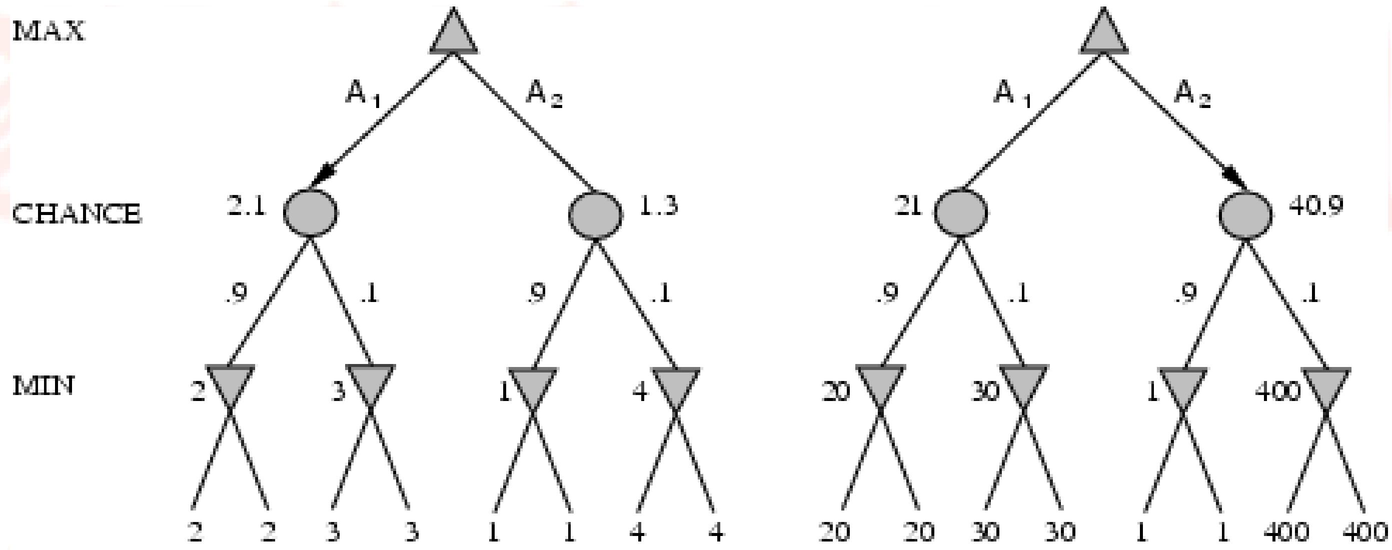
Expected minimax value

EXPECTED-MINIMAX-VALUE(n) =

$$
\begin{cases}
\text{UTILITY}(n) & \text{If n is a terminal} \\
\max_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s) & \text{If n is a max node} \\
\min_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s) & \text{If n is a min node} \\
\sum_{s \in \text{successors}(n)} P(s) \,.\, \text{EXPECTEDMINIMAX}(s) & \text{If n is a chance node}
\end{cases}
$$

- These equations can be backed-up recursively all the way to the root of the game tree.

Position evaluation with chance nodes



- Left, A1 wins
- Right, A2 wins
- Outcome of evaluation function may not change when values are scaled differently.
- Behavior is preserved only by a positive linear transformation of EVAL.

Summary

- Games are fun (and dangerous)
- They illustrate several important points about AI
  - Perfection is unattainable -> approximation
  - Good idea what to think about
  - Uncertainty constrains the assignment of values to states
- Games are to AI as grand prix racing is to automobile design.

Thank you for your attention!